Developer's Guide

JDataStore™ 7

Borland®
Excellence Endures™

Borland Software Corporation
100 Enterprise Way
Scotts Valley, California 95066-3249
www.borland.com

Refer to the file `deploy.html` located in the `redist` directory of your JBuilder product for a complete list of files that you can distribute in accordance with the JBuilder License Statement and Limited Warranty.

Borland Software Corporation may have patents and/or pending patent applications covering subject matter in this document. Please refer to the product CD or the About dialog box for the list of applicable patents. The furnishing of this document does not give you any license to these patents.

For third-party conditions and disclaimers, see the Release Notes on your JBuilder product CD.

Printed in the U.S.A.

DG7 7E10R0304
0405060708-9 8 7 6 5 4 3 2 1
PDF

Contents

Tables

Figures

Chapter

1

What's new in JDataStore 7

The list below briefly describes each new feature in JDataStore 7 and provides a link that takes you to the point in the *JDataStore Developer's Guide* where that feature is discussed.

New features

- **The JDataStore High Availability Server** provides incremental backup and automatic failover for your databases.

  Read "The JDataStore High Availability Server" on page 24 to understand how these new features fit into the overall JDataStore architecture.

  See the Server Console links, next, for details on how to use the Server Console for failover, incremental backup, and much more.

- **Server Console**

  The new **Server Console** provides graphical access to many new and old features. This *Developer's Guide* contains a new chapter, Chapter 6, "The JDataStore Server Console," that describes how to use this new Server Console to perform tasks:

  - "Overview" on page 57

    Get a quick overview of what you can do with Server Console

  - "About datasources" on page 57

    If you're not familiar with how JDataStore uses datasources to connect to databases, this is the place to find out.

  - "The Server Console interface" on page 58

    Get familiar with the new Server Console interface.

  - "Starting and stopping the server" on page 60

    You can use Server Console to start and stop the local server.

  - "Connection settings" on page 60

    A detailed description of how to create, modify, and delete datasources, and how to connect to and disconnect from them.

- "Verifying a database" on page 62

  Server Console makes it easier than ever.

- "Managing connections" on page 61

  View and optionally terminate both local and remote datasources.

- "Managing database properties" on page 62

  See an editable grid of database properties for both local and remote connections.

- "Viewing table and row locks" on page 63

  Server Console makes it easy to see what's happening with locks on both the local and remote server.

- "Viewing the database status log file" on page 63

  You exactly what's been going on with a server by viewing its server status file.

- "Incremental backup, auto failover, and load balancing" on page 63

  Use mirrors to protect against failures and to incrementally backup your data on a schedule of your choosing. Using Server Console, you can add

- "Interactive SQL" on page 71

  A special SQL pane lets you run SQL or SQL scripts against connected datasources.

- "Choosing between JdsExplorer and the Server Console" on page 29

  Server Console is new, JdsExplorer has been around for a while. How do you know which to use?

- **New data type**

  TINYINT or BYTE

- **New keywords**

  JDataStore's keywords are now in two categories: reserved and nonreserved. For a complete list of each category, see "Keywords" on page 88. For a list of keywords that are new in this release, see **New keywords** in the Release Notes.

- **New SQL functions**

  - BIT_LENGTH
  - CASE
  - COALESCE
  - CURRENT_ROLE
  - CURRENT_USER
  - NULLIF
  - OCTET_LENGTH
  - USER
  - DB_ADMIN
  - DB_UTIL: numeric, string, and date/time functions

  About DB_ADMIN and DB_UTIL
  `DB_UTIL` and `DB_ADMIN` are two new JDataStore Java classes. The methods can be called from SQL using the `CALL` statement. You can call them without creating a `JAVA_METHOD` alias because JDataStore SQL recognizes the methods in `DB_ADMIN` and `DB_UTIL` as built-in java methods.

  You can use the `DB_ADMIN` functions to perform a variety of database administration tasks such as configuring automatic failover and incremental backup, changing database properties, managing datasources, verifying tables, and displaying

database privileges and properties, locks, status log IDs, procedure privileges, and roles granted.

`DB_UTIL` is a collection of SQL utility functions that perform numeric, string and date/time operations on data stored in database tables. These functions are available in three forms: as methods of the DB_UTIL class, as SQL functions, and as JDBC escape functions.

- **New and updated SQL statements**
  JDataStore 7 now implements schemas, views, and roles. These along with the new `GRANT` and `REVOKE` statements greatly enhance security options. There are also new transaction management controls through `SET TRANSACTION`. This new SQL affects existing SQL statements such as `CREATE TABLE` and `ALTER TABLE` as well as offering many new JDataStore SQL statements.
  - `CREATE SCHEMA`, `DROP SCHEMA`
  - `CREATE VIEW`, `ALTER VIEW`, `DROP VIEW`
  - `CREATE ROLE`, `SET ROLE`, `DROP ROLE`
  - `CREATE JAVA_CLASS`, `DROP JAVA_CLASS`
  - New `CREATE TABLE` options:
    - Schema name
    - Track data changes with the `RESOLVABLE` option
    - Specify column position
  - New `ALTER TABLE` options:
    - `RESOLVABLE`
    - `RENAME` table (permits schema change)
    - Change column data type
    - Change or drop column default
    - Change column NULL constraint
    - Change column name
    - Change column position
  - New `DROP TABLE` options: `CASCADE` and `RESTRICT`
  - Security management:
    - `GRANT` and `REVOKE`
    - `CREATE USER`, `ALTER USER`, `DROP USER`
  - Output table rename for `SELECT` expressions: see “Select expressions” on page 108.
  - Transaction management: `SET TRANSACTION`
  - Many new JDBC escape functions
- **ISQL cross-platform SQL command-line utility**

Improved capacity

Significant work has been done on the JDataStore internals to improve speed, capacity, and reliability. In particular, JDataStore now has 64-bit database support for up to 281 trillion rows.

New and updated documentation

The JDataStore documentation has been extensively updated to describe the new features listed above. In particular, Chapter 10, “SQL reference” has major additions. Two new chapters have been added to this book. You are reading the new “What’s New” chapter now. There is also an entire new chapter for the new Server Console.

- There are extensive additions and revisions to Chapter 10, “SQL reference.”
- A new chapter has been added, Chapter 6, “The JDataStore Server Console,” documenting the new JDataStore Server Console.
- Chapter 8, “Stored procedures and UDFs” contains new documentation on returning JDBC ResultSets.
- Calling stored procedures is now documented in Chapter 10, “SQL reference.”
- Chapter 5, “JDataStore administration” now contains material on migrating older databases to JDataStore 7.
- Chapter 3, “JDBC quickstart” contains a section on using JDataStore with JBuilder and the Borland Enterprise Server.

Chapter

2

Introduction

JDataStore is a high-performance, small-footprint, all-Java™ transactional database. JDataStore provides:

- Industry standards compliance
  - JDBC 3 - Highest compliance level
  - J2EE 1.3.1 - Full certification suite completed
  - Entry level SQL-92
  - XA/JTA Distributed transactions
  - JavaBean data access components
  - Java collation key support for sorting and indexing
- High performance and scalability for demanding online transaction processing (OLTP) and decision support system (DSS) applications
- Zero-administration, single-jar deployment
- International support, including collation keys and Unicode storage for character data

JDataStore and DataStore

JDataStore is the name of the product, its tools, and of the file format. Within this product, there is a `datastore` package that includes a `DataStore` class, as well as several classes that have “DataStore” as part of their name.

What you should know

The *JDataStore Developer’s Guide* assumes you have a working knowledge of the following:

- Java programming
- Basic DataExpress
- Basic JDBC
- Basic SQL

Useful resources

This section identifies some useful resources for working with JDataStore.

JDBC

- For extensive JDBC documentation, see “JDBC™ API Documentation” on the `java.sun.com` website. The “Getting Started” section is especially useful if you are new to JDBC.
- *JDBC API Tutorial and Reference*, Addison Wesley, Seth White, Maydene Fisher, Rick Cattel, Graham Hamilton, Mark Hapner. If you do a lot of work with JDBC, this book very worthwhile.

SQL

“A Guide to The SQL Standard”, Addison Wesley, C. J. Date, Hugh Darwen.

DataExpress

“Database application programmers guide” and the “DataExpress Component Library Reference”.

JDataStore news group

`news://newsgroups.borland.com/borland.public.jdatastore`

Deploying JDataStore application components

You can find information on deploying the JDataStore Server in Chapter 12, “Deploying JDataStore database applications.” For tips on reducing the deployed size of JDataStore client applications, see “Pruning deployed resources” on page 152.

When you are ready to deploy JDataStore, you need to purchase additional deployment licenses. Please contact Borland Customer Service for more information or go to `http://shop.borland.com`.

You can also obtain licenses from Borland’s Online Store at `http://shop.borland.com`.

Contacting Borland

Borland offers a variety of support options for JDataStore. You can search our extensive information base and connect with other users of Borland products. You can also choose from several levels of support. These range from pre-sales support and support for installing JDataStore to fee-based consultant-level support and detailed assistance.

- For links to pre-sales support, installation support and a variety of technical support options, visit:

  Americas: `http://www.borland.com/support/americas/`
  Asia/Pacific: `http://www.borland.com/support/asia_pacific/`
  Europe/Middle East/Africa: `http://www.borland.com/support/emea/`

- To purchase licenses and upgrades, click the JDataStore link at:

  `http://shop.borland.com`

- Borland maintains an Internet site for general JDataStore information at:

  `http://www.borland.com/jdatastore`

- White papers, technical information, and FAQs about JDataStore technical information, such as white papers and FAQs, can be found on the Borland Developer's Network at:

  `http://bdn.borland.com/jdatastore`

- For information about how to contact JDataStore representatives outside the U.S. and Canada, look at the following web page:

  `http://www.borland.com/bww/`

- To discuss issues with other JDataStore users, visit:

  `http://info.borland.com/newsgroups`

When contacting support, be prepared to provide complete information about your environment, the version of the product you are using, and a detailed description of the problem.

Chapter 3

JDBC quickstart

This chapter is for developers who are familiar with the JDBC API and just want to know the basics of making a JDBC connection to a JDataStore database.

There are two types of JDBC connections: local and remote.

- **Local connections** access the JDataStore database engine in-process. This is a fast API layer because the application and the database engine operate in the same process.
- **Remote connections** require that the JdsServer database server be launched first. They use a tcp/ip protocol to communicate with the JdsServer.

A database file can be open by only one process at a time, so the fast local connections must all be made from the same process. Remote connections can be slower for chatty JDBC interactions, but allow more than one process running on one or more computers to access the same database.

A third option is to use a combination of the local and remote JDBC drivers. If there is one process that performs the majority of JDBC interactions, this process can launch the JdsServer inside its own process. In this way, the demanding JDBC process can use the local driver while still allowing other processes to access the same database using the remote JDBC driver. For more information see "Using both local and remote JDBC drivers to access a database" on page 14.

Local JDBC connections

Local connections using the Driver Manager

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

You can use JDataStore's Type 4 direct all-Java JDBC driver `com.borland.datastore.jdbc.DataStoreDriver` to access both local and remote JDataStore database files. The URL format for local connections is:

```
jdbc:borland:dslocal:<filename>
```

The following code provides a simple example of how to establish a local JDBC Connection:

```
import com.borland.datastore.jdbc.DataStoreDriver;
import com.borland.datastore.driver.cons.ExtendedProperties;

Class.forName("com.borland.datastore.jdbc.DataStoreDriver");
java.sql.Connection con =
 java.sql.DriverManager.getConnection(DataStoreDriver.URL_START_LOCAL
   + "/acme/db/acme.jds");
```

Local Connection using a JDBC DataSource

```
import java.sql.*;
import java.util.*;
import com.borland.javax.sql.*;

JdbcDataSource dataSource = new JdbcDataSource();
```

For JDataStore, this should be set to the operating system file name for the database. The `user.dir` system property can be used to qualify the full path name of this file. This is useful if you need to use relative paths for your file locations:

```
dataSource.setDatabaseName(sampleDatabaseFileName);

java.sql.Connection = dataSource.getConnection("frank", "borland");
```

Remote JDBC driver

Starting the JdsServer

The JdsServer must be started before you attempt to use the remote JDBC driver. The easiest way to start the JDataStore server is to execute the `JdsServer` executable from the JDataStore installation `bin` directory or from the desktop shortcut.

See Chapter 5, “JDataStore administration” for more information on configuration options for the JdsServer.

Remote Connection using the DriverManager

You can use JDataStore’s Type 4 direct all-Java JDBC driver `com.borland.datastore.jdbc.DataStoreDriver` to access both local and remote JDataStore database files. The URL format for local connections is:

```
jdbc:borland:dsremote://<hostname>/<filename>
```

Note that on Unix, filenames that are relative to root start with a slash, so URLs for those files must have two slashes between the hostname and filename.

The following code offers a simple example of how to establish a remote JDBC Connection:

```
Class.forName( "com.borland.datastore.jdbc.DataStoreDriver" );
java.util.Properties info = new java.util.Properties();
info.setProperty( "user", "MyUserName" );
Connection con = DriverManager.getConnection(
  "jdbc:borland:dsremote://mobile.mycompany.com/c:/someApp/ecom.jds", info );
```

Remote connection using a JDBC DataSource

```
import java.sql.*;
import java.util.*;
import com.borland.javax.sql.*;

JdbcDataSource dataSource = new JdbcDataSource();
```

For JDataStore, this should be set to the operating system file name for the database. The `user.dir` System property can be used to qualify the full path name of this file. This is useful if you need to use relative paths for your file locations.

```
dataSource.setDatabaseName(sampleDatabaseFileName);
```

By default, the JDataStore datasources use the local in-process JDBC driver to access the database. The setting below causes the remote JDBC driver to be used. The remote JDBC driver allows more than one process to access the same database. The JdsServer must be started before a remote connection can be made.

```
dataSource.setNetworkProtocol("tcp");

java.sql.Connection = dataSource.getConnection("frank", "borland");
```

Specifying extended properties

The API documentation for the `com.borland.datastore.driver.cons.ExtendedProperties` interface explains how to specify extended properties and gives the latest collection of extended properties that can be set for a JDBC connection.

The extended properties can be specified in JDBC URLs using semicolons to separate the properties. For example:

```
jdbc:borland:dslocal:mydatabase.jds;create=true;tempDir=/tmp
```

If a DataSource implementation is used to establish a connection, the DataSource `setProperties` method can be used to specify extended properties.

Using JDataStore with JBuilder and BES

To make the latest version of JDataStore available to JBuilder and the Borland Enterprise Server (BES), you need to copy certain files form the JDataStore *lib* directory to the *lib* directory of the target product:

1. For each product, find the listed files in the *lib* directory of the install tree and copy them to a backup directory.
2. Find the files listed below in the *lib* directory of the JDataStore 7 install tree and copy them to the *lib* directory of the target product—JBuilder or BES.

Copy the following files:

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

Unless you follow these steps, JBuilder and Borland Enterprise Server cannot handle files that include new JDataStore 7 features.

Chapter

4

System architecture

JDataStore is a relational database written entirely in the Java programming language. The design and implementation of this product emphasizes database performance, scalability, ease of use, and a strong adherence to industry standards. The fact that the product is developed in Java and that the database file system has no dependencies on the operating system or hardware in use makes JDataStore an extremely portable database engine.

JDataStore adheres to many industry standards including:

- SQL-92 entry level support for SQL
- JDBC 3 - highest level of compliance
- J2EE CTS 1.3.1 - complete compliance
- XA/JTA distributed transaction APIs
- JavaBean data access components

JDataStore programmatic interfaces

The JDataStore database engine can be accessed by two Java-based interface layers and one native interface:

- **JDBC** is the industry standard SQL call-level interface for Java applications.
- **DataExpress JavaBeans** provide additional functionality not addressed by the JDBC standard.
- **ODBC to JDBC Gateway** provided by EasySoft is an industry standard SQL call-level interface. The EasySoft ODBC to JDBC gateway lets native applications access JDataStore databases.

For extensive JDBC documentation, see “JDBC™ API Documentation” on the `java.sun.com` website.

JDataStore JDBC drivers

There are two JDataStore Type 4 JDBC drivers:

- **Local** The JDataStore database engine executes in the same process as the application using the JDBC interface.
- **Remote** The JDataStore database engine executes in either the same process or a different process as the application using the JDBC interface.

Local JDBC driver

The local JDataStore JDBC driver provides the following benefits:

- It is a high-speed interface to the database: JDBC API calls are direct. There are no remote procedure calls to a database server running in another process.
- It is easier to embed in an application: the database server does not need to be configured or started. The only requirement is to add `jds.jar` or `jdsserver.jar` to a location that is on the application's class path.

Remote JDBC driver

Use the remote JDataStore JDBC driver to execute JDataStore in a separate database server process. Before the application can use the remote JDBC driver, the JDataStore server (JdsServer) process must be started. Executing the JDataStore database engine in a separate database server process provides the following benefits:

- Multiprocess access to a database. If multiple processes on one or more computers need to use the JDBC API to access JDataStore, a JDataStore server must be started and the remote JDBC driver must be used.
- Improved performance when multiple computers are in use. 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 JDataStore server on a separate computer.
- Improved fault tolerance.

Using both local and remote JDBC 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 JDataStore database file can be held open by only one operating system process. When you connect using the local JDBC driver, the process that uses the local JDBC driver holds the database file open. When the remote JDBC driver makes the connection, the JdsServer process holds the database file open.

Since the local JDBC 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 JDBC driver also starts a JDataStore server in the same process, then other processes using the remote JDBC driver can access the same database as the local JDBC driver.

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

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

DataExpress JavaBeans

DataExpress is a set of JavaBean components that surface 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.

The DataExpress components are included in the JBuilder Visual Designer's component palette. However, you don't *have* to have JBuilder to develop and deploy applications that use DataExpress components. DataExpress is a set of runtime components. JBuilder includes some visual design tools for JavaBean components such as DataExpress. For more information on developing DataExpress from within JBuilder, see JBuilder's *Developing Database Applications*.

The majority of DataExpress JavaBean components are centered around components needed to build both server- and client-side database applications. Client-side applications need high quality data binding to visual components such as grid controls and also need support for reading and writing data to a database.

Server-side applications need data access components to help with reading and writing data to a database, but presentation is typically handled by some 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 keeps the presentation separate from its 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's 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 JDataStore 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 JDataStore tables.

Automating administrative functions with DataExpress JavaBeans

There are many DataExpress components that can be used to automate administrative tasks, including:

- 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 allows developers to more easily debug their applications and to gain a better understanding of the DataExpress JavaBean components.

Native access to JDataStore using ODBC

EasySoft provides an ODBC to JDBC gateway driver that can be used to access JDataStore databases from native applications and tools on a Windows platform.

Specifications

The following are the specifications of the JDataStore database file format.

JDataStore database file capacity

Minimum block size: 1 KB

Maximum block size: 32 KB

Default block size: 4 KB

Maximum JDataStore 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: 2 billion

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

JDataStore stream names

Directory separator character: /

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. JDataStore has the following system tables:

- SYS/Connections
- SYS/Queries
- SYS/Users

The JDataStore file system

A JDataStore database file can contain three basic types of data streams: *table* streams and two types of *file* streams. A single JDataStore database can contain all three stream types.

These streams are organized in a file system directory. The ability to store both tables and arbitrary files in the same file system allows all an application’s data to be in a single portable, transactional file system. A JDataStore database can also be encrypted and password protected.

Table streams are database tables created by the JDBC or DataExpress APIs. They can also be cached table data from an external data source such as a database server. Setting the `store` property of a `StorageDataSet` to the `DataStore` creates the cached table data.

A table stream can have secondary indexes and Blob storage associated with it. If the table's `resolvable` property is set, it also tracks all insert, update, and delete operations made against the table. This edit tracking feature enables DataExpress components to synchronize changes from a replicated table to the database the table was replicated from.

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.

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 JDataStore 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 JDataStore file system

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

- It's simpler. Only 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 your objects to the same file.
- Your application can use less storage space, because of how disk clusters are allocated by some operating systems. The default block size in a JDataStore database file is small (4KB).
- Because you're not at the mercy of the host file system, your application is more portable. For example, different operating systems have different allowable characters for names. Some systems are case-sensitive, while others are not. Naming rules inside the JDataStore file system are consistent on all platforms.
- JDataStore provides a transactional file system that can also be encrypted and password protected.

JDataStore directory contents

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

Table 4.1 JDataStore 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 `storeName`
4	`Type`	`DIR_TYPE`	short	Bit fields that indicate the type of streams
5	`Id`	`DIR_ID`	int	A unique ID number

Table 4.1 JDataStore directory table columns (continued)

#	Name	Constant	Type	Contents
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 a table stream's Blobs, 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. They provide compile-time checking to ensure that you are referencing a valid column. Constants with names that end with `_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 that end with `_STREAM`. See `com.borland.datastore.DataStore` for a listing of these constants.

Stream details

Times in the JDataStore directory are Coordinated Universal Time (UTC). They are suitable for creating dates with `java.util.Date` (long).

As with many file systems, when you delete something in JDataStore, the space it occupied is marked as available, but the contents and the directory entry that points to it are not wiped clean. This means you can sometimes undelete a deleted stream if it has not been overwritten.

For more information on deleting and undeleting streams, see "Deleting streams" on page 19, "How JDataStore reuses blocks" on page 19, and "Undeleting streams" on page 20.

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

These internal streams have the same `StoreName` as the table stream with which they're associated. This means that the `StoreName` alone doesn't always uniquely identify each stream when it interacts with the JDataStore database at a low level. Some internal stream types can have multiple instances. Therefore, the ID for each stream must guarantee uniqueness at a low level. But the `StoreName` is unique enough 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. They are then followed by all deleted streams ordered by their delete time, oldest to most recent. (You can't use a `DataSetView` to create a different sort order.)

Storage allocation within the JDataStore file system

The 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 JDataStore 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 JDataStore 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 these case an entire block is reallocated from the list of deleted blocks.
- **Blocks that are partially full** These free space can only be reused on a per-stream basis. Specifically, the free space in a table "A" block can be reused only by a new allocation for a row in table "A". Table, secondary index, Blob, and file are all separate streams from an allocation perspective.

Partially allocated blocks are kept at least 50 percent full on average. The file system goes to great lengths to make sure this is true for all stream types in the JDataStore file system. The one exception to this rule occurs when a stream has a small number of blocks allocated.

A JDataStore 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 using the `DataStoreConnection.copyStreams()` method.

Deleting streams

Deleting a stream doesn't 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 the stream was deleted is recorded. Over time, new stream allocations overwrite the space that was formerly occupied by the deleted stream, making the content of the deleted streams unrecoverable.

Streams can be deleted using JdsExplorer, or they can be deleted programmatically using `DataStoreConnection.deleteStream()`, which takes as an argument the name of the stream to delete.

For an example of how to delete and undelete streams, see the tutorial "Demonstration class: DeleteTest.java" on page 195.

How JDataStore reuses blocks

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

- JDataStore 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—the ones with the earliest delete times—are reclaimed first.
- For table streams, the support streams (such as those for indexes and aggregates) are reclaimed first.
- Space is reclaimed from the beginning of the stream to the end of the stream, meaning that 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 undelete.

Undeleting streams

JDataStore allows deleted streams to be undeleted if their space has not be consumed by new allocations as described in the prior section. A stream can be undeleted in JdsExplorer or by calling the `DataStoreConnection.undeleteStream()` method.

Because table streams have multiple streams with the same name, the stream name alone isn't sufficient for attempting to undelete a stream programmatically. You must use a row from the JDataStore directory. It 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 undelete.

If you create a new stream with the name of a deleted stream, you can't undelete that stream while its name is being used by an active stream.

Transaction management

A transaction's lifecycle begins with any read or write operation through a connection. JDataStore uses stream locks to control access to resources. To read a stream or make a change to any part of a stream (a byte in a file, a row in a table), you must be able to acquire a lock on that stream. Once a connection acquires a lock, it holds on to it until the transaction is committed or rolled back.

In single-connection applications, transactions primarily provide crash recovery and allow you to undo changes. Or you might have made a JDataStore database transactional so that it can be accessed through JDBC. If you want to access that JDataStore database using DataExpress, you must now deal with transactions. How transactions work has deeper ramifications for multiconnection multiuser or single-user multisession applications. These are discussed in "Avoiding blocks and deadlocks" on page 160, along with other multiuser issues in Chapter 3, "JDBC quickstart."

Transaction isolation levels

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

The serializable isolation level designated by `java.sql.Connection.TRANSACTION_SERIALIZABLE` provides complete transaction isolation. An application can choose 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 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 wasn't 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. These are shown in the following table.

Table 4.2 SQL isolation level definitions

Isolation Level	Dirty Read	NonRepeatable Read	PhantomRead
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

Setting isolation levels on JDataStore connections

Setting isolation levels on JDataStore connections:

- Use `java.sql.Connection.setTransactionIsolation(int level)` to specify the isolation level for JDataStore JDBC connections.
- Use `com.borland.datastore.DataStoreConnection.setTxIsolation(intlevel)` for JDataStore DataExpress connections.

In both of the above methods, the “level” parameter can be one of the four following values:

```
java.sql.TRANSACTION_READ_UNCOMMITTED
java.sql.TRANSACTION_READ_COMMITTED
java.sql.TRANSACTION_REPEATABLE_READ
java.sql.TRANSACTION_SERIALIZABLE
```

To choose an isolation level, refer to the following table:

Table 4.3 JDataStore connection isolation levels

Isolation level	Description
`TRANSACTION_READ_UNCOMMITTED`	Suitable for single-user applications for reports that do not mind transactionally inconsistent views of the data. Especially useful when browsing JDataStore tables with dbSwing and DataExpress DataSet components. This isolation level incurs minimal locking overhead.
`TRANSACTION_READ_COMMITTED`	Commonly used for high-performance applications. Ideal for data access models that use Optimistic Concurrency Control. DataExpress `QueryDataSet` and Borland Application Server use Optimistic Concurrency Control approaches to data access. In these data access models, read operations are generally performed first. In some cases, read operations are actually performed in a separate transaction from the write operations.
`TRANSACTION_REPEATABLE_READ`	Provides more protection for transactionally consistent data access without the reduced concurrency of `TRANSACTION_SERIALIZABLE`. This isolation level, however, results in increased locking overhead because row locks must be acquired and held for the duration of the transaction.
`TRANSACTION_SERIALIZABLE`	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 JDataStore lock manager to escalate to using table locks.

Locks used by the JDataStore lock manager

The following table describes the locks used by the JDataStore lock manager:

Table 4.4 Locks used by the JDataStore lock manager

Lock	Description
Critical section locks	Internal locks used to protect internal data structures. These locks are usually held for a short period of time. They are acquired and released independent of when the transaction is committed.
Row locks	Shared and exclusive lock modes supported. These locks are released when the transaction commits.
Table locks	Shared and exclusive lock modes supported. These locks are released when the transaction commits.
DDL table locks	Shared and exclusive lock modes supported: ■ 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 JDataStore locking behavior

You specify extended JDBC properties by setting them in a `java.util.Properties` object that is passed when creating a JDBC connection.

See the `com.borland.datastore.driver.cons.ExtendedProperties` class for a list of all JDataStore extended properties. Property names are case sensitive and are described in the following table.

Table 4.5 Extended JDBC Properties that control locking behavior

Property	Behavior
`tableLockTables`	A string of semicolon-separated, case-sensitive table names. Row locking will not be used for tables specified in this list. To specify all tables, set this property to "*". This property can also be set for `DataStoreConnection` components by calling the `setTableLockTables()` method.
`maxRowLocks`	The maximum number of row locks per table that a transaction should acquire before escalating to a table lock. The default value is 50. This property can also be set for `DataStoreConnection` components by calling the `setMaxRowLocks()` method.
`lockWaitTime`	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. This property can also be set for `DataStoreConnection` components by calling the `setLockWaitTime` method.
`readOnlyTxDelay`	The maximum number of milliseconds to wait before starting a new read-only view of the database. For more information, see the discussion of the `java.sql.Connection.readOnly` property in "Tuning JDataStore concurrency control performance" on page 151.

JDataStore lock usage 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 following table describes the row locking behavior of the JDataStore connection isolation levels:

Table 4.6 Lock usage and isolation levels

Connection isolation level	Row locking behavior
`TRANSACTION_READ_UNCOMMITTED`	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`	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.
`TRANSACTION_REPEATABLE_READ`	Acquires shared row locks for read operations.
`TRANSACTION_SERIALIZABLE`	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 nonuniquely specified row

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

Concurrency control changes

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

- Support for `TRANSACTION_READ_UNCOMMITTED` and `TRANSACTION_SERIALIZABLE`.
- Using shared table locks for read operations; earlier versions of JDataStore software used exclusive table locks for read and write operations.

Connection pooling and distributed transaction support

JDataStore 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 you need to download the `J2EE.jar` file from Sun, and add it to your classpath. If you are using JBuilder, you also need to add it to your project as a required library. See “Adding a required library to a project” in JBuilder’s *Developing Database Applications* for instructions on adding a required library.

Connection pooling

The basic idea behind connection pooling is simple. 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.

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

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

To get a distributed connection to a JDataStore database from a `JdbcConnectionPool`, call `JdbcConnectionPool.getXAConnection()`. The connection returned by this method works only with the JDataStore 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

JDataStore 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), you can specify one of three possible string settings for this property:

- `commit`: causes the transaction to be heuristically committed when JDataStore reopens the database.
- `rollback`: causes the transaction to be heuristically rolled back when JDataStore reopens the database.
- `none`: causes JDataStore 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 default setting for this property is `commit`.

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

The JDataStore High Availability Server

Overview

One of the most important areas of concern for any database application is eliminating single points of failure. The JDataStore 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 face 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 this 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 JDataStore 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.

Types of mirrors

There are three mirror types that can be used by an application: primary, read-only, and 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 point in time.

Read-only mirrors

There can be any number of read-only mirrors. Connections to these databases can perform only read operations. Read-only mirrors always 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 only read operations. 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 JDataStore engine and failover

Transaction log records

JDataStore 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. JDataStore's existing 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 hotswapped 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 JDataStore High Availability Server provides a broad range of benefits. Some of the key advantages are described below.

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 technology such as shared disks, private LANs, or fiber channels and no need for additional software including Linux, Windows, Solaris, and 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 JDataStore's 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

Mirrors can specify which mirror they are synchronized by. 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 JDataStore

The replication support provided by DataExpress for JDataStore is quite simple. This makes it much 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. JDataStore does not need the server-side software or database triggers that are required for more complex publish-subscribe solutions. Complex multilevel hierarchies and network topologies are not directly supported.

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

Provide	Provide the client database with a snapshot of the server tables being replicated
Edit	Client application, which need not be connected to the database, reads/edits the client database
Resolve	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 JDataStore 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 JDataStore 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's provider is a pluggable interface. QueryDataSet and ProcedureDataSet are extensions of StorageDataSet, which preconfigure JdbcProviders that know how to 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 JDataStore tables that are replicating the database. All insert/update/delete write operations since the last Provide operation are automatically tracked by the JDataStore storage system. Part of the StorageDataSet store interface contract is that all insert/update/delete operations must be recorded if the StorageDataSet "resolvable" property 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 using a JDBC driver. An optimistic concurrency approach is used to save the changes back. 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 allow an application to respond with ignore, retry, abort, log, and so on when an error occurs.

There are also higher-level DataStorePump and DataStoreSync components that you can use to perform Provide and Resolve operations for a collection of tables. See "Custom synchronization" on page 52.

Chapter

5

JDataStore administration

There are three tools that are important for administering JDataStore databases:

- JdsExplorer is a graphical interface that provides a number of administration tools including backup, user administration, data import, database packing, and database verification.
- Server Console is a graphical tool for managing datasources, database connections and properties, verifying databases, creating mirrors for failover and incremental backup, viewing locks and log files, and performing tasks with ISQL. See Chapter 6, “The JDataStore Server Console” for details on using the Server Console for these tasks.
- JavaBean components offer a third approach: You can programmatically provide administrative functions using the same JavaBeans that are used by JdsExplorer and JdsServer. See the *DataExpress Component Library Reference* for information on using Java Beans to administer JDataStore databases.

To start and stop the server, see “Launching JdsServer” on page 30 and “Running JDataStore as a service” on page 31.

This chapter is primarily focused on using JdsExplorer for administration tasks. See Chapter 6, “The JDataStore Server Console” for how to use the Server Console.

Choosing between JdsExplorer and the Server Console

The Server Console is a graphical interface that is new in JDataStore 7. In it, you can perform some of the tasks that you previously performed in JdsExplorer. In addition, Server Console provides access to many new features.

From Server Console you can:

- Create and edit datasources, connect to local or remote databases, and view and manage connections.
- View and manage database properties, and view database status logs, view table and row locks. and verify databases. You cannot create databases in Server Console.

- Create and manage mirrors for incremental backup, auto failover, and load balancing.
- Use the ISQL tab to execute all SQL and ISQL commands. There is a list of ISQL commands at the end of the SQL Reference chapter in the *JDataStore Developer's Guide*. JdsExplorer off an environment for executing SQL commands, but you cannot execute ISQL commands there.

You need JdsExplorer if you want to:

- Perform user administration
- Manage license keys
- Work with and view table structure and content, including indexes.
- Import tables from other databases and import text and images into a database
- Upgrade, downgrade, copy, encrypt/decrypt or pack a database. Database verification can be performed from either JdsExplorer or Server Console.

Most of the tasks listed for JdsExplorer can be performed in the Server Console using the ISQL window.

JDBC drivers

JDataStore offers two JDBC drivers: local and remote:

- To access a JDataStore database through a **remote** JDBC connection, you must have a JdsServer running on a machine that has access to your database files.

  You can manage remote connections using the JDataStore Server Console. Most of the functionality that was formerly accessed through the JdsServer GUI is now found in the Server Console interface. The Server Console also offers much new functionality.
- Accessing a JDataStore database through a **local** JDBC connection does not require that a JDataStore server be running.

Launching JdsServer

There are several ways to launch the JDataStore server as an application. For another option, see "Running JDataStore as a service" on page 31.

- Click the Actions menu in the Server Console. There is a pause while JDataStore checks to see whether the local server is running. Then the Actions menu opens and displays the appropriate command: If the server is running, you are offered a command for stopping the server. If the server is not running, you are offered the option of starting it. The menu item changes to reflect the local server status.
- Run <JDataStore_home>/bin/`JdsServer` (`JdsServer.exe` on Windows).
  - If the server is not running, this action starts the server and displays a system console. Closing this console closes the JdsServer application. This executable uses the classpath, main class name, and other settings provided in the `JdsServer.config` file.
  - If the server is already running, re-executing JdsServer stops the server.
- If you are using JBuilder, use the Tools | JdsServer command.

- Open a system console and execute `JdsServer`. There are a number of options available, which are listed in the following table:

Table 5.1 JdsServer command line options

Option	Description
`-port=<number>`	Specifies the port to listen to; default is 2508
`-ui=<uiType>`	Selects the look and feel of the UI; one of the following: ■ windows ■ motif ■ metal ■ none ■ *<LookAndFeel class name>* The default is `-ui=none`. If you specify `-ui=<platform_name>`, it displays the now-deprecated JdsServer interface. JdsServer has been replaced with Server Console, but is still available for backward compatibility.
`-temp=<dirName>`	Specifies the directory to use for all temporary files
`-doc=<helpDir>`	Specifies the directory that contains the online help files
`-?` `-help`	Displays a message listing these options
`-shutdown`	Shuts down a server that is already running on this computer.

Running JDataStore as a service

During installation on Windows, Linux, and Solaris, you choose whether JDataStore is able to run as a service. Depending on the choice you make during installation, JDataStore either runs as a service automatically at startup, can be manually started as a service, or does not run as a service at all.

Windows

During installation you are prompted with the question “How would you like the JDataStore Service to run?”

- Choosing “On System Startup” registers `JdsServer.exe` with the Windows Service Manager and enables JDataStore to start as a service when your computer starts. You can start or stop the JdsServer service by using the Services applet from the Administrative tools in the Control Panel.
- Choosing “Manual” registers `JdsServer.exe` with the Windows Service Manager. The service does not start automatically, but can be started or stopped by using the Services applet from the Administrative tools in the Control Panel.
- Choosing “Do not install as a service” does not register `JdsServer.exe` as a service. JDataStore runs only as an application. If you want JDataStore to run as a service later, you must reinstall JDataStore and choose “On System Startup” or “Manual” at the prompt.

If you choose either “on System Startup” or “Manual” during installation, you can start and stop the JDataStore server at will starting/stopping it in the Services applet. If you access the Serves Applet often, you can put a shortcut to it on your desktop or some other convenient location so that you don’t have to dig for it every time.

Linux and Solaris

During installation you are prompted with the question “Would you like JDataStore to run as a service when your system starts?”

- Choosing `YES` allows JDataStore to run as a service when the system enters runlevel 3 or higher. If you want to remove JDataStore as a service at a later time, you can run `$JDATASTORE_HOME/bin/jdservice.sh -r` and all files and links will be removed.
- Choosing `NO` does not start JDataStore as a service. If you want JDataStore to start as a service at a later time, you can run `$JDATASTORE_HOME/bin/jdservice.sh -s` and the appropriate links and files will be made to start the JDsserver when the system enters runlevel 3 or higher.

Shutting down the server

To halt the server, perform one of the following two actions:

- Execute another instance of the JdsServer executable using the `-shutdown` command line option.
- Choose File | Shutdown from the File menu of the graphical JdsServer interface.

Getting help and running JdsExplorer

From the JdsServer graphical interface, you can launch JdsExplorer and access the JDataStore help system.

- To launch JdsExplorer, choose File | JDataStore Explorer.
- To access the help system, choose Help | Contents. This provides access to the JDataStore *Developer’s Guide*, the JBuilder *Developing Database Applications*, and the JBuilder *DataExpress Component Library Reference*.

Creating custom JDataStore servers

You can create custom servers with additional functionality. For example, because the server will probably be running all the time, you can add a maintenance thread that backs up files at the same time every night.

The core of any JDataStore Server is the `com.borland.datastore.jdbc.DataStoreServer` class. For more information, see the `com.borland.datastore.jdbc.DataStoreServer` class in the *JDataStore Component Library Reference*.

Uninstalling JDataStore on Linux and Solaris

To uninstall JDataStore on Linux or Solaris, you must run `$JDATASTORE_HOME/bin/jdservice.sh -r` before uninstalling. This removes links and files that are not removed by the uninstaller.

Working with JdsExplorer

You can use JdsExplorer to perform a wide range of administrative tasks through a graphical user interface. Note that virtually all of the administrative tasks surfaced by JdsExplorer are provided by public JavaBean components. These JavaBean components are accessible to applications so that administrative tasks can be customized or automated.

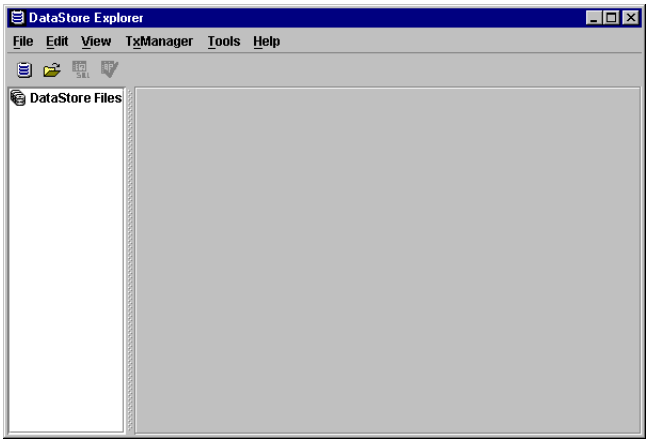
Using the all-Java JdsExplorer, you can:

- Examine the contents of a JDataStore database. The database's directory is shown in a tree on the left side of the window, with each table and its indexes grouped together. Regular files can also be stored in the JDataStore directory as "data streams." When you select a data stream in the tree, its contents appear (assuming that it's a supported file type such as a text file, GIF image, or table for which JdsExplorer has a viewer).
- Perform many JDataStore operations without writing code. You can create a new JDataStore database, create tables and indexes, import delimited text files into tables, import files as file streams, delete tables and indexes, delete tables or other data streams, and verify the integrity of a JDataStore database.
- Interactively establish automated heterogeneous replication and synchronization.
- Administer security features of a JDataStore database, including users, passwords, and encryption.

There are four ways to launch JdsExplorer:

- Use the desktop shortcut that's created during the install.
- Invoke `JdsExplorerW` (`JdsExplorerW.exe` on Windows) from the command line.
- If you are using JBuilder, use the Tools | JdsExplorer menu command.

Figure 5.1 JdsExplorer after launch



Starting JdsExplorer from the command line

There are a number of options available that you can access by invoking the `JdsExplorerW` (`JdsExplorerW.exe` on Windows) executable at the command line. These options are listed in the following table:

Table 5.2 JdsExplorer startup options

Option	Description
`-ui=<uiType>`	Determines the look and feel of the UI The options are: ■ windows ■ motif ■ metal ■ none ■ `<LookAndFeel class name>`
`-h=<helpDir>`	Specifies the directory that contains online help files
`<.jds filename>`	Specifies a JDataStore database file to open on startup
`-?`	Displays a message listing these options

Working with JDataStore databases

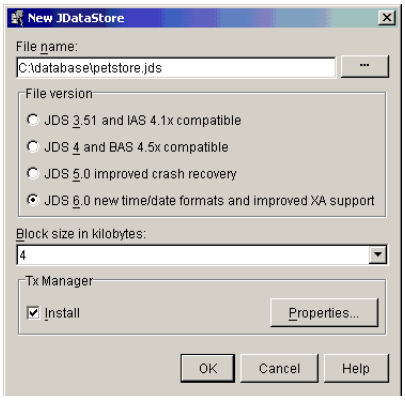
This section describes how to create and open JDataStore databases, how to toggle them between read-write and read-only, how to make them transactional or non-transactional) and how to open a database that was not properly shut down.

Creating a new JDataStore database

To create a new JDataStore database using JdsExplorer:

1 Select File | New or click the New JDataStore toolbar button. This opens the New JDataStore dialog box:

Figure 5.2 New JDataStore dialog box



2 Enter a name for the new JDataStore database file.

3 (Optional) Choose a block size. The default is 4KB.

4 Make sure the Install TxManager check box is set properly:

- To create a transactional database, ensure that the TxManager Install check box is checked. You can click Properties to set the transaction management properties for the new JDataStore database file.

  A transactional database provides data protection through crash recovery support. This mode is recommended for most purposes.

- To create a non-transactional database, ensure that the TxManager Install check box is empty.

  Non-transactional databases are appropriate for temporary and read-only databases, and for those in memory-constrained platforms, such as PDA applications.

5 Click OK. The database is created and opens in JdsExplorer.

You should create (“register”) at last one user for the new database. The next time you open the database, you will be prompted for a user name and password. If you do not create at least one user now, user authentication is disabled and you can provide any user name at the prompt when you next open the database. To create a user, choose Tools | Administer Users. Make sure that the initial user has administrative privileges by placing a check in the Administrator box.

Opening an existing JDataStore database

To open an existing JDataStore database:

1. Select File | Open or click the Open JDataStore toolbar button. This opens the Java version of the standard File Open dialog box.
2. Choose the database file to open and click Open. JDataStore presents a dialog box in which you must enter a user name and password. If the database does not yet have any users defined for it, you can enter any user name in this dialog box. If there is already at least one user defined, you must enter a valid user name and password.

JdsExplorer keeps track of the five most recently opened files. You can open them directly from its File menu.

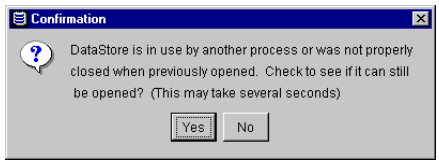
Opening a JDataStore database that was not shut down properly

There are several circumstances that can lead to a situation where a JDataStore database has not been shut down properly. This includes computer power failure, operating system crash, and abnormal termination of application code. When you attempt to open a JDataStore database after such an event, JDataStore automatically detects that the database was not shut down properly. For applications that use JDataStore programmatically, the database engine automatically initiates crash recovery when an attempt is made to reopen the database.

If your data is valuable, it is highly recommended that you keep transaction support enabled when updating your database. Only a transactional JDataStore database can provide the safety of crash recovery. If the JDataStore database has transaction support disabled, you may lose all or part of your data if the database is not shut down properly.

If the JDataStore database is already marked as open, which happens if the JDataStore database was not shut down properly, a dialog box appears asking if you want to try and open the JDataStore database anyway.

Figure 5.3 JDataStore database marked open dialog box



If this occurs, follow these steps:

1. Verify that there are no rogue Java process that might still have the JDataStore database open: check the Task Manager in Windows or run the **ps** command in Unix:
   - If there is no process that has the JDataStore database open, click Yes to reopen the JDataStore database.
   - If you click No, JdsExplorer responds with an error dialog box that states that the database is still marked open by another process and cannot be opened.
2. Attempt to reopen the JDataStore database. This takes about eight seconds while the operating system tries to detect whether the database is still open by another process. If the JDataStore database was not shut down properly, another dialog box informs you of this condition. Click OK to attempt to recover the JDataStore database.
3. After you successfully open a JDataStore database that was not shut down properly, a dialog box appears that gives you the opportunity to verify the contents of the database. Click Yes to verify the contents or No to skip the verification.

For a transactional database, the verify step should not be necessary since the JDataStore transaction manager provides crash recovery. If the database has transaction support disabled, the database should be verified to make sure it is in a good state. This database verifier is similar to file system check facilities provided by operating systems.

Migrating older databases to JDataStore 7

This section describes how to determine whether a database is current and how to perform the upgrade. It also discusses changes that occurred between JDataStore 6 and JDataStore 7.

To check whether a database is the current version, open the database and look on the Tools menu. If Upgrade JDataStore is grayed out, the database is already the most current version.

To migrate an older database to the current version:

1. In JdsExplorer, open the database that you want to migrate.
2. Choose Tools|Upgrade JDataStore. JDataStore backs up the older database and rewrites the database in the current format. It then displays a note that announces a successful upgrade and gives you the name and location of the backup.

JdataStore 7 offers a number of new features that are structurally incompatible with older versions of JDataStore, such as schemas and roles. In addition, JDataStore 7 has mirrors, which can be used to implement auto failover, incremental backup, and load balancing.

In JDataStore 7, every table belongs to a schema: the schema name is part of the table name. In general, when a user creates an object such as a table or a view, that object belongs to the user's own schema. There are a number of issues regarding the use of schemas. It will be helpful to you to read "CREATE SCHEMA" on page 113 to understand how JDataStore 7 uses schemas before proceeding with migration.

When a database is first migrated from an earlier version to JDataStore 7, the tables, views, and methods are all assigned to the `DEFAULT_SCHEMA` schema. To assign tables to different schemas, use `ALTER TABLE`'s new `RENAME` option.

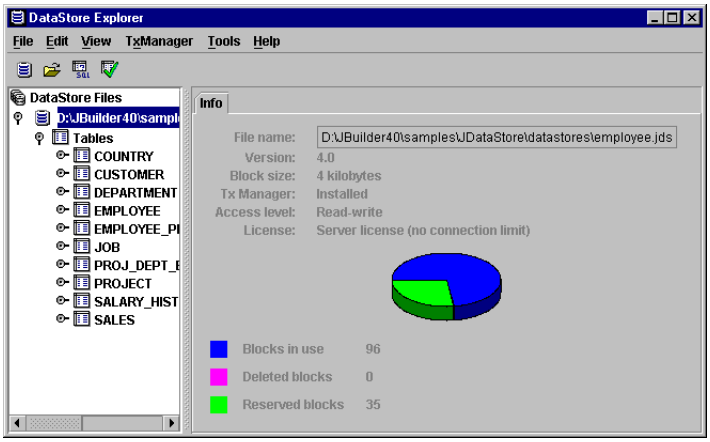
For an example of how to upgrade a database programmatically, see "Demonstration class: Dup.java" on page 193.

It is possible to access a JDataStore 6 or earlier database using JDataStore 7. However, your activities are limited because you cannot, by and large, use JDataStore 7 features. There is little reason not to upgrade.

Viewing JDataStore database information

When a JDataStore database file opens, its directory appears in a tree on the left side of JdsExplorer. Each open JDataStore database is a node directly off the root node. Information about the JDataStore database appears in the viewer area on the right.

Figure 5.4 JdsExplorer displaying JDataStore database information



This information includes:

- The name of the JDataStore database file
- The JDataStore file format version number
- The block size
- Whether the TxManager is installed; that is, whether the database is transactional
- Whether the database was opened as read-write or read-only
- What sort of license is being used
- A graphical representation and count of how blocks are allocated between:
  - Blocks in use
  - Blocks formerly occupied by data that is now marked deleted and are available for reuse
  - Reserved blocks preallocated for future use (transactional JDataStores preallocate disk space to improve reliability)

You can view this information at any time by selecting the node in the tree that contains the JDataStore database name.

About the version number The version number displayed here is the Borland internal number. To see the public version number, look at the database properties in the Server Console. You can tell whether a database is the current version in JdsExplorer by opening it and checking the Tools menu. If Update JDataStore is grayed out, the database is the current version.

Verifying a JDataStore database

Verification is not normally necessary for a transactional database. Such databases have crash recovery to prevent database corruption. However, this facility can be very useful when a non-transactional database was shut down while data was being written to it.

To verify the structure and contents of the JDataStore database, select Tools | Verify JDataStore or click the Verify JDataStore toolbar button.

JdsExplorer verifies the entire database and displays the results in the Verifier Log window. After you've closed the log window, you can view it again by selecting View | Verifier Log.

To verify a database programmatically, use the `com.borland.datastore.StreamVerifier` class.

Copying and backing up JDataStore databases

There are two commands on the JdsExplorer Tools menu for copying a database: Copy JDataStore and Copy Into Existing JDataStore. These commands work slightly differently, but either can be used to create a copy of a database for backup or for other reasons. The Copy Into Existing JDataStore command can also be used to downgrade a database to an older version or to decrypt a database.

There are other ways to create and maintain database backups. To maintain multiple mirrors of a database that are automatically kept up to date, see "Incremental backup, auto failover, and load balancing" on page 63.

To copy a database

1. In JdsExplorer, open the database you want to copy.
2. Choose Tools|Copy JDataStore to display the Copy To dialog box.
3. Type in a new database name or choose the database you want to copy to or .

   If you choose an existing database name, JDataStore makes a backup copy of the existing database file and then copies the chosen database to the existing name. The result is a copy of the chosen database, not a combination of the two.
4. Choose Open.

The chosen database is copied to the new or existing database name.

To copy into an existing database

This command allows you to copy a database into an existing empty database. You can use this command to downgrade a database to an older version or to decrypt a database.

1. Create a new, empty database
   a. In JdsExplorer, choose File|New or click the New JDataStore button to display the New JDataStore dialog box.
   b. Select the properties that you want the new database copy to have. Note that you can select a previous version of JDataStore in order to downgrade the database to an older version.
   c. Click okay to create the empty database.
2. Open the database you want to copy.
3. Choose Tools|Copy Into Existing JDataStore.
4. In the Select JDataStore to Copy Into dialog box, choose the empty database that you just created.
5. Choose Open.

JDataStore copies the selected open database into the new database. The new database created by the copy operation is always unencrypted, regardless of whether the source database was encrypted.

Important If you choose an earlier version of JDataStore when creating the new database, be aware that JDataStore "remembers" this setting. The next time you create a database,

the earlier version is still selected. To avoid accidentally creating older databases, you could create a new database in the current version and then delete it. This leaves the New JDataStore dialog box set to the current version.

Backing up programmatically

To back up the database programmatically, see the com.borland.datastore.DataStoreConnection.copyStreams() method. You might also want to set the com.borland.datastore.DataStoreConnection.setReadOnlyTx method to avoid blocking write transactions while the `copyStreams()` method is executing.

Making a JDataStore database transactional

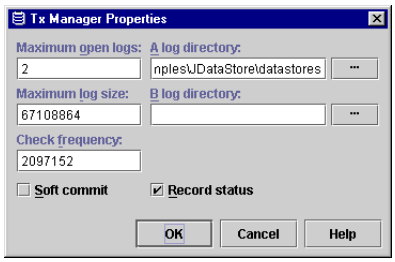
Transaction support for a JDataStore database can be switched on and off. JDataStore will open a non-transactional database only if it is also read-only. Some benefits of turning off transaction support include:

- New databases load faster, since transaction support slows down write operations.
- Databases can be transported more compactly without their associated log files.
- You can install populated databases onto CD-ROM devices.

You can make a non-transactional JDataStore transactional using JdsExplorer:

1 Select TxManager | Install. (If the database is already transactional, that menu option is disabled.) This opens the TxManager Properties dialog box:

Figure 5.5 TxManager Properties dialog box



2 The dialog box contains default settings for the `TxManager` object:

- You can change the default settings for the maximum number of open log files (2), maximum log file size (64MB), and checkpoint frequency (2MB). JDataStore performs a check of the database each time the log file grows by the number of megabytes specified in Check Frequency.
- By default, one set of log files is written in the same directory as the JDataStore database. To choose another directory for this set, specify a different A log directory.
- To maintain a second redundant set of log files, specify a B log directory.
- You can enable Soft commit, which increases performance by not immediately forcing a disk write when a transaction is committed. For more about soft commits, see “Using soft commit mode” on page 151.
- You can disable status logging for a slight performance improvement.

3 Click OK.

To install transaction support programmatically, use `TxManager`. See TxManager in online help for more information.

Modifying transaction settings

To modify transaction settings for a JDataStore database:

1. Select TxManager | Modify. (If the database is not transactional, that menu option is disabled.) This opens the TxManager Properties dialog box.
2. Change settings as desired.
3. Click OK.

To modify transaction settings programmatically, use `TxManager`. See TxManager in online help for more information. This class offers a greater range of control than the JdsExplorer interface.

Removing transaction support

You can check whether a JDataStore database is transactional by examining the TxManager menu in JdsExplorer. If the Enabled menu option is checked, the database is transactional. If the menu option is disabled, the database is not transactional.

To make a transactional database be non-transactional, uncheck TxManager | Enabled. This immediately removes transaction support.

Packing a JDataStore database

Packing a JDataStore database file removes all free space in the database file system. This makes the database file smaller and removes fragmentation. This can improve the performance of read operations. Write operations are sometimes slightly slower for a recently packed database. Packing the JDataStore database file renames the existing file (by prepending “BackupX_of_” where X is an auto-incrementing number), and copies all the streams to a new version of the database, using the original name.

To pack the database, select Tools | Pack JDataStore.

To pack the database programmatically, use the `com.borland.datastore.DataStoreConnection.copyStreams()` method.

Upgrading the JDataStore database

JdsExplorer can open older versions of the JDataStore database file format. When you open a database that is stored in an older format, the only available operation is to upgrade the file to the current version. Upgrading the JDataStore database file renames the existing file (by prepending “BackupX_of_” where X is an auto-incrementing number), and copies all the streams to a new version of the database, using the original name.

To upgrade the database, select Tools | Upgrade JDataStore. When the current JDataStore is the current version, this menu option is disabled.

To upgrade the database programmatically, use the `com.borland.datastore.DataStoreConnection.copyStreams()` method.

Deleting a JDataStore database

You can use JdsExplorer to delete a JDataStore database and its transaction log files. To delete a JDataStore database, follow these steps:

1. Open the database you want to delete.
2. Select Tools | Delete JDataStore. A confirmation dialog box appears.
3. Click Yes to delete the JDataStore database.

Closing a JDataStore database

To close the current JDataStore database, select File | Close. To close all open JDataStore databases, select File | Close All.

Encrypting and unencrypting a JDataStore database

This topic describes how to encrypt and unencrypt databases using JdsExplorer. For information on encrypting databases programmatically, see "JDataStore encryption" on page 74. Chapter 7, "Using JDataStore's security features" also contains a fairly extensive description of how JDataStore handles encryption

Encrypting a database

1. In JdsExplorer, open the database you want to encrypt.
2. Choose Tools|Encrypt

   JDataStore encrypts the database after making a copy of the original unencrypted database.

Unencrypting a database

1. Create a new, empty database
   a. In JdsExplorer, choose File|New or click the New JDataStore button to display the New JDataStore dialog box.
   b. Select the properties that you want the new database copy to have.
   c. Click okay to create the empty database.
2. Open the database you want to unencrypt.
3. Choose Tools|Copy Into Existing JDataStore.
4. In the Select JDataStore to Copy Into dialog box, choose the empty database that you just created.
5. Choose Open.

JDataStore copies the selected open database into the new database. The new database created by the copy operation is always unencrypted, regardless of whether the source database was encrypted.

Table and file streams

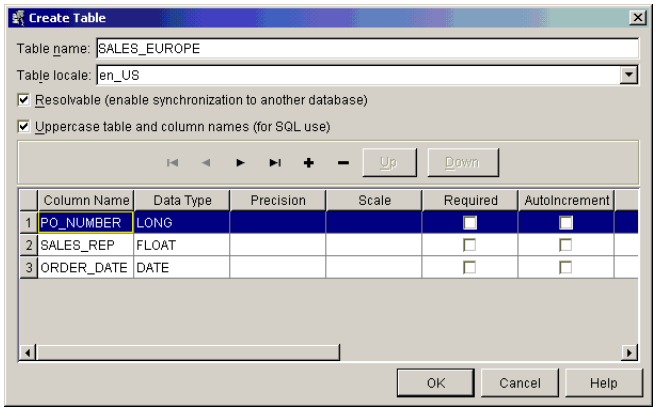
A JDataStore database can contain both table streams and file streams.

Creating tables

You can use JdsExplorer to visually create new tables for a JDataStore database. To create a table:

1. Open JdsExplorer. (If you are in JBuilder, choose Tools | JdsExplorer.)
2. Choose File | Open in JdsExplorer and select the database for which you want to create a new table.

3 Choose Tools | Create Table to open the Create Table dialog box.



4 Type a name for the new table in the Table Name field.

5 Select a locale if you are internationalizing the table. Otherwise, leave the value <null>.

6 Click the Insert New Row button on the Navigation bar to create a new row.

7 Click in the Column Name field and type the name of the new column.

8 Click in each of the column property fields on that row you want to define, and select or enter a value. For each column, you must specify at least a column name and data type. You may also specify other properties as needed. (See the `Column` class for a description of the `Column` properties.)

9 Continue creating the rest of the columns in this manner, rearranging their order in the table as desired. Use the Navigation bar buttons to add or insert additional rows, move to different rows, and rearrange the rows you have added.

10 Click OK when you are finished creating and defining all the columns.

For more information on using the Create Table dialog box, click the Help button on the dialog box.

Modifying tables

You can modify an existing table's structure in JdsExplorer.

1 Select a table in the JdsExplorer tree.

2 Click the Structure tab.

3 Follow the steps given above for creating a table. The UI for the Structure tab is the same as the Create Table dialog box.

Data types

JDataStore provides the following data types. See "Data types" on page 86 for a list of corresponding SQL data types.

Table 5.3 JDataStore data types

Java data type	Description
`short`	Exact numeric with a range of -32768 to 32767 *bytes*: 1 to 3
`int`	Exact numeric with a range of -2147483648 to 2147483647 *bytes*: 1 to 5
`long`	Exact numeric with a range of -9223372036854775808 to 9223372036854775807 *bytes*: 1 to 9
`java.math.BigDecimal`	Exact numeric with a precision of *p* digits and *d* decimals. The precision is limited to 72 digits. Default value of *p* is 72 and of *d* is 0. *bytes*: 1 to 32
`double`	Approximate numeric with a range of 4.9E-324 to 1.8E308 and a precision of 52 bits. *bytes*: 1 to 9
`float`	Approximate numeric with a range of 1.4E-45 to 3.4E38 and a precision of 23 bits. *bytes*: 1 to 5
`java.lang.String`	Variable length string with a max length of *p* characters and a max inline of *m* bytes. ■ When the string contains more than *m* bytes, the rest of the string is stored as a Blob. ■ Default value of *p* is unlimited and of *m* is 64. *bytes*: 1 to *m*
`byte[]`	Variable length binary with a max length of *p* bytes and a max inline of *m* bytes. ■ When the data contains more than *m* bytes, the rest of the data is stored as a Blob. ■ Default value of *p* is unlimited and of *m* is 64. *bytes*: 1 to *m*
`java.io.Serializable`	Variable-length binary that contain a Java object of type *t* and a max inline of *m* bytes. The Java object is stored using Java serialization. If *t* is omitted, any type can be stored; default for *m* is 64. *bytes*: 1 to *m*
`boolean`	A two-valued type: `true`/`false` *bytes*: 1
`java.sql.Date`	A local date after the year 0. *bytes*: 1 to 9
`java.sql.Time`	A local time in the range 00:00:00 to 23:59:59. *bytes*: 1 to 9
`java.sql.Timestamp`	A timestamp in Greenwich Mean Time (GMT). In contrast to `DATE` and `TIME`, the value read depends on the time zone of the machine where it is read. A `TIMESTAMP` can contain BCE dates. *bytes*: 1 to 9

DATE and TIME data types

The `DATE` and `TIME` data types store local date and local time. Prior to JDataStore 6.0, they stored GMT time. However, this approach was problematic.

`DATE` and `TIME` values are now independent of the time zone JDataStore is running in. A `TIME` value of 09:00:00 in San Francisco displays as 09:00:00 in New York.

The `TIMESTAMP` data type is stored in GMT time. Thus a `TIMESTAMP` of 2002-05-25 09:00:00 in San Francisco displays as 2002-05-25 12:00:00 in New York.

The FLOAT data type

Prior to JDataStore 6.0 the `FLOAT` data type was equivalent to the Java `float` type. However, according to the SQL92 standard, the `FLOAT` data type is an abbreviation of `FLOAT(p)` where *p* is the largest supported precision in bits of this floating point number.

This change means that the corresponding SQL type to a Java `float` is `REAL`. For JDataStore, a `FLOAT` is now an abbreviation of `FLOAT(52)` which corresponds to a Java `double`.

- `FLOAT(p)` where *p* is 1 through 23 corresponds to a Java `float`.
- `FLOAT(p)` where *p* is 24 through 52 corresponds to a Java `double`.

Data type coercions

When you change (coerce) the data type of a table column, you should be aware of the possible consequences. The following table describes what happens when a data type is coerced to another data type. The data types on the left indicate the original data type of the column with the data types listed along the top of the table indicating the new data type of the column.

Table 5.4 Consequences of different data type coercions

From\To	Big Decimal	Double	Float	Long	int	Short	boolean	Time	Date	Time stamp	String	Input Stream	Object
Big Decimal	None	Prec	Prec	Prec	Prec	Prec	Prec	Prec	Prec	Prec	OK	Loss	Loss
Double	Prec	None	Prec	Prec	Prec	Prec	Prec	Prec	Prec	Prec	OK	Loss	Loss
Float	Prec	OK	None	Prec	Prec	Prec	Prec	Prec	Prec	Prec	OK	Loss	Loss
Long	OK	Prec	Prec	None	Prec	Prec	Prec	Prec	Prec	Prec	OK	Loss	Loss
int	OK	OK	Prec	OK	None	Prec	Prec	Prec	Prec	Prec	OK	Loss	Loss
Short	OK	OK	OK	OK	OK	None	Prec	Prec	Prec	Prec	OK	Loss	Loss
boolean	OK	OK	OK	OK	OK	OK	None	Prec	Prec	Prec	OK	Loss	Loss
Time	Prec	Prec	Prec	Prec	Prec	Prec	Prec	Prec	Prec	Prec	OK	Loss	Loss
Date	Prec	Prec	Prec	Prec	Prec	Prec	Prec	None	None	Prec	OK	Loss	Loss
Time stamp	Prec	Prec	Prec	Prec	Prec	Prec	Prec	Prec	Prec	None	OK	Loss	Loss

Table legend:

- Loss: All data lost in this coercion.
- None: No coercion necessary.
- Prec: Potential precision loss with this coercion.
- OK: No data lost with this coercion.

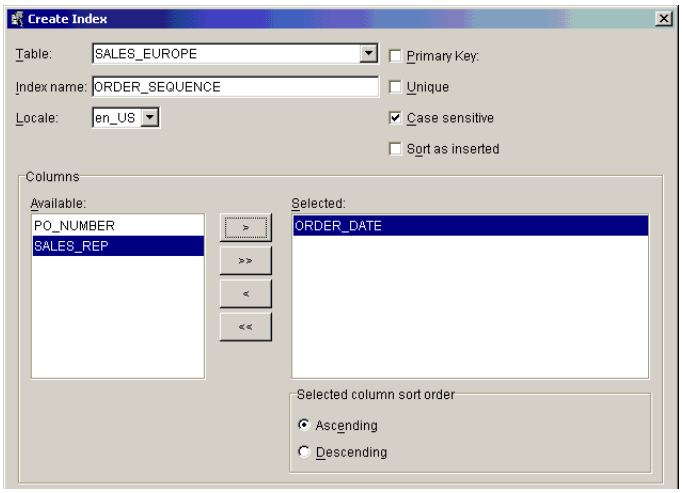
Creating indexes

You can use JdsExplorer to visually create a secondary index for a JDataStore table. To create an index:

1 Open JdsExplorer. (If you are in JBuilder, choose Tools | JDataStore Explorer.)

2 Choose File | Open in JdsExplorer and select a database.

3 Choose Tools | Create Index in JdsExplorer to open the Create Index dialog box.



4 Choose the name of the table for which you want to create an index in the Table Name drop-down list.

5 Type the name for the index in the Index Name field.

6 Specify a Locale object to use for determining how to do the sort if needed. Otherwise, leave the value `<null>`.

7 Check Unique if you want the combined results of the selected columns to be unique for every row.

8 Check Case Insensitive if you don't want the sort to match items by case.

9 Check Sort as Inserted if you want new rows to remain where they were inserted.

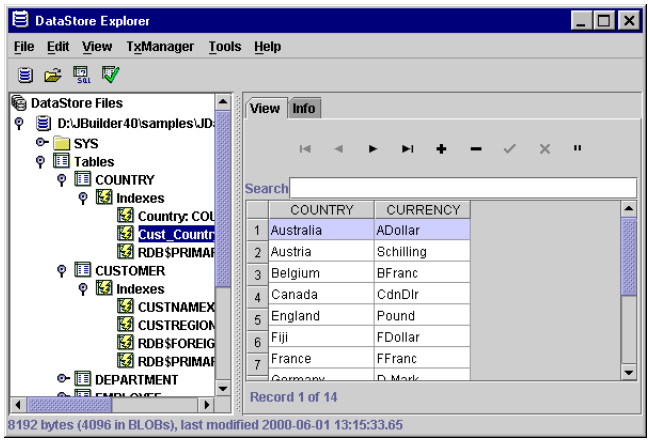
10 Select which columns to include in the sort. Use the arrow buttons to move columns from the Available list to the Selected list and back.

11 Specify the sort order as Ascending or Descending for each column in the Selected list.

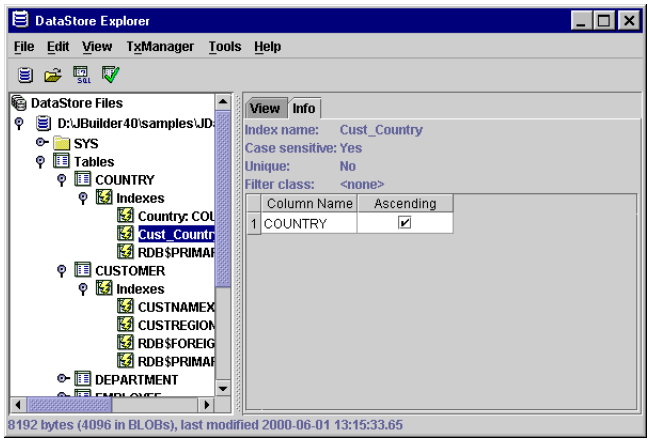
12 Click OK when you are finished specifying the index criteria. The index is added to the list of indexes for that table in the tree at the left of JdsExplorer.

Two tabs are now present in the right half of JdsExplorer: View and Info.

- The View tab shows the results of the sort for the selected index.



- The Info tab shows the properties of the selected index.



For more information on using the Create Index dialog box, click the Help button on the dialog box.

Adding a file stream to a database

You can add files to a JDataStore database by following these steps:

1. In JdsExplorer, highlight the name of the database to which you are adding a file stream.
2. Choose Tools | Import | File to display the Import File dialog box. Use the button next to the Filename field to navigate to the file you want to add. You can add any type of file you want, but you can view only text, GIF, JPEG files.

3 Optionally, you can change the name in the “Store name” field, to change the name of the stream within the JDataStore database. If you use name that already exists within the database, the old stream is overwritten with the one you are adding.

4 Click OK to add the new file stream to the database.

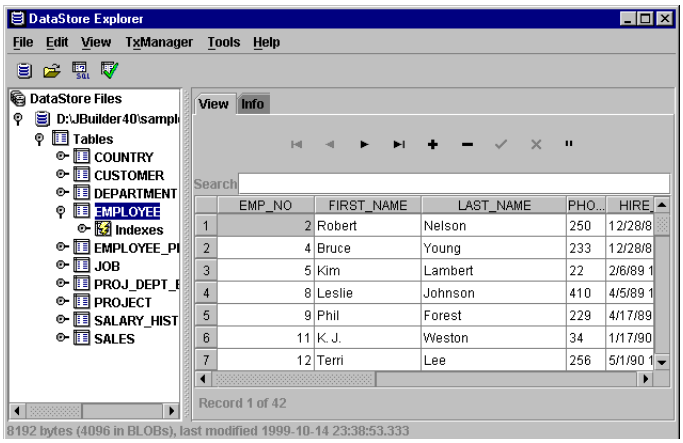
Viewing stream contents

The JDataStore directory appears in the tree control on the left. The directory uses forward slashes (“/”) in the stream names to synthesize a hierarchical structure. In addition, known stream types such as tables and text files appear under their corresponding node in the tree.

You can use the View | Expand All and View | Collapse All menu items to help manage the directory tree.

When you select a stream in the tree, the stream contents display if there is an appropriate viewer. There is a built-in viewer for table streams.

Figure 5.6 JdsExplorer displaying a table stored in JDataStore



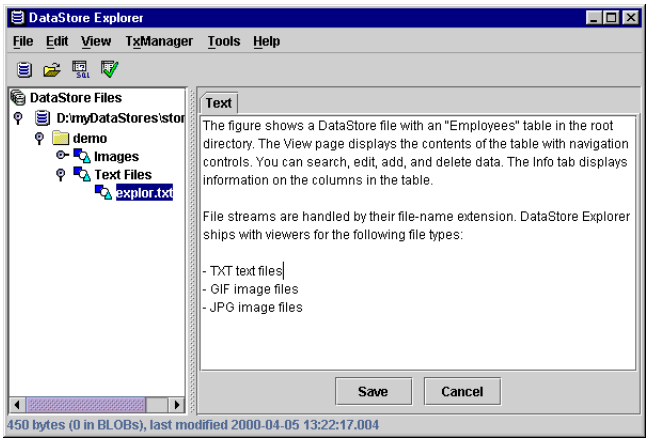
The figure shows a JDataStore database with an “Employees” table in the root directory. The View page displays the contents of the table with navigation controls. You can search, edit, add, and delete data. The Info tab displays information about the columns in the table.

File streams are handled based on their file-name extension. JdsExplorer ships with viewers for the following file types:

- TXT text files
- GIF image files
- JPG image files

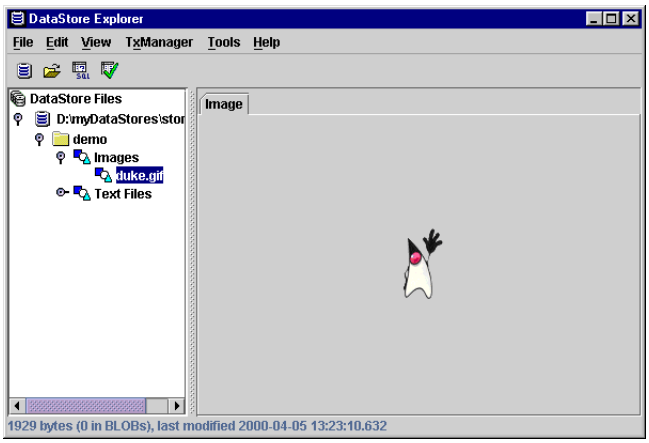
The following figure shows the “demo/explor.txt” text file displayed by JDSExplorer.

Figure 5.7 JdsExplorer displaying a text file stored in JDataStore



The next figure show the image file “demo/duke.gif”:

Figure 5.8 JdsExplorer displaying an image stored in JDataStore



Renaming streams

Use this operation to rename a stream or to move a stream to another directory:

1. In the directory tree, select the stream to rename/move. Note that you can’t rename deleted streams. You must undelete them first.
2. Select Edit | Rename.
3. Type a new name in the Rename dialog box. You can’t use the name of an existing active stream.
4. Click OK.

Deleting streams

When you delete a stream, the blocks it used are marked as available. It’s possible to undelete a stream, although some of the blocks might have been reclaimed by other streams.

For more information on deleting and undeleting streams, see “Deleting streams” on page 19, “How JDataStore reuses blocks” on page 19, and “Undeleting streams” on page 20.

To delete a stream:

1 Select the stream to delete in the directory tree.
2 Select Edit | Delete. The stream is marked as deleted in the tree.

Undeleting streams

If a deleted stream is visible in the directory tree, you can attempt to undelete it. You can't undelete a stream if there is another active stream with the same name in the same directory, because you can't have two streams with the same name.

Undeleting a stream doesn't guarantee all the data in the stream will be recovered. See "How JDataStore reuses blocks" on page 19 for more details.

To undelete a stream using JdsExplorer:

1 Select the deleted stream to undelete in the directory tree.
2 Select Edit | Undelete. The deleted mark is removed from the stream icon.

For an example of how to delete and undelete streams programmatically, see "Demonstration class: DeleteTest.java" on page 195.

Queries

This section explains how to create queries in three different ways:

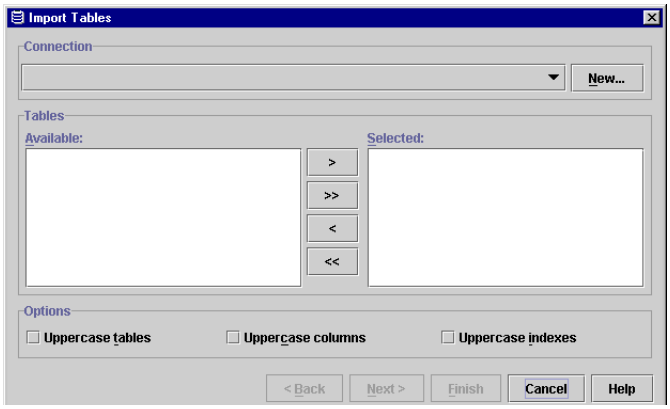
- Using Tools | Import | Tables
- Interactive SQL query
- Executing a file that contains SQL queries

Creating and maintaining queries and connections

To use JdsExplorer to manage queries or to import a table from another database, you must have an open JDataStore database.

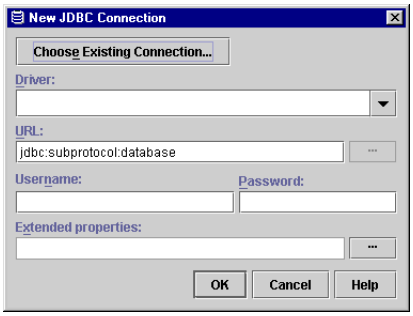
1 To define a query, select Tools | Import | Tables to open the Import Tables dialog box:

Figure 5.9 First page of Import Tables dialog box



2 The first time you define a query, there won't be any connections to associate it with. The New button next to the Connection field lets you define a new connection through the New JDBC Connection dialog box.

Figure 5.10 New JDBC Connection dialog box

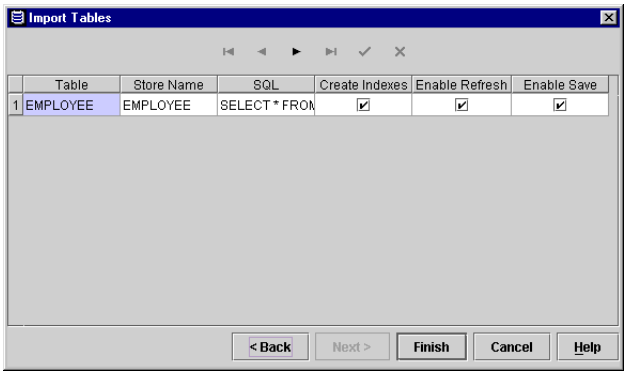


Enter the same parameters that you would enter in the Connection Property editor for a database: JDBC Driver Name, URL, Username, and Password. You can also specify extended properties for the connection.

When defining queries later, you can choose an existing connection or define a new one.

3 Once you have a connection to a database, you see a list of available tables. After selecting the desired tables, you can click Finish to simply import those tables. If you want more control over what is imported, click Next to go to the next page.

Figure 5.11 Second page of Import Tables dialog box



This page lists all the tables that will be imported, including the following information:

- The original name of the table in the database to be imported.
- The StoreName of the table stream that will be created in the JDataStore database. This defaults to the original name, but you can change it. It can include a path such as `Data/Tutorial/Employee`. JdsExplorer's tree pane displays this as a data set named `Employee` in a folder named `Tutorial` which in turn is in a folder named `Data`.
- The SQL statement used to retrieve the data; you can edit this statement to change the columns or selection criteria.
- Check boxes that control whether indexes should be created, and whether refresh and save should be enabled for this query. Refresh and save settings for the query are saved to the SYS/DataStore Queries/Queries table. If you have read-only tables, you should not enable Save. If you know the data in a table is not going to change, you should not enable Refresh.

4 Click Finish to import the data and store the queries.

The first time you open the Import Tables dialog box, two empty table streams named `SYS/Connections` and `SYS/Queries` are created. Queries that you create are saved to the SYS/Queries table, and connections you create are saved to the SYS/Connections table. When you finish defining the first query by clicking OK, each table has one row.

To maintain connections or queries, select the Connections or Queries table under the `SYS/DataStore Queries` branch in the JdsExplorer tree. You can:

- View and modify existing connections and queries.
- Delete a connection or query definition.
  To do so, select it and press "–" on the navigation toolbar or press *Ctrl+Del*.
- Insert a new definition.
  To do so, press "+" on the navigation toolbar or press *Ctrl+Ins*.

Retrieving and editing replicated data

As soon as you save a new query, JdsExplorer attempts to execute that query to fetch its data. You'll see the tree pane of JdsExplorer update to show the newly imported table using the database name you specified. After that, you can re-execute the query to refresh the data manually. Note that refreshing data discards any unsaved changes.

To execute a single query, select it in the "SYS/Queries" table, click Refresh Table, and respond "Yes" to the warning about unsaved changes.

To view a table, select it in the JdsExplorer tree. This displays the table in a grid on the right side of the Explorer.

After editing, you can save your changes or discard them. To discard changes, refresh the data set and respond "Yes" to the warning about unsaved changes.

Saving changes and refreshing data

You can save changes and refresh data on three different levels:

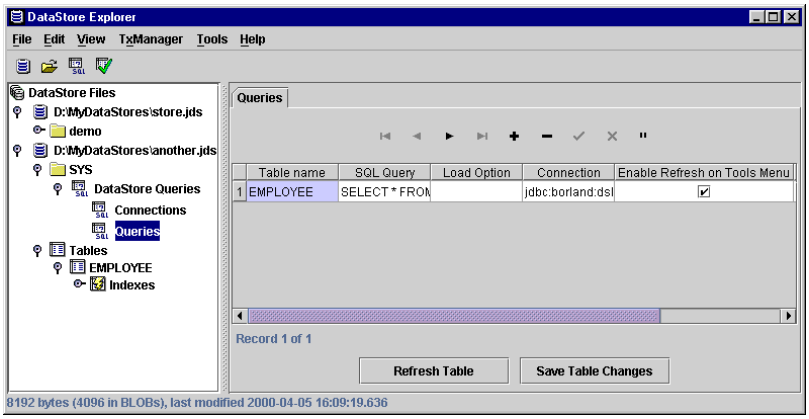
- Individual queries
- All the queries for a particular connection
- All the queries in all the connections stored in the database

To refresh or save changes from a single table, select the row in `SYS/Queries` for the query that creates that table. Buttons labeled Refresh Table and Save Table Changes are available, indicating that only the table provided by that query will be affected.

To refresh or save changes from all the tables for a connection to a database, select that connection's row in the "SYS/Connections" table. Buttons labeled Refresh Connection Queries and Save Connection Changes are available, indicating that all tables produced by querying that connection's database will be affected.

To refresh or save changes from all the tables for which you've defined queries through JdsExplorer, select Tools | Refresh JDataStore or Tools | Save JDataStore Changes. These commands re-execute every query or save changes for every table that has an associated query, for those queries that have their Enable Refresh and Enable Save options enabled. You can change these settings for each query in the "SYS/Queries" table.

Figure 5.12 Entries in the SYS/Queries table



Automating heterogeneous replication and synchronization

JdsExplorer utilizes DataExpress JavaBean components to provide automatic replication and synchronization technology. The level of replication available supports replication for a simple client-server relationship. Synchronization from the client to the primary server is optimized and configurable. Synchronization from the server to the client is not optimized. It requires execution of a query to obtain a fresh snapshot. A JDataStore database is always used as the client. The server can be any database that provides a JDBC driver. While some applications may require a more sophisticated solution, there are some key benefits to the DataExpress solution:

- A JDataStore database can replicate and synchronize with any database that has a JDBC driver.
- No software needs to be installed on the server. More advanced solutions require server-side software/trigger deployment.
- Replication and synchronization operations can be performed in a single transaction.

While this solution can be achieved programmatically using DataExpress JavaBean components, JdsExplorer can automate this process with the following facilities:

- The Tools | Import | Tables menu makes it easy to define JDBC connections and queries used for replication. These connections and query definitions are persistent in system tables.
- The Tools | Save Changes menu automatically saves edits to replicated data back to the server.
- Tools | Refresh menu option to automatically provide a fresh snapshot of replicated data from the server.

The connection information and SQL query statements, which are usually embedded in your application code, are saved in the JDataStore database in two system tables named “SYS/Connections” and “SYS/Queries.”

Synchronization error handling

By default, the synchronization initiated by JdsExplorer is all or nothing. If any update conflicts or constraint violations are encountered, the synchronization transaction is rolled back.

Custom synchronization

The synchronization process can be customized programmatically by using the same DataExpress JavaBeans that JdsExplorer does. These components have various properties, methods, and events for customizing synchronization behavior and handling errors. If you need to perform custom synchronization, the documentation for the following components may be of help:

- `com.borland.datastore.DataStorePump`
  Imports/refreshes one or more replicated tables
- `com.borland.datastore.DataStoreSync`
  Saves edits made to one or more replicated tables back to the server
- `com.borland.dx.sql.dataset.QueryDataSet`
  Refreshes and saves edits to tables using SQL queries
- `com.borland.dx.sql.dataset.ProcedureDataSet`
  Refreshes and saves edits to tables using stored procedures

- `com.borland.dx.sql.dataset.Database`
  Wrapper object for java.sql.connection
- `com.borland.dx.sql.dataset.SqlResolutionManager`
  Coordinates a synchronization across one or more tables

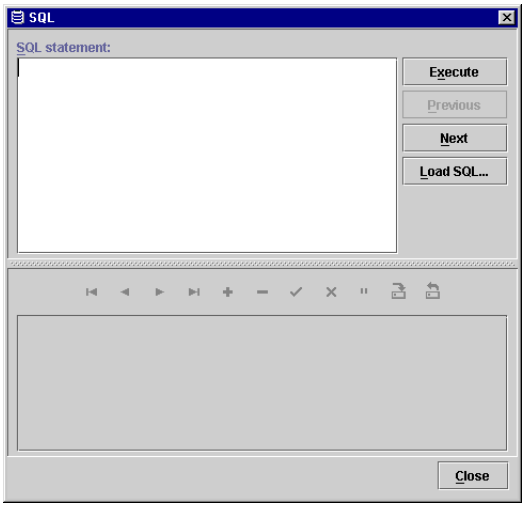
Executing SQL interactively or from a file

You can execute arbitrary SQL statements against the current JDataStore database. If the database is not transactional, you can perform only read-only queries. If an open non-transactional JDataStore database is not already in read-only mode, it is automatically closed and reopened in read-only mode.

To execute SQL statements, open the SQL dialog box by performing one of the following actions:

- Select Tools | SQL
- Click the SQL toolbar button

Figure 5.13 SQL dialog box



- You can type in SQL statements directly.
- Statements that you type are recorded and you can scroll through them with the Previous and Next buttons to modify and re-execute recorded statements.
- You can execute files containing SQL statements.
- Result sets returned by SQL statements are displayed in the lower half of the dialog box.

Importing text into tables and files

In addition to importing tables from other databases, JdsExplorer makes it easy to import delimited text files as table streams and arbitrary files as file streams.

Importing text files into database tables

The contents of the text file must be in the delimited format that DataExpress uses for export, and there must be a `SCHEMA` file with the same name in the directory to define the structure of the target data set.

SCHEMA files, which end with a .schema filename extension, are created when you export a JDataStore table to a text file using the com.borland.dx.dataset.TextDataFile.save() method. It's recommended that you export data from your table to generate the SCHEMA file. To give you an idea of what one looks like, here is one for a simple three-column table:

```
[]
FILETYPE = VARYING
FILEFORMAT = Encoded
ENCODING = Cp1252
DELIMITER = "
SEPARATOR = 0x9
FIELD0 = ID,Variant.INT,-1,-1,
FIELD1 = Name,Variant.STRING,-1,-1,
FIELD2 = Update,Variant.TIMESTAMP,-1,-1,
```

This SCHEMA file defines the double quote as the string delimiter and the tab character as the field separator. There are three columns, an integer, a string, and a timestamp.

Once you have a SCHEMA file to accompany the text file, follow these steps to import the text file as a table:

1. Select Tools | Import | Text Into Table. This opens the Import Delimited Text File dialog box.
2. Supply the input text file and the database name of the data set to be created. Because this operation creates a table rather than a file stream, you'll probably want to omit the extension from the database name.
3. Click OK.

Importing files into JDataStore databases

To import a file as a file stream:

1. Select Tools | Import | File.
2. Supply an input file name and the store name of the file stream to be created.
3. Click OK.

If the selected file has subdirectories, those subdirectories are also imported and are created in the JDataStore database file structure.

JDataStore security

You can perform the following security-related tasks using JdsExplorer:

Administering Users

To administer users for a JDataStore database, display JdsExplorer and select Tools | Administer Users. If an administrator has not previously been defined for the database, you receive a prompt for the administrator's user name and password when you select this menu option. When you choose an administrator user name and password, the administrator user is added and is assigned all rights for the database by default. The process of adding a user is also called *registering* a user.

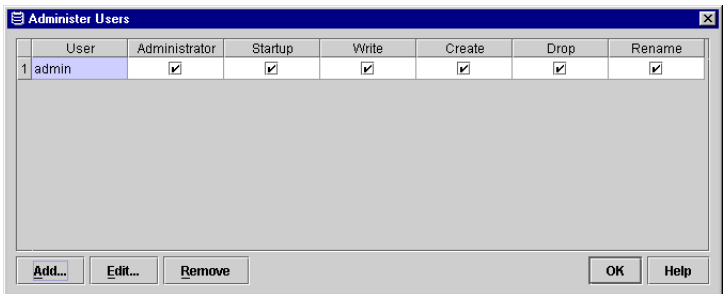
If you are logged in as a user with administrator's rights, the Administer Users dialog appears. If a user without administrator's rights tries to open this dialog, they are

prompted for an administrator's user name and password. The Administer Users dialog allows the administrator to add, edit, and remove users, and assign rights to each user.

This section discusses how to administer users using JdsExplorer.

- To performs these tasks programmatically, use `com.borland.datastore.DataStoreConnection`, which has `addUser`, `removeUser`, `changeRights`, `changePassword`, and `encrypt` methods, plus an `isProtected` method to test whether the database is password protected. These topics are also discussed in Chapter 7, "Using JDataStore's security features."
- JDataStore also implements SQL commands for managing users. See "CREATE USER" on page 130, "ALTER USER" on page 130, and "DROP USER" on page 131.

Figure 5.14 The Administer Users dialog box

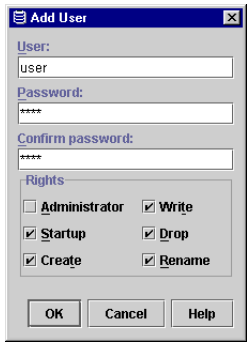


The existing users are displayed in a table. Checkboxes in the table columns indicate which rights are assigned to each user. For an explanation of the various rights, see "Authorization" on page 74.

Adding a user

To add a user, display the Administer Users dialog box (see "Administering Users" on page 54) and click the Add button to display the Add User dialog box.

Figure 5.15 The Add User dialog box



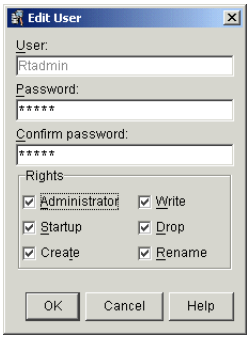
Enter a name for the new user. Type the user's password in the Password field, and then type it again in the Confirm password field to confirm the password. Users are able to change their own passwords when they log in.

Select which rights the user should have. For an explanation of the various rights, see "Authorization" on page 74. Click OK when you are done assigning rights to the user.

Editing a user

To edit a user, display the Administer Users dialog box (see “Administering Users” on page 54) and select the row for the user you want to edit. Click the Edit button to display the Edit User dialog box.

Figure 5.16 The Edit User dialog box



You can edit the rights for the selected user, but not the user name or password. Choose the rights you want to give the user and then choose OK.

Removing a user

To remove a user, display the Administer Users dialog box (see “Administering Users” on page 54), select the row for the user you want to delete, and click the Remove button. At the prompt, click Yes to confirm that you want to remove the user. You cannot remove the only administrator.

Changing a password

A user must be currently logged in to change their password. To change the password for the current user, display JdsExplorer and select Tools | Change Password to display the Change Password dialog box.

Figure 5.17 The Change Password dialog box

You must enter the old password, then enter the new password twice for confirmation. Click OK.

Chapter

6

The JDataStore Server Console

Overview

The Server Console—introduced in JDataStore 7—provides an integrated user interface for performing a large number of tasks relating to databases, datasources, remote connections, and monitoring. Specifically, it is possible to:

- Create and manage mirrors for failover and incremental backup
- View and optionally terminate the connections made to a server
- View and change a number of database settings
- View current table and row locks
- View the database status file
- Verify a database
- Use Interactive SQL in the ISQL window

Note that Server Console allows you to manage connections remotely. In versions prior to JDataStore 7, it was necessary to log onto the server itself to manage connections to that server.

About datasources

A datasource is a named collection of connection parameters. The exact options available depend on the connection class used in the datasource className. By default, JDataStore uses `com.borland.javax.sql.JdbcDataSource`.

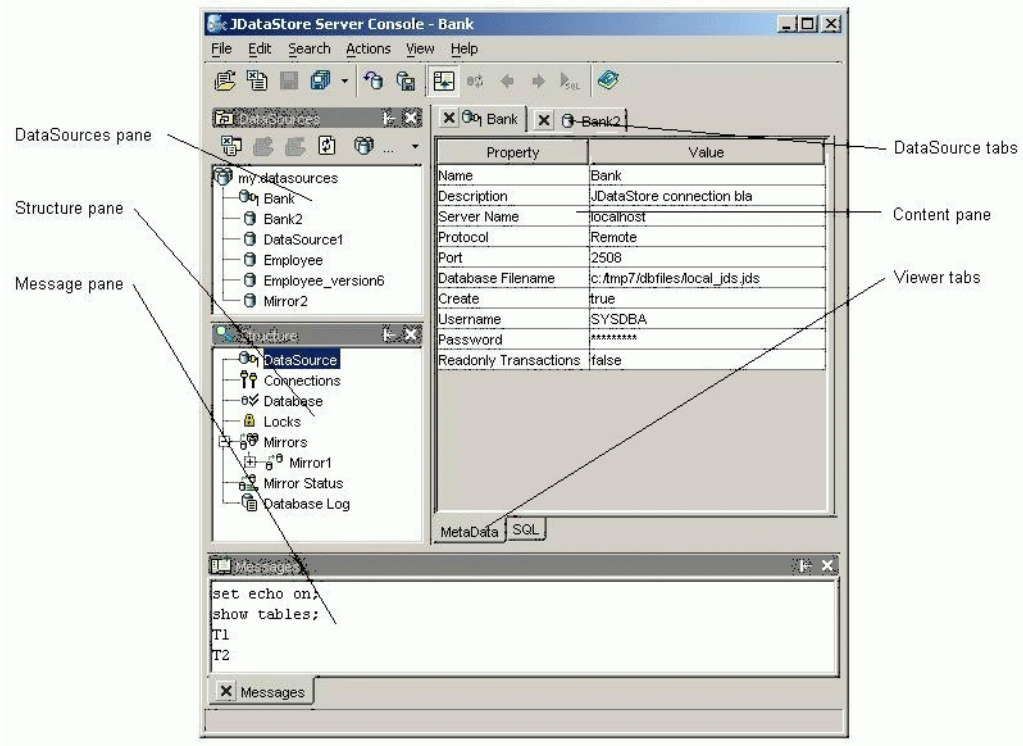
To connect to a database, you connect to a *datasource*, rather than naming the database file and all the connection parameters explicitly.

When you create a datasource, JDataStore stores it, by default, in the `my.datasources` file that is in the system user directory for the current user. This `my.datasources` is called a *project file* and it contains editable information about all the various datasources that you create. It is possible to create and use other project files, although for ease of use, JDataStore recommends using the default.

If you need or want to create other project files, you can do this by choosing File|New Project in the Server Console.

For more detailed information about datasources, see "Connection settings" on page 60.

The Server Console interface



The Datasources pane

The **Datasources pane** contains a tree view of a datasources project file—by default, `my.datasources`—where the root is the datasources project file itself, and the nodes are the names of the datasources saved in the file.

The Structure pane

Double-click a datasource in the Datasources pane to display its structure nodes in the **Structure pane**. The contents of the Structure pane depend on whether the connection is local or remote. When the datasource is not connected at all, only the DataSource node shows in the Structure pane. The Structure pane contains the following nodes for the selected datasource:

For all connections

The following items appear in the Structure pane for both local and remote connections. The description tells what appears in the Contents pane when the structure item is selected.

- **Datasource** Displays the connection properties of the selected datasource; these properties are editable when you are not connected through the datasource. You can also display these connection properties by double-clicking the datasource in the Datasources pane.

- **Database** Displays database properties:
  - **Directory for Log Files** specifies the directory for both transaction log files and database status log files
  - **Block Size of Log Files**, **Logs Enabled**, **Log Files Preallocated**, **Max Log Size** and **Soft Commit Enabled** refer only to the transaction log files.
  - **Record Status** Specifies whether a database status log file should be maintained. Note that `DB_ADMIN` contains a number of methods for determining and retrieving the contents of this file.
- **Locks** Displays a table of detailed information about current database locks
- **Database log** Displays the database status log.

For remote connections only

The next three Structure pane items appear only if the selected datasouce is a remote connection:

- **Connections** Displays a subset of the properties in the `com.borland.javax.sql.JdbcDataSource` datasource class.
- **Mirrors** Displays a list of mirrors for the selected database; this is also where you define, alter, and delete mirrors, as well as scheduling synchronization.
- **Mirror Status** Displays a read-only display that gives the status of each defined mirror.

The Content pane

The Content pane is the main window area. It has two tabs at the bottom: MetaData and SQL. At the top are Datasource tabs.

- **DataSource tabs** There is one **Datasource tab** for each open datasource. To make a datasource **active**, click its Datasource tab. This is then the datasource on which all operations are performed.
- **Metadata tab** This tab shows various properties for the active datasource (the active Datasource tab). If the datasource is not connected, the connection properties can be edited here. When the datasource is connected, many other properties can be viewed. First click the Datasource tab to activate that datasource, then click a node in the Structure pane to see the desired properties in the Content pane MetaData tab.
- **SQL tab** Click the SQL tab at the bottom of the Content pane to display a work area in which you can run interactive SQL commands against the active datasource

The Message pane

The Message pane contains output from the interactive SQL window. Depending on its nature, this output is either textual or a data grid.

Starting and stopping the server

In order to connect to a datasource using Server Console, the server must be running on the machine where the database resides. You can use Server Console to start the local server. It cannot start remote servers, although it can view and terminate connections on remote servers.

To start or stop the local server

Click the Actions menu. There is a pause while JDataStore checks to see whether the local server is running. Then the Actions menu opens and displays the appropriate command: If the server is running, you are offered a command for stopping the server. If the server is not running, you are offered the option of starting it. The menu item changes to reflect the local server status.

When you connect to a local datasource, JDataStore checks to see whether the local server is running. If it is not, you are given the option to start it and to continue with the connection. If you attempt to make a remote connection when the remote server is not running, the attempt fails with an error.

Managing datasources and databases

Connection settings

A **datasource** is a named collection of connection settings. When you create a datasource using the Server Console, those settings are saved in a project file, which always has a `datasources` extension. By default, the Server Console uses the `my.datasources` file that is in the system home directory for the current user.

It is possible to manage connections using a different project file, but it is not recommended. To create a new project file, choose File|New Project. To open a different project file, choose File|Open Project.

The `my.datasources` file can be deployed by applications that use the com.borland.dx.DataSourceSet Java Bean component, which knows how to load this file into a set of datasources.

The connection settings are given as properties on the `com.borland.javax.sql.JdbcDataSource` class. The Console handles only a limited set of the properties available in this class.

Adding a new datasource

1. In the Datasources pane, right-click `my.datasources` and select Add DataSource. This adds a new datasource node with the name DataSource*n* where *n* is the lowest unused number. All the connection settings are the default settings. You can now modify all these settings, including the connection name.
2. In the Datasources pane, double click the new datasource node, or right click the new node and select Open. This displays a property inspector for the selected datasource in the Content pane. If you are editing an existing datasource, be sure that the DataSource node is selected in the Structure pane. Note that you can edit datasource properties only when the datasource is not connected.
3. To edit the datasource, click the field that you want to edit and supply the information:
   a. Name: Originally is the same as the name of the datasource from the Datasources pane. This is where you can supply your own name for the datasource.

b Description: An informative description for remembering what this connection is about.

c Server Name: The name (or IP address) of the server machine.

d Protocol: Select either Remote or Local.

e Port: If the remote protocol is used, specifies which port is the server listening on; the default is 2508.

f Database Filename: The file specification of the database on the server machine.

g Create: True means create the database if it doesn't exist; False means fail if it doesn't exist.

h Username: Add the username you usually want to log in as.

i Password: This field is optional: if left blank, a connection dialog is presented before a connection can be made. Warning: The password is NOT encrypted in the `my.datasources` file. So leave this blank if security is an issue.

j Readonly Transactions: False means open for write operations, True means disallow write operations. You must specify True when connecting to a readonly mirror.

Note After JDataStore is installed, there are datasources for all databases in the `samples` directory. You can delete these datasources at any time.

Connecting to a datasource

To connect to a dataSource right click the datasource in the Datasources pane and select Connect. If username and password were not supplied in the datasource properties, a dialog prompts for the username and password. After a successful connect, the icon of the datasource changes.

To disconnect from a dataSource right click the datasource node in the Datasources pane and select Disconnect.

Other datasource operations

To rename a datasource right-click the datasource node and select Rename, or display the datasource in the Content pane and change the name in the property inspector.

To delete a datasource right-click that datasource node in the Datasources pane and choose Delete DataSource.

Managing connections

You can view all connections for a server and can terminate a remote connection using Server Console.

- In the Datasource pane, double-click a datasource that is on the server you're interested in.
- **Viewing connections** In the Structure pane, click the Connections node. This displays a grid in the Content pane. The grid displays all connections to databases that are on the same server as the selected datasource. Each row in the grid is one connection. For each connection, the grid displays the connection ID, the name of the user making the connection, the filename of the database that the connection is to, the host name and IP address of the user, and the port that the server is using for the connection. One of the connections will always be the one that you created to look at the server connections.

- **Terminating a connection** To terminate a connection, select its row in the grid and click the (–) button.
- **Refreshing the view** To refresh the list of connections, click the Refresh button in the toolbar.

Managing database properties

You can view and change database properties by following these steps. Any changes that you make to properties are applied only after you have explicitly saved the changes using the Save or SaveAll toolbar buttons or File|Save All.

1. Make the desired datasource active: either double-click it in the Datasource pane, or click its Datasource tab in the Content pane.
2. Make sure that the Metadata tab is displayed in the Content pane.
3. If you plan to change any of the database properties, make sure that the database is shut down; there can be no connections to it except the current connection.
4. Click the Database node in the Structure pane to display the database properties for the selected datasource. You can edit these properties by clicking in the Value field. The properties are:
   - Block Size: The block size of the database in Kbytes.
   - Database File Version: The version of the actual database file (not the server version).
   - Directory for Logfiles: The directory in which the transaction log files should be stored.
   - Block Size of Log Files: Block size used to read and write the transaction log files.
   - Logs Enabled: Toggles whether transaction log files are enabled for this database; you must enable transaction log files to make use of the High Availability server's crash recovery features
   - Log Files Preallocated: Number of files preallocated for transaction log files.
   - Max Log Size: Maximum size of a transaction log file.
   - Max Open Logs: Maximum number of open transaction log files.
   - Record Status: Specifies whether a database status log file should be created and maintained.
   - Soft Commit Enabled: Enables faster commit at the risk of losing recent transactions if the system crashes

Transaction log files are binary files that contain a record of all transactions against the database. These files are used to create and update the mirrors that are used for AutoFailover and for incremental backups. See "Managing mirrors" on page 66 for more information.

Database status log files are text files that list server events such as shutdowns, restarts, errors, and SQL logging (if it is enabled).

Verifying a database

You can verify the contents of a database in Server Console. This step is not generally needed for transactional databases because JDataStore has automatic crash recovery. However, verification can be useful for nontransactional databases that might not have been properly shut down.

1. In the Datasource pane, ensure that the datasource you want to verify is connected. Then double-click the datasource, so that its elements appear in the Structure pane.

2 In the Content pane, make sure that the MetaData viewer tab is selected.

3 Make sure that the Structure pane is visible. If it is not, display it by choosing View| Structure (*Ctrl-Alt-S*).

4 In the Structure pane, right-click the DataSource node and select Verify Database. This displays the Message pane, which reports any errors or reports a successful operation.

Viewing table and row locks

You can display a read-only grid of all table and row locks help against a selected database by following these steps:

1 In the Datasource pane, ensure that the datasource you want to verify is connected. Then double-click the datasource, so that its elements appear in the Structure pane.

2 In the Content pane, make sure that the MetaData viewer tab is selected.

3 Click the Locks node in the Structure pane. This displays a grid in the Content pane, showing all table and row locks for the selected database. The grid includes the object name, user name, type of lock, and several other types of information.

To refresh this view, click the Refresh button in the toolbar.

Viewing the database status log file

JDataStore optionally maintains a read-only text file containing a log of server events such as startups, shutdowns, error, and SQL logging. To enable the status file for a database, set its database property to `True`. (`True` is the default for this property.) You can specify the location for this file and the transaction log file setting the Directory for Log Files database property. See "Managing database properties" on page 62 for more information about setting these properties.

1 In the Datasource pane, ensure that the datasource you want to verify is connected. Then double-click the datasource, so that its elements appear in the Structure pane.

2 In the Content pane, make sure that the MetaData viewer tab is selected.

3 Click the Database Log node in the Structure pane to display the database status log file in the Content pane.

To refresh this view, click the Refresh button in the toolbar.

Incremental backup, auto failover, and load balancing

This section describes how to use mirroring in JDataStore to provide automatic failover, incremental backup, and load balancing. These features are available in JDataStore 7 and higher. They do not work in earlier versions. If you have a database that was created in an earlier version of JDataStore, update it to at least JDataStore 7 before proceeding with any operations that require mirrors. See "Migrating older databases to JDataStore 7" on page 36 for more information about migrating older databases.

Using mirrors programmatically

This section is primarily devoted to describing how to use Server Console for managing mirrors. JDataStore offers a `DB_ADMIN` class that provides all this functionality and more for use in Java programming. For more information on DB_ADMIN methods, see `com.borland.datastore.driver.DB_ADMIN.html`. In addition, the methods of this class are

implemented SQL as built-in Java methods that can be called using the CALL statement. You do not need to first create a `JAVA_METHOD` alias.

Even if you plan to access these features only through Java or SQL, the following section should provide you with an overview of available functionality and issues associated with using it.

Overview

This section provides a practical approach to using mirrors for incremental backup, auto failover, and load balancing. For a discussion of mirroring architecture, please see "The JDataStore High Availability Server" on page 24.

You use mirrors to incrementally back up the primary database. When auto-failover is in place, mirrors also provide a mechanism whereby a mirror takes over as the primary database when the original primary mirror goes down or is isolated by network failure. Finally, directory mirrors provide load balancing, so that the workload is spread out appropriately between the primary database and its mirrors.

Types of mirrors

In the following discussion, the term *source database* means the database against which a mirror is synchronized. This source database can be the primary database (see the definition below), or it can be another mirror. This architecture reduces the load on the primary database. The source database for a mirror is specified as one of the mirror's properties.

The terms *primary mirror* and *primary database* are interchangeable.

There are three types of JDataStore mirrors.

- **Primary mirrors** A primary mirror is the only writable database in a group of mirrored databases. Initially, a JDataStore database has no mirrors. The first mirror that you create for a database does not make a copy of the database. Rather, it designates the original database as the *primary mirror*. In a collection of mirrored databases there must be one, and only one, primary mirror at all times.
- **Read-only mirrors** A read-only mirror is a copy of the primary database. It can be kept in synch with its source database in several different ways:
  - **Auto Failover** At one end of the spectrum, you can set the mirror's Auto Failover property to True. A mirror with Auto Failover enabled is a candidate for becoming the primary database in case the primary fails. When Auto Failover is enabled for a mirror, Instant Synchronization is also automatically set to True, which copies the transaction log from the source database to the mirror. Thereafter it automatically updates the log before any transaction is committed on the source database. That transaction log, which contains the records for all transactions that were run against the source database, is not actually replayed against the mirror until a synchronization—manual or scheduled—is performed, or until a failure occurs that causes the mirror to be designated as the new primary database. Auto Failover ensures that the mirror is always up to date.
  - **Instant synchronization** When a mirror has the Instant Synchronization property set to True, but not Auto Failover, a copy of the source's transaction log is maintained on the mirror. The new transactions in the log are replayed against the mirror whenever a scheduled or manual synchronization command is executed. The only difference between this configuration and Auto Failover is that the mirror is not a candidate for becoming a primary mirror is case of a failure that takes the primary mirror offline.

  - **Automatic synchronization** You can specify that the mirror is to be synchronized with its source database automatically at designated intervals. The interval can be given as a specified day and time of the week, as a time of day, or in milliseconds. When the scheduled time for the synchronization arrives, the log files are incrementally updated and the newer log records are replayed against the mirror to bring the database view up to date. For more information, see "Adding a synchronization schedule" on page 70.
  - **Manual synchronization** You can manually synchronize a mirror with its specified source database at any time. this updates the transaction log files and replays them against the mirror to bring the mirror up to date. For more information, see "Manually synchronizing a mirror" on page 68.
- **Directory mirrors** A directory mirror mirrors only the mirror configuration table and security tables from its source database. Directory mirrors are therefore very small. However, because they contain the mirror configuration table, they are aware of all the other mirrors in the group. When a directory mirror receives a connection request, it redirects that request to the primary mirror if it is a write request. If the request is for a read-only connection, the directory mirror redirects it to one of the read-only mirrors, distributing the workload among them.

Auto failover

When enabled, auto failover ensures that when a primary mirror fails, or when there is a network failure that isolates the primary mirror, an up-to-date mirror exists that automatically takes over as the primary mirror.

To enable auto failover, you must create two or more read-only mirrors that have the Auto Failover property set to True. The need for at least two mirrors has to do with the algorithm that an autofail-enabled mirror uses to determine whether or not it is a primary mirror, based on whether it can connect to a majority of existing read-only mirrors.

In most cases, you will want to enable the Auto Failover property for the original primary mirror as well as for at least two read-only mirrors.

Important Automatic failover is triggered when a client rolls back a transaction after detecting a failure. You must ensure that you trap exceptions and perform the rollback.

Incremental backup

A read-only mirror, by its very nature, is an incremental backup of its source database. It is never out of date by more than the specified synchronization interval if it has scheduled synchronization enabled. If the mirror has its Instant Synchronization property or Auto Failover property set to True, it is always up to date with the possible exception of uncommitted transactions on the source database.

To enable incremental backup, first prepare the original database by designating it as the primary mirror. Then create at least one read-only mirror. Typically, you would either enable the Instant Synchronization property for this mirror, or create scheduled backups at fairly small intervals. For step-by-step instructions on how to prepare a database, add mirrors, and schedule synchronization, see "Preparing a database to use mirrors" on page 67, "Adding a read-only mirror" on page 68, and "Adding a synchronization schedule" on page 70.

Load balancing

JDataStore's directory mirrors provide load balancing. A directory mirror is not a copy of another mirror. Rather, it contains the mirror configuration table from its source database. This table tells it what other mirrors exist, and where the primary mirror is. When a directory mirror receives a connection request, it redirects that request to the primary mirror if it is a write request. If the request is read only, the directory mirror directs the request to one of the read-only mirrors, tracking the load for each database. Load balancing becomes useful when you have several read-only mirrors in addition to the one primary mirror. To enable load balancing, first ensure that two or more mirrors exist, and then create at least one directory mirror.

Managing mirrors

The operations described below are applied only after they are explicitly saved. Use the Save or the SaveAll button to save the changes.

About mirror properties

The procedures in the following "Managing mirrors" section make much use of mirror properties. It will help you to understand these properties before you begin.

Viewing mirror properties

To see a mirror's properties, make sure that the MetaData tab is displayed in the Content pane. Then expand the mirror node in the Structure pane and click the mirror whose properties you want to see. The properties are displayed in a grid in the Content pane.

Changing mirror properties

Once you have displayed a mirror's properties in the Content pane, you can change any of them. First click in the Value field next to the property you want to change. For multiple-choice properties, such as Type or Auto Failover, you get a drop-down list, in which you choose your desired value. In text fields, such as Name and Database Filename, you double-click to edit the field. When you have completed the field, remember to press the *Enter* key.

Mirror properties

The following list describes each of the properties that are displayed in a mirror's property sheet.

- **Name** is the unique name that identifies a mirror. By default, JDataStore names mirrors sequentially as "Mirror1," "Mirror2," and so on. You can change the name to any SQL identifier. The values in the mirror Name properties are what appear in the Synch Mirror Name drop-down list.
- **Type** is one of three types: Primary, Readonly, or Directory. See "Types of mirrors" on page 64 for a description of each type of mirror.
- **Host** is the host name of the server on which the mirror should be located.
- **Port** is the port at which the host should listen. The default is 2508.
- **Database Filename** is the full path and filename of the mirror.

- **Synch Mirror Name** is the source database that the new mirror should be synchronized from. In order to reduce the load on the primary mirror, you can select another read-only mirror to be the source database for the new mirror. Typically, you would select a mirror that has the Auto Failover or Instant Synchronization property set to True, and that is synchronized frequently.
- **Auto Failover** designates the mirror as a candidate for primary mirror in case of failure of the current primary mirror. Values are True and False. When Auto Failover is set to True, Instant Synchronization is automatically enabled as well. This causes the transaction log file from the source database to be copied to the current mirror. Transactions against the source database are copied to the log file on the mirror before being committed. Note that this transaction log file is replayed against the current mirror only when a scheduled or manual synchronization occurs, or when failure causes the current mirror to be designated as the new primary database.
- **Failover Priority** specifies the order in which auto failover-enabled mirrors should be considered for promotion to primary when the need arises. This is a numeric field in which you specify the desired order.
- **Instant synchronization** causes the transaction log file for the source database to be copied to the current mirror. Transactions against the source database are copied to the log file on the mirror before being committed. Note that this transaction log file is replayed against the current mirror only when a scheduled or manual synchronization occurs. A mirror that has Instant Synchronization enabled, but not Auto Failover, is kept fully synchronized with its source database, but is not a candidate for becoming a primary mirror.
- **Last known log** is the oldest log file still needed by this mirror database. Log files older than this will be dropped by that mirror. This value also affects when a primary or update mirror can drop their log files. The primary mirror and update mirrors will not drop their log files if any read-only mirrors still need to be synchronized with these log files. Periodically synchronizing read-only mirrors will increase the last known log setting. This in turn allows primary and update mirrors to drop older log files to free up disk space.

Preparing a database to use mirrors

Before you can add mirrors to a database, you “prepare” it by designating the original database itself as the primary mirror. You do this by adding the first mirror, which by default has its Type property set to Primary. This initial step does not create any copies of the original database, it just designates that database as the primary mirror. Every collection of mirrors requires one and only one primary mirror.

1. In the Datasource pane, ensure that the datasource you want to work on is connected. Then double-click the datasource, so that its elements appear in the Structure pane.
2. In the Content pane, make sure that the MetaData viewer tab is selected.
3. In the Structure pane, right-click the Mirrors node and choose Add Mirror from the context menu. This adds a node with the default mirror name of “Mirror1” under the Mirrors node and adds a line to the list of mirrors in the Content pane.
4. Highlight the Mirror1 node in the Structure pane to display its properties sheet in the Content pane.
5. Edit any fields that you want to change. By default, this first mirror is of Type Primary. You should not change this. You are free to change any other properties that you wish. Setting the Auto Failover property to True for this first mirror is recommended. It becomes significant only when the mirror comes back on line after a failure that has caused another mirror to become the primary. It simply means that this mirror is eligible to become the primary mirror again in case of further failures.

6 To undo all changes since you last saved, right click the datasource's tab at the top of the Content pane and choose Revert *datasource_name*.

7 To apply your changes, choose Save or Save All.

Adding a read-only mirror

Before you can add read-only mirrors to a database, you must prepare it by designating the original database as a primary mirror. This is described in "Preparing a database to use mirrors" on page 67.

When you have prepared the database by adding the original primary mirror, follow these steps to add as many read-only mirrors as you need.

1 In the Datasource pane, ensure that the datasource you want to work on is connected. Then double-click the datasource, so that its elements appear in the Structure pane.

2 In the Content pane, make sure that the MetaData viewer tab is selected.

3 In the Structure pane, right-click the Mirrors node and choose Add Mirror from the context menu. This adds a node with the new default mirror name under the Mirrors node and adds a new line to the list of mirrors in the Content pane.

4 Highlight the new mirror node in the Structure pane to display its properties sheet in the Content pane.

5 Edit any fields that you want to change. By default, all mirrors after the first one are added as Read-only mirrors. You can select another mirror type, bearing in mind that there can be only one primary mirror at one time. You must ensure that the Database Filename is unique. This property sheet is also where you set properties such as Auto Failover and Instant Synchronization.

6 To undo all changes since you last saved, right click the datasource's tab at the top of the Content pane and choose Revert *datasource_name*.

7 To apply your changes, choose Save or Save All.

Deleting a mirror

1 In the Datasource pane, ensure that the datasource you want to work on is connected. Then double-click the datasource, so that its elements appear in the Structure pane.

2 In the Content pane, make sure that the MetaData viewer tab is selected.

3 In the Structure pane, right-click the mirror to be deleted and choose Delete Mirror from the context menu. This causes the mirror node to disappear from the Server Console.

4 To undo all changes since you last saved, right click the datasource's tab at the top of the Content pane and choose Revert *datasource_name*.

5 To apply your changes, choose Save or Save All.

Manually synchronizing a mirror

Synchronizing a mirror causes the mirror to be brought current with it source mirror. Manual synchronization is one way to perform an incremental backup. To automate the process, see "Adding a synchronization schedule" on page 70.

1 In the Datasource pane, ensure that the datasource you want to work on is connected. Then double-click the datasource, so that its elements appear in the Structure pane.

2 In the Content pane, make sure that the MetaData viewer tab is selected.

3 In the Structure pane, right-click the mirror that should be synchronized and choose Synchronize Mirror from the context menu.

Because this operation may take some time, a dialog is displayed that allows you to cancel before the synchronization begins.

If there are any pending changes, those are first saved. Also note that this operation is carried out immediately once confirmed, in contrast to editing a mirror state, where you must explicitly save.

Notes

- You cannot synchronize the primary mirror.
- In order to synchronize a newly created mirror you must save your work first.

Synchronizing all mirrors

These steps describe how to manually synchronize all read-only mirrors simultaneously. You can automate synchronization by following the steps in “Adding a synchronization schedule” on page 70.

1 In the Datasource pane, ensure that the datasource you want to work on is connected. Then double-click the datasource, so that its elements appear in the Structure pane.

2 In the Content pane, make sure that the MetaData viewer tab is selected.

3 In the Structure pane, right-click the Mirrors node or any mirror node and choose Synchronize All Mirrors from the context menu.

Because this operation may take some time, a dialog is displayed that allows you to cancel before the synchronization begins.

If there are any pending changes, those are first saved. Also note that this operation is carried out immediately once confirmed, in contrast to editing a mirror state, where you must explicitly save.

Notes

- You cannot synchronize the primary mirror.
- In order to synchronize a newly created mirror you must save your work first.

Specifying a new primary mirror

1 In the Datasource pane, ensure that the datasource you want to work on is connected. Then double-click the datasource, so that its elements appear in the Structure pane.

2 In the Content pane, make sure that the MetaData viewer tab is selected.

3 In the Structure pane, right click the mirror that is to become the new primary mirror and choose Set Primary from the context menu.

A dialog box displays in which you are asked to specify the following:

- How long will you wait for pending transactions to commit?
- If that wait period is exceeded, do you want to terminate the pending transactions?
- Do you want to continue the process of making this mirror the new primary even if some mirrors fail to update?

If there are any pending changes, those are first saved. Also note that this operation is carried out immediately once confirmed, in contrast to editing a mirror state, where you must explicitly save.

Note You cannot designate a newly created mirror as a primary until you have saved it.

Adding a synchronization schedule

A synchronization schedule causes a mirror to be automatically synchronized with its specified source database at specified intervals. This is how you perform scheduled incremental backups of a database.

1. In the Datasource pane, ensure that the datasource you want to work on is connected. Then double-click the datasource, so that its elements appear in the Structure pane.
2. In the Content pane, make sure that the MetaData viewer tab is selected.
3. In the Structure pane, expand the Mirrors node; then expand the node for the mirror you want to schedule. This displays the Schedule Updates node for that mirror.
4. Click Schedule Updates to display the scheduling grid in the Content pane. If this is the first schedule, the grid is empty.
5. To add a scheduled time or interval, click the [+] button above the grid. This adds a row in which you can edit the fields to specify the synchronization times.
6. By default, a newly added schedule synchronizes at a fixed interval of 60000 milliseconds, or once every minute.
   - To choose a different interval, edit the Fixed Interval field to specify a different interval in milliseconds.
   - To schedule synchronization at a certain time every day, choose Daily from the Update Period drop-down list and then edit the Time of Day field.
   - To schedule a weekly synchronization, choose Weekly from the Update Period drop-down list. Then choose a day from the Day of Week drop-down list and edit the Time field. To schedule a daily backup on more than one day of the week, add a row for each day.
7. To apply your changes, choose Save or Save All.

Modifying a mirror schedule

1. In the Datasource pane, ensure that the datasource you want to work on is connected. Then double-click the datasource, so that its elements appear in the Structure pane.
2. In the Content pane, make sure that the MetaData viewer tab is selected.
3. In the Structure pane, expand the Mirrors node; then expand the node for the mirror you want to schedule. This displays the Schedule Updates node for that mirror.
4. Click Schedule Updates to display the scheduling grid in the Content pane.
5. Edit the schedule row that you want to change, following the instructions in “Adding a synchronization schedule” on page 70.
6. To apply your changes, choose Save or Save All.

Deleting a mirror schedule

1. In the Datasource pane, ensure that the datasource you want to work on is connected. Then double-click the datasource, so that its elements appear in the Structure pane.
2. In the Content pane, make sure that the MetaData viewer tab is selected.
3. In the Structure pane, expand the Mirrors node; then expand the node for the mirror you want to schedule. This displays the Schedule Updates node for that mirror.

4 Click Schedule Updates to display the scheduling grid in the Content pane. If this is the first schedule, the grid is empty.

5 In the Content pane, click in the row for the schedule that you want to delete and then click the (–) button above the table.

6 To undo all changes since you last saved, right click the datasource's tab at the top of the Content pane and choose Revert *datasource_name*.

7 To apply your changes, choose Save or Save All.

Showing the mirror status

1 In the Datasource pane, ensure that the datasource you want to work on is connected. Then double-click the datasource, so that its elements appear in the Structure pane.

2 In the Content pane, make sure that the MetaData viewer tab is selected.

3 In the Structure pane, select the Mirror Status node. The Content pane then displays a readonly grid that shows the status of all mirrors for the selected database. Use the refresh button to refresh the information in the grid.

Interactive SQL

Server Console provides an environment for issuing interactive SQL commands against the active datasource.

The commands available in the ISQL window are the same as the commands available in the command-line ISQL utility. The only exception is that you cannot connect to a different datasource in this context.

Running ISQL in Server Console

1 To begin using ISQL, ensure that you are running against the desired datasource by double-clicking the datasource in the Datasource pane to make it active. Ensure that the datasource is connected.

2 In the Content pane, click the SQL tab to display the ISQL work pane.

3 Type SQL statements in the ISQL work pane. When you are ready, execute them in one of the following ways:

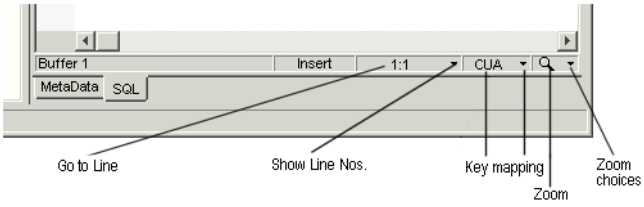
- Click the Execute SQL button in the toolbar.
- Choose Actions | Execute SQL.
- Use the *F9* shortcut key.

4 If the query is a successful `SELECT` query, the results are displayed in a grid in the Message pane. All other output is displayed as text in the Message pane and also in a separate messages window.

After a successful execution, a new empty editor buffer displays. The SQL executed earlier can be redisplayed by clicking the Previous Buffer and Next Buffer buttons in the toolbar, or by accessing the corresponding items on the Actions menu.

Note Connections in the ISQL window are not in autocommit mode. You must explicitly commit changes. Use the Commit and Rollback buttons on the toolbar or the corresponding commands on the Actions menu.

ISQL configuration and controls



The status bar at the bottom of the ISQL work area offers information about the editing environment and provides several controls.

- **Information** The left area of the status bar shows which buffer is currently displayed. The next area to the right shows whether the editing mode is Insert or Overwrite. You toggle this property using the Insert key on the keyboard.
- **Go to Line** To go directly to a line number, click the Go to Line area. This displays the Go to Line dialog box, in which you specify the desired line number.
- **Show line numbers** Click the down arrow to the right of the Go to Line area to choose whether line numbers are displayed. This down arrow also has an item for displaying the Go To Line dialog box.
- **Key mapping**
  - To toggle between the last two selected key mappings, click the Key Mapping area.
  - To choose the key mapping used by the editor, click the down arrow at the right of the Key Mapping area. You can choose from CUA, Emacs, Brief, Visual Studio, Macintosh, or Macintosh CodeWarrior.
- **Font size (zoom)**
  - Click the zoom symbol to increase the size of the font in the ISQL work area.
  - Click the down arrow to the right of the zoom symbol to choose from Zoom In, Zoom Out, and Normal.

Chapter 7

Using JDataStore's security features

JDataStore provides several built-in security features. These features provide user authentication, user authorization, and encryption for JDataStore databases.

In addition to the programmatic approaches described in this chapter, see Chapter 10, "SQL reference" for a description of how to use SQL statements to manage schemas, roles, and users, and using `GRANT` and `REVOKE` to assign privileges to roles and users.

User authentication

By default, JDataStore databases do not require users to be authenticated to access the database. JDataStore authentication support can be enabled by adding at least one user to the SYS/USERS system table in a JDataStore database. This can be done either programmatically, or by using JdsExplorer.

The SYS/USERS table is read-only if it is accessed by a JDBC query or a `StorageDataSet`.

When you add users, you supply their initial password and database rights.

There are three methods that you can use to administer users. `DataStoreRights.ADMINISTRATOR` rights are required to call these methods.

- To add a user, call the `DataStoreConnection.addUser` method.
- To remove a user, call the `DataStoreConnection.removeUser` method.
- To change a user's rights, call the `DataStoreConnection.changeRights` method.

JdsExplorer: See "Administering Users" on page 54 for an explanation of using JdsExplorer to administer users.

There is a `DataStoreConnection.changePassword` method that can be used to change a password. All users can change their own password. It requires knowledge of the existing password, but does not require `DataStoreRights.ADMINISTRATOR` rights.

JdsExplorer: For instructions on changing a password through JdsExplorer, see "Changing a password" on page 56.

Authorization

Database rights are supported by specifying the constants in the `com.borland.datastore.DataStoreRights` interface. The rights specified by `DataStoreRights` include:

- `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. The `DataStoreConnection.changeRights` method's `pass` parameter must not be null, and must match the user's password when calling this method to add `STARTUP` rights. `STARTUP` rights can also be specified at the time the user is added.
- `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`. `STARTUP` rights are given to an administrator by default when the administrator is added, but `STARTUP` rights can be removed from an administrator. `WRITE`, `CREATE`, `DROP`, and `RENAME` rights cannot be removed from an administrator. Any attempt to remove these rights from an administrator is ignored.
- `WRITE`: Confers the ability to write to file or table streams in the JDataStore database.
- `CREATE`: Confers the ability to create new file or table streams in the JDataStore database.
- `DROP`: Confers the ability to remove file or table streams from the JDataStore database.
- `RENAME`: Confers the ability to rename file or table streams in the JDataStore database.

JDataStore encryption

A `DataStoreRights.ADMINISTRATOR` user can encrypt an empty database that has a `DataStoreConnection.getVersion` of `DataStore.VERSION_6_0` or greater. The `DataStoreConnection.encrypt` method can be used to encrypt databases.

`DataStoreConnection.encrypt` removes `DataStoreRights.STARTUP` from all users except for the administrator that is adding the encryption. This is because the user's password is required to add `STARTUP` rights to a user. To provide `STARTUP` rights to a user, call `DataStoreConnection.changeRights`, or drop the user and then add the user back.

Note A database with existing table or file streams cannot be encrypted. If you want to encrypt an existing database, create a new database, call `DataStoreConnection.copyUsers` to copy the existing users to the new database, then go ahead and encrypt the new database. Then call `DataStoreConnection.copyStreams()` to copy the contents of the old database into the encrypted database. You can encrypt a database with existing tables using the JdsExplorer interface. See "Encrypting and unencrypting a JDataStore database" on page 41.

Deciding how to apply JDataStore security

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

The authentication and authorization support is secure for server-side applications where opponents do not have access to the physical JDataStore 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 cypher. A pseudorandom 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 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 JDataStore database is unencrypted, it is important to restrict physical access to the file, for the following reasons:

- If a JDataStore 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 JDataStore 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 JDataStore database, the database and log files should be encrypted, in addition to being password protected.

Warning The cryptographic techniques that JDataStore uses to encrypt data blocks are state-of-the-art. The TwoFish block cypher used by JDataStore has never been defeated. This means that if you forget your password for an encrypted JDataStore database, you are really out of luck. 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 virgin database that has one secret user who has `DataStoreRights.ADMINISTRATOR` rights (which includes `DataStoreRights.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 JDataStore cache to the JDataStore database and are decrypted when they are read from the JDataStore database into the JDataStore cache. So the cost of encryption is only incurred when file I/O is performed.

JDataStore 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 that the remote JDBC driver for JDataStore currently uses `Java.net.Socket` classes to communicate with a JDataStore Server. This communication is not secure. Since the local JDBC driver for JDataStore is in-process, it *is* secure.

Chapter

8

Stored procedures and UDFs

Stored procedures are user-defined functions that are designed to handle business logic. These functions serve to centralize business logic operations on tables. Stored procedures can have both input and output parameters and are called directly. Stored procedures can also return a JDBC ResultSet. For example:

```
CALL INCREASE_SALARY(10000);
```

UDFs are user-defined functions that are designed to be used in subexpressions of a SQL statement. Typically a `SELECT` statement can use a UDF in its `WHERE` clause. For example:

```
SELECT * FROM TABLE1
   WHERE MY_XOR_UDF(COL1,COL2) = 8;
```

Programming language for stored procedures and UDFs

Stored procedures and UDFs for JDataStore must be written in Java. The compiled Java classes for stored procedures and UDFs must be added to the `CLASSPATH` of the JDataStore 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 JDataStore tools by adding the classes to `<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 JDataStore server process, use the following SQL syntax to associate a method name with it:

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

where <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 following section provides an example of how to define and call a UDF.

A UDF example

The following 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 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. JDataStore 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. For 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 parameter

If the first parameter of a Java method is of type `java.sql.Connection`, JDataStore 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 JDataStore do it.

An 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);
```

Notes

- `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.

Examples
The following example returns a `ResultSet`.

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

```
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;
}
```

The following example returns a `DataSet`, which is automatically converted to a ResultSet.

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

```
package com.mycompany.util;

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

Both of these implementations would be registered and called 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.

An 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	JDataStore 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` see (*)	`INPUTSTREAM`
`java.sql.Date`	`DATE`
`java.sql.Time`	`TIME`
`java.sql.Timestamp`	`TIMESTAMP`
All other types:	`OBJECT`

Note Any type derived from `java.io.InputStream` is handled as an `INPUTSTREAM`.

Chapter 9

Java triggers for JDataStore tables

Java trigger support can be provided for both JDBC and DataExpress APIs.

DataExpress triggers

`StorageDataSet` has an `addEditListeners(EditListener)` method. This method can be used to wire a `com.borland.dx.dataset.EditListener` implementation to the `StorageDataSet`. The `EditListener` interface has many event methods that are called for a variety of insert/update/delete actions performed against the database. The `EditAdapter` class implements the `EditListener` interface with methods that do nothing. The `EditAdapter` class can be extended instead of implementing `EditListener` directly if you only need to implement a subset of the `EditListener` interface.

JDBC triggers

The JDataStore database engine internally uses `StorageDataSet` components for all insert, delete, and update (DML) operations made using a JDBC driver. The `StoreClassFactory.getStorageDataSet` method can be implemented as a callback for the JDataStore database engine to call when a `StorageDataSet` is needed for a JDBC DML operation. For trigger support, `StoreClassFactory` implementations typically instantiate a `StorageDataSet` and wire an `EditListener` implementation. A `StoreClassFactory` implementation must be registered for a table using the `StorageDataSet.setStoreClassFactory` method.

By setting the `StoreClassFactory`, the same trigger logic can be used for both DataExpress and JDBC access to the database. Note however, that the `StoreClassFactory.getStorageDataSet` method is called only when the table is accessed from JDBC API. An application that instantiates its own `StorageDataSet` must call the `StorageDataSet.addEditListner(EditListener)` method to register a trigger.

Here is a small code snippet that shows how to register a `StoreClassFactory` with a JDataStore table:

```
public static void registerTrigger(DataStoreConnection con, String tableName)
    throws IOException, SQLException  {
       StorageDataSet table = new StorageDataSet();
       table.setStoreName(tableName);
       table.setStore(con);
```

```
    // This persists the factory implementations class name inside the
    // table metadata for the JDataStore tableName table.

    table.setStoreClassFactory(new TriggerSnippetFactory);
    table.open();
    table.close();
}
```

Here is an implementation of `TriggerFactorySnippet`. The `getStorageDataSet` method is called only when a JDBC API causes a JDataStore table to be opened. Even handlers such as `EditListener` can be added when this method is called. This has no effect for tables that are opened directly by DataExpress.

```
public class TriggerFactorySnippet implements StoreClassFactory {
  public StorageDataSet getStorageDataSet(Store store, String storeName) {
    StorageDataSet table = new StorageDataSet();
    table.setStore(store);
    table.setStoreName(storeName);
    table.addEditListener(new TableTriggers());
    return table;
  }
}
```

`EditAdapter` is a class that provides default implementations for all of the methods in the `EditListener` interface. This class overrides some of these methods to display before and after notifications when data is added, deleted, or updated.

```
class TableTriggers extends EditAdapter {

  public void adding(DataSet dataSet, ReadWriteRow newRow)
  throws Exception {
     System.out.println("adding: ");
  }

  public void added(DataSet dataSet) {
     System.out.println("added: ");
  }

  public void updating(DataSet dataSet, ReadWriteRow newRow, ReadRow oldRow)
  throws Exception {
     System.out.println("updating: ");
  }

  public void updated(DataSet dataSet) {
     System.out.println("updated: ");
  }

  public void deleting(DataSet dataSet) throws Exception {
     System.out.println("deleting: ");
  }

  public void deleted(DataSet dataSet) {
     System.out.println("deleted: ");
  }
}
```

Chapter

10

SQL reference

The SQL Reference includes the following topics:

Using JDBC

The JDataStore JDBC driver provides the following methods for creating and executing SQL statements:

From `java.sql.Connection`, for creating statements:

```
Statement createStatement();
PreparedStatement prepare(String query);
CallableStatement prepare(String query);
```

From `java.sql.Statement`:

```
int executeUpdate(String query);
ResultSet executeQuery(String query);
```

From `java.sql.PreparedStatement` and `java.sql.CallableStatement`:

```
int executeUpdate();
ResultSet executeQuery();
```

Each query string must contain exactly one SQL statement.

The following table describes when to use each of these methods:

Table 10.1 Accessing SQL with java methods

Method	When to use
`executeQuery`	Use for statements that return a `ResultSet`. For example: `SELECT * FROM EMPLOYEE.`
`executeUpdate`	Use for statements that do not return a `ResultSet` `executeUpdate` throws an exception if the statement executed actually produces a `ResultSet`.
`CallableStatement`	Use if a statement contains output parameters.
`PreparedStatement`	Use if a statement contains input parameters.

Example

The following statements correct the spelling of the name “Overbeck”. You would use `executeUpdate` with these statements, because they do not return a `ResultSet`.

```
CREATE TABLE MYTABLE (COLUMN1 INT, LAST_NAME VARCHAR(20));

INSERT INTO MYTABLE VALUES (1, 'Overbek');

UPDATE MYTABLE SET LAST_NAME='Overbeck' WHERE COLUMN1=1;
```

Data types

In SQL, you can specify data types by using JDataStore names or by using synonyms, which are more portable to other SQL dialects. The following table lists the JDataStore SQL data types and their Java equivalents. See “Data types” on page 43 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.

Table 10.2 Java and SQL data types supported by JDataStore

Java data type	SQL equivalents [1]
byte	**`TINYINT`** `BYTE`
`short`	**`SMALLINT`** `SHORT`
`int`	**`INT`** `INTEGER`
`long`	**`BIGINT`** `LONG`
`java.math.BigDecimal`	**`DECIMAL(p,d)`** `BIGDECIMAL(`*p*`,`*d*`)`

`double`	`FLOAT(`*p*`)`, *p*=24 through 52
	`FLOAT`
	`DOUBLE`
	`DOUBLE PRECISION`

Table 10.2 Java and SQL data types supported by JDataStore (continued)

Java data type	SQL equivalents [1]
float	**REAL** FLOAT(*p*), *p*=1 through 23
java.lang.String	**VARCHAR(p,m)** STRING(*p*,*m*)
byte[]	**VARBINARY(p,m)** BINARY(*p*,*m*) INPUTSTREAM(*p*,*m*)
java.io.Serializable	OBJECT(*t*,*m*)
boolean	**BOOLEAN** BIT
java.sql.Date	**DATE**
java.sql.Time	**TIME**
java.sql.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:

JDataStore 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

JDataStore data type	Examples	Description
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 JDataStore SQL.

Keywords

The two lists below show all the current keywords for JDataStore. 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 JDataStore SQL engine. For maximum portability, don't use identifiers that are treated as keywords in any SQL dialect.

Reserved JDataStore 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	ACTION	ADD	ADMIN
ADMINISTRATOR	ALL	ALTER	AND
ANY	AS	ASC	AUTHORIZATION
AUTOINCREMENT	AVG	BETWEEN	BIT
BIT_LENGTH	BOTH	BY	CALL
CASCADE	CASE	CAST	CHAR
CHAR_LENGTH	CHARACTER	CHARACTER_LENGTH	CHECK
COALESCE	COLUMN	CONSTRAINT	COUNT
CREATE	CROSS	CURRENT_DATE	CURRENT_ROLE
CURRENT_TIME	CURRENT_TIMESTAMP	CURRENT_USER	DATE

DECIMAL	DEFAULT	DELETE	DESC
DISTINCT	DOUBLE	DROP	ELSE
END	ESCAPE	EXCEPT	EXECUTE
EXISTS	EXTRACT	FALSE	FLOAT
FOR	FOREIGN	FROM	FULL
GRANT	GROUP	HAVING	IN
INDEX	INNER	INSERT	INT
INTEGER	INTERSECT	INTO	IS
ISOLATION	JOIN	KEY	LEADING
LEFT	LEVEL	LIKE	LOWER
MAX	MIN	NATURAL	NO
NONE	NOT	NULL	NULLIF
NUMERIC	OCTET_LENGTH	ON	ONLY
OPTION	OR	ORDER	OUTER
POSITION	PRECISION	PRIMARY	PRIVILEGES
PUBLIC	REAL	REFERENCES	RENAME
RESOLVABLE	RESTRICT	REVOKE	RIGHT
SCHEMA	SELECT	SET	SMALLINT
SOME	SQRT	STARTUP	SUBSTRING
SUM	TABLE	THEN	TIME
TIMESTAMP	TO	TRAILING	TRANSACTION
TRIM	TRUE	UNION	UNIQUE
UNKNOWN	UPDATE	UPPER	USER
USING	VALUES	VARCHAR	VARYING
VIEW	WHEN	WHERE	WITH

JDataStore keywords that are not reserved

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	AUTOCOMMIT	BOOLEAN	BIGDECIMAL
BIGINT	BINARY	BYTE	CASEINSENSITIVE
CLASS	COMMIT	COMMITTED	CONCAT
CONVERT	CURDATE	CURTIME	D
DAY	DAYOFMONTH	DEC	FN
GRANTED	HOUR	IFNULL	INPUTSTREAM
JAVA_METHOD	LCASE	LENGTH	LOCATE
LOCK	LONG	LONGINT	LONGVARBINARY
LONGVARCHAR	LTRIM	METHOD	MINUTE
MONTH	NOW	NOWAIT	OBJECT
OFF	OJ	PASSWORD	READ

REPEATABLE	ROLE	ROLLBACK	RTRIM
SECOND	SERIALIZABLE	SHORT	STRING
T	TIMESTAMPADD	TIMESTAMPDIFF	TIMEZONEHOUR
TIMEZONEMINUTE	TINYINT	TS	TYPE
UCASE	UNCOMMITTED	VARBINARY	WORK
WRITE	YEAR		

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

Table 10.3 Valid identifiers

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

Table 10.4 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"
```

About 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 JDataStore, see “Functions” on page 95.

```
<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 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;
```

DB_ADMIN

DB_ADMIN is a SQL implementation of JDataStore's DB_ADMIN java class. You use these methods to perform a variety of database administration tasks such as configuring automatic failover and incremental backup, changing database properties, managing datasources, verifying tables, and displaying database privileges and properties, locks, status log IDs, procedure privileges, and roles granted.

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

Some methods return DataSet objects as java.sql.ResultSet objects for the JDBC driver. When a DataSet is returned from a stored procedure, it is received as a java.sql.ResultSet by the JDBC driver.

Methods

For more information on DB_ADMIN methods, see com.borland.datastore.driver.DB_ADMIN.html. The following list provides the syntax and a brief description of each method. Click the method name to display greater detail from DB_ADMIN.html in the JDataStore API Reference.

DB_ADMIN provides the following methods:

Method: ALTER_DATABASE

ALTER_DATABASE(*connection*, *mirror*, *properties*)

Alters database properties. connection is implicit; do not specify it from a SQL CALL statement.

Method: ALTER_MIRROR

ALTER_MIRROR(*connection*, *mirrorName*, *properties*)

Alters an existing mirror configuration. connection is implicit; do not specify it from a SQL CALL statement.

Method: ALTER_MIRROR_SCHEDULE

ALTER_MIRROR_SCHEDULE(*connection*, *mirrorName*, *properties*)

Alters an existing mirror schedule item. connection is implicit; do not specify it from a SQL CALL statement.

Method: CLOSE_CONNECTION

CLOSE_CONNECTION(*connection*, *connection_ID*, *milliseconds*)

Closes an open connection. connection is implicit; do not specify it from a SQL CALL statement.

Method: CREATE_MIRROR

CREATE_MIRROR(*connection*, *properties*)

Creates a new mirror configuration. connection is implicit; do not specify it from a SQL CALL statement.

Method: CREATE_MIRROR_SCHEDULE

CREATE_MIRROR_SCHEDULE(*connection*, *mirrorName*, *properties*)

Creates a new mirror synchronization schedule. connection is implicit; do not specify it from a SQL CALL statement.

Method: DROP_MIRROR

DROP_MIRROR(*connection*, *mirrorName*)

Drops an existing mirror configuration. connection is implicit; do not specify it from a SQL CALL statement.

Method: DROP_MIRROR_SCHEDULE

DROP_MIRROR_SCHEDULE(*connection*, *mirrorName*, *properties*)

Drops an existing mirror schedule item. connection is implicit; do not specify it from a SQL CALL statement.

Method: GET_CONNECTIONS

GET_CONNECTIONS(*connection*, *boolean*)

Provides a ResultSet of the open connections for the current server connection. connection is implicit; do not specify it from a SQL CALL statement.

Method: GET_DATABASE_PRIVILEGES

GET_DATABASE_PRIVILEGES(*connection*, *boolean*)

Retrieves a description of the database access rights for each use or role. connection is implicit; do not specify it from a SQL CALL statement.

Method: GET_DATABASE_PROPS

GET_DATABASE_PROPS(*connection*)

Provides the properties for the current database. connection is implicit; do not specify it from a SQL CALL statement.

Method: GET_DATABASE_STATUS_LOG_FILTER

GET_DATABASE_STATUS_LOG_FILTER(*connection*)

Retrieves filter that controls what kind of logging information is logged to the status log file for all current database connections. The bit masks in LogFilterCodes can be ORed together to enable logging categories. connection is implicit; do not specify it from a SQL CALL statement.

Method: GET_LOCKS

GET_LOCKS(*connection*)

Provides a ResultSet of all the currently held table and row locks for all connections to the current database.

Method: GET_MIRRORS

GET_MIRRORS(*connection*, *mirror*, *status*)

Retrieves a table of mirrors that have been configured for this database. The structure of this table is defined in SysMirrors. connection is implicit; do not specify it from a SQL CALL statement.

Method: GET_NEWEST_STATUS_LOG_ID

GET_NEWEST_STATUS_LOG_ID(*connection*)

Provides the ID of the newest log file that can be retrieved using the GET_STATUS_LOG() method. connection is implicit; do not specify it from a SQL CALL statement.

Method: GET_OLDEST_STATUS_LOG_ID

GET_OLDEST_STATUS_LOG_ID(*connection*)

Provides the ID of the oldest log file that can be retrieved using the GET_STATUS_LOG() method. connection is implicit; do not specify it from a SQL CALL statement.

Method: GET_PROCEDURE_PRIVILEGES
GET_PROCEDURE_PRIVILEGES(*connection*)

Retrieves a description of the access rights for each procedure. connection is implicit; do not specify it from a SQL CALL statement.

Method: GET_ROLE_GRANTS
GET_ROLE_GRANTS(*connection*, *boolean*)

Retrieves a description of all role grants. connection is implicit; do not specify it from a SQL CALL statement.

Method: GET_STATUS_LOG_FILTER
GET_STATUS_LOG_FILTER(*connection*)

Retrieves the filter that controls what kind of logging information is logged to the status log for this connection. connection is implicit; do not specify it from a SQL CALL statement.

Method: GET_STATUS_LOG
GET_STATUS_LOG(*connection*, *log_id*, *offset*)

Provides the status log for the current database. connection is implicit; do not specify it from a SQL CALL statement.

Method: GET_THIS_MIRROR
GET_THIS_MIRROR(*connection*, *boolean*)

Like GET_MIRRORS except that it returns information for only the specified mirror. connection is implicit; do not specify it from a SQL CALL statement.

Method: SET_DATABASE_STATUS_LOG_FILTER
SET_DATABASE_STATUS_LOG_FILTER(*connection*, *filter*)

Sets the filter that controls what kind of logging information is entered in the status log file for all current database connections. connection is implicit; do not specify it from a SQL CALL statement.

Method: SET_PRIMARY_MIRROR
SET_PRIMARY_MIRROR(*connection*, *mirror*, *milliseconds*, *boolean*, *boolean*)

Sets mirrorName as the primary mirror. connection is implicit; do not specify it from a SQL CALL statement.

Method: SET_STATUS_LOG_FILTER
SET_STATUS_LOG_FILTER(*connection*)

Sets the filter that controls what kind of logging information is logged to the status log file for the current connection. connection is implicit; do not specify it from a SQL CALL statement.

Method: SYNCH_MIRROR
SYNCH_MIRROR(*connection*, *mirror*)

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

Method: VERIFY
VERIFY(*connection*, *pattern*, *options*, *errors*, *output*)

Verifies one or more tables in the database. connection is implicit; do not specify it from a SQL CALL statement.

DB_UTIL: numeric, string, and date/time functions

DB_UTIL is a collection of SQL utility functions that perform numeric, string and date/time operations on data stored in database tables. These functions are implemented as Java UDFs in com.borland.datastore.driver.DB_UTIL.

Examples

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

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

2 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;
```

Methods

For more information on DB_UTIL methods, see com.borland.datastore.driver.DB_UTIL.html. The following list provides the syntax and a brief description of each method. Click the method name to display greater detail from DB_UTIL.html in the JDataStore API Reference.

Numeric functions

Method: ACOS

ACOS(*expression*)

Returns the arccosine in radians of a number.

Method: ASIN

ASIN(*expression*)

Returns the arcsine in radians of a number.

Method: ATAN

ATAN(*expression*)

Returns the arctangent in radians of a number.

Method: 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.

Method: 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.

Method: COS

COS(*expression*)

Returns the cosine of an angle.

Method: `COT`
`COT(`*`expression`*`)`

Returns the cotangent of an angle.

Method: `DEGREES`
`DEGREES(`*`expression`*`)`

Converts an angle in radians to degrees.

Method: `EXP`
`EXP(`*`expression`*`)`

Returns the exponential value of *expression*.

Method: `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.

Method: `LOG`
`LOG(`*`expression`*`)`

Returns the natural logarithm of a number.

Method: `LOG10`
`LOG10(`*`expression`*`)`

Returns the base 10 logarithm of a number.

Method: `MOD`
`MOD(`*`expression1`*`, `*`expression2`*`)`

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

Method: `PI`
`PI()`

Returns the constant PI.

Method: `POWER`
`POWER(`*`expression1`*`, `*`expression2`*`)`

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

Method: `RADIANS`
`RADIANS(`*`expression`*`)`

Converts an angle in degrees to radians.

Method: `RAND`
`RAND()`

Generates a random floating point number.

Method: `RAND`
`RAND(`*`expression`*`)`

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

Method: ROUND
ROUND(*expression1*, *expression2*)

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

Method: SIGN
SIGN(*expression*)

Returns –1 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.

Method: SIN
SIN(*expression*)

Returns the sine in radians of an angle.

Method: SQRT
SQRT(*expression*)

Returns the square root of a number.

Method: TAN
TAN(*expression*)

Returns the tangent of an angle given in radians.

Method: TRUNCATE
TRUNCATE(*expression1*, *expression2*)

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

String functions

Method: ASCII
ASCII(*string*)

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

Method: TO_CHAR
TO_CHAR(*ascii_code*)

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

Method: 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.

Method: 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*.

Method: `LEFT_STRING`
`LEFT_STRING(`*`string`*`, `*`count`*`)`

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

Method: `REPEAT`
`REPEAT(`*`string`*`, `*`count`*`)`

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

Method: `REPLACE`
`REPLACE (`*`string1`*`, `*`string2`*`, `*`string3`*`)`

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

Method: `RIGHT`
`RIGHT_STRING(`*`string`*`, `*`count`*`)`

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

Method: `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.

Method: `SPACE`
`SPACE(`*`count`*`)`

Returns a character string consisting of *count* spaces.

Date and time functions

Method: `DAYNAME`
`DAYNAME(`*`date`*`)`

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

Method: `DAYOFWEEK`
`DAYOFWEEK(`*`date`*`)`

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

Method: `DAYOFYEAR`
`DAYOFYEAR(`*`date`*`)`

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

Method: `MONTHNAME`
`MONTHNAME(`*`date`*`)`

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

Method: `QUARTER`
`QUARTER(`*`date`*`)`

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

Method: 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 data types: java.sql.Timestamp, java.sql.Time, or java.sql.Date.

Method: 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 data types: java.sql.Timestamp, java.sql.Time, or java.sql.Date.

Method: 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.

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" on page 97.

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" on page 108.

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 JDataStore, 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
	`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.

`ON`, `USING`, and `NATURAL` 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 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 JDataStore 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`.

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 java method statement>
   | <drop java method statement>
   | <create java class statement>
   | <drop java 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 java 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 JAVA_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 JAVA_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, JDataStore 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 java method belongs to your default schema.
  - If you have not created a personal default schema, the table, view, or java 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 JDataStore that did not support schemas belong to the DEFAULT_SCHEMA schema when migrated to JDS 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 java method statement>
  | <grant statement>
```

See "GRANT" on page 132 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 java methods.

Note The DROP SCHEMA command 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 Java methods. It also drops any dependent views and foreign keys.

```
DROP SCHEMA BORIS CASCADE;
```

CREATE TABLE

The CREATE TABLE statement creates a JDataStore 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. JDataStore 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" on page 113 for more information about schemas.

Tracking data changes for DataExpress

If you specify `RESOLVABLE` as part of the table definition, JDataStore 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 JDataStore 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>
```

```
<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 JDataStore 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
  - Java 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 JDataStore 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 JDataStore 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 JAVA_METHOD

The CREATE JAVA_METHOD statement makes a stored procedure or a UDF written in Java available for use in JDataStore SQL. The class files for the code must be added to the classpath of the JDataStore server process before use. See Chapter 8, "Stored procedures and UDFs" for details about how to implement stored procedures and UDFs for JDataStore.

To create a method in a particular schema, specify the schema name as part of the table name:

```
CREATE JAVA_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 java method is created in that schema.
- If you have not created a personal default schema, the java method is created in the DEFAULT_SCHEMA schema.

  See "CREATE SCHEMA" on page 113 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 java method statement> ::=
    CREATE JAVA_METHOD <method name> [AUTHORIZATION <username>]
         AS <method definition>

<method name> ::=
    [ <schema name> . ] <SQL identifier>

<schema name> ::= <SQL identifier>

<method definition> ::= <string literal>
```

Example

```
CREATE JAVA_METHOD ABS AS 'java.lang.Math.abs';
```

DROP JAVA_METHOD

The DROP JAVA_METHOD statement drops a stored procedure or a UDF, making it unavailable for use in JDataStore SQL.

Syntax

```
<drop java method statement> ::=
    DROP JAVA_METHOD <method_name>
```

Example

```
DROP JAVA_METHOD ABS;
```

CREATE JAVA_CLASS

The CREATE JAVA_CLASS statement makes all public static methods of a Java class available to JDataStore SQL as stored procedures or UDFs. You must ensure that the class files for the code are on the classpath of the JDataStore server process before use. See Chapter 8, "Stored procedures and UDFs" for details about how to implement JDataStore stored procedures and UDFs.

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 java class statement> ::=
    CREATE JAVA_CLASS <class name> [AUTHORIZATION <username>]
        AS <class definition>

<class name> ::=
    [ <schema name> . ] <SQL identifier>

<schema name> ::= <SQL identifier>

<class definition> ::= <string literal>
```

Examples

```
CREATE JAVA_CLASS MATH AS 'java.lang.Math';
```

After the above statement executes, all public static methods in java.lang.Math can be called from SQL. Note that the method names are case sensitive.

Usage

The following statement calls the abs() method in java.lang.Math:

```
SELECT * FROM CUSTOMER WHERE MATH."abs"(AGE - 50) < 5;
```

DROP JAVA_CLASS

The DROP JAVA_CLASS statement drops a stored class, making it unavailable for use in JDataStore SQL.

Syntax

```
<drop java_class statement> ::=
    DROP JAVA_CLASS <method_name>
```

Example

```
DROP JAVA_CLASS MATH;
```

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” on page 20 for a detailed discussion of JDataStore 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.

In the following description of isolation levels, it is necessary to 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

JDatastore 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 JDataStore 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> ::=
    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 should be used in connection with a `PreparedStatement` in JDBC. 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;
```

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 JDataStore 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 Java method implementing `IncreaseSalaries` updates the `salaries` table with an increase of some percentage for all employees. A `java.sql.Connection` will implicitly be passed to the Java 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—such as INSERT or SELECT—on tables or java methods to PUBLIC, users, or roles.
- It grants database privileges—such as STARTUP or RENAME—to users or roles.
- It grants roles to users or roles.

GRANT options:

- When object privileges are granted WITH 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 ADMIN OPTION, the grantee has the power to pass on the granted database privileges or roles to other users.
- The ADMINISTRATIVE database privilege grants STARTUP, WRITE, CREATE, DROP, RENAME, CREATE ROLE, and CREATE SCHEMA privileges. When these privileges are acquired through the ADMINISTRATIVE privilege, they can be revoked only by revoking the ADMINISTRATIVE privilege. In other words, if you grant ADMINISTRATIVE 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 java method, using the required JAVA_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>
    | JAVA_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 java 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 java method, using the required JAVA_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>
    | JAVA_METHOD <method name>

<grantor> ::=
      CURRENT_USER
    | CURRENT_ROLE
```

Example 1

In all of the following examples, the name before the colon is the name of 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;
```

JDBC escape syntax

JDataStore supports JDBC escape sequences for:

- Date and time literals
- `OUTER JOIN`s
- The escape character for a `LIKE` clause
- Calling stored procedures

JDBC 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 Java method implementing IncreaseSalaries updates the salaries table with an increase of some percentage for all employees. A `java.sql.Connection` is implicitly passed to the Java 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)};
```

JDBC 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`

Function name	Function returns
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 string count times

Function name	Function returns
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

Function name	Function returns
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

Time and date functions

```
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

JDataStore's interactive ISQL utility is available from the SQL tab of the Server Console. The following ISQL commands can be issued from within the Server Console SQL tab or from a command console.

Getting help

To see a help display for JDataStore ISQL, issue one of the following help commands:

- From the system prompt:
  - `isql -?` displays ISQL startup options.
  - `isql -help` displays Java launcher options.
- From the SQL prompt:
  - `HELP CREATE` displays help on creating datasources.
  - `HELP SHOW` displays a list of `SHOW` commands with brief descriptions.
  - `HELP SET` displays a list of `SET` commands with brief descriptions of each.

Starting isql

To start JDataStore ISQL, either ensure that *jdatastore_home*`\bin` is on your system path, or `cd` to that directory to issue the `isql` command. The following options are available:

Table 10.5 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:

```
SHOW CREATE;
```

There are two additional groups of commands that are discussed later in this section: "SHOW commands" on page 145 and "SET commands" on page 146.

The SQL commands that are available for data definition, data manipulation, security, and transaction management are discussed throughout this chapter.

Table 10.6 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" on page 144, for information on creating datasources in ISQL.
`CONNECT dataSourceName [user password]`	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 [user password]`	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 [user password]`	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:

```
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 Java 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*'

  You can supply values for any properties in the datasource class. For example, to create a new database, add CREATE=true:

```
CREATE DATASOURCE JDS user=SYSDBA, password=masterkey,
  databaseName='c:/databases/test.jds',CREATE=true';
```

The following two examples both use the JDataStore 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';
```

SHOW commands

Table 10.7 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 procedures
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.

Table 10.7 ISQL SHOW commands (continued)

Command	Description
`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.

SET commands

Table 10.8 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

11

Optimizing JDataStore applications

This section discusses ways to improve the performance, reliability, and size of JDataStore applications. Unless otherwise specified, “DataStoreConnection” refers to either a `DataStoreConnection` or `DataStore` object used to open a connection to a JDataStore database file.

Loading databases quickly

Here are some tips that can improve the performance of your application when loading databases:

- Use prepared statements whenever possible. If the number of parameters changes from one insert to the next, call `PreparedStatement.clearParameters()` before setting `PreparedStatement` parameters.
- Use the DataExpress `TextDataFile` class to import text files. It has a fast parser and knows how to 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 JDataStore database.
- When loading a new database, first create the database as non-transactional. Load the database while it is non-transactional using a DataExpress `StorageDataSet.addRow` or `TextDataFile` component. After loading is complete, make the database transactional using the `DataStore.TxManager` property. This technique should make the load operation perform two to three times faster.
- 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.

General usage recommendations

Here are a few performance tips for all types of JDataStore 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 because JDataStore needs about 8-10 seconds to ensure that no other process has the database open.

To ensure that a database is shut down properly, make sure all JDBC and DataExpress connections are closed when they are no longer needed. The `DataStore.shutdown()` method can be used to make sure all database connections are closed before an application terminates.

Closing a JDataStore 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 VM 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 JDataStore could get corrupted.

A transactional JDataStore database is guaranteed to not lose data, but the transaction manager rolls back any uncommitted changes.

Another benefit to closing connections is that when they are all closed, the memory allocated to the JDataStore cache is released.

If your application is using DataExpress components, close all `StorageDataSet`s that have their `store` property set to a `DataStoreConnection` when you are done with them. This frees up JDataStore resources associated with the `StorageDataSet` and allows the `StorageDataSet` to be garbage collected.

Optimizing the JDataStore disk cache

The default maximum cache size for a JDataStore database is 512 cache blocks. The default block size is 4096 bytes. Therefore, the cache memory reaches its maximum capacity out at approximately about 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. The minimum cache size can be specified with the `DataStore.MinCacheSize` property.

Note Do not arbitrarily change the database cache size. Be sure to verify beforehand that doing so will improve the performance of your application.

Keep in mind the following considerations when changing the JDataStore cache size:

- Modern OS caches are typically high performance. In many cases, increasing the JDataStore cache size does not significantly improve performance and simply uses more memory.
- Depending on the Java VM you are using, allocating large amounts of memory could slow the performance of JVM garbage collection operations.
- There is only one JDataStore disk cache for all JDataStore databases open in the same process. When all JDataStore 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

JDataStore databases perform the majority of read/write operations against the following four file types:

- The JDataStore database file itself (file extension is `.jds`) as specified by the `DataStore.FileName` property.
- JDataStore transactional log files. The names of log files end with an extension of `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.

Performance can potentially be improved by telling JDataStore to place the files mentioned above on different disk drives.

Here are some file storage suggestions that can help improve your application's performance:

- It is especially important to place the log files on a separate disk drive. Note that the log files are generally written to in sequential order and their contents must be forced to disk in order to complete commit operations. As such, it is advantageous to have a disk drive that can complete write operations quickly.
- On Win32 platforms, it has been observed that performance can be improved by placing JDataStore log files in a separate directory, and that 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 JDataStore 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 that your log files are written to.
- Set the extended JDBC `fileio` property to "native" for Win32 platforms or "new" for non-Win32 platforms to improve log file performance. See the reference documentation for this property in `com.borland.datastore.driver.cons.ExtendedProperties`.

Non-transactional database disk cache write options

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 JDataStore 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 JDataStore 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.

- 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.
- Try increasing the DataStore.minCacheBlocks property, which controls the minimum number of blocks that are cached. For JDBC connections, use the minCacheBlocks extended property.
- The DataStore.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. For JDBC connections, use the maxSortBuffer extended property.

Miscellaneous performance tips

Here are some tips that can help performance:

- Setting the DataStore.tempDirName property, used by the query engine, to a directory on another (fast) disk drive can often help. For JDBC connections, use the tempDirName extended property.
- Try setting the TxManager.checkFrequency higher. A higher value can improve performance but might result in slower crash recovery. You can also set this property in JdsExplorer by choosing TxManager | Modify.
- For simple operations, DataExpress can be somewhat faster than JDBC/SQL. JDBC/SQL is faster for more complex queries.

Optimizing transactional applications

The increased reliability and flexibility you gain from using transactional JDataStore databases comes at the price of some performance. You can reduce this cost in several ways.

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 DataStoreConnection's readOnlyTx property controls whether a transaction is read-only. For JDBC connections, it is controlled by the readOnly property of the java.sql.Connection object.

Read-only transactions work by simulating a snapshot of the JDataStore 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 its `commit` method is called.

Another benefit of read-only transactions is that they aren't blocked by writers or other readers. Both reading and writing usually require a lock. But because a read-only transaction uses a snapshot, it doesn't need any locks.

You can further optimize the application by specifying a `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 `TxManager 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.

Transaction log files

Disabling status logging

You can improve performance by disabling the logging of status messages. To do this, set the `recordStatus` property of the `TxManager` to `false`.

Tuning JDataStore concurrency control performance

Use the following guidelines to optimize the performance of JDataStore concurrency control operations:

- Choose the weakest isolation level that your application can function properly with. Lower isolations tend to acquire fewer and weaker locks.
- Set the `setTableLockTables()` property for tables that are infrequently updated. There is less overhead for table locks. For details, see `com.borland.datastore.driver.cons.ExtendedProperties` (JDBC) or `com.borland.datastore.DatastoreConnection.html#setTableLockTables(java.lang.String)` (Data Express).
- Set `java.sql.Connection.autoCommit` to `false` to group multiple operations into a single transaction. The `java.sql.Connection commit` and `rollback` properties can be used for terminating transactions.
- Commit transactions as soon as possible. Most locks are not released until a transaction is committed or rolled back.
- Reuse `java.sql.Statement` objects whenever possible, or better yet, use `java.sql.PreparedStatement` when possible.
- Close all `Statement`, `PreparedStatement`, `ResultSet`, and `Connection` objects when they are no longer needed. Note that single directional `ResultSet` objects automatically close when the last row is read.
- Use read-only transactions for long running reports or on line backup operations. Use the `com.borland.datastore.DataStoreConnection.copyStreams()` method for online backups. Read-only transactions provide a transactionally consistent

(serializable), read-only view of the tables they access. They acquire no locks, so lock timeouts and deadlocks are not possible. This property can be set using the `java.sql.Connection.readOnly` method for JDBC connections or the `com.borland.datastore.DataStoreConnection.setReadOnlyTx` method for DataExpress components.

- There is some overhead for maintaining a read-only view. Consequently, multiple transactions can share the same read-only view. The `readOnlyTxDelay` property specifies how old the read-only view can be when a read-only transaction is started. Committing a read-only connection's transaction refreshes its view of the database. Note that a read-only transaction uses the transactional log files to maintain its 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 via the "group commit" support provided by JDataStore.

Pruning deployed resources

When deploying a JDataStore application, you can exclude certain classes and graphics files that aren't used. In particular:

- If JDataStore is used without the JDBC driver, exclude these 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 these classes:

```
com.borland.dx.memorystore.*
```

- If `StorageDataSet` is used, but not `QueryDataSet`, `QueryProvider`, `StoredProcedureDataSet` or `StoredProcedureProvider`, exclude these classes:

```
com.borland.dx.sql.*
```

- If DataExpress isn't using any visual components from the JBCL or dbSwing libraries, exclude these classes:

```
com.borland.dx.text.*
```

- If `com.borland.dx.dataset.TextDataFile` isn't used, exclude these classes:

```
com.borland.jb.io.*
com.borland.dx.dataset.TextDataFile.class
com.borland.dx.dataset.SchemaFile.class
```

AutoIncrement columns

Columns of type `int` and `long` can now be specified as having `AutoIncrement` values.

These properties apply to all `AutoIncrement` column values:

- They are always unique
- They can never be null
- Values from deleted rows can never be reused

These properties make `AutoIncrement` columns ideal for single column `integer/long` primary keys.

An AutoIncrement column provides the fastest random access path to a particular row in a JDataStore table because it is the internal row identifier for a row.

Note 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 JDataStore query optimizer optimizes queries that reference an AutoIncrement column in a WHERE clause.

AutoIncrement columns using DataExpress

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. For more information, see "Creating JDataStore tables with DataExpress" on page 168.

AutoIncrement columns using SQL

To create or modify a table to have an AutoIncrement column using SQL, see "Using autoIncrement columns with SQL" on page 116.

JDataStore 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 JDataStore 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 JDataStore databases that are attached to components it cleans up.

  For example, if you drop a DBDisposeMonitor into a JFrame that contains dbSwing components attached to a JDataStore database, when you close the JFrame, DBDisposeMonitor automatically closes the JDataStore database for you. This component is particularly handy when building simple applications to experiment with JDataStore.

- DBExceptionHandler has an Exit button. You can hide it with a property setting, but it's visible by default. Clicking this button automatically closes any open JDataStore database files it can find. DBExceptionHandler is the default dialog box displayed by dbSwing components when an exception occurs.

Using data modules for DataExpress components

When using a JDataStore table with a StorageDataSet, you should consider grouping them all inside data modules. Make any references to these StorageDataSets through DataModule accessor methods such as businessModule.getCustomer. You should do this because much of the functionality surfaced through StorageDataSets is driven by property and event settings.

Although most of the important structural StorageDataSet properties are persisted in the JDataStore table itself, the classes that implement the event listener interfaces aren't. 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 12

Deploying JDataStore database applications

When application development is complete, the next step is to deploy those applications as needed. This step involves licensing considerations and determining which `jar` files are needed for distribution.

Overview

The following lists presents an outline of the steps for deploying a JDataStore database application. Later sections describe in more detail how to create licenses and how to determine what JDataStore `jar` files are required.

1. Determine what deployment licenses are required for the application.
   - If the application will be accessed by only one user at a time, you need only a local server license.
   - For multiple users, you need a JDataStore server license with a specified maximum number of connections. Consider how many users will need to be connected at one time, and how many simultaneous connections each user might require in order to calculate the number of concurrent connections you need.
2. Obtain the necessary Serial Number/Authorization Key pairs. You can do this by talking to your sales representative or by going to `http://shop.borland.com`.
3. Load JdsExplorer and create a JDataStore license file for deployment, using the Serial Number/Authorization Key pairs that you purchased. See "Creating a JDataStore license" on page 156 for details.
4. Determine which `jar` files you need to include with your distribution by consulting "Distributing JDataStore files" on page 157.
5. Bundle the application classes you have written up into a `jar` file.
6. Distribute your `jar` file with the necessary JDataStore `jar` files and with the license file you generated. The `jdatastore.license` file must be placed in a file system directory or `jar` file that is included on the deployed application's classpath.

Important If you are distributing `jds.jar` or `jdsserver.jar` files that contain trial licenses, you must make sure that your generated `jdatastore.license file` appears in the classpath *before* the jars containing the trial licenses. If the JDataStore server finds the trial license first, it uses that license.

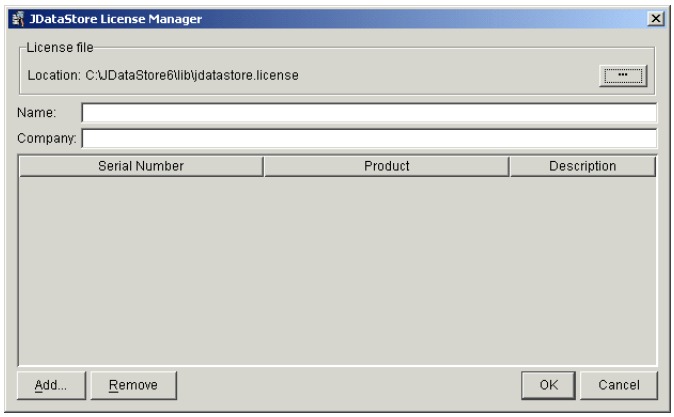
Creating a JDataStore license

You use JdsExplorer to create a new JDataStore licenses. These licenses are all stored together in the `jdatastore.license` file. Collectively, they determine whether the application can access a JDataStore database and how many concurrent connections are allowed.

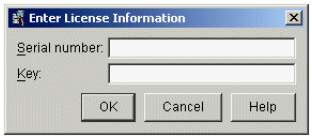
These licenses are cumulative, meaning that if you have already licensed *x* connections and you wish to increase that number by an amount *y*, you purchase serial numbers and keys for *y* connections and follow the steps below to create new licenses. You then can have *x* + *y* concurrent connections. The new licenses are appended to the existing `jdatastore.license` file.

Generating the license

1. Acquire Serial Number/Authorization Key pairs for the number of concurrent connections that you need. To do this, talk to your sales representative or go to `http://shop.borland.com`.
2. Launch JdsExplorer and choose File | License Manager to display the JDataStore License Manager.



3. Click the Add button to display the Enter License Information dialog box.



   Enter a Serial Number and its associated Authorization Key in the designated fields and click OK. When the License Terms dialog box displays, read the License Agreement and click “I understand and agree...” to enable the check box and then click OK.
4. If you have more license keys to add, repeat the process. The first time you add a license, License Manager creates a file named `jdatastore.license` in the `/lib` subdirectory of the JDataStore install directory. Each time you add another license, it is appended to this file.
5. Type your name and company number in the Name and Company fields and click OK to complete the process.

Distributing the license

You can distribute the `jdatastore.license` file as a separate file or you can include it in a jar file. In either case, it is important to make sure that your generated license file appears in the classpath before `jds.jar` or `jdsserver.jar`, because those two files contain trial licenses. If the server finds the trial license first, it uses that one.

Note It is recommended that you not place your license file in `jds.jar` or `jdsserver.jar`, because these files get replaced when you upgrade JDataStore.

Distributing JDataStore files

JDataStore includes a number of different jar files. Which ones you need to distribute depend on the type of application you have written. The table below provides a guide for choosing which JDataStore files to distribute with you JDataStore application.

Table 12.1 Description of JDataStore JAR files

File name	Description
`jds.jar`	Local JDBC database connectivity
`dx.jar`	Local DataExpress 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`
	Needed for JdsServer and JdsExplorer GUI interfaces

`beandt.jar`	Needed for compiling and visual design of JavaBean components
`dbswing.jar`	Needed for Swing-based user interfaces
`jbcl-awt.jar`	Needed for Awt-based user interfaces
`<`*jds_home*`>/bin/JdsServer` (Windows: `JdsServer.exe`)	Launcher for the graphical interface to the JDataStore server
`JdsServer.config`	Configuration file for the JdsServer launcher
`<`*jds_home*`>/bin/JdsExplorer` (Windows: `JdsExplorer.exe`)	Launcher for the graphical interface to JdsExplorer
`JdsExplorer.config`	Configuration file for the JdsExplorer launcher
`<`***jds_home***`>/doc/*`	The JDataStore help system content, accessible directly or through JdsExplorer and JdsServer

Chapter 13

Troubleshooting

Here are some guidelines for dealing with difficulties that can arise in connection with creating, maintaining, and accessing JDataStore databases.

Relative path database file names

The Java `user.dir` property dictates how database file names are resolved when a fully qualified path name is not specified. The Java VM defaults this property to the current working directory of the process. This property can be set with a JVM command line option. For example:

```
-Duser.dir=/myapplication
```

This property can also be set from within a java application using the `java.util.System.setProperty` method.

Enable JDBC logging

JDBC logging for a variety of JDBC activities can be enabled in the following ways:

- Call the `java.sql.DriverManager.setLogStream` method.
- Call the `java.sql.DriverManager.setLogWriter` method.
- 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`.

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 is longer than the milliseconds set in the `lockWaitTime` property. In such cases, an exception is thrown that identifies which connection encountered the timeout and which connection

is currently holding the needed lock. The transaction that encounters the lock timeout is not rolled back.

JDataStore has automatic, high speed deadlock detection that should detect all deadlocks. An appropriate exception is thrown that identifies which connection encountered the deadlock and which connection it is deadlocked with. 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. It has information on what tables and what connections are involved.
- Set the `java.sql.DriverManager.SetLogWriter` property to a log writer stream. To restrict log output to lock-related issues, set the `extendedlogFilter` property to `LOCK_ERRORS`.
- Use the `DataStoreConnection.dumpLocks` method 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 stream or row. It can be blocked by another connection that is reading or writing. You can prevent blocks in two ways:

- Minimize the lifespan of transactions that write.
- Use read-only transactions, since they don't require locks to read.

Using short-duration write transactions

Connections should try to 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 aren't blocked by writers or other readers, and because they don't get locks, they never block other transactions.

To make JDBC connections use read-only transactions, set the `readOnly` property of the `java.sql.Connection` object (returned by the `java.sql.DriverManager.getConnection` and `com.borland.dx.sql.dataset.Database.getJdbcConnection` methods) to `true`. When using `DataStoreConnection` objects, set the `readOnlyTx` property to `true` before opening the connection.

Read-only transactions work by simulating a snapshot of the JDataStore 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 see if there were pending changes and roll them back whenever it accessed the data. A snapshot begins when the `DataStoreConnection` opens. It's refreshed every time the `commit` method is called.

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, there are two approaches:

1 Debug using the DataStoreServer JavaBean:

   Inside your application instantiate a `com.borland.datastore.jdbc.DataStoreServer` JavaBean component and execute its `start` method.

2 Debug using the JdsServer:

   - 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
```

   - Execute the JdsServer. The server will not come up until a remote debugger such as JBuilder's is launched to attach to the JdsServer process on port 5000.

Accessing and creating tables from SQL and DataExpress

SQL table creation forces unquoted identifiers to be upper case. Case sensitivity can only be achieved by quoting the identifiers. See "Identifiers" on page 90.

When DataExpress components are used to create a table, the table and column names are case sensitive. If these identifiers are specified in lowercase or mixed case, SQL is not able to access them unless the identifiers are quoted. When DataExpress is used to access a table, the `StorageDataSet storeName` property is case sensitive. However, the column identifiers can be referenced in a case-insensitive fashion. So for DataExpress, an "address" column can be accessed 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.

Non-transactional databases

Set the `saveMode` property to 2 when you are debugging an application that uses a non-transactional JDataStore database. The debugger stops all threads when you are single-stepping through code or when breakpoints are hit. If the `saveMode` property isn't set to 2, the JDataStore daemon thread cannot save modified cache data. For more information, see "Non-transactional database disk cache write options" on page 149.

Verifying JDataStore contents

If you suspect that cache contents were not properly saved on a non-transactional JDataStore database, you can verify the integrity of the file with JdsExplorer. See "Verifying a JDataStore database" on page 37 for more information.

There is also a `com.borland.datastore.StreamVerifier` class with public static `verify` methods that can verify a single stream or all streams in a database. For more information, see the *DataExpress Component Library Reference*.

Note that transactional JDataStore databases have automatic crash recovery when they open. You never need to verify them.

Problems locating and ordering data

Sun Microsystems makes changes to its `java.text.CollationKey` classes from time to time as it corrects problems. The secondary indexes for tables stored inside a JDataStore 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 indexes and rebuild them with the current JDK. The `StorageDataSet.restructure()` method also drops all the secondary indexes.

Resources

Use the Index and Find tabs in the online help to search for information. Another way to find information is to obtain the PDF version of this document and use the Find tool in Acrobat Reader to search the document. You should find the PDF in the PDF subdirectory of the JDataStore install directory.

The books referenced in the introduction may also be of use.

The JDataStore public newsgroup is often helpful. It is at `news://newsgroups.borland.com/borland.public.jdatastore`.

`http://www.google.com/` also has a nice search facility for newsgroups that can be used to see if someone else has already encountered a problem similar to yours. Go to the site and click the Groups tab.

Chapter

14

Using DataExpress with JDataStore

JDataStore provides embedded database functionality in your applications with a single JDataStore database file and the JDataStore JDBC driver (and its supporting classes). No server process is needed for local connections. In addition to industry-standard JDBC support, you can take advantage of the added convenience and flexibility of accessing the JDataStore database directly through the DataExpress API. You can use both types of access in the same application.

JDBC access requires that the JDataStore database be transactional. DataExpress does not require this. This chapter begins with DataExpress access, then discusses transactional JDataStore databases, and finally describes the local JDBC driver. The remote JDBC driver and JDataStore Server are discussed in Chapter 4, "System architecture."

Using DataExpress for data access

To access a JDataStore table using DataExpress, associate a `StorageDataSet` component or one of its extensions, such as `ProcedureDataSet` or `QueryDataSet`, to a table inside a JDataStore database. (The `TableDataSet` extension is identical to `StorageDataSet`.) This allows the `StorageDataSet` to perform navigation and editing operations on the table. This can be thought of as a high powered ISAM (indexed sequential access method) for JDataStore tables.

Using DataStore instead of MemoryStore

By default, once data is loaded into a `StorageDataSet` component, it's stored in memory through the use of a `MemoryStore`. You can create persistent storage systems by setting the `StorageDataSet` object's `store` property. Currently `MemoryStore` and `DataStore`/`DataStoreConnection` are the only implementations of the `Store` interface required by the `store` property.

The main advantages of `DataStore` over `MemoryStore` are transaction semantics, SQL support, and persistence, which enable offline computing. A `DataStore` remembers the rows fetched in a table, even after the application terminates and restarts. In addition, you can use large `StorageDataSets` to increase the performance of any application. `StorageDataSets` using `MemoryStore` have a small performance edge over `DataStore` for a

small number of rows. JDataStore stores data and indexes in an extremely compact format, however. As the number of rows in a `StorageDataSet` increases, using a `DataStore` provides much better performance and requires considerably less memory than using a `MemoryStore`.

You work with a `StorageDataSet` or other data-aware controls connected to the `StorageDataSet` in much the same way whether you are using `MemoryStore` or `DataStore` for data storage. Storing Java objects in columns, however, does require you to use Java serialization (`java.io.Serializable`). If this isn't possible, you can't use `DataStore` components and you should use the default in-memory storage mechanism.

Tutorial: Offline editing with JDataStore

The following tutorial takes you through the steps for creating an application that uses a `DataStore` component to enable the offline editing of data. It shows you how to do this using Borland's JBuilder application, but you could accomplish the same thing in any editing environment.

The server database is a sample JDataStore database file, `employee.jds`, accessed through the JDataStore Server. Don't confuse this file with the JDataStore database file used for persistence. Locate the sample file before beginning. It's installed in `samples/JDataStore/datastores`.

1. In JBuilder, start the JDataStore Server by choosing Tools | JDataStore Server.
2. Create a new application by selecting File | New from the menu and double-clicking the Application icon. In the Application Wizard:
   - On page 1, use the Class name `PersistApp`.
   - On page 2, change the Frame Class name to `PersistFrame`. Click Finish.
3. Switch to design view for the newly created `PersistFrame.java`.
4. Add a `Database` component from the Data Express tab to the component tree.
5. Open the `connection` property editor for the `Database` component in the Inspector. Set the connection properties to the database, using the correct path to the sample `employee.jds` file in place of `<drive letter>:/<jdatastore_home>` in the URL:

Property name	Setting
Driver	`com.borland.datastore.jdbc.DataStoreDriver`
URL	`jdbc:borland:dsremote://localhost/<drive letter>:/<jdatastore_home>/samples/JDataStore/datastores/employee.jds`
Username	Use any name.
Password	Leave blank.

6. Click the Test Connection button to check that you've set the connection properties correctly. When the connection is successful, click OK.
7. Add a `DataStoreConnection` component from the Data Express tab to the component tree. Adding a `DataStoreConnection` component writes an `import` statement for the `datastore` package to your code and adds the JDataStore library to your project properties if it wasn't already listed.
8. Open the `fileName` property editor for the `DataStoreConnection` component. Type in a name for the new JDataStore database file. Be sure to include the full path. You can use the Browse button to help. You don't have to specify a file extension because a JDataStore database always has the extension `.jds`. Click OK.

Note The Designer automatically creates this JDataStore database for you when it's connected to the `StorageDataSet` so that the tools work fully. When you run the application, the JDataStore database file is already there. But if you run the

application on another computer, the JDataStore database file won't be there. You will have to add extra code to create the JDataStore database file if necessary as shown in "Creating a JDataStore database file" on page 182.

9 Add a `QueryDataSet` component from the Data Express tab to the component tree.

10 Open the `query` property editor for the `QueryDataSet` component in the Inspector and set the following properties:

Property name	Value
Database	`database1`
SQL Statement	`select * from employee`

11 Click Test Query to ensure that the query is runnable. When the gray area beneath the button indicates Success, click OK to close the dialog box.

12 Set the `storeName` property of the `QueryDataSet` to `employeeData`.

13 Set the `store` property to `dataStoreConnection1` (the only choice).

14 Add a `JdbNavToolbar` component from the dbSwing tab to the North position of the frame. Set its `dataSet` property to `queryDataSet1`.

15 Add a `JdbStatusLabel` component from the dbSwing tab to the South position of the frame. Set its `dataSet` property to `queryDataSet1`.

16 Add a `TableScrollPane` component from the dbSwing tab to the Center position of the frame.

17 Add a `JdbTable` component from the dbSwing tab to the `TableScrollPane`. Set its `dataSet` property to `queryDataSet1`.

18 Instead of adding code to call `DataStore.shutdown()` before exiting the application, you can use a `DBDisposeMonitor` component to close JDataStore database files automatically when you close the frame.

19 Add a `DBDisposeMonitor` component from the More dbSwing tab to the component tree. Set its `dataAwareComponentContainer` property to `this`.

20 Run `PersistApp.java`.

In the running application, make some changes to the data and click the Post button on the navigator to save the changes to the JDataStore database file (the filename you specified in step 8). Changes are also saved to the file when you move off a row, just as they are with an in-memory data set created with `MemoryStore`.

Note A table in a JDataStore database can have thousands or hundreds of thousands of rows. Handling that much data using an in-memory data set would greatly slow application performance.

If you want, you can exit the application you just created and then run it again. Without any connection to the SQL database, you can continue to view and edit data in the JDataStore database because you created a permanent copy with the `StorageDataSet`. You'll find this especially useful when you want to work with data offline at home or on an airplane.

Understanding how JDataStore manages offline data

So far in the tutorial, nothing has been saved back to the SQL database on the server. On the `JdbNavToolbar`, there are several buttons:

- The Post button saves the changes in the current row to the JDataStore database file.
- The Save button saves all changes that have been accumulated in the database back to the server. DataExpress automatically figures out how to resolve changes

back to the SQL server. In code, the corresponding method is `DataSet.saveChanges()`.

- The Refresh button reruns the query, overwriting the data in the database with the results of the query, including any edits not saved back to the server. In code, the corresponding method is `DataSet.executeQuery()`.

Options set in the `queryDescriptor` also have an effect on how data is stored, saved, and refreshed. In the `queryDescriptor` in this example, we selected the `Execute Query Immediately When Opened` option. This option specifies how data is loaded into the database when the application is first run. On subsequent runs, the execution of the query is suppressed because the data set is found in the JDataStore database instead of on the server. This has the following results:

- Changes that haven't been saved to the server are preserved when you exit and restart the application.
- You don't need to write special code to get data into the JDataStore database on the first run.
- Once data is in the database, you can work offline. In fact, a connection to the database is not even established until you perform an operation that requires it, such as saving changes.

When the `Execute Query Immediately When Opened` option is selected, existing data can't be overwritten (unless you call the `StorageDataSet.refresh()` method explicitly). This means that you can safely close and reopen a data set to change property settings in either a `MemoryStore` or in a `DataStore` without losing editing changes.

Once you've got data in the database, you can run this application and edit data whether the database server is available or not. When you are working offline, you have to remember not to click the navigator's Save or Refresh button. If you do, you'll get an exception because the attempt to connect will fail, but you won't lose any of the changes you have made.

Binding a StorageDataSet to a table

To cache and persist data, you can bind a `StorageDataSet` component to a JDataStore table by setting the values of the three properties discussed below. Persisting data from a provider usually involves `QueryDataSet` (which uses `QueryProvider`) or `ProcedureDataSet` (which uses `ProcedureProvider`). These are the two subclasses of `StorageDataSet` that have predefined providers.

The `StorageDataSet` class is the focal point of DataExpress semantics. `StorageDataSet` has three subclasses that are used for different kinds of data sources:

- `QueryDataSet` is for data from SQL queries.
- `ProcedureDataSet` is for data from a SQL stored procedure.
- `TableDataSet` has no predefined provider of data. It is exactly the same as `StorageDataSet`.

In the following discussion, we use the `StorageDataSet` component to create a new table.

Each `StorageDataSet` has a `store` property that is `null` when the object is instantiated. If it's still `null` when the dataset is opened, a `com.borland.dx.memorystore.MemoryStore` is assigned automatically, which means that the data is stored in memory. If you assign a `DataStoreConnection` or `DataStore` to the `store` property, the data is stored in a persistent JDataStore component instead.

To connect (or "bind") a `StorageDataSet` to a JDataStore table, assign values to these three properties:

- Set the `fileName` property of the `DataStoreConnection`to the name of the JDataStore database to connect to.
- Set the `storeName` property of the `StorageDataSet`to the name of the table stream inside the JDataStore database. You can reuse an existing name if it's for the same table. Otherwise, you must use a new name. It's up to you to manage what's inside the JDataStore database and to choose names that don't conflict.
- Set the `store` property of the `StorageDataSet` to the `DataStoreConnection` (or `DataStore`) object. This connects the two together.

Perform these three steps in any order. Once you've set all three properties, you have a fully qualified connection between a `StorageDataSet` and a JDataStore table.

In `DxTable.java`, the JDataStore database file is `Basic.jds`, which you created in "Creating a JDataStore database file" on page 182. The table stream is named "Accounts." Think of it as the name of the table. `DxTable.java` assigns `DataStoreConnection` as the value of the `StorageDataSet`'s `store` property.

Opening a `StorageDataSet` that is bound to a JDataStore table automatically opens that JDataStore database. If the database opens successfully, the program creates the named table stream if it doesn't already exist. If it does exist, then the table stream opens. This establishes an open connection between the `StorageDataSet` and its table stream in the JDataStore database.

Demonstration class: DxTable.java

Create a new file in the `dsbasic` package and name it `DxTable.java`:

```
// DxTable.java
package dsbasic;

import com.borland.datastore.*;
import com.borland.dx.dataset.*;

public class DxTable {

    DataStoreConnection store = new DataStoreConnection();
    StorageDataSet        table = new StorageDataSet();

    public void demo() {
        try {
            store.setFileName( "Basic.jds" );
            table.setStoreName( "Accounts" );
            table.setStore( store );
            table.open();
        } catch ( DataSetException dse ) {
            dse.printStackTrace();
        } finally {
            try {
                store.close();
                table.close();
            } catch ( DataSetException dse ) {
                dse.printStackTrace();
            }
        }
    }

    public static void main( String[] args ) {
        new DxTable().demo();
    }
}
```

Because the program uses DataExpress, it imports the DataExpress package in addition to the JDataStore package. The class has two fields: a `DataStoreConnection` and a `StorageDataSet`. The `main()` method instantiates a new instance of the class and executes its `demo()` method.

Creating JDataStore tables with DataExpress

Opening a `StorageDataSet` that is bound to a JDataStore table results in an open table stream. For new table streams, `QueryDataSet` and `ProcedureDataSet` then invoke their provider to populate the table stream as explained in "Tutorial: Offline editing with JDataStore" on page 164. But `StorageDataSet` has no provider. You start with an empty and undefined table stream.

Add the highlighted statements to `DxTable.java`:

```
try {
   table.open();
      if ( table.getColumns().length == 0 ) {
         createTable();
      }
} catch ( DataSetException dse ) {
```

To detect that a table stream is new, check the number of columns in the `StorageDataSet`. If there are no columns, you can then define the columns in the table. In this case, it's done by a method called `createTable()`. Add it to `DxTable.java`:

```
public void createTable() throws DataSetException {
    table.addColumn( "ID"    , Variant.INT );
    table.addColumn( "Name"  , Variant.STRING );
    table.addColumn( "Update", Variant.TIMESTAMP );
    table.addColumn( "Text"  , Variant.INPUTSTREAM );
    table.restructure();
}
```

In this demonstration program, the `createTable()` method uses the simplest form of the `StorageDataSet.addColumn()` method to add columns individually by name and type. The columns have no constraints. Character columns, defined as `Variant.STRING`, can contain strings of any length. You can define columns with constraints by defining `Column` objects, setting the appropriate properties such as `precision`, and then adding them with the `addColumn()` or `setColumns()` methods to the table.

After you modify the structure of the table by adding these new columns, activate the changes by calling the `StorageDataSet.restructure()` method. The result is an empty but structured table stream, a new table in the JDataStore database.

If you know the table doesn't exist, you can use `addColumns()` to define the structure before opening the `StorageDataSet`. Then you won't need to call `restructure()`.

You can store as many tables as you want in a single JDataStore database file. Each stream must have a unique name. You can use the same `DataStoreConnection` object in the `store` properties of each `StorageDataSet`.

There are at least two other ways to create tables in a JDataStore database: you can use JDSExplorer, or you can use a `SQL CREATE TABLE` statement through the JDataStore JDBC driver.

Using JDataStore tables with DataExpress

Once the tables in the JDataStore database have been defined (no matter how they were created), you can use the rest of the DataExpress API through a `StorageDataSet` object, just as you would with any dataset. You can create filters, indexes, master-detail links, and so on. Such secondary indexes are also persisted and maintained in the JDataStore database.

To complete the demonstration program, add a smattering of DataExpress functionality with this new method:

```
public void appendRow( String name ) throws DataSetException {
    int newID;
    table.last();
    newID = table.getInt( "ID" ) + 1;
    table.insertRow( false );
    table.setInt( "ID", newID );

    table.setString( "Name", name );
    table.setTimestamp( "Update", new java.util.Date().getTime() );
    table.post();
}
```

Add the highlighted statements to the `demo()` method:

```
if ( table.getColumns().length == 0 ) {
        createTable();
    }
    table.setSort( new SortDescriptor( new String[] {"ID"} ) );
    appendRow( "Rabbit season" );
    appendRow( "Duck season" );
    table.first();
    while ( table.inBounds() ) {
        System.out.println( table.getInt( "ID" ) + ": "
                + table.getString( "Name" ) + ", "
                + table.getTimestamp( "Update" ) );
        table.next();
    }
} catch ( DataSetException dse ) {
```

What this code does: The program opens the table, creating the table's structure as necessary. It then sets a `SortDescriptor` on the ID field. To add some rows, it calls the `appendRow()` method. The `appendRow()` method begins by going to the last row in the table and obtaining the value of the ID field. Because of the sort order, this value should be the highest ID number used so far. (If the table is empty, the `getInt()` method returns zero.) The new ID value is one greater than the last. `appendRow()` inserts a new row and sets its attributes, including the Update field, which is set to the current date and time. Finally `appendRow()` saves the new row by calling the `post()` method. After appending a few rows, a loop navigates through the table, displaying its contents in the console.

Finally, close the JDataStore database and `StorageDataSet`.

If you run the program a few times, you'll see that the new rows get unique ID numbers. This method of generating ID numbers works for a simple single-threaded demonstration program like this that always commits new rows after getting the old ID number. But for more realistic programs, such an approach might not be safe. To use a more robust approach, you must understand locks and transactions.

Transactional JDataStore databases

So far, changes you have made to a JDataStore database have been direct and immediate. If you write an object, change some bytes in a file stream, or add a new row to a table, it's done without concern for other connections that might be accessing the same stream. Such changes are immediately visible to all other connections.

While this behavior is safe for simple applications, more robust applications require some level of transaction isolation. Not only do transactions ensure that you are not reading dirty or phantom data, but you can also undo changes made during a

transaction. Transaction support also enables automatic crash recovery. It's required for JDBC access.

See "Transaction management" on page 20 for an overview of JDataStore transactions and a discussion of isolation levels and lock management.

Enabling transaction support

Transaction support is provided by the `com.borland.datastore.TxManager` class. A JDataStore database can be transactional when it is first created, or you can add transaction support later. In either case, you assign a `TxManager` object as the value of the `txManager` property of the `DataStore` object, usually before calling the `create()` or `open()` method.

If the `TxManager.ALogDir` and `TxManager.BLogDir` properties are not set, the location of the log files is always assumed to be in the same directory as the JDataStore database file. This behavior lets you move the JDataStore database file from one directory to another without warnings that the log files exist in the original location and the new location.

If you assign the `txManager` property on an open JDataStore database, it causes the `TxManager` to automatically shut down and attempt to reopen the database so that the new property setting can take effect immediately. If the `DataStoreConnection.userName` property has not been set, the JDataStore database fails to reopen, and an exception is thrown.

The properties of the `TxManager` object determine various aspects of the transaction manager. When instantiated, the `TxManager` has usable default settings for these properties. If you want to change any of these settings, it's better to do it before creating or opening the JDataStore database.

The first time the now-transactional JDataStore database opens, it stores its transactional settings internally. The next time you open the database, you don't have to assign a `TxManager`. Instead the JDataStore database automatically instantiates a `TxManager` with its stored settings.

To open (or create) a transactional JDataStore database, you must also set the `DataStoreConnection.userName` property. The `userName` property is used to identify individuals in a multiuser environment when necessary, such as during lock contention. If there is no name in particular that you find appropriate, you can set it to a dummy name.

Creating new transactional JDataStore databases

Here's the minimum code for creating a new transactional JDataStore database with default settings:

```
DataStore store = new DataStore();

store.setFileName( "SomeFileName.jds" );
store.setUserName( "AnyNameWouldWork" );
store.setTxManager( new TxManager() );
store.create();
```

The two differences between this code and that for a non-transactional JDataStore database is the setting of the `userName` and `txManager` properties. If you don't want the

default settings, the code generally looks something like this:

```
DataStore store = new DataStore();
TxManager txMan = new TxManager();

// Make changes to TxManager
txMan.setRecordStatus( false );

store.setFileName( "SomeFileName.jds" );
store.setUserName( "AnyNameWouldWork" );
store.setTxManager( txMan );
store.create();
```

In this example, the `recordStatus` property, which controls whether status messages are written, is set to `false`.

Adding transaction support to existing JDataStore databases

The code for making an existing JDataStore database transactional is very similar. The main difference is that you use the `open()` call instead of `create()`. For a default `TxManager`, the code might look like this:

```
DataStore store = new DataStore();

store.setFileName( "SomeFileName.jds" );
store.setUserName( "AnyNameWouldWork" );
store.setTxManager( new TxManager() );
store.open();
```

Note that even though you are much more likely to use a `DataStoreConnection` to open an existing JDataStore database, you can't use one when you are adding transaction support, because `txManager` is a property of `DataStore`, not `DataStoreConnection`.

Opening a transactional JDataStore database

The only difference when opening a JDataStore database that's transactional and one that's not is that you must specify a `userName`. Because it doesn't hurt to specify a `userName` for a non-transactional JDataStore database (it's simply ignored), you might want to always specify a `userName` when opening a JDataStore database. The code would look something like this:

```
DataStoreConnection store = new DataStoreConnection();

store.setFileName( "SomeFileName.jds" );
store.setUserName( "AnyNameWouldWork" );
store.open();
```

Because no `TxManager` was assigned, when the JDataStore database opens, a `TxManager` is automatically instantiated with its properties set to the values that were persisted in the database. The `TxManager` is assigned to the database's `txManager` property. You can get the values of the persisted transaction management properties from there, but you can't change them directly.

Changing transaction settings

To change a JDataStore database's transaction setting, assign a new `TxManager` object before opening. The `TxManager` object knows which properties have been assigned and which ones have been left at their default value. If you assign a `TxManager` to a transactional JDataStore database, only those properties that have been assigned in

the new TxManager are changed. All other properties remain as they were; they do not revert to the default values in the new TxManager.

You should assign the TxManager with the new values before you open the JDataStore database. For example, suppose you want to change the softCommit property to true. Doing so improves performance by not guaranteeing recently committed transactions (within approximately one second before a system failure) yet it still guarantees crash recovery:

```
DataStore store = new DataStore();
TxManager txMan = new TxManager();

// Make changes to TxManager
txMan.setSoftCommit( true );

store.setFileName( "SomeFileName.jds" );
store.setUserName( "AnyNameWouldWork" );
store.setTxManager( txMan );
store.open();
```

Note that the other properties, such as recordStatus, aren't set. Although the new TxManager has the default setting when it is assigned to the JDataStore database, the setting in the database isn't affected, even if it's not the default.

Transaction log files

The transaction manager works by logging changes made to the JDataStore database, including the previous values, so that the transaction can be rolled back. Changes aren't removed from the log file when the changes are committed, so if you archive the log files, it's possible to extract a complete change log or to reconstruct the contents of the older JDataStore database.

The TxManager properties control these attributes of the transaction log files:

- Whether to duplex them; that is, whether to keep two separate but identical copies for greater reliability at the expense of some performance.
- Where to put them. When the log files are duplexed, the two copies are usually kept in different locations. Keeping them on different physical drives increases reliability and the chance for recovery even further, and it might also offset some of the performance penalty.
- How big each one can get before another one is started.

By default, you get one copy of the log files (simplexing instead of duplexing) in the same directory that contains the JDataStore database.

The first time a JDataStore database is transaction-enabled, it creates its log files. The log file names use the name of the JDataStore database file without the file extension. For example, if the JDataStore database MyStore.jds uses simplex transaction logging, these log files are created:

- The status file MyStore_STATUS_0000000000,
- The anchor file MyStore_LOGA_ANCHOR
- The record file MyStore_LOGA_0000000000.

Duplex logging adds the files MyStore_LOGB_ANCHOR and MyStore_LOGB_0000000000. These two sets of log files are referred to as the "A" and "B" log files. The ALogDir and BLogDir properties control the location of these files.

Once a log files reaches the size determined by the TxManager's maxLogSize property, additional status and record files are created with the log file number incrementing by one each time. As old log files are no longer needed for active transactions or crash recovery, they are automatically deleted.

Moving transaction log files

If the ALogDir and BLogDir properties are not set, then the location of the log files is always assumed to be in the same directory as the directory of the JDataStore database file. This makes it easier to move the JDataStore database from one directory to another. If the ALogDir and BLogDir properties are set, they include the drive and full path, which means two things:

- If you're creating transactional JDataStore databases in one location but know that you will be moving them to another location, try to make the path in the creation location the same as the path in the deployment location. For example, if you intend to deploy the files to the D: drive, but the JDataStore database files were created on the C: drive because you don't have a D: drive on your development computer, you must go through the extra steps of moving the log files when you deploy because the drives are different.
- To move log files, follow these steps:
  a. Move the log files to the new location. Be sure to remove or rename any copies of the files in the original location.
  b. Create a new TxManager with the new location property settings. Assign it to the txManager property of the DataStore.
  c. Open the JDataStore database. The TxManager looks in the new location, sees that the log files are there, and changes the persisted settings in the JDataStore database.

Bypassing transaction support

Sometimes you want to access a transactional JDataStore database, but need to bypass transaction support. Here are some examples of how this might happen:

- The transaction log files are lost. You can't open the JDataStore database normally.
- You only have read-only access to the JDataStore database. For example, it might be on a CD-ROM or on a network directory where you don't have write access.

In such cases, you can temporarily bypass transaction support by opening the JDataStore database in read-only mode. Do this through a DataStore object. Before opening the JDataStore, set its readOnly property to true. For example,

```
DataStore store = new DataStore();

store.setFileName( "SomeReadOnly.jds" );
store.setReadOnly( true );
store.open();
```

Because you are bypassing the TxManager, you don't need to set the userName. If the transaction log files are lost, use the copyStreams() method to copy the streams to another file.

Removing transaction support

To make a JDataStore database non-transactional, assign a new TxManager that has its enabled property set to false. (Its default value is true.) If the DataStore's consistent property is false, the JDataStore database is internally inconsistent and you won't be allowed to make the change. Because you are disabling the TxManager, you don't need to set the userName.

The following code removes transaction support:

```
DataStore store = new DataStore();
TxManager txMan = new TxManager();

// Disable TxManager
txMan.setEnabled( false );

store.setFileName( "SomeFileName.jds" );
store.setTxManager( txMan );
store.open();
```

Disabling the `TxManager` doesn't remove any existing log files. Disabling the `TxManager` *does* cause all `TxManager` properties to be forgotten. If you make the JDataStore database transactional again, the `TxManager` properties revert to their defaults, so if the `ALogDir` and `BLogDir` properties were previously set to a non-default value, you must remember to set them again.

Deleting transactional JDataStore databases

When you delete a transactional JDataStore database, be sure to delete its log files also. If you don't, you won't be allowed to create a new JDataStore database with the same name, because the log files won't match.

Controlling JDataStore transactions

Once you've made a JDataStore database transactional, you can usually ignore the `TxManager`. The only time you need to reference a `TxManager` is when you want to examine or change the JDataStore database's transaction settings. The interface for controlling transactions is on the `DataStoreConnection` object, primarily through the `commit()` and `rollback()` methods.

Understanding the transaction architecture

Each `DataStoreConnection` is a separate transaction context. This means that all the changes made through a particular `DataStoreConnection` are treated as a group and are separate from changes made through all others.

Note that as a subclass of `DataStoreConnection`, a `DataStore` object can also act as a separate transaction context. The difference is that you can have just one `DataStore` object accessing a particular JDataStore database, while you can have many `DataStoreConnection` objects. When you open a `DataStoreConnection`, it contains a reference to a `DataStore` object, as explained in "Referencing the connected JDataStore database" on page 190. If there is no suitable `DataStore` object in memory, the `DataStoreConnection` automatically opens a `DataStore` to satisfy this reference. This means that if you open a `DataStoreConnection` first, subsequent `DataStore` objects accessing the same JDataStore database file have their allowed functionality reduced so that they behave like a `DataStoreConnection`.

Committing and rolling back transactions

Transaction control uses three methods of `DataStoreConnection`:

- To see if a transaction has been started, call `transactionStarted()`.
- To commit a transaction, call `commit()`.
- To roll back a transaction, call `rollback()`.

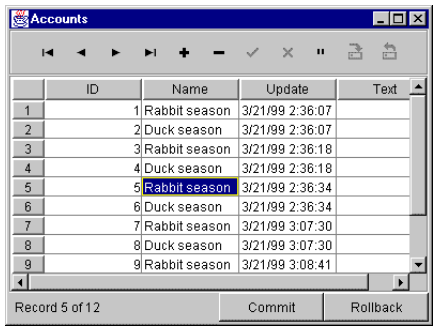
When you close `DataStoreConnection`, it attempts to commit any pending transaction. You can control this automatic behavior by listening to the `DataStore`'s `Response` event for a `COMMIT_ON_CLOSE`, as shown in the following tutorial.

Tutorial: Using DataExpress to control transactions

This tutorial uses JBuilder to create a simple Swing-based application that can commit and roll back transactions. It also detects the automatic commit on close, allowing the user to decide whether to commit. In addition, it shows some important details about using a JDataStore database in a GUI application. You don't need JBuilder to do this. You can use any editing environment

The end result looks like this:

Figure 14.1 The complete AccountsFrame



Step 1: Create a transactional JDataStore table with test data

If you completed the previous tutorial, you already have a JDataStore database with some data in it, `Basic.jds`. Instead of making that file transactional, make a copy of the file and make the copy transactional. This way, you have both kinds of JDataStore databases, transactional and non-transactional, to play with.

Make a copy of the file, naming it `Tx.jds`. Then add the following program to the project, `MakeTx.java`:

```
// MakeTx.java
package dsbasic;

import com.borland.datastore.*;

public class MakeTx {

    public static void main( String[] args ) {
        if ( args.length > 0 ) {
            DataStore store = new DataStore();

            try {
                store.setFileName( args[0] );
                store.setUserName( "MakeTx" );
                store.setTxManager( new TxManager() );
                store.open();
                store.close();
            } catch ( com.borland.dx.dataset.DataSetException dse ) {
                dse.printStackTrace();
            }
        }
    }
}
```

This utility program makes any JDataStore database transactional if it isn't already. For JDataStore databases that are already transactional, nothing happens because no properties are set on the `TxManager` object.

Set the runtime parameters in the Project Properties dialog box to `Tx.jds` and run the program. It takes a moment to create the three transaction log files `Tx_STATUS_0000000000`, `Tx_LOGA_ANCHOR`, and `Tx_LOGA_0000000000`.

Step 2: Create a data module

The next step creates a data module with the JDataStore database and a `StorageDataSet`:

1. Select File | New.
2. Select Data Module in the dialog box and click OK.
3. In the Data Module Wizard, make sure the Package name is `dsbasic` and set the Class name to `AccountsDM`. Make sure the Invoke Data Modeler check box is not selected. Click OK.
4. Switch to design view for the new file `AccountsDM.java`.
5. Add a `DataStore` component from the Data Express tab to the component tree. Change its name to `dataStore` (easily done by pressing *F2* after adding it to the tree).
6. In the Inspector, use the file chooser to set the `fileName` property of the `DataStore` to the `Tx.jds` you created earlier, and set the `userName` property to some name.
7. Add a `StorageDataSet` component from the Data Express tab to the component tree.
8. In the Inspector, set the `storeName` property to `Accounts` and the `store` property to `dataStore`.
9. Switch back to source view to see the generated code.
10. Save the file.

Step 3: Create a GUI for the JDataStore table

Create a simple table grid to display the data:

1. Select File | New.
2. Select Application in the dialog box and click OK.
3. On page 1 of the Application wizard, set the Class name to `AccountsApp`. Click Next.
4. On page 2 of the Application wizard, set the Frame Class name to `AccountsFrame`, and the Title to `Accounts`. Make sure the Center frame on screen check box is selected; deselect the rest. Click Finish.
5. Switch to design view for the new file `AccountsFrame.java`.
6. Select Wizards | Use DataModule.
7. The wizard should scan, find, and select the `AccountsDM` data module. Set the Field name to `dataModule`. Select the option to use a shared (static) instance. Click OK.
8. Add a `JdbNavToolBar` component from the dbSwing tab to the North position of the frame.
9. Add a `JdbStatusLabel` component from the dbSwing tab to the South position of the frame.
10. Add a `TableScrollPane` component from the dbSwing tab to the Center position of the frame.
11. Add a `JdbTable` component from the dbSwing tab to the `TableScrollPane`.
12. In the Inspector, set the `dataSet` property for all three `Jdb` components to `dataModule.StorageDataSet1` (the only choice).
13. Switch back to source view.

14 Go to the `processWindowEvent()` method. This method is generated so that `System.exit()` is called when the window is closed. It's important that you close the JDataStore database before terminating the program.

   In this case, with only one connection, the `close()` method would work, but because you are calling `System.exit()`, you want to make sure the JDataStore database is closed, no matter how many connections the application is using. You should use `DataStore.shutdown()` in this situation, which closes the JDataStore database file directly. That is why this application uses a `DataStore` instead of a `DataStoreConnection`.

   You could place the `shutdown()` method call just before `System.exit()`, but for reasons that you'll see soon, you want to do this before the window physically closes. Insert the highlighted statements:

```
//Overridden so we can exit on System Close
protected void processWindowEvent(WindowEvent e)

(e.getID() == WindowEvent.WINDOW_CLOSING) {
   try {
     dataStore.shutdown();
   } catch ( DataSetException dse ) {
       dse.printStackTrace();
     }
  }
super.processWindowEvent(e);
if (e.getID() == WindowEvent.WINDOW_CLOSING) {
    System.exit(0);
}
```

15 That code references the `DataStore` object as `dataStore`, which hasn't been defined. First, add the following `import` statements:

```
import com.borland.datastore.*;
import com.borland.dx.dataset.*;
```

16 Declare a new field (after the components is a good place to do this):

```
TableScrollPane tableScrollPane1 = new TableScrollPane();
JdbTable jdbTable1 = new JdbTable();
DataStore dataStore;
```

17 Get a reference to the `DataStore` from the data module. Add the highlighted statement to the `jbInit()` method:

```
private void jbInit() throws Exception  {
  dataModule = dsbasic.AccountsDM.getDataModule();
  dataStore = dataModule.getDataStore();
```

Run `AccountsApp.java`. You can navigate through the table and add, edit, and delete rows. All the changes you make are done within the context of a single transaction, although this is not apparent at this point. When you close the window, the JDataStore database closes, and the changes you made are committed. You can verify this by running the application again.

If you had left the JDataStore database open and had not committed the transaction before terminating the application, you would have an uncommitted transaction in the transaction log. As a result, the changes would have been orphaned and not written to the JDataStore database. No changes you made in the application would ever apply. Closing the JDataStore database commits those changes automatically.

Step 4: Add direct transaction control

This step adds direct control over the transaction by allowing the user to explicitly commit and roll back the current transaction:

1. Switch to design view for `AccountsFrame.java`.
2. Delete the `JdbStatusLabel` object.
3. Add a `JPanel` component from the Swing Containers tab to the South position of the frame.
4. Set its `layout` property to `GridLayout`.
5. Add a `JdbStatusLabel` component from the dbSwing tab to the `JPanel`.
6. Set its `dataSet` property to `dataModule.StorageDataSet1` (the only choice).
7. Add another `JPanel` component from the Swing Containers tab to the first `JPanel`. It should appear to the right of the `JdbStatusLabel`.
8. Set its `layout` property to `GridLayout`.
9. Add a `JButton` component from the Swing tab to the nested `JPanel`.
10. Set its name to `commitButton` and its `text` property to `Commit`.
11. Add another `JButton` component from the Swing tab to the nested `JPanel`.
12. Set its name to `rollbackButton` and its `text` property to `Rollback`.
13. Set the `actionPerfomed` event handler for the `commitButton`:

```
void commitButton_actionPerformed(ActionEvent e) {
  try {
      dataStore.commit();
  } catch ( DataSetException dse ) {
      dse.printStackTrace();

  }
}
```

14. Set the `actionPerfomed` event handler for the `commitButton`:

```
void rollbackButton_actionPerformed(ActionEvent e) {
  try {
      dataStore.rollback();
  } catch ( DataSetException dse ) {
      dse.printStackTrace();
  }
}
```

These buttons now call `commit()` or `rollback()` on the `DataStore` to commit or roll back any changes made during the current transaction. The current transaction is all that has happened since the last commit or rollback.

Control transaction handling when a connection closes

This last step enables the application to handle uncommitted transactions when the JDataStore database is closed and allows the user to decide whether to commit or rollback changes.

1. In `AccountsFrame.java`, modify the class definition so that it implements `ResponseListener`:

```
public class AccountsFrame extends JFrame implements ResponseListener {
```

2 Add the `response` method for the `ResponseListener` interface:

```
public void response( ResponseEvent response ) {
    if ( response.getCode() == ResponseEvent.COMMIT_ON_CLOSE ) {
        if ( JOptionPane.showConfirmDialog( this,
                "Posted changes have not been committed. Do that now?",
                "Commit or rollback",
                JOptionPane.YES_NO_OPTION ) == JOptionPane.YES_OPTION ) {
            response.ok();
        } else {
            response.cancel();
        }
    }
}
```

This method checks for the `COMMIT_ON_CLOSE` event. When that occurs, a simple yes/no dialog box appears, asking the user if they want to commit the changes. “Yes” sends the `ok` response, which signals the JDataStore to commit the changes. “No” sends the `cancel` response, which signals the JDataStore to roll back the changes.

3 Add the highlighted statement to add the frame as one of the JDataStore’s `ResponseListener`s:

```
private void jbInit() throws Exception  {
   dataModule = dsbasic.AccountsDM.getDataModule();
   dataStore = dataModule.getDataStore();
   dataStore.addResponseListener( this );
```

With these additions, the user gets a dialog box if there are unsaved changes. The dialog box asks the user whether to commit them. Remember that the JDataStore database is closed before the window is. If it’s not, the dialog box would appear after the window had already disappeared.

You can now run the completed application. In addition to using the buttons to commit and roll back changes, try making some changes and then close the window to exercise the commit handling when the file closes.

Chapter

15

The JDataStore file system

This section contains several simple tutorials that demonstrate basic JDataStore file system concepts. If you haven't already read Chapter 2, "Introduction," please take a moment to do so before beginning the tutorials.

The first part of this chapter covers the fundamentals using JDataStore file streams. For information about working with table streams, see "Creating a basic JDBC application using JDataStore" on page 197, "Using DataExpress with JDataStore" on page 163, and "Binding a StorageDataSet to a table" on page 166. You may also wish to look at the sample that uses JDataStore to create a basic JDBC application in `/samples/JDataStore/HelloJDBC/`.

JDataStore basics

Chapter 4, "System architecture" provides details about many JDataStore fundamentals.

- Drivers are discussed in "JDataStore programmatic interfaces" on page 13.
- A JDataStore database file can contain three types of data streams: *table* streams and two types of *file* streams. See "The JDataStore file system" on page 16 for more information about how JDataStore makes use of file streams.
- Read "Transaction management" on page 20 to understand how JDataStore handles transactions.

Serializing objects

A `DataStore` is a JavaBean component that can be used to access a JDataStore database. This tutorial provides some simple examples that make use of the DataStore JavaBean component using the JBuilder IDE.

The classic first exercise for a new language is how to display "Hello, World!" We'll carry on that tradition here.

First, create a new project for the `dsbasic` package, which you'll use throughout this chapter.

Important Add the JDataStore library to the project so that you can access the JDataStore classes. If you are using JBuilder and don't know how to create a project or add a library, see "Adding a required library to a project" in JBuilder's *Developing Database Applications* for instructions on adding a required library.

Demonstration class: Hello.java

Add a new file to the project, `Hello.java`, and type in this code:

```
// Hello.java
package dsbasic;

import com.borland.datastore.*;

public class Hello {

    public static void main( String[] args ) {
        DataStore store = new DataStore();

        try {
            store.setFileName( "Basic.jds" );
            if ( !new java.io.File( store.getFileName() ).exists() ) {
                store.create();
            } else {
                store.open();
            }
            store.close();
        } catch ( com.borland.dx.dataset.DataSetException dse ) {
            dse.printStackTrace();
        }
    }
}
```

After declaring its package, this class imports all the classes in the `com.borland.datastore` package. That package contains most of the public JDataStore classes. (The rest of the public JDataStore classes are in the `com.borland.datastore.jdbc` package, which is needed only for JDBC access. It contains the JDBC driver class, and classes used to implement a JDataStore Server. These classes are covered in Chapter 14, "Using DataExpress with JDataStore" and Chapter 3, "JDBC quickstart.") You can also access JDataStore through DataExpress components (packages under `com.borland.dx`). In this example, these classes are referenced explicitly so that you can see where each class comes from.

Creating a JDataStore database file

A new `DataStore` object is created in the `main()` method of `Hello.java`. This object represents a physical JDataStore database file—a database—and it contains properties and methods that represent its structure and configuration.

Next, the name "Basic.jds" is assigned to the `DataStore` object's `fileName` property. It contains the default file extension ".jds" in lowercase. If the file name doesn't end with the default extension, the extension is appended to the file name when the property is set.

You can't create the JDataStore database if a file with that name already exists in the directory. If the file doesn't exist, the `create()` method creates it. If the method fails for any reason (for example, there's no room on the disk, or someone just created the file in the nanoseconds between this statement and the last), it throws an exception. If the method succeeds, you have an open connection to a new JDataStore database file.

For more information about JDataStore databases, See “Creating a new JDataStore database” on page 34. When creating the file, you can also specify options such as block size and whether the JDataStore database is transactional.

Opening and closing a connection

If the file does exist, a connection opens through the `open()` method. The `open()` method is actually a method of the `DataStore` class' superclass, `DataStoreConnection`, which contains properties and methods for accessing the contents of a JDataStore database. (The `fileName` property is also a property of `DataStoreConnection`, which means that you can and often do access a JDataStore database without a `DataStore` object, as you'll see shortly.) Because `DataStore` is a subclass of `DataStoreConnection`, it has its own built-in connection, which is suitable for simple applications like this. (Note that `DataStore` can create a new JDataStore database file, but `DataStoreConnection` cannot.)

But the excitement is short-lived. Immediately after opening a connection to the JDataStore database, creating the database in the process if necessary, that connection closes with the `close()` method. The `close()` method is also inherited from `DataStoreConnection`. Because there was only one built-in connection, when all the connections to the JDataStore database are closed, the JDataStore database itself shuts down.

You must close any connections that you open before you exit your application (or call the `DataStore.shutdown()` method, which closes all connections). Opening a connection starts a daemon thread that continues to run and prevents your application from terminating properly. If you don't close the connections, your application hangs on exit.

Handling basic JDataStore exceptions

Most of the methods in the JDataStore classes can throw a `DataSetException`, or more specifically, one of its subclasses, `DataStoreException`. Most of these exceptions are of the fatal “should never happen” or “don't do that” variety. For example, you can't set the `fileName` property if the connection is already open. You can't create the JDataStore database if one already exists. You can't open a connection if the named file isn't really a JDataStore database file. You might get a `java.io.IOException` when writing data while closing a connection.

Therefore, almost all JDataStore code is inside a `try` block. In this case, if an exception is thrown, a stack trace prints.

Deleting JDataStore database file

If you run the application now, all it does is create the file `Basic.jds`. If you then run it a second time, it does even less—just opening and closing a connection. Before you go further, you should delete the file.

There is no special function for deleting a JDataStore database file. You can use the `java.io.File.delete()` method or anything else that accomplishes the task. As an aside example, if you always want to create a new JDataStore database file, you write something like this code fragment:

```
// store is DataStore with fileName property set

java.io.File storeFile = new java.io.File( store.getFileName() );
if ( storeFile.exists() ) {
    storeFile.delete();
}

store.create();
```

If the JDataStore database is transactional, it is accompanied by transaction log files, which must also be deleted. For more information on transaction log files, see "Transaction log files" on page 172.

See "Upgrading the JDataStore database" on page 40. JdsExplorer automatically deletes any associated transaction log files.

Storing Java objects

Add the boldfaced statements to the `if` block in the `main()` method:

```
if ( !new java.io.File( store.getFileName() ).exists() ) {
    store.create();
    try {
       store.writeObject( "hello", "Hello, JDataStore! It's "
                          + new java.util.Date() );
   } catch ( java.io.IOException ioe ) {
       ioe.printStackTrace();
   }
} else {
```

The `writeObject()` method attempts to store a Java object as a file stream in the JDataStore database using Java serialization. (Note that you can also store objects in a table.) The object to be stored must implement the `java.io.Serializable` interface. A `java.io.IOException` (more specifically, a `java.io.NotSerializableException`) is thrown if it doesn't. Another reason for the exception would be if the write failed, which would happen, for example, if you ran out of disk space.

The first parameter of `writeObject()` specifies the `storeName`, the case-sensitive name that identifies the object in the JDataStore database. The second parameter is the object to store. In this case, it is a string with a greeting and the current date and time. The `java.lang.String` class implements `java.io.Serializable`, so the string can be stored with `writeObject()`.

Retrieving Java objects

Add the boldfaced statements to the `else` block in the `main()` method:

```
} else {
    store.open();
    try {
       String s = (String) store.readObject( "hello" );
       System.out.println( s );
   } catch ( com.borland.dx.dataset.DataSetException dse ) {
       dse.printStackTrace();
   } catch ( java.lang.ClassNotFoundException cnfe ) {
       cnfe.printStackTrace();
   } catch ( java.io.IOException ioe ) {
       ioe.printStackTrace();
   }
}
```

The `readObject` method attempts to retrieve the named object from the JDataStore database. Like `writeObject()`, it can throw an `IOException` for reasons such as disk failure. It also can't reconstitute the stored object without the object's class. If that class is not in the classpath, `readObject` throws a `java.lang.ClassNotFoundException`.

If the named object can't be found, a `DataStoreException` with the error code `STORE_NOT_FOUND` is thrown. `DataStoreException` is a subclass of `DataSetException`. It's important to catch that exception here, even though there's another `catch` at the bottom of the method, because jumping there would bypass the call to `close` the JDataStore

database connection. (The code is structured in this somewhat awkward way to teach certain principles.)

Because `readObject` returns a `java.lang.Object`, you generally cast the return value to the expected data type. (If the object isn't actually of that expected type, you get a `java.lang.ClassCastException`.) Here, it's more of a formality, because the `System.out.println()` method can take a generic `Object` reference.

You can now run `Hello.java`. The first time it runs, it creates the JDataStore database file and stores the greeting string. When you run it again, the greeting, date, and time display in the console.

See "Advantages of using the JDataStore file system" on page 17 for a list of reasons why to use the JDataStore file system for persistent storage of arbitrary files and objects, rather than using the JDK classes in the `java.io` package.

Using the directory

The `DataStoreConnection.openDirectory()` method returns the contents of the JDataStore database in a searchable structure. Before you begin your work with directories, add the following program, `AddObjects.java`, to the project and run it to add a few more objects to the JDataStore:

```
// AddObjects.java
package dsbasic;

import com.borland.datastore.*;

public class AddObjects {

    public static void main( String[] args ) {
        DataStoreConnection store = new DataStoreConnection();

        int[]                intArray   = { 5, 7, 9 };
        java.util.Date       date       = new java.util.Date();
        java.util.Properties properties = new java.util.Properties();
        properties.setProperty( "a property", "a value" );

        try {
            store.setFileName( "Basic.jds" );
            store.open();
            store.writeObject( "add/create-time", date );
            store.writeObject( "add/values", properties );
            store.writeObject( "add/array of ints", intArray );
        } catch ( com.borland.dx.dataset.DataSetException dse ) {
            dse.printStackTrace();
        } catch ( java.io.IOException ioe ) {
            ioe.printStackTrace();
        } finally {
            try {
                store.close();
            } catch ( com.borland.dx.dataset.DataSetException dse ) {
                dse.printStackTrace();
            }
        }
    }
}
```

The program does things slightly differently than `Hello.java`. First, it uses a `DataStoreConnection` object instead of a `DataStore` to access the JDataStore database

file, but it's used in the same way. You set the `fileName` property, `open()` the connection, use the `writeObject()` method to store objects, and `close()` the connection.

The location of the `close()` method call is another difference. Because you always want to call `close()` no matter what happens in the main body of the method, it's placed after the `catch` blocks inside a `finally` block. This way, the connection always closes, even if there is an unhandled error. The `close()` method is safe to call even if the connection never opened. In that case, `close()` does nothing.

This time, three objects are written to the JDataStore: an array of integers, a `Date` object (not a `Date` object converted into a string), and a hashtable. They are named so that they will be in a directory named `add`. The forward slash (/) is the directory separator character. One of the names contains spaces, which is perfectly valid.

Demonstration class: Dir.java

Add another file to the `Dir.java` project:

```
// Dir.java
package dsbasic;

import com.borland.datastore.*;

public class Dir {

    public static void print( String storeFileName ) {
        DataStoreConnection store = new DataStoreConnection();
        com.borland.dx.dataset.StorageDataSet storeDir;

        try {
            store.setFileName( storeFileName );
            store.open();

            storeDir = store.openDirectory();
            while ( storeDir.inBounds() ) {
                System.out.println( storeDir.getString(
                        DataStore.DIR_STORE_NAME ) );
                storeDir.next();
            }
            store.closeDirectory();
        } catch ( com.borland.dx.dataset.DataSetException dse ) {
            dse.printStackTrace();
        } finally {
            try {
                store.close();
            } catch ( com.borland.dx.dataset.DataSetException dse ) {
                dse.printStackTrace();
            }
        }
    }

    public static void main( String[] args ) {
        if ( args.length > 0 ) {
            print( args[0] );
        }
    }
}
```

This class needs a command-line argument, the name of a JDataStore database file, which is passed to its `print()` method. The `print()` method accesses the JDataStore database file using code similar to that in the previous examples.

Opening a JDataStore directory

Dir.java defines a DataStoreConnection that accesses the JDataStore database. It also declares a StorageDataSet. After opening a connection to the JDataStore database, the program calls the openDirectory() method of the DataStoreConnection to get the contents of the JDataStore's directory. The directory of a JDataStore database is represented by a table.

See also

- "Viewing JDataStore database information" on page 36

The JDataStore directory

The JdsExplorer tree view provides a hierarchical view of the JDataStore directory. This directory can also be opened programmatically with a DataExpress DataSet component to provide a tabular view of all streams stored in the JDataStore file system. The table gives the state (active or deleted), delete time (if deleted), storeName, stream type, id, properties and events, last modification time, stream length, and blob length.

For a discussion of JDataStore directory contents, stream details, and directory sort order, see "JDataStore directory contents" on page 17 for a table that lists the constant, data type, and contents description for each column in the directory table.

Reading a JDataStore directory

You use the DataExpress API to manipulate the JDataStore directory table just as you would manipulate any other table. Use the next() and inBounds() methods to navigate through each entry in the directory. Use the appropriate get*XXX*() method to read the desired information for each stream.

You can't write to the JDataStore directory because it is read-only.

To run Dir.java, set the runtime parameters in the Project Properties dialog box to the JDataStore database file you want to check. In this case, set it to Basic.jds. When it runs, a loop goes through the directory, listing the name of every stream:

```
add/array of ints
add/create-time
add/values
hello
```

You can include a lot more information in the directory listing. The most difficult part is making the formatting decisions for the various bits of information available in all the columns of the JDataStore directory. To display whether the stream is a table or file stream, for example, add the boldfaced statements to the beginning of the loop:

```
while ( storeDir.inBounds() ) {
    short dirVal = storeDir.getShort( DataStore.DIR_TYPE );
    if ( (dirVal & DataStore.TABLE_STREAM) != 0 ) {
        System.out.print( "T" );
    } else if ( (dirVal & DataStore.FILE_STREAM) != 0 ) {
        System.out.print( "F" );
    } else {
        System.out.print( "?" );
    }
    System.out.print( " " );
    System.out.println( storeDir.getString( DataStore.DIR_STORE_NAME ) );
    storeDir.next();
}
```

That addition changes the output to this:

```
F add/array of ints
F add/create-time
F add/values
F hello
```

The output indicates that all the serialized objects are indeed file streams.

Closing the JDataStore directory

When you're not using the JDataStore directory, close it by calling the `DataStoreConnection.closeDirectory()` method. Most JDataStore operations modify the directory in some way. If the directory is open, it must be notified, which slows down your application.

If you try to access the directory `StorageDataSet` when the directory is closed, you get a `DataSetException` with the error code `DATASET_NOT_OPEN`.

Checking for existing streams

Although you could search the JDataStore directory manually, the `DataStoreConnection` provides two methods for checking whether a stream exists without having to open the directory. The `tableExists()` method checks for table streams and the `fileExists()` method checks for file streams. Both methods take a `storeName` parameter and they ignore streams that are deleted. They return `true` if there is an active stream of the corresponding type with that name in the JDataStore database, or `false` otherwise. Remember that stream names are case-sensitive and that you can't have a table stream and a file stream with the same name.

For example, suppose you ran the following code fragment against `Basic.jds` as it is at this point in the tutorial:

```
store.tableExists( "hello" )
```

It returns `false` because although there is a stream named "hello", it's a file stream, not a table stream. The same result occurs with this:

```
store.fileExists( "Hello" )
```

This time the name doesn't match case. Here the name and type match:

```
store.fileExists( "hello" )
```

Now it returns `true`.

Storing arbitrary files

In addition to serializing discrete objects as file streams, you can store and retrieve data streams in a JDataStore database through a `com.borland.datastore.FileStream` object. Although `FileStream` is a subclass of `java.io.InputStream`, it has a method for writing to the stream as well, so the same object can be used for both read and write access. It also provides random access with a `seek()` method. Because `FileStream` is a subclass of `InputStream`, it's easy to use streams stored in the JDataStore database in generic situations that expect an input stream. You'll probably read a stream more often than you write one.

For more information, see "Importing files into JDataStore databases" on page 54.

Demonstration class: ImportFile.java

Suppose you have an application that uses boilerplate documents that are modified for individual customers. A field in the customer table contains their personalized copy, but you also need to store the original somewhere so that you can make fresh copies for new customers. The following utility program, `ImportFile.java`, stores the original as a file stream in the JDataStore database. Add it to the project.

```
// ImportFile.java
package dsbasic;

import com.borland.datastore.*;

public class ImportFile {

    private static final String DATA     = "/data";
    private static final String LAST_MOD = "/modified";

    public static void read( String storeFileName,
                             String fileToImport ) {
        read( storeFileName, fileToImport, fileToImport );
    }

    public static void read( String storeFileName,
                             String fileToImport,
                             String streamName ) {
        DataStoreConnection store = new DataStoreConnection();

        try {
            store.setFileName( storeFileName );
            store.open();

            FileStream fs = store.createFileStream( streamName + DATA );

            byte[] buffer = new byte[ 4 * store.getDataStore().getBlockSize()
                                      * 1024 ];
            java.io.File file = new java.io.File( fileToImport );
            java.io.FileInputStream fis = new java.io.FileInputStream( file );

            int bytesRead;
            while ( (bytesRead = fis.read( buffer )) != -1 ) {
                fs.write( buffer, 0, bytesRead );
            }
            fs.close();
            fis.close();

            store.writeObject( streamName + LAST_MOD,
                               new Long( file.lastModified() ) );
        } catch ( com.borland.dx.dataset.DataSetException dse ) {
            dse.printStackTrace();
        } catch ( java.io.FileNotFoundException fnfe ) {
            fnfe.printStackTrace();
        } catch ( java.io.IOException ioe ) {
            ioe.printStackTrace();
        } finally {
            try {
                store.close();
            } catch ( com.borland.dx.dataset.DataSetException dse ) {
                dse.printStackTrace();
            }
        }
    }
```

```
    public static void main( String[] args ) {
        if ( args.length == 2 ) {
            read( args[0], args[1] );
        } else if ( args.length >= 3 ) {
            read( args[0], args[1], args[2] );
        }
    }
}
```

The program takes as parameters the name of a JDataStore database file, the name of the file to import, and an optional stream name. If you don't specify a file stream name, the file name is used. The `main()` method calls the appropriate form of the `read()` method, because the two-argument `read()` method calls the three-argument `read()` method.

When the file is imported, the date it was last modified is recorded with it. The `/modified` suffix appends to the stream name for this date, while the `/data` suffix appends to the stream name to contain the data from the file. These suffixes are defined as class variables.

The `read()` method then begins by opening a connection to the JDataStore database with a `DataStoreConnection` object.

Creating a file stream

As with most file stream APIs, there are separate methods for creating new file streams and accessing existing file streams. The method for creating a new file stream is `createFileStream()` and its only parameter is the `storeName` of the stream to create.

If there is already a file stream with that name, even if it's actually a serialized object, it will be lost without warning. You might want to check whether such a file stream exists with the `fileExists()` method first, since `ImportFile.java` does not check. If there is a table stream with that name, `createFileStream()` throws a `DataStoreException` with the error code `DATASET_EXISTS`, because you can't have a table stream and a file stream with the same name.

When `createFileStream()` is successful, it returns a `FileStream` object that represents the new, empty file stream.

Referencing the connected JDataStore database

A simple copy operation like this uses a loop to read and write the file in chunks. The question is, how big should those chunks be? There's the obvious problem of making them too small, but making them really large can cause performance problems as well. As a conservative start, you can make the size a small multiple of the database's block size.

The block size of the JDataStore database is stored in the `DataStore` object's `blockSize` property. Whenever you use a `DataStoreConnection` to access a JDataStore database, it automatically creates an instance of `DataStore`. Other `DataStoreConnection` objects in the same process that connect to the same JDataStore database share that `DataStore` object. (Access to a JDataStore database is exclusive to a single process. Multiuser access is provided through a single server process.) The `DataStoreConnection` has a read-only property named `dataStore` that contains a reference to the connected `DataStore` object.

The `FileStream` object writes an array of bytes. The array is declared in this statement:

```
byte[] buffer = new byte[ 4 * store.getDataStore().getBlockSize() * 1024 ];
```

The `getDataStore()` method gets the reference to the `DataStore` object, and from that the `getBlockSize()` method gets the `blockSize` property. The property value is in

kilobytes so it is multiplied by 1024. The resulting block size is multiplied by four, the arbitrarily chosen number of blocks to read in each chunk.

Writing to a file stream

The `FileStream` object's `write()` method takes an array of bytes such as a `java.io.OutputStream`, although the only form of the method is the one that also specifies the starting offset and length.

The `java.io.FileInputStream` object reads from a file into an array of bytes. It returns the number of bytes read, or -1 if the end-of-file is reached. In the loop, the number of bytes read is checked for the end-of-file value. If it's not the end-of-file, the number of bytes read are written, starting with the first byte in the array. For every iteration of the loop except the last, the entire array is filled by reading and writing into the `FileStream`. The last iteration probably won't fill the entire array.

Closing a file stream

Once you're done with a file stream, you should close it. The `FileStream` object uses the `close()` method (as does the `FileInputStream`).

After the file stream is closed, the last-modified date is written using a `java.lang.Long` object to encapsulate the primitive `long` value. You cannot save primitives with serialization.

To test `ImportFile.java`, try importing some source code files into `Basic.jds`.

Opening, seeking, and reading a file stream

Use the `openFileStream()` method to open an existing file stream by name. Like `createFileStream()`, it returns a `FileStream` object at the beginning of the stream. You can then go to any position in the stream with the `seek()` method, write to the stream, and read from it with the `read()` method. `FileStream` also supports `InputStream` marking with the `mark()` and `reset()` methods.

The `PrintFile.java` program demonstrates opening, seeking, and reading. Add it to the project.

```
// PrintFile.java
package dsbasic;

import com.borland.datastore.*;

public class PrintFile {

    private static final String DATA     = "/data";
    private static final String LAST_MOD = "/modified";

    public static void printBackwards( String storeFileName,
                                       String streamName ) {
        DataStoreConnection store = new DataStoreConnection();

        try {
            store.setFileName( storeFileName );
            store.open();

            FileStream fs = store.openFileStream( streamName + DATA );
            int streamPos = fs.available();
```

```
                while ( --streamPos >= 0 ) {
                    fs.seek( streamPos );
                    System.out.print( (char) fs.read() );
                }
                fs.close();

                System.out.println( "Last modified: " + new java.util.Date(
                        ((Long) store.readObject( streamName
                                                + LAST_MOD )).longValue() ) );
            } catch ( com.borland.dx.dataset.DataSetException dse ) {
                dse.printStackTrace();
            } catch ( java.io.IOException ioe ) {
                ioe.printStackTrace();
            } catch ( java.lang.ClassNotFoundException cnfe ) {
                cnfe.printStackTrace();
            } finally {
                try {
                    store.close();
                } catch ( com.borland.dx.dataset.DataSetException dse ) {
                    dse.printStackTrace();
                }
            }
        }

        public static void main( String[] args ) {
            if ( args.length == 2 ) {
                printBackwards( args[0], args[1] );
            }
        }
    }
```

To demonstrate random access with the `seek()` method (and to make things slightly more interesting), this program prints a file stream backwards. It determines the length of the file stream by calling the `FileStream`'s `available()` method and uses it as a file pointer. When reading from the file, the program moves the file pointer forward. The position of the file pointer decrements and is set for each byte read in the loop. There are two forms of the `read()` method. The first reads into a byte array (the same form of the method used by the `FileInputStream` in `ImportFile.java`). The second returns a single byte. Here the single-byte form is used. Each byte is cast into a character to be printed.

Copying streams

The `DataStoreConnection.copyStreams()` method can be used to copy one or more streams from one JDataStore database to another.

Naming and renaming the streams to copy

Forward slashes in stream names are used to simulate a hierarchical directory structure. The `copyStreams()` method is unaware of directory structure. It simply treats names as strings. You must use the forward slash when necessary to impose structure.

The first two parameters, `sourcePrefix` and `sourcePattern`, determine which streams are copied. `sourcePrefix` is used in combination with the `destPrefix` parameter to rename a stream when it is copied; that is, to change the prefix (the beginning) of the `storeName` of the resulting copy of the stream.

If you specify a `sourcePrefix`, the stream name must start with that string. It's usually used to specify the name of a directory ending with a forward slash. The `destPrefix` is

then set to a different directory name also ending with a forward slash. The `sourcePrefix` is stripped from the name, and the `destPrefix` is prepended to the name of the copy. For example, suppose you have the stream named “add/create-time” and you want to create a copy named “tested/create-time”. The effect is to make a copy in a different directory. You would set `sourcePrefix` to “add/” and `destPrefix` to “tested/”.

Although the prefix parameters are usually used for directories, you can rename streams in other ways. For example, you can rename “hello” to “jello” by specifying “h” and “j” for the `sourcePrefix` and `destPrefix` respectively. Or you can change “three/levels/deep” to “not-a-peep” by specifying “three/levels/d” and “not-a-p”. The effect is to move a stream up to the root directory of the JDataStore database. You can also do the reverse by making the `destPrefix` longer (with more directory levels) than the `sourcePrefix`. For example, by leaving the `sourcePrefix` blank, but specifying a `destPrefix` that ends with a forward slash, all the streams from the original JDataStore database file are placed under a directory in the destination JDataStore.

If you’re not renaming the copy of the stream, there’s no reason to use either prefix parameter, so you should set both of them to an empty string or `null`. Note that if you’re making a copy of a stream in the same JDataStore database file, you must rename the copy.

The `sourcePattern` parameter is matched against everything after the `sourcePrefix`, using the standard wildcard characters “*” (for zero or more characters) and “?” (for a single character). If the `sourcePrefix` is empty, that means that the pattern is matched against the entire string. If you want to copy all the streams in a directory, you can put the directory name in the `sourcePattern`, followed by a forward slash, and leave the `sourcePrefix` empty. For example, if you want to copy everything in the “add” directory, you want to copy everything that starts with “add/”, so the `sourcePattern` would be “add/*”. That includes everything in subdirectories, because the `sourcePattern` matches the remainder of the string. (There is no direct way to prevent the copying of streams in subdirectories.)

The `sourcePattern` is matched against names of active streams only. `copyStreams()` doesn’t copy deleted streams.

Demonstration class: Dup.java

You can use the following program, `Dup.java`, to make a backup copy of a JDataStore database file or upgrade an older file to the current format:

```
// Dup.java
package dsbasic;

import com.borland.datastore.*;

public class Dup {

    public static void copy( String sourceFile, String destFile ) {
        DataStoreConnection store1 = new DataStoreConnection();
        DataStore           store2 = new DataStore();

        try {
            store1.setFileName( sourceFile );
            store2.setFileName( destFile );
            if ( !new java.io.File( store2.getFileName() ).exists() ) {
                store2.create();
            } else {
                store2.open();
            }
            store1.open();
```

```
            store1.copyStreams( "",     // From root directory
                                "*",    // Every stream
                                store2,
                                "",     // To root directory
                                DataStore.COPY_IGNORE_ERRORS,
                                System.out );
        } catch ( com.borland.dx.dataset.DataSetException dse ) {
            dse.printStackTrace();
        } finally {
            try {
                store1.close();
                store2.close();
            } catch ( com.borland.dx.dataset.DataSetException dse ) {
                dse.printStackTrace();
            }
        }
    }

    public static void main( String[] args ) {
        if ( args.length == 2 ) {
            copy( args[0], args[1] );
        }
    }
}
```

This program copies the contents of one database into another. It uses a `DataStoreConnection` object to open the source JDataStore and copies the contents to a `DataStore` object so that the JDataStore database can be created if it doesn't already exist.

For the `copyStreams()` method, the `sourcePrefix` and `destPrefix` are empty strings, and the `sourcePattern` is just "*", which copies everything without renaming anything. The program ignores unrecoverable errors and displays status messages in the console.

You can use this program to combine the contents of multiple JDataStore database files into a single file, as long as the stream names are different, since `COPY_OVERWRITE` is not specified as an option.

Deleting and undeleting streams

Deleting streams is easy and certain. Undeleting them might not always work and requires a bit more effort. Streams are deleted by name. Understanding what happens when you delete or try to undelete a file stream, whether it's an arbitrary file or serialized object, is simpler because there's only one stream with that name. Table streams often have additional internal support streams with the same name, as discussed in "Stream details" on page 18, so undeleting them is a little more complicated.

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

To delete a stream, use `DataStoreConnection.deleteStream()`, which takes the name of the stream to delete. For a file stream, the individual stream is deleted. For a table stream, the main stream and all its support streams are deleted.

For more information on deleting and undeleting streams, see "Deleting streams" on page 19 "How JDataStore reuses blocks" on page 19 and "Undeleting streams" on page 20.

Demonstration class: DeleteTest.java

The following program, `DeleteTest.java`, demonstrates both deletion and undeletion.

```
// DeleteTest.java
package dsbasic;

import com.borland.datastore.*;

public class DeleteTest {

    public static void main( String[] args ) {
        DataStoreConnection store = new DataStoreConnection();
        com.borland.dx.dataset.StorageDataSet storeDir;
        com.borland.dx.dataset.DataRow        locateRow, dirEntry;
        String storeFileName = "Basic.jds";
        String fileToDelete  = "add/create-time";

        try {
            store.setFileName( storeFileName );
            store.open();

            storeDir  = store.openDirectory();
            locateRow = new com.borland.dx.dataset.DataRow( storeDir,
                                new String[] { DataStore.DIR_STATE,
                                               DataStore.DIR_STORE_NAME } );
            locateRow.setShort(  DataStore.DIR_STATE, DataStore.ACTIVE_STATE );
            locateRow.setString( DataStore.DIR_STORE_NAME, fileToDelete );

            if ( storeDir.locate( locateRow,
                                  com.borland.dx.dataset.Locate.FIRST ) ) {
                System.out.println( "Deleting " + fileToDelete );
                dirEntry = new com.borland.dx.dataset.DataRow( storeDir );
                storeDir.copyTo( dirEntry );
                store.closeDirectory();
                System.out.println( "Before delete, fileExists: "
                        + store.fileExists( fileToDelete ) );

                store.deleteStream( fileToDelete );
                System.out.println( "After delete, fileExists: "
                        + store.fileExists( fileToDelete ) );

                store.undeleteStream( dirEntry );
                System.out.println( "After undelete, fileExists: "
                        + store.fileExists( fileToDelete ) );
            } else {
                System.out.println( fileToDelete
                                    + " not found or already deleted" );
                store.closeDirectory();
            }
        } catch ( com.borland.dx.dataset.DataSetException dse ) {
            dse.printStackTrace();
        } finally {
            try {
                store.close();
            } catch ( com.borland.dx.dataset.DataSetException dse ) {
                dse.printStackTrace();
            }
        }
    }

}
```

In this program, the name of the JDataStore database file and the stream to be deleted are hard-coded, which you would seldom do. The stream is “add/create-time”, which was added to `Basic.jds` in the `AddObjects.java` demonstration program. You know that it’s a file stream rather than a table stream because the `fileExists()` method is used to check whether the deletion and undeletion worked.

Locating directory entries

The program begins by opening a connection to the JDataStore database and opening its directory. Next, it locates the directory entry for the stream that is about to be deleted.

Typically, you would locate the directory entry for the stream after it has been deleted and use the directory dataset to undelete the stream. It’s done differently here to demonstrate individual directory rows, which are explained shortly.

To locate the row, a new `com.borland.dx.dataset.DataRow` is instantiated from the directory dataset, specifying the two columns that are used in the search: State and StoreName. The program then attempts to locate the directory entry for the specified stream, which must be active. Finding the row not only positions the directory at the desired entry, but it also indicates that the stream exists and is active so that the program can proceed to the next step.

Using individual directory rows

When you pass a directory dataset to a method like `undeleteStream()`, the current row is used. But because of the way the JDataStore directory is sorted (as explained in “Directory sort order” on page 18), when a stream is deleted, its directory entry generally moves to its new position at the bottom of the directory as the most recently deleted stream. The current row is then referencing something else (probably the next stream alphabetically). To undelete the same stream, you could either attempt to relocate the directory entry for the now-deleted stream, or you would copy the directory data for the stream into a separate directory row before you delete.

Using an individual directory row has a few advantages. Unlike the live JDataStore directory dataset, an individual row is a static copy. It’s smaller. After making the copy, you can close the directory dataset to make operations faster. (For this simple demonstration, the overhead for creating the individual row probably outweighs any performance benefit.) You can make static copies of as many directory entries as you want, and manage them any way you want to.

To create the individual directory row, another `DataRow` is instantiated from the directory dataset (so that it has the same structure), and the `copyTo()` method copies the data from the current row. And just to prove that it really works, the JDataStore directory is closed.

The file stream is then deleted by name using the plain name string defined at the beginning of the method. Finally, the stream is undeleted using the individual directory entry.

Packing JDataStore files

The only way to shrink a JDataStore database file programmatically, that is, to remove unused blocks and directory entries for deleted streams, is to copy the streams to a new JDataStore database file using `copyStreams()`. Only active streams are copied, which results in a packed version of the file.

For more information, see “Packing a JDataStore database” on page 40.

Chapter 16

Using JDBC for data access

You can access JDataStore tables with JDataStore's Type 4 direct all-Java JDBC driver, `com.borland.datastore.jdbc.DataStoreDriver`.

For extensive JDBC documentation, see "JDBC™ API Documentation" on the `java.sun.com` website.

You can use this driver for both local and remote access. Remote access requires a JDataStore Server, which is also used for multiuser access. For details on remote access and multiuser issues, see Chapter 3, "JDBC quickstart."

This is the local connection URL:

```
jdbc:borland:dslocal:<filename>
```

As with any JDBC driver, you can access tables with `QueryDataSet` and `ProcedureDataSet` using the JDBC API or an added-value API such as DataExpress.

Creating a basic JDBC application using JDataStore

Now that you've learned about creating and manipulating file streams in a JDataStore database (Chapter 15, "The JDataStore file system"), it's time to learn the basics of creating a JDBC application using JDataStore. (For more detailed information about creating DataExpress applications using JDataStore, see Chapter 14, "Using DataExpress with JDataStore.".)

To create (or open) a transactional JDataStore database, you must provide a user name when obtaining a JDBC connection. If no users have been registered, then you can specify any user name you want. If user authentication has not been enabled for the database by registering one or more users, a user name is still needed when a JDBC connection is obtained so that the connection can be identified when database lock errors occur.

The next step is to write some code that connects to the `DataStore`. Type the following code into the new file:

```
//HelloJDBC.java

import java.sql.*;

public class HelloJDBC {

  public HelloJDBC() {
  }

  public static void main(String args[]) {

    // Both the remote and local JDatastore drivers use the same
    // driver string:
    String DRIVER =   "com.borland.datastore.jdbc.DataStoreDriver";
    // Use this string for the local driver:
    String URL    =   "jdbc:borland:dslocal:";
    // Use this string for the remote driver (and start JDataStore Server):
    //  String URL =  "jdbc:borland:dsremote://localhost/";
    String FILE   =   "BasicTX.jds";
    String PROPS  =   ";create=true";

    boolean c_open=false;
    Connection con = null;
    try {
       Class.forName(DRIVER);
       con = DriverManager.getConnection(URL + FILE + PROPS, "user", "");
       c_open = true;
    }
    catch(Exception e) {
       System.out.println(e);
    }
    // This way the connection will be closed even when exceptions are thrown
    // earlier.  This is important, because you may have trouble reopening
    // a JDataStore database file after leaving a connection to it open.
    try {
      if(c_open)
        con.close();
    }
    catch(Exception e3) {
      System.out.println(e3.toString());
    }
  }
}
```

Note the boldface lines of code in this program. First, the driver string for the JDataStore JDBC driver is specified. This string is the same for both the local and remote JDBC drivers. Next, the URL string for connecting to a local JDataStore is shown. For your information, the code also includes the remote string in a comment. The last two boldface lines are common to many JDBC applications, and they're where we actually connect to the JDataStore database.

Notice that the code uses an extended JDBC property (`create=true`) that requests that a database be created if it does not already exist.

Once you've connected to the JDataStore database, you'll probably want to add and manipulate some data. We'll show you how to do that next. We won't spend a lot of time on it here, just enough to let you know that you have connected to the JDataStore

database, and can add, manipulate, print, and delete data. Add the following boldfaced lines to the code as shown:

```
package dsbasic;

import java.sql.*;

public class HelloJDBC {

  public HelloJDBC() {
  }

  public static String formatResultSet(ResultSet rs) {
  // This method formats the result set for printing.

    try {
      ResultSetMetaData rsmd = rs.getMetaData();
      int numberOfColumns = rsmd.getColumnCount();
      StringBuffer ret = new StringBuffer(500);

      for (int i = 1; i <= numberOfColumns; i++) {
        String columnName = rsmd.getColumnName(i);
        ret.append(columnName + "," );
      }
      ret.append("\n");
      while (rs.next()) {
        for (int i = 1; i <= numberOfColumns; i++)
          ret.append(rs.getString(i) + "," );
        ret.append("\n");
      }
      return(ret.toString());
    }
    catch(Exception e) {
      return e.toString();
    }
  }

  static void main(String args[]) {
    // Both the remote and local JDatastore drivers use the
    // same driver string:
    String DRIVER =   "com.borland.datastore.jdbc.DataStoreDriver";

    // Use this string for the local driver:
    String URL    =   "jdbc:borland:dslocal:";

    // Use this string for the remote driver (and start JDataStore Server):
    //  String URL =  "jdbc:borland:dsremote://localhost/";

    String FILE   =   "BasicTX.jds";
    String PROPS  =   ";create=true";

    boolean s_open=false, c_open=false;
    Statement stmt = null;
    Connection con = null;

    try {
      Class.forName(DRIVER);
      con = DriverManager.getConnection(URL + FILE, + PROPS, "user", "");
      c_open = true;
           stmt = con.createStatement();
      s_open = true;
```

```
      // The following line creates a table in the JDataStore database.
      stmt.executeUpdate("create table HelloJDBC" +
        "(COLOR varchar(15), " +
        " NUMBER int, " +
        " PRICE float)");

      // Values are inserted into the table with
      // the next three statements.
      stmt.executeUpdate("insert into HelloJDBC values('Red', 1, 7.99)");
      stmt.executeUpdate("insert into HelloJDBC values('Blue', 2, 8.99)");
      stmt.executeUpdate("insert into HelloJDBC values('Green', 3, 9.99)");

      // Now we query the table
      ResultSet rs = stmt.executeQuery("select * from HelloJDBC");

      // Call to formatResultSet() to format the
      // printed output.
      System.out.println(formatResultSet(rs));

      // The next line deletes the table.
      stmt.executeUpdate("drop table HelloJDBC");
    }
    catch(Exception e) {
      System.out.println(e);
    }

    try {
      // Attempt to clean up by calling the
      // java.sql.Statement.close() method.
      if(s_open)
        stmt.close();
    }
    catch(Exception e2){
      System.out.println(e2.toString());
    }

    // This way the connection will be closed even when exceptions are thrown
    // earlier.  This is important, because you may have trouble reopening
    // a JDataStore database file after leaving a connection to it open.
    try {
      if(c_open)
        con.close();
    }
    catch(Exception e3) {
      System.out.println(e3.toString());
    }
  }
}
```

In the preceding example, the code added to the `main()` method creates a table and inserts rows in the table. Then it calls the `formatResultSet()` method and prints the results. Next, it deletes the table from the JDataStore database. Finally, it attempts to clean up by calling the `close()` method of the `java.sql.Statement` object.

Demonstration class: JdbcTable.java

The following program, `JdbcTable.java`, is functionally identical to its DataExpress twin, `DxTable.java`. It uses the JDBC API.

```
// JdbcTable.java
package dsbasic;

import java.sql.*;

public class JdbcTable {

    static final String DRIVER = "com.borland.datastore.jdbc.DataStoreDriver";
    static final String URL    = "jdbc:borland:dslocal:";

    Connection        con;
    Statement         stmt;
    DatabaseMetaData  dmd;
    ResultSet         rs;
    PreparedStatement appendPStmt, getIdPStmt;

    public JdbcTable() {
        try {
            Class.forName( DRIVER );
            con  = DriverManager.getConnection( URL + "Tx.jds", "Chuck", "" );
            stmt = con.createStatement();
            dmd  = con.getMetaData();
            rs   = dmd.getTables( null, null, "Accounts", null );
            if ( !rs.next() ) {
                createTable();
            }
            appendPStmt = con.prepareStatement("INSERT INTO
                        \"Accounts\" VALUES"
                    + "(?, ?, CURRENT_TIMESTAMP, NULL)" );
            getIdPStmt = con.prepareStatement(
                    "SELECT MAX(ID)FROM \"Accounts\"");
        } catch ( SQLException sqle ) {
            sqle.printStackTrace();
        } catch ( ClassNotFoundException cnfe ) {
            cnfe.printStackTrace();
        }
    }

    public void createTable() throws SQLException {
        stmt.executeUpdate( "CREATE TABLE \"Accounts\" ("
                + "ID INTEGER,"
                + "\"Name\" VARCHAR,"
                + "\"Update\" TIMESTAMP,"
                + "\"Text\" BINARY)" );
    }
```

```
    public void appendRow( String name ) throws SQLException {
        int newID;
        rs = getIdPStmt.executeQuery();
        if ( rs.next() ) {
            newID = rs.getInt( 1 ) + 1;
        } else {
            newID = 1;
        }
        appendPStmt.setInt( 1, newID );
        appendPStmt.setString( 2, name );
        appendPStmt.executeUpdate();
    }

    public void demo() {
        try {
            appendRow( "Rabbit season" );
            appendRow( "Duck season" );

            rs = stmt.executeQuery( "SELECT * FROM \"Accounts\"" );
            while ( rs.next() ) {
                System.out.println( rs.getInt( "ID" ) + ": "
                        + rs.getString( "Name" ) + ", "
                        + rs.getTimestamp( "Update" ) );
            }
            stmt.close();
            con.close();
        } catch ( SQLException sqle ) {
            sqle.printStackTrace();
        }
    }

    public static void main( String[] args ) {
        new JdbcTable().demo();
    }
}
```

This JDBC application uses two prepared statements: one to append rows, and the other to get the ID value for the last appended row. Initialize these prepared statements before calling the `appendRow()` method. A good place to do this is in the class constructor. Because the constructor is used, the organization of the code is a little different than in `DxTable.java`.

The first thing that happens in the class constructor is the loading of the JDataStore JDBC driver using `Class.forName`. Both the driver name and the beginning of the connection URL are defined as class variables for convenience. A `Connection` to `Tx.jds` is created, and from that, a generic `Statement`.

The next step is to determine whether the table exists. You can do this using `DatabaseMetaData.getTables()`. The code asks for a list of tables named “Accounts.” If that list is empty, that means there is no such table and you must create it by calling the `createTable()` method. The `createTable()` method uses a SQL `CREATE TABLE` statement. Note that the SQL parser usually converts identifiers to uppercase. To keep the proper casing used by `DxTable.java`, enclose the identifiers in quotes in this and other SQL statements. Finally, the two prepared statements are created.

The `demo()` method calls `appendRow()` to add a couple of test rows. As in `DxTable.java`, the last/largest ID value is retrieved and incremented for the new row. But instead of using a sort order and going to the last row, the JDBC approach uses an SQL SELECT statement that fetches the maximum value. As a result, the empty table condition, when there is no last value, must be handled specifically.

Finally, the contents of the table are displayed using a SQL `SELECT` statement to fetch the rows and a loop that's very similar to the one in `DxTable.java`. The statement and connection are closed as required by JDBC.

You can run this program interchangeably with `DxTable.java`. Both of them add two more test rows to the same table.

Controlling transactions through JDBC

Each JDBC connection actually uses its own internal instance of `DataStoreConnection` for the connection. That's how the JDataStore JDBC driver is implemented. But this internal object is not accessible, so you must use the JDBC API.

For control over transactions, disable autocommit mode by calling `Connection.setAutoCommit(false)`. You can then call `commit()` and `rollback()` on the `Connection` object.

Index

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y