Blackfish™ SQL Developer's Guide

by Steven T. Shaughnessy and Jens Ole Lauridsen

Preface

This Preface describes the manual, lists technical resources, and provides CodeGear contact information.

Intended Audience

This document is for:

- Developers implementing Blackfish SQL database applications
- System administrators responsible for installing, deploying, and maintaining Blackfish SQL databases

Blackfish SQL for Windows users should have a working knowledge of:

- Delphi, C++, C#, or VB.NET programming
- Basic dbExpress 4 or ADO.NET 2.0
- Basic SQL

Blackfish SQL for Java users should have a working knowledge of:

- Java programming
- DataExpress
- JDBC
- SQL

Document Conventions

This document uses the following typographic conventions:

Font or Symbol	Use
`mono space`	Computer output, files, directories, URL, and code elements such as command lines, keywords, function names, and code examples
`mono space italic`	Parameter names and variables
Bold	GUI elements, emphasis, section subheadings
Italic	Book titles, new terms
underlined font	Hyperlink

Online Technical Resources

General information is at: http://www.codegear.com/Blackfish SQL.

Technical information and community contributions are at: http://cdn.codegear.com/Blackfish SQL.

To discuss issues with other Blackfish SQL users, visit the newsgroups: support.codegear.com/newsgroups/directory.

CodeGear Support

CodeGear offers a variety of support options for Blackfish SQL. For pre-sales support, installation support, and a variety of technical support options, visit: http://support.codegear.com.

When you are ready to deploy Blackfish SQL, you may need additional deployment licenses. To purchase licenses and upgrades, visit the CodeGear Online Shop at: http://shop.codegear.com.

Additional Resources

Useful technical resources include:

JDBC

- JDBC™ API Documentation at java.sun.com
- *JDBC API Tutorial and Reference*, by Seth White, et al; published by Addison Wesley

SQL

- *A Guide to The SQL Standard*, by C. J. Date and Hugh Darwen; published by Addison Wesley

DataExpress JavaBeans

- *DataExpress Component Library Reference* in the Blackfish SQL for Java Help

Contents

Chapter 1: Overview

This chapter describes Blackfish SQL features.

Blackfish SQL

Blackfish™ SQL is a high-performance, small-footprint, transactional database. Blackfish SQL was originally implemented as an all-Java database called JDataStore. This is now called *Blackfish SQL for Java*. Blackfish SQL was then ported from Java to C#. The C# implementation is called *Blackfish SQL for Windows*.

The design and implementation of Blackfish SQL emphasizes database performance, scalability, ease of use, and a strong adherence to industry standards. Blackfish SQL capabilities include the following:

- Industry standards compliance:
  - Entry level SQL-92
  - Unicode storage of character data
  - Unicode-based collation key support for sorting and indexing
  - dbExpress 4 drivers for Win32 Delphi and C++
  - ADO.NET 2.0 providers for .NET
  - JDBC for Java
  - JavaBean data access components for Java
  - XA/JTA Distributed transactions for Java
- High performance and scalability for demanding online transaction processing (OLTP) and decision support system (DSS) applications
- Delphi, C#, and VB.NET stored procedures and triggers for Windows
- Java-stored procedures and triggers
- Zero-administration, single assembly or single-jar deployment
- Database incremental backup and failover in the Java version

Blackfish SQL DataStore

Blackfish SQL is the name of the product, its tools, and of the file format. Within RAD Studio, there

are assemblies that include classes that start with `DataStore`.

Compatibility Between Windows and Java

Blackfish SQL for Windows and Blackfish SQL for Java are compatible, though some restrictions apply. The database file format is binary-compatible between the two. The database clients and servers are interchangeable. Windows clients can connect to Java servers and Java clients can connect to Windows servers. However, because the Blackfish SQL for Windows implementation is more recent, some Blackfish SQL for Java features are not yet supported for the Windows version.

For additional information about Blackfish SQL compatibility, see System Architecture.

Blackfish SQL for Windows Connectivity

Blackfish SQL for Windows provides the following database drivers:

- **DBXClient:** This 100% Object Pascal dbExpress 4 database driver enables C++ and Win32 Delphi applications to connect to a remote Blackfish SQL for Windows or Blackfish SQL for Java server.
- **Local ADO.NET 2.0 Provider:** This 100% managed code driver enables .NET applications to connect to a local Blackfish SQL for Windows server. The local ADO.NET driver executes in the same process as the BlackFish SQL database kernel, for better performance.
- **Remote ADO.NET 2.0 Provider:** This 100% managed code driver enables .NET applications to acquire a remote connection to either a Blackfish SQL for Windows or Blackfish SQL for Java server.

For instructions on using these drivers, see Establishing Connections.

Blackfish SQL API for Windows

The Blackfish SQL API may be used in Delphi and C++ programs with the DBXClient DBX4 driver. . NET applications can use the API with the ADO.NET Provider. In RAD Studio the API is in the `Borland.Data.DataStore and Borland.Data.MetaData namespaces.`

The administrative capabilities listed below are not yet supported in DataExplorer for Blackfish SQL for Windows. Use SQL commands or Blackfish SQL built-in DB_ADMIN stored procedures to complete these tasks.

- Create/alter autoincrement and max inline properties for columns
- Create secondary indexes
- Create, alter, drop users and roles
- Create, alter, drop database mirrors
- Database encryption

- Database backup

Administration and Utility Functions Available From SQL

Two classes are available, DB_ADMIN and DB_UTIL. These methods can be called from SQL using the CALL statement. They can be called without creating a METHOD alias, because the Blackfish SQL dialect recognizes methods in DB_ADMIN as built-in methods.

DB_ADMIN Class

DB_ADMIN is a group of stored procedures for performing a variety of database administration tasks. Some of the capabilities include:

- Configuring automatic failover and incremental backup
- Viewing and altering database configurations
- Backing up databases (explicit)
- Encrypting databases
- Mirror management

For more information, see the Stored Procedures Reference.

DB_UTIL Class

DB_UTIL is a set of SQL stored procedures for performing numeric, string and date/time operations on data stored in database tables. These procedures include such functions as:

- Mathematical functions, such as trigonometric, arithmetic, and random
- String manipulation functions
- Date and time functions

For more information, see the Stored Procedures Reference.

ADO.NET

Blackfish SQL includes an ADO.NET implementation. This is similar to the AdoDbx Client, which is also an ADO.NET implementation.

- DataStoreCommand: Provides execution of SQL statements and execution of stored procedures.
- DataStoreCommandBuilder: Generates single-table commands to reconcile changes made to a DataSet

with its underlying database.

- DataStoreConnection: Provides a connection to a database.
- DataStoreConnectionPool: Provides access to a connection pool.
- DataStoreDataAdapter: Fills a DataSet and updates a database.
- DataStoreDataReader: Gives access to a result table from a database server.
- DataStoreDataSourceEnumerator: Provides enumerator for finding all data sources on local network.
- DataStoreParameter: Specifies parameter for DataStoreCommand.
- DataStoreParameterCollection: Collection of parameters for DataStoreCommand.
- DataStoreProviderFactory: Base class for a provider's implementation of data source classes.
- DataStoreRowUpdatedEventArgs: Holds data for RowUpdated event of DataStoreDataAdapter.
- DataStoreRowUpdatingEventArgs: Holds data for RowUpdating event of DataStoreDataAdapter.
- DataStoreTransaction: Provides a transaction.

Chapter 2:
System Architecture

This chapter provides an overview of Blackfish SQL system architecture.

Blackfish SQL Compatibility

Blackfish SQL for Windows and Blackfish SQL for Java are compatible in these ways:

- The database file format is binary-compatible between the two
- The database clients and servers are interchangeable
- Windows clients can connect to Java servers and Java clients can connect to Windows servers

Compatibility is restricted in the following ways:

- The `Object` type uses platform-specific serialization; therefore the data cannot be shared between two different clients:
  - An ADO driver cannot read a Java serialized object.
  - A Java driver cannot read a .NET serialized object.
  - A DbxClient driver cannot read Java or .NET serialized objects.
- The maximum scale for a decimal is different in Java and .NET.
- For Blackfish SQL for Java, the `Timestamp` type has two more digits in the fractional portion.

The following Blackfish SQL for Java features are not yet supported in the Windows version:

- ISQL SQL Command Line Interpreter
- High Availability features, including incremental backup and failover
- Graphical tooling for some administrative capabilities
- Access to file and object streams
- Tracking and resolving of row-level insert, update and delete operations
- Access to the Blackfish SQL File System directory

Blackfish SQL for Windows Connectivity

Windows applications can use one or more of the following connectivity solutions to access a Blackfish SQL database programmatically:

- **DBXClient**

  DBXClient is a 100% Object Pascal dbExpress 4 database driver that enables Win32 Delphi and C++ applications to connect to a Blackfish SQL for Windows or Blackfish SQL for Java server.

- **ADO.NET**

  ADO.NET is the Microsoft standard for database connectivity on the .NET platform. Blackfish SQL for Windows has the following ADO.NET providers:

  - **Local ADO.NET 2.0 Provider:** This 100% managed code driver enables .NET applications to connect to a local Blackfish SQL server. The local ADO.NET driver executes in the same process as the BlackFish SQL database kernel, for better performance.
  - **Remote ADO.NET 2.0 Provider:** This 100% managed code driver enables .NET applications to acquire a remote connection to either a Blackfish SQL for Windows or Blackfish SQL for Java server.

See Establishing Connections for instructions and code examples for using these drivers.

Blackfish SQL for Java Connectivity

Java applications can use one or more of the following connectivity solutions to access a Blackfish SQL database programmatically:

- **JDBC Type 4 Drivers**

  JDBC is the industry standard SQL call-level interface for Java applications. Blackfish SQL for Java provides the following JDBC drivers:

  - **Local JDBC driver:** This 100% managed code driver enables Java applications to connect to a local Blackfish SQL server. The local JDBC driver executes in the same process as the BlackFish SQL database kernel, for better performance.
  - **Remote JDBC driver:** This 100% managed code driver enables Java applications to acquire a remote connection to either a Blackfish SQL for Windows or Blackfish SQL for Java server.

- **ODBC to JDBC Gateway**

  Provided by EasySoft Limited, this gateway is an industry standard SQL call-level interface. The EasySoft ODBC to JDBC Gateway enables native applications to access Blackfish SQL databases.

- **DataExpress JavaBeans**

  DataExpress JavaBeans provides additional functionality not addressed by the JDBC standard. See DataExpress JavaBeans for details.

See Establishing Connections for instructions and code examples for using these drivers.

DataExpress JavaBeans

NOTE: This feature is available only with Blackfish SQL for Java.

DataExpress is a set of JavaBean runtime components that provide functionality not addressed by the JDBC standard. JavaBean is an industry-standard component architecture for Java. The JavaBean standard specifies many important aspects of components needed for RAD development environments. JavaBean components can be designed in a visual designer and can be customized with the properties, methods, and events that they expose.

DataExpress is included in the component palette of CodeGear JBuilder Visual Designer. For information on using DataExpress from within JBuilder, see the JBuilder Help.

Because DataExpress is a set of runtime components, you need not use JBuilder to develop and deploy applications that use DataExpress.

The majority of DataExpress JavaBean components are those required to build both server-side and client-side database applications. Client-side applications require high quality data binding to visual components such as grid controls, as well as support for reading and writing data to a database.

Server-side applications require data access components to help with reading and writing data to a database, but presentation is typically handled by a web page generation system such as Java Server Pages (JSPs). Even though DataExpress has extensive support for client-side data binding to visual component libraries such as dbSwing and JBCL, the DataExpress design still separates the presentation from the data access layer. This allows DataExpress components to be used as a data access layer for other presentation paradigms such as the JSP/servlet approach employed by JBuilder InternetBeans Express technology.

The DataExpress architecture allows for a pluggable storage interface to cache the data that is read from a data source. Currently, there are only two implementations of this interface, `MemoryStore` (the default), and `DataStore`. By setting just two properties on a `StorageDataSet` JavaBean component, a Blackfish SQL table can be directly navigated and edited with a `StorageDataSet` JavaBean. By setting the `DataSet` property of a dbSwing grid control, the entire contents of large tables can be directly browsed, searched, and edited at high speed. This effectively provides an ISAM-level data access layer for Blackfish SQL tables.

Automating Administrative Functions with DataExpress JavaBeans

There are many DataExpress components that can be used to automate administrative tasks. Commonly-used components are:

DataExpress Administrative Components

Task	Component
Custom server start and shutdown	`com.borland.datastore.jdbc.DataStoreServer`
Database backup, restore, and pack	`com.borland.datastore.DataStoreConnection.copyStreams()`

Security administration	`com.borland.datastore.DataStoreConnection`
Transaction management	`com.borland.datastore.TxManager` `com.borland.datastore.DataStore`

DataExpress JavaBean Source Code

JBuilder provides a source code `.jar` file that includes a large portion of the DataExpress JavaBean components. This enables you to more easily debug your applications and gain a better understanding of the DataExpress JavaBean components.

Differences Between Local and Remote Drivers

The primary difference between using local and remote drivers with Blackfish SQL is:

- **Local driver:** The Blackfish SQL database engine executes in the same process as the application.
- **Remote driver:** The Blackfish SQL database engine executes in either the same process or in a different process as the application.

Advantages of Using a Local Driver to Access a Database

A local Blackfish SQL driver provides the following benefits:

- **High-speed interface to the database**

  Driver calls are made directly into the database kernel on the same call stack. There are no remote procedure calls to a database server running in another process.
- **Easy to embed in an application**

  The database server does not need to be configured or started. The executable code for the database kernel, database driver and application execute in the same process.

Advantages of Using a Remote Driver to Access a Database

You can use a remote Blackfish SQL driver to execute Blackfish SQL in a separate database server process. However, before the application can use a remote driver, the Blackfish SQL server process must be started. Executing the Blackfish SQL database kernel in a separate database server process provides the following benefits:

- **Multi process access to a database**

  If multiple processes on one or more computers need to access a single Blackfish SQL database, a Blackfish SQL server must be started and the remote drivers must be used by the application.
- **Improved performance using multiple computers**

If your application or web server is consuming a large portion of the memory or CPU resources, it is often possible to achieve better performance by running the Blackfish SQL server on a separate computer.

- **Improved fault tolerance**

  Applications that use a remote connection typically run in a separate process. Errant applications can be terminated without having to shutdown the database server.

Advantages of Using Both Local and Remote Drivers to Access a Database

Using both the local and remote driver to access the same database can give you the best of both worlds. A Blackfish SQL database file can be held open by only one operating system process. When you connect using the local driver, the process that uses the local driver holds the database file open. When the remote driver makes the connection, the Blackfish SQL server process holds the database file open.

Since the local driver causes the database file to be open in the same process, it prevents connections from the remote driver. However, if the process that uses the local driver also starts a Blackfish SQL server in the same process, then other processes using the remote driver can access the same database as the local driver.

The Blackfish SQL server can be started inside an application by using a single line of Java code that instantiates a `DataStoreServer` component and executes its `start` method. The `DataStoreServer` runs on a separate thread and services connection requests from processes that use the remote driver to access a Blackfish SQL database on the computer that on which the `DataStoreServer` was started.

In addition, the local driver can be used by the application that launched the `DataStoreServer` for faster in-process driver calls into the Blackfish SQL database engine.

Blackfish SQL Database Files

These files are created and used by Blackfish SQL:

- ***file-name*.jds:** a single file storage for all database objects.
- ***database-name*_LOGA_*:** transactional log files. If the database file is moved, the log files must be moved with it.
- ***database-name*_LOGA_ANCHOR:** redundantly stores log file configuration information.
- ***database-name*_STATUS*:** log files created if status logging is enabled for the database.

A Blackfish SQL database can still be used if the anchor or status log files do not exist.

A non-transactional (read only) database only needs the .jds database file.

The specifications for Blackfish SQL database file capacity are:

Blackfish SQL Database File Capacity

Specification	Value
Minimum block size:	1 KB
Maximum block size:	32 KB
Default block size	4 KB
Maximum Blackfish SQL database file size	2 billion blocks. For the default block size, that yields a maximum of 8,796,093,022,208 bytes (8TB).
Maximum number of rows per table stream	281,474,976,710,656
Maximum row length	1/3 of the block size. Long Strings, objects, and input streams that are stored as Blobs instead of occupying space in the row.
Maximum Blob size	2GB each
Maximum file stream size	2GB each

Blackfish SQL Database File System

A Blackfish SQL database file can contain these types of data streams:

- **Table streams:** These are database tables typically created using SQL. A table stream can have secondary indexes and Blob storage associated with it.
- **File streams:** There are two types of file streams:
  - Arbitrary files created with `DataStoreConnection.createFileStream()`
  - Serialized Java objects stored as file streams

A single Blackfish SQL database can contain all stream types.

Streams are organized in a file system directory. The ability to store both tables and arbitrary files in the same file system allows all of the data for an application to be contained in a single portable, transactional file system. A Blackfish SQL database can also be encrypted and password protected.

The specifications for Blackfish SQL database file systems are:

Blackfish SQL Database File System Specifications

Specification	Value
Directory separator character for streams	/

Maximum stream name length	192 bytes • Best case (all single-byte character sets): 192 characters • Worst case (all double-byte character sets): 95 characters (one byte lost to indicate DBCS)
Reserved names	Stream names that begin with SYS are reserved. Blackfish SQL has the following system tables: • SYS/Connections • SYS/Queries • SYS/Users

Blackfish SQL for Java Specific Streams

Some table streams and all file streams are currently only accessible from Java applications.

If the `resolvable` property for the table is set, all insert, update, and delete operations made against the table are recorded. This edit tracking feature enables DataExpress components to synchronize changes from a replicated table to the database from which the table was replicated.

File streams are random-access files. File streams can be further broken down into two different categories:

- **Arbitrary files created with `DataStoreConnection.createFileStream()`:** You can write to, seek in, and read from these streams.
- **Serialized Java objects stored as file streams:** You can write to, seek in, and read from these streams.

Each stream is identified by a case-sensitive name referred to as a `storeName` in the API. The name can be up to 192 bytes long. The name is stored along with other information about the stream in the internal directory of the Blackfish SQL database. The forward slash (`/`) is used as a directory separator in the name to provide a hierarchical directory organization. JdsExplorer uses this structure to display the contents of the directory in a tree.

Advantages of Using the Blackfish SQL File System

For the simple persistent storage of arbitrary files and objects, using the Blackfish SQL file system has a number of advantages over using the JDK classes in the `java.io` package:

- It is simpler, because one class is needed instead of four (`FileOutputStream`, `ObjectOutputStream`, `FileInputStream`, `ObjectInputStream`).
- You can keep all your application files and objects in a single file and access them easily with a logical name instead of streaming all of your objects to the same file.
- Your application can use less storage space, due to how disk clusters are allocated by some

operating systems. The default block size in a Blackfish SQL database file is small (4KB).

- Your application is more portable, since you are no longer at the mercy of the host file system. For example, different operating systems have different allowable characters for names. Some systems are case sensitive, while others are not. Naming rules inside the Blackfish SQL file system are consistent on all platforms.
- Blackfish SQL provides a transactional file system that can also be encrypted and password protected.

Blackfish SQL Directory Contents

Note: Currently, the directory for the Blackfish SQL database can be accessed only from Java applications. Fortunately, most applications do not need to access the directory directly.

The JdsExplorer tree provides a hierarchical view of the the Blackfish SQL directory. The Blackfish SQL directory can also be opened programmatically with a DataExpress `StorageDataSet` component. This provides a tabular view of all streams stored in the Blackfish SQL file system. The directory table has the following structure:

Blackfish SQL Directory Table Columns

#	Name	Constant	Type	Contents
1	`State`	`DIR_STATE`	short	Whether the stream is active or deleted
2	`DeleteTime`	`DIR_DEL_TIME`	long	If deleted, when; otherwise zero
3	`StoreName`	`DIR_STORE_NAME`	String	The store name
4	`Type`	`DIR_TYPE`	short	Bit fields that indicate the type of streams
5	`Id`	`DIR_ID`	int	A unique ID number
6	`Properties`	`DIR_PROPERTIES`	String	Properties and events for a DataSet stream
7	`ModTime`	`DIR_MOD_TIME`	long	Last time the stream was modified
8	`Length`	`DIR_LENGTH`	long	Length of the stream, in bytes
9	`BlobLength`	`DIR_BLOB_LENGTH`	long	Length of the Blobs in a table stream, in bytes

You can reference the columns by name or by number. There are constants defined as `DataStore` class variables for each of the column names. The best way to reference these columns is to use these constants. The constants provide compile-time checking to ensure that you are referencing a valid column. Constants with names ending with the suffix `_STATE` exist for the different values for the `State` column. There are also constants for the different values and bit masks for the `Type` column, with names ending with the suffix `_STREAM`. See the online help for the `DataStore` class for a listing of these constants.

Stream Details

Time columns in the Blackfish SQL directory are Coordinated Universal Time (UTC).

As with many file systems, when you delete a stream in Blackfish SQL, the space it occupied is marked

as available, but the contents and the directory entry that points to it are not immediately reused for new allocations. This means you can sometimes restore a deleted stream if it has not been overwritten.

For more information on deleting and restoring streams, see Deleting Streams, How Blackfish SQL Reuses Blocks, and Restoring Streams.

The `Type` column indicates whether a stream is a file or table stream, but there are also many internal table stream subtypes (for example, for indices and aggregates). These internal streams are marked with the `HIDDEN_STREAM` bit to indicate that they should not be displayed. Of course, when you are reading the directory, you can decide whether these streams should be hidden or visible.

These internal streams have the same `StoreName` as the table stream with which they are associated. This means that the `StoreName` alone does not always uniquely identify a stream. Some internal stream types can have multiple instances. The ID for each stream is always unique; however, the `StoreName` is sufficiently unique for the `storeName` parameter used at the API level. For example, when you delete a table stream, all the streams with that `StoreName` are deleted.

Directory Sort Order

The directory table is sorted by the first five columns. Because of the values stored in the `State` column, all active streams are listed first in alphabetical order by name. These are then followed by all deleted streams ordered by their delete time, oldest to most recent.

NOTE: You cannot use a `DataSetView` to create a different sort order.

Blackfish SQL File System Storage Allocation

Database contents are stored in a single file. If the database has transaction support enabled, there are additional files for transactional logs.

A database file has a block size property that defaults to 4096 bytes. The database block size property is the unit size used for new allocations in the database. This size also determines the maximum storage size of a Blackfish SQL database. The formula for computing the maximum database file size is:

<*bytes-per-block*> * 2^31

For a block size of 4096 bytes, this is about 8.8 terabytes.

A Blackfish SQL database file does not automatically shrink as data is deleted or removed from it. However, new allocations reuse the space from deleted allocations. Deleted space in the file system is made available to new allocations in two ways:

- **Deleted blocks**

  In this case, an entire block is reallocated from the list of deleted blocks.
- **Blocks that are partially full**

In this case, free space can be reused only on a per-stream basis. Specifically, the free space in a block in *Table A* can be reused only by a new allocation for a row in *Table A*. From an allocation perspective, tables, secondary indices, Blobs, and files are all separate streams.

On average, partially allocated blocks are kept at least 50 percent full. The file system goes to great lengths to ensure this is true for all stream types in the Blackfish SQL file system. The one exception to this rule occurs when a stream has a small number of blocks allocated.

A Blackfish SQL database file can be compacted to remove all deleted space and to defragment the file system so that blocks for each stream are located in contiguous regions. To compact a database using JdsExplorer, choose *Tools* > *Pack*. You can accomplish this programmatically by using the `DB_ADMIN.COPY_USERS` and `DB_ADMIN.COPY_STREAMS` methods.

Deleting Streams

Deleting a stream does not actually overwrite or clear the stream contents. As in most file systems, the space used by the stream is marked as available, and the directory entry that points to that space is marked as deleted. The time at which the stream was deleted is recorded in the directory. Over time, new stream allocations overwrite the space that was formerly occupied by the deleted stream, making the content of the deleted streams unrecoverable.

You can use JdsExplorer to delete streams, or you can delete streams programmatically using the `DataStoreConnection.deleteStream()` method, which takes as an argument the name of the stream to delete.

How Blackfish SQL Reuses Blocks

Blocks in the Blackfish SQL database file that were formerly occupied by deleted streams are reclaimed according to the following rules:

- Blackfish SQL always reclaims deleted space before allocating new disk space for its blocks.
- If the database is transactional, the transaction that deleted the stream must commit before the used space can be reclaimed.
- The oldest deleted streams (those with the earliest delete times) are reclaimed first.
- For table streams, the support streams (those for indices and aggregates) are reclaimed first.
- Space is reclaimed from the beginning of the stream to the end of the stream. This means you are more likely to recover the end of a file or table than the beginning.
- Because of the way table data is stored in blocks, you never lose or recover a partial row in a table stream, only complete rows.
- When all the space for a stream has been reclaimed, the directory entry for the stream is automatically erased, since there is nothing left to restore.

Restoring Streams

Blackfish SQL allows deleted streams to be restored if their space has not be consumed by new allocations as described in above. You can restore a stream either by using JdsExplorer to restore it, or by calling the `DataStoreConnection.undeleteStream()` method.

Because table streams have multiple streams with the same name, the stream name alone is not sufficient

for attempting to restore a stream programmatically. You must use a row from the Blackfish SQL directory. The row contains enough information to uniquely identify a particular stream.

The `DataStoreConnection.undeleteStream()` method takes such a row as a parameter. You can pass the directory dataset itself. The current row in the directory dataset is used as the row to restore.

If you create a new stream with the name of a deleted stream, you cannot restore that stream while the same name is being used by an active stream.

Transaction Management

The lifecycle of a transaction begins with any read or write operation through a connection. Blackfish SQL uses stream locks to control access to resources. To read a stream or modify any part of a stream (e. g., a byte in a file, a row in a table), a connection must acquire a lock on that stream. Once a connection acquires a lock on a stream, it holds the lock until the transaction is committed or rolled back.

In single-connection applications, transactions primarily provide crash recovery and allows an application to undo changes. Or, you may decide to make a Blackfish SQL database transactional so that it can be accessed through JDBC. If you want to access that Blackfish SQL database using DataExpress, you must deal with transactions.

Transaction Isolation Levels

Blackfish SQL supports all four isolation levels specified by the ANSI/ISO SQL (SQL/92) standards.

The serializable isolation level provides complete transaction isolation. An application can employ a weaker isolation level to improve performance or to avoid lock manager deadlocks. Weaker isolation levels are susceptible to one or more of the following isolation violations:

- **Dirty reads**

  One connection is allowed to read uncommitted data written by another connection.
- **Nonrepeatable reads**

  A connection reads a committed row, another connection changes and commits that row, and the first connection rereads that row, getting a different value the second time.
- **Phantom reads**

  A connection reads all of the rows that satisfy a `WHERE` condition, a second connection adds another row that also satisfies that condition, and the first connection sees the new row that was not there before, when it reads a second time.

SQL-92 defines four levels of isolation in terms of the behavior that a transaction running at a particular isolation level is permitted to experience, which are:

SQL Isolation Level Definitions

Isolation Level	Dirty Read	Nonrepeatable Read	Phantom Read
Read uncommitted	Possible	Possible	Possible
Read committed	Not Possible	Possible	Possible
Repeatable read	Not possible	Not possible	Possible
Serializable	Not possible	Not possible	Not possible

Choosing an Isolation Level for a Blackfish SQL Connection

Guidelines for choosing an isolation level for a connection include:

Isolation Level Guidelines

Isolation level	Description
`Read Uncommitted`	This isolation level is suitable for single-user applications for reports that allow transactionally inconsistent views of the data. It is especially useful when browsing Blackfish SQL tables with dbSwing and DataExpress DataSet components. This isolation level incurs minimal locking overhead.
`Read Committed`	This isolation level is commonly used for high-performance applications. It is ideal for data access models that use Optimistic Concurrency Control. In these data access models, read operations are generally performed first. In some cases, read operations are actually performed in a separate transaction than write operations.
`Repeatable Read`	This isolation level provides more protection for transactionally consistent data access without the reduced concurrency of `TRANSACTION_SERIALIZABLE`. However, this isolation level results in increased locking overhead because row locks must be acquired and held for the duration of the transaction.
`Serializable`	This isolation level provides complete serializability of transactions at the risk of reduced concurrency and increased potential for deadlocks. Although row locks can still be used for common operations with this isolation level, some operations cause the Blackfish SQL lock manager to escalate to using table locks.

Blackfish SQL Locking

The locks used by the Blackfish SQL Lock Manager are:

Blackfish SQL Locking

Lock	Description
Critical section locks	These are internal locks used to protect internal data structures. Critical section locks are usually held for a short period of time. They are acquired and released independent of when the transaction is committed.
Row locks	Row locks are used to lock a row in a table. These locks support shared and exclusive lock modes. Row locks are released when the transaction commits.

Table locks	Table locks are used to lock an entire table. These locks support shared and exclusive lock modes. Table locks are released when the transaction commits.
DDL table locks	DDL table locks are locks acquired when database metadata is created, altered, or dropped. These support shared and exclusive lock modes: • **Shared DDL locks** are held by transactions that have tables opened. Shared DDL locks are held until the transaction commits and the connection closes the table and all statements that refer to the table. • **Exclusive DDL locks** are used when a table must be dropped or structurally modified and are released when a transaction commits.

Controlling Blackfish SQL Locking Behavior

You can specify case-sensitive connection properties to control locking behavior. The property names are:

Case-Sensitive Connection Properties for Controlling Locking Behavior

Property	Behavior
`tableLockTables`	Specifies the tables for which row locking is to be disabled. This can be a list of tables, defined as a string of semicolon-separated, case-sensitive table names. Set this property to "*".
`maxRowLocks`	Specifies the maximum number of row locks per table that a transaction should acquire before escalating to a table lock. The default value is 50.
`lockWaitTime`	Specifies the maximum number of milliseconds to wait for a lock to be released by another transaction. When this timeout period expires, an appropriate exception is thrown.
`readOnlyTxDelay`	Specifies the maximum number of milliseconds to wait before starting a new read-only view of the database. For details, see the discussion on *read only transaction*, in Tuning Blackfish SQL Concurrency Control Performance

Blackfish SQL Locking and Isolation Levels

The use of table locks and row locks varies between the different isolation levels.
The `tableLockTables` connection property disables row locking and affects all isolation levels. Critical section and DDL locks are applied in the same manner for all isolation levels.

All isolation levels acquire at least an exclusive row lock for row update, delete, and insert operations.
In some lock escalation scenarios, an exclusive table lock occurs instead.

The row locking behavior of the Blackfish SQL connection isolation levels are:

Lock Use and Isolation Levels

Connection isolation level	Row locking behavior

`Read Uncommitted`	This level does not acquire row locks for read operations. It also ignores exclusive row locks held by other connections that have inserted or updated a row.
`Read Committed`	This level does not acquire row locks for read operations. A transaction using this isolation level blocks when reading a row that has an exclusive lock held by another transaction for an insert or update operation. This block terminates when one of the following occurs: the write transaction commits, a deadlock is detected, or the `lockTimeOut` time limit has expired.
`Repeatable Read`	This level acquires shared row locks for read operations.
`Serializable`	This level acquires shared row locks for queries that select a row based on a unique set of column values such as a primary key or `INTERNALROW` column. In SQL, the `WHERE` clause determines whether or not unique column values are being selected. Exclusive row locks are also used for insert operations and update/delete operations on rows that are identified by a unique set of column values. The following operations escalate to a shared table lock: • Read operations that are not selected based on a unique set of column values • Read operations that fail to find any rows • Insert and update operations performed on a non-uniquely specified row

Note that although lock escalation from row locks to table locks occurs in some situations for `Serializable` as described above, it also occurs for all isolation levels if the `maxRowLocks` property is exceeded.

Concurrency Control Changes

Blackfish SQL database files created with earlier versions of Blackfish SQL continue to use table locking for concurrency control. There are, however, some minor concurrency control improvements for older database files. These include:

- Support for `Read Uncommitted` and `Serializable` connection isolation levels
- Shared table locks for read operations; earlier versions of Blackfish SQL software used exclusive table locks for read and write operations.

Blackfish SQL for Windows Connection Pooling

In an application that opens and closes many database connections, it is efficient to keep unused `Connection` objects in a pool for future reuse. This saves the overhead of having to create a new physical connection each time a connection is needed.

Using the DBXClient dbExpress Driver

Use the `TDBXPool` delegate driver to provide connection pooling support for DBXClient connections.

Using ADO.NET Providers

By default, the .NET client drivers implement a connection pool that is always in effect. Each `ConnectionString` has its own pool. There are connection properties that affect the maximum number of connections in a pool, and other attributes. There is also a `ConnectionPool` property that provides access to all active connection pools.

Blackfish SQL for Java Connection Pooling and Distributed Transaction Support

Blackfish SQL provides several components for dealing with JDBC `DataSources`, connection pooling, and distributed transaction (XA) support. These features require J2EE. If you are running with a JRE version less than 1.4, download the `J2EE.jar` file from java.sun.com, and add it to the classpath.

Blackfish for Java Connection Pooling

The `JdbcConnectionPool` object supports pooled XA transactions. This feature allows Blackfish SQL to participate in a distributed transaction as a resource manager. Blackfish SQL provides XA support by implementing these standard interfaces specified in the Java Transaction API (JTA) specification:

- `javax.sql.XAConnection`
- `javax.sql.XADataSource`
- `javax.transaction.xa.XAResource`

To acquire a distributed connection to a Blackfish SQL database from a `JdbcConnectionPool`, call `JdbcConnectionPool.getXAConnection()`. The connection returned by this method works only with the Blackfish SQL JDBC driver. XA support is useful only when combined with a distributed transaction manager, such as the one provided by Borland Enterprise Server.

Under normal operation, all global transactions should be committed or rolled back before the associated `XAConnection` is closed. If a connection is participating in a global transaction that is not yet in a prepared state but is in a successful started or suspended state, the transaction will be rolled back during any crash recovery that may occur.

The Heuristic Completion JDBC Extended Property

Blackfish SQL provides `heuristicCompletion`, an extended JDBC property that allows you to control the behavior when one or more databases fail during a two-phase commit. When XA transactions are prepared but not completed (no commit or rollback has been performed), the behavior is determined by one of these property settings:

- **`commit`** causes the transaction to be heuristically committed when Blackfish SQL reopens the database. This is the default.
- **`rollback`** causes the transaction to be heuristically rolled back when Blackfish SQL reopens the database.
- **`none`** causes Blackfish SQL to keep the transaction state when reopening a database. When this option is used, the locks that were held when the transaction was prepared are reacquired and held until the transaction is committed or rolled back by a JTA/JTS-compliant transaction manager.

The heuristic `commit` and `rollback` options allow for more efficient execution, because the locks can be released sooner and less information is written to the transaction log file.

The Blackfish SQL High Availability Server

NOTE: This feature is currently available only for Blackfish SQL for Java. It is not currently available in Blackfish SQL for Windows.

One of the most important areas of concern for any database application is eliminating single points of failure. The Blackfish SQL server provides a broad range of capabilities for making a database application fail-safe by avoiding application down time and loss of critical data. The High Availability server uses database mirroring technologies to ensure data availability in the event of software or hardware failure, and to provide a method of routine incremental backup. While a more general database replication scheme could provide similar protection, a mirroring approach provides advantages in terms of simplicity, ease of use, and performance.

A more general data replication solution could be employed to solve many of the same problems that the Blackfish SQL High Availability server addresses. Even though a more general solution would solve a broader variety of synchronization needs, it would do so at a much higher set of costs, including greater complexity and slower performance.

The Blackfish SQL database engine uses its transactional log files to maintain read-only mirror images of a database. The same TCP/IP database connections used for general database access are also used to synchronize mirrored databases.

Mirror Types

These mirror types can be used by an application:

- Primary
- Read-only
- Directory

The Primary Mirror

The primary mirror is the only mirror type that can accept both read and write operations to the database. Only one primary mirror is allowed at any time.

Read-only Mirrors

There can be any number of read-only mirrors. Connections to these databases can only perform read operations. Read-only mirrors provide a transactionally consistent view of the primary mirror database. However, a read-only mirror database might not reflect the most recent write operations against the primary mirror database. Read-only mirrors can be synchronized with changes to the primary mirror instantly, on a scheduled basis, or manually. Instant synchronization is required for

automatic failover. Scheduled and manual synchronization can be used for incremental synchronization or backup.

Directory Mirrors

Directory mirror databases mirror only the mirror configuration table and the tables needed for security definition. They do not mirror the actual application tables in the primary mirror.

There can be any number of directory mirrors. Connections to these databases can perform read operations, only. The storage requirements for a directory mirror database are very small, since they contain only the mirror table and security tables. Directory mirrors redirect read-only connection requests to read-only mirrors. Writable connection requests are redirected to the primary mirror.

The Blackfish SQL Engine and Failover

Blackfish SQL Engine failover handling capabilities include the following:

- Transaction log records
- Automatic failover
- Manual failover

Transaction log records

Blackfish SQL uses transaction log records to incrementally update mirrored databases. It transmits these log records to mirrors at high speed during synchronization operations. The same mechanism used for crash recovery and rollback is used to apply these changes. Existing code is used for all synchronization. The existing Blackfish SQL support for read-only transactions provides a transactionally consistent view of the mirrored data while the mirror is being synchronized with contents of more recent write transactions from the primary mirror.

Automatic failover

When a primary mirror that is configured with two or more automatic failover mirrors fails, one of the read-only mirrors that is configured for automatic failover is promoted to be the primary mirror. The application can be affected in one of two ways:

- If an application has already connected to the primary before it failed, all operations attempted against the failed primary will encounter SQLException or IOException errors. The application can cause itself to be hot swapped over to the new primary by rolling back the transaction. This is identical to how database deadlock is handled in high-concurrency OLTP applications.
- If an application has never connected to the primary before it failed, its connection attempt fails. Directory mirrors can be used to automatically redirect new connection requests to the new primary mirror.

Manual failover

Unlike automatic failover, manual failover is performed only on request. Any read-only mirror can become the primary mirror. This is useful when the computer the primary server is executing on needs to

be taken off line for system maintenance.

Advantages of the High Availability Server

The Blackfish SQL High Availability server provides a broad range of benefits, including:

- **No single point of failure**

  Since multiple copies of the same database are maintained across several computers, there is no need for a shared storage device. The High Availability server maintains the highest level of database availability with no single point of failure and with high speed failover and recovery, guaranteed data consistency, and transaction integrity.

- **Complete data and disaster protection**

  By maintaining copies of the database on multiple servers, the High Availability server guarantees that data remains intact in the event of media failure, a server crash, or any other catastrophic event.

- **Single, highly tuned network transport layer**

  The high-performance transport layer used for current database connections is also used for all synchronization operations.

- **Portability**

  The file format and synchronization is portable across all platforms that are capable of executing a Java Virtual Machine.

- **Large cost savings**

  The High Availability server provides a significant savings on today's high availability equipment and labor costs. It runs on standard low-cost hardware. There is no need for special technology such as shared disks, private LANs, or fiber channels, and no need for additional software or operating systems such as Linux, Windows, Solaris, or Mac OSX.

- **Easy to set up, administer and deploy**

  The High Availability server provides a high-performance, easy-to-use solution for some common database problems. There is no need for clustering expertise. All configuration settings and explicit operations can be performed using the Blackfish SQL Server Console, SQL scripts, or Java code.

- **Increased scalability and load balancing**

  Read-only operations can be performed against read-only mirrors, reducing the transaction work load of the primary mirror, which must be used for all transactions that perform write operations. By connecting to directory mirrors, new connection requests can be balanced across several read-only mirrors. This can dramatically reduce the work load of the primary server.

- **Synch delegation**

  You can specify the mirror to be used to synchronize another mirror. This allows the primary mirror to synchronize just one or a small number of read-only mirrors. These read-only mirrors can then synchronize other mirrors. This reduces the workload of the primary mirror, which must service all write requests.

- **Incremental database backup**

  Read-only mirrors can be synchronized with the primary mirror automatically by scheduling one or more synchronization time periods. Read-only mirrors can also be used for manual backup by making an explicit synchronization request.

- **Distributed directory**

Since this failover system supports the automatic and manual failover of servers, a distributed directory mechanism is useful for locating the primary mirror and available read-only mirrors. All mirrors maintain a table of all other mirrors. An application can make any type of connection request (read/write or read-only) to any existing mirror. The mirror uses the mirror table to determine where the current mirrors are located.

Heterogeneous Replication Using DataExpress with Blackfish SQL

NOTE: This feature is supported only for Blackfish SQL for Java.

The replication support provided by DataExpress for Blackfish SQL easier to use and deploy than most replication solutions. This replication solution is also heterogeneous because it uses JDBC for database access.

The replication topology provided is best described as a simple client-server relationship. Blackfish SQL does not require the server-side software or database triggers required for more complex publish-subscribe solutions. Complex multi-level hierarchies and network topologies are not directly supported.

There are three distinct phases in the replication cycle when using DataExpress JavaBean components with Blackfish SQL for a disconnected or mobile computing model:

- **Provide Phase:** Provides the client database with a snapshot of the server tables being replicated.
- **Edit Phase:** The client application, which need not be connected to the database, reads/edits the client database.
- **Resolve Phase:** The client database edits are saved back to the server database.

The Provide Phase

A `StorageDataSet` provider implementation initially replicates database contents from the server into a client. The client is always a Blackfish SQL database. The server is typically some server that can be accessed with a JDBC driver. The JDBC provider uses either a SQL query or stored procedure to provide data that will be replicated in the client-side Blackfish SQL database. Since there is no server-side software running in this architecture, there is no support for incremental updates from the server to the client. If the client needs to be refreshed, the same SQL query/stored procedure used to provide that the initial replication must be re-executed.

A `StorageDataSet` provider is a pluggable interface. `QueryDataSet` and `ProcedureDataSet` are extensions of `StorageDataSet`, which preconfigure `JdbcProviders` that can execute SQL queries and stored procedures to populate a `StorageDataSet`. For memory-constrained clients such as PDAs, a `DataSetData` component can be used to provide data. The `DataSetData` component uses Java Serialization to create a data packet that can be easily transmitted between a client and server.

The provide operation for a collection of database tables can be performed in a single transaction, so that

a transactionally consistent view of a collection of tables can be replicated.

The Edit Phase

Once the provide phase is complete, both DataExpress and JDBC APIs can read and write to the Blackfish SQL tables that are replicating the database. All insert/update/delete write operations since the last provide operation are automatically tracked by the Blackfish SQL storage system. Part of the `StorageDataSet` store interface contract is that all insert/update/delete operations must be recorded if the `StorageDataSet` property *`resolvable`* is set.

The Resolve Phase

DataExpress provides an automatic mechanism for using SQL DML or stored procedures to save all changes made on the client back to a server, via a JDBC driver. An optimistic concurrency approach is used to save the changes back to the server. One or more tables can be resolved in a single transaction. By default, any conflicts, such as two users updating the same row, cause the transaction to roll back. However, there is a `SqlResolutionManager` JavaBean component that you can use to customize the handling of resolution errors. The `SqlResolutionManager` has event handlers that enable an application to respond to an error condition with an `ignore`, `retry`, `abort`, `log`, or other appropriate response.

There are also higher-level `DataStorePump` and `DataStoreSync` components that you can use to perform provide and resolve operations for a collection of tables. For details, see Administering Blackfish SQL.

Chapter 3:
Establishing Connections

This chapter explains the basics for establishing a connection to a Blackfish SQL database, using dbExpress, ADO.NET, or JDBC.

Types of Connections

Connections can be local, remote, or a combination of both:

- **Local connections**

  Local connections access the Blackfish SQL database engine in-process. This provides improved performance over a remote driver, but requires that the Blackfish SQL engine be present and running in the same process as the application. Several simultaneous local connections created in the same process can connect to a database. However, only one process can have a database file open at a time. Consequently, while a process has a database file open, it becomes the only process that can connect to that database over a local connection.

- **Remote connections**

  To use a remote connection, first launch the Blackfish SQL server. (For instructions, see Administering Blackfish SQL.) The remote driver communicates with the server over TCP/IP. Remote connections can be slower for database interactions that involve multiple round trips between the client and the database for small packets of data. Remote connections allow more than one process running on one or more computers to access the same database. If several processes require simultaneous access, it is best to use remote connections.

- **Combination of local and remote connections**

  A third option is to use a combination of the local and remote connections. If there is one process that performs the majority of database interactions, this process can launch the Blackfish SQL Server inside its own process. In this way, the demanding database process can use the local driver while still allowing other processes to access the same database via the remote driver.

Using dbExpress to Connect to the Server

Native applications can use dbExpress to establish remote connections with a Blackfish SQL server. Local connections are currently not supported for dbExpress. You must start the Blackfish SQL server before you can use the remote dbExpress driver to connect. (For instructions, see Administering Blackfish SQL.)

Example

This example shows how to acquire a remote dbExpress connection:

```
[BlackfishSQL]
uses DBXCommon;
uses DBXClient;
var Connection: TDBXConnection;
Connection := TDBXConnectionFactory.GetConnectionFactory.
GetConnection('BLACKFISHSQLCONNECTION');
```

The dbExpress `dbxdriver.ini` file contains default driver properties appropriate for most applications. The dbExpress `dbxconnections.ini` file has a `BLACKFISHSQLCONNECTION` section that contains the default connection settings. New connections can copy most of these properties. This list of properties are typically customized for new connections:

- `[BLACKFISHCUSTOMCONNECTION]`
- `HostName=localhost`
- `port=2508`
- `Database=c:/tmp/test`

Using ADO.NET to Connect to the Server

The Blackfish SQL assembly `Borland.Data.BlackfishSQL.LocalClient.dll` contains an ADO.NET 2.0 driver. You can build an application without directly referencing this assembly by using the `DbProviderFactory` class. For this approach to work, the file `machine.config` must contain references to the Blackfish SQL assemblies in the `DbProviderFactory` section, and the Blackfish SQL assemblies must be installed in the Global Assembly Cache (GAC). For easier deployment, use a direct reference to the Blackfish SQL assembly.

You can use a local ADO connection, a remote ADO connection, or a combination of both to connect with the Blackfish SQL server.

Local Connections Using ADO.NET

You can establish a local ADO connection in either of the following ways:

Local ADO.NET Connection Using DbProviderFactory

Example

This example illustrates how to acquire a local ADO connection using `DbProviderFactory`:

```
[References: System.Data.dll]
uses System.Data.Common;
var Factory: DbProviderFactory;
var Connection: DbConnection;
Factory := DbProviderFactories.GetFactory('Borland.Data.BlackfishSQL.
LocalClient');

Connection := Factory.CreateConnection();
Connection.ConnectionString := 'database=<filename>;user=<username>;
password=<password>';

Connection.Open;
```

Local ADO.NET Connection Using a Direct Class Reference

Example

This example illustrates how to acquire a local ADO connection by using a direct class reference:

```
[References: System.Data.dll]
[References: Borland.Data.BlackfishSQL.LocalClient.dll]

uses System.Data.Common;
uses Borland.Data.DataStore;
var Connection: DbConnection;
Connection := DataStoreConnection.Create;
Connection.ConnectionString := 'database=<filename>;user=<username>;
password=<password>';

Connection.Open;
```

Remote Connections Using ADO.NET

Managed applications can use ADO.NET to establish remote connections with the Blackfish SQL server. You must start the server before you can use the remote ADO driver to connect. (For Instructions, see Administering Blackfish SQL.) Once the server is running, you can acquire a remote ADO connection in either of the following ways:

- Remote ADO.NET Connection Using DbProviderFactory
- Remote ADO.NET Connection Using a Direct Class Reference

Remote ADO.NET Connection Using DbProviderFactory

Example

This example shows how to acquire a remote ADO connection using `DbProviderFactory`:

```
[References: System.Data.dll]
uses System.Data.Common;
var Factory: DbProviderFactory;
var Connection: DbConnection;
Factory := DbProviderFactories.GetFactory('Borland.Data.BlackfishSQL.
RemoteClient');

Connection := Factory.CreateConnection();
Connection.ConnectionString :=
'database=<filename>;user=<username>;password=<password>;
host=<servername>;protocol=TCP';

Connection.Open;
```

Remote ADO.NET Connection Using a Direct Class Reference

Example

This example shows how to acquire a remote ADO connection using a direct class reference:

```
[References: System.Data.dll]
[References: Borland.Data.BlackfishSQL.RemoteClient.dll]

uses System.Data.Common;
uses Borland.Data.DataStore;
var Connection: DbConnection;
Connection := DataStoreConnection.Create;
Connection.ConnectionString :=
'database=<filename>;user=<username>;password=<password>;
host=<servername>;protocol=TCP';

Connection.Open;
```

Using JDBC to Connect to the Server

You can use a local JDBC connection, a remote JDBC connection, or a combination of both to connect with the Blackfish SQL server. The following sections provide instructions for each of these procedures.

Local Connections Using JDBC

A Blackfish SQL local JDBC connection allows an application to run in the same process as the Blackfish SQL engine. Applications that make large numbers of method calls into the JDBC API will see a significant performance advantage using the local Blackfish SQL driver.

You can establish a local JDBC connection in either of the following ways:

- Local JDBC Connection Using the DriverManager
- Local JDBC Connection Using a JDBC DataSource

Local JDBC Connection Using the DriverManager

Example

This example shows how to acquire a local JDBC connection using the DriverManager:

```
[jdsserver.jar must be in classpath]
java.sql.DriverManager.registerDriver(new com.borland.datastore.jdbc.
DataStoreDriver());
connection = java.sql.DriverManager.getConnection("jdbc:borland:dslocal:
<filename>", "<username>", "<password>");
```

Local JDBC Connection Using a JDBC DataSource

Example

This example shows how to acquire a local JDBC connection using a JDBC DataSource:

```
[jdsserver.jar must be in classpath]
com.borland.javax.sql.JdbcDataSource dataSource = new com.borland.javax.
sql.JdbcDataSource();
dataSource.setDatabaseName("<filename>");
connection = dataSource.getConnection("<username>", "<password>");
```

Remote Connections Using JDBC

Managed applications can use JDBC to establish remote connections with the Blackfish SQL server. You must start the server before you can use the remote ADO driver to connect. (For instructions, see Administering Blackfish SQL.)

You can establish a remote JDBC connection in either of the following ways:

- Remote JDBC Connection Using the DriverManager
- Remote JDBC Connection Using a JDBC DataSource

Remote JDBC Connection Using the DriverManager

Example

This example shows how to acquire a remote JDBC connection using the DriverManager:

```
[jdsremote.jar must be in classpath]
java.sql.DriverManager.registerDriver(new com.borland.datastore.
jdbc.DataStoreDriver());
connection = java.sql.DriverManager.getConnection("jdbc:
borland:dsremote://<servername>/<filename>", "<username>", "<password>");
```

Remote JDBC Connection Using a JDBC DataSource

Example

This example shows how to acquire a remote JDBC connection using a JDBC DataSource:

```
[jdsremote.jar must be in classpath]
com.borland.javax.sql.JdbcDataSource dataSource = new com.borland.javax.
sql.JdbcDataSource();
dataSource.setDatabaseName("<filename>");

dataSource.setNetworkProtocol("tcp");
datasource.setServerName("<servername>");

connection = dataSource.getConnection("<username>", "<password>");
```

Specifying Connection Properties

You can specify connection properties for:

- dbExpress
- ADO
- JDBC

For more information, see the RAD Studio help for `Borland.Data.DataStore.ConnectionProperties`.

Specifying dbExpress Connection Properties

dbExpress connection properties are documented in the `DbxCommon.pas` unit and in the Blackfish SQL `ConnectionProperties` class. dbExpress connection properties are stored in the `dbxconnections.init` file.

Example

This example shows a sample Blackfish SQL connection properties section in the `dbxconnections.ini` file:

```
[BLACKFISHSQLCONNECTION]
DriverName=BlackfishSQL
HostName=localhost
port=2508
Database=/tmp/test
create=true
User_Name=sysdba
Password=masterkey
BlobSize=-1
TransIsolation=ReadCommitted
```

The `HostName`, `port`, and `create` properties are documented in `ConnectionProperties`. The `DriverName`, `User_Name`, `BlobSize`, and `TransIsolation` properties are documented in `TDBXPropertyNames` of the DBXCommon unit.

Specifying ADO Connection Properties

The `ConnectionString` property in `DbConnection` or `DataStoreConnection` can contain settings from `ConnectionProperties`.

You can use DataExplorer to set and modify values for `ConnectionProperties`. For instructions, see the RAD Studio help for the DataExplorer Connection dialog box.

Specifying JDBC Connection Properties

You can specify JDBC connection properties using either:

- a JDBC URL
- `java.util.Properties`

Specifying JDBC Connection Properties in a JDBC URL

You can specify connection properties in a JDBC URL, using semicolons to separate the properties:

```
jdbc:borland:dslocal:c:/mydb.jds;create=true
```

Specifying JDBC Connection Properties with java.util.Properties

Example

This example shows how to specify JDBC connection properties using a `java.util.Properties` object:

```
java.util.Properties props = new java.util.Properties();
props.setProperty("create","true");
props.setProperty("user","SYSDBA");
props.setProperty("password","masterkey");
connection = DriverManager.getConnection("jdbc:borland:dslocal:c:/mydb.
jds", props);
```

Using Blackfish SQL with JBuilder and Borland Enterprise Server

To make the latest version of Blackfish SQL available to JBuilder and the Borland Enterprise Server (BES), copy these files from the Blackfish SQL `lib` directory to the `lib` directory of the target product:

- `beandt.jar`
- `dbtools.jar`
- `dx.jar`
- `jds.jar`
- `jdsremote.jar`
- `jdsserver.jar`

1. For JBuilder or BES, find the listed files in the `lib` directory of the install tree and copy them to a backup directory.
2. Find the files in the `lib` directory of the Blackfish SQL installation and copy them to the `lib` directory of JBuilder or BES.

DataDirectory Macro Support

You can use the `DataDirectory` macro to specify relative path names for database files. The `DataDirectory` macro is supported for both the Blackfish SQL ADO.NET and DBXClient drivers.

If a database file name is prepended with the following string:

```
|DataDirectory|
```

for example:

```
|DataDirectory|employee.jds
```

the string `|DataDirectory|` will be replaced with the appropriate string, as follows:

Blackfish SQL for Windows:

- For ASP.NET web based applications, this will be the `App_Data` folder name.
- For non-web applications, this defaults to the directory of the application executable. You can override the default by setting the `DataDirectory property` for `AppDomain`:

```
AppDomain.CurrentDomain.SetData("DataDirectory", "CustomAppPath")
```

Blackfish SQL for Java:

If the `System` property `blackfishsql.datadirectory` is set, the setting for this property will be used as the replacement string. Otherwise the setting for the `user.home` property will be used.

Chapter 4:
Administering Blackfish SQL

This chapter provides a brief overview of basic Blackfish SQL administrative procedures and tools. For related information see the SQL Reference.

Using the Graphical Consoles for Administrative Tasks

You can use visual tools to administer Blackfish SQL.

For Blackfish SQL for Windows:

You can use RAD Studio DataExplorer to perform many administrative tasks. DataExplorer has been enhanced with a connection string editor for Blackfish SQL for Windows, and the ability to create or alter Blackfish SQL databases. DataExplorer enables you to browse and view stored procedures. Some DataExplorer tasks are not yet supported for Blackfish SQL for Windows. See the DataExplorer help for more information.

For Blackfish SQL for Java:

You can use one of the JBuilder administrative consoles.

- JdsExplorer
- ServerConsole

Documentation for both JdsExplorer and ServerConsole are provided with JBuilder.

Using SQL for Administrative Tasks

You can perform virtually all Blackfish SQL administrative tasks either by using SQL commands or by using the built-in administrative stored procedures in the `DB_ADMIN` class.

Use administrative SQL commands to:

- Create, alter, and drop tables and views

- Create, alter, and drop users and roles
- Create and drop stored procedures and triggers

For details, see the SQL Reference.

Use DB_ADMIN stored procedures to:

- Alter database properties
- Verify database integrity
- Configure Database logging
- View open server connections
- Create, alter, and drop database mirrors
- Miscellaneous mirror administration capabilities

For details, see the Stored Procedures Reference.

Starting the Blackfish SQL Server

A server must be running before you can establish a remote connection. The server may be started as a .NET process or a Java process.

Starting and Stopping the Server as a .NET Process

To start or stop the server as a .NET process use either:

- The `BSQLServer.exe` command
- The Windows Management Console

Using `BSQLServer.exe`

- **To start the server:**
  `BSQLServer.exe`
- **To stop the server:**
  `BsqlServer.exe -shutdown` or
  Type `Ctrl-C` in the Console window
- **To install the server as a Windows service:**
  `BsqlServer.exe -install`

- **To remove the Windows server:**
  `BsqlServer.exe -remove`

- **To explore other options for server configuration:**
  `BsqlServer.exe -?`

Using the Windows Management Console

- **To start the server:**
  `net start BlackfishSQL`

- **To stop the server:**
  `net stop BlackfishSQL`

Starting the Server as a Java Process

To start or stop the server as a .NET process use either:

- The `JdsServer.exe` command
- The Windows Management Console

Using `JdsServer.exe`

- **To start the server:**
  `JdsServer.exe`

- **To stop the server:**
  `JdsServer.exe -shutdown` or

  Type `Ctrl-C` in the Console window

- **To install the server as a Windows service:**
  `JdsServer.exe -install JDataStore`

- **To remove the Windows server:**
  `JdsServer.exe -remove JDataStore`

- **To explore other options for the server configuration:**
  `JdsServer.exe -?`

Using the Windows Management Console

- **To start the server:**

```
net start JDataStore
```

- **To stop the server:**

```
net stop JDataStore
```

Chapter 5:
Using Blackfish SQL Security

This chapter provides a brief overview of basic Blackfish SQL security features and the SQL commands you can use to implement them. For a complete description of the syntax, use, and examples for a specific command, see the SQL Reference or the Stored Procedures Reference.

Blackfish SQL provides the following built-in security features:

- User authentication
- User authorization
- Database encryption

User Authentication

User authentication restricts access to a Blackfish SQL database to authorized users only. Users must log into the database using an authorized user account and password. Permissions can be granted to or revoked from an account to fine tune access. In general, full access is reserved for the Administrator account(s), and a more restricted account or accounts are provided for general users.

The Administrator Account

By default, Blackfish SQL has one built-in Administrator account, `sysdba/masterkey`. You can secure a database by changing the password for the `sysdba` account and restricting use of that account to database administrators only. You can then create a user account with limited access rights to be used for general access. You can also create additional Administrator accounts, which may or may not be granted database startup privileges.

The following section describes how to create and modify user accounts.

Managing User Accounts

You can use the following SQL statements to add, delete, and modify user accounts:

Adding a User

```
CREATE <userid> PASSWORD <password>
```

Where:

<userid> is the account to be added

<password> is the password for this account

Removing a User

```
DROP <userid> [ CASCADE|RESTRICT ]
```

Where:

<userid> is the account to be removed.

CASCADE deletes the user and all objects that the user owns.

RESTRICT causes the statement to fail if the user owns any objects, such as tables, views, or methods.

(no option) causes the statement to fail if the user owns any objects, such as tables, views, or methods.

Changing a User's Password

```
ALTER USER <userid> SET PASSWORD "<password>";
```

Where:

<userid> is the account for which the password should be changed.

<password> is the new password.

User Authorization

There are several database access privileges which you can grant to or revoke from an account
The following section describes the set of access privileges, and how to grant and revoke privileges for an account.

Changing User Access Privileges

You can use the GRANT and REVOKE statements to change the access privileges for one or more user accounts. You can grant or revoke access to specific database resources or specific objects in the database. In addition, you can grant specific privileges to named roles, and you can then grant or revoke these roles for specific users.

To grant or revoke a privilege for an account, use the following SQL commands:

`GRANT <role>|<privilege> TO <userid>`

Grants the specified privilege or role to the specified user account.

`REVOKE <role>|<privilege> FROM <userid>`

Revokes the specified privilege or role from the specified user account.

Where:

<userid> is the account to be modified.

<role> is the user role to be granted or revoked, such as `ADMIN`. This can be a single role, or a comma-separated list of roles.

<privilege> is the privilege to be granted or revoked. This can be a single privilege or a comma-separated list of privileges, and can be one or more of the following:

- `STARTUP` confers the ability to open a database that is shut down. The user's password is required to add `STARTUP` rights to a user account. You can also specify `STARTUP` rights at the time the user account is created.
- `ADMINISTRATOR` confers the ability to add, remove, and change rights of users, and the ability to encrypt the database. Also includes the four stream rights: `WRITE`, `CREATE`, `DROP`, `RENAME`. By default, `STARTUP` rights are granted to an Administrator account when the account is created, but you can remove `STARTUP` rights from the account. You cannot remove `WRITE`, `CREATE`, `DROP`, or `RENAME` privileges from an Administrator account; attempts to remove these rights are ignored.
- `WRITE` confers the ability to write to file or table streams in the Blackfish SQL database.
- `CREATE` confers the ability to create new file or table streams in the Blackfish SQL database.
- `DROP` confers the ability to remove file or table streams from the Blackfish SQL database.
- `RENAME` confers the ability to rename file or table streams in the Blackfish SQL database.

Database Encryption

Only a user with `Administrator` privileges can encrypt a database. When a database is encrypted, the `STARTUP` privilege is automatically revoked for all users (including Administrators) other than the Administrator issuing the encryption command. Consequently, after encrypting you must use the same Administrator account to restart the database. You can reassign `STARTUP` privileges to other users after the database has been encrypted and restarted.

Encrypting a Blackfish SQL Database

You can use the built-in stored procedure DB_ADMIN.ENCRYPT() to encrypt a new or empty Blackfish SQL database. For instructions on encrypting a non-empty database, see Encrypting a Database with Existing Content

To encrypt a new database, log in from an Administrator account, and issue the following SQL command:

```
CALL DB_ADMIN.ENCRYPT(<AdminPassword>,<EncryptionSeed>)
```

Where:

DB_ADMIN.ENCRYPT() is the built-in stored procedure for encrypting a database.

<AdminPassword> is the password for the user issuing the encryption command.

<EncryptionSeed> is a 16 character seed value.

Encrypting a Database with Existing Content

For Blackfish SQL for Java:

To encrypt a Blackfish SQL for Java database that has existing tables, use the JBuilder utility, JdsExplorer. For instructions, see the JBuilder online help for JdsExplorer.

For Blackfish SQL for Windows:

To encrypt a database with existing content, do the following:

1. Use RAD Studio DataExplorer to create a new database. For instructions, see the online help for DataExplorer.
2. Copy the existing users to the new database.

   DB_ADMIN.COPY_USERS(*<OtherFilename>, <AdminUser>, <AdminPass>,<DoCopyEncryption>, <ReplaceExistingUsers>*)

   Where:

   <OtherFilename> is the filename of the destination database.

   <AdminUser> is a user with ADMIN privileges in destination database.

   <AdminPass> is a password of the specified adminUser in destination database.

   If *<DoCopyEncryption>* is TRUE and the current database is encrypted, then encrypt the target database with the same key. If *<ReplaceExistingUsers>* is TRUE, then the existing users in the target database is replaced with the users in the source database.
3. Encrypt the new database.
   DB_ADMIN.ENCRYPT(*<password>,<EncryptionSeed>*)

4. Copy the contents of the old database into the newly encrypted database.

   `DB_ADMIN.COPY_STREAMS(<OtherFilename>, <AdminUser>, <AdminPass>, <DoOverwrite>, <DoIgnoreErrors>)`

   Where:

   `<OtherFilename>` is the filename of the destination database.

   `<AdminUser>` is a user with ADMIN privileges in destination database.

   `<AdminPass>` is a password of the specified adminUser in destination database.

   If `<DoOverwrite>` is TRUE, it allows tables to be overwritten. If FALSE, this would be an error.

   If `<IgnoreErrors>` is TRUE, then this method can be used to repair a corrupted database.

For additional information, see the Stored Procedures Reference.

Deciding How to Apply Blackfish SQL Security

In this discussion, an *opponent* is someone who is trying to break the Blackfish SQL security system.

The authentication and authorization support is secure for server-side applications where opponents do not have access to the physical Blackfish SQL database files. The SYS.USERS table stores passwords, user IDs, and rights in encrypted form. The table also stores the user ID and rights in an unencrypted column, but this is for display purposes only. The encrypted values for user ID and rights are used for security enforcement.

The stored passwords are encrypted using a strong *TwoFish* block cipher. A pseudo-random number generator is used to salt the encryption of the password. This makes traditional password dictionary attacks much more difficult. In a dictionary attack, the opponent makes guesses until the password is guessed. This process is easier if the the opponent has personal information about the user, and the user has chosen an obvious password. There is no substitution for a well chosen (obscure) password as a defense against password dictionary attacks. When an incorrect password is entered, the current thread sleeps for 500 milliseconds.

If a Blackfish SQL database is unencrypted, it is important to restrict physical access to the file, for the following reasons:

- If a Blackfish SQL database file is not password protected, and it is possible for an opponent to write to it with a separate file editing utility or program, the authentication and authorization support can be disabled.
- If it is possible for an opponent to read a Blackfish SQL database file that is not encrypted with a separate file-editing program, the opponent might be able to reverse-engineer the file format and view its contents.

For environments where a dangerous opponent may gain access to physical copies of a Blackfish SQL database, the database and log files should be encrypted, in addition to being password protected.

WARNING: The cryptographic techniques that Blackfish SQL uses to encrypt data blocks are state-of-the-art. The TwoFish block cipher used by Blackfish SQL has never been defeated. This means that if

you forget your password for an encrypted Blackfish SQL database, you will not be able to access the database. The best chance of recovering the data would be to have someone guess the password.

There are measures that can be used to guard against forgetting a password for an encrypted database. It is important to note that there is a master password used internally to encrypt data blocks. Any user that has STARTUP rights has the master password encrypted using their password in the SYS.USERS table. This allows one or more users to open a database that has been shut down, because their password can be used to decrypt a copy of the master password. This feature can be used to create a new database that has one secret user who has Administrator privileges (which includes STARTUP rights). If you use this virgin database whenever a new empty database is needed, you will always have one secret user who can unlock the encryption.

Encrypting a database has some effect on performance. Data blocks are encrypted when they are written from the Blackfish SQL cache to the Blackfish SQL database and are decrypted when they are read from the Blackfish SQL database into the Blackfish SQL cache. So the cost of encryption is only incurred when file I/O is performed.

Blackfish SQL encrypts all but the first 16 bytes of `.jds` file data blocks. There is no user data in the first 16 bytes of a data block. Some blocks are not encrypted. This includes allocation bitmap blocks, the header block, log anchor blocks and the SYS.USERS table blocks. Note that the sensitive fields in the SYS.USERS table are encrypted using the user's password. Log file blocks are completely encrypted. Log anchor and status log files are not encrypted. The temporary database used by the query engine is encrypted. Sort files used by large merge sorts are not encrypted, but they are deleted after the sort completes.

NOTE: The remote client for Blackfish SQL currently uses sockets to communicate with a Blackfish SQL Server. This communication is not secure. Since the local client for Blackfish SQL is in-process, it is secure.

Chapter 6:
Using Stored Procedures and User Defined Functions

Blackfish SQL supports stored procedures to encapsulate business logic in the schema of a database. In addition, Blackfish SQL supports User Defined Functions (UDFs) to extend the built-in SQL support. Where many other database vendors have invented their own SQL-like language for stored procedures, Blackfish SQL can access stored procedures and UDFs created in any .NET language such as Delphi, C#, VB and Java.

Stored procedures can also increase the performance of an application, since they are executed in the same Virtual Machine as the Blackfish SQL database engine itself. This results in execution with minimal overhead. While a stored procedure is executing SQL statements, no network traffic is generated. The stored procedure uses an in-process ADO.NET connection. This provides the same performance advantage as using the in-process Blackfish SQL ADO.NET driver rather than the remote driver.

Stored procedures and UDFs provide these additional benefits:

- Business logic, such as integrity constraints, is isolated in the database engine, where the logic is available and reinforced for all clients.
- Data is retrieved locally, which is faster than sending that data to and from the client.
- Blackfish SQL language can be extended with C#, Delphi, or Visual Basic functions.
- There is no performance penalty, since the stored procedures are executing in the same virtual machine as the database itself.
- You can debug .NET stored procedures in the same manner as debugging the client application.

This chapter covers:

About Stored Procedures

Stored procedures are procedures that are stored on the database server and executed on request from an SQL client. Often the stored procedure executes several SQL queries against the tables of the database to yield the desired result. In Blackfish SQL, these SQL queries are written in the language of choice, that is available on the .NET or Java platforms. The desired effect may be to update a set of tables, or to calculate an accumulated value from one or more tables, or to add specialized integrity constraints. A stored procedure may have several parameters, which can be either input only, output only, or both.

Example

Consider an ADD_ORDER procedure that takes a customerId, an itemId, and a quantity value as input, and adds a record to the ORDERS table. However, suppose that you also want to verify that this customer has paid for previous orders. To achieve this, you can cause the procedure to throw an exception if this is not the case.

The stored procedure is executed with an IDbCommand object by setting the properties CommandType and CommandText, and then adding the appropriate parameters.

CommandText	CommandType	Parameters
'CALL ADD_ORDER(?,?,?)'	CommandType.Text	Added in order from left to right
'CALL ADD_ORDER(:CUSTID,:ITEMID,:QUANTITY)'	CommandType.Text	Added by name of marker
'ADD_ORDER'	CommandType.StoredProcedure	Added by name of parameter

Notice the difference in the interpretation of the parameters, depending on the combination of CommandType and the style of the parameter markers that are used. If the CommandType is StoredProcedure, the parameter names are taken from the implementation of the stored procedure, in which case it is possible to omit optional parameters.

About User Defined Functions (UDFs)

A User Defined Function is a code snippet that is written to extend the built-in SQL support. Like stored procedures, they are executed on the database server and called from an SQL client. UDFs must return a value, and are usually written to be used in the WHERE clause of SELECT queries. However, a UDF may also be called by itself, similar to a stored procedure.

Example

Consider a MAX_VALUE function that takes two values, <*value1*> and <*value2*>, and returns the greater of the two. The UDF can be executed in an SQL statement:

```
'SELECT * FROM PEOPLE WHERE MAX_VALUE(HEIGHT,5*WIDTH) < ?'
```

Or, in an SQL CALL statement:

```
'?=CALL MAX_VALUE(?,?)'
```

Creating Stored Procedures for the .NET Platform

This section provides detailed information on how to create Blackfish SQL stored procedures and UDFs for the .NET platform.

Creating a Stored Procedure for a Blackfish SQL Database

There are three steps involved in creating a Blackfish SQL stored procedure:

1. Write the code for the stored procedure as a static public member of a class.
2. Build an assembly with the stored procedures.

   Blackfish SQL must be able to locate the assembly. When developing in Delphi, Blackfish SQL is able to find the assembly in `BDSCOMMONDIR`. That is, it is not necessary to move the assembly to any special location. For deployment, it is recommended that you copy the assembly to the subdirectory where the executable for the Blackfish SQL server (`BSQLServer.exe`) resides, or install it in the Global Assembly Cache (GAC).
3. Create the binding of a SQL identifier to the assembly member.

Example

This example uses the sample `ADD_ORDER` from the previous example in About Stored Procedures, with this schema:

CUSTOMER TABLE

Field	Type	Description
`CUST_ID`	`INT`	Customer identifier
`CREDIT`	`DECIMAL(10,2)`	Credit amount available to the customer
`NAME`	`VARCHAR(80)`	Customer name

ORDERS TABLE

Field	Type	Description

CUST_ID	INT	Customer identifier
ITEM_ID	INT	Item identifier
QUANTITY	INT	How many items
SALE_AMOUNT	DECIMAL(10,2)	Total sale amount
PAID	DECIMAL(10,2)	Amount paid

ITEMS TABLE

Field	Type	Description
ITEM_ID	INT	Item identifier
NAME	VARCHAR(60)	Name of the item
PRICE	DECIMAL (10,2)	Unit price
STOCK	INT	Stock count

Step 1: Write the code for the stored procedure.

1. Create a Delphi.NET package and name it `MyProcs.dll`.
2. Add a reference to `System.Data.dll`.
3. Add a unit:

```
P1 := Command.Parameters.Add('P1', DbType.Decimal);
P2 := Command.Parameters.Add('P2', DbType.Int32);
P1.Direction := ParameterDirection.Output;
P2.Value := CustId;
Command.ExecuteNonQuery;

if P1.Value = DBNull.Value then

  Owed := 0

else

  Owed := Decimal(P1.Value);

Owed := Owed + Amount;

Command.Parameters.Clear;

Command.CommandText := 'SELECT CREDIT INTO ? FROM CUSTOMER WHERE
CUST_ID=?';
P1 := Command.Parameters.Add('P1', DbType.Decimal);
P2 := Command.Parameters.Add('P2', DbType.Int32);
P1.Direction := ParameterDirection.Output;
P2.Value := CustId;
Command.ExecuteNonQuery;

Credit := Decimal(P1.Value);

if Owed > Credit then

 raise Exception.Create('Customer doesn''t have that much credit');

Command.Parameters.Clear;

Command.CommandText := 'UPDATE ITEMS SET STOCK=STOCK-? WHERE ITEM_ID=?';
P1 := Command.Parameters.Add('P1', DbType.Int32);
P2 := Command.Parameters.Add('P2', DbType.Int32);
P1.Value := Quantity;
P2.Value := ItemId;
Command.ExecuteNonQuery;

Command.Parameters.Clear;

Command.CommandText := 'INSERT INTO ORDERS (CUST_ID, ITEM_ID, QUANTITY,
SALE_AMOUNT) '+ 'VALUES (?, ?, ?, ?)';
P1 := Command.Parameters.Add('P1', DbType.Int32);
P2 := Command.Parameters.Add('P2', DbType.Int32);
P3 := Command.Parameters.Add('P3', DbType.Int32);
P4 := Command.Parameters.Add('P4', DbType.Decimal);

P1.Value := CustId;
P2.Value := ItemId;
P3.Value := Quantity;
```

```
P4.Value := Amount;
Command.ExecuteNonQuery;
Command.Free;
end;

end.
```

Step 2: Build the assembly and make it available to the Blackfish SQL server process.

After completing the code for the stored procedure:

1. Build an assembly DLL (for example, `Procs.dll)` which contains the class `MyClass` shown in Step 1.
2. When deploying, copy the assembly to the subdirectory where the executable for the Blackfish SQL server (`BSQLServer.exe`) resides.

Step 3: Create the binding of an SQL identifier to the class member.

Now that the code is ready to be executed, the Blackfish SQL database must be made aware of the class member that can be called from SQL. To do this, start DataExporer and issue a `CREATE METHOD` statement:

```
CREATE METHOD ADD_ORDER AS 'MyProcs::SampleStoredProcedures.TMyClass.
AddOrder';
```

`MyProcs` is the name of the package, and the method name is fully-qualified with unit name and class name.

Execute the stored procedure `ADD_ORDER` from a Delphi Console application:

```
unit MyCompany;

interface

implementation
uses
  System.Data;

type
  TSomething = class
  public
    procedure AddOrder(
      Connection: DbConnection;
      CustId: Integer;
      ItemId: Integer;
      Quantity: Integer);
    end;

{ Assume:
   Connection: is a valid Blackfish SQL connection.
   CustId:     is a customer in the CUSTOMER table.
   ItemId:     is an item from the ITEMS table.
   Quantity:   is the quantity of this item ordered.
   }

procedure TSomething.AddOrder(
  Connection: DbConnection;
  CustId: Integer;
  ItemId: Integer;
  Quantity: Integer);
var
  Command: DbCommand;
  P1, P2, P3: DbParameter;
begin
   Command := con.CreateCommand;
   Command.CommandText := 'ADD_ORDER';
   Command.CommandType := CommandType.StoredProcedure;
   P1 := Command.Parameters.Add('custId', DbType.Int32);
   P2 := Command.Parameters.Add('itemId', DbType.Int32);
   P3 := Command.Parameters.Add('quantity', DbType.Int32);
   P1.Value := CustId;
   P2.Value := ItemId;
   P3.Value := Quantity;
   Command.ExecuteNonQuery;
   Command.Free;
  end;

 end.
```

When `TSomeThing.AddOrder` is called in the client application, this in turn calls the stored procedure `ADD_ORDER`, which causes `TMyClass.AddOrder` to be executed in the

Blackfish SQL server process. By making `TMyClass.AddOrder` into a stored procedure, only one statement has to be executed over a remote connection. The five statements executed by `TMyClass.AddOrder` are executed in-process of the Blackfish SQL server, using a local connection.

Note that the application is not passing a connection instance to the call of the `ADD_ORDER` stored procedure. Only the actual logical parameters are passed.

Blackfish SQL generates an implicit connection object, when it finds a stored procedure or UDF where the first argument is expected to be a `System.Data.IDbConnection` instance.

Handling Output Parameters and DBNull Values

The Delphi language supports output parameters and reference parameters. The Blackfish SQL database recognizes these types of parameters and treats them accordingly.

Database NULL values require special handling. The `System.String` can be handled by the NULL value. However, for all other types, the formal parameter type must be changed to `TObject`, since NULL is not a valid value for a .NET `ValueType`. If the formal parameter is a `TObject` type, then the value of `System.DBNull` is used for a database NULL value. Blackfish SQL will also accept nullable types in stored procedures written in C# (for example, `int`).

Examples:

Example of a stored procedure with an INOUT parameter; NULL values are ignored:

```
class procedure TMyClass.AddFive(ref Param: Integer);
begin
  Param := Param + 5;
end;
```

Example of a stored procedure with an INOUT parameter; NULL values are kept as NULL values:

```
class procedure TMyClass.AddFour(ref Param: TObject);
begin
  if Param <> nil then
    Param := TObject(Integer(Param) + 4);
end;
```

Use:

```
procedure TryAdding(Connection: DbConnection);
var
  Command: DbCommand;
begin
  Command := Connection.CreateCommand;
  Command.CommandText := 'ADD_FIVE';
  Command.CommandType := CommandType.StoredProcedure;
  P1 := Command.Parameters.Add('param', DbType.Int32);
  P1.Direction := ParameterDirection.InputOutput;
  P1.Value = 17;
  Command.ExecuteNonQuery;
    if 22 <> Integer(P1.Value) then
      raise Exception.Create('Wrong result');

  Command.Parameters.Clear;
  Command.CommandText := 'ADD_FOUR';
  Command.CommandType := CommandType.StoredProcedure;
  P1 := Command.Parameters.Add('param', DbType.Int32);
  P1.Direction := ParameterDirection.InputOutput;
  P1.Value = 17;
  Command.ExecuteNonQuery;
    if 21 <> Integer(P1.Value) then
      raise Exception.Create('Wrong result');

  P1.Value = DBNull.Value;
  Command.ExecuteNonQuery;
    if DbNull.Value <> P1.Value then
      raise Exception.Create('Wrong result');

  Command.Free;
end;
```

The above implementation of `AddFour` uses a `TObject` wrapper class for integers. This allows the developer of `addFour` to recognize NULL values passed by Blackfish SQL, and to set an output parameter to NULL to be recognized by Blackfish SQL.

In contrast, in the implementation for `AddFive`, it is impossible to know if a parameter was NULL, and it is impossible to set the result of the output parameter to NULL.

Expanding SQL for the Blackfish SQL Database

If for some reason an operator (for example: a bitwise AND operator) is needed for a `where` clause, and Blackfish SQL does not offer that operator, you can create one in Delphi, Visual Basic, C#, or C++ and call it as a UDF. However, use this capability with caution, since Blackfish SQL will not recognize the purpose of such a function, and will not be able to use any indices to speed up this part of the query.

Consider the UDF example given earlier, involving the `MAX_VALUE` UDF:

```
'SELECT * FROM PEOPLE WHERE MAX_VALUE(HEIGHT,5*WIDTH) < ?'
```

That query is equivalent to this query:

```
'SELECT * FROM PEOPLE WHERE HEIGHT < ? AND 5*WIDTH < ?'
```

where the same value is given for both parameter markers. This SQL statement yields the same result, because the implementation of `MAX_VALUE` is known. However, Blackfish SQL will be able to use only indices available for the `HEIGHT` and `WIDTH` column for the second query. If there were no such indices, the performance of the two queries would be about the same. The advantage of writing a UDF occurs when functionality does not already exist in Blackfish SQL (for example: a bit wise AND operator).

Debugging .NET Stored Procedures

To debug .NET stored procedures:

- When the protocol is in-process or not set
- When the protocol is TCP

Debugging Stored Procedures When the Protocol Is In-process or Not Set

To debug stored procedures when the protocol is in-process or not set:

1. Create a project to use for debugging.

   Using your favorite IDE, create a project that includes the client code of the application, the stored procedures, and a reference to the `Borland.Data.BlackfishSQL.LocalClient.dll` library.

2. Add breakpoints to the stored procedure(s).

   The debugger will handle the stored procedures in the same way as with the client code.

Debugging Stored Procedures When the Protocol Is TCP

To debug stored procedures when the protocol is TCP:

If your IDE supports remote debugging:

Delphi will be able to attach to the Blackfish SQL Server process.

1. Compile the stored procedures with debug information.
2. Copy the assembly to the bin directory of the Blackfish SQL installation.
3. Start the client application in the debugger and attach to the server process.
4. Add breakpoints to the stored procedure(s).

   The debugger will handle the stored procedures in the same way as with the client code.

If your IDE does not support remote debugging:

1. Create a project to use for debugging.

   Set up the project to debug the server directly.
2. Create an executable that calls `Borland.Data.DataStore.DataStoreServer.StartDefaultServer`.
3. Add a breakpoint to the stored procedure.
4. Run the separate client process.

Using a Stored Procedure to Produce an ADO.NET IDataReader

A stored procedure can produce an ADO.NET `DbDataReader` simply by returning a `DbDataReader`.

Example

```
class function GetRiskyCustomers(
  Connection: DbConnection;
  Credit: Decimal credit): DbDataReader;
var
  Command: DbCommand;
  P1: DbParameter;
begin
  Command := Connection.CreateCommand;
  Command.CommandText := 'SELECT NAME FROM CUSTOMER WHERE CREDIT > ? ';
  P1 := Command.Parameters.Add('param', DbType.Decimal);
  P1.Value := Credit;
  Result := Command.ExecuteReader;
end;
```

Note that the `command` object is not freed at the end of the method. If the command was freed, it would implicitly close the `DbDataReader`, which results in no data being returned from the stored procedure. Instead, Blackfish SQL closes the command implicitly after the stored procedure has been called.

The `GetRiskyCustomers` stored procedure can be used as follows, in ADO:

```
function GetRiskyCustomers(
  Connection: DbConnection): ArrayList;
var
  Command: DbCommand;
  Reader: DbReader;
  List: ArrayList;
begin
  List := ArrayList.Create;
  Command := Connection.CreateCommand;
  Command.CommandText := 'GETRISKYCUST';
  Command.CommandType := CommandType.StoredProcedure;
  P1 := Command.Parameters.Add('Credit', DbType.Decimal);
  P1.Value := 2000;
  Reader := Command.ExecuteReader;
  while Reader.Read do
    List.Add(Reader.GetString(0));
  Command.Free;
  Result := List;
end;
```

Creating Stored Procedures for the Java Platform

This section provides detailed information on how to create Blackfish SQL stored procedures and UDFs for the Java platform.

Creating a Stored Procedure for a Blackfish SQL Database

Stored procedures and UDFs for Blackfish SQL for Java must be written in Java. The compiled Java classes for stored procedures and UDFs must be added to the `CLASSPATH` of the Blackfish SQL server process in order to be available for use. This should give the database administrator a chance to control which code is added. Only `public static` methods in `public` classes can be made available for use.

You can update the classpath for the Blackfish SQL tools by adding the classes to the `<jds_home>/lib/storedproc` directory.

- If the stored procedure consists of a `.jar` file, then place the jar file in `<jds_home>/storedproc/lib/jars`.
- If the stored procedure consists of one or more class files, place the class files in `<jds_home>/storedproc/classes`. For example, if your stored procedure file is `com.acme.MyProc`, then you would place it as: `c:<jds_home>/lib/storedproc/classes/com/acme/MyProc.class`

Making a Stored Procedure or UDF Available to the SQL Engine

After a stored procedure or a UDF has been written and added to the CLASSPATH of the Blackfish SQL server process, use this SQL syntax to associate a method name with it:

```
CREATE JAVA_METHOD <method-name> AS <method-definition-string>
```

<method-name> is a SQL identifier such as INCREASE_SALARY and *<method-definition-string>* is a string with a fully qualified method name. For example:

```
com.mycompany.util.MyClass.increaseSalary
```

Stored procedures and UDFs can be dropped from the database by executing:

```
DROP JAVA_METHOD <method-name>
```

After a method is created, it is ready for use. The next example shows how to define and call a UDF.

A UDF Example

This example defines a method that locates the first space character after a certain index in a string. The the first SQL snippet defines the UDF and and the second shows an example of how to use it.

Assume that TABLE1 has two VARCHAR columns: FIRST_NAME and LAST_NAME.
The CHAR_LENGTH function is a built-in SQL function.

```
package com.mycompany.util;
public class MyClass {
   public static int findNextSpace(String str, int start) {
     return str.indexOf(' ',start);
   }
}

CREATE JAVA_METHOD FIND_NEXT_SPACE AS
  'com.mycompany.util.MyClass.findNextSpace';

SELECT * FROM TABLE1
  WHERE FIND_NEXT_SPACE(FIRST_NAME, CHAR_LENGTH(LAST_NAME)) < 0;
```

Input Parameters

A final type-checking of parameters is performed when the Java method is called. Numeric types are cast to a higher type if necessary in order to match the parameter types of a Java method. The numeric type order for Java types is:

1. `double` or `Double`
2. `float` or `Float`
3. `java.math.BigDecimal`
4. `long` or `Long`
5. `int` or `Integer`
6. `short` or `Short`
7. `byte` or `Byte`

The other recognized Java types are:

- `boolean` or `Boolean`
- `String`
- `java.sql.Date`
- `java.sql.Time`
- `java.sql.Timestamp`
- `byte[]`
- `java.io.InputStream`

Note that if you pass NULL values to the Java method, you cannot use the primitive types such as `short` and `double`. Use the equivalent encapsulation classes instead (`Short`, `Double`). A SQL `NULL` value is passed as a Java `null` value.

If a Java method has a parameter or an array of a type that is not listed in the tables above, it is handled as SQL `OBJECT` type.

Output Parameters

If a Java method parameter is an array of one of the recognized input types (other than `byte[]`), the parameter is expected to be an output parameter. Blackfish SQL passes an array of length 1 (one) into the method call, and the method is expected to populate the first element in the array with the output value. The recognized Java types for output parameters are:

- `double[]` or `Double[]`
- `float[]` or `Float[]`
- `java.math.BigDecimal[]`
- `long[]` or `Long[]`
- `int[]` or `Integer[]`
- `short[]` or `Short[]`
- `Byte[]` (but not `byte[]` since that is a recognized input parameter by itself)
- `boolean[]` or `Boolean[]`
- `String[]`
- `java.sql.Date[]`
- `java.sql.Time[]`
- `java.sql.Timestamp[]`
- `byte[][]`
- `java.io.InputStream[]`

Output parameters can be bound only to variable markers in SQL. All output parameters are essentially `INOUT` parameters, since any value set before the statement is executed is passed to the Java method. If no value is set, the initial value is arbitrary. If any of the parameters can output a SQL NULL (or have a valid NULL input), use the encapsulated classes instead of the primitive types.

Example

```
package com.mycompany.util;
public class MyClass {
   public static void max(int i1, int i2, int i3, int result[]) {
       result[0] = Math.max(i1, Math.max(i2,i3));
  }
}

CREATE JAVA_METHOD MAX
   AS 'com.mycompany.util.MyClass.max';

CALL MAX(1,2,3,?);
```

The `CALL` statement must be prepared with a `CallableStatement` in order to get the output value. See the JDBC documentation for how to use `java.sql.CallableStatement`. Note the assignment of `result[0]` in the Java method. The array passed into the method has exactly one element.

Implicit Connection Parameters

If the first parameter of a Java method is of type `java.sql.Connection`, Blackfish SQL passes a connection object that shares the transactional connection context used to call the stored procedure. This connection object can be used to execute SQL statements using the JDBC API.

Do *not* pass anything for this parameter. Let Blackfish SQL do it.

Example

```
package com.mycompany.util;
public class MyClass {
    public static void increaseSalary(java.sql.Connection con,
       java.math.BigDecimal amount) {
    java.sql.PreparedStatement stmt
        = con.prepareStatement("UPDATE EMPLOYEE SET SALARY=SALARY+?");
              stmt.setBigDecimal(1,amount);
              stmt.executeUpdate();
              stmt.close();
    }
}


CREATE JAVA_METHOD INCREASE_SALARY
   AS 'com.mycompany.util.MyClass.increaseSalary';

CALL INCREASE_SALARY(20000.00);
```

Note:

- `INCREASE_SALARY` requires only one parameter: the amount by which to increase the salaries. The corresponding Java method has two parameters.
- Do not call `commit()`, `rollback`, `setAutoCommit()`, or `close()` on the connection object passed to stored procedures. For performance reasons, it is not recommended to use this feature for a UDF, even though it is possible.

Stored Procedures and JDBC ResultSets

A Java stored procedure can produce a `ResultSet` on the client by returning either a `ResultSet` or a DataExpress `DataSet` from the Java implementation of the stored procedure. The `DataSet` is automatically converted to a `ResultSet` for the user of the stored procedure.

Example

This example returns a `ResultSet`:

```
package com.mycompany.util;

public class MyClass {
    public static void getMarriedEmployees(java.sql.Connection con)
      java.sql.Statement stmt = con.getStatement();
      java.sql.ResultSet rset
        = stmt.executeQuery("SELECT ID, NAME FROM EMPLOYEE
                             WHERE SPOUSE IS NOT NULL");
      return rset;
}
```

Note: Do not close the `stmt` statement. This statement is closed implicitly.

Example

This example returns a `DataSet`, which is automatically converted to a ResultSet:

```
package com.mycompany.util;

public class MyClass {
    public static void getMarriedEmployees()
    com.borland.dx.dataset.DataSet dataSet = getDataSetFromSomeWhere();
    return dataSet;
}
```

Note: Do not close the `stmt` statement. This statement is closed implicitly.

Example

Register and call the previous examples like this:

```
java.sql.Statement stmt = connection.getStatement();
stmt.executeUpdate("CREATE JAVA_METHOD GET_MARRIED_EMPLOYEES AS "+
                   "'com.mycompany.util.MyClass.getMarriedEmployees'");
java.sql.ResultSet rset = stmt.executeQuery("CALL GET_MARRIED_EMPLOYEES
()");
int id = rset.getInt(1);
String name = rset.getString(2);
```

Overloaded Method Signatures

Java methods can be overloaded to avoid numeric loss of precision.

Example

```
package com.mycompany.util;
public class MyClass {
  public static int abs(int p) {
    return Math.abs(p);
  }

  public static long abs(long p) {
    return Math.abs(p);
  }

  public static BigDecimal abs(java.math.BigDecimal p) {
    return p.abs();
  }

  public static double abs(double p) {
    return Math.abs(p);
  }
}


CREATE JAVA_METHOD ABS_NUMBER AS 'com.mycompany.util.MyClass.abs';

SELECT * FROM TABLE1 WHERE ABS(NUMBER1) = 2.1434;
```

The overloaded method `abs` is registered only once in the SQL engine. Now imagine that the `abs` method taking a `BigDecimal` is not implemented! If `NUMBER1` is a `NUMERIC` with decimals, then the `abs` method taking a double would be called, which can potentially lose precision when the number is converted from a `BigDecimal` to a double.

Return Type Mapping

The return value of the method is mapped into an equivalent SQL type. Here is the type mapping table:

Return type of method	Blackfish SQL SQL type
`byte` or `Byte`	`SMALLINT`
`short` or `Short`	`SMALLINT`
`int` or `Integer`	`INT`
`long` or `Long`	`BIGINT`
`java.math.BigDecimal`	`DECIMAL`
`float` or `Float`	`REAL`
`double` or `Double`	`DOUBLE`

`String`	`VARCHAR`
`boolean` or `Boolean`	`BOOLEAN`
`java.io.InputStream` (Any type derived from `java.io.InputStream` is also handled as an `INPUTSTREAM`)	`INPUTSTREAM`
`java.sql.Date`	`DATE`
`java.sql.Time`	`TIME`
`java.sql.Timestamp`	`TIMESTAMP`
All other types:	`OBJECT`

Chapter 7:
Using Triggers in Blackfish SQL Tables

This chapter discusses triggers and Blackfish SQL tables.

About Triggers

You can create row level triggers for a Blackfish SQL table. In Blackfish SQL for Java, you can implement them in Java. In Blackfish SQL for Windows, you can implement them in Delphi, C#, or VB.NET.

All trigger methods must be declared static and have only one parameter of type `TriggerContext`. The `TriggerContext` class is symantically identical in Blackfish SQL for Java and Blackfish SQL for Windows. However, there are differences in syntax for the two different platforms. On the Windows platform, the class `TriggerContext` is in the `Borland.Data.DataStore` namespace. On the Java platform, the `TriggerContext` class is in the `com.borland.datastore` package.

The trigger context can be used to obtain access to:

- A connection object
- The new row for insert, and update triggers
- The old row for update and delete triggers

These rules apply:

- New row values are modified in a `BEFORE UPDATE` or `BEFORE INSERT` trigger.
- Old row values cannot be modified.
- Foreign key and primary key constraints are applied after `BEFORE` triggers.
- `AFTER` triggers are called after the operation has completed, including successful completion of foreign key and primary key constraints.
- The trigger implementation should avoid DML operations against the same table. This has the potential for infinite recursion.
- If there are multiple triggers of the same type for the same table, the calling order of the triggers is the order in which they were created.
- Commit and rollback operations are ignored inside the execution of the trigger If an exception is thrown from inside a trigger. The affects of the statment that caused the trigger to be executed will be

rolled back.

The requirements for deploying applications with trigger implementations are the same for both stored procedures and triggers.

Viewing Triggers

To view the triggers created for a table, call the `DB_ADMIN.GET_TRIGGERS` stored procedure. For a complete description see the Stored Procedures Reference.

Creating Triggers in Blackfish SQL for Windows Databases

Use the `CREATE TRIGGER` statement to create a trigger in a Blackfish SQL for Windows database. See the SQL Reference for syntax and examples of the `CREATE TRIGGER` statement.

Example

These example statements create triggers for the `Customer` class:

```
CREATE TRIGGER BEFORE_INSERT_CUSTOMER BEFORE INSERT ON CUSTOMER AS
  OrderEntryAssembly::OrderEntry.Customer.beforeInsertTrigger
CREATE TRIGGER BEFORE_UPDATE_CUSTOMER BEFORE UPDATE ON CUSTOMER AS
  OrderEntryAssembly::OrderEntry.Customer.beforeUpdateTrigger
CREATE TRIGGER BEFORE_DELETE_CUSTOMER BEFORE DELETE ON CUSTOMER AS
  OrderEntryAssembly::OrderEntry.Customer.beforeDeleteTrigger

CREATE TRIGGER AFTER_INSERT_CUSTOMER AFTER INSERT ON CUSTOMER AS
  OrderEntryAssembly::OrderEntry.Customer.afterInsertTrigger
CREATE TRIGGER AFTER_UPDATE_CUSTOMER AFTER UPDATE ON CUSTOMER AS
  OrderEntryAssembly::OrderEntry.Customer.afterUpdateTrigger
CREATE TRIGGER AFTER_DELETE_CUSTOMER AFTER DELETE ON CUSTOMER AS
OrderEntryAssembly::OrderEntry.Customer.afterDeleteTrigger
```

This shows the Delphi implementation of the `Customer` class triggers:

```
TCustomer = class
  class procedure BeforeInsertTrigger(Context: TriggerContext); static;
  class procedure BeforeUpdateTrigger(Context: TriggerContext); static;
  class procedure BeforeDeleteTrigger(Context: TriggerContext); static;
  class procedure AfterInsertTrigger(Context: TriggerContext); static;
  class procedure AfterUpdateTrigger(Context: TriggerContext); static;
  class procedure AfterDeleteTrigger(Context: TriggerContext); static;
end;

{ TCustomer }
  class procedure TCustomer.AfterDeleteTrigger(Context: TriggerContext);
    begin
      HandleBeforeInsert(Context.GetNewRow());
        end;
  class procedure TCustomer.AfterInsertTrigger(Context: TriggerContext);
    begin
      HandleBeforeUpdate(Context.GetOldRow(), Context.GetNewRow());
        end;
  class procedure TCustomer.AfterUpdateTrigger(Context: TriggerContext);
        begin
          HandleBeforeDelete(Context.GetOldRow(), Context.GetNewRow());
        end;
  class procedure TCustomer.BeforeDeleteTrigger(Context: TriggerContext);
        begin
          HandleAfterInsert(Context.getNewRow());
        end;
  class procedure TCustomer.BeforeInsertTrigger(Context: TriggerContext);
        begin
          HandleAfterUpdate(Context.getNewRow());
        end;
  class procedure TCustomer.BeforeUpdateTrigger(Context: TriggerContext);
        begin
          HandleAfterDelete(Context.getNewRow());
        end;
```

This shows the C# implementation of the `Customer` class triggers:

```
public class Customer {
  public static void BeforeInsertTrigger(TriggerContext Context)
  {
    HandleBeforeInsert(Context.GetNewRow());
  }
  public static void BeforeUpdateTrigger(TriggerContext Context)
  {
    HandleBeforeUpdate(Context.GetOldRow(), Context.GetNewRow());
  }
  public static void beforeDeleteTrigger(TriggerContext Context)
  {
    HandleBeforeDelete(Context.getNewRow());
  }
  public static void afterInsertTrigger(TriggerContext Context)
  {
    HandleAfterInsert(Context.getNewRow());
  }
  public static void afterUpdateTrigger(TriggerContext Context)
  {
    HandleAfterUpdate(Context.getNewRow());
  }
  public static void afterDeleteTrigger(TriggerContext Context)
  {
  HandleAfterDelete(Context.getNewRow());
  }
}
```

Creating Triggers in Blackfish SQL for Java Databases

Use the `CREATE TRIGGER` statement to create a trigger in a Blackfish SQL for Java database. See the SQL Reference for syntax and examples of the `CREATE TRIGGER` statement.

Examples:

These example statements create triggers for the `Customer` class:

```
CREATE TRIGGER BEFORE_INSERT_CUSTOMER BEFORE INSERT ON CUSTOMER AS
  orderentry.Customer.beforeInsertTrigger
CREATE TRIGGER BEFORE_UPDATE_CUSTOMER BEFORE UPDATE ON CUSTOMER AS
  orderentry.Customer.beforeUpdateTrigger
CREATE TRIGGER BEFORE_DELETE_CUSTOMER BEFORE DELETE ON CUSTOMER AS
  orderentry.Customer.beforeDeleteTrigger

CREATE TRIGGER AFTER_INSERT_CUSTOMER AFTER INSERT ON CUSTOMER AS
  orderentry.Customer.afterInsertTrigger
CREATE TRIGGER AFTER_UPDATE_CUSTOMER AFTER UPDATE ON CUSTOMER AS
  orderentry.Customer.afterUpdateTrigger
CREATE TRIGGER AFTER_DELETE_CUSTOMER AFTER DELETE ON CUSTOMER AS
  orderentry.Customer.afterDeleteTrigger
```

This shows the Java implementation of the `Customer` class triggers:

```
public class Customer {
  public static void beforeInsertTrigger(TriggerContext context)
    throws Exception
    {
      handleBeforeInsert(context.getNewRow());
    }
  public static void beforeUpdateTrigger(TriggerContext context)
    throws Exception
    {
      handleBeforeUpdate(context.getOldRow(), context.getNewRow());
    }
  public static void beforeDeleteTrigger(TriggerContext context)
    throws Exception
    {
      handleBeforeDelete(context.getOldRow(), context.getNewRow());
    }
  public static void afterInsertTrigger(TriggerContext context)
    {
      handleAfterInsert(context.getNewRow());
    }
  public static void afterUpdateTrigger(TriggerContext context)
    {
      handleAfterUpdate(context.getNewRow());
     }
  public static void afterDeleteTrigger(TriggerContext context)
    {
      handleAfterDelete(context.getNewRow());
    }
 }
```

Chapter 8:
Stored Procedures Reference

Many administrative tasks are supported by stored procedures in the DB_ADMIN class. The DB_UTIL class provides stored procedures for numeric, string, and date/time operations.

DB_ADMIN Stored Procedures

DB_ADMIN is a group of stored procedures for performing a variety of database administration tasks. For example:

- Viewing metadata
- Altering automatic failover and incremental backup
- Altering database properties
- Verifying tables
- Database copy for backup purposes
- Database encryption
- Change database password
- Displaying database staus, such as the following:
    - locks
    - status log IDs

These methods can be called from SQL using the CALL statement. They can be called without creating a METHOD alias because the Blackfish SQL recognizes the methods in DB_ADMIN as built-in methods.

DB_ADMIN Methods

The following sections provide the syntax and a brief description of each DB_ADMIN method.

ALTER_DATABASE

ALTER_DATABASE(*string properties*)

Alters specified database properties.

properties is a comma-separated list of settings for the columns from the DatabaseColumns class. Each property is specified as follows: <COLUMN_NAME>=<VALUE>.

See the online help for information on the DatabaseColumns class.

ALTER_MIRROR

ALTER_MIRROR(*string mirrorName, string properties*)

Alters an existing mirror configuration.

mirrorName is a value from the SysMirrors.NAME column.
properties is a comma-separated list of settings for the columns from the SysMirrors class.
Each property is specified as follows: <COLUMN_NAME>=<VALUE>.

See the online help for information on the SysMirrors class.

ALTER_MIRROR_SCHEDULE

ALTER_MIRROR_SCHEDULE(*INT32 mirrorId, string properties*)

Alters an existing mirror schedule item.

mirrorName is value from the SysMirrorSchedule.NAME column.
properties A comma separated list of settings for the columns from the SysMirrorSchedule class. Each property is specified as follows: <COLUMN_NAME>=<VALUE>.

See the online help for information on the SysMirrorSchedule class.

CHANGE_PASSWORD

CHANGE_PASSWORD(*string oldpassword, string newPassword*)

Changes the password for the currently connected user.

CLOSE_CONNECTION

CLOSE_CONNECTION(*INT32 connectionId, INT64 birthTimeMilliseconds*)

Closes an open connection. Can be used to close unwanted connections.

connectionId From the ID column returned by GET_CONNECTIONS.
birthTimeMilliseconds from the BIRTH column returned by GET_CONNECTIONS.

Returns true if successful

See GET_CONNECTIONS to obtain a table of connections to close.

CLOSE_OTHER_CONNECTIONS

CLOSE_OTHER_CONNECTIONS()

Closes all other open connections. Administrator rights are required to execute this method.

COPY_STREAMS

COPY_STREAMS(*string otherFilename, string adminUser, string adminPass, boolean doOverwrite, boolean doIgnoreErrors*)

Copies all tables and indexes from the current database to another database. COPY_USERS method should be called first if users have been added to the database.

otherFilename is the file name of the destination database.
adminUser is the user with ADMIN privileges in the destination database.
adminPass is the password of the ADMIN user in the destination database.
overwrite allows the tables to be overwritten. If false this will cause an error.
ignoreErrors causes errors to be ignored when recovering a corrupt database.

COPY_USERS

COPY_USERS(*string otherFilename, string adminUser, string adminPass, boolean doCopyEncryption, boolean replaceExistingUsers*)

Copies all users from the current database to another specified database.

otherFilename is the file name of the destination database.
adminUser user with ADMIN privileges in the destination database.
adminPass password of the ADMIN user in the destination database.
copyEncryption if the current database is encrypted then encrypt the target database with the same key.
replaceExistingUsers if true replace the existing users in the target database.

CREATE_MIRROR

CREATE_MIRROR(*string properties*)

Creates a new mirror with the configuration properties provided.

properties A comma-separated list of settings for the columns from the SysMirrors class. Each property is specified as follows: <COLUMN_NAME>=<VALUE>.

Returns a unique ID for new new mirror.

See the online help for information on the SysMirrors class.

CREATE_MIRROR_SCHEDULE

CREATE_MIRROR_SCHEDULE(*string mirrorName, string properties*)

Creates a new mirror synchronization schedule.

mirrorName the name of the mirror to add the mirror schedule item for.
properties a comma-separated list of settings for the columns in the SysMirrorSchedule table. Each property is defined by specified as follows: <COLUMN_NAME>=<VALUE>.

Returns a unique INT64 identifier for the new schedule item.

See the online help for information on the SysMirrorSchedule class.

DROP_MIRROR

DROP_MIRROR(*string mirrorName*)

Drops an existing mirror configuration.

mirrorName is a value from the SysMirrors.NAME column.

See the online help for information on the SysMirrors class.

DROP_MIRROR_SCHEDULE

DROP_MIRROR_SCHEDULE(*INT32 mirrorID*)

Drops an existing mirror schedule item.

mirrorID is a value from the SysMirrorSchedule.ID column

See the online help for information on the SysMirrorSchedule class.

ENCRYPT

ENCRYPT(*string adminPassword, string masterKeySeed*)

Encrypts an empty database.

adminPass password of the user performing this command.
masterKeySeed a random sequence of 16 characters that is used internally as the master password. Once provided, it does not needed to be provided for access to the database. This should be very random sequence of characters.

GET_ALL_LICENCES

GET_ALL_LICENCES( )

Returns a result table with zero or more rows of all licenses that could be found. The columns for this result table are defined in LicenseColumns class.

See the online help for information on the LicenseColumns class.

GET_COLUMN_PRIVILEGES

GET_COLUMN_PRIVILEGES(*string catalogPattern, string schemaPattern, string tablePattern, string columnPattern*)

catalogPattern specifies the LIKE catalog search pattern. Not used. Reserved for future use.
schemaPattern specifies the LIKE schema search patttern. null means means that the schema name should not be used to narrow the search.
tablePattern specifies the LIKE table search patttern. null means means that the table name should not be used to narrow the search.
columnPattern specifies the LIKE column search patttern. null means means that the column name should not be used to narrow the search.

Returns a result table with column privileges for the specified table(s). The columns for this result table are defined in the ColumnPrivilegeColumns class.

See the online help for information on the ColumnPrivilegeColumns class.

GET_COLUMNS

GET_COLUMNS(*string catalogPattern, string schemaPattern, string tablePattern, string columnPattern*)

catalogPattern specifies the LIKE catalog search pattern. Not used. Reserved for future use.
schemaPattern specifies the LIKE schema search patttern. null means means that the schema name should not be used to narrow the search.
tablePattern specifies the LIKE table search patttern. null means means that the table name should not be used to narrow the search.
columnPattern specifies the LIKE column search patttern. null means means that the column name should not be used to narrow the search.

Returns a result table with metadata for the columns of the specified table(s). The columns for this result table are defined in the ColumnsColumns class.

See the online help for information on the ColumnsColumns class.

GET_CONNECTIONS

GET_CONNECTIONS()

Returns a result table of the open connections for the current server connection. The columns for this result table are defined in the ConColumns class.

See the online help for information on the ConColumns class.

GET_DATABASE_PRIVILEGES

GET_DATABASE_PRIVILEGES(*boolean forRoles*)

forRoles if true, grantee is a role; if false, grantee is a user.

Returns a result table with database access rights for each user or role using the following columns:

1. GRANTOR String => grantor of access
2. GRANTEE String => grantee of access
3. PRIVILEGE String => name of access (STARTUP, ADMINISTRATOR, WRITE, CREATE, DROP, RENAME, CREATE_ROLES, CREATE_SCHEMAS)
4. IS_GRANTABLE String => YES if grantee is permitted to grant to others; NO if not

GET_DATABASE_PRODUCT_NAME

GET_DATABASE_PRODUCT_NAME()

Returns the product name of the server as a string.

GET_DATABASE_PRODUCT_VERSION

GET_DATABASE_PRODUCT_VERSION()

Returns the product version of the server as a string

GET_DATABASE_PROPS

GET_DATABASE_PROPS()

Returns a result table with the properties for the current database. The columns for this result table are defined in the DatabaseColumns class.

See the online help for information on the DatabaseColumns class.

GET_DATABASE_STATUS

GET_DATABASE_STATUS()

Returns a result table with one row of status information about the current database. The columns for this result table are defined in the `DatabaseStatusTable` class.

See the online help for information on the `DatabaseStatusTable` class.

GET_DATABASE_STATUS_LOG_FILTER

GET_DATABASE_STATUS_LOG_FILTER()

Returns an INT32 filter that controls what kind of logging information is logged to the status log file for all current database connections. The meaning of the bit masks is found in `LogFilterCodes` class.

See the online help for information on the `LogFilterCodes` class.

GET_DATATYPES

GET_DATATYPES()

Returns a result table with a row for each supported data type in the database server. The columns for this result table are defined in the `DataTypesColumns` class.

See the online help for information on the `DataTypesColumns` class.

GET_FOREIGN_KEYS

GET_FOREIGN_KEYS(*catalogPattern, schemaPattern, tablePattern*)

`catalogPattern` specifies the `LIKE` catalog search pattern. Not used. Reserved for future use.
`schemaPattern` specifies the `LIKE` schema search patttern. `null` means means that the schema name should not be used to narrow the search.
`tablePattern` specifies the `LIKE` table search patttern. `null` means means that the table name should not be used to narrow the search.

Returns a result table with a row for each foreign key in the specified table(s). The columns for this result table are defined in the `ForeignKeyColumnsColumns` class.

See the online help for information on the `ForeignKeyColumnsColumns` class.

GET_FOREIGN_KEY_COLUMNS

GET_FOREIGN_KEY_COLUMNS(*string catalogPattern,*

string schemaPattern, string tablePattern,
string foreignKeyPattern, string primaryCatalogPattern,
string primarySchemaPattern, string primaryTablePattern,
string primaryIndexPattern)

`catalogPattern` specifies the `LIKE` catalog search pattern. Not used. Reserved for future use.
`schemaPattern` specifies the `LIKE` schema search patttern. `null` means means that the schema name should not be used to narrow the search.
`tablePattern` specifies the `LIKE` table search patttern. `null` means means that the table name should not be used to narrow the search.
`foreignKeyPattern` specifies the `LIKE` foreign key search patttern. `null` means means that the table name should not be used to narrow the search.
`primaryCatalogPattern` specifies the `LIKE` primary catalog search pattern. Not used. Reserved for future use.
`primarySchemaPattern` specifies the `LIKE` primary schema search patttern. `null` means means that the primary schema name should not be used to narrow the search.
`primaryTablePattern` specifies the `LIKE` primary table search patttern. `null` means means that the table name should not be used to narrow the search.
`primaryIndexPattern` specifies the `LIKE` primary table index search patttern. `null` means means that the table name should not be used to narrow the search.

Returns a result table with a row for each foreign key column pairs in the specified table(s). The columns for this result table are defined in the `ForeignKeyColumns` class.

See the online help for information on the `ForeignKeyColumns` class.

GET_INDEXES

GET_INDEXES(*string catalogPattern, string schemaPattern,*
string tablePattern)

`catalogPattern` specifies the `LIKE` catalog search pattern. Not used. Reserved for future use.
`schemaPattern` specifies the `LIKE` schema search patttern. `null` means means that the schema name should not be used to narrow the search.
`tablePattern` specifies the `LIKE` table search patttern. `null` means means that the table name should not be used to narrow the search.

Returns a result table with the indexes of the specified table(s). The columns for this result table are defined in the `IndexesColumns` class.

See the online help for information on the `IndexesColumns` class.

GET_INDEX_COLUMNS

GET_INDEX_COLUMNS(*string catalogPattern, string*
schemaPattern, string tablePattern, string indexPattern)

catalogPattern specifies the LIKE catalog search pattern. Not used. Reserved for future use.
schemaPattern specifies the LIKE schema search patttern. null means means that the schema name should not be used to narrow the search.
tablePattern specifies the LIKE table search patttern. null means means that the table name should not be used to narrow the search.
indexPattern specifies the LIKE index search patttern. null means means that the table name should not be used to narrow the search.

Returns a result table with the column information of the specified table index(es). The columns for this result table are defined in the IndexColumnsColumns class.

See the online help for information on the IndexColumnsColumns class.

GET_KEYWORDS

```
GET_KEYWORDS()
```

Returns a result table with the reserved keywords in this database.

GET_LICENSE

```
GET_LICENSE()
```

Returns a result table with one row of license information for the best deployment license found. The columns for this result table are defined in the LicenseColumns class.

See the online help for information on the LicenseColumns class.

GET_LICENSE_SEARCH_DIRS

```
GET_LICENSE_SEARCH_DIRS()
```

Returns a result table with a row for each directory that is searched for license files.

GET_LOCKS

```
GET_LOCKS()
```

Returns a result table of all the currently held table and row locks for all connections to the current database. The columns for this result table are defined in the LockColumns class.

See the online help for information on the LockColumns class.

GET_MIRROR_ID

GET_MIRROR_ID()

Returns the INT64 mirror id of the current mirror if this is a mirror.

GET_MIRRORS

GET_MIRRORS(*mirrorName*, check*Status*)

name is the name of the mirror or null to get all mirrors.
checkStatus is set to TRUE to provide additional columns on the status of the mirror. Status checking requires more work to be performed, but provides additional information on a mirror.

Returns a result table with a row for each mirror. The columns for this result table are defined in the MirrorStatusColumns class.

See the online help for information on the MirrorStatusColumns class.

GET_NEWEST_STATUS_LOG_ID

GET_NEWEST_STATUS_LOG_ID()

Returns the INT32 ID of the newest log file that can be retrieved using the GET_STATUS_LOG() method.

GET_OLDEST_STATUS_LOG_ID

GET_OLDEST_STATUS_LOG_ID()

Returns the INT32 ID of the oldest log file that can be retrieved using the GET_STATUS_LOG() method.

GET_PROCEDURES

GET_PROCEDURES(*string catalogPattern, string schemaPattern, string procedurePattern, string type*)

catalogPattern specifies the LIKE catalog search pattern. Not used. Reserved for future use.
schemaPattern specifies the LIKE schema search patttern. null means means that the schema name should not be used to narrow the search.
procedurePattern specifies the LIKE procedure search patttern. null means means that the procedure name should not be used to narrow the search.
procedureType a procedure type; must be either PROCEDURE or FUNCTION or null for any procedure type.

Returns a result table with metadata for the known stored procedures. The columns for this result table are defined in the `ProcedureColumns` class.

See the online help for information on the `ProcedureColumns` class.

GET_PROCEDURE_COLUMNS

```
GET_PROCEDURE_COLUMNS(string catalogPattern,
string schemaPattern, string procedurePattern,
string parameterPattern)
```

`catalogPattern` specifies the `LIKE` catalog search pattern. Not used. Reserved for future use.
`schemaPattern` specifies the `LIKE` schema search patttern. `null` means that the schema name should not be used to narrow the search.
`procedureNamePattern` specifies the `LIKE` procedure search patttern. `null` means that the procedure name should not be used to narrow the search.
`columnNamePattern` specifies the `LIKE` column search patttern. `null` means that the column name should not be used to narrow the search.

Returns a result table with the parameters of the specified procedure(s). The columns for this result table are defined in the `ProcedureParametersColumns` class.

See the online help for information on the `ProcedureParametersColumns` class.

GET_PROCEDURE_PRIVILEGES

```
GET_PROCEDURE_PRIVILEGES()
```

Returns a result table with descriptions of the access rights for each procedure.

The result table has the following columns:

1. `PROCEDURE_CAT` String => procedure catalog (is always `null`)
2. `PROCEDURE_SCHEM` String => procedure schema
3. `PROCEDURE_NAME` String => procedure name
4. `GRANTOR` String => grantor of access
5. `GRANTEE` String => grantee of access
6. `PRIVILEGE` String => name of access (EXECUTE)
7. `IS_GRANTABLE` String => `YES` if grantee is permitted to grant to others; `NO` if not

GET_ROLES

```
GET_ROLES()
```

Returns a result table with the roles in the database. The columns for this result table are defined in the `RolesColumns` class.

See the online help for information on the `RolesColumns` class.

GET_ROLE_GRANTS

```
GET_ROLE_GRANTS(boolean forRoles)
```

forRoles set to `true`, grantee is a role; set to `false`, grantee is a user.

The result table has the following columns:

1. `ROLE_NAME` String => name of the role granted
2. `GRANTOR` String => grantor of role
3. `GRANTEE` String => grantee of role
4. `IS_GRANTABLE` String => `YES` if grantee is permitted to grant to others; `NO` if not.

GET_SCHEMAS

```
GET_SCHEMAS(string catalogPattern)
```

`catalogPattern` specifies the `LIKE` catalog search pattern. Not used. Reserved for future use.

Returns a result table with the schemas in the database.

GET_STATUS_LOG_FILTER

```
GET_STATUS_LOG_FILTER()
```

Returns the INT32 filter that controls the type of logging information to be logged to the status log for this connection. The meaning of the bit masks is found in `LogFilterCodes` class.

See the online help for information on the `LogFilterCodes` class.

GET_STATUS_LOG

```
GET_STATUS_LOG(INT32 log_id, INT64 offset)
```

`id` is the ID of the log file being retrieved.
`offset` is the offset into the log file from the start of the file.

Returns the status log for the current databas as a stream.

GET_STATUS_LOGS

GET_STATUS_LOGS()

Returns a result table with id of the existing logs for this database. The columns for this result table are defined in the StatusLogColumns class.

See the online help for information on the StatusLogColumns class.

GET_TABLE_PRIVILEGES

GET_TABLE_PRIVILEGES(*string catalogPattern, string schemaPattern, string tablePattern*)

catalogPattern specifies the LIKE catalog search pattern. Not used. Reserved for future use.
schemaPattern specifies the LIKE schema search patttern. null means means that the schema name should not be used to narrow the search.
tablePattern specifies the LIKE table search patttern. null means means that the table name should not be used to narrow the search.

Returns a result table privilege descriptions for the selected table(s). The columns for this result table are defined in the TablePrivilegeColumns class.

See the online help for information on the TablePrivilegeColumns class.

GET_TABLES

GET_TABLES(*string catalogPattern, string schemaPattern, string tablePattern, string type*)

catalogPattern specifies the LIKE catalog search pattern. Not used. Reserved for future use.
schemaPattern specifies the LIKE schema search patttern. null means means that the schema name should not be used to narrow the search.
tablePattern specifies the LIKE table search patttern. null means means that the table name should not be used to narrow the search.
types comma separated list of TABLE, VIEW, SYSTEM_TABLE or null.

Returns a result table with metadata for the specified table(s). The columns for this result table are defined in the TableColumns class.

See the online help for information on the TableColumns class.

GET_THIS_MIRROR

GET_THIS_MIRROR(*boolean checkStatus*)

checkStatus `TRUE` to provide additional columns on the status of the mirror.

Like `GET_MIRRORS` except that it returns information for only the mirror this procedure is executed against.

Returns a result table with a row for this mirror. The columns for this result table are defined in the `MirrorStatusColumns` class.

See the online help for information on the `MirrorStatusColumns` class.

GET_TRIGGERS

`GET_TRIGGERS(`*`string catalogPattern, string schemaPattern, string tablePattern, string trigger`*`)`

`catalogPattern` specifies the `LIKE` catalog search pattern. Not used. Reserved for future use.
`schemaPattern` specifies the `LIKE` schema search patttern. `null` means means that the schema name should not be used to narrow the search.
`tablePattern` specifies the `LIKE` table search patttern. `null` means means that the table name should not be used to narrow the search.
`triggerPattern` specifies the `LIKE` trigger search patttern. `null` means means that the trigger name should not be used to narrow the search.

Returns a result table of the triggers of the specified table(s).

The result table has the following columns:

1. `TRIGGER_CAT` String => trigger catalog (is always `null`)
2. `TRIGGER_SCHEM` String => trigger schema
3. `TRIGGER_TABLE` String => trigger table
4. `TRIGGER_TYPE` String => trigger type
5. `TRIGGER_METHOD` String => name of the trigger method

GET_USERS

`GET_USERS()`

Returns a result table with the users in the database.

GET_VIEWS

`GET_VIEWS(`*`string catalogPattern, string schemaPattern, string view`*`)`

`catalogPattern` specifies the `LIKE` catalog search pattern. Not used. Reserved for future use.
`schemaPattern` specifies the `LIKE` schema search patttern. `null` means means that the schema name should not be used to narrow the search.
`viewNamePattern` specifies the `LIKE` view search patttern. `null` means that the view name should not be used to narrow the search.

Returns a result table with the definitions of the specified view(s). The columns for this result table are defined in the `ViewsColumns` class.

See the online help for information on the `ViewsColumns` class.

SET_DATABASE_STATUS_LOG_FILTER

`SET_DATABASE_STATUS_LOG_FILTER(`*`INT32 filter`*`)`

Sets the filter that controls the type of logging information to be entered in the status log file for all current database connections.

SET_PRIMARY_MIRROR

`SET_PRIMARY_MIRROR(`*`INT64 txTerminationTimeout, boolean forceTransactionAbort, boolean forceSwitch`*`)`

Sets the current mirror to the primary mirror.
`txTerminationTimeout` is milliseconds to wait for existing transactions to terminate.
`forceTxTermination` causes existing transactions to abort after `txTerminationTimeout` milliseconds have elapsed.
`forceSwitch` causes this mirror to become the primary mirror even if other mirrors could not be synchronized with this change.

SET_STATUS_LOG_FILTER

`SET_STATUS_LOG_FILTER(`*`INT32 filter`*`)`

Sets the filter that controls the type of logging information to be logged to the status log file for the current connection.

SYNCH_MIRROR

`SYNCH_MIRROR(`*`string mirrorName`*`)`

Updates the mirror specified for `mirrorName` with the most recent log files of its update mirror if necessary.

VALIDATE_PRIMARY_MIRROR

VALIDATE_PRIMARY_MIRROR()

Validates a primary mirror so that write transactions can be performed against it.

VERIFY

VERIFY(*string catalogPattern, string schemaPattern, string tablePattern, INT32 displayOptions, INT32 errorCount, out INT32 errorsEncountered, out String output*)

Verifies one or more tables in the database.

catalogPattern specifies the LIKE catalog search pattern. Not used. Reserved for future use.
schemaPattern specifies the LIKE schema search patttern. null means means that the schema name should not be used to narrow the search.
tablePattern specifies the LIKE table search patttern. null means means that the table name should not be used to narrow the search.
displayOptions one or more of the StreamVerifierDisplay bit settings ORed together that instruct the verifier to display progress messages as the stream is verified.
errorCount specifies the number of errors that can be ignored before an exception is thrown.
errorsEncountered is an output parameter that returns the number of errors that were encountered.
output is an output parameter that can be used to return a string with diagnostic output from the verifier.

VERIFY

VERIFY(*string tablePattern, INT32 displayOptions, INT32 errorCount, out INT32 errorsEncountered, out String output*)

Verifies one or more tables in the database.

tablePattern specifies the LIKE table search patttern. null means means that the table name should not be used to narrow the search.
displayOptions one or more of the StreamVerifierDisplay bit settings ORed together that instruct the verifier to display progress messages as the stream is verified.
errorCount specifies the number of errors that can be ignored before an exception is thrown.
errorsEncountered is an output parameter that returns the number of errors that were encountered.
output is an output parameter that can be used to return a string with diagnostic output from the verifier.

DB_UTIL: Numeric, String, and Date/Time Functions

DB_UTIL is a set of SQL stored procedures for performing numeric, string and date/time operations on data stored in database tables. These functions are implemented as Java UDFs in DB_UTIL.

Examples

The following statement computes the square root of the column COL1:

```
SELECT DB_UTIL.SQRT(COL1) FROM TABLE1;
```

The following statement computes some timestamps that are equal to the timestamp COL2 plus five hours.

```
SELECT  DB_UTIL.TIMESTAMPADD('SQL_TSI_HOUR',5,COL2) FROM TABLE1;
```

Numeric Functions

ACOS

ACOS(*expression*)

Returns the arccosine in radians of a number.

ASIN

ASIN(*expression*)

Returns the arcsine in radians of a number.

ATAN

ATAN(*expression*)

Returns the arctangent in radians of a number.

ATAN2

ATAN2(*y, x*)

Returns the arctangent of the quotient of its two arguments. The angle returned is a numeric value in radians between PI and -PI and represents the counterclockwise angle between the positive X axis and the point (*x*, *y*). Note that the *y* value is passed in first.

CEILING

CEILING(*expression*)

Returns the smallest integer that is greater than or equal to the argument. The return is of the same data type as the input.

COS

COS(*expression*)

Returns the cosine of an angle.

COT

COT(*expression*)

Returns the cotangent of an angle.

DEGREES

DEGREES(*expression*)

Converts an angle in radians to degrees.

EXP

EXP(*expression*)

Returns the exponential value of *expression*.

FLOOR

FLOOR(*expression*)

Returns the largest integer that is equal to or less than *expression*. The return is of the same data type as the input.

LOG

LOG(*expression*)

Returns the natural logarithm of a number.

LOG10

LOG10(*expression*)

Returns the base 10 logarithm of a number.

MOD

MOD(*expression1, expression2*)

Returns the remainder for *expression* divided by *expression*, where both expressions evaluate to integers of type SHORT, INTs or LONGs. The return is of the same data type as the input.

PI

PI()

Returns the constant PI.

POWER

POWER(*expression1, expression2*)

Returns the value of *expression1* raised to the power of *expression2*.

RADIANS

RADIANS(*expression*)

Converts an angle in degrees to radians.

RAND

RAND()

Generates a random floating point number.

RAND

RAND(*expression*)

Generates a random floating point number using *expression* as a seed integer.

ROUND

ROUND(*expression1, expression2*)

Rounds *expression1* to *expression2* number of decimal places.

SIGN

SIGN(*expression*)

Returns –l if the value of *expression* is negative, zero if *expression* is zero, and 1 if *expression* is positive. The return is of the same data type as the input.

SIN

SIN(*expression*)

Returns the sine in radians of an angle.

SQRT

SQRT(*expression*)

Returns the square root of a number.

TAN

TAN(*expression*)

Returns the tangent of an angle given in radians.

TRUNCATE

TRUNCATE(*expression1, expression2*)

Truncates the value of *expression1* to *expression2* decimal places.

String Functions

ASCII

ASCII(*string*)

Returns an integer representing the ASCII code value of the leftmost character in `string`.

TO_CHAR

TO_CHAR(*ascii_code*)

Returns the *char* equivalent of the ASCII code argument.

DIFFERENCE

DIFFERENCE(*string1, string2*)

Returns an integer in the range 0 through 4 indicating how many of the four digits returned by the function SOUNDEX for *string1* are the same as those returned for *string2*. A return value of 4 indicates that the SOUNDEX codes are identical.

INSERT_STRING

INSERT_STRING(*string1, start, length, string2*)

Returns a character string formed by deleting *length* characters from *string1* beginning at *start* and then inserting *string2* into *string1* at *start*.

LEFT_STRING

LEFT_STRING(*string, count*)

Returns the leftmost *count* characters from *string*.

REPEAT

REPEAT(*string, count*)

A character string formed by repeating *string* string *count* times.

REPLACE

REPLACE (*string1*, *string2*, *string3*)

Returns a character string formed by replacing all occurrences of *string2* in *string1* with *string3*.

RIGHT

RIGHT_STRING(*string*, *count*)

Returns a string formed by taking the right-hand *count* characters from *string*.

SOUNDEX

SOUNDEX (*string*)

Returns a string that represents the sound of the words in string; the return is data source-dependent and could be a four-digit SOUNDEX code, a phonetic representation of each word, or some other form.

SPACE

SPACE(*count*)

Returns a character string consisting of *count* spaces.

Date and Time Functions

DAYNAME

DAYNAME(*date*)

Returns the day of the week as a string from the given date.

DAYOFWEEK

DAYOFWEEK(*date*)

Returns the day of the week as as a number: 1=Sunday, 7=Saturday.

DAYOFYEAR

DAYOFYEAR(*date*)

Returns the day of the year as a number: 1=January 1.

MONTHNAME

MONTHNAME(*date*)

Returns a string representing the month component of the given date.

QUARTER

QUARTER(*date*)

Returns the quarter as a number from the given date: 1=January through March, 2=April through June.

TIMESTAMPADD

TIMESTAMPADD(*interval, count, timestamp*)

Returns a timestamp calculated by adding *count* number of *interval*s to *timestamp*.

interval can be any one of the following and must be enclosed in single quotes: SQL_TSI_FRAC_SECOND, SQL_TSI_SECOND, SQL_TSI_MINUTE, SQL_TSI_HOUR, SQL_TSI_DAY, SQL_TSI_WEEK, SQL_TSI_MONTH, SQL_TSI_QUARTER, or SQL_TSI_YEAR.

timestamp can be any of the following SQL data types: DATE, TIME, TIMESTAMP.

TIMESTAMPDIFF

TIMESTAMPDIFF(*interval, timestamp1, timestamp2*)

Returns a number representing the number of intervals by which *timestamp2* is greater than *timestamp1*.

interval can be any one of the following and must be enclosed in single quotes: SQL_TSI_FRAC_SECOND, SQL_TSI_SECOND, SQL_TSI_MINUTE, SQL_TSI_HOUR, SQL_TSI_DAY, SQL_TSI_WEEK, SQL_TSI_MONTH, SQL_TSI_QUARTER, or SQL_TSI_YEAR.

timestamp1 and *timestamp2* can be any of the following SQL data types: DATE, TIME, TIMESTAMP.

WEEK

WEEK(*date*)

Returns an integer from 1 to 53 representing the week of the year in *date*. 1=the first week of the year.

Chapter 9:
SQL Reference

The SQL Reference includes the following topics:

Data Types

In SQL, you can specify data types by using Blackfish SQL names or by using synonyms, which are more portable to other SQL dialects. The following table lists the Blackfish SQL SQL data types and their Java equivalents. See Administering Blackfish SQL for a description of each data type.

Strings are stored in UNICODE character format. However, if a string contains no high-bit characters. the high bytes are not saved and the number of bytes is equal to the number of characters. In double-byte languages such as Japanese, the number of bytes is double the number of characters.

NOTE: The word “inline” refers to the portion of the field data that is stored in the table row. When the maximum inline value is surpassed, the remaining data is stored in a separate stream as a Blob.

SQL Data Types Supported by Blackfish SQL

The following table describes the SQL data types supported by Blackfish SQL:

Data Type	SQL Equivalents [1]
8 bit byte	**TINYINT** BYTE

16 bit integer	**SMALLINT** SHORT
32 bit integer	**INT** INTEGER
64 bit integer	**BIGINT** LONG
Exact decimal number	**DECIMAL(*p*,*d*)** BIGDECIMAL(*p*,*d*)
64 bit floating point	FLOAT(*p*), *p*=24 through 52 FLOAT DOUBLE **DOUBLE PRECISION**
32 bit floating point	**REAL** FLOAT(*p*), *p*=1 through 23
Unicode string	**VARCHAR(*p*,*m*)** STRING(*p*,*m*)
Array of bytes	**VARBINARY(*p*,*m*)** BINARY(*p*,*m*) INPUTSTREAM(*p*,*m*)
Serializable object	OBJECT(*t*,*m*)
Boolean	**BOOLEAN** BIT
Date	**DATE**
Time	**TIME**
Timestamp	**TIMESTAMP**

[1] In the SQL Equivalents column, **bold** indicates the more portable forms.

Examples

VARCHAR(30,10)	A string with a maximum size of 30 characters; the first 10 bytes are stored inline, the remainder in a Blob (a separate stream for large objects)
VARCHAR(30)	A string with a maximum size of 30 characters, all stored inline because the precision is less than default inline value of 64
VARCHAR	A string with no length limit; the first 64 bytes are stored inline, any additional bytes are stored in a Blob (a separate stream for large objects)
DECIMAL(5,2)	A `BigDecimal` with a precision of at least 5 and exactly 2 decimal places

`DECIMAL(4)`	A `BigDecimal` with a precision of at least 4 and exactly 0 decimal places
`DECIMAL`	A `BigDecimal` with space for at least 72 significant digits and exactly 0 decimal places
`OBJECT`	A serializable Java object
`OBJECT('java.math.BigInteger')`	A serializable Java object that must consist of `java.math.BigInteger` objects

Literals

The following table lists the types of scalar literal values supported:

Blackfish SQL Data Type	Examples	Description

`SMALLINT`
`INT`
`BIGINT`
	`8`	Integer data types

`DECIMAL(p,d)`	`2.`
	`15.7`
	`.9233`
		An exact numeric; can contain a decimal point

`REAL`
`DOUBLE`
`FLOAT(p)`
	`8E0`
	`4E3`
	`0.3E2`
	`6.2E-72`
		An approximate numeric: a number followed by the letter `E`, followed by an optionally signed integer

`VARCHAR(p,m)`	`'Hello'` `'don''t do that'`	A string: must be enclosed in single quotes. The single quote character is represented by two consecutive single quotes

`VARBINARY(p,m)`	`B'1011001'`
	`X'F08A'`
	`X'f777'`
		A binary or hexadecimal sequence enclosed in single quotes and preceded by the letter `B` for binary or `X` for hexadecimal

`BOOLEAN`	`TRUE` `FALSE`	
`DATE`	`DATE '2002-06-17'`	Displays local time of origin; format is `DATE 'yyyy-mm-dd'`
`TIME`	`TIME '15:46:55'`	Displays local time of origin; format is `TIME 'hh:mm:ss'` in 24-hour format
`TIMESTAMP`	`TIMESTAMP '2001-12-31 13:15:45'`	Displays local time of display; format is `TIMESTAMP 'yyyy-mm-dd hh:mm:ss'`

NOTE: There are no object literals in Blackfish SQL.

Keywords

The two lists below show all the current keywords for Blackfish SQL. The words in the first list are *reserved* and can be used as SQL identifiers only when enclosed in double quotation marks. The keywords in the second list are not reserved and can be used either with or without quotation marks.

Note that not all SQL-92 keywords are treated as a keyword by the Blackfish SQL SQL engine.
For maximum portability, don’t use identifiers that are treated as keywords in any SQL dialect.

Reserved Blackfish SQL Keywords

The words in this list are *reserved keywords*. They can be used as SQL identifiers only if they are enclosed in double quotation marks. When quoted in this fashion, they are case sensitive.

ABSOLUTE	CURRENT_ROLE	INTO	RESTRICT
ACTION	CURRENT_TIME	IS	REVOKE
ADD	CURRENT_TIMESTAMP	ISOLATION	RIGHT
ADMIN	CURRENT_USER	JOIN	SCHEMA
ADMINISTRATOR	DATE	KEY	SELECT
ALL	DECIMAL	LEADING	SET
ALTER	DEFAULT	LEFT	SMALLINT
AND	DELETE	LEVEL	SOME
ANY	DESC	LIKE	SQRT
AS	DISTINCT	LOWER	STARTUP
ASC	DOUBLE	MAX	SUBSTRING
AUTHORIZATION	DROP	MIN	SUM
AUTOINCREMENT	ELSE	NATURAL	TABLE
AVG	END	NO	THEN
BETWEEN	ESCAPE	NONE	TIME
BIT	EXCEPT	NOT	TIMESTAMP
BIT_LENGTH	EXECUTE	NULL	TO
BOTH	EXISTS	NULLIF	TRAILING
BY	EXTRACT	NUMERIC	TRANSACTION
CALL	FALSE	OCTET_LENGTH	TRIM
CASCADE	FLOAT	ON	TRUE
CASE	FOR	ONLY	UNION
CAST	FOREIGN	OPTION	UNIQUE
CHAR	FROM	OR	UNKNOWN
CHAR_LENGTH	FULL	ORDER	UPDATE

CHARACTER	GRANT	OUTER	UPPER
CHARACTER_LENGTH	GROUP	POSITION	USER
CHECK	HAVING	PRECISION	USING
COALESCE	IN	PRIMARY	VALUES
COLUMN	INDEX	PRIVILEGES	VARCHAR
CONSTRAINT	INNER	PUBLIC	VARYING
COUNT	INSERT	REAL	VIEW
CREATE	INT	REFERENCES	WHEN
CROSS	INTEGER	RENAME	WHERE
CURRENT_DATE	INTERSECT	RESOLVABLE	WITH

Unreserved Blackfish SQL Keywords

The keywords in the following list are not reserved. They can be used as SQL identifiers either with or without quotation marks. When used without quotation marks, they are case insensitive and are interpreted as all caps by the SQL parser. When enclosed in double quotation marks, they are case sensitive.

ABS	DAYOFMONTH	METHOD	SHORT
AUTOCOMMIT	DEC	MINUTE	STRING
BOOLEAN	FN	MONTH	T
BIGDECIMAL	GRANTED	NOW	TIMESTAMPADD
BIGINT	HOUR	NOWAIT	TIMESTAMPDIFF
BINARY	IFNULL	OBJECT	TIMEZONEHOUR
BYTE	INPUTSTREAM	OFF	TIMEZONEMINUTE
CASEINSENSITIVE	METHOD	OJ	TINYINT
CLASS	LCASE	PASSWORD	TS
COMMIT	LENGTH	READ	TYPE
COMMITTED	LOCATE	REPEATABLE	UCASE
CONCAT	LOCK	ROLE	UNCOMMITTED
CONVERT	LONG	ROLLBACK	VARBINARY
CURDATE	LONGINT	RTRIM	WORK
CURTIME	LONGVARBINARY	SECOND	WRITE
D	LONGVARCHAR	SERIALIZABLE	YEAR
DAY	LTRIM		

Identifiers

Unquoted SQL identifiers are case insensitive and are treated as uppercase. An identifier can be enclosed in double quotes, and is then treated as case sensitive. An unquoted identifier must follow these rules:

- The first character must be a letter recognized by the `java.lang.Character` class.
- Each following character must be a letter, digit, underscore (_), or dollar sign ($).
- Keywords can't be used as identifiers.

Quoted identifiers can contain any character string including spaces, symbols, and keywords.

Examples

Valid identifiers:

Identifier	Description
`customer`	Treated as `CUSTOMER`
`Help_me`	Treated as `HELP_ME`
`"Hansen"`	Treated as `Hansen`
`" "`	Treated as a single space

Invalid identifiers:

Identifier	Problem
`_order`	Must start with a character
`date`	`date` is a reserved keyword
`borland.com`	Dots are not allowed

The forms in the following list are all the same identifier and are all treated as `LAST_NAME`:

- `last_name`
- `Last_Name`
- `lAsT_nAmE`
- `"LAST_NAME"`

List Syntax

The following section contains element names ending with the words “list” or “commalist” that are not further defined. For example:

```
<select item commalist>
<column constraint list>
```

These definitions are to be read as a lists with at least one element, comma separated in the case of a commalist:

```
<select item commalist> ::=
   <select item> [ , <select item> ] *
<column constraint list> ::=
   <column constraint> [ <column constraint> ] *
```

Expressions

Expressions are used throughout the SQL language. They contain several infix operators and a few prefix operators. This is the operator precedence from strongest to weakest:

- prefix `+ -`
- infix `* /`
- infix `+ - ||`
- infix `= <> < > <= >=`
- prefix `NOT`
- infix `AND`
- infix `OR`

Syntax

```
<expression> ::=
      <scalar expression>
    | <conditional expression>
```

```
<scalar expression> ::=
      <scalar expression> {+ | - | * | / | <concat> }
          <scalar expression>
    | {+ | -} <scalar expression>
    | ( <expression> )
    | ( <table expression> )
    | <column reference>
    | <user defined function reference>
    | <literal>
    | <aggregator function>
    | <function>
    | <parameter marker>
```

For a list of functions supported in Blackfish SQL, see Functions.

```
<conditional expression> ::=
      <conditional expression> OR <conditional expression>
    | <conditional expression> AND <conditional expression>
    | NOT <conditional expression>
    | <scalar expression> <compare operator> <scalar expression>
    | <scalar expression> <compare operator> { ANY | SOME | ALL }
        (<table expression>)
    | <scalar expression> [NOT] BETWEEN <scalar expression>
    | <scalar expression> [NOT] LIKE <scalar expression>
        [ ESCAPE <scalar expression> ]
    | <scalar expression> [NOT] IS { NULL | TRUE | FALSE | UNKNOWN }
    | <scalar expression> IN ( <scalar expression commalist> )
    | <scalar expression> IN ( <table expression> )
    | EXISTS ( <table expression> )
```

```
<compare operator> ::=
      = | <> | < | > | <= | >=
```

```
<concat> ::=   ||
```

```
<table expression> ::=
      <table expression> UNION [ ALL ] <table expression>
    | <table expression> EXCEPT [ ALL ] <table expression>
    | <table expression> INTERSECT [ ALL ] <table expression>
    | <join expression>
    | <select expression>
    | ( <table expression> )
```

```
<aggregator function> ::=
      <aggregator name> ( <expression> )
    | COUNT ( * )

<aggregator name> ::=
      AVG
    | SUM
    | MIN
    | MAX
    | COUNT
```

```
<column reference> ::=  [ <table qualifier> . ] <column name>
<user defined function reference> ::=
    <method name> ([ <expression commalist> ])

<table qualifier> ::=
      <table name> | <correlation name>

<correlation name> ::=  <SQL identifier>
```

Examples

The following statement selects the calculated value of `Amount` times `Price` from the `Orders` table for a to-be-provided customer for orders in January:

```
SELECT Amount * Price FROM Orders
    WHERE CustId = ? AND EXTRACT(MONTH FROM Ordered) = 1;
```

The following statement gets data using a scalar subquery:

```
SELECT Name, (SELECT JobName FROM Job WHERE Id=Person.JobId)
    FROM Person;
```

Note that it is an error if the subquery returns more than one row.

Predicates

The following predicates, used in condition expressions, are supported.

BETWEEN

The BETWEEN predicate defines an inclusive range of values. The result of:

```
expr BETWEEN leftExpr AND rightExpr
```

is equivalent to the expression:

```
leftExpr <= expr AND expr <= rightExpr
```

Syntax

```
<between expression> ::=
     <scalar expression> [NOT] BETWEEN <scalar expression>
       AND <scalar expression>
```

Example

The following statement selects all the orders where a customer orders between 3 and 7 items of the same kind:

```
SELECT * from Orders WHERE Amount BETWEEN 3 AND 7;
```

EXISTS

An EXISTS expression evaluates to either TRUE or FALSE depending on whether there are any elements in a result table.

Syntax

```
<exists predicate> ::= EXISTS ( <table expression> )
```

Example

The following statement finds all diving equipment where the beginning of the name is the same as the beginning of a name of a different piece of equipment.

```
SELECT * FROM zodiac z
   WHERE EXISTS
      ( SELECT * FROM zodiac z2 WHERE POSITION(z.name IN z2.name) = 1
          AND z.name < > z2.name );
```

IN

The IN clause indicates a list of values to be matched. Any one of the values in the list is considered a match for the SELECT statement containing the IN clause.

Syntax

```
<in expression> ::=
    <scalar expression> IN ( <scalar expression commalist> )
```

Example

The following statement returns all records where the name column matches either "leo" or "aquarius":

```
SELECT * FROM zodiac WHERE name IN ('leo', 'aquarius');
```

The IN clause also has a variant where a subquery is used instead of an expression list.

Syntax

```
<in expression> ::= <scalar expression>
   IN ( <table expression> )
```

Example

```
SELECT * FROM zodiac WHERE name IN (SELECT name FROM people);
```

IS

The IS predicate tests expressions. Any expression can evaluate to the value NULL, but conditional expressions can evaluate to one of the three values: TRUE, FALSE, or UNKNOWN. UNKNOWN is equivalent to NULL for conditional expressions. Note that for a SELECT query with a WHERE clause, only rows that evaluate to TRUE are included. If the expression evaluates to FALSE or UNKNOWN, the row isn't included. The output of the IS predicate can have two results: TRUE or FALSE.

Syntax

```
<is expression> ::=
      <scalar expression> IS [NOT] { NULL | TRUE | FALSE | UNKNOWN }
```

Examples

TRUE IS TRUE evaluates to TRUE.

FALSE IS NULL evaluates to FALSE.

LIKE

The LIKE predicate provides SQL with simple string pattern matching. The search item, pattern, and escape character (if given) must all evaluate to strings. The pattern can include the special wildcard characters _ and % where:

- An underscore (_) matches any single character
- A percent character (%) matches any sequence of *n* characters where *n* >= 0

The escape character, if given, allows the two special wildcard characters to be included in the search pattern. The pattern match is case-sensitive. Use the LOWER or UPPER functions on the search item for a case-insensitive match.

Syntax

```
<like expression> ::=
      <search item> [NOT] LIKE <pattern> [ ESCAPE <escape char> ]

<search item>  ::= <scalar expression>

<pattern>  ::= <scalar expression>

<escape char>  ::= <scalar expression>
```

Examples

1. The following expression evaluates to TRUE if Item contains the string "shoe" anywhere inside it:

   ```
   Item LIKE '%shoe%'
   ```

2. The following expression evaluates to TRUE if Item is exactly three characters long and starts with the letter "S":

   ```
   Item LIKE 'S__'
   ```

3. The following expression evaluates to TRUE if Item ends with the percent character. The * is defined to escape the two special characters. If it precedes a special character, it is treated as a normal character in the pattern:

   ```
   Item Like '%*%' ESCAPE '*'
   ```

Quantified Comparisons

An expression can be compared to some or all elements of a result table.

Syntax

```
<quantified comparison> ::=
     <scalar expression> <compare operator>
        { ANY | SOME | ALL } ( <table expression> )
```

Example

```
SELECT * FROM zodiac
    WHERE quantify <= ALL ( SELECT quantify FROM zodiac );
```

Functions

Functions that act on strings work for strings of any length. Large strings are stored as Blobs, so you might want to define large text fields as VARCHAR to enable searches.

ABSOLUTE

The ABSOLUTE function works on numeric expressions only, and yields the absolute value of the number passed.

Syntax

```
<absolute function> ::= ABSOLUTE( <expression> )
```

Example

```
SELECT * FROM Scapes WHERE ABSOLUTE( Height - Width ) < 50;
```

BIT_LENGTH

The BIT_LENGTH function gives the length in bits of a STRING, INPUTSTREAM, or OBJECT value.

Syntax

```
<bit length function> ::=
     BIT_LENGTH( <expression> )
```

Example

```
SELECT * FROM TABLE1 WHERE BIT_LENGTH( binary_column ) > 8192;
```

CASE

The CASE function returns a conditional value.

Syntax

```
<case function> ::=
    CASE  [ <expression> ]
        <when clause commalist>
        ELSE  <expression>
    END
<when clause> ::=
    WHEN <expression>  THEN <expression>
```

Examples

```
CASE
   WHEN COL1 > 50 THEN 'Heavy Item'
   WHEN COL1 > 25 THEN 'Middle weight Item'
   WHEN COL1 >  0  THEN 'Light Item'
   ELSE  'No weight specified'
END

CASE COL2
   WHEN 4 THEN 'A'
   WHEN 3 THEN 'B'
   WHEN 2 THEN 'C'
   WHEN 1 THEN 'D'
   ELSE 'Invalid Grade'
END
```

CAST

The CAST function casts one data type to another data type.

Syntax

```
<cast function> ::=
    CAST ( <column name> AS <data type> )
```

Example

The following example yields a row where a string column ID equals '001234'

```
SELECT * FROM employee WHERE CAST ( id AS long ) = 1234;
```

CHAR_LENGTH and CHARACTER_LENGTH

The SQL CHAR_LENGTH and CHARACTER_LENGTH functions yield the length of the given string.

Syntax

```
<char length function> ::=
      CHAR_LENGTH ( <scalar expression> )
      CHARACTER_LENGTH ( <scalar expression> )
```

COALESCE

The COALESCE function returns the first non-NULL value from the expression list.

Syntax

```
<coalesce function> ::=
    COALESCE( expression commalist )
```

Example

The following statement yields a list of names. The name is the last_name if this column is not NULL, otherwise it is the first_name.

```
SELECT COALESCE(last_name, first_name) AS name FROM  table1;
```

CURRENT_DATE, CURRENT_TIME, and CURRENT_TIMESTAMP

These SQL functions yield the current date and/or time. If one of these functions occurs more than once in a statement, it yields the same result each time when the statement is executed.

Example

```
SELECT * from Returns where ReturnDate <= CURRENT_DATE;
```

CURRENT_ROLE

The CURRENT_ROLE function returns the current role, or NULL if no role has been set using the SET ROLE statement.

Syntax

```
<current_role_function> ::= CURRENT_ROLE
```

Example

The following statement returns all notes from the CUSTOMERS table that were placed there by anyone using the MANAGER role. The SOURCE column has a data type of VARCHAR.

```
SET ROLE MANAGER;

SELECT * FROM CUSTOMERS
    WHERE SOURCE = CURRENT_ROLE;
```

CURRENT_USER

The CURRENT_USER function returns the name of the current user.

Syntax

```
<current_user function> ::= CURRENT_USER
```

Example

The following statement returns returns all notes from the INVOICES table that were placed there by the current user. The SOURCE column has a data type of VARCHAR.

```
SELECT * FROM INVOICES
    WHERE SOURCE = CURRENT_USER;
```

EXTRACT

The SQL EXTRACT function extracts parts of date and time values. The expression can be a DATE, TIME, or TIMESTAMP value.

Syntax

```
<extract function> ::=
      EXTRACT ( <extract field> FROM <scalar expression> )

<extract field> ::=
      YEAR
    | MONTH
    | DAY
    | HOUR
    | MINUTE
    | SECOND
```

Examples

`EXTRACT(MONTH FROM DATE '1999-05-17')` yields 5.

`EXTRACT(HOUR FROM TIME '18:00:00')` yields 18.

`EXTRACT(HOUR FROM DATE '1999-05-17')` yields an exception.

LOWER and UPPER

The SQL `LOWER` and `UPPER` functions convert the given string to the requested case, either all lowercase or all uppercase.

Syntax

```
<lower function> ::=
      LOWER ( <scalar expression> )

<upper function> ::=
      UPPER ( <scalar expression> )
```

NULLIF

The `NULLIF` function compares two expressions. It returns NULL if the expressions are equal. Otherwise, it returns the first expression. It is logically equivalent to the following `CASE` expression: `CASE WHEN expr1 = expr2 THEN NULL ELSE expr1 END`.

Syntax

```
<NULLIF> ::=
    ( <scalar expression>, <scalar expression> )
```

Example

The following statement returns a row with the `last_name` value for each row in `TABLE1` where the first name is not the same as the last name. If the *first_name* value is the same as the *last_name* value, it returns NULL.

```
SELECT NULLIF(last_name,first_name) FROM TABLE1;
```

OCTET_LENGTH

The `OCTET_LENGTH` function gives the length in bytes of a `STRING`, `INPUTSTREAM`, or `OBJECT` value.

Syntax

```
<octet_length> ::= OCTET_LENGTH(<expression>)
```

Example

```
SELECT * FROM TABLE1 WHERE OCTET_LENGTH(binary_column)>1024;
```

POSITION

The SQL `POSITION` function returns the position of a string within another string. If any of the arguments evaluate to `NULL`, the result is `NULL`.

Syntax

```
<position function> ::=
      POSITION ( <string> IN <another> )
```

Examples

```
POSITION('BCD' IN 'ABCDEFG') yields 2.
```

```
 POSITION('' IN 'ABCDEFG') yields 1.
```

```
 POSITION('TAG' IN 'ABCDEFG') yields 0.
```

SQRT

The SQRT function works on numeric expressions only, and yields the square root of the number passed.

Syntax

```
<sqrt function>  ::=  SQRT( <expression> )
```

Example

```
SELECT * FROM Scapes WHERE SQRT(HEIGHT*WIDTH - ?) > ?;
```

SUBSTRING

The SQL SUBSTRING function extracts a substring from a given string. If any of the operands are NULL, the result is NULL. The `start` position indicates the first character position of the substring, where 1 indicates the first character. If FOR is used, it indicates the length of the resulting string.

Syntax

```
<substring function> ::=
    SUBSTRING ( <string expression>
        FROM <start pos> [ FOR <length> ] )
```

Examples

```
SUBSTRING('ABCDEFG' FROM 2 FOR 3) yields 'BCD'.
```

```
SUBSTRING('ABCDEFG' FROM 4) yields 'DEFG'.
```

```
SUBSTRING('ABCDEFG' FROM 10) yields ''.
```

```
SUBSTRING('ABCDEFG' FROM -6 FOR 3) yields 'ABC'.
```

```
SUBSTRING('ABCEDFG' FROM 2 FOR -1) raises an exception.
```

TRIM

The SQL `TRIM` function removes leading and/or trailing padding characters from a given string.
The `<padding>` must be a string of length 1, which is the character that is removed from the string.

- If `<padding>` is omitted, space characters are removed.
- If the `<trim spec>` is omitted, `BOTH` is assumed.
- If both `<padding>` and `<trim spec>` are omitted, the `FROM` keyword must be omitted.

Syntax

```
<trim function> ::=
      TRIM ( [<trim spec>] [<padding>] [FROM] <scalar expression> )

<trim spec> ::=
      LEADING
    | TRAILING
    | BOTH

<padding> ::=
    <scalar expression>
```

Examples

`TRIM(' Hello world ')` yields 'Hello world'.

`TRIM(LEADING '0' FROM '00000789.75')` yields '789.75'.

USER

The USER function returns the name of the current user; this function is the same as CURRENT_USER.

Syntax

```
<user function> ::= USER
```

Example

The following statement returns all notes from the INVOICES table that were placed there by the current user.

```
SELECT * FROM INVOICES
    WHERE SOURCE = USER;
```

Table Expressions

This section describes a number of conventions that are used in the following statements reference. Specifically:

- Select expressions
- Unions, intersections, and differences
- Join expressions

```
<table expression> ::= <table expression> UNION [ALL] <table expression> | <table expression>
EXCEPT [ALL] <table expression> |
<table expression> INTERSECT [ALL] <table expression> | <join expression> | <select expression> |
( <table expression> )
```

Select Expressions

A *select expression* is the table expression most often used in a `SELECT` statement.

- Specify `DISTINCT` to remove any duplicates in the result.
- Specify `GROUP BY` and `HAVING` in connection with aggregate functions to calculate summary values from the data in a table. The `WHERE` clause (if present) limits the number of rows included in the summary. If an aggregate function is used without a `GROUP BY` clause, a summary for the whole table is calculated. If a `GROUP BY` clause is present, a summary is computed for each unique set of values for the columns listed in the `GROUP BY`. Then, if the `HAVING` clause is present, it filters out complete groups given the conditional expression in the `HAVING` clause.

Summary queries have additional rules about where columns can appear in expressions:

- There can be no aggregate functions in the `WHERE` clause.
- Column references appearing outside an aggregator must be in the `GROUP BY` clause.
- You cannot nest aggregator functions.

Syntax

```
<select expression> ::=
    SELECT [ ALL | DISTINCT ] <select item commalist>
    FROM <table reference commalist>
        [ WHERE <conditional expression> ]
        [ GROUP BY <column reference commalist> ]
        [ HAVING <conditional expression> ]
```

```
<select item> ::=
      <scalar expression> [ [AS] <output column name> ]
     | [ <range variable> . ] *
```

```
<table reference> ::=
      <join expression>
    | <table name> [ <output table rename> ]
    | ( <table expression> ) [ <output table rename> ]
<output table rename> ::=
      [AS] <range variable> [ ( <column name commalist> ) ]
<conditional expression> ::=
      <conditional expression> OR <conditional expression>
    | <conditional expression> AND <conditional expression>
    | NOT <conditional expression>
    | <scalar expression> <compare operator> <scalar expression>
    | <scalar expression> <compare operator> { ANY | SOME | ALL }
        (<table expression>)
    | <scalar expression> [NOT] BETWEEN <scalar expression>
    | <scalar expression> [NOT] LIKE <scalar expression>
        [ ESCAPE <scalar expression> ]
    | <scalar expression> [NOT] IS { NULL | TRUE | FALSE | UNKNOWN }
    | <scalar expression> IN ( <scalar expression commalist> )
    | <scalar expression> IN ( <table expression> )
    | EXISTS ( <table expression> )
<column reference> ::=
      [ <table qualifier> . ] <column name>
<scalar expression> ::=
      <scalar expression> {+ | - | * | / | <concat> } <scalar expression>
    | {+ | -} <scalar expression>
    | ( <expression> )
    | ( <table expression> )
    | <column reference>
    | <user defined function reference>
    | <literal>
    | <aggregator function>
    | <function>
    | <parameter marker>
<table name> ::=
      [ <schema name> . ] <SQL identifier>
<schema name> ::=
      <SQL identifier>
<user defined function reference> ::=
    <method name> ([ <expression commalist> ])
```

Example 1

The following statement yields a single row with the total value of all orders.

```
SELECT SUM(Amount * Price) FROM Orders;
```

Example 2

The following statement returns a single row with the number of orders where Amount is non-null for the customer 123.

```
SELECT COUNT(Amount) FROM Orders WHERE CustId = 123;
```

Example 3

The following statement returns a set of rows where the total value of all orders grouped by customers for the customers with an ID number less than 200.

```
SELECT CustId, SUM(Amount * Price), COUNT(Amount)
        WHERE CustId < 200 GROUP BY CustId;
```

Example 4

The following example yields a set of big customers with the value of all their orders.

```
SELECT CustId, SUM(Amount * Price), COUNT(Amount)
  GROUP BY CustId HAVING SUM(Amount * Price) > 500000;
```

Example 5

The following statement is illegal because it has nested aggregators.

```
SELECT CustId, COUNT(23 + SUM(Amount)) GROUP BY CustId;
```

Example 6

The following statement is illegal because the `CustId` column is referenced in the select item list, but it is not present in the `GROUP BY` reference list.

```
SELECT CustId, SUM(Amount* Price) GROUP BY Amount;
```

For the syntax of table expressions see "Table expressions".

Unions, Intersections, and Differences

A table expression is an expression that evaluates to an unnamed table. Of the following operators, `INTERSECT` binds the strongest and `UNION` and `EXCEPT` are equal.

`UNION ALL`	Creates the union of two tables including all duplicates.
`UNION`	Creates the union of two tables. If a row occurs multiple times in both tables, the result has this row exactly twice. Other rows in the result have no duplicates.
`INTERSECTION ALL`	Creates the intersection of two tables including all duplicates.
`INTERSECTION`	Creates the intersection of two tables. If a row has duplicates in both tables, the result has this row exactly twice. Other rows in the result has no duplicates.
`EXCEPT ALL`	Creates a table that has all rows that occur only in the first table. If a row occurs *m* times in the first table and *n* times in the second, the result holds that row the larger of zero and *m*-*n* times.
`EXCEPT`	Creates a table that has all rows that occur only in the first table. If a row occurs *m* times in the first table and *n* times in the second, the result holds the row exactly twice if $m > 1$ and $n = 0$. Other rows in the result has no duplicates.

Example 1

```
SELECT * FROM T1 UNION SELECT * FROM T2 UNION SELECT * FROM T3;
```

is executed as:

```
(SELECT * FROM T1 UNION SELECT * FROM T2) UNION SELECT * FROM T3;
```

Example 2

```
SELECT * FROM T1 UNION SELECT * FROM T2 INTERSECT SELECT * FROM T3;
```

is executed as:

```
SELECT * FROM T1 UNION (SELECT * FROM T2 INTERSECT SELECT * FROM T3);
```

Join Expressions

In Blackfish SQL, join expressions give access to a wide variety of join mechanisms. The two most commonly used, inner joins and cross joins, can be expressed with a `SELECT` expression alone, but any kind of outer join must be expressed with a `JOIN` expression.

`CROSS JOIN`	*A* `CROSS JOIN` *B* produces the same result set as `SELECT A.*, B.* FROM A,B`

`INNER JOIN`	`A INNER JOIN B ON A.X=B.X`
	produces the same result as
	`SELECT A.*, B.* FROM A,B WHERE A.X=B.X`

`LEFT OUTER`	`A LEFT OUTER JOIN B ON A.X=B.X` produces the rows from the corresponding inner join plus the rows from A that didn't contribute, filling in the spaces corresponding to columns in B with NULLs.
`RIGHT OUTER`	`A RIGHT OUTER JOIN B ON A.X=B.X` produces the rows from the corresponding inner join plus the rows from B that didn't contribute, filling in the spaces corresponding to columns in A with NULLs.
`FULL OUTER`	`A FULL OUTER JOIN B ON A.X=B.X` produces the rows from the corresponding inner join plus the rows from A and B that didn't contribute, filling in the spaces corresponding to columns in B and A with NULLs.
`UNION`	`A UNION JOIN B` produces a result similar to the following:

```
A LEFT OUTER JOIN B ON FALSE
    UNION ALL
    A RIGHT OUTER JOIN B ON FALSE
```

a table with columns for all columns in A and B, with all the rows from A having NULL values for columns from B appended with all the rows from B having NULL values for columns from A.

The following are mutually exclusive: `ON` `ON` is an expression that needs to be fulfilled for a `JOIN` expression. `USING` `USING( C1, C2, C3)` is equivalent to the `ON` expression above `A.C1=B.C1 AND A.C2=B.C2 AND A.C3=B.C3`,

except that the resulting table has columns C1, C2, and C3 occurring once each as the first three columns. NATURAL NATURAL is the same as a USING clause with all the column names that appear in both tables A and B.

Syntax

```
<join expression> ::=
      <table reference> CROSS JOIN <table reference>
    | <table reference> [NATURAL] [INNER] JOIN <table reference>
         [ <join kind> ]
    | <table reference> [NATURAL] LEFT [OUTER] JOIN <table reference>
         [ <join kind> ]
    | <table reference> [NATURAL] RIGHT [OUTER] JOIN <table reference>
         [ <join kind> ]
   | <table reference> [NATURAL] FULL [OUTER] JOIN <table reference>
         [ <join kind> ]
   | <table reference> UNION JOIN <table reference>
<table reference> ::=
      <join expression>
    | <table name> [ <output table rename> ]
          <table reference> CROSS JOIN <table reference>
    | <table reference> [NATURAL] [INNER] JOIN <table reference>
         [ <join kind> ]
    | <table reference> [NATURAL] LEFT [OUTER] JOIN <table reference>
         [ <join kind> ]
    | <table reference> [NATURAL] RIGHT [OUTER] JOIN <table reference>
         [ <join kind> ]
   | <table reference> [NATURAL] FULL [OUTER] JOIN <table reference>
         [ <join kind> ]
   | <table reference> UNION JOIN <table reference>
<table reference> ::=
      <join expression>
    | <table name> [ <output table rename> ]| ( <table
expression> ) [ <output table rename> ]
```

```
<output table rename> ::=
      [AS] <range variable> [ ( <column name commalist> ) ]
<range variable> ::=
      <SQL identifier>
<join kind> ::=
      ON <conditional expression>
    | USING ( <column name commalist> )
```

Examples

```
SELECT * FROM Tinvoice FULL OUTER JOIN Titem USING ("InvoiceNumber");
SELECT * FROM Tinvoice LEFT JOIN Titem ON Tinvoice."InvoiceNumber"
    = Titem."InvoiceNumber";
SELECT * FROM Tinvoice NATURAL RIGHT OUTER JOIN Titem;
SELECT * FROM Tinvoice INNER JOIN Titem USING ("InvoiceNumber");
SELECT * FROM Tinvoice JOIN Titem ON Tinvoice."InvoiceNumber"
    = Titem."InvoiceNumber";
```

Statements

The Blackfish SQL JDBC driver supports a subset of the ANSI/ISO SQL-92 standard. In general, it provides:

- Data definition language for managing tables and indexes, schemas, views, and security elements.
- Data manipulation and selection with `INSERT`, `UPDATE`, `DELETE`, and `SELECT`; but no cursors.
- Support for general table expressions including `JOIN`, `UNION`, and `INTERSECT`.

For Blackfish SQL for Java, cursor operations are supported through the JDBC version 3.0 ResultSet API.

Syntax

```
<SQL statement> ::=
     <data definition statement>
   | <transaction control statement>
   | <data manipulation statement>
```

```
<data definition statement> ::=
     <create schema statement>
   | <drop schema statement>
   | <create table statement>
   | <alter table statement>
   | <drop table statement>
   | <create view statement>
   | <alter view statement>
   | <drop view statement>
   | <create index statement>
   | <drop index statement>
   | <create method statement>
   | <drop method statement>
   | <create class statement>
   | <drop class statement>
   | <create user statement>
   | <alter user statement>
   | <drop user statement>
```

```
    | <create role statement>
    | <drop role statement>
    | <grant statement>
    | <revoke statement>
    | <set role statement>
```

```
<transaction control statement> ::=
     <commit statement>
   | <rollback statement>
   | <set autocommit statement>
   | <set transaction statement>
<data manipulation statement> ::=
     <select statement>
   | <single row select statement>
   | <delete statement>
   | <insert statement>
   | <update statement>
   | <call statement>
   | <lock statement>
```

Data Definition Statements

CREATE SCHEMA

The CREATE SCHEMA statement creates a name space for tables, views, and methods. You can use it to create multiple objects in one SQL statement.

- You can create a table, view, or method in an existing schema in two ways:
  - You can create it as part of a CREATE SCHEMA statement.
  - You can specify a schema name as part of the object name when you issue a standalone CREATE TABLE, CREATE VIEW, or CREATE METHOD statement. If you use the latter method (using CREATE TABLE, for example), you must specify a schema name that already exists.
- To create an object in a new schema, specify a new schema name in the CREATE SCHEMA statement and then create the table, view, or method as part of the CREATE SCHEMA statement.
- The AUTHORIZATION clause names the owner of the schema. If you do not specify an owner, the owner is the user of the SQL session. Only an administrator can specify a user name other than their own user name in the AUTHORIZATION clause.
- If you issue a standalone CREATE TABLE, CREATE VIEW, or CREATE METHOD statement (meaning that it s not embedded in a CREATE SCHEMA statement) and you do not specify a schema name as part of the CREATE statement, Blackfish SQL uses the following algorithm to assign the new object to a schema:
  - If you have explicitly created a schema that has the same name as your current user name, then you

have created a personal default schema. The table, view, or method belongs to your default schema.
    - If you have not created a personal default schema, the table, view, or method belongs to the DEFAULT_SCHEMA schema.
- You can create schemas with names other than your user name, but you cannot create schemas that have other users' names unless you have administrative privileges.
- All objects created in early versions of Blackfish SQL that did not support schemas belong to the DEFAULT_SCHEMA schema when migrated to version 7 or later.
- A semicolon marks the end of the CREATE SCHEMA statement. There cannot be any semicolons between the schema elements.
- All the statements in the schema element list are executed as one statement in the same transaction.

Default Schemas

Initially your default schema is DEFAULT_SCHEMA. When you create a schema with the same name as your current user name, that schema becomes your default schema. You can create objects without specifying a schema name and those objects automatically belongs to your default schema.

Assume, for example, that user PETER created a schema PETER. At a later time, PETER creates a table without specifying a schema. The table belongs to the PETER schema.

In the following example, the created table would actually be named PETER.FOO.

```
[USER: PETER]
CREATE TABLE FOO (COL1 INT, COL2 VARCHAR);
```

You are permitted to create schemas with names other than your user name, but they can never be your default schema. You cannot create a schema that has another user's name unless you are an administrator.

Syntax

```
<create schema statement> ::=
    CREATE SCHEMA [ <schema name> ]
        [ AUTHORIZATION <user name> ]
         <schema element list>
<schema name> ::=
        <SQL identifier>
<schema element commalist> ::=
    <create table statement>
  | <create view statement>
  | <create method statement>
  | <grant statement>
```

See GRANT for more information about GRANT statements.

Example

The following statement creates the schema BORIS with a table T1 and a view V1. In this schema, the user BJORN is granted SELECT privileges on view V1. After this statement executes, BORIS is the default schema for user BORIS.

```
[USER: BORIS]
CREATE SCHEMA BORIS
    CREATE TABLE T1 (C1 INT, C2 VARCHAR)
    CREATE VIEW V1 AS SELECT C2 FROM T1
    GRANT SELECT ON V1 TO BJORN;
```

DROP SCHEMA

The DROP SCHEMA statement deletes the specified schema. If the command is used without options, it is the same as specifying the RESTRICT option: the schema to be dropped must be empty. The command fails if the schema contains any objects.

- The RESTRICT option causes the statement to fail if there are any objects in the schema. RESTRICT is the default option.
- Used with the CASCADE option, DROP SCHEMA deletes the named schema including all of its tables, views, foreign key dependencies, and methods.

NOTE: The DROP SCHEMAcommand used with the CASCADE option is extremely powerful and should be used with caution. When this command is issued, it drops the schema and all of its objects and dependencies without any chance to change your mind. There is no undo.

TIP: If you want to drop a schema but wish to preserve some of its tables, use the ALTER TABLE command to assign the tables to another schema. For example:

```
ALTER TABLE OLDSCHEMA.JOBS
RENAME TO NEWSCHEMA.JOBS;
```

Syntax

```
<drop schema statement> ::=
  DROP SCHEMA <schema name> [ CASCADE | RESTRICT ]
```

Examples

1. The following two statements are the same: they drop the schema BORIS; they both fail if the schema contains any objects.

```
DROP SCHEMA BORIS;
DROP SCHEMA BORIS RESTRICT;
```

2. The following statement drops the schema BORIS and all of its tables, views, and methods. It also drops any dependent views and foreign keys.

```
DROP SCHEMA BORIS CASCADE;
```

CREATE TABLE

The CREATE TABLE statement creates a Blackfish SQL table. Each column definition must include at least a column name and data type. Optionally, you can specify a default value for each column, along with uniqueness constraints.

You can also optionally specify a foreign key and primary key. Blackfish SQL supports the use of one or more columns as a primary key or foreign key.

Specifying Schemas

To create a table in a particular schema, specify the schema name as part of the table name:

```
CREATE TABLE SOMESCHEMA.MYTABLE(. . .);
```

If you do not specify a schema name, the table is created in your default schema. See CREATE SCHEMA for more information about schemas.

Tracking Data Changes for DataExpress

NOTE: This feature is supported for Blackfish SQL for Java, only.

If you specify RESOLVABLE as part of the table definition, Blackfish SQL keeps track of changes made to the data. The recorded changes are available to the DataExpress application, but not to SQL. The default is NOT RESOLVABLE.

Overriding Consistency Checks

The `NO CHECK` option creates the foreign key without checking the consistency at creation time. Use this option with caution.

Using AutoIncrement Columns with SQL

To create or alter a column to have the Autoincrement property using SQL, add the `AUTOINCREMENT` keyword to your `<table element>` definition.

The following statement creates table `T1` with an integer autoincrement column called `C1`:

```
CREATE TABLE T1 ( C1 INT AUTOINCREMENT, C2 DATE, C3 CHAR(32) );
```

To obtain the Autoincrement value of a newly inserted row using the JDS JDBC driver (JVM version 1.3 or earlier), call the `JdsStatement.getGeneratedKeys` method. This method is also available in the statement interface of JDBC 3 in JVM 1.4.)

Specifying Column Position

In the columns definition, use the `POSITION` option to force a column to be in a particular position in the table (second column, for example). The following code snippet forces column `COLD` to be the second column:

```
CREATE TABLE(COLA INT, COLB STRING, COLC INT, COLD STRING POSITION 2);
```

Syntax

```
<create table statement> ::=
      CREATE TABLE <table name> ( <table element commalist> )
```

```
<table name> ::=
      [ <schema name> . ] <SQL identifier>
<schema name> ::=
      <SQL identifier>
```

```
<table element> ::=
        <column definition>
       | <primary key>
       | <unique key>
       | <foreign key>
       | [NOT] RESOLVABLE
```

```
<column definition> ::=
    <column name> <data type>
    [ DEFAULT <default value> ]
    [ [NOT] NULL ]
    [ AUTOINCREMENT ]
    [ POSITION <integer literal> ]
    [ [ CONSTRAINT <constraint name> ] PRIMARY KEY ]
    [ [ CONSTRAINT <constraint name> ] UNIQUE ]
    [ [ CONSTRAINT <constraint name> ] <references definition> ]
```

```
<column name> ::=
      <SQL identifier>

<default value> ::=
      <literal>
      | <current date function>
```

```
<current date function> ::=
      CURRENT_DATE
     | CURRENT_TIME
     | CURRENT_TIMESTAMP
```

```
<primary key> ::=
         [ CONSTRAINT <constraint name> ] PRIMARY KEY <column
name commalist>)
<unique key> ::=
    [ CONSTRAINT <constraint name> ] UNIQUE ( <column
name commalist> )
<foreign key> ::=
      [ CONSTRAINT <constraint name> ] FOREIGN KEY ( <column
name commalist> )
         <references definition>
<references definition> ::=
     REFERENCES <table name> [ ( <column name commalist> ) ]
     [ ON DELETE <action> ]
     [ ON UPDATE <action> ]
     [ NO CHECK ]

<action> ::=
     NO ACTION
    | CASCADE
    | SET DEFAULT
    | SET NULL

<constraint name> ::=
    <SQL identifier>
```

Example 1

The following statement creates a table with four columns. The `CustId` column is the primary key and the `OrderDate` column has the current time as the default value.

```
CREATE TABLE Orders ( CustId INTEGER PRIMARY KEY, Item VARCHAR(30),
     Amount INT, OrderDate DATE DEFAULT CURRENT_DATE);
```

Example 2

The following statement creates a table that uses two columns for the primary key constraint:

```
CREATE TABLE T1 (C1 INT, C2 STRING, C3 STRING, PRIMARY KEY (C1, C2));
```

Example 3

The following statement creates a table `T1` in the `BORIS` schema:

```
CREATE TABLE BORIS.T1 (C1 INT, C2 STRING, C3 STRING);
```

ALTER TABLE

The ALTER TABLE statement performs the following operations:

- Adds or removes columns in a Blackfish SQL table
- Sets or drops column defaults and NULLability
- Changes column data types
- Adds or drops primary key, unique key, and foreign key column constraints and table constraints; changes the referenced table and type of action for these constraints
- Renames columns
- Renames tables; this also allows you to move tables from one schema to another
- Adds or drops the RESOLVABLE table property
- Repositions columns within the table

Syntax

```
<alter table statement> ::=
    ALTER TABLE <table name> <change definition commalist>

<table name> ::=  [ <schema name> . ]<SQL identifier>
```

```
<change definition> ::=
      <add column element>
    | <drop column element>
    | <alter column element>
    | <add constraint>
    | <drop constraint>
    | [RENAME] TO <table name>
    | [NOT] RESOLVABLE
<add column element> ::= ADD [COLUMN] <column definition>
<column definition> ::=
       <column name> <data type>     [ DEFAULT <default value> ]
    [ [NOT] NULL ]
    [ AUTOINCREMENT ]
    [ POSITION <integer literal> ]
    [ [ CONSTRAINT <constraint name> ] PRIMARY KEY ]
    [ [ CONSTRAINT <constraint name> ] UNIQUE ]
    [ [ CONSTRAINT <constraint name> ] <references definition> ]
<drop column element> ::= DROP [COLUMN] <column name>
```

```
<alter column element> ::=
       ALTER [COLUMN] <column name> [TYPE] <data type>
     | ALTER [COLUMN] <column name> SET DEFAULT <default-value>
     | ALTER [COLUMN] <column name> DROP DEFAULT
     | ALTER [COLUMN] <column name> [NOT] NULL
     | ALTER [COLUMN] <column name> [RENAME] TO <column name>
     | ALTER [COLUMN] <column name> [POSITION] <integer literal>
     | ALTER [COLUMN] <column name> AUTOINCREMENT
     | ALTER [COLUMN] <column name> DROP AUTOINCREMENT
<add constraint> ::= ADD <base table constraint>
<base table constraint> ::=
    <primary key> |  <unique key> |  <foreign key>
<drop constraint> ::= DROP CONSTRAINT <constraint name>
<primary key> ::=
    [ CONSTRAINT <constraint name> ]
    PRIMARY KEY <column name commalist>)
<unique key> ::=
    [ CONSTRAINT <constraint name> ]
      UNIQUE ( <column name commalist> )
<foreign key> ::=
    [ CONSTRAINT <constraint name> ]
       FOREIGN KEY ( <column name commalist> )
   <references definition>

<references definition> ::=
      REFERENCES <table name> [ ( <column name commalist> ) ]
    [ ON DELETE <action> ]
    [ ON UPDATE <action> ]
    [ NO CHECK ]
<action> ::=
        NO ACTION
    | CASCADE
    | SET DEFAULT
    | SET NULL

<constraint name> ::=  <SQL identifier>
```

In `ALTER [COLUMN]`, the optional `COLUMN` keyword is included for SQL compatibility. It has no effect.

Example

The following example adds a column named `ShipDate` to the `Orders` table and drops the `Amount` column from the table.

```
ALTER TABLE Orders
    ADD ShipDate DATE,
     DROP Amount;
```

The following example moves the `Jobs` table from the `OldSchema` schema to the `NewSchema` schema.

```
ALTER TABLE OldSchema.Jobs
 RENAME TO NewSchema.Jobs;
```

DROP TABLE

The `DROP TABLE` statement deletes a table and its indexes from a Blackfish SQL database.

- The `RESTRICT` option guarantees that the statement will fail if there are foreign key or view dependencies on the table.
- The `CASCADE` option causes all dependent views and foreign keys to be dropped when the table is dropped.
- Specifying neither `RESTRICT` nor `CASCADE` drops the table and any foreign keys that reference it. The statement fails if there are dependent views.

Syntax

```
<drop table statement> ::=
      DROP TABLE [ <schema name> . ]<table name> [ CASCADE|RESTRICT ]
<schema name> ::= <SQL identifier>
```

Examples

1. The following statement drops the `Orders` table only if there are no dependent views. If there are dependent foreign keys, the statement succeeds and the foreign keys are dropped.

   ```
   DROP TABLE Orders;
   ```

2. The following statement drops the `Orders` table only if there are no dependent views or foreign keys.

```
DROP TABLE Orders RESTRICT;
```

3. The following statement drops the `Orders` table. All dependent views and dependent foreign keys are also dropped.

```
DROP TABLE Orders CASCADE;
```

CREATE VIEW

The `CREATE VIEW` statement creates a derived table by selecting specified columns from existing tables. Views provide a way of accessing a consistent subcollection of the data stored in one or more tables. When the data in the underlying tables changes, the view reflects this change.

Views look just like ordinary database tables, but they are not physically stored in the database. The database stores only the view definition, and uses this definition to filter the data when a query referencing the view occurs.

When you create a view, you can specify names for the columns in the view using the optional <column name commalist> portion of the syntax. If you do not specify column names, the names of the table columns from which the view columns are derived are used. If you do specify column names, you must specify exactly the number of columns that will be returned from the `SELECT` query.

The `WITH CHECK OPTION` clause causes a runtime check to be performed to ensure that an inserted or updated row will not be filtered out by the `WHERE` clause of the view definition.

Views are updatable only under limited conditions. If you want to execute `INSERT`, `UPDATE`, or `DELETE` on a view, it must meet all of the following conditions:

- It is derived from a single table.
- None of the columns are calculated.
- The `SELECT` clause that defines the view does not contain the `DISTINCT` keyword.
- The `SELECT` expression that defines the view does not contain any of the following:
  - Subqueries
  - A `HAVING` clause
  - A `GROUP BY` clause
  - An `ORDER BY` clause
  - Aggregate functions
  - methods

Syntax

```
<create view statement> ::=
     CREATE VIEW <view name> [ ( <column name commalist> ) ]
    AS <select expression> [ WITH CHECK OPTION ]
<view name> ::=
    [ <schema name> . ] <SQL identifier>
```

Example

The following statement creates a view `V1` from table `T1`. The columns in the view are named `C1` and `C2`.

```
CREATE VIEW V1(C1,C2)
     AS SELECT  C8+C9, C6 FROM T1 WHERE C8 < C9;
```

ALTER VIEW

The `ALTER VIEW` statement modifies a view without losing dependent views and existing `GRANT`s. This statement can be used to change the name of a view, the columns that comprise the view, and whether the view has the `WITH CHECK OPTION` constraint.

Note that after `ALTER VIEW` executes, it is possible that there are dependent views that are no longer valid.

Syntax

```
<alter view statement> ::=
     ALTER VIEW <view name> [ ( <column name commalist> ) ]
    AS <select expression> [ WITH CHECK OPTION ]
```

Example

The following statements show how the `ALTER VIEW` statement can be used to validate an invalid view. The first two statements create a table and then create a view based on that table. The third statement, `SELECT`, succeeds.

```
CREATE TABLE T1 (C1 INT, C2 VARCHAR);
CREATE VIEW V1 AS SELECT C1, C2 FROM T1;
SELECT * FROM V1;
```

The following statement changes a column name in the table.

```
ALTER TABLE T1 ALTER COLUMN C1 RENAME TO ID;
```

The next `SELECT` statement therefore fails because there is no longer a `C1` column in the table `T1`, which is accessed by view `V1`.

```
SELECT * FROM V1;
```

The following `ALTER VIEW` statement changes the definition of the view, so that the next `SELECT` statement succeeds.

```
ALTER VIEW V1 (C1, C2) AS SELECT ID, C2 FROM T1;
SELECT * FROM V1;
```

DROP VIEW

The `DROP VIEW` statement drops the named view. It fails if there are dependencies on the view.

- The `RESTRICT` option is the same as specifying no options: the statement fails if there are dependencies on the view.
- The `CASCADE` option drops the view and any dependent views.

Syntax

```
<drop view statement> ::=
     DROP VIEW <view name> [ CASCADE | RESTRICT ]
```

Example

The following code creates a table and two views:

```
CREATE TABLE T1 (C1 INT, C2 VARCHAR);
CREATE VIEW V1 AS SELECT C1, C2 FROM T1;
CREATE VIEW V2 AS SELECT C1, C2 FROM V1;
```

The following statement fails because view V1 has a dependent view (V2).

```
DROP VIEW V1 RESTRICT;
```

The following statement succeeds and both V1 and V2 are dropped.

```
DROP VIEW V1 CASCADE;
```

CREATE INDEX

The `CREATE INDEX` statement creates an index for a Blackfish SQL table. Each column can be ordered in ascending or descending order. The default value is ascending order.

Syntax

```
<create index statement> ::=
      CREATE [UNIQUE] [CASEINSENSITIVE] INDEX <index name>
           ON <table name> ( <index element commalist> )
<table name> ::=
    [ <schema name> . ]<SQL identifier>

<index name> ::=
      <SQL Identifier>

<index element> ::=
      <column name> [ DESC|ASC ]
```

Example

The following statement generates a non-unique, case-sensitive, ascending index on the `Item` column of the `Orders` table:

```
CREATE INDEX OrderIndex ON Orders (Item ASC);
```

DROP INDEX

The `DROP INDEX` statement deletes an index from a Blackfish SQL table.

Syntax

```
<drop index statement> ::=
      DROP INDEX <index name> ON <table name>
```

Example

The following statement deletes the OrderIndex index from the Orders table:

```
DROP INDEX OrderIndex ON Orders;
```

CREATE METHOD

The `CREATE METHOD` statement makes a stored procedure or a UDF implemented in Java or a .NET language (e.g., Delphi, C#, or VB.NET) available for use in Blackfish SQL. The class files for the code must be added to the classpath of the Blackfish SQL server process before use. See Stored Procedures and UDFs for details about how to implement stored procedures and UDFs for Blackfish SQL.

To create a method in a particular schema, specify the schema name as part of the table name:

```
CREATE METHOD SOMESCHEMA.MYMETHOD AS . . .
```

If you do not specify a schema name, the method is assigned to a schema as follows:

- If you have created a personal default schema (a schema that has the same name as your user name), the method is created in that schema.
- If you have have not created a personal default schema, the method is created in the DEFAULT_SCHEMA schema.

See CREATE SCHEMA for more information about schemas.
The AUTHORIZATION clause causes the called stored procedure to be run as if the username in the AUTHORIZATION clause were the actual user. If this clause is omitted, the *current_user* is used as the actual user during method calls. This feature allows the current user controlled access to tables and views that would not otherwise be accessible.

Syntax

```
<create method statement> ::=
    CREATE METHOD <method name> [AUTHORIZATION <username>]
         AS <method definition>

<method name> ::=
    [ <schema name> . ] <SQL identifier>

<schema name> ::= <SQL identifier>

<method definition> ::= <string literal>
```

Example

```
CREATE METHOD ABS AS 'MathClass.abs';
```

DROP METHOD

The DROP METHOD statement drops a stored procedure or a UDF, making it unavailable for use in Blackfish SQL SQL.

Syntax

```
<drop method statement> ::=
     DROP METHOD <method_name>
```

Example

```
DROP METHOD ABS;
```

CREATE CLASS

The CREATE CLASS statement makes all public static methods of a class available to Blackfish SQL SQL as stored procedures or UDFs. You must ensure that the class files for the code are on the classpath of the Blackfish SQL server process before use. See the Stored Procedures and UDFs chapter for details.

The AUTHORIZATION clause causes the called stored procedure to be run as if the username in the AUTHORIZATION clause were the actual user. If this clause is omitted, the *current_user* is used as the actual user during method calls. This feature allows the current user controlled access to tables and views that would not otherwise be accessible.

Syntax

```
<create class statement> ::=
    CREATE CLASS <class name> [AUTHORIZATION <username>]
        AS <class definition>

<class name> ::=
    [ <schema name> . ] <SQL identifier>

<schema name> ::= <SQL identifier>

<class definition> ::= <string literal>
```

Examples

```
CREATE CLASS MATH AS 'mscorlib::System.Math';
```

After the above statement executes, all public static methods in System.Math can be called from SQL. Note that the method names are case sensitive.

Usage

The following statement calls the Abs() method in System.Math:

```
SELECT * FROM CUSTOMER WHERE MATH."Abs"(AGE - 50) < 5;
```

DROP CLASS

The `DROP CLASS` statement drops a stored class, making it unavailable for use in Blackfish SQL.

Syntax

```
<drop class statement> ::=
   DROP CLASS <method_name>
```

Example

```
DROP CLASS MATH;
```

CREATE TRIGGER

The `CREATE TRIGGER` statement creates a row level trigger for a table. You must ensure that the classes can be loaded by the Blackfish SQL server process. See Triggers for Blackfish SQL for details on implementing trigger methods and ensuring that the method classes can be loaded.

Syntax

```
create trigger statement> ::= CREATE TRIGGER  <trigger name>
    <trigger action time> <trigger action name>
   ON <table name> AS <trigger spec>
<trigger name> ::=   <SQL identifier>
 <tablename> ::= <SQL identifier>
 <triggeraction time> ::= <BEFORE | AFTER>
  <trigger action name> ::= <INSERT | UPDATE | DELETE >
```

Blackfish SQL for Java:

```
  <trigger spec> ::= "[<package>.]<class-name>.<method-name>"
```

Blackfish SQL for windows:

```
<trigger spec> ::="<assembly-name>::[<name-space>.]<class-name>.<method-name>"
```

Examples

Blackfish SQL for Windows:

```
CREATE TRIGGER VALIDATE_CUSTOMER BEFORE INSERT ON CUSTOMER AS
 OrderEntryAssembly::OrderEntry.Customers.ValidateCustomer
```

Blackfish SQL for Java:

```
CREATE TRIGGER VALIDATE_CUSTOMER BEFORE INSERT ON CUSTOMER AS
OrderEntry.Customers.validateCustomer
```

DROP TRIGGER

The `DROP TRIGGER` statement drops a trigger, making it unavailable for use in Blackfish SQL.

Syntax

```
<drop trigger statement> ::= DROP TRIGGER  <trigger name>
      ON <table name>

  <trigger name> ::=  <SQL identifier>
  <table name> ::= <SQL identifie
```

Example

```
DROP TRIGGER VALIDATE_CUSTOMER on CUSTOMER
```

Transaction Control Statements

COMMIT

The `COMMIT` statement commits the current transaction. It has an effect only if `AUTOCOMMIT` is turned off.

Syntax

```
<commit statement> ::=
      COMMIT [WORK]
```

ROLLBACK

The `ROLLBACK` statement rolls back the current transaction. This statement does not have any effect when `AUTOCOMMIT` is turned on.

Syntax

```
<rollback statement> ::=
      ROLLBACK [WORK]
```

SET AUTOCOMMIT

The `SET AUTOCOMMIT` statement changes the autocommit mode. Autocommit is initially `ON` when a JDBC connection is created.

The autocommit mode is also controllable using the JDBC `Connection` instance.

Syntax

```
<set autocommit statement> ::=
    SET AUTOCOMMIT { ON | OFF };
```

SET TRANSACTION

The `SET TRANSACTION` statement sets the properties for the following transaction. You can use it to specify the isolation level and whether the transaction is read-write or read-only. See Transaction management for a detailed discussion of Blackfish SQL transaction management.

This command must be issued when there is no open transaction. It affects only the next transaction and does not itself start a transaction.

To understand isolation levels, you should understand the following terms:

- A **dirty read** occurs when a row changed by one transaction is read by another transaction before any changes in that row have been committed.

- A **non-repeatable read** occurs when one transaction reads a row, a second transaction alters the row, and the first transaction rereads the row, getting different values the second time.
- A **phantom read** occurs when one transaction reads all rows that satisfy a `WHERE` condition, a second transaction inserts a row that satisfies that `WHERE` condition, and the first transaction rereads for the same condition, retrieving the additional "phantom" row in the second read.

Blackfish SQL offers the following transaction isolation levels:

`TRANSACTION_READ_UNCOMMITTED` permits dirty reads, non-repeatable reads, and phantom reads. If any of the changes are rolled back, the row retrieved by the second transaction is invalid. This isolation level does not acquire row locks for read operations. It also ignores exclusive row locks held by other connections that have inserted or updated a row.

`TRANSACTION_READ_COMMITTED` prevents dirty reads; non-repeatable reads and phantom reads are permitted. This level only prohibits a transaction from reading a row with uncommitted changes in it. This level does not acquire row locks for read operations, but blocks when reading a row that has an exclusive lock held by another transaction.

`TRANSACTION_REPEATABLE_READ` prevents dirty reads and non-repeatable reads but prevents phantom reads. It acquires shared row locks for read operations. This level provides protection for transactionally consistent data access without the reduced concurrency of `TRANSACTION_SERIALIZABLE`, but results in increased locking overhead.

`TRANSACTION_SERIALIZABLE` provides complete serializability of transactions at the risk of reduced concurrency and increased potential for deadlocks.

Syntax

```
<set transaction statement> ::=
    SET TRANSACTION <transaction option commalist>

<transaction option> ::=
    READ ONLY
  | READ WRITE
  | ISOLATION LEVEL <isolation level>

<isolation level> ::=
    READ UNCOMMITTED
  | READ COMMITTED
  | REPEATABLE READ
  | SERIALIZABLE
```

Example

In the following example the select from T1 will be a dirty read, meaning that the data cannot yet be committed by another user. After the second `COMMIT`, the isolation level returns to whatever was specified for the session.

```
COMMIT;
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
SELECT * FROM T1;
COMMIT;
```

Data Manipulation Statements

SELECT

A SELECT statement retrieves data from one or more tables. The optional keyword DISTINCT eliminates duplicate rows from the result set. The keyword ALL, which is the default, returns all rows including duplicates. The data can optionally be sorted using ORDER BY. The retrieved rows can optionally be locked for an upcoming UPDATE by specifying FOR UPDATE.

Syntax

```
<select statement> ::=
    <table expression> [ ORDER BY <order item list> ]
        [ FOR UPDATE|FOR READ ONLY ]
<table expression> ::=
      <table expression> UNION [ALL] <table expression>
    | <table expression> EXCEPT [ALL] <table expression>
    | <table expression> INTERSECT [ALL] <table expression>
    | <join expression>
    | <select expression>
    | ( <table expression> )
<order item> ::= <order part> [ ASC|DESC ]
<order part> ::=
     <integer literal> | <column name> | <expression>
<select expression> ::=
    SELECT [ ALL|DISTINCT ] <select item commalist>
    FROM <table reference commalist>
        [ WHERE <conditional expression> ]
        [ GROUP BY <column reference commalist> ]
        [ HAVING <conditional expression> ]
```

Examples

The following statement orders the output by the first column in descending order.

```
SELECT Item FROM Orders ORDER BY 1 DESC;
```

The next statement orders by the calculated column CALC:

```
SELECT CustId, Amount*Price+500.00
    AS CALC FROM Orders
    ORDER BY CALC;
```

The next statement orders the output by the given expression, `Amount*Price`:

```
SELECT CustId, Amount
    FROM Orders
    ORDER BY Amount*Price;
```

SELECT INTO

A `SELECT INTO` statement is a `SELECT` statement that evaluates into exactly one row, whose values are retrieved in output parameters. It is an error if the `SELECT` evaluates into more than one row or to the empty set.

Syntax

```
<single row select statement> ::=
     SELECT [ ALL|DISTINCT ] <select item commalist>
        INTO <parameter commalist>
        FROM <table reference commalist>
            [ WHERE <conditional expression> ]
            [ GROUP BY <column reference commalist> ]
            [ HAVING <conditional expression> ]
```

Example

In the following statement, the first two parameter markers indicate output parameters from which the result of the query can be retrieved:

```
SELECT CustId, Amount
     INTO ?, ?
     FROM Orders
     WHERE CustId=? ;
```

INSERT

The INSERT statement inserts rows into a table in a Blackfish SQL database. The INSERT statement lists columns and their associated values. Columns that aren't listed in the statement are set to their default values.

Syntax

```
<insert statement> ::=
    [ SELECT AUTOINCREMENT FROM ]
    INSERT INTO <table name> [ ( <column name commalist> ) ]
        [ <insert table expression>|DEFAULT VALUES ]
<table name> ::=
      [ <schema name> . ]<SQL identifier>
<insert table expression> ::=
      <select expression>     |  VALUES ( <expression commalist> )
```

Example 1

The following statement inserts one row each time it is executed. It inserts one row each time it is executed. The columns not mentioned are set to their default values. If a column doesn't have a default value, it is set to NULL.

```
INSERT INTO Orders (CustId, Item) VALUES (?,?);
```

Example 2

The following statement finds all the orders from the customer with CustId of 123 and inserts the Item of these orders into the ResTable table.

```
INSERT INTO ResTable
    SELECT Item FROM Orders
    WHERE CustId = 123;
```

Example 3

The following statement inserts one row each time it is executed. In this case, the `CustId` column of the `Orders` table is not specified. It is assumed that the `CustId` is an `AUTOINCREMENT` column, for which the SQL engine automatically generates an incremented value. In this example, `SELECT AUTOINCREMENT FROM` is specified, causing the generated value to be returned as a result set. If the `Orders` table does not have an `AUTOINCREMENT` column, and the result set will be the `INTERNALROW` values for the inserted rows.

```
SELECT AUTOINCREMENT FROM INSERT INTO Orders (Item) VALUES (?)
```

UPDATE

The `UPDATE` statement is used to modify existing data. The columns to be changed are listed explicitly. All the rows for which the `WHERE` clause evaluates to `TRUE` are changed. If no `WHERE` clause is specified, all rows in the table are changed.

Syntax

```
<update statement> ::=
      UPDATE <table name>
      SET <update assignment commalist>
    [ WHERE <conditional expression> ]

<table name> ::=
    [ <schema name> . ] <SQL identifier>
<update assignment> ::=
      <column reference> = <update expression>
<update expression> ::=
      <scalar expression>
    | DEFAULT
    | NULL
```

Example 1

The following statement changes all orders from customer 123 to orders from customer 500:

```
UPDATE Orders SET CustId = 500 WHERE CustId = 123;
```

Example 2

The following statement increases the amount of all orders in the table by 1:

```
UPDATE Orders SET Amount = Amount + 1;
```

Example 3

The following statement reprices all disposable underwater cameras to $7.25:

```
UPDATE Orders SET Price = 7.25
    WHERE Price > 7.25 AND Item = 'UWCamaras';
```

DELETE

A DELETE statement deletes rows from a table in a Blackfish SQL database. If no WHERE clause is specified, all the rows are deleted. Otherwise only the rows that match the WHERE expression are deleted.

Syntax

```
<delete statement> ::=
      DELETE FROM <table name>
    [ WHERE <conditional expression> ]
```

```
<table name> ::=
                                                                [ <schema
name> . ] <SQL identifier>
```

Example

The following statement deletes all orders for shorts from the `Orders` table.

```
DELETE FROM Orders WHERE Item = 'Shorts';
```

CALL

A `CALL` statement calls a stored procedure.

Syntax

```
<call statement>  ::=
     [ ? = ] CALL <method name> ( <expression commalist> )
```

Example 1

The parameter marker indicates an output parameter position from which the result of the stored procedure can be retrieved.

```
?=CALL ABS(-765);
```

Example 2

The method implementing `IncreaseSalaries` updates the `salaries` table with an increase of some percentage for all employees. A connection object will be passed implicitly to the method. An `updateCount` of all the rows affected by `IncreaseSalaries` will be returned from `Statement.executeUpdate`.

```
CALL IncreaseSalaries(10);
```

LOCK TABLE

The `LOCK TABLE` statement explicitly locks a table. The lock ceases to exist when the transaction

is committed or rolled back.

Syntax

```
<lock statement>  ::=
    `LOCK <table name commalist>
```

```
<table name> ::=
    [ <schema name> . ] <SQL identifier>
```

Example

The following statement locks the `Orders` and `LineItems` tables.

```
LOCK Orders, LineItems;
```

Security Statements

CREATE USER

The `CREATE USER` statement adds the named user and associated password to the database. Only an administrator can create users.

NOTE: The password that you enter is always case sensitive. The user name is not case-sensitive.

A newly created user has all database privileges except `ADMINISTRATOR` by default. That is, they have `STARTUP`, `WRITE`, `CREATE`, `DROP`, `CREATE ROLE`, and `CREATE SCHEMA` privileges. If you wish to remove certain privileges from a user, use `REVOKE` to remove them.

Syntax

```
<create user statement> ::=
    CREATE USER <user name> PASSWORD <SQL identifier>
```

Example

```
CREATE USER jmatthews PASSWORD "@nyG00dPas2d";
```

ALTER USER

The ALTER USER statement sets a new password for an existing user. Only an administrator or the named user can change a password.

NOTE: The password that you enter is stored in all caps unless you enclose the password string in double quotation marks. It is recommended that you always use the double quotes when specifying the password.

Syntax

```
<alter user statement> ::=
    ALTER USER <user name> SET PASSWORD <SQL identifier>
```

Example

```
ALTER USER GSMITH SET PASSWORD "usethisOnen0w";
```

DROP USER

The DROP USER statement drops a user and all objects that the user owns.

- Used with RESTRICT or with no option, the statement fails if the user owns any objects, such as tables, views, or methods.
- Used with CASCADE, it deletes the user and all objects that the user owns.

Syntax

```
<drop user statement> ::=
    DROP USER <user name> [ CASCADE|RESTRICT ]
```

Example

The following statement drops the user `gsmith` and all tables, views, and methods that he owns.

```
DROP USER gsmith CASCADE;
```

CREATE ROLE

The `CREATE ROLE` statement creates a named role.

Using roles is a four-step process:

- Create a role using the `CREATE ROLE` statement.
- Grant privileges to the role using the `GRANT` statement.
- Grant the role to one or more users using the `GRANT` statement, thus authorizing that user to use that role.
- An authorized user accesses the privileges granted to a role by using the `SET ROLE` statement.

To create a role, the user must have the `CREATE ROLE` system privilege. All users have this by default, but this privilege can be explicitly revoked.

Syntax

```
<create role statement> ::=
    CREATE ROLE <role name>
```

Example

```
CREATE ROLE salesperson;
```

SET ROLE

The `SET ROLE` statement makes the named role active. The current user acquires all privileges assigned

to that role. Use SET ROLE NONE to deactivate the current role without setting another role.

IMPORTANT: This command must be issued when there is no active transaction. The role remains active until the end of the session or until another SET ROLE command is issued.

Syntax

```
<set role statement> ::=
    SET ROLE <role specification>

<role specification> ::=
    NONE
  | <role name>
```

Example

The following statement makes the Manager role active:

```
SET ROLE Manager;
```

The following statement removes the active role and makes no roles active:

```
SET ROLE NONE;
```

DROP ROLE

The DROP ROLE statement drops the specified role.

- When DROP ROLE is used with CASCADE, all privileges that were granted through this role are revoked.
- When DROP ROLE is used with RESTRICT, the statement fails if the role is currently granted to any users or roles.
- Issuing DROP ROLE with neither option is the same as DROP ROLE with RESTRICT.

Syntax

```
<drop role statement> ::=
     DROP ROLE <role name> [ CASCADE|RESTRICT ]
```

Example

The following statement drops the `Sales` role. All privileges that were granted to users or other roles through the `Sales` role are revoked.

```
DROP ROLE Sales CASCADE;
```

GRANT

The `GRANT` statement performs the following three actions:

- It grants object privileges (for example, `INSERT` or `SELECT`) on tables or methods to `PUBLIC`, users, or roles.
- It grants database privileges (for example, `STARTUP` or `RENAME`) to users or roles.
- It grants roles to users or roles.

`GRANT` options:

- When object privileges are granted with the `GRANT` option, the grantee has the power to pass on the granted object privileges to other users.
- When database privileges or roles are granted with the `ADMINISTRATOR` option, the grantee has the power to pass on the granted database privileges or roles to other users.
- The `ADMINISTRATOR` database privilege grants `STARTUP`, `WRITE`, `CREATE`, `DROP`, `RENAME`, `CREATE ROLE`, and `CREATE SCHEMA` privileges. When these privileges are acquired through the `ADMINISTRATOR` privilege, they can be revoked only by revoking the `ADMINISTRATOR` privilege. In other words, if you grant `ADMINISTRATOR` to a user and then revoke `CREATE`, that user still has `CREATE` privileges.

Note that when specifying the privilege object, you can use the optional `TABLE` keyword to grant privileges on either tables or views. You do not use the `VIEW` keyword in this context. You can also revoke privileges on a method, using the required `METHOD` keyword.

It is possible to grant the following database privileges:

Privilege	Description

ADMINISTRATOR	Grants startup, write, create, drop, rename, create role, and create schema privileges
STARTUP	User can start the database
WRITE	User can write to the database
CREATE	User can create tables
DROP	User can drop tables
RENAME	User can rename tables
CREATE ROLE	User can create roles
CREATE SCHEMA	User can create schemas

CREATE ROLE and CREATE SCHEMA are granted by default when a user is created.

Syntax

```
<grant statement> ::=
      <grant database privileges statement>
    | <grant object privileges statement>
    | <grant role statement>
<grant database privileges statement> ::=
      GRANT <database privilege commalist>
      TO <grantee commalist>
      [ WITH ADMIN OPTION ]
<grant object privileges statement> ::=
      GRANT < object privileges>
      ON <privilege object>
      TO <grantee commalist>
      [ WITH GRANT OPTION ]
      [ GRANTED BY <grantor> ]
<grant role statement> ::=
      GRANT <role name commalist>
      TO <grantee commalist>
      [ WITH ADMIN OPTION ]
      [ GRANTED BY <grantor> ]
<database privilege> ::=
      STARTUP
    | ADMINISTRATOR
    | WRITE
    | CREATE
    | DROP
    | RENAME
    | CREATE ROLE
    | CREATE SCHEMA
<grantee> ::=
      PUBLIC
    | <user name>
    | <role name>
<object privileges> ::=
      ALL PRIVILEGES
    | <privilege commalist>
<privilege>::=
```

```
      SELECT
    | INSERT [ ( <column name commalist> ) ]
    | UPDATE [ ( <column name commalist> ) ]
    | REFERENCES [ ( <column name commalist> ) ]
    | DELETE
    | EXECUTE
<privilege object> ::=
      [TABLE] <table name or view name>
    | METHOD <method name> <grantor> ::=
      CURRENT_USER
    | CURRENT_ROLE
```

Examples

In the following example, USER_1 receives SELECT and INSERT privileges on table T1. USER_2 receives SELECT privileges on table T1 because the SELECT privilege was granted to ROLE_B and ROLE_B was granted to USER_2. However, USER_2 can use this SELECT privilege only after enabling ROLE_B with a SET ROLE statement.

```
GRANT SELECT ON TABLE T1 TO USER_1, ROLE_B;
GRANT INSERT ON T1 TO USER_1;
GRANT ROLE_B TO USER_2;
```

REVOKE

The REVOKE statement can perform the following operations:

- It revokes object privileges—such as INSERT or SELECT—on tables or methods from PUBLIC, users, or roles.
  - If the user or role has granted the now-revoked privilege to others, CASCADE revokes the privileges from those others as well. If any views depend on the revoked privileges, they are dropped.
  - When the REVOKE statement includes RESTRICT, the statement fails if the grantee has granted the acquired privileges to others.
- It revokes database privileges—such as STARTUP or RENAME—from users or roles.
- It revokes roles from users or roles.
- It revokes the ADMIN option from a role without revoking the role itself.
- REVOKE GRANT OPTION FOR *privilege* revokes the power to grant the privilege to others without revoking the privilege itself. REVOKE ADMIN OPTION FOR *role* similarly revokes the power to grant the named role without revoking the role itself.

Note that when specifying the privilege object, you can use the optional TABLE keyword to revoke privileges on either tables or views. You do not use the VIEW keyword in this context. You can

also revoke privileges on a method, using the required METHOD keyword.

Syntax

```
<revoke statement> ::=
      <revoke database privileges statement>
    | <revoke object privileges statement>
    | <revoke role statement>
<revoke database privileges statement> ::=
      REVOKE <database privilege commalist>
      FROM <grantee commalist>
<revoke object privileges statement> ::=
      REVOKE [ GRANT OPTION FOR ] < object privileges>
          ON <privilege object>
           FROM <grantee commalist>
      [ GRANTED BY <grantor> ]
      [ CASCADE|RESTRICT ]
<revoke role statement> ::=
      REVOKE [ ADMIN OPTION FOR ] <role name commalist>
      FROM <grantee commalist>
      [ GRANTED BY <grantor> ]
      [ CASCADE|RESTRICT ]
<database privilege> ::=
      STARTUP
    | ADMINISTRATOR
    | WRITE
    | CREATE
    | DROP
    | RENAME
    | CREATE ROLE
    | CREATE SCHEMA
<grantee> ::=
      PUBLIC
    | <user name>
    | <role name>
<object privileges> ::=
      ALL PRIVILEGES
    | <privilege commalist>
<privilege>::=
      SELECT
    | INSERT [ ( <column name commalist> ) ]
    | UPDATE [ ( <column name commalist> ) ]
    | REFERENCES [ ( <column name commalist> ) ]
    | DELETE
    | EXECUTE
<privilege object> ::=
      [TABLE] <table name or view name>
    | METHOD <method name> <grantor> ::=
      CURRENT_USER
    | CURRENT_ROLE
```

Example 1

In all of the following examples, the name before the colon is the nameof the user executing the statement.

The following GRANT statements are issued by users U1, U2, and U3 and are the context for the examples that follow:

Statement 1:

```
U1: GRANT SELECT ON TABLE T1 TO U2 WITH GRANT OPTION;
```

Statement 2:

```
U2: GRANT SELECT ON TABLE T1 TO U3 WITH GRANT OPTION;
```

Statement 3:

```
U3: GRANT SELECT ON TABLE T1 TO U4 WITH GRANT OPTION;
```

Example 1a:

The RESTRICT option causes the following REVOKE statement to fail because in Statement 2, user U2 exercised the privilege he acquired in Statement 1.

```
U1: REVOKE SELECT ON TABLE T1 FROM U2 RESTRICT;
```

Example 1b:

The following example succeeds and Statements 1, 2, and 3 are negated.

```
U1: REVOKE SELECT ON TABLE T1 FROM U2 CASCADE;
```

Example 1c:

The RESTRICT option causes the following statement to fail because in Statement 2, user U2 exercised the GRANT OPTION privilege he acquired in Statement 1.

```
U1: REVOKE GRANT OPTION FOR SELECT ON TABLE T1 FROM U2 RESTRICT;
```

Example 1d:

The following statement succeeds and negates Statements 2 and 3. U2 retains SELECT privilege on T1, but cannot grant this privilege to others.

```
U1: REVOKE GRANT OPTION FOR SELECT ON TABLE T1 FROM U2 CASCADE;
```

Example 2

The following GRANT and CREATE statements are issued by users U1, U2, and U3 and are the context for the examples that follow. The name before the colon is the name of the user who issued the statement.

Statement 1:

```
U1: GRANT SELECT ON TABLE T1 TO U2 WITH GRANT OPTION;
```

Statement 2:

```
U2: GRANT SELECT ON TABLE T1 TO U3 WITH GRANT OPTION;
```

Statement 3:

```
U3: GRANT SELECT ON TABLE T1 TO U4 WITH GRANT OPTION;
```

Statement 4:

```
U2: CREATE VIEW V2 AS SELECT A, B FROM T1;
```

Statement 5:

```
U3: CREATE VIEW V3 AS SELECT A, B FROM T1;
```

Example 2a:

The following statement succeeds and negates Statements 1, 2, and 3. In addition, views V2 and V3 are dropped because U2 and U3 no longer have the SELECT privileges on T1 that are required by the views.

```
U1: REVOKE SELECT ON TABLE T1 FROM U2 CASCADE
```

Example 2b:

The following statement succeeds and negates Statements 2 and 3. User U2 retains the SELECT privilege on T1, but cannot grant this privilege to others. In addition, the view V3 is dropped because U3 no longer has the SELECT privilege on T1. View V2 is not dropped because U2 still holds SELECT privileges on T1.

```
U1: REVOKE GRANT OPTION FOR SELECT ON TABLE T1 FROM U2 CASCADE
```

Example 3

The following GRANT and CREATE statements are issued by users U1, U2, and U3 and are the context for the examples that follow. The name before the colon is the name of the user who issued the statement.

Statement 1:

```
U1: CREATE ROLE R1;
```

Statement 2:

```
U1: GRANT SELECT ON TABLE T1 TO R1;
```

Statement 3:

```
U1: GRANT R1 TO U2 WITH ADMIN OPTION;
```

Statement 4:

```
U2: GRANT R1 TO U3 WITH ADMIN OPTION;
```

Statement 5:

```
U3: GRANT R1 TO U4 WITH ADMIN OPTION;
```

Example 3a:

The following statement fails because user U2 has exercised the privileges acquired as a result of being granted role R1.

```
U1: REVOKE R1 FROM U2 RESTRICT;
```

Example 3b:

The following statement succeeds. Statements 3, 4, and 5 above are negated.

```
U1: REVOKE R1 FROM U2 CASCADE;
```

Example 3c:

The following statement fails because in Statement 3, user U2 exercised the `ADMIN OPTION`.

```
U1: REVOKE ADMIN OPTION FOR R1 FROM U2 RESTRICT;
```

Example 3d:

The following statement succeeds. Statements 4 and 5 are negated. U2 retains the privileges granted by role R1, but cannot grant this role to others.

```
U1: REVOKE ADMIN OPTION FOR R1 FROM U2 CASCADE;
```

Escape Syntax

Blackfish SQL supports escape sequences for the following:

- Date and time literals
- `OUTER JOIN`
- The escape character for a `LIKE` clause
- Calling stored procedures

Escapes must always be enclosed in braces {}. They are used to extend the functionality of SQL.

Date and Time Literals

`{T 'hh:mm:ss'}`	Specifies a time, which must be entered in the sequence: hours, followed by minutes, followed by seconds.
`{D 'yyyy-mm-dd}`	Specifies a date, which must be entered in the sequence; year, followed by month, followed by day.
`{TS 'yyyy-mm-dd hh:mm:ss'}`	Specifies a timestamp, which must be entered in the format indicated; year, month, day, hour, minute, second.

Examples

```
INSERT INTO tablename VALUES({D '2004-2-3'}, {T '2:55:11'});
SELECT {T '10:24'} FROM tablename;
SELECT {D '2000-02-01'} FROM tablename;
SELECT {TS '2000-02-01 10:24:32'} FROM tablename;
```

Outer Joins

`{OJ <join_table_expression>}`	An outer join is performed on the specified table expression.

Example

```
SELECT * FROM {OJ a LEFT JOIN b USING(id)};
```

Escape Character for LIKE

`{ESCAPE <char>}`	The specified character becomes the escape character in the preceding `LIKE` clause.

Example

```
SELECT * FROM a WHERE name LIKE '%*%' {ESCAPE '*'}
```

Calling Stored Procedures

```
{call <procedure_name> (<argument_list>)}
```

Or, if the procedure returns a result parameter:

```
{? = call <procedure_name> (<argument_list>)}
```

Example 1

The method implementing IncreaseSalaries updates the salaries table with an increase of some percentage for all employees. A connection object is passed implicitly to the method. An `updateCount` of all the rows affected by `IncreaseSalaries` is returned from `Statement.executeUpdate`.

```
{CALL IncreaseSalaries(10)};
```

Example 2

The parameter marker indicates an output parameter position from which the result of the stored procedure can be retrieved.

```
{?=CALL ABS(-765)};
```

Escape Functions

Functions are written in the following format, where `FN` indicates that the function following it should be performed:

```
{fn <function_name>(<argument_list>) }
```

Numeric functions

Function name	Function returns
`ABS(number)`	Absolute value of `number`
`ACOS(float)`	Arccosine, in radians, of `float`
`ASIN(float)`	Arcsine, in radians, of `float`

ATAN(float)	Arctangent, in radians, of float
ATAN2(float1, float2)	Arctangent, in radians, of float2 divided by float1
CEILING(number)	Smallest integer >= number
COS(float)	Cosine of float radians
COT(float)	Cotangent of float radians
DEGREES(number)	Degrees in number radians
EXP(float)	Exponential function of float
FLOOR(number)	Largest integer <= number
LOG(float)	Base e logarithm of float
LOG10(float)	Base 10 logarithm of float
MOD(integer1, integer2)	Remainder for integer1 divided by integer2
PI()	The constant pi
POWER(number, power)	number raised to (integer) power
RADIANS(number)	Radians in number degrees
RAND(integer)	Random floating point for seed integer
ROUND(number, places)	number rounded to places places
SIGN(number)	–1 to indicate number is < 0; 0 to indicate number is = 0; 1 to indicate number is > 0
SIN(float)	Sine of float radians
SQRT(float)	Square root of float
TAN(float)	Tangent of float radians
TRUNCATE(number, places)	number truncated to places places

String Functions

Function name	Function returns
ASCII(string)	Integer representing the ASCII code value of the leftmost character in string
CHAR(code)	Character with ASCII code value code, where code is between 0 and 255
CONCAT(string1, string2)	Character string formed by appending string2 to string1; if a string is null, the result is DBMS-dependent

DIFFERENCE(string1, string2)	Integer indicating the difference between the values returned by the function SOUNDEX for string1 and string2
INSERT(string1, start, length, string2)	A character string formed by deleting length characters from string1 beginning at start, and inserting string2 into string1 at start
LCASE(string)	Converts all uppercase characters in string to lowercase
LEFT(string, count)	The count leftmost characters from string
LENGTH(string)	Number of characters in string, excluding trailing blanks
LOCATE(string1, string2[, start])	Position in string2 of the first occurrence of string1, searching from the beginning of string2; if start is specified, the search begins from position start. Returns zero if string2 does not contain string1. Position 1 is the first character in string2.
LTRIM(string)	Characters of string with leading blank spaces removed
REPEAT(string, count)	A character string formed by repeating stringcount times
REPLACE(string1, string2, string3)	Replaces all occurrences of string2 in string1 with string3
RIGHT(string, count)	The count rightmost characters in string
RTRIM(string)	The characters of string with no trailing blanks
SOUNDEX(string)	A data source-dependent character string representing the sound of the words in string; this can be, for example, a four-digit SOUNDEX code or a phonetic representation of each word.
SPACE(count)	A character string consisting of count spaces
SUBSTRING(string, start, length)	A character string formed by extracting length characters from string beginning at start
UCASE(string)	Converts all lowercase characters in string to uppercase

Examples

```
SELECT {FN LCASE('Hello')} FROM tablename;
SELECT {FN UCASE('Hello')} FROM tablename;
SELECT {FN LOCATE('xx', '1xx2')} FROM tablename;
SELECT {FN LTRIM('Hello')} FROM tablename;
SELECT {FN RTRIM('Hello')} FROM tablename;
SELECT {FN SUBSTRING('Hello', 3, 2)} FROM tablename;
SELECT {FN CONCAT('Hello ', 'there.')} FROM tablename;
```

Date and Time Functions

Function name	Function returns
CURDATE()	The current date as a date value
CURTIME()	The current local time as a time value
DAYNAME(date)	A character string representing the day component of date; the name for the day is specific to the data source
DAYOFMONTH(date)	An integer from 1 to 31 representing the day of the month in date
DAYOFWEEK(date)	An integer from 1 to 7 representing the day of the week in date; Sunday = 1
DAYOFYEAR(date)	An integer from 1 to 366 representing the day of the year in date
HOUR(time)	An integer from 0 to 23 representing the hour component of time
MINUTE(time)	An integer from 0 to 59 representing the minute component of time
MONTH(date)	An integer from 1 to 12 representing the month component of date
MONTHNAME(date)	A character string representing the month component of date; the name for the month is specific to the data source
NOW()	A timestamp value representing the current date and time
QUARTER(date)	An integer from 1 to 4 representing the quarter in date; January 1 through March 31 = 1
SECOND(time)	An integer from 0 to 59 representing the second component of time
TIMESTAMPADD(interval, count, timestamp)	A timestamp calculated by adding count number of intervals to timestamp; interval can be any one of the following: SQL_TSI_FRAC_SECOND, SQL_TSI_SECOND, SQL_TSI_MINUTE, SQL_TSI_HOUR, SQL_TSI_DAY, SQL_TSI_WEEK, SQL_TSI_MONTH, SQL_TSI_QUARTER, or SQL_TSI_YEAR
TIMESTAMPDIFF(interval, timestamp1, timestamp2)	An integer representing the number of intervals by which timestamp2 is greater than timestamp1; interval can be any one of the following: SQL_TSI_FRAC_SECOND, SQL_TSI_SECOND, SQL_TSI_MINUTE, SQL_TSI_HOUR, SQL_TSI_DAY, SQL_TSI_WEEK, SQL_TSI_MONTH, SQL_TSI_QUARTER, or SQL_TSI_YEAR
WEEK(date)	An integer from 1 to 53 representing the week of the year in date
YEAR(date)	An integer representing the year component of date

Examples

```
SELECT {FN NOW()} FROM tablename;
SELECT {FN CURDATE() } FROM tablename;
SELECT {FN CURTIME() } FROM tablename;
SELECT {FN DAYOFMONTH(datecol) } FROM tablename;
SELECT {FN YEAR(datecol)} FROM tablename;
SELECT {FN MONTH(datecol)} FROM tablename;
SELECT {FN HOUR(timecol) } FROM tablename;
SELECT {FN MINUTE(timecol) } FROM tablename;
SELECT {FN SECOND(timecol) } FROM tablename;
```

System Functions

Function name	Function returns
`DATABASE()`	Name of the database
`IFNULL(expression, value)`	`value` if `expression` is null; `expression` if `expression` is not null
`USER()`	User name in the DBMS

Conversion Functions

Function name	Function returns
`CONVERT(value, SQLtype)`	`value` converted to `SQLtype` where `SQLtype` can be one of the following SQL types: `BIGINT`, `BINARY`, `BIT`, `CHAR`, `DATE`, `DECIMAL`, `DOUBLE`, `FLOAT`, `INTEGER`, `LONGVARBINARY`, `LONGVARCHAR`, `REAL`, `SMALLINT`, `TIME`, `TIMESTAMP`, `TINYINT`, `VARBINARY`, or `VARCHAR`

Example

```
SELECT {FN CONVERT('34.5',DECIMAL(4,2))} FROM tablename;
```

ISQL

ISQL is a SQL command interpreter that can be used to execute SQL statements interactively. This feature is currently available for Blackfish SQL for Java only.

Getting Help

To see a help display for Blackfish SQL ISQL, issue one of the following help commands:

From the system prompt:

- `isql -?` displays ISQL startup options.
- `isql -help` displays ISQL options.

From the SQL prompt:

- `HELP CREATE` displays help on creating datasources.`HELP SHOW` displays a list of `SHOW` commands with briefdescriptions.
- `HELP SET` displays a list of `SET` commands with brief descriptions of each.

Starting ISQL

To start Blackfish SQL ISQL, either ensure that *<BlackfishSQL_install_dir>*`\bin` is in your system path, or go to that directory to issue the `ISQL` command. These options are available:

Startup Options for ISQL

Option and arguments	Description
`-user` *`userName`*	Specifies the userName for this connection.
`-password` *`password`*	Specifies the password associated with `userName`.
`-role` *`roleName`*	Activates the named role for the user.
`-input` *`filename`*	Executes all commands in the specified file and then quits.
`-output` *`filename`*	Redirects all output to the named file.
`-datasource` *`filename`*	Specifies an alternative datasource file
`-echo`	Prints all commands before executing them.
`-stacktrace`	Prints a stacktrace for each error encountered.
`-pagelength length`	Prints column headers every `length` number of rows.
`-x`	Prints all the data definition statements from the current connection and exits.
`-z`	Shows version information and exits.

Datasource and File Management

Once you have started ISQL, the following commands are available for managing datasource connections, file management and session management. You can see a list of these commands during an ISQL session by issuing the following:

```
SHOW CREATE;
```

There are two additional groups of commands that are discussed later in this section: SHOW commands and SET commands. The SQL commands that are available for data definition, data manipulation, security, and transaction management are discussed throughout this chapter.

ISQL Datasource and File Management Commands

Command	Description
`CREATE DATASOURCE dataSourceName [dataSourceClassName] properties`	Associates a datasource with the `dataSourceName`. You pass this `dataSourceName` to `CONNECT` in order to connect to a database. See "Creating datasources with ISQL" below, for information on creating datasources in ISQL.
`CONNECT dataSourceName [userpassword]`	Connects to the datasource specified by *dataSourceName*. Before you can use `CONNECT`, you must use `CREATE DATASOURCE` to associate a database with the `dataSourceName` that you pass to `CONNECT`. You do not need to specify user name or password if it was specified as part of the `CREATE DATASOURCE` statement.
`INPUT filename`	Takes the contents of the named SQL file as input.
`OUTPUT filename`	Writes the output to the specified file.
`OUTPUT`	Writes the output to stdout.
`EXPORT`	Exports the data definition statements and data of the current database to SQL.

EXPORT [*userpassword*]	Exports the data definition statements and data of the current database to the specified datasource.
	
	To export to a file, use EXPORT in conjunction with OUTPUT:
	
	`OUTPUT sqlfile.txt;`
	`EXPORT;`

IMPORT [*userpassword*]	Imports the data definition statements and data from the specified datasource.
VERSION	Shows the version of ISQL and the version of any connected database.
EXIT	Commits changes and exits.
QUIT	Rolls back changes and exits.

Creating Datasources with ISQL

This section provides more detail about creating datasources in ISQL using the CREATE DATASOURCE command listed above. The CREATE DATASOURCE syntax is as follows:

```
CREATE DATASOURCE dataSourceName [dataSourceClassName] properties
```

The arguments for the CREATE DATASOURCE command are:

dataSourceName identifies the new datasource; it can be any SQL identifier assigned by you.

dataSourceClassName is the class that specifies the properties needed to connect to a JDBC database. It must be an implementation of the standard JDBC javax.sql.DataSource interface. If this argument is not provided, com.borland.javax.sql.JdbcDataSource is used. To access InterBase databases, you can use interbase.interclient.DataSource.

properties can include any properties in the class supplied as the *dataSourceClassName*. Properties are separated by commas and commonly include the following:

- user='*username*' If you do not supply a user name here, you can supply it as part of the CONNECT statement.

- `password='password'` If you do not supply a password here, you can supply it as part of the `CONNECT` statement.
- `databaseName='database_name_to_connect_to'`

Example

You can supply values for any properties in the datasource class. For example, to create a new database, add the following:

```
CREATE=true: CREATE DATASOURCE
JDS user=SYSDBA, password=masterkey,
databaseName='c:/databases/test.jds',CREATE=true';
```

Examples

These examples both use the Blackfish SQL default class `com.borland.javax.sql.JdbcDataSource`, since no `className` is specified.

The example below creates a local datasource, JDS_LOCAL:

```
CREATE DATASOURCE JDS_LOCAL
  user=SYSDBA,
  password=masterkey,
  create=true,
  databaseName='c:/test.jds';
```

The next example creates a remote datasource, JDS_REMOTE. It also creates the test.jds database.

```
CREATE DATASOURCE JDS_REMOTE
  user=SYSDBA,
  password=masterkey,
  networkProtocol=tcp,
  serverName=localhost,
  portNumber=2508,
  create=true,
  databaseName='c:/test.jds';
```

ISQL SHOW Commands

Command	Description
SHOW DATASOURCE [*name*]	Displays all datasources or the specified datasource.
SHOW DATABASE	Displays settings for the current database.
SHOW VERSION	Displays the ISQL version and the version of any connected database.
SHOW DDL	Displays the data definition statements for the current database.
SHOW SYSTEM	Displays the system tables.
SHOW TABLE [[*schema.*]*table*]	Displays all tables or the specified table.
SHOW VIEW [[*schema.*]*view*]	Displays all views or the specified view.
SHOW PROCEDURE [[*schema.*]*name*]	Displays all procedures or the specified pprocedures
SHOW FUNCTION [[*schema.*]*name*]	Displays all functions or the specified function.
SHOW INDEX [*index* [ON [*schema.*]*table*]]	Displays all indexes or the specified index.
SHOW ROLES	Lists all roles defined in the database.
SHOW USERS	Lists all users defined in the database.
SHOW GRANT TABLE [[*schema.*]*table*]	Lists all privileges on tables that have been granted WITH GRANT OPTION.
SHOW GRANT VIEW [[*schema.*]*view*]	Lists all privileges on views that have been granted WITH GRANT OPTION.
SHOW GRANT PROCEDURE [[*schema.*]*name*]	Lists all privileges on procedures that have been granted WITH GRANT OPTION.
SHOW GRANT FUNCTION [[*schema.*]*name*]	Lists all privileges on functions that have been granted WITH GRANT OPTION.
SHOW GRANT ROLE [*role*]	Lists all users who have been granted the specified role.
SHOW GRANT DATABASE [*user*|*role*]	Lists all database privileges that have been granted to the specified user or role.

ISQL SET Commands

Command	Description
SET	Displays the current value of ECHO, STACKTRACE, and PAGELENGTH.
SET ECHO {ON|OFF}	Toggles echoing of all commands to standard out.
SET STACKTRACE {ON|OFF}	Toggles display of error traces.
SET PAGELENGTH *number*	Sets the page length in lines; default is 0, meaning that the column headings print out only once.

Chapter 10:
Optimizing Blackfish SQL Applications

This section discusses ways to improve the performance, reliability, and size of Blackfish SQL applications. Unless otherwise specified, `DataStoreConnection` refers to either a `DataStoreConnection` or `DataStore` object used to open a connection to a Blackfish SQL database file.

Loading Databases Quickly

Here are some tips that can improve the performance of your application when loading databases:

- Use prepared statements or commands whenever possible. If the number of parameters changes from one insert to the next, clear the parameters before setting the new parameters.
- Create the table without primary keys, foreign keys, or secondary indexes. Load the table and then create any needed primary keys, foreign keys, or secondary indexes.

Java-specific Database Loading Optimizations

Use the DataExpress `TextDataFile` class to import text files. It has a fast parser and can load data quickly. You must set the `StorageDataSet` store to a `DataStoreConnection,` and set the `StoreName` property to the name of your table in the Blackfish SQL database. When loading a new database:

1. First create the database as non-transactional.
2. While the database is non-transactional, use a DataExpress `StorageDataSet.addRow` or `TextDataFile` component to load the database.
3. After the database has loaded, use the `DataStore.TxManager` property to make the database transactional.

This technique should enable the database to load two to three times faster.

General Recommendations

This section provides some general guidelines for improving performance for Blackfish SQL applications.

Proper Database Shutdown

If a database is not properly shut down, the next time a process opens the database, there will be a delay. This is because Blackfish SQL needs about 8 to 10 seconds to ensure that no other process has the database open. To ensure that a database is shut down properly, make sure all connections are closed when they are no longer needed. If it is difficult to ensure that all connections are closed, a connection with Administrator rights can call the `DB_ADMIN.CLOSE_OTHER_CONNECTIONS` built-in stored procedure to ensure that all other connections are closed.

Another benefit to closing connections is that when they are all closed, the memory allocated to the Blackfish SQL cache is released.

Currently, non-transactional databases can be accessed from SQL only if the database read-only property is true. However, DataExpress JavaBeans can perform write operations on a non-transactional database.

Closing a non-transactional Blackfish SQL database ensures that all modifications are saved to disk. There is a daemon thread for all open `DataStoreConnection` instances that is constantly saving modified cache data. (By default modified data is saved every 500 milliseconds.) If you directly exit the Java Virtual Machine without closing the database, the daemon thread might not have the opportunity to save the last set of changes. There is a small chance that a non-transactional Blackfish SQL could become corrupted.

A transactional Blackfish SQL database is guaranteed not to lose data, but the transaction manager rolls back any uncommitted changes.

Java-specific Database Shutdown

If your application is using DataExpress JavaBean components, close all `StorageDataSet`s that have the `store` property set to a `DataStoreConnection` when you are finished with them. This frees up Blackfish SQL resources associated with the `StorageDataSet` and allows the `StorageDataSet` to be garbage collected.

You can use the `DataStore.shutdown()` method to make sure all database connections are closed before an application terminates.

Optimizing the Blackfish SQL Disk Cache

The default maximum cache size for a Blackfish SQL database is 512 cache blocks. The default block size is 4096 bytes. Therefore, the cache memory reaches its maximum capacity at approximately 512*4096 (2MB). Note that this memory is allocated as needed. In some rare situations when all blocks are in use, the cache may grow beyond 512 cache blocks. You can use the `DataStore.MinCacheSize` property to specify the minimum cache size.

NOTE: Do not arbitrarily change the database cache size. Be sure first to verify that doing so will improve the performance of your application.

Keep in mind the following considerations when changing the Blackfish SQL cache size:

- Modern OS caches are typically high performance. In many cases, increasing the Blackfish SQL cache size does not significantly improve performance, and simply uses more memory.
- There is only one Blackfish SQL disk cache for all Blackfish SQL databases open in the same process. When all Blackfish SQL databases are shut down, the memory for this global disk cache is released.
- For handheld devices with small amounts of memory, set the `DataStore.MinCacheSize` property to a smaller number, such as 96.

Optimizing File Access

Blackfish SQL databases perform the majority of read/write operations against the following four file types:

- The Blackfish SQL database file itself (filename extension is `.jds`) as specified by the `DataStore.FileName` property
- Blackfish SQL transactional log files (filename extension is `LOGAnnnnnnnnnn`, where `n` is a numeric digit) as specified by the `TxManager.ALogDir` property
- Temporary files used for large sort operations as specified by the `DataStore.TempDirName` property
- Temporary `.jds` files used for SQL query results as specified by the `DataStore.TempDirName` property

You can potentially improve performance by instructing Blackfish SQL to place the files mentioned above on different disk drives.

File Storage

The following are some guidelines for file storage handling that can improve performance of your applications:

- It is especially important to place the log files on a separate disk drive. Note that log files are generally appended in sequential order, and their contents must be forced to disk in order to complete commit operations. Consequently, it is advantageous to have a disk drive that can complete write operations quickly.
- On Win32 platforms, performance can be improved by placing Blackfish SQL log files in a separate directory. Storing numerous files other than the log files in the log file directory can slow down the performance of commit operations. This performance tip may also apply to platforms other than Windows NT/2000/XP.
- Remember to defragment your disk drive file systems on a regular basis. This practice is especially important for the disk drive that stores the log files, because Blackfish SQL performs

many sequential read/write operations to this file.
- For Win32 platforms, consider using a FAT32 file system with a large cluster size such as 64KB for the disk drive to which your log files are written.

Non-transactional Database Disk Cache Write Options for Java

Note: This section applies only to Blackfish SQL for Java, which uses the DataExpress JavaBean components.

Use the `saveMode` property of the `DataStore` component to control how often cache blocks are written to disk. This property applies only to non-transactional Blackfish SQL databases. The following are valid values for the method:

0	Let the daemon thread handle all cache writes. This setting gives the highest performance but the greatest risk of corruption.
1	Save immediately when blocks are added or deleted; let the daemon thread handle all other changes. This is the default mode. Performance is almost as good as with `saveMode(0)`.
2	Save all changes immediately. Use this setting whenever you debug an application that uses a `DataStore` component.

Unlike other properties of `DataStore`, `saveMode` can be changed when the connection is open. For example, if you are using a `DataStoreConnection`, you can access the value through the `dataStore` property:

```
DataStoreConnection store = new DataStoreConnection();
...
store.getDataStore().setSaveMode(2);
```

Note that this changes the behavior for all `DataStoreConnection` objects that access that particular Blackfish SQL database file.

Tuning Memory

You can tune the use of memory in a number of ways. Be aware that asking for too much memory can be as bad as having too little.

- Try increasing the `ConnectionProperties.MinCacheBlocks` property, which controls the minimum number of blocks that are cached.
- The `ConnectionProperties.MaxSortBuffer` property controls the maximum size of the buffer used for in-memory sorts. Sorts that exceed this buffer size use a slower disk-based sort.

The Java heap tends to resist growing beyond its initial size, forcing frequent garbage collection with an ever-smaller amount of free heap. Use the JVM `-Xms` option to specify a larger initial heap size. It is often beneficial to make the JVM `-Xms` and `-Xmx` settings equal.

Miscellaneous Performance Tips

Here are some tips that can help performance:

- Setting the `ConnectionProperties.TempDirName` property, used by the query engine, to a directory on another (fast) disk drive can often help.
- Try changing the check point frequency for the Transaction Manager. A higher value can improve performance, but might result in slower crash recovery. This can be updated from SQL by using the `DB_ADMIN.ALTER_DATABASE` built-in stored procedure. In Blackfish SQL for Java, you can use JdsExplorer to set this property by choosing `TxManager > Modify`.

Optimizing Transactional Applications

The increased reliability and flexibility you gain from using transactional Blackfish SQL databases comes at the price of some performance. You can reduce this cost in several ways, as described in the following sections.

Using Read-only Transactions

For transactions that are reading but not writing, significant performance improvements can be realized by using a read-only transaction. The connection `readOnly` property controls whether a transaction is read-only. The Blackfish SQL for Java `DataStoreConnection` JavaBean has a `readOnlyTx` property to enable read-only transactions.

Read-only transactions work by simulating a snapshot of the Blackfish SQL database. This snapshot sees only data from transactions that were committed at the point the read-only transaction starts. This snapshot is created when the `DataStoreConnection` opens, and it refreshes every time a `commit` method is called.

Another benefit of read-only transactions is that they are not blocked by writers or other readers. Both reading and writing usually require a lock. But because a read-only transaction uses a snapshot, it does not require any locks.

You can further optimize the application by specifying a value for the property `readOnlyTxDelay`. The `readOnlyTxDelay` property specifies the maximum age (in milliseconds) for an existing snapshot that the connection can share. When the property is non-zero, existing snapshots are searched from most recent to oldest. If there is one that is under `readOnlyTxDelay` in age, it is used and no new snapshot is taken. By default, this property

is set to 5000 milliseconds.

Using Soft Commit Mode

If you enable soft commit mode through the `SoftCommit` property, the transaction manager still writes log records for committed transactions, but does not use a synchronous write mechanism for commit operations. With soft commit enabled, the operating system cache can buffer file writes from committed transactions. Typically the operating system ends up writing dirty cache blocks to disk within seconds. Soft commit improves performance, but cannot guarantee the durability of the most recently committed transactions. You can set the `SoftCommit` property by calling the `DB_ADMIN.ALTER_DATABASE` built-in stored procedure.

Disabling Status Logging for Transaction Log Files

You can improve performance by disabling the logging of status messages. To do this, set the `RecordStatus` property to `false`. You can set the `RecordStatus` property by calling the `DB_ADMIN.ALTER_DATABASE` built-in stored procedure.

Tuning Blackfish SQL Concurrency Control Performance

The following are guidelines for optimizing the performance of Blackfish SQL concurrency control operations:

- Choose the weakest isolation level with which your application can function properly. Lower isolations tend to acquire fewer and weaker locks.
- Batch multiple statements into a single transaction. Connections default to autocommit mode commit after every statement execution.
- Commit transactions as soon as possible. Most locks are not released until a transaction is committed or rolled back.
- Reuse statement or command objects whenever possible, or better yet, use prepared statements or commands when possible.
- Close all statements or commands, all result sets or readers, and all connection objects when they are no longer needed. Single-directional result set or reader objects automatically close when the last row is read.
- Use read-only transactions for long-running reports or online backup operations. Use the `DB_ADMIN.COPYDATABASE` method for online backups. Read-only transactions provide a transactionally consistent (serializable), read-only view of the tables they access. They do not acquire locks, so lock timeouts and deadlocks are not possible. See the section Using Read-only Transactions.
- There is some overhead for maintaining a read-only view. Consequently, multiple transactions can share the same read-only view. The `ConnectionProperties.ReadOnlyTxDelay` property specifies how old the read-only view can be when a read-only transaction is started. Committing the transaction for a read-only connection refreshes the view of the database. Note that a read-only transaction uses the transactional log files to maintain views. Therefore, read-only connections should be closed as soon as they are no longer needed.

Using Multithreaded Operations

Write transaction throughput can increase as more threads are used to perform operations, because each thread can share in the overhead of commit operations through the “group commit” support provided by Blackfish SQL.

Pruning Deployed Resources for Blackfish SQL for Java Applications

When deploying a Blackfish SQL application, you can exclude certain classes and graphics files that are not used.

- If Blackfish SQL is used without the JDBC driver, exclude the following classes:
  - `com.borland.datastore.Sql*.class`
  - `com.borland.datastore.jdbc.*`
  - `com.borland.datastore.q2.*`
- If you are using DataExpress, and the `StorageDataSet.store` property is always set to an instance of `DataStore` or `DataStoreConnection`, exclude the following classes:
  - `com.borland.dx.memorystore.*`
- If `StorageDataSet` is used, but not `QueryDataSet`, `QueryProvider`, `StoredProcedureDataSet` or `StoredProcedureProvider`, exclude the following classes:
  - `com.borland.dx.sql.*`
- If DataExpress isn't using any visual components from the JBCL or dbSwing libraries, exclude the following classes:
  - `com.borland.dx.text.*`
- If `com.borland.dx.dataset.TextDataFile` is not used, exclude the following classes:
  - `com.borland.jb.io.*`
  - `com.borland.dx.dataset.TextDataFile.class`
  - `com.borland.dx.dataset.SchemaFile.class`

AutoIncrement Columns

You can specify columns of type `int` and `long` as having `AutoIncrement` values.

The following attributes apply to all `AutoIncrement` column values:

- They are always unique
- They can never be null
- Values from deleted rows can never be reused

These attributes make `AutoIncrement` columns ideal for single column `integer/long`

primary keys.

An `AutoIncrement` column is the internal row identifier for a row, and so provides the fastest random access path to a particular row in a Blackfish SQL table.

Each table can have only one `AutoIncrement` column. Using an `AutoIncrement` column saves the space of one integer column and one secondary index in your table if you use it as a replacement for your primary key. The Blackfish SQL Query Optimizer optimizes queries that reference an `AutoIncrement` column in a `WHERE` clause. For instructions on using `AutoIncrement` columns with SQL, see "Using AutoIncrement Columns with SQL" in the SQL Reference.

AutoIncrement Columns Using Blackfish SQL for Java DataExpress JavaBeans

To create a table with an `AutoIncrement` column using DataExpress, set the `Column.AutoIncrement` property to `true` before opening a table. If you are modifying an existing table, you need to call the `StorageDataSet.restructure()` method.

Blackfish SQL Companion Components

The dbSwing component library provides two components (on the **More dbSwing** page of the **Component Palette**) that make it easier to produce robust Blackfish SQL applications.

- `DBDisposeMonitor` automatically disposes of data-aware component resources when a container is closed. It has a `closeDataStores` property. When `true` (the default), it automatically closes any Blackfish SQL databases that are attached to components it cleans.

  For example, if you drop a `DBDisposeMonitor` into a `JFrame` that contains dbSwing components attached to a Blackfish SQL database, when you close the `JFrame`, `DBDisposeMonitor` automatically closes the Blackfish SQL database for you. This component is particularly handy when building simple applications to experiment using Blackfish SQL.
- `DBExceptionHandler` has an **Exit** button. You can hide it with a property setting, but it is visible by default. Clicking this button automatically closes any open Blackfish SQL database files it can find. `DBExceptionHandler` is the default dialog box displayed by dbSwing components when an exception occurs.

Using Data Modules for Blackfish SQL for Java DataExpress JavaBean Components

When using a Blackfish SQL table with a `StorageDataSet`, you should consider grouping them all inside data modules. Make any references to these `StorageDataSet`s through `DataModule` accessor methods, such as `businessModule.getCustomer`. You should do

this because much of the functionality surfaced through `StorageDataSet`s is driven by property and event settings.

Although most of the important structural `StorageDataSet` properties are persisted in the Blackfish SQL table itself, the classes that implement the event listener interfaces are not. Instantiating the `StorageDataSet` with all event listener settings, constraints, calculated fields, and filters implemented with events, ensures that they are properly maintained at both run time and design time.

Chapter 11:
Deploying Blackfish SQL Database Applications

When you have finished developing your application, the next step is to deploy it. Deployment involves licensing considerations and determining which Blackfish SQL files are needed for distribution. This chapter discusses the Blackfish SQL distribution files. For information regarding deployment licensing, please contact Customer Support.

Deploying Blackfish SQL for Windows Applications

The specific files you will need to distribute depend upon the type of application you have written. The table below provides some guidelines for determining which files to distribute.

Blackfish SQL for Windows Application Deployment Files

File name	Description

`DbxClientDriver100.bpl`
`DbxCommonDriver100.bpl`
`DbxReadOnlyMetaData100.bpl`
	Remote dbExpress driver for Win32 applications that use packages. If you are not using packages, then the driver is linked into the application exe file.

`Borland.Data.DbxClientDriver.dll`
`Borland.Data.DbxCommonDriver.dll`
`Borland.Data.DbxReadOnlyMetaData.dll`
	Remote dbExpress driver for .NET applications that use assemblies. If you are not using assemblies, then the driver is linked into the application exe file.

`Borland.Data.BlackfishSQL.LocalClient.dll`	.NET assembly containing the local ADO.NET provider and the database kernel itself. This is the only file needed for applications that only use the local ADO.NET provider.
`Borland.Data.BlackfishSQL.RemoteClient.dll`	.NET assembly containing the remote ADO.NET provider.
`BSQLServer.exe` `BSQLServer.exe.config`	Launchers required for launching the server at the command line or as a windows service. You must also deploy `Borland.Data.BlackfishSQL.LocalClient.dll` with these files.

Deploying Licenses for Blackfish SQL for Windows

Blackfish SQL for Windows searches the following locations for licensing SLIP files:

- The `blackfishsql.licenseDirectory` system property

Set this property by calling the `System.AppDomain.CurrentDomain.SetData` method, or by specifying a setting for this property in the `BSQLServer.exe.config` file.

- The directory of the server if you are using a remote driver
- The directory of the application `.exe` file that is using a local driver
- The GAC location of the assembly, if the local driver is in use and has been added to the GAC
- The `CodeGear` subdirectory of the `System.Environment.SpecialFolder.CommonApplicationData` folder
- `$(BDSCOMMONDIR)\license`

The `BDSCOMMONDIR` environment variable is set when BDS is installed.

Deploying Blackfish SQL for Java Applications

Blackfish SQL includes a number of different jar files. The specific files you will need to distribute depend upon the type of application you have written. The table below provides some guidelines for determining which files to distribute.

Blackfish SQL for Java Application Deployment JAR Files

File Name	Description
`dx.jar`	Local DataExpress database connectivity
`jds.jar`	Local JDBC database connectivity
`jdsremote.jar`	Remote JDBC thin client database connectivity
`jdsserver.jar`	Full runtime embedded database server for local and remote JDBC database connectivity

`jdshelp.jar`
`beandt.jar`
`dbtools.jar`
`jdsserver.jar`
	Required for JdsServer and JdsExplorer Graphical User Interfaces

`beandt.jar`	Required for compiling and for visual design of JavaBean components
`dbswing.jar`	Required for Swing-based user interfaces
`jbcl-awt.jar`	Required for AWT-based user interfaces
<BlackfishSQL_home>`/bin/JdsServer` Windows: `JdsServer.exe`	Launcher for the graphical interface to the Blackfish SQL server

`JdsServer.config`	Configuration file for the JdsServer launcher
<BlackfishSQL_home>`/bin/JdsExplorer` Windows: `JdsExplorer.exe`	Launcher for the graphical interface to JdsExplorer
`JdsExplorer.config`	Configuration file for the JdsExplorer launcher
<BlackfishSQL_home>`/doc/*`	Blackfish SQL help, accessible directly or through JdsExplorer and JdsServer

Deploying Licenses for Blackfish SQL for Java

Blackfish SQL for Java searches the following locations for licensing SLIP files:

- The `blackfishsql.licenseDirectory` system property

You can set this property by calling the `java.lang.System.setProperty` method. Specify this property with the `-D` option on the command line of the Java Virtual Machine:

```
-DblackfishSQL.licenseDirectory=/mylicenseDir
```

Alternatively, you can set this property in the `JdsServer.config` file by adding a `vmparam -D` statement:

```
vmparam -DblackfishSQL.licenseDirectory=/mylicenseDir
```

- The Java `user.home` system property
- All directories specified in the Java `classpath` setting

Chapter 12: Troubleshooting

This chapter presents some guidelines for troubleshooting and resolving possible error conditions and other issues that may arise when creating, maintaining, and accessing Blackfish SQL databases.

Relative Path Database Filenames

You can use the DataDirectory macro in the specification of database filenames to provide support for relative pathnames. For more information on the DataDirectory macro, see Establishing Connections.

If you do not use the DataDirectory macro, relative pathnames are relative to the current directory of the process in which Blackfish SQL is executing. On Java platforms, the `user.dir` property dictates how database filenames are resolved when a fully qualified path name is not specified. The Java Virtual Machine (JVM) defaults this property to the current working directory of the process. You can set this property with a JVM command line option. For example:

```
-Duser.dir=/myapplication
```

You can also set this property from within a Java application by using the `java.util.System.setProperty` method.

Enabling Blackfish SQL System Logging

System logging is performed for all connections and all databases accessed in the same process.

You can enable Blackfish SQL system logging in the following ways.

For the local Blackfish SQL client:

Set the `blackfishsql.logFile` system property to the name of the file to which the log output should be written. If you set this to `con`, the log output is displayed to the console. You can specify the types of operations to include in the log file by setting the `blackfishsql.logFilters` property.

For the remote Blackfish SQL client:

In the Blackfish SQL configuration file set the `blackfishsql.logFile` property to the name of the file to which the log output should be written. If you set this to `con`, the log output is displayed to the console. You can specify the types of operations to include in the log file by setting the `blackfishsql.logFilters` property.

Setting System Properties

All Blackfish SQL system properties are case sensitive and begin with the `blackfishsql.` prefix. The `SystemProperties` class has constant strings for all system properties.

For Windows system properties:

If your application uses the Blackfish SQL server, set system properties in the `BSQLServer.exe.config` file. If your application does not use the Blackfish SQL server, set system properties by calling the `System.AppDomain.CurrentDomain.SetData` method.

For Java system properties:

If your application uses the Blackfish SQL server, set system properties in the `BSQLServer.config` file by prefixing the property setting with `vmparam -D`. If your application does not use the Blackfish SQL server, set system properties by calling the `System.setProperty` method.

Blackfish SQL for Java JDBC Logging Options

For Blackfish SQL for Java, these are additional logging options:

- If you are using a `javax.sql.DataSource` implementation, call the `setLogWriter` method of the `DataSource` implementation. See `com.borland.javax.sql.JdbcDataSource` and `com.borland.javax.sql.JdbcConnectionPool`.
- Call the `java.sql.DriverManager.setLogStream` method.
- Call the `java.sql.DriverManager.setLogWriter` method.

Enabling Blackfish SQL Database Logging

Database logging output is performed on a per-database basis and is sent to the status log files for that database. The lifetime of status log files is managed in the same fashion as the transactional log files for the database. When a transactional log file is dropped, the corresponding status log file is dropped also. When you create a database, status logging is disabled by default. You can enable database status logging by calling the `DB_ADMIN.ALTER_DATABASE` built-in stored procedure. You can set the log filtering options for all connections to a database by calling the `DB_ADMIN.SET_DATABASE_STATUS_LOG_FILTER` built-in stored procedure. You can set the log filtering options for a single connection by setting the `logFilter` connection property or by calling the `DB_ADMIN.SET_STATUS_LOG_FILTER` built-in stored procedure.

Debugging Lock Timeouts and Deadlocks

Locks can fail due to lock timeouts or deadlocks. Lock timeouts occur when a connection waits to acquire a lock held by another transaction and that wait exceeds the milliseconds set in the `lockWaitTime` connection property. In such cases, an exception is thrown that identifies which connection encountered the timeout and which connection is currently holding the required lock. The transaction that encounters the lock timeout is not rolled back.

Blackfish SQL has automatic, high speed deadlock detection that should detect all deadlocks. An appropriate exception is thrown that identifies which connection encountered the deadlock, and the connection with which it is deadlocked. Unlike lock timeout exceptions, deadlock exceptions encountered by a `java.sql.Connection` cause that connection to automatically roll back its transaction. This behavior allows other connections to continue their work.

Use the following guidelines to detect timeouts and deadlocks:

- Read the exception message from the timeout or deadlock. The message has information on what tables and what connections are involved.
- Enable system or database logging. To restrict log output to lock-related issues, set the log filter options to `LOCK_ERRORS`.
- Use the `DB_ADMIN.GET_LOCKS` built-in stored procedure to report locks held by all connections.

Avoiding Blocks and Deadlocks

A connection usually requires a lock when it either reads from or writes to a table stream or row. It can be blocked by another connection that is reading or writing. You can prevent blocks in two ways:

- Minimize the life span of transactions that write.
- Use read-only transactions, since these do not require locks to read.

Using Short Duration Write Transactions

Connections should use short-duration transactions in high concurrency environments. However, in low- or no-concurrency environments, a long-duration transaction can provide better throughput, since fewer commit requests are made. There is a significant overhead to the commit operation because it

must guarantee the durability of a transaction.

Using Read-only Transactions

Read-only transactions are not blocked by writers or other readers, and since they do not acquire locks, they never block other transactions.

Setting the `readOnlyTx` connection property to true causes a connection to use read only connections. Note that there is also a `readOnly` connection property, which is very different from the `readOnlyTx` connection property. The `readOnly` connection property causes the database file to be open in read only mode, preventing any other connections from writing to the database.

For Blackfish SQL for Java JDBC connections you can also enable read-only transactions by setting the `readOnly` property of the `java.sql.Connection` object or the `com.borland.dx.sql.dataset.Database.getJdbcConnection` methods to `true`. When using Blackfish SQL for Java `DataStoreConnection` objects, set the `readOnlyTx` property to `true` before opening the connection.

Read-only transactions work by simulating a snapshot of the Blackfish SQL database. The snapshot sees only data from transactions that are committed at the point the read-only transaction starts; otherwise, the connection would have to check if there were pending changes and roll them back whenever it accessed the data. A snapshot begins when the connection opens. The snapshot is refreshed each time the `commit` method is called.

Verifying the Integrity of a Blackfish SQL Database

If you suspect that cache contents were not properly saved on a non-transactional Blackfish SQL database, you can verify the integrity of the file by calling the `DB_ADMIN.VERIFY` built-in stored procedure.

Note that transactional Blackfish SQL databases have automatic crash recovery when they are opened. Under normal circumstances, Blackfish SQL databases do not require verification.

Troubleshooting Blackfish SQL for Java

This section provides more Java-specific troubleshooting guidelines.

Debugging Triggers and Stored Procedures

The approach to debugging triggers and stored procedures depends on whether your application uses the local or remote JDBC driver.

If your application uses the local JDBC driver, there is nothing special to set up, since the database engine is executing in the same process as your application.

If your application uses the remote JDBC driver, you can use either of the following procedures.

Using the DataStoreServer JavaBean for debugging:

In your application, instantiate a `com.borland.datastore.jdbc.DataStoreServer` JavaBean component and execute its `start` method.

Using the JdsServer for debugging:

Complete the following steps:

1. Add the following lines to your `<jds_home>/bin/JdsServer.config` file:
   ```
   vmparam -Xdebug
   vmparam -Xnoagent
   vmparam -Djava.compiler=NONE
   vmparam -Xrunjdwp:transport=dt_socket,server=y,
   address=5000,suspend=y
   ```
2. Execute the JdsServer. The server will not come up until a remote debugger (such as the JBuilder debugger) is launched to attach to the JdsServer process on port 5000.

Accessing and Creating Tables from SQL and DataExpress JavaBeans

Creating an SQL table forces unquoted identifiers to be uppercase. You must quote the identifiers to enable case sensitivity. See “Identifiers” in the SQL Reference.

When you use DataExpress components to create a table, the table and column names are case sensitive. If you specify these identifiers in lowercase or mixed case, SQL is not able to access them unless the identifiers are quoted.

When you use DataExpress to access a table, the `StorageDataSet` `storeName` property is case sensitive. However, the column identifiers can be referenced in a case-insensitive fashion. Consequently, for DataExpress, you can access an `address` column by using `ADDRESS` or `address`.

The simplest way to avoid problems with identifiers for both SQL and DataExpress components is to always use uppercase identifiers when your application creates or accesses tables.

Debugging Non-transactional Database Applications

Set the `saveMode` property to `2` when you are debugging an application that uses a non-transactional Blackfish SQL database. The debugger stops all threads when you are single-stepping

through code or when breakpoints are hit. If you do not set the `saveMode` property to `2`, the Blackfish SQL daemon thread cannot save modified cache data. For more information, see "Non-transactional Database Disk Cache Write Options" in Optimizing Blackfish SQL Applications.

Resolving Problems with Locating and Ordering Data

Sun Microsystems makes changes to the `java.text.CollationKey` classes from time to time as it corrects problems. The secondary indices for tables stored inside a Blackfish SQL database use these `CollationKey` classes to generate sortable keys for non-US locales. When Sun changes the format of these `CollationKeys` classes, the secondary indexes created by an older Sun JDK may not work properly with a new Sun JDK. The problems resulting from such a situation manifest themselves in the following ways:

- Locate and query operations might not find records that they should find.
- A table viewed in secondary index order (by setting the `StorageDataSet.sort` property) might not be ordered properly.

Currently, the only way to correct this is to drop the secondary indices and rebuild them with the current JDK. The `StorageDataSet.restructure()` method also drops all the secondary indexes.