



Programmer's Guide



VERSION 3.5

Borland®
JDataStore™

Inprise Corporation
100 Enterprise Way, Scotts Valley, CA 95066-3249

Refer to the file DEPLOY.TXT located in the root directory of your JBuilder or JDataStore product for a complete list of files that you can distribute in accordance with the JBuilder or JDataStore License Statement and Limited Warranty.

Inprise may have patents and/or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents.

COPYRIGHT © 1997, 1999 Inprise Corporation. All rights reserved. All Inprise and Borland brands and product names are trademarks or registered trademarks of Inprise Corporation. Other product names are trademarks or registered trademarks of their respective holders.

Printed in the U.S.A.

JBE0030WW21008 2E3R0300

0001020304-9 8 7 6 5 4 3 2 1

PDF

Contents

Chapter 1

Introduction 1-1

When to use JDataStore	1-1
JDataStore and DataStore	1-2
What you should know	1-2
What's in this book	1-2
Starting with the DataStore Explorer	1-3
Deploying DataStore application components	1-3
Contacting Borland developer support	1-3

Chapter 2

DataStore fundamentals 2-1

DataStore primer	2-1
Serializing objects	2-2
Demonstration class: Hello.java	2-2
Creating a DataStore file	2-3
Opening and closing connections	2-3
Handling basic DataStore exceptions	2-4
Deleting DataStore files	2-4
Storing Java objects	2-4
Retrieving Java objects	2-5
Advantages for persistent object storage	2-6
Using the directory	2-6
Demonstration class: Dir.java	2-7
Opening a DataStore directory	2-8
DataStore directory contents	2-9
Stream details	2-9
Directory sort order	2-10
Reading a DataStore directory	2-10
Closing the DataStore directory	2-11
Checking for existing streams	2-11
Storing arbitrary files	2-12
Demonstration class: ImportFile.java	2-12
Creating a file stream	2-13
Referencing the connected DataStore	2-14
Writing to a file stream	2-14
Closing a file stream	2-15
Opening, seeking, and reading a file stream	2-15
Copying streams	2-16
copyStreams parameters	2-17
Naming and renaming the streams to copy	2-17
Demonstration class: Dup.java	2-18

Deleting and undeleting streams	2-20
Deleting streams	2-20
How DataStore reuses deleted blocks	2-20
Undeleting DataStore streams	2-21
Demonstration class: DeleteTest.java	2-21
Locating directory entries	2-22
Using individual directory rows	2-23
Packing DataStore files	2-23

Chapter 3

DataStore as an embedded database 3-1

Using DataExpress for data access	3-1
Demonstration class: DxTable.java	3-2
Connecting to a DataStore with <i>StorageDataSet</i>	3-2
Creating DataStore tables with DataExpress	3-3
Using DataStore tables with DataExpress	3-4
Transactional DataStores	3-6
Enabling transaction support	3-6
Creating new transactional DataStores	3-6
Adding transaction support to existing DataStores	3-7
Opening a transactional DataStore	3-7
Changing transaction settings	3-8
Transaction log files	3-8
Moving transaction log files	3-9
Bypassing transaction support	3-10
Removing transaction support	3-10
Deleting transactional DataStores	3-11
Controlling DataStore transactions	3-11
Understanding the transaction architecture	3-11
Committing and rolling back transactions	3-12
Tutorial: Controlling transactions via DataExpress	3-12
Step 1: Create a transactional DataStore with test data	3-12
Step 2: Create a data module	3-13
Step 3: Create a GUI for the DataStore table	3-14
Step 4: Add direct transaction control	3-16
Step 5: Add control over auto-commit	3-17
Using JDBC for data access	3-17
Demonstration class: JdbcTable.java	3-18
Controlling transactions through JDBC	3-20

Chapter 4

Multi-user and remote access to DataStores 4-1

Using the JDBC driver for remote access	4-1
Running the DataStore JDBC server	4-2
Reconfiguring the server	4-3
Deploying the DataStore JDBC server	4-3
Packaging the server	4-3
Starting the server	4-3
Creating custom JDBC servers	4-4
Multi-user transaction issues	4-4
Transaction isolation level	4-5
Avoiding blocks and deadlocks	4-5
Conserving write transactions	4-5
Using read-only transactions	4-5
Detecting blocks and deadlocks	4-6

Chapter 5

Persisting data in a DataStore 5-1

Using <i>DataStore</i> instead of <i>MemoryStore</i>	5-1
Using a DataStore with <i>StorageDataSets</i>	5-2
Tutorial: Offline editing with DataStore	5-2
Understanding how DataStore manages offline data	5-5
Restructuring DataStore <i>StorageDataSets</i> . . .	5-6
Data type coercions	5-7
Persistent column editing	5-7
Understanding structure changes	5-8

Chapter 6

Using the DataStore Explorer 6-1

Launching the DataStore Explorer	6-1
Starting the DataStore Explorer from the command line	6-2
Basic DataStore operations	6-2
Creating a new DataStore file	6-2
Opening an existing DataStore file	6-3
Setting options for opening DataStore files	6-3
Opening a DataStore file that was not closed properly	6-4
Viewing DataStore file information	6-4
Viewing stream contents	6-5
Renaming streams	6-7
Deleting streams	6-8
Undeleting streams	6-8
Copying DataStore streams	6-8
Verifying the DataStore	6-9

Making the DataStore transactional	6-9
Modifying transaction settings	6-10
Removing transaction support	6-10
Closing DataStore files	6-11
DataStore Explorer as a query console	6-11
Using DataStore Explorer to manage queries	6-11
DataStore Explorer limitations	6-12
Creating and maintaining queries and connections	6-12
Fetching and editing data	6-14
Saving changes and refreshing data	6-15
Importing tables and files	6-16
Importing text files as tables	6-16
Importing files	6-17
Executing SQL	6-17
DataStore file operations	6-18
Packing the DataStore file	6-18
Upgrading the DataStore file	6-18
Deleting the DataStore file	6-18

Chapter 7

Optimizing DataStore applications 7-1

General usage recommendations	7-1
Closing the DataStore	7-1
Controlling how often cache blocks are written to disk	7-2
Tuning memory	7-2
Miscellaneous performance tips	7-3
DataStore companion components	7-3
Using data modules	7-3
Optimizing transactional applications	7-4
Using read-only transactions	7-4
Using soft commit mode	7-5
Transaction log files	7-5
Disabling status logging	7-5
Pruning deployed resources	7-5

Chapter 8

Troubleshooting 8-1

Debugging DataStore applications	8-1
Verifying DataStore contents	8-1
Problems locating and ordering data	8-2
Saving log files	8-2

Appendix A

Specifications A-1

DataStore file capacity	A-1
DataStore stream names	A-2

Appendix B

Changes from previous versions **B-1**

API changes B-1

Deployment changes B-1

Changes to the DataStore Explorer B-2

Appendix C

SQL reference **C-1**

Lists in syntax notation C-1

Data types. C-1

Literals C-3

Keywords. C-4

Identifiers C-5

Expressions C-5

Predicates C-7

 BETWEEN. C-7

 IS C-7

 LIKE C-7

Functions C-8

 CHAR_LENGTH and

 CHARACTER_LENGTH C-8

CURRENT_DATE, CURRENT_TIME,

 and CURRENT_TIMESTAMP C-8

EXTRACT C-9

LOWER and UPPER C-9

POSITION C-9

SUBSTRING C-10

TRIM C-10

Statements C-11

 CREATE TABLE. C-11

 ALTER TABLE. C-12

 CREATE INDEX. C-12

 SELECT. C-13

 GROUP BY and HAVING C-13

 ORDER BY C-15

 INSERT C-15

 UPDATE C-16

 DELETE C-17

 DROP INDEX C-17

 DROP TABLE C-17

Index **I-1**

Tables

2.1	DataStore directory table columns.	2-9	6.1	Copy Streams options	6-9
2.2	DataStoreConnection.copyStreams method parameters	2-17	C.1	Data types supported by DataStore.	C-2
4.1	DataStore JDBC server startup options	4-4	C.2	DataStore SQL literal values.	C-3
			C.3	Keywords for DataStore SQL engine.	C-4

Figures

3.1	The complete AccountsFrame	3-12	6.8	DataStore Explorer displaying image stored in DataStore	6-7
4.1	DataStore JDBC server	4-2	6.9	Copy Streams dialog box	6-9
6.1	DataStore Explorer after launch	6-2	6.10	TxManager Properties dialog box.	6-10
6.2	New DataStore dialog box	6-3	6.11	First page of Import Tables dialog box	6-13
6.3	DataStore options dialog box.	6-4	6.12	New JDBC Connection dialog box	6-13
6.4	DataStore marked open dialog box	6-4	6.13	Second page of Import Tables dialog box	6-14
6.5	DataStore Explorer displaying DataStore file information	6-5	6.14	Entries in the SYS/Queries table	6-16
6.6	DataStore Explorer displaying table stored in DataStore	6-6	6.15	Entries in the SYS/Queries table	6-17
6.7	DataStore Explorer displaying text file stored in DataStore.	6-7			

Introduction

JDataStore is a high-performance, small-footprint, 100% Pure Java™ multifaceted data storage solution. It is:

- A zero-administration embedded relational database, with both JDBC and DataExpress interfaces, that supports transactional multi-user access with crash recovery.
- An object store, for storing serialized objects, tables, and other file streams.
- A JavaBean component, that can be manipulated with visual bean builder tools like JBuilder.

When to use JDataStore

With a *JDataStore*, you can:

- Embed SQL-92-compliant database functionality directly into your application, without having to rely on an external database engine. Databases can be accessed through the *JDataStore* JDBC driver, or through the DataExpress API. *JDataStore* supports most JDBC data types, including Java Object.
- Serialize all your application's objects and file streams into a single physical file for convenience and portability.
- Enable mobile and off-line applications. Using DataExpress datasets, *JDataStore* asynchronously replicates and caches data from an arbitrary data source (for example, a database server, a CORBA application server, SAP, BAAN, and so forth), allows access and updates, and resolves changes back into the data source.
- Increase the performance of online applications with large datasets by using a *DataStore* instead of the default *MemoryStore*.

You can do these kinds of things at the same time, with a single DataStore file!

JDataStore and DataStore

JDataStore is the name of the product. DataStore is the name of the packages, classes, tools, and file format.

What you should know

The *JDataStore Programmer's Guide* assumes a basic working knowledge of:

- Java programming.
- The JBuilder UI (how to create, manage, and run projects; how to use the design tools).
- Basic DataExpress.
- Basic JDBC.

What's in this book

The *JDataStore Programmer's Guide* is a general guide and tutorial, followed by reference material. It comprises the following chapters and appendixes:

- Chapter 2, "DataStore fundamentals," describes the basic structure of a DataStore file, and uses file streams to demonstrate various administrative tasks.
- Chapter 3, "DataStore as an embedded database," explains how to make a DataStore transactional and use it as an embedded database with a sample GUI application.
- Chapter 4, "Multi-user and remote access to DataStores," introduces the DataStore JDBC server used for remote access, and discusses multi-user transactional issues.
- Chapter 5, "Persisting data in a DataStore," explains how to use the DataStore as a persistent data cache for off-line computing.
- Chapter 6, "Using the DataStore Explorer," describes the DataStore Explorer.
- Chapter 7, "Optimizing DataStore applications," contains a variety of tips on optimizing the performance, reliability, and size of DataStore applications.

- Chapter 8, “Troubleshooting,” discusses debugging DataStore applications and fixing common problems.
- Appendix A, “Specifications,” lists the specifications for the DataStore file format.
- Appendix B, “Changes from previous versions,” highlights the major changes since the previous version.
- Appendix C, “SQL reference,” is a reference guide for the SQL-92 dialect supported by the DataStore JDBC driver.

Starting with the DataStore Explorer

DataStore Explorer (DSX) is an all-Java visual tool that helps you manage your DataStores. It is covered in detail in Chapter 6, “Using the DataStore Explorer.” Used in conjunction with sample DataStore files that ship with JBuilder, you can get an early idea of what a DataStore can do.

The DataStore Explorer provides visual tools for performing many maintenance tasks. The Programmer’s Guide chooses to explain the basics using the bare-bones DataStore API before explaining and demonstrating more complex topics. But there’s nothing to prevent you from looking at that chapter first.

Throughout the *JDataStore Programmer’s Guide*, you will see the following notation in the margin, which indicates some task that can be performed visually with the DataStore Explorer:

DSX This notation is accompanied by a reference to that task in the DataStore Explorer.

Deploying DataStore application components

Information on deploying the DataStore JDBC server for remote access is described in “Deploying the DataStore JDBC server” on page 4-3. Tips for reducing the deployed size of DataStore client applications can be found in “Pruning deployed resources” on page 7-5.

Reminder DataStore is provided with a license for development only. For deployment, additional licenses must be purchased. Contact Borland Customer Service for more information.

Contacting Borland developer support

Borland offers a variety of support options. These include free services on the Internet, where you can search our extensive information base and connect with other users of Borland and Inprise products. In addition, you

can choose from several categories of support, ranging from support on installation of the borland.com product to fee-based consultant-level support and detailed assistance.

For more information about Borland's developer support services, please see our Web site at <http://www.borland.com/devsupport>, call Borland Assist at (800) 523-7070, or contact our Sales Department at (831) 431-1064.

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.

For information about year 2000 issues and our products, see the following URL: <http://www.inprise.com/about/y2000/>.

DataStore fundamentals

This chapter contains a number of simple tutorials which demonstrate basic DataStore concepts.

DataStore primer

A DataStore file can contain two basic types of data streams: *table* streams and *file* streams:

- Table streams include complete database tables, created by the JDBC or DataExpress APIs; and cached table data from an external data source like a database server, created when the DataStore is used as the *store* for a *StorageDataSet*.
- There are two different kinds of file streams:
 - Arbitrary files, created with *DataStoreConnection.createFileStream()*. You can write to, seek in, and read from these streams.
 - Serialized Java objects are stored as file streams.

One key to understanding the possible uses of DataStore in applications is that all three kinds of streams may be stored in the same DataStore file.

Each stream is identified by a case-sensitive name, referred to as *storeName* in the API, which can be up to 192 bytes long. The name is stored along with other information about the stream in the DataStore's internal directory. The forward slash ("/") is used as a directory separator in the name, to provide a hierarchical directory organization. This structure is used by the DataStore Explorer to display the contents of a DataStore in a tree.

This chapter covers DataStore fundamentals, including the directory, using file streams. For table streams, see Chapter 3, "DataStore as an embedded database," and Chapter 5, "Persisting data in a DataStore."

Serializing objects

One of the DataStore's features is the fact that it is a component that you can program visually. But visual programming sometimes presents too many choices at once, making things seem more complicated than they really are. A set of simple exercises can better demonstrate the DataStore's basic essence.

The classic first exercise for a new language is how to display "Hello, World!" The spirit of that tradition will be carried on here. (You will be spared from performing the classic second exercise, a Fahrenheit to Celsius converter.)

First, create a new project for the *dsbasic* package, which will be used throughout this chapter.

Important Add the DataStore 3.1 library to the project so that you can access the DataStore classes.

Demonstration class: Hello.java

Add a new file to the project, *Hello.java*, and start with the following:

```
// 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 DataStore classes. (The rest of the public DataStore 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 DataStore JDBC server. These classes will be covered in Chapter 3, "DataStore as an embedded database" and Chapter 4, "Multi-user and remote access

to DataStores.”) *DataStore* can also be accessed by DataExpress components (packages under *com.borland.dx*), but those classes will be referenced explicitly so that you can see where each class comes from.

Creating a DataStore file

In the *main* method, a new *DataStore* object is created. This object represents a physical *DataStore* file, and 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 does not end with the default extension, it would be appended to the file name when the property is set.

You cannot *create* the *DataStore* if a file with that name already exists. Because the *fileName* property may have been altered when it was set, it’s safer to get the property value for the actual file name to search for.

If the file does not exist, the *create* method will create 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 will throw an exception. Otherwise, you will have an open connection to a new *DataStore* file.

DSX See “Creating a new *DataStore* file” on page 6-3. When creating the file, you can also specify options like block size and whether the *DataStore* will be transactional.

Opening and closing connections

If the file does exist, then a connection will be opened through the *open* method. The *open* method is actually a method of the *DataStore* class’ superclass, *DataStoreConnection*, which (in being what its name implies) contains properties and methods for accessing the contents of a *DataStore*. (The *fileName* property is also a property of *DataStoreConnection*, which means that you can and often do access a *DataStore* without a *DataStore* object, as you will 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 *DataStore* file, but *DataStoreConnection* cannot.)

But the excitement is short-lived. Immediately after opening a connection to the *DataStore* file, creating the file in the process if necessary, that connection is closed with the *close* method (this is also inherited from *DataStoreConnection*). Because there was only that one built-in connection, now that all the connections to the *DataStore* are closed, the *DataStore* file itself will shutdown.

It is vital that you 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 will keep running and prevent your application from terminating properly; you must close those connections or your application will hang on exit.

Handling basic DataStore exceptions

Most of the methods in the DataStore classes may 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 DataStore file if one already exists. You can’t open a connection if the named file isn’t really a DataStore file. You might get an IO exception when writing data when closing a connection.

Subsequently, almost all DataStore code is inside a **try** block. In this case, if an exception is thrown, a stack trace is printed.

Deleting DataStore files

If you run the application now, it won’t do much; just create the file *Basic.jds*. If you then run it a second time, it will do even less; just open and close a connection. Before proceeding, you should delete the file.

There is no special function for deleting a DataStore file. You can use the *java.io.File.delete* method or anything else. As an aside example, if you always want to create a new DataStore file, you could do 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 DataStore file 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 3-8.

DSX See “Deleting the DataStore file” on page 6-18. The DataStore Explorer will automatically delete any associated transaction log files.

Storing Java objects

Add the highlighted statements to the **if** block in the *main* method:

```
if ( !new java.io.File( store.getFileName() ).exists() ) {
    store.create();
    try {
        store.writeObject( "hello", "Hello, DataStore! 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 DataStore 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*) will be thrown if it doesn't. Another reason for the exception would be if the write failed (for example, you ran out of disk space).

The first parameter specifies the *storeName*, the name that identifies the object in the DataStore. The name is case-sensitive. 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 highlighted 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 *DataStore*. Like *writeObject*, it may throw an *IOException* for mundane reasons like disk failure. It also cannot reconstitute the stored object without the object's class. If that class is not in the classpath, a *java.lang.ClassNotFoundException* is thrown.

If the named object cannot be found, a *DataStoreException* with the error code *STORE_NOT_FOUND* is thrown. It's important to catch that exception (a subclass of *DataSetException*) here, even though there's another **catch** at the bottom of the method, because jumping there would bypass the call to *close* the *DataStore* connection. (The code is structured in this somewhat awkward way for pedagogical reasons.)

Because *readObject* is defined to return a *java.lang.Object*, you almost always cast the return value to the expected data type. (If the object is not actually of that expected type, you will get a *java.lang.ClassCastException*.) Here, it is more of a formality, because the *System.out.println* method can take a generic *Object* reference.

Advantages for persistent object storage

You can now run *Hello.java*. The first time it runs, it will create the *DataStore* file and store the greeting string. When you run it again (and again...) the greeting with the date and time it was created will be displayed in the console.

For the simple persistent storage of objects, the *DataStore* has a number of advantages over using the JDK classes in the *java.io* package:

- It's simpler: only one class instead of four (*FileOutputStream*, *ObjectOutputStream*, *FileInputStream*, *ObjectInputStream*)
- You can keep all your objects in a single file and easily access them with a logical name, unlike streaming all your objects to the same file.
- With a single file, you can't accidentally lose an object or two, as you might with separate files. You may also use less storage space, because separate files can waste a lot of space due to the way disk clusters are allocated; the default block size in a *DataStore* file is small (4KB).
- You're not at the mercy of the host file system, which makes your application more portable. For example, different operating systems have different allowable characters for names; some are case-sensitive, while others are not. Naming rules inside the *DataStore* are consistent on all platforms.

Of course, an internal directory system would be practically useless without a way to get the contents of the directory.

Using the directory

The *DataStoreConnection.openDirectory* method returns the contents of the *DataStore* in an appropriately searchable structure. More on that in a moment. First, add the following program, *AddObjects.java*, to the project and run it to add a few more objects to the *DataStore*:

```
// 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 *DataStore* 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 will always be closed, even if there is an unhandled error. The *close* method is safe to call even if the connection never opened; in that case, it does nothing.

This time, three objects are written to the DataStore: an array of integers, a *Date* object (as opposed to 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 (or solidus or virgule: “/”) is the directory separator character. One of the names contains spaces, which is perfectly valid.

Demonstration class: Dir.java

Add another file to the project, Dir.java:

```
// 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 DataStore file, which is passed to its *print* method. The *print* method accesses that DataStore, using code similar to what you’ve seen before.

Opening a DataStore directory

What's significant here is that in addition to defining a *DataStoreConnection* to access the DataStore, a *StorageDataSet* is declared. After opening a connection to the DataStore, the *openDirectory* method of the *DataStoreConnection* is called to get the contents of the DataStore's directory. The directory of a DataStore is represented by a table.

DSX See "Viewing DataStore file information" on page 6-5.

DataStore directory contents

The DataStore directory table has nine columns—nine pieces of information about each stream in the DataStore, as shown in this table:

Table 2.1 DataStore 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 <i>storeName</i>
4	Type	DIR_TYPE	short	Bit fields that indicate the type of streams
5	Id	DIR_ID	int	A unique ID number
6	Properties	DIR_PROPERTIES	String	Properties and events for a DataSet stream
7	ModTime	DIR_MOD_TIME	long	Last time the stream was modified
8	Length	DIR_LENGTH	long	Length of the stream, in bytes
9	BlobLength	DIR_BLOB_LENGTH	long	Length of a table stream's BLOBs, in bytes

The columns may be referenced by name or number. There are constants defined as *DataStore* class variables for each of the column names. These constants are the preferred way of referencing a column; they provide compile-time checking to ensure that you are referencing a valid column. There are also constants with names that end with *_STATE* for the different values for the State column, and constants for the different values and bit masks for the Type column with names that end with *_STREAM*.

Stream details

Times in the DataStore directory are UTC (a compromise between the French [TUC] and English [CUT] acronyms for Coordinated Universal Time), suitable for creating dates with *java.util.Date(long)*.

As with many file systems, when you delete something in a DataStore, the space it occupied is marked as available, but the contents and the directory entry that points to it are not wiped clean. This leaves the

possibility of undeleting something. For more details, see “Deleting and undeleting streams” on page 2-20.

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 get to decide.

These internal streams have the same StoreName as the table stream with which they are associated. This means that the StoreName alone does not always uniquely identify each stream when interacting with the DataStore at a low level. Some of the internal stream types often have multiple instances. Therefore, the ID for each stream is required to 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. Due to the values stored in the State column, this means that all active streams will be 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 cannot use a *DataSetView* to use a different sort order.)

Reading a DataStore directory

You manipulate the DataStore directory table as you would any table with the DataExpress API. Use the *next* and *inBounds* methods to navigate through each entry in the directory, and the appropriate *get...* method to read the desired information for each stream.

You may not write to the DataStore directory; it is read-only.

To run *Dir.java*, set the runtime parameters in the Project Properties dialog box to the DataStore file to check; in this case, *Basic.jds*. When it runs, a loop goes through the directory, listing the name of every stream, something like:

```
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 DataStore directory. As a simple example, to display whether the stream is a table or file stream, add the highlighted 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 would change the output to:

```

F add/array of ints
F add/create-time
F add/values
F hello

```

indicating that all the serialized objects are indeed file streams.

Closing the DataStore directory

When you're not using the DataStore directory, you should close it by calling the *DataStoreConnection.closeDirectory* method. Most DataStore operations would modify the directory in some way. If the directory is open, it would have to be notified, which would slow down your application.

If you try to access the directory *StorageDataSet* when the directory is closed, you will get a *DataSetException* with the error code *DATASET_NOT_OPEN*.

Checking for existing streams

Although you could search the DataStore directory manually, the *DataStoreConnection* provides two methods for checking if 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 ignore streams that are deleted. They return **true** if there is an active stream of the corresponding type with that name in the DataStore, or **false** otherwise. Remember that stream names are case-sensitive, and you cannot have a table stream and a file stream with the same name.

For example, if you ran the following code fragment against *Basic.jds* as it is at this point in the tutorial:

```
store.tableExists( "hello" )
```

it would return **false**, because although there is a stream named “hello”, it’s a file stream, not a table stream. You would get the same result from:

```
store.fileExists( "Hello" )
```

this time because the name does not match case. When the name and type match:

```
store.fileExists( "hello" )
```

it would return **true**.

Storing arbitrary files

In addition to serializing discrete objects as file streams, you can store and retrieve data streams in a `DataStore` 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 via a `seek` method. Being a subclass of `InputStream` makes it easy to use streams stored in the `DataStore` in generic situations that expect an input stream; you will probably read a stream more often than you write one.

DSX See “Importing files” on page 6-17.

Demonstration class: `ImportFile.java`

Suppose you have an application that uses boilerplate documents that are modified for individual customers. There is a field in the customer table that contains their personalized copy, but you need to store the original somewhere as well, to make fresh copies for new customers. You could store the original as a file stream in the `DataStore`. The following utility program, `ImportFile.java`, will do this for you; 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 the name of a DataStore file, the name of the file to import, and an optional stream name as parameters. If no stream name is specified, the file name is used. The *main* method calls the appropriate form of the *read* method; the two-argument *read* method calls the three-argument *read* method.

When importing the file, the date it was last modified is recorded with it. The “/modified” suffix is appended to the stream name for this date,

while the `"/data"` suffix is appended to the stream name to contain the data from the file. These suffixes are defined as class variables.

The *read* method then begins with some now-familiar preliminaries: it opens a connection to the DataStore file 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 to create 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 may want to check if such a file stream exists with the *fileExists* method first (*ImportFile.java* does not). If there is a table stream with that name, *createFileStream* will throw 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 DataStore

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, and making them really large may cause performance problems as well. As a conservative start, you can make it a small multiple of the DataStore's block size.

The DataStore's block size is stored in the *DataStore* object's *blockSize* property. Whenever you use a *DataStoreConnection* to access a DataStore, it automatically creates an instance of *DataStore*. Other *DataStoreConnection* objects in the same process that connect to the same DataStore share that *DataStore* object. (Access to a DataStore file is exclusive to a single process; multi-user 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. This property is in kilobytes, so it is multiplied by 1024, and 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, just like 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 will be filled by reading and written into the *FileStream*. The last iteration will most likely not 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 closing the file stream, 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`, you could import 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.

To demonstrate opening, seeking, and reading is the following program, `PrintFile.java`. 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. The length of the file stream is determined by calling the *FileStream*'s *available* method and used as a file pointer. When reading from the file, the file pointer is moved forward, so the position must be decremented and set for each byte read in the loop. There are two forms of the *read* method: one that reads into a byte array (the same form of the method used by the

FileInputStream in `ImportFile.java`), and one that returns a single byte. The single-byte form is used; each byte is cast into a character to be printed.

Copying streams

The *DataStoreConnection* class' *copyStreams* method makes a new copy of one or more streams in the same *DataStore*, or copies the streams to a different *DataStore*. If an error is encountered in an original stream, an attempt will be made to correct that error in the copy. *copyStreams* is also the way to upgrade an older *DataStore* file into the current format.

copyStreams parameters

The *copyStreams* method takes six parameters, as listed in the following table:

Table 2.2 *DataStoreConnection.copyStreams* method parameters

Parameter name	Description
<i>sourcePrefix</i>	Stream name must begin with this to be pattern-matched; empty string to pattern-match all streams
<i>sourcePattern</i>	Stream name pattern to match, with standard * and ? wildcard characters
<i>destCon</i>	Connection to destination <i>DataStore</i>
<i>destPrefix</i>	Names of copies of streams have their <i>sourcePrefix</i> replaced with this; should be the same as <i>sourcePrefix</i> if the name should not be changed
<i>options</i>	Zero or more of the following <i>DataStore</i> class variables: <ul style="list-style-type: none"> • <code>COPY_CASE_SENSITIVE</code> • <code>COPY_IGNORE_ERRORS</code> • <code>COPY_OVERWRITE</code>
<i>out</i>	<i>java.io.PrintStream</i> to direct status messages; null to suppress output

Each of the *options* reverses the default behavior of *copyStreams*, which is to:

- Ignore case when matching stream names,
- Stop if an unrecoverable error is encountered, and
- Stop if a stream with the target name already exists in the destination.

If *copyStreams* stops because either of the last two conditions occur, it throws a *DataSetException*. Status messages for each stream that is copied will be output to the designated *PrintStream*.

DSX The *DataStore Explorer* provides a UI for copying streams to a new *DataStore* file with these parameters. See “Copying *DataStore* streams” on page 6-8.

Naming and renaming the streams to copy

As mentioned earlier, forward slashes in stream names are used to simulate a hierarchical directory structure—the key word being *simulate*. *copyStreams* is oblivious to a 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 get 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* will be stripped from the name, and the *destPrefix* will be 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”, in effect making 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 “three/levels/deep” to “not-a-peep” by specifying “three/levels/d” and “not-a-p”, in effect moving a stream up to the root directory of the DataStore. You can also do the reverse, 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 of the original DataStore file will be placed under a directory in the destination DataStore.

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 DataStore 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, that translates to copying everything that starts with “add/”, so the *sourcePattern* would be “add/*”. That would include everything in subdirectories, because the *sourcePattern* matches the entire rest 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* does not copy deleted streams.

Demonstration class: Dup.java

You may use the following program, `Dup.java`, to make a backup copy of a `DataStore` file or upgrade an older file:

```
// 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 store into another. A `DataStoreConnection` object is used to open the source `DataStore`. A `DataStore` object is used for the destination, so that the `DataStore` file can be created if necessary.

For the *copyStreams* method, the *sourcePrefix* and *destPrefix* are empty strings, and the *sourcePattern* is just *"**"*, which copies everything, without renaming. Unrecoverable errors will be ignored, and status messages are displayed in the console.

With this program, you can combine the contents of more than one DataStore file into a single file, as long as the stream names are different (*COPY_OVERWRITE* is not specified as an option).

Deleting and undeleting streams

Deleting streams is easy and certain; undeleting them is not guaranteed to 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 explained under "Stream details" on page 2-9; they're a little more complicated.

Deleting streams

The *DataStoreConnection.deleteStream* method takes the name of the stream to delete. For a file stream, the individual stream is deleted; for table stream, the main stream and all its support streams are deleted.

Deleting a stream does not actually overwrite or clear the stream contents. Like 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 contents may overwrite the space that was formerly occupied by the deleted stream, making the content of the deleted stream unrecoverable.

DSX See "Deleting streams" on page 6-8.

How DataStore reuses deleted blocks

Blocks in the DataStore file formerly occupied by deleted streams are reclaimed according to the following rules:

- The DataStore always reclaims deleted space before allocating new disk space for its blocks.
- If the DataStore 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 (like 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 will 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 (there is no chance for recovery anyway).

Undeleting DataStore streams

Because table streams have multiple streams with the same name, the stream name alone is not sufficient for attempting to undelete a stream. You must use a row from the DataStore 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 will be used as the row to undelete.

Note that you can create a new stream with the name of a deleted stream. You cannot undelete that stream while its name is being used by an active stream.

DSX See “Undeleting streams” on page 6-8.

Demonstration class: DeleteTest.java

The following program, *DeleteTest.java*, demonstrates both deletion and undeletion in a pedagogical (and therefore somewhat atypical) way:

```
// 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 DataStore file and the stream to be deleted are hard-coded. The stream is “add/create-time”, which was added to `Basic.jds` in the demonstration program `AddObjects.java`, on page 2-7. It’s a file stream primarily 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 DataStore and opening its directory. Next, it locates the directory entry for the stream that is about to be deleted.

Note In normal usage, you would probably 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, to be explained shortly.

To locate the row, a new *com.borland.dx.dataset.DataRow* is instantiated from the directory dataset, specifying the two columns that will be used in the search: the 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 *undeleStream*, the current row is used. But because of the way the DataStore directory is sorted (as explained in “Directory sort order” on page 2-10) when a stream is deleted, its directory entry will probably “fly away” to its new position at the bottom of the directory as the most recently deleted stream; the current row will then be 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 can 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 DataStore directory dataset, an individual row is a static copy. It's smaller, and 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 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 DataStore directory is closed.

The file stream is then deleted by name, using the plain name string defined at the beginning of the method. (You could use the name from the directory entry, which should be the same, but that's a little too convoluted.) Finally, the stream is undeleted, using the individual directory entry.

Packing DataStore files

The only way to shrink a DataStore file—removing unused blocks and directory entries for deleted streams—is to copy the streams to a new DataStore file using *copyStreams*. Only active streams are copied, resulting in a packed version of the file.

DSX See “Packing DataStore files” on page 2-24.

DataStore as an embedded database

DataStore provides embedded database functionality in your applications with a single DataStore file and the DataStore JDBC driver (and its supporting classes). No server process is needed for local connections. In addition to industry-standard JDBC support, you may take advantage of the added convenience and flexibility of accessing the DataStore directly through the DataExpress API. You may use both types of access in the same application.

JDBC access requires that the DataStore be transactional; DataExpress does not. This chapter begins with DataExpress access, then discusses transactional DataStores, and finally the local JDBC driver. The remote JDBC driver and DataStore JDBC server are discussed in Chapter 4, “Multi-user and remote access to DataStores.”

Using DataExpress for data access

To use a DataStore as a database file, all you have to do is associate a component that extends from *StorageDataSet*, like *TableDataSet*, to a stream inside a DataStore. The *StorageDataSet* represents a table in the embedded database, and provides all the methods necessary to navigate, locate, add, edit, and delete data.

Demonstration class: DxTable.java

Start a new file in the *dsbasic* project/package, DxTable.java:

```
// DxTable.java
package dsbasic;

import com.borland.datastore.*;
import com.borland.dx.dataset.*;

public class DxTable {

    DataStoreConnection store = new DataStoreConnection();
    TableDataSet        table = new TableDataSet();

    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 DataExpress will be used heavily, this program imports the DataExpress package in addition to the DataStore package. The class has two fields: a *DataStoreConnection*, and a *TableDataSet*. The *main* method instantiates a new instance of the class and executes its *demo* method.

Connecting to a DataStore with *StorageDataSet*

The focal point of DataExpress semantics is the class *StorageDataSet*. This class has three subclasses to be 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.

When defining the table from scratch with DataExpress, the *TableDataSet* is appropriate.

Each *StorageDataSet* has a *store* property, which is **null** when the object is instantiated. If it is still **null** when the dataset is opened, a *com.borland.dx.memorystore.MemoryStore* will be assigned automatically, which means that the data is stored in memory. By assigning a *DataStoreConnection* or *DataStore* to the *store* property, persistent storage in a *DataStore* file will be used instead.

A *StorageDataSet* connects to a *DataStore* by assigning three properties:

- 1 The *fileName* property of the *DataStoreConnection*. This indicates which *DataStore* to connect to.
- 2 The *storeName* property of the *StorageDataSet*. This indicates the name of the table stream inside the *DataStore*. You can reuse an existing name, if it's for the same dataset. Otherwise, you must use a new name. (It's up to you to manage what's inside the *DataStore*, and choose names that don't conflict.)
- 3 The *store* property of the *StorageDataSet* is set to the *DataStoreConnection* (or *DataStore*) object. This connects the two together.

These three steps can be done in any order. Once all three properties are set, you have a fully qualified connection between a *StorageDataSet* and a *DataStore*.

In *DxTable.java*, the *DataStore* file is *Basic.jds*, created in "Creating a *DataStore* file" on page 2-3. The table stream is named "Accounts"; for all intents and purposes, this is the name of the table. The *DataStoreConnection* is assigned to the *TableDataSet*'s *store* property.

Then the *TableDataSet* is opened. Opening a dataset that has a *DataStore* attached will automatically open that *DataStore* file. If the *DataStore* is opened successfully, the named table stream will be created if it doesn't already exist, or reopened. This establishes an open connection between the dataset and its table stream in the *DataStore*. Note that if there is a file stream in the *DataStore* with the same name, an exception will be thrown because you cannot have a table stream with the same name as a file stream.

Creating *DataStore* tables with DataExpress

So opening a *StorageDataSet* connected to a *DataStore* results in an open table stream. For new table streams, *QueryDataSet* and *ProcedureDataSet* will then fetch data from their data source and populate the table stream, as explained in "Tutorial: Offline editing with *DataStore*" on page 5-2. But *TableDataSet* has no data source, so you start with an empty and undefined table stream.

Add the highlighted statements to `DxTable.java`:

```
table.open();
if ( table.getColumns().length == 0 ) {
    createTable();
}
} catch ( DataSetException dse ) {
```

You can detect that a table stream is brand new by checking the number of columns in the *TableDataSet*. If it's zero, you can then define the columns in the table. In this case, it's done by a method arbitrarily but sensibly named *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 demo 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.

After the structure of the table has been modified with the addition of these new columns, the last step is to activate the changes by calling the *StorageDataSet.restructure* method. The result is an empty but structured table stream, a new table in the *DataStore*. (If you know the table does not exist, you can use *addColumn*s to define the structure before opening the *TableDataSet*, and then you won't need to call *restructure*.)

You can store as many tables as you want in a single *DataStore* file. Of course, they must use different table stream names. You may use the same *DataStoreConnection* object in the *store* properties of each *TableDataSet*.

There are other ways to create tables in a *DataStore*. In particular, you can use an SQL CREATE TABLE statement through the *DataStore JDBC* driver.

Using DataStore tables with DataExpress

Once the tables in the *DataStore* have been defined (no matter how they were created), you are free to use the rest of the DataExpress API through a *TableDataSet* object, just as you would with any dataset. You can create filters, indexes, master-detail links, and so forth; in fact, such secondary indexes are also persisted and maintained in the *DataStore* file, making the *DataStore* a complete embedded database.

To complete the demonstration program, add a smattering of DataExpress functionality, including the following 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 ) {
```

After opening the table, creating its structure if necessary, a *SortDescriptor* on the ID field is set. Then the new *appendRow* method is called to add some rows.

That *appendRow* method starts by going to the last row in the table, and getting the value of the ID field. Because of the sort order, this should be the highest ID number used so far. (If the table is empty, the *getInt* method returns zero.) The new ID value will be one greater than the last. A new row is inserted, the values are set, including the Update field which is set to the current date and time, and the new row is saved with the *post* method.

After appending a few rows, a loop navigates through the table, displaying its contents in the console. Finally, the *DataStore* and *TableDataSet* are closed.

If you run the program a few times, you will 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 may not be safe. A more robust approach requires understanding locks and transactions.

Transactional DataStores

So far, changes you have made to a DataStore 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 is done without concern for other connections that might be accessing the same stream. Such changes are immediately visible to those 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, and is required for JDBC access.

Enabling transaction support

Transaction support is provided by the *com.borland.datastore.TxManager* class. A DataStore may be transactional when it is first created; or you can add transaction support later. In either case, you assign a *TxManager* object to the *txManager* property of the *DataStore* object before calling the *create* or *open* method. (Attempting to assign the *txManager* property for an open DataStore causes an exception.)

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 must be done before creating or opening the DataStore.

The first time the now-transactional DataStore opens, it stores its transactional settings internally. The next time you open the DataStore, you do not have to assign a *TxManager*; the DataStore will automatically instantiate a *TxManager* with the settings it has stored.

To open (or create) a transactional DataStore, you must also set the *DataStoreConnection.userName* property. The *userName* property is used to identify individuals in a multi-user environment when necessary, for example during lock contention. If there is no name in particular that would be appropriate, you can set it to a dummy name.

Creating new transactional DataStores

The minimum code for creating a new transactional DataStore with default settings is something like:

```

DataStore store = new DataStore();

store.setFileName( "SomeFileName.jds" );
store.setUserName( "AnyNameWouldWork" );
store.setTxManager( new TxManager() );
store.create();

```

The two differences between this code and a non-transactional *DataStore* is the setting of the *userName* and *txManager* properties (which can be in either order). If you don't want the default settings, the code will generally look 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**.

DSX See “Creating a new *DataStore* file” on page 6-3.

Adding transaction support to existing *DataStores*

The code for making an existing *DataStore* transactional is very similar; the main difference is the call to *open* instead of *create*. For a default *TxManager*, the code would be something like:

```
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 *DataStore* file, you cannot use one when you are adding transaction support, because *txManager* is a property of *DataStore*, not *DataStoreConnection*.

DSX See “Making the *DataStore* transactional” on page 6-10.

Opening a transactional *DataStore*

The only difference when opening a *DataStore* 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 *DataStore* (it's ignored), you might want to always specify a *userName* when opening a *DataStore*. The code would look something like:

```
DataStoreConnection store = new DataStoreConnection();

store.setFileName( "SomeFileName.jds" );
store.setUserName( "AnyNameWouldWork" );
store.open();
```

Because no *TxManager* was assigned, when the *DataStore* opens, a *TxManager* is automatically instantiated, with its properties set to the values that were persisted in the *DataStore*, and assigned to the

DataStore's *txManager* property. You can get the values of the persisted transaction management properties from there, but you cannot change directly.

Changing transaction settings

To change a DataStore'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 DataStore, only those properties that have been assigned in the new *TxManager* will be changed. All other properties remain as they were; they do not revert to default values (the ones in the new *TxManager*).

As when adding transaction support, the *TxManager* with the new values must be assigned before you open the DataStore. For example, suppose you want to change the *softCommit* property to **true**, which would improve performance by not guaranteeing recently committed transactions (within approximately one second before a system failure), while still guaranteeing 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*, are not set. Although the new *TxManager* has the default setting when it is assigned to the DataStore, the setting in the DataStore will not be affected, even if it is not the default.

DSX See “Modifying transaction settings” on page 6-10.

Transaction log files

The transaction manager works by logging changes made to the DataStore, including the previous values so that the transaction can be rolled back. (The changes are not 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 reconstruct the contents of the DataStore.) Most of the *TxManager* properties control the transaction log files, including:

- 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 duplexing, the two copies are usually kept in different locations. If they can be kept in different physical drives, that not only increases reliability and the chance for recovery even further, it also may offset some of the performance penalty.
- How big they should get before starting another one.

By default, you get one copy of the log files (simplexing instead of duplexing), in the same directory that contains the DataStore file.

Whenever a DataStore is transaction-enabled the first time, it creates its log files. The names of the log files use the name of the DataStore file, without the file extension. For example, if the DataStore file `MyStore.jds` uses simplex transaction logging, the following log files are created:

- The status file `MyStore_STATUS_0000000000`,
- The anchor file `MyStore_LOGA_ANCHOR`, and
- 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 location of these files is controlled by the *ALogDir* and *BLogDir* properties.

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. Conversely, as old log files are no longer needed for active transactions or crash recovery, they are automatically deleted. For information on archiving them, see “Saving log files” on page 8-2.

Moving transaction log files

The *ALogDir* and *BLogDir* properties include the drive and full path, which means two things:

- If you’re going to create transactional DataStore files before moving them to another computer, you should try to create them in a location with the same drive and directory name. For example, if you intend to deploy the files to the D: drive, but the DataStore files were created on the C: drive because you don’t have a D: drive on your development computer, you will have to go through the extra steps of moving the log files when you deploy, because the drives will be different.
- Whenever you move log files, follow these steps:
 - 1 Move the log files to the new location. Be sure to remove or rename any copies of the files in the original location.
 - 2 Create a new *TxManager* with the new location property settings. Assign it to the *txManager* property of the DataStore.
 - 3 Open the DataStore. The *TxManager* will look in the new location, see that the log files are there, and change the persisted settings in the DataStore.

Bypassing transaction support

Sometimes you will want to access a transactional `DataStore`, but need to bypass transaction support. Situations include:

- The transaction log files are lost. You will not be able to open the `DataStore` normally.
- You only have read-only access to the `DataStore` files. For example, they may be on a CD-ROM or a network directory where you don't have write access.

In both cases, you can temporarily bypass transaction support by opening the `DataStore` in read-only mode. This must be done through a *`DataStore`* object; before opening, set its *`readOnly`* property to **true**, like this:

```
DataStore store = new DataStore();

store.setFileName( "SomeReadOnly.jds" );
store.setReadOnly( true );
store.open();
```

Because you are bypassing the *`TxManager`*, you do not need to set the *`userName`*. If the transaction log files are lost, use the *`copyStreams`* method (or the `DataStore Explorer`, see “Copying `DataStore` streams” on page 6-8) to copy the streams to another file.

Removing transaction support

A `DataStore` is made non-transactional by assigning a new *`TxManager`* that has its *`enabled`* property set to **false** (by default, this property is **true**). The `DataStore`'s *`consistent`* property must be **true**—it reflects that the `DataStore` is internally consistent—or the change will not be allowed to take effect. Because you are disabling the *`TxManager`*, you do not need to set the *`userName`*. For example:

```
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`* does not affect any existing log files. If you make the `DataStore` transactional again, any existing log files will be reused if appropriate.

DSX See “Removing transaction support” on page 6-11.

Deleting transactional DataStores

When you delete a transactional DataStore file, be sure to delete its log files as well. If you don't, you won't be allowed to create a new DataStore file with the same name, because the log files won't match.

Controlling DataStore transactions

Once a DataStore has been made transactional, you can basically ignore the *TxManager*; in fact, the only time you will need to reference one is if you want to examine or change the DataStore'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 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 only have one *DataStore* object accessing a particular DataStore file, while you can have many *DataStoreConnection* objects. When you open a *DataStoreConnection*, it will contain a reference to a *DataStore* object, as explained in "Referencing the connected DataStore" on page 2-14. If there is no suitable *DataStore* object in memory, the *DataStoreConnection* will automatically open a *DataStore* to satisfy this reference. This means that if you open a *DataStoreConnection* first, subsequent *DataStore* objects accessing the same DataStore file will have their allowed functionality reduced so that they behave like a *DataStoreConnection*.)

A transaction's lifecycle begins with any read or write operation through a connection. The DataStore 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 will hold on to it until the transaction is committed or rolled back.

In single-connection applications, transactions may be considered primarily as a feature that allows you to undo changes and provide crash recovery. Or you may have made a DataStore transactional so that it can be accessed via JDBC; if you want to access that DataStore via DataExpress, you must now deal with transactions. The way transactions work has deeper ramifications for multi-connection (multi-user or single-user multi-session) applications. These are discussed in "Avoiding blocks and deadlocks" on page 4-5, along with other multi-user issues in Chapter 4, "Multi-user and remote access to DataStores."

Committing and rolling back transactions

Controlling transactions boils down to three methods of *DataStoreConnection*:

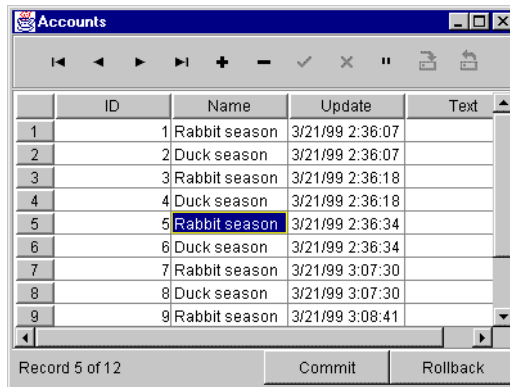
- To see if a transaction has been started, call *transactionStarted*.
- To commit a transaction, call *commit*.
- To roll back a transaction, call *rollback*.

When a *DataStoreConnection* is closed, it will attempt 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: Controlling transactions via DataExpress

This tutorial creates a simple Swing-based application that provides a means to commit and roll back transactions. It also detects the automatic commit on close, allowing the user to decide whether to commit. In the process, it shows some important details about using a *DataStore* in a GUI application. The end result looks like this:

Figure 3.1 The complete AccountsFrame



Step 1: Create a transactional DataStore with test data

At this point, you should already have a *DataStore* file 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 *DataStore* files, transactional and non-transactional, to play with.

Make a copy of the file, naming it `Tx.jds`. This is a copy of the entire file, so you can use the Windows Explorer, or the command line copy command. 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 will make any DataStore file transactional, if it isn't already. For DataStores 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 will take 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 is to create a data module with the DataStore and a *TableDataSet*:

- 1 Select File | New.
- 2 Select the 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 checkbox 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. (If you can't think of a name, how about Chuck.)

- 7 Add a *TableDataSet* 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 admire the generated code.
- 10 Save the file.

Step 3: Create a GUI for the DataStore table

Now, a simple table grid to display the data, with a very important detail at the end:

- 1 Select File | New.
- 2 Select the 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 checkbox 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.TableDataSet1* (the only choice).
- 13 Switch back to source view.
- 14 Go to the *processWindowEvent* method. This method is generated so that *System.exit* will be called when the window is closed. That's fine, but it's important that you close the DataStore 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 DataStore is closed, no matter how many connections the application is using. You should use *DataStore.shutdown* in this situation, which closes

the DataStore file directly. That is why this application uses a *DataStore* instead of a *DataStoreConnection*.

You could place the *shutdown* method call just before the *System.exit*, but for reasons that will become apparent shortly, you want to do this before the window physically closes. Insert the highlighted statements:

```
//Override so we can exit on System Close
protected void processWindowEvent(WindowEvent e) {
    if (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 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):

```
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—not that you can tell at the moment. When you close the window, the DataStore will be closed, and the changes you made will be committed, which you can verify by running the application again.

If you did not close the DataStore (or at least commit the current 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 DataStore. No changes you made in the application would ever apply. Closing the DataStore 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.TableDataSet1` (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 *actionPerformed* event handler for the `commitButton` to:

```
void commitButton_actionPerformed(ActionEvent e) {
    try {
        datastore.commit();
    } catch ( DataSetException dse ) {
        dse.printStackTrace();
    }
}
```

- 14 Set the *actionPerformed* event handler for the `commitButton` to:

```
void rollbackButton_actionPerformed(ActionEvent e) {
    try {
        datastore.rollback();
    } catch ( DataSetException dse ) {
        dse.printStackTrace();
    }
}
```

These buttons will now call *commit* or *rollback* on the `DataStore` to commit or roll back any changes made during the current transaction; that is, since the last commit or rollback.

Step 5: Add control over auto-commit

This last step enables the application to detect the auto-commit when the `DataStore` 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 is displayed, asking the user if they want to commit the changes. “Yes” will send the `ok` response, which signals the `DataStore` to commit the changes. “No” will send the `cancel` response, which signals the `DataStore` to roll back the changes.

- 3 Add the highlighted statement to add the frame as one of the `DataStore`’s `ResponseListeners`.

```
private void jbInit() throws Exception {
    dataModule = dsbasic.AccountsDM.getDataModule();
    dataStore = dataModule.getDataStore();
    dataStore.addResponseListener( this );
}
```

With these additions, the user will get a dialog box if there are unsaved changes, asking them if they want to commit them. Remember that the `DataStore` is closed before the window is; otherwise 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 closing the window to exercise the auto-commit handling.

Using JDBC for data access

`DataStore` tables may be accessed via the `DataStore`’s Type 4 (direct pure Java) JDBC driver, `com.borland.datastore.jdbc.DataStoreDriver`.

This driver may be used for both local and remote access. Remote access requires a DataStore JDBC server, which is also the vehicle for multi-user access. For details on remote access and multi-user issues, see Chapter 4, "Multi-user and remote access to DataStores."

The local connection URL is:

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

As with any JDBC driver, you may use the JDBC API, or an added-value API like DataExpress with *QueryDataSet* and *ProcedureDataSet* to access tables.

Demonstration class: JdbcTable.java

The following program, *JdbcTable.java*, is functionally identical to its DataExpress twin, *DxTable.java*, on page 3-2. 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 last ID value for that append. These prepared statements should be initialized before calling the *appendRow* method, and 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 *DataStore* 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 if the table exists. One way to do this is through *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 it must be created by calling the *createTable* method. The *createTable* method uses a fairly straightforward SQL CREATE TABLE statement. Note that the SQL parser usually converts identifiers to uppercase. To keep the proper casing used by *DxTable.java*, the identifiers must be enclosed in quotes in this and other SQL statements. (In the long run, you may get the notion that case-sensitivity is more trouble than it’s worth.) The final job of the constructor is to create the two prepared statements.

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 an SQL SELECT statement to fetch the rows and a loop that’s very similar to the one in *DxTable.java*, and 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 DataStore JDBC driver is implemented. But this internal object is not accessible, so you must use the JDBC API.

For control over transactions, disable auto-commit mode by calling *Connection.setAutoCommit(false)*. You can then call *commit* and *rollback* on the Connection object.

Multi-user and remote access to DataStores

DataStore applications are not limited to accessing local DataStore files. Through the use of a DataStore JDBC server, you can access DataStores from other machines. The DataStore JDBC server also enables multi-user access. Access requests from numerous clients are handled by the server process.

Using the JDBC driver for remote access

DataStore's Type 4 (direct pure Java) JDBC driver *com.borland.datastore.jdbc.DataStoreDriver* may be used to access both local and remote DataStore files. While the URL for local connections is:

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

the URL for remote connections is:

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

A DataStore JDBC server process must be running on the *<hostname>* machine. Communications between the client application and the DataStore JDBC server use port 2508 by default. You can change the port number when starting (or restarting) the server, and on the client by getting the connection with extended properties. For example, if you want to access the DataStore file *c:\someApp\ecom.jds* on the computer *mobile.mycompany.com* through port 9876, you might do something like:

```
Class.forName( "com.borland.datastore.jdbc.DataStoreDriver" );
java.util.Properties info = new java.util.Properties();
info.setProperty( "user", "MyUserName" );
info.setProperty( "port", "9876" );
Connection con = DriverManager.getConnection(
    "jdbc:borland:dsremote://mobile.mycompany.com/c:/someApp/ecom.jds", info );
```

For more information on connection properties, see the `DataStoreDriver` component “Driver properties,” in Chapter 3, “`datastore.jdbc` package,” of the *DataExpress Component Library Reference*.

Other than these differences, a remote JDBC connection operates in much the same manner as a local JDBC connection, from the client application’s perspective. For more information, see “Using JDBC for data access” on page 3-17.

Because DataStores accessed via a remote connection are potentially DataStores accessed by multiple users, you must consider some concurrency issues, as detailed in “Avoiding blocks and deadlocks” on page 4-5.

Running the DataStore JDBC server

To access DataStore files through a remote JDBC connection, a DataStore JDBC server must be running on a machine that has local access to those files. This could be the actual machine that contains the files, or a machine that has direct network access to DataStore files on a networked drive.

For development purposes, you can start the server from the JBuilder menu. Select **Tools | DataStore Server**. This starts a minimized server process. Restoring the server reveals a simple UI:

Figure 4.1 DataStore JDBC server



From the menu, you can get help, terminate the server, or run the DataStore Explorer (for more information on the DataStore Explorer, see Chapter 6, “Using the DataStore Explorer”). You can also change a few items in the server’s configuration.

Reconfiguring the server

You cannot change the server configuration while it is running. The File | Shutdown menu option will stop the actual server process. All connections will be closed in an orderly fashion.

Once the server has stopped, you can use the File | Port Number and File | Temp Directory menu options to change those aspects of the server. Then use File | Startup to restart the server process.

Deploying the DataStore JDBC server

Once an application goes into real-world use, you will need to deploy the DataStore JDBC server to a server machine.

Packaging the server

The JARs you need to run the server depends on whether you want the GUI. Without the GUI, you only need:

- dsserver3.1.jar
- datastore3.1.jar
- dx3.1.jar

With the GUI you will need, in addition to the JARs already listed:

- dbswing3.1.jar
- dbtools3.1.jar
- pthelp3.1.jar

If you want the server's online help to be available, you will also need to copy the help JARs from the `/doc` directory. The help files for the DataStore JDBC server are in `jb_ui.jar`. The *JDataStore Programmer's Guide* is in `jb_dspg.jar`.

Starting the server

To start the server, you must add the necessary JARs to the classpath. The main class for the DataStore JDBC server is *com.borland.dbtools.dsserver.Server*, so the command line for starting the server with default options is:

```
java com.borland.dbtools.dsserver.Server
```

You may also specify the options listed in the following table:

Table 4.1 DataStore JDBC server startup options

Option	Description
-port=<number>	The port to listen to. Default: 2508
-ui=<uiType>	The look and feel of the UI. One of the following: <ul style="list-style-type: none">• windows• motif• metal• none• <LookAndFeel class name>
-temp=<dirName>	The directory to use for all temporary files
-doc=<helpDir>	The directory that contains online help files
-?	Displays a message listing these options
-help	

If you do not use the `-ui` option (or specify `none`), the server will start as a console application. Without a UI, you will not be able to reconfigure the server or launch the DataStore Explorer. To halt the server, use the appropriate operating system or shell action. For example, when running in the foreground, press `Ctrl+C`; when running in the background on Unix, use the `kill` command.

Creating custom JDBC servers

The DataStore JDBC server that comes with JBuilder provides remote access to DataStore files. 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. Another example would be to add the ability to retrieve file streams stored in a DataStore; file streams are not accessible through JDBC.

The core of any DataStore JDBC server is the `com.borland.datastore.jdbc.DataStoreServer` thread. In addition to anything else it might do, the server runs an instance of this thread. For more information, see the `datastore.jdbc` package in the *DataExpress Component Library Reference*.

Multi-user transaction issues

Multi-user access introduces potential problems with transactions, ranging from decreased performance to deadlock. (These same issues may also be a problem for complex single-user applications that use multiple transactional connections.) Avoiding or minimizing these problems requires an understanding of the transaction mechanisms employed by the DataStore.

Transaction isolation level

The DataStore uses both stream locks and row locks to manage access to resources. Stream locks provide the strongest possible transaction isolation level, designated by *java.sql.Connection.TRANSACTION_SERIALIZABLE*. It prevents:

- Dirty reads, where one connection is allowed to read data written by another connection that has not been committed.
- Nonrepeatable reads, where 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, where 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.

Serializable transactions guarantee that the data is consistent during the transaction's lifetime.

Row locks are used to enforce the next strongest transaction isolation level, designated by *java.sql.Connection.REPEATABLE_READ*. It allows phantom reads, but prevents nonrepeatable and dirty reads. This level of isolation allows more concurrent access at the expense of absolute consistency.

Avoiding blocks and deadlocks

A connection normally requires a lock to either read from or write to a stream or row, and can be blocked by another connection that's either reading or writing. You can prevent blocks in two ways:

- Minimize the lifespan of transactions that write.
- Use read-only transactions, which do not require locks, to read.

Conserving write transactions

Connections should use "burst" writes; that is, accumulate changes, then when necessary start a transaction, immediately write those changes, and commit them. This happens to be the preferred model for most database servers, and is the model used by DataExpress in its provider/resolver paradigm.

Using read-only transactions

Read-only transactions are not 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.dataset.sql.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 *DataStore*. Only data from transactions committed at the point the transaction started are seen in this snapshot (otherwise, the connection would have to see if there are pending changes and roll them back whenever it accesses the data). A snapshot is taken when the *DataStoreConnection* is opened, and is refreshed every time its *commit* method is called.

Detecting blocks and deadlocks

The *DataStoreConnection* has a *lockWaitTime* property that defaults to ten seconds. If a lock cannot be secured in that time, the transaction will abort with an informative exception. This mechanism applies to both long-duration blocks and deadlocks.

When a transaction aborts, it holds on to its locks. You must decide whether to commit or roll back the transaction, or retry the lock. Note that in a deadlock situation, the second connection will continue to be deadlocked until you end the transaction (by committing or rolling back), or that second transaction also times out.

Persisting data in a DataStore

The data-aware dbSwing and JBCL controls bind to DataExpress datasets. The DataExpress paradigm cleanly separates data sources from datasets, through providers and resolvers, making DataExpress ideal for multi-tier applications.

After data is provided to an application or a data module, you can view and work with the data locally in data-aware controls. You can store your data to local memory (*MemoryStore*) or to a local file (*DataStore*). When you want to save the changes back to your database, you must resolve the data.

Using *DataStore* instead of *MemoryStore*

By default, once data is loaded into a JBuilder component the storage mechanism is in-memory storage through the use of a *MemoryStore*. Alternate storage systems such as the *DataStore* are allowed 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* will remember the rows fetched in a table, even after the application has been terminated and restarted. In addition, you can increase the performance of any application with large *StorageDataSets*. *StorageDataSets* using *MemoryStore* have a small performance edge over *DataStore* for a small number of rows; however, *DataStore* stores data and indexes in an extremely compact format. As the number of rows in a *StorageDataSet* increases, using a *DataStore* is much more performant and requires much less memory than using a *MemoryStore*.

Whether the data's storage is *MemoryStore* or *DataStore* often does not effect how you work with a *StorageDataSet* or other data-aware controls connected to the *StorageDataSet*. However, storing Java objects in columns does require the use of Java serialization (*java.io.Serializable*). If this is not possible, you cannot use DataStore components and should use the default in-memory storage mechanism.

Using a DataStore with StorageDataSets

Caching and persisting *StorageDataSets* in a *DataStore* is accomplished through the three property settings discussed in “Connecting to a DataStore with StorageDataSet” on page 3-2. Persisting data from a provider usually involves the two subclasses of *StorageDataSet* with predefined providers: *QueryDataSet* (which uses *QueryProvider*) and *ProcedureDataSet* (which uses *ProcedureProvider*).

To store and persist the data from one of those *StorageDataSets* in a *DataStore* using the design tools:

- 1 Select the application's Frame file and switch to design view.
- 2 Add a *DataStoreConnection* component from the Data Express tab of the component palette to the component tree.
- 3 In the Inspector, select the *fileName* property of the *DataStoreConnection*. Use the Browse button to bring up the Open dialog box and enter the DataStore file name. Click Open.
- 4 Select the *QueryDataSet* or *ProcedureDataSet* component in the Component tree. (Note that you can also use a *TableDataSet* with its *provider* property set to a *QueryProvider* or *ProcedureProvider*.)
- 5 In the Inspector, set the *storeName* property to the name you want to use for the table stream in the DataStore.
- 6 In the Inspector, set the *store* property of the *StorageDataSet* to the *DataStoreConnection* component.

Tutorial: Offline editing with DataStore

This tutorial provides the steps for creating an application that uses a *DataStore* component to enable offline editing of data. The server database will be a sample DataStore file, *employee.jds*, accessed through the DataStore Server. Do not confuse this file with the DataStore used for persistence. Locate the sample file before beginning (it is installed in *samples/JDataStore/datastores*).

- 1 Start the DataStore Server from the menu item Tools | DataStore 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, use the Frame Class name to `PersistFrame`
 and 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 `d:/jbuilder` in the URL:

Property name	Setting
Driver	<code>com.borland.datastore.jdbc.DataStoreDriver</code>
URL	<code>jdbc:borland:dsremote://localhost/d:/jbuilder/samples/JDataStore/datastores/employee.jds</code>
Username	<code><use any name></code>
Password	<code><leave blank></code>

Click the Test Connection button to check that the connection properties have been correctly set. When the connection is successful, click OK.

- 6 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 “Datastore 3.1” library to your project properties if it was not already listed.
- 7 Open the *fileName* property editor for the *DataStoreConnection* component. Type the name for a new DataStore file. Be sure to include the full path; use the Browse button to help. You do not have to specify a file extension; a DataStore always has the extension `.jds`. Click OK.

Note

The Designer will automatically create this DataStore file for you when it is connected to the *StorageDataSet*, so that the tools work fully. When you run the application, the DataStore file will already be there. But if you run the application on another computer, the DataStore file won't be there. You will have to add extra code to create the DataStore file if necessary, as shown in “Creating a DataStore file” on page 2-3.

- 8 Add a *QueryDataSet* component from the Data Express tab to the component tree.

Open the *query* property editor for the *QueryDataSet* component in the Inspector and set the following properties:

Property name	Value
Database	<i>database1</i>
SQL Statement	<code>select * from employee</code>

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.

- 9 Set the *storeName* property of the *QueryDataSet* to *employeeData*.
- 10 Set the *store* property to *dataStoreConnection1* (the only choice).
- 11 Add a *JdbNavToolbar* component from the dbSwing tab to the North position of the frame. Set its *dataSet* property to *queryDataSet1*.
- 12 Add a *JdbStatusLabel* component from the dbSwing tab to the South position of the frame. Set its *dataSet* property to *queryDataSet1*.
- 13 Add a *TableScrollPane* component from the dbSwing tab to the Center position of the frame.
- 14 Add a *JdbTable* component from the dbSwing tab to the *TableScrollPane*. Set its *dataSet* property to *queryDataSet1*.
- 15 Instead of adding code to call *DataStore.shutdown* before exiting the application (as done in an earlier tutorial) you can use a *DBDisposeMonitor* component to close DataStore files automatically when you close the frame.

Add a *DBDisposeMonitor* component from the More dbSwing tab to the component tree. Set its *dataAwareComponentContainer* property to *this*.
- 16 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 *DataStore* file (the persistence DataStore specified in step 7 above). Changes are also saved to the file when you move off of a row, just as they are with an in-memory data set (*MemoryStore*).

Note A data set in a DataStore can have tens or hundreds of thousands of rows. Handling that much data using an in-memory data set would have an undesirable impact on application performance.

Close the application and run it again. You see the data as you edited it in the previous run. This, of course, is very different from the behavior of an in-memory data set. If you want, you can exit the application, shut down the DataStore Server, and run the application again. Without any connection to the SQL database, you can continue to view and edit data in the *DataStore*. This would be especially useful if you want to work with data offline, such as at home or on an airplane.

Understanding how DataStore manages offline data

So far in the tutorial, nothing has been saved back to the SQL database on the server. On the *JdbNavToolbar*,

- The Post button saves the changes in the current row to the DataStore file.
- The Save button saves all changes that have been accumulated in the DataStore 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 DataStore, including any edits not saved back to the server, with the results of the query. 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, the Execute Query Immediately When Opened option is selected. This option indicates how data is loaded into the DataStore file when the application was *first* run. On subsequent runs, the execution of the query is suppressed, because the data set is found in the DataStore file instead of on the server. As a result,

- Changes that haven't been saved to the server are preserved when you exit and restart the application.
- No special code needs to be written to get data into the DataStore on the first run.
- Once data is in the DataStore, you can work offline. In fact, a connection to the database is not even established until you do an operation that needs it, such as saving changes.

The Execute Query Immediately When Opened option is not allowed to overwrite existing data (unless the *StorageDataSet.refresh* method is called 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 DataStore file, 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; you will get an exception because the attempt to connect will fail, but you won't lose any of the changes you have made.

Restructuring DataStore StorageDataSets

The Column Designer in the JBuilder UI designer provides support for moving, deleting, and inserting columns. Column data types can also be changed. When the data type of a *Column* component in a *StorageDataSet* that is bound to a DataStore is changed, type coercions occur when going from one type to another.

To activate the Column Designer,

- 1 Select the Data Module file in the Navigation pane.
- 2 Press the Design tab for your *DataModule*.
- 3 Right-click a *StorageDataSet* in the JBuilder Structure pane.
- 4 Select Activate Designer.

This designer works for *StorageDataSets* that are using a *MemoryStore* or a *DataStore*. *MemoryStore* performs all operations instantly. When a Column data type is changed, *MemoryStore* does not convert data values to the new data type. The old values are lost.

DataStore does not perform the move/insert/delete/change type operations on *StorageDataSets* immediately. The structural change is noted inside the DataStore directory as a "pending" operation. The *StorageDataSet.getNeedsRestructure* method returns **true** when there is a "pending" restructure operation. A *StorageDataSet* with pending structural changes can still be used.

- Moved columns can be read and written to.
- Deleted columns are not visible.
- Inserted columns can be read, but not written.
- Changed data type columns can be read but not written to.

To make a pending restructure operation take effect, click the Restructure toolbar button in the Column Designer. You can also cause a restructure operation to happen programmatically by calling the *StorageDataSet.restructure* method.

The *restructure* method can be used even when there are no pending structural changes to repair or compact a *StorageDataSet* and its associated indexes. (See the *DataStoreConnection.copyStreams* method for another way of repairing damaged streams.)

Data type coercions

When the data type of a *Column* component in a *StorageDataSet* that is bound to a *DataStore* is changed, type coercions occur when going from one type to another. 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*.

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

Where the table values represent the following:

- 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.

Persistent column editing

Many *DataSets*, like *QueryDataSet* and *ProcedureDataSet*, obtain their structure and row data from a SQL server. But there is also the need to add columns to these *DataSets* or to create standalone tables. This can be accomplished with the JBuilder Column Designer. *DataSet* columns can be changed, deleted, moved, and added instantly at design time. The *DataStore* uses a mapping table to provide an illusion that each structural

operation has occurred. When all structural changes are complete, the Restructure button can be pressed.

When the UI Designer is activated while a *StorageDataSet* node is selected, the columns for that node are displayed in a grid on the design surface. A tool bar for adding, deleting, moving up, moving down, and restructuring the data set is displayed above the grid.

- Move Up and Move Down manipulate the column's preferred ordinal property.
- Add inserts a new column at the preferred ordinal of the highlighted column in the grid.
- Delete removes the column from the dataset.
- Restructure compiles the **this** component and launches a separate VM to perform a restructure of the DataStore associated with the data set. While the restructure is running, a dialog box is presented that shows the status of the restructure and has a Cancel button from which it can be aborted. Restructure is only available if the dataset's *store* property has been set to a *DataStore* or *DataStoreConnection* component (see "Using a DataStore with StorageDataSets" on page 5-2).

Understanding structure changes

Regardless of whether a DataStore is used, columns provided by a query or procedure must be merged with columns defined in code. When a DataStore is being used, this merge still must happen, followed by a merge between the resulting column list and the columns found in the data set in the store.

Ideally, columns in the DataStore are the same as those provided by the merge. But when developing an application, there are often changes, and even a production application sometimes has to change. You can continue to use a data set in a DataStore when the columns in the DataStore are not the same as those from the application/provider merge. The provide operation will automatically restructure the table after merging.

If you change a data set's structure in the Designer instead of editing the source code, the DataStore tracks the changes. This makes the merge between application/provider columns and DataStore columns easier. For instance, if you change the name of a calculated column through the Designer, you can continue to display and edit values in the column, because the DataStore can map the old column name to the new one. If you make the same change in code, the DataStore can only determine that one column was deleted and another was added. The DataStore will not recognize that the column was renamed. So the "deleted" column will not be displayed, and the "added" column will be empty and non-editable.

The following list provides more information for different types of data set structure changes.

- Insert column: The column will be visible but empty and not editable until the restructure is done.
- Delete column: The column will not be visible, but will still exist in the store until the restructure is done.
- Change column name: The name change takes effect immediately.
- Change column order: The column continues to be visible and editable. Until the restructure is done, there is a very small performance penalty in mapping between the column order specified in the application and the order that actually exists in the DataStore.
- Change column's data type: The column will be visible but will not be editable until the restructure is done.

Restructuring also packs the dataset and deletes indexes. The indexes will be rebuilt when needed.

Using the DataStore Explorer

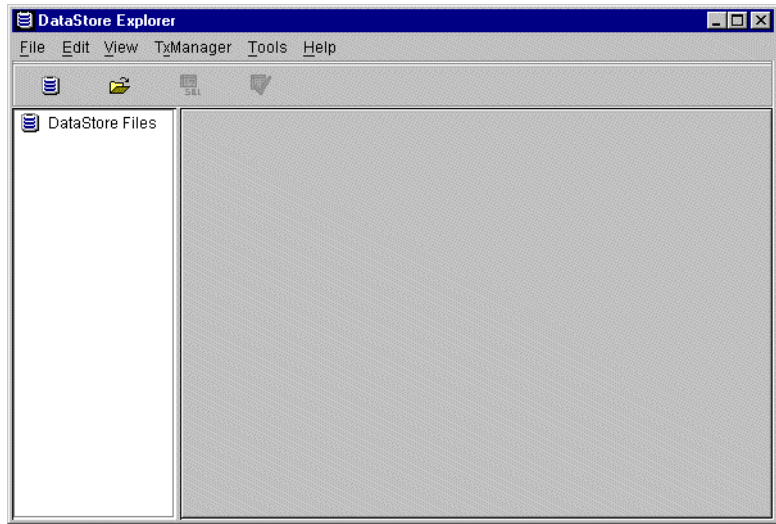
Using the all-Java DataStore Explorer, you can:

- Examine the contents of a DataStore. The DataStore's directory is shown in a tree control, with each table and its indexes grouped together. When a data stream is selected in the tree, its contents are displayed (assuming it's a file type like text file, GIF image, or table, for which the Explorer has a viewer).
- Perform many DataStore operations without writing code. You can create a new DataStore file, import delimited text files into datasets, import files as file streams, delete indexes, delete datasets or other data streams, and verify the integrity of the DataStore.
- Manage queries that provide data into datasets in the DataStore, edit the datasets, and save changes back to server tables.

Launching the DataStore Explorer

There are three ways to launch the DataStore Explorer:

- From JBuilder use the Tools | DataStore Explorer menu command.
- From the DataStore JDBC server (see "Running the DataStore JDBC server" on page 4-2) use the File | DataStore Explorer menu command.
- From the command line or a shortcut.

Figure 6.1 DataStore Explorer after launch

Starting the DataStore Explorer from the command line

DataStore Explorer requires the following JARs:

- datastore3.1.jar
- dx3.1.jar
- dbswing3.1.jar
- dbtools3.1.jar
- pthelp3.1.jar

The main class for the DataStore Explorer is *com.borland.dbtools.dsx.DataStoreExplorer*, and it takes one command-line argument for the location of the online help. The command line for starting the server (after setting the classpath) is:

```
java com.borland.dbtools.dsx.DataStoreExplorer -h=<onlineHelpDir>
```

Basic DataStore operations

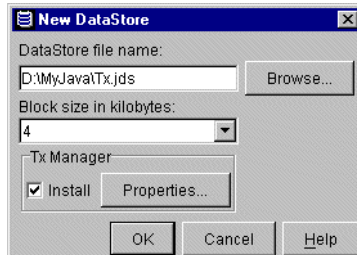
Most DataStore operations in the DataStore Explorer require a DataStore file. You can create a new one, or open an existing DataStore file. You may have more than one DataStore file opened at the same time.

Creating a new DataStore file

To create a new DataStore file:

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

Figure 6.2 New DataStore dialog box



- 2 Enter a name for the new DataStore file.
- 3 (Optional) Choose a block size other than the default 4KB.
- 4 Make sure the Install TxManager checkbox is as desired:
 - For a non-transactional DataStore, the checkbox should be empty
 - For a transactional DataStore, the checkbox should be selected. You can click Properties to set the transaction management properties for the new DataStore file.
- 5 Click OK. The store is created and opened in the DataStore Explorer.

Opening an existing DataStore file

To open an existing DataStore file:

- 1 Select File | Open or click the Open DataStore toolbar button. This opens the Java version of the standard File Open dialog box.
- 2 Choose the file to open and click Open.

The DataStore Explorer will keep track of the five most recently opened files. You can open them directly from the File menu.

Setting options for opening DataStore files

Transactional DataStore files require a user name. You may also want to open a DataStore file in read-only mode to temporarily bypass transaction

support, or to attempt to open a file that has been damaged. These settings are made by selecting View | Options to open the Options dialog box:

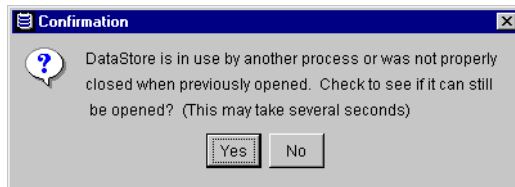
Figure 6.3 DataStore options dialog box



Opening a DataStore file that was not closed properly

If the DataStore file is already marked open—which will happen if the DataStore file was not closed properly—you will get the following dialog box, asking if you want to try and open the DataStore anyway:

Figure 6.4 DataStore marked open dialog box



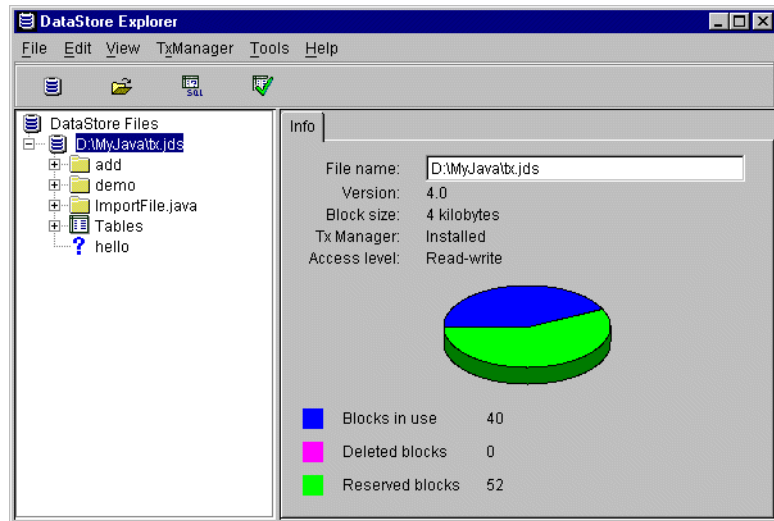
If this occurs:

- 1 Verify that there is no process that might still have the DataStore file open (in particular, check the Task Manager for any rogue `javaw` processes):
 - If there is no process that has the DataStore file open, click Yes to reopen the DataStore file.
 - If you click No, the DataStore Explorer responds with an error dialog (that the DataStore file is still marked open by another process) and the file will not be opened.
- 2 Attempting to reopen the file may take several seconds. If the DataStore file was not closed properly, another dialog box will inform you of this condition. Click OK to attempt to recover the DataStore file.
- 3 After successfully opening a DataStore file that was left marked open, a dialog box will give you the opportunity to verify the contents of the DataStore file. Click Yes to verify the DataStore contents, or No to skip the verification.

Viewing DataStore file information

When a DataStore file is opened, its directory is displayed in a tree control on the left. Each open DataStore file is a node directly off the root node. Information about the DataStore file is displayed in the viewer area on the right:

Figure 6.5 DataStore Explorer displaying DataStore file information



This information includes:

- The name of the DataStore file
- The DataStore file format version number
- The block size
- Whether the TxManager is installed; that is, whether the DataStore is transactional
- Whether the DataStore file was opened read-write or read-only
- 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 DataStores preallocate disk space to improve reliability)

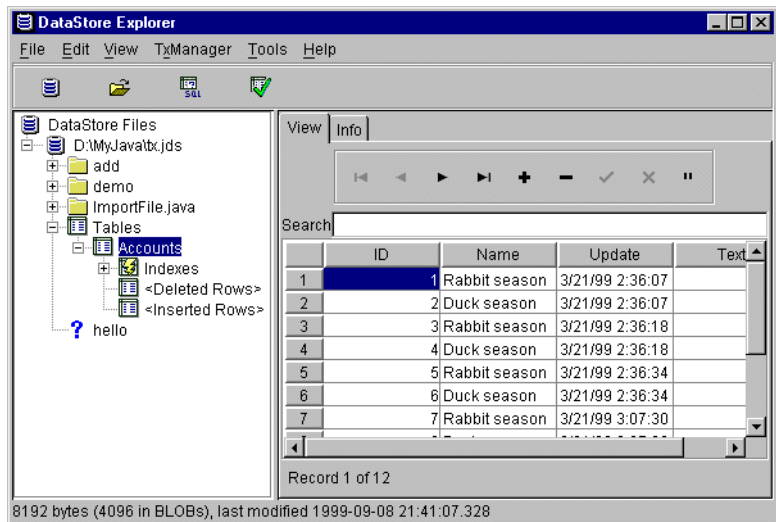
You can view this information at any time by selecting the node in tree that contains the DataStore file name.

Viewing stream contents

The DataStore directory is displayed in the tree control on the left, using forward slashes (“/”) in the stream names to synthesize a hierarchical structure. In addition, known stream types, such as tables and text files, are placed under a corresponding node in the tree. You may 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 will be displayed if there is an appropriate viewer. There is a built-in viewer for table streams:

Figure 6.6 DataStore Explorer displaying table stored in DataStore



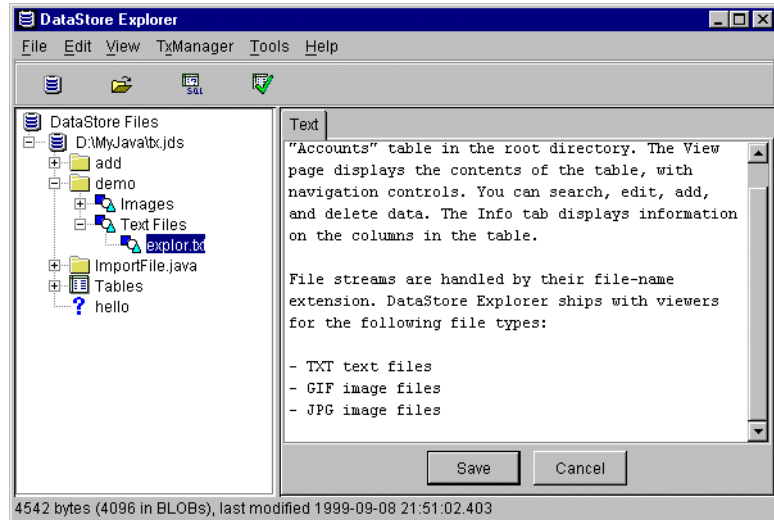
The figure shows a DataStore file with an “Accounts” table in the root directory. The View page displays the contents of the table, with navigation controls. You can search, edit, add, and delete data. The Info tab displays information on the columns in the table.

File streams are handled by their file-name extension. DataStore Explorer ships with viewers for the following file types:

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

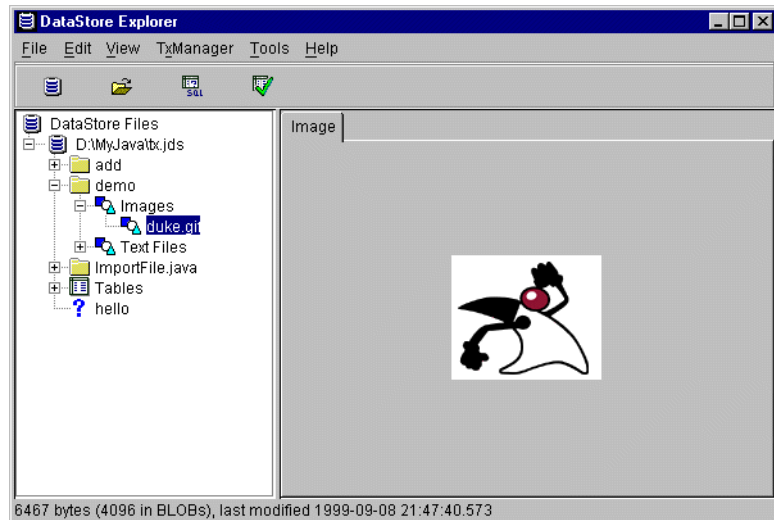
For example, here is the text file “demo/explor.txt”:

Figure 6.7 DataStore Explorer displaying text file stored in DataStore



This is the image file “demo/duke.gif”:

Figure 6.8 DataStore Explorer displaying image stored in DataStore



Renaming streams

Use this operation to rename a stream, or move a stream to another directory.

- 1 Select the stream to rename/move in the directory tree. (You cannot rename deleted streams; you must undelete them first.)
- 2 Select Edit | Rename
- 3 Type a new name in the Rename dialog box. You cannot use the name of another existing active stream.
- 4 Click OK.

Deleting streams

Deleting a stream marks the blocks it used as available. It is possible to undelete a stream, although some of the blocks may have been reclaimed by other streams. See “How DataStore reuses deleted blocks” on page 2-20 for more details.

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, it may be undeleted. You cannot undelete a stream if there is another active stream with the same name in the same directory, because you cannot have two streams with the same name.

Undeleting a stream is not guaranteed to recover all the data in the stream. See “How DataStore reuses deleted blocks” on page 2-20 for more details.

To undelete a stream:

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

Copying DataStore streams

You may use the DataStore Explorer to copy streams to another DataStore file, much like the *DataStoreConnection.copyStreams* method. While you cannot use this option to make copies of streams within the same DataStore file, the DataStore Explorer will automatically create a new DataStore file for you.

To copy streams:

- 1 Select Tools | Copy DataStore. This opens the Copy Streams dialog box:

Figure 6.9 Copy Streams dialog box



- 2 Specify the various *copyStreams* options, as summarized in the following table. (For more information, see “copyStreams parameters” on page 2-17.)

Table 6.1 Copy Streams options

Name	Description
Destination DataStore	The name of the DataStore file to copy to.
Destination directory	Names of copies of streams have their <i>Source directory</i> replaced with this; should be the same as <i>Source directory</i> if the name should not be changed.
Source directory	Stream name must begin with this to be pattern-matched; empty string to pattern-match all streams.
Source Pattern	Stream name pattern to match, with standard * and ? wildcard characters.

- 3 Click OK.

Verifying the DataStore

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

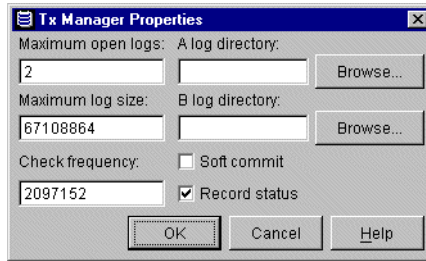
The entire store is verified and the results are displayed in the Verifier Log window. After you’ve closed the log window, you view it again by selecting View | Verifier Log.

Making the DataStore transactional

You can make a non-transactional DataStore transactional with the DataStore Explorer:

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

Figure 6.10 TxManager Properties dialog box



- 2 The dialog box contains default settings for the *TxManager* object:
 - You may change the default settings for maximum number of open log files (2), maximum log file size (64MB), and checkpoint frequency (2MB).
 - By default, one set of log files are written in the same directory as the DataStore file. To choose another directory, specify a different A log directory.
 - To maintain a second redundant set of log files, specify a B log directory.
 - You may enable Soft commit, which increases performance by not immediately forcing a disk write when a transaction is committed.
 - You may disable status logging for a slight performance improvement.
- 3 Click OK.

Modifying transaction settings

To modify transaction settings for a DataStore:

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

Removing transaction support

You can check if a DataStore is transactional by checking the TxManager menu. If the Enabled menu option is checked, the DataStore is transactional. If the menu option is disabled, the DataStore is not transactional.

To make a transactional DataStore non-transactional, select TxManager | Enabled. This immediately removes transaction support.

Closing DataStore files

When you are done with the DataStore file, you should close it to ensure that all changes are written properly. You do not have to close open DataStore files before exiting the DataStore Explorer; this is done automatically.

To close the current DataStore file, select File | Close. To close all open DataStore files, select File | Close all.

DataStore Explorer as a query console

When the DataStore is used as a persistent data cache (which means that the data isn't lost when the application is shut down), it's the application's job to contain the logic to manipulate the data. This logic may include code to connect to a server, SQL statements to load data from the server into datasets, and code to save updates back to the server, in addition to whatever presentation and manipulation of the data the application allows.

The DataStore Explorer can take the place of a simple application. Within the Explorer, you can define a connection to a database, define SQL queries against tables in that database, run the queries to produce datasets that are saved in the DataStore, edit the datasets, save your changes back to the database, and rerun queries to get the latest server data—all without writing code.

This same mechanism can be used to simply import data from another database into a DataStore. Because the query and connection information used to import the data is saved, you can easily reimport the data if desired.

Using DataStore Explorer to manage queries

This discussion of ways to store and execute queries may bring to mind the distinction between queries that run against a server to produce store datasets and queries against store tables that produce new tables (which may or may not be saved in a DataStore). In the former case, the DataStore

provides persistent storage for tables produced by querying a server; in the latter, the DataStore itself functions as a server and, like other servers, is accessed by running SQL queries through a JDBC connection.

Here we introduce a third, complementary use of the DataStore: as a place to store connection information and queries. These queries may be of either type just mentioned; they may get their data either from server tables or from datasets in a DataStore. In either case, the result of running a query through the DataStore Explorer is always another store table.

When using the Explorer to manage queries, it is important to understand the objects involved and how they are related:

- In a DataStore, you can define several connections to databases.
- Many queries can share a connection; they all select data from the tables in the same database.
- Each query creates one table. When you define and run the query in the Explorer, the resulting table is always saved in the DataStore.

The hierarchy is this:

- Many connections per DataStore
- Many queries per connection
- One table per query

The user interface for saving changes and refreshing data reflects this organization, as described in “Saving changes and refreshing data” on page 6-15.

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

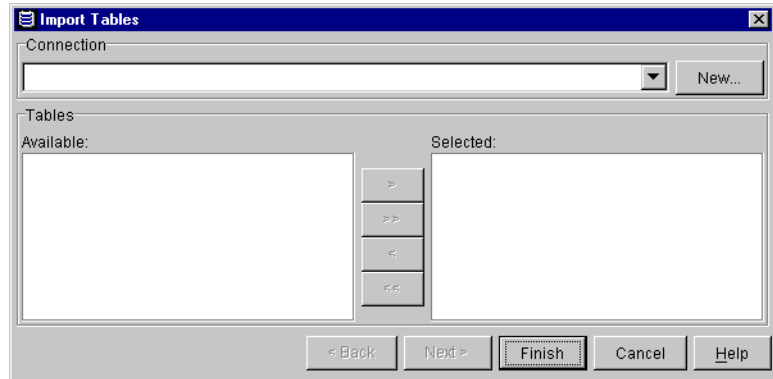
DataStore Explorer limitations

Keep in mind that although the DataStore Explorer can execute queries and save changes to dataset back to server tables, it has no application-specific logic. Presentation of data is limited to a simple tabular display. Data entry support and data validation that is usually provided by code in a data module won’t be performed. If you edit data in the Explorer, you won’t see any edit masks or pick lists, won’t be prevented from entering values that violate minimum or maximum constraints, and may be able to leave required fields empty or violate referential integrity constraints. (Constraints defined in the server will be enforced, but not until you attempt to save your changes.) The Explorer is useful for working with test data or making small, careful changes to production data, but is not a substitute for a data module written with the specific integrity requirements of its datasets in mind.

Creating and maintaining queries and connections

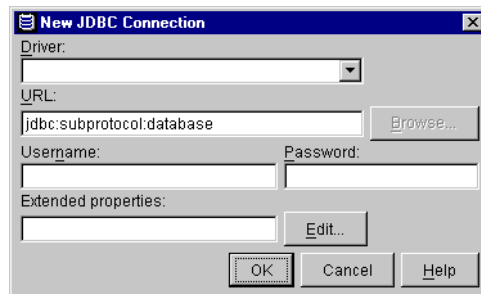
To use the DataStore Explorer to manage queries or to import a table from another database, you must have a DataStore open. Then, to define a query, select Tools | Import | Tables. The Import Tables dialog box opens:

Figure 6.11 First page of Import Tables dialog box



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

Figure 6.12 New JDBC Connection dialog box

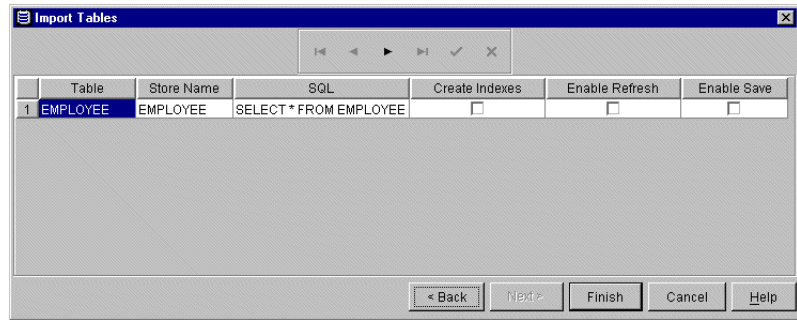


Enter the same parameters as you would in the Connection property editor for a database: JDBC Driver Name, URL, Username, and Password. You may also specify extended properties for the connection. When defining queries later, you can choose an existing connection or define a new one.

Once you have a connection to a database, a list of available tables are shown. 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 6.13 Second page of Import Tables dialog box



This page lists all the tables that will be imported:

- The original name of the table in the database to be imported.
- The name of the table stream that will be created in the DataStore, which defaults to the original name, but can be changed. It may include a “path”, as in “Data/Tutorial/Employee”. The Explorer’s tree pane will display this as a dataset named “Employee” in a folder named “Tutorial” which in turn is in a folder named “Data”.
- The SQL statement used to retrieve the data, which can be edited to change the fields, conditions, and grouping.
- Checkboxes to control whether indexes should be created, and whether to enable refresh and save from the Tools menu for the table stream.

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 go into “SYS/Queries”, and connections you create go into “SYS/Connections”. When you finish defining the first query by clicking OK, each table will have one row.

To maintain connections or queries, select the “Connections” or “Queries” table under the “SYS/DataStore Queries” branch in the Explorer 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 Navigator or type *Ctrl+Del*.
- Insert a new definition. To do so, press “+” on the Navigator or type *Ctrl+Ins*.

Fetching and editing data

Right after saving a new query, the DataStore Explorer will attempt to execute that query to fetch its data. You will see the tree pane of the Explorer update to show the newly imported table, using the store name you specified. After that, you can re-execute the query to refresh the data manually. Note that refreshing data will discard any unsaved changes.

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

View a table by selecting it in the Explorer’s tree. On the right side of the Explorer, you see the table in a grid. Choose the Info page to see the dataset’s column names and their data types. You can edit the table on the View page, but be sure you understand the risk to data integrity first.

After editing, you can save your changes or discard them. You discard changes by refreshing the dataset, and responding “Yes” to the warning about unsaved changes.

Saving changes and refreshing data

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

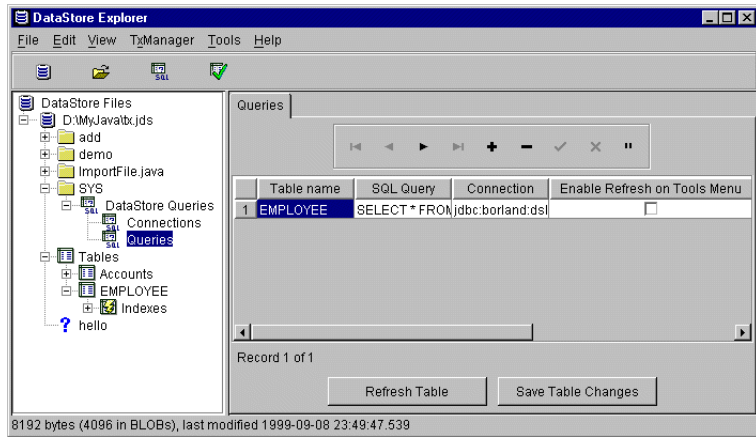
- Individual queries
- All the queries for a particular connection
- All the queries in all the connections stored in the DataStore

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” dataset. 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 datasets for which you’ve defined queries through the DataStore Explorer, select Tools | Refresh DataStore or Tools | Save DataStore Changes. These commands re-execute every query or save changes for every dataset with an associated query, for those queries that have their Enable Refresh on Tools Menu and Enable Save on Tools Menu options enabled. You can change these settings for each query in the “SYS/Queries” table.

Figure 6.14 Entries in the SYS/Queries table



Importing tables and files

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

Importing text files as tables

The contents of the text file must be in the delimited format that DataExpress exports to, and there must be a SCHEMA file with the same name in the directory to define the structure of the target dataset.

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

```
[ ]
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, 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 store name of the dataset to be created. Since this operation creates a dataset, not a file stream, you'll probably want to omit the extension from the store name.
- 3 Click OK.

Importing files

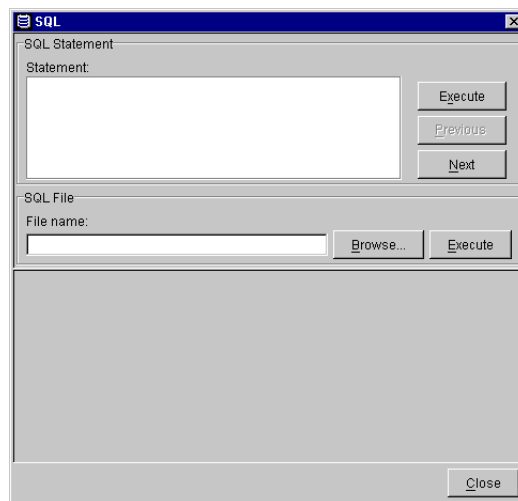
To import a file as a file stream,

- 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.

Executing SQL

You may execute arbitrary SQL statements with the currently selected DataStore file as the database. This requires that the DataStore file is transactional, or that it is opened in read-only mode. To execute SQL, select Tools | SQL or click the SQL toolbar button, which opens the SQL dialog box:

Figure 6.15 Entries in the SYS/Queries table



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

DataStore file operations

The DataStore Explorer also provides a few functions that do not have a direct analogue in the DataStore API.

Packing the DataStore file

Packing the DataStore file renames the existing file (by prepending "BackupX_of_", where X is an auto-incrementing number), and copies all the streams from the old file to the new copy of the current file.

To pack the DataStore, select Tools | Pack DataStore.

Upgrading the DataStore file

The DataStore Explorer will open older versions of the DataStore file format. The only available operation is to upgrade the file to the current version. Upgrading the DataStore file renames the existing file (by prepending "BackupX_of_", where X is an auto-incrementing number), and copies all the streams from the old file to a new version of the current file.

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

Deleting the DataStore file

You can use the DataStore Explorer to delete a DataStore file, and its transaction log files. To use this feature, you must open the DataStore file to delete. Then select Tools | Delete DataStore. You will get a confirmation dialog box. Click Yes to delete the DataStore file(s).

Optimizing DataStore applications

This chapter discusses issues that will help improve the performance, reliability, and size of DataStore applications. Unless otherwise specified, *DataStoreConnection* refers to either a *DataStoreConnection* or *DataStore* object used to open a connection to a DataStore file.

General usage recommendations

Here are a few items that concern usage for all types of DataStore applications.

Closing the DataStore

Be sure to call *DataStoreConnection.close* when the *DataStoreConnection* is no longer needed, or call *DataStore.shutdown* to close the DataStore file regardless of how many connections there may be. In particular:

- To guarantee proper closure, do not call *System.exit* until you have called *DataStore.shutdown*.

Closing a DataStore 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 100 milliseconds). If you directly exit the Java VM, the daemon thread may not have the opportunity to save the last set of changes. There is a small chance that a non-transactional DataStore may get corrupted.

A transactional DataStore is guaranteed to not lose data, but the transaction manager will roll back any uncommitted changes.

If a *DataStore* file is not closed properly, it will take several seconds to reset/recover the *DataStore* file. Proper closure means that the *DataStore* will reopen quickly every time.

Another benefit to closing *DataStoreConnection* objects is that when they are all closed, the memory allocated to the *DataStore* cache is released.

Close all *StorageDataSets* that have their *store* property set to a *DataStoreConnection* when you are done with them. This frees up *DataStore* resources associated with the *StorageDataSet* and allows the *StorageDataSet* to be garbage collected.

Controlling how often cache blocks are written to disk

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 *DataStores*. 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*.

Unlike other properties of *DataStore*, *saveMode* may 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 will change the behavior for all *DataStoreConnection* objects that access that particular *DataStore* file.

Tuning memory

You can tune the use of memory in a number of ways. Beware 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 *-Xms* option to specify a larger initial heap size.
- You might try increasing the *DataStore.minCacheBlocks* property, which controls the minimum number of blocks that are cached.

- 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.

Miscellaneous performance tips

- Setting the *DataStore.tempDirName* property, used by the query engine, to a directory on another (fast) disk drive may help.
- Try setting the *TxManager.checkFrequency* higher. Higher values will result in slower recovery, but why be pessimistic?
- For simple operations, DataExpress may be faster than JDBC/SQL. JDBC/SQL is probably faster for more complex queries.
- Write your applications with an eye toward efficient multithreading. Try to keep the number of monitor objects to a minimum, to take advantage of multi-processor systems.

DataStore 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 DataStore applications.

- *DBDisposeMonitor* (which automatically disposes of data-aware component resources when a container is closed) has a *closeDataStores* property. When **true** (the default) it will automatically close any DataStores attached to components it cleans up.

For example, if you drop a *DBDisposeMonitor* into a *JFrame* you're designing that contains dbSwing components attached to a DataStore, when you close the *JFrame* *DBDisposeMonitor* will automatically close the DataStore for you. This component is particularly handy when building simple applications to experiment with DataStore.

- *DBExceptionHandler*, dbSwing's version of JBCL's *ExceptionDialog*, has an Exit button (can be hidden via a property setting, but visible is by default). Clicking this button which will also automatically close any open DataStores it can find. *DBExceptionHandler* is the default dialog box displayed by dbSwing components when an exception occurs.

Using data modules

When using a DataStore with a *StorageDataSet*, it is strongly recommended that they all be grouped inside data modules. Any references to these *StorageDataSets* should be made through *DataModule* accessor methods such as *businessModule.getCustomer()*. The reason for this

is that 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 *DataStore* itself, the classes that implement the event listener interfaces are not. By instantiating the *StorageDataSet* with all event listener settings, constraints, calculated fields, and filters implemented with events, they will be properly maintained at run time and design time.

Optimizing transactional applications

The increased reliability and flexibility gained from using transactional *DataStores* 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. This property must be set to **true** before the connection is open. For JDBC connections, it is controlled by the *readOnly* property of the *java.sql.Connection* object (returned by the *java.sql.DriverManager.getConnection* and *com.borland.dx.dataset.sql.Database.getJdbcConnection* methods).

Read-only transactions work by simulating a snapshot of the *DataStore*. Only data from transactions committed at the point the transaction started are seen in this snapshot (otherwise, the connection would have to see if there are pending changes and roll them back whenever it accesses the data). A snapshot is taken when the *DataStoreConnection* is opened, and is refreshed every time its *commit* method is called.

Another benefit of read-only transactions is that they are not blocked by writers (or other readers). Both reading and writing normally require a stream lock. But because it uses a snapshot, a read-only transaction does not need a lock, and therefore would not be blocked by a writer that has a lock on a stream that it is modifying.

You can further optimize the application by specifying a *readOnlyTxDelay*. By default, this property is zero, which means that whenever a read-only transaction starts or is refreshed, a new snapshot is taken. 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.

Using soft commit mode

If you enable soft commit mode through the *TxManager's* *softCommit* property, the transaction manager will force disk writes for the purposes of crash recovery, but not guarantee transaction commits. This will improve performance, but make you slightly more susceptible to power loss, operating system crashes, and hardware failures.

Very recently committed transactions—depending on the operating system, but in general those committed within approximately the last second—are not guaranteed to be written. This is not the case when soft commit mode is disabled (the default), when committed changes are written immediately.

Transaction log files

The location of the transaction log files is controlled by the *TxManager's* *ALogDir* and *BLogDir* properties.

- Do not place the files on slow disk drives, including network drives. (Putting the log files on a floppy disk is a very bad idea.)
- You may duplex the log files by specifying a *BLogDir* in addition to an *ALogDir* to increase reliability; if either copy is damaged or lost, you have a backup.
- Specify the directory for the *BLogDir* on a different physical drive than the *ALogDir*. This may reduce some of the performance penalty for writing two separate log files (both drives can write “simultaneously”), and increases reliability because it’s unlikely that both drives will fail.

Disabling status logging

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

Pruning deployed resources

When deploying a DataStore application, you can exclude certain classes and graphics files that are not used. In particular:

- If DataStore is used without transaction support the following classes can be excluded:
 - *com.borland.datastore.Tx*.class*

- If *DataStore* is used without using the JDBC driver, exclude:
 - *com.borland.datastore.Sql*.class*
 - *com.borland.jdbc.**
 - *com.borland.sql.**
- If *DataExpress* is used and the *StorageDataSet.store* property is always set to an instance of *DataStore* or *DataStoreConnection*, exclude:
 - *com.borland.dx.memorystore.**
- If *TableDataSet* is used, but not *QueryDataSet*, *QueryProvider*, *StoredProcedureDataSet* or *StoredProcedureProvider*, exclude:
 - *com.borland.dx.sql.**
- If *DataExpress* is not using any visual components from the JBCL or dbSwing libraries, exclude:
 - *com.borland.dx.text.**
- If *com.borland.dx.dataset.TextDataFile* is not used, exclude:
 - *com.borland.jb.io.**
 - *com.borland.dx.dataset.TextDataFile.class*
 - *com.borland.dx.dataset.SchemaFile.class*

Troubleshooting

Debugging DataStore applications

Set the *saveMode* property to 2 when debugging a non-transactional DataStore. The debugger stops all threads when single-stepping or when breakpoints are hit. If the *saveMode* property is not set to 2, this keeps the DataStore daemon thread from saving modified cache data. For more information, see “Controlling how often cache blocks are written to disk” on page 7-2.

Verifying DataStore contents

If you suspect that cache contents were not properly saved on a non-transactional DataStore, you can verify the integrity of the file with the DataStore Explorer. See “Verifying the DataStore” on page 6-9 for more information.

There is also a *borland.datastore.StreamVerifier* class with public static *verify()* methods that can verify a single stream or all streams in DataStore. For more information, see the *DataExpress Component Library Reference*.

Note that transactional DataStores have automatic crash recovery when they open. You never need to verify them.

If problems are encountered, use the DataStore Explorer (see “Copying DataStore streams” on page 6-8) or the *DataStoreConnection.copyStreams* method to repair the damage.

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 *DataStore* use these *CollationKey* classes to generate sortable sort keys if a non-US locale is being used. When Sun changes the format of these *CollationKeys*, 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 will manifest themselves in the following ways:

- Locate and query operations may not find records that they should.
- Viewing a table in secondary index order (by setting the *StorageDataSet.sort* property) may 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 will also drop all the secondary indexes.

Saving log files

As old log files are no longer needed for active transactions or crash recovery, they are automatically deleted. Old log files can be saved by listening to the *DataStore.response* event for a *ResponseEvent.DROP_LOG* notification. At that point, you can copy out the log file to another location before it is deleted, or *cancel* the event to prevent the deletion of the log file.



Specifications

The following specifications apply to the 3.1 version of the DataStore file format.

DataStore file capacity

Minimum block size: 1 KB

Maximum block size: 32 KB

Default block size: 4 KB

Maximum DataStore file size: 2 billion (2G) 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: 4 billion (4G)

Maximum row length: 1/3 block size. Strings, objects, and inputstreams that exceed the inline size (default 64 bytes) are stored as BLOBs instead of occupying space in the row.

Maximum BLOB size: 2 GB each

Maximum file stream size: 2 GB each

DataStore stream names

Directory separator character: /

Maximum length: 192 bytes

- Best case (all single-byte character set): 192 characters
- Worst case (all double-byte character set): 95 characters (one byte lost to indicate DBCS)

Reserved names:

- SYS/Connections
- SYS/Queries

B

Changes from previous versions

This appendix highlights major changes since version 3.0.

API changes

In the *com.borland.datastore* package:

- *DataStore* transactions now support the repeatable read isolation level, which uses row locks.
- The *TxManager*'s *softRecovery* option has been changed to a more robust *softCommit* option, which guarantees crash recovery but not very recently committed transactions.

In the *com.borland.datastore.jdbc* package:

- The *ClientConnection* interface was replaced by the *ServerConnection* class.
- The *AcceptClient* class was changed to *DataStoreServer*.
- *DataStoreServer* uses an event listener architecture to notify the server application of changes. As a result:
 - The *ClientListener* interface has been removed.
 - The *ServerStatus* and *ServerStatusListener* interfaces and the *ServerStatusEvent* class have been added.

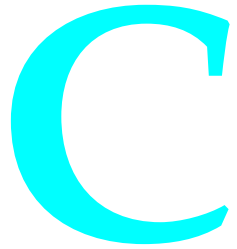
Deployment changes

- The JAR file names now use version number 3.1
- JBCL and JGL are no longer needed.

- The main class name for the DataStore Server has changed from *com.borland.jbuilder.dsserver.Server* to *com.borland.dbtools.dsserver.Server*.
- The main class name for the DataStore Explorer has changed from *com.borland.jbuilder.dsx.DataStoreExplorer* to *com.borland.dbtools.dsx.DataStoreExplorer*.

Changes to the DataStore Explorer

- You may now open more than one DataStore file at a time.
- There is a Close all item in the File menu.
- The File menu tracks the last five most recently opened files.
- There is a dialog box (select View | Options) to specify your user name and whether to open DataStore files read-only.
- The functionality of the New SQL Query action have been replaced with two separate items:
 - A SQL option provides a way to execute SQL statements directly on the DataStore.
 - To use the DataStore Explorer as a query console, use the Tools | Import | Tables menu option to create connections and queries, and import data.



SQL reference

This is a description of the SQL dialect supported by the query engine for the DataStore JDBC driver. The methods in the JDBC API that take an SQL statement are:

```
int Statement.executeUpdate(String query)
ResultSet Statement.executeQuery(String query)
PreparedStatement Connection.prepareStatement(String query)
```

Each query string must contain exactly one SQL statement. Use *executeQuery* for SELECT statements, and *updateQuery* for all other queries. If there are parameter markers in the query, a *PreparedStatement* must be used to pass the actual parameters.

Lists in syntax notation

A number of syntax element names ending with the word “list”; for example:

```
<select item list>
```

are not defined. These are to be read as a comma-separated list with at least one element; in this case:

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

Data types

The following data types are supported in the DataStore:

- SHORT
- INT
- LONG
- BIGDECIMAL
- FLOAT
- DOUBLE
- STRING
- BOOLEAN
- INPUTSTREAM

In SQL these types can be specified using those names or by using synonym type names, which are more portable to other SQL engines. The possible types are given by the following table:

Table C.1 Data types supported by DataStore

DataStore type	Synonyms	Attributes	Bytes of storage	Precision	Range
SHORT	SMALLINT		1–3	exact	–32768...32767
INT	INTEGER		1–5	exact	–2147483648... 2147483647
LONG	BIGINT		1–9	exact	–9223372036854775808... 9223372036854775807
BIGDECIMAL	DECIMAL, NUMERIC	prec,dec	1–32	exact	–10^72...10^72
FLOAT	REAL		1–5	approximate	{+/-}1.4E–45...3.4E38
DOUBLE	DOUBLE PRECISION		1–9	approximate	{+/-}4.9E–324...1.8E308
STRING	VARCHAR, CHAR	prec,inline	1–MAX_INT	n/a	
BOOLEAN	BIT		1	n/a	
INPUTSTREAM	BINARY		1–MAX_INT	n/a	
OBJECT		JavaType	1–MAX_INT	n/a	

For some types additional attributes may be applied. For strings a precision and an inline length may be specified. The precision is the maximum number of bytes allocated to such a string, and the inline length is the maximum number of bytes a string can take before the string is treated as a large object (which yields slower performance) similar to the BINARY types. The default inline value is 64.

Example A string with a maximum size of 30 bytes, and any string over 10 bytes is stored in a separate stream for large objects:

```
CHAR(30,10)
```

A string with a maximum size of 30 bytes, never inlined (precision is less than default inline value of 64):

```
CHAR(30)
```

A string with no length limit, use default inline size:

```
CHAR
```

A *BigDecimal* with 2 decimals and space for at least 5 significant digits:

```
DECIMAL(5,2)
```

A *BigDecimal* with 0 decimals and space for at least 4 significant digits:

```
NUMERIC(4)
```

A *BigDecimal* with 0 decimals and space for at least 72 significant digits:

```
NUMERIC
```

Any Java object that is serializable:

```
OBJECT
```

Only Java strings using Java serialization:

```
OBJECT(java.lang.String)
```

Literals

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

Table C.2 DataStore SQL literal values

Type	Details	Examples
String	Strings are enclosed in single quotes. The single quote character is represented by two consecutive single quotes.	'Hello' 'don't'
Binary number	A binary or hexadecimal sequence enclosed in single quotes and preceded by the letter B or X	B'1011001' X'F08A' X'f777'
Exact numeric	A number that may contain a decimal point	8 2. 15.7 .9233
Approximate numeric	A number followed by the letter E, followed with an optionally signed integer	8E0 4E3 0.3E2 6.2E-72
Date	Written in the format: DATE 'yyyy-mm-dd'	DATE '1999-03-23'
Time	24-hour time written in the format: TIME 'hh:mm:ss'	TIME '00:46:55'
Timestamp	Written in the format: TIMESTAMP 'yyyy-mm-dd hh:mm:ss'	TIMESTAMP '2001-12-31 13:15:45'

There is no literal syntax for the OBJECT data type.

Keywords

This list contains all identifiers reserved for keywords in this version of the DataStore SQL engine. All keywords are case-insensitive; for example, `select`, `SELECT`, and `SeLeCT` are all considered to be the keyword `SELECT`.

Note that not all SQL-92 keywords are treated as a keyword by the DataStore SQL engine. For maximum portability do not use identifiers that are treated as keywords in any SQL dialect.

Table C.3 Keywords for DataStore SQL engine

ALL	DROP	OR
ALTER	ELSE	ORDER
AND	END	POSITION
AVG	ESCAPE	PRECISION
AS	EXTRACT	PRIMARY
ASC	FALSE	REAL
BETWEEN	FLOAT	SECOND
BINARY	FOR	SELECT
BIT	FROM	SET
BOTH	GROUP	SMALLINT
BY	HAVING	SUBSTRING
CASE	HOURL	SUM
CAST	IN	TABLE
CHAR	INDEX	THEN
CHAR_LENGTH	INSERT	TIME
CHARACTER	INTO	TIMESTAMP
CHARACTER_LENGTH	INT	TIMEZONEHOUR
COLUMN	INTEGER	TIMEZONEMINUTE
COUNT	IS	TRAILING
CREATE	KEY	TRIM
CURRENT_DATE	LEADING	TRUE
CURRENT_TIME	NOT	UNIQUE
CURRENT_TIMESTAMP	LIKE	UNKNOWN
DATE	LOWER	UPDATE
DAY	MAX	UPPER
DEC	MIN	VALUES
DECIMAL	MINUTE	VARCHAR
DEFAULT	MONTH	VARYING
DELETE	NOT	WHEN
DESC	NULL	WHERE
DISTINCT	NUMERIC	YEAR
DOUBLE	ON	

Identifiers

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

- First character must be a letter as recognized by the *java.lang.Character* class.
- The rest of the characters must be either a letter, digit, underscore (`_`), or a dollar sign (`$`).
- Keywords cannot be used as identifiers.

Quoted identifiers may contain any character string including spaces, symbols, keywords, etc.

Example Valid identifiers:

```
customer    // treated as CUSTOMER
Help_me     // treated as HELP_ME
"Hansen"    // treated as Hansen
"DATE"      // treated as DATE
" "         // treated as a single space
```

Invalid identifiers:

```
_order      // must start with a character
date        // date is a keyword
borland.com // dots are not allowed
```

This list denotes the very same identifier, they are all treated as “ORDER”:

```
order
Order
oRDeR
"ORDER"
```

Expressions

Expression are used throughout the SQL language. They contain several infix operators, and a few prefix operators. Operator precedence from strongest to weakest is:

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

Syntax

```

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

<conditional expression> ::=
    <expression> OR <expression>
  | <expression> AND <expression>
  | NOT <expression>
  | <scalar expression> <compare operator> <scalar expression>
  | <scalar expression> [NOT] BETWEEN <scalar expression>
  | <scalar expression> [NOT] LIKE <scalar expression> [ESCAPE <scalar expression> ]
  | <scalar expression> [NOT] IS { NULL | TRUE | FALSE | UNKNOWN }

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

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

<concat> ::=  ||

<function> ::=
    <substring function>
  | <position function>
  | <trim function>
  | <extract function>
  | <lower function>
  | <upper function>
  | <char length function>
  | <current date function>

<current date function> ::=
    CURRENT_DATE
  | CURRENT_TIME
  | CURRENT_TIMESTAMP

```

Example

Select 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

```

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 Select all the orders where a customer has orders between 3 and 7 items of the same kind:

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

IS

The IS predicate is defined to test expressions. Any expression may evaluate to the value NULL, but conditional expressions may evaluate to one of the three values: TRUE, FALSE, UNKNOWN. UNKNOWN is equivalent with NULL for conditional expressions. Note that for a SELECT query with a WHERE clause, only rows that evaluate to TRUE will be included. If the expression evaluates to FALSE or UNKNOWN it will not be 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 }

Example

```
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 may include the special “wildcard” characters _ and % where:

- An underscore (_) will match any single character
- A percent character (%) will match any sequence of n characters where $n \geq 0$

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>

Example Item LIKE '%shoe%'

will evaluate to true if Item contains the string “shoe” anywhere inside it.

Item LIKE 'S_'

will evaluate to true if Item is exactly 3 characters long, starting with the letter “S”.

Item Like '%*%' ESCAPE '*'

will evaluate to true if Item ends with the percent character. The * is defined to escape the two special characters; if it precedes one, the special character is treated as a normal character in the pattern.

Functions

Note that functions that act on strings will work for strings of any length. Large strings are stored as blobs, so you may want to define large text fields as CHAR to enable searches.

CHAR_LENGTH and CHARACTER_LENGTH

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

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

CURRENT_DATE, CURRENT_TIME, and CURRENT_TIMESTAMP

These SQL functions will yield the current date and/or time. If the functions are placed more than once in a statement, they will all yield the same result when the statement is executed.

Example SELECT * from Returns where ReturnDate <= CURRENT_DATE

EXTRACT

The SQL EXTRACT function is able to extract parts of date and time values. The expression may be a DATE, TIME or TIMESTAMP value.

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

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

Example

```
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 will simply yield the given string, converted to all-lowercase or all-uppercase.

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

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

POSITION

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

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

Example

```
POSITION('BCD' IN 'ABCDEFGH')    // yields 2
POSITION(' ' IN 'ABCDEFGH')      // yields 1
POSITION('TAG' IN 'ABCDEFGH')    // yields 0
```

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 where the substring is taken, where 1 indicates the first character. If the FOR part is present it indicates the length of the resulting string.

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

Example

```

SUBSTRING('ABCDEFGH' FROM 2 FOR 3) // yields 'BCD'
SUBSTRING('ABCDEFGH' FROM 4)      // yields 'DEFG'
SUBSTRING('ABCDEFGH' FROM 10)     // yields ''
SUBSTRING('ABCDEFGH' FROM -6 FOR 3) // yields 'ABC'
SUBSTRING('ABCDEFGH' FROM 2 FOR -1) // raises an exception

```

TRIM

The SQL TRIM function is able to remove 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 will be 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>

Example

```

TRIM(' Hello world ') // yields 'Hello world'
TRIM(LEADING '0' FROM '00000789.75') // yields '789.75'

```

Statements

The DataStore JDBC driver supports a subset of the ANSI/ISO SQL-92 standard. In general, it provides:

- Some Data Definition Language for managing tables and indexes, but no schema, domain, views, or security elements.
- Data manipulation and selection with INSERT, UPDATE, DELETE, and SELECT; but no cursors.
- No subqueries or JOINS (equi-joins are supported).

Syntax <SQL statement> ::=
 <data definition statement>
 | <data manipulation statement>

<data definition statement> ::=
 <create table statement>
 | <alter table statement>
 | <drop table statement>
 | <create index statement>
 | <drop index statement>

<data manipulation statement> ::=
 <select statement>
 | <delete statement>
 | <insert statement>
 | <update statement>

CREATE TABLE

This statement will create a table in the DataStore. A column name and data type must be defined for each column.

Optionally, a default value may be specified for each column, along with uniqueness constraints.

Syntax <create table statement> ::=
 CREATE TABLE <table name> (<table element list>)

<table element> ::=
 <column name> <data type>
 [DEFAULT <default value>]
 [NOT NULL]
 [PRIMARY KEY]
 [UNIQUE]

```

<default value> ::=
    <literal>
    | <current date function>

<table name> ::=
    <SQL identifier>

<column name> ::=
    <SQL identifier>

```

Example

```

CREATE TABLE Orders ( CustId INTEGER PRIMARY KEY, Item VARCHAR(30),
    Amount INT, OrderDate DATE DEFAULT CURRENT_DATE)

```

ALTER TABLE

This statement adds and removes columns in a table in the DataStore.

Syntax

```

<alter table statement> ::=
    ALTER TABLE <table name> <column change list>

<column change> ::=
    <add column element>
    | <drop column element>

<add column element> ::=
    ADD [COLUMN] <table element>

<drop column element> ::=
    DROP [COLUMN] <column name>

```

The optional COLUMN keyword is included for SQL compatibility; it has no effect.

Example

```

ALTER TABLE Orders Add ShipDate DATE, DROP Amount

```

CREATE INDEX

This statement will create an index for a table in the DataStore. Each column may be specified to be ordered in ascending or descending order. Default is ascending order.

Syntax

```

<create index statement> ::=
    CREATE [UNIQUE] [CASEINSENSITIVE] INDEX <index name> ON <table name>
    ( <index element list> )

<index name> ::=
    <SQL Identifier>

```

```
<index element> ::=
    <column name> [DESC | ASC]
```

Example This will generate a non-unique, case-sensitive, ascending index on the column “ITEM” of the table “ORDERS”:

```
CREATE INDEX OrderIndex ON Orders (Item ASC)
```

SELECT

SELECT statements are used to retrieve data from one or more tables. The optional keyword DISTINCT eliminates duplicate rows from the result of a SELECT statement. The keyword ALL, which is the default, will get all rows including duplicates. The data may optionally be grouped or sorted.

Syntax

```
<select statement> ::=
    SELECT [ ALL | DISTINCT ] * | < item list>
    FROM   <table reference list>
    [ WHERE <expression> ]
    [ GROUP BY <column reference list> ]
    [ HAVING <expression> ]
    [ ORDER BY <order item list> ]
```

```
< item> ::=
    <expression> [ [AS] <column name> ]
```

```
<table reference> ::=
    <table name> [ [AS] <correlation name> ]
```

```
<column reference> ::=
    [ <table qualifier> . ] <column name>
```

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

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

Example

```
SELECT * FROM Orders WHERE Item = 'Shorts'
```

GROUP BY and HAVING

Aggregate functions can be used to calculate summary values from the data in a table. The WHERE clause (if present) will limit the number of rows included in the summary. If no GROUP BY clause is present, a summary for the whole table is calculated. If a GROUP BY clause is present, a summary will be computed for each unique set of values for the columns listed in the GROUP BY. Then if the

HAVING clause is present, it will filter out complete groups given the conditional expression in the HAVING clause.

Summary queries have additional rules for 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 statement> ::=
    SELECT [ ALL | DISTINCT ] * | < item list>
    FROM <table reference list>
    [ WHERE <expression> ]
    [ GROUP BY <column reference list> ]
    [ HAVING <expression> ]
    [ ORDER BY <order item list> ]
```

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

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

Example

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

yields a single row with the total value of all orders

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

yields a single row with the number of orders where Amount is non-NULL for the customer 123

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

yields a set of rows, with the sum of the value of all orders grouped by customers for the customers with an ID number less than 200.

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

yields a set of big customers with the value of all their orders

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

Illegal: nested aggregators present

```
SELECT CustId, SUM(Amount* Price) GROUP BY Amount
```

Illegal: the column `CustId` is referenced in the select item list, but is not present in the GROUP BY reference list.

ORDER BY

The output of a SELECT statement is usually a set of unordered rows of data. The ORDER BY clause can be used to sort the data before it is retrieved. Each ordering factor can be:

- An ordinal number identifying an output column; the first column is ordinal number 1.
- A output column name.
- An arbitrary expression.

Specify DESC to order in descending order. If neither ASC or DESC is specified ASC (ascending) is assumed.

Syntax

```
<select statement> ::=
    SELECT [ ALL | DISTINCT ] * | < item list>
    FROM  <table reference list>
    [ WHERE <expression> ]
    [ GROUP BY <column reference list> ]
    [ HAVING <expression> ]
    [ ORDER BY <order item list> ]
```

```
<order item> ::= <order part> [ ASC | DESC ]
```

```
<order part> ::=
    <integer literal>
    | <column name>
    | <expression>
```

Example

```
SELECT Item FROM Orders ORDER BY 1 DESC
```

orders by the first column, the Item column.

```
SELECT CustId, Amount*Price+500.00 AS CALC FROM Orders ORDER BY CALC
```

orders by the calculated column CALC

```
SELECT CustId, Amount FROM Orders ORDER BY Amount*Price
```

orders by the expression given

```
SELECT CustId FROM Orders ORDER BY 2
```

Illegal: ordinal outside the range

INSERT

An INSERT statement will insert rows into a table in the DataStore. A list of columns with associated values are listed in the INSERT statement. Columns that are not listed in the statement will be set to their default values. The values given are either a list of expressions or a SELECT expression. A SELECT expression has the same syntax as a SELECT statement, except that an ORDER BY clause is not accepted. The resulting set of rows from the SELECT statement will be inserted in the target table.

Syntax

```
<insert statement> ::=
    INSERT INTO <table name> ( <column name list> )
        [ <table expression> | DEFAULT VALUES ]

<table expression> ::=
    VALUES ( <scalar expression list> )
    | <select statement>

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

Example This statement should be used in connection with a *PreparedStatement* in JDBC. It will insert one row each time it is executed. The columns not mentioned will be set to their default value. If a column doesn't have a default value it will be set to NULL.

```
INSERT INTO Orders (CustId, Item) VALUES (?,?)
```

This statement will find all the orders from the customer with CustId = 123, and insert the Item of these orders into the table ResTable.

```
INSERT INTO ResTable SELECT Item from Orders
    WHERE CustId = 123
```

UPDATE

An UPDATE statement is used to modify existing data. The columns that are changed by the statement are listed explicitly. All the rows where the WHERE clause evaluates to TRUE are changed. If no WHERE clause is given all rows in the table are changed.

Syntax <update statement> ::=
 UPDATE <table name> SET <update assignment list>
 [WHERE <expression>]

<update assignment> ::=
 <column reference> = <update expression>

<update expression> ::=
 <scalar expression>
 | DEFAULT
 | NULL

Example All orders from the customer 123 are changed to orders from the customer 500:

```
UPDATE Orders SET CustId = 500 WHERE CustId = 123
```

Increase the amount of all orders in the table:

```
UPDATE Orders SET Amount = Amount + 1
```

Reprice all underwater stock options to 2.25:

```
UPDATE Options SET Price = 2.25 WHERE Price > 2.25
```

DELETE

A DELETE statement will delete rows from a table in the DataStore. If no WHERE clause is given, all the rows will be deleted. Otherwise only the rows that match the WHERE expression will be deleted.

Syntax <delete statement> ::=
 DELETE FROM <table name>
 [WHERE <expression>]

Example DELETE FROM Orders WHERE Item = 'Shorts'

DROP INDEX

This statement will delete an index from a table in the DataStore.

Syntax <drop index statement> ::=
 DROP INDEX <index name> ON <table name>

Example This will delete the index “ORDERINDEX” on the table “ORDERS”:

```
DROP INDEX OrderIndex ON Orders
```

DROP TABLE

This statement will delete a table and its indexes from the DataStore.

Syntax <drop table statement> ::=
 DROP TABLE <table name>

Example DROP TABLE Orders

Index

A

- access 2-12
 - DataStore tables and JDBC 3-17
 - multiple users and 2-14, 4-1
 - remote 4-1
- accessor methods 7-3
- adding data rows
 - to DataStore tables C-13, C-15
- aggregate functions C-13
- allocation
 - DataStore streams 2-20
- ALogDir property 3-9, 7-5
- ALTER TABLE statement C-12
- appendRow method 3-5
- applications
 - creating for offline editing 5-2
- ASC keyword C-15
- ascending sort order C-15
- auto-commit 3-17
- automatic crash recovery 3-6

B

- backing up DataStore files 2-18
- BETWEEN operator C-7
- blocked connections 4-5
- blockSize property 2-14
- BLogDir property 3-9, 7-5
- boilerplates 2-12
- bypassing transaction support 3-10
- byte arrays 2-16

C

- cache 7-1
 - controlling disk writes 7-2
 - DataStore components as 6-11
 - saving for DataStore 8-1
- case conversions C-9
- changing data
 - in DataStore components 3-11
 - in persistent columns 5-7
 - with SQL statements C-16
- changing transaction settings 3-8, 6-10
- CHAR_LENGTH function C-8
- CHARACTER_LENGTH function C-8
- closeDirectory method 2-11

- closing
 - connections 2-3
 - DataStore components 3-14, 6-11, 7-1
 - file streams 2-15
- coercions 5-7
- CollationKey classes 8-2
- commit method 3-11
- commit mode 7-5
- commits 3-12, 7-5
- connections
 - blocked 4-5
 - DataStore component 2-3, 3-2
 - referencing DataStore 2-14
 - remote drivers for 4-1
 - transaction processing and 3-11, 3-20
- constants 2-9
- contacting borland.com 1-3
- controlling disk writes 7-2
- Copy Streams dialog box 6-9
- copying DataStore components 2-12
- copying DataStore streams 2-16, 6-8
- copyStreams method 2-16, 2-23
- crash recovery 3-6, 7-5
- CREATE INDEX statement C-12
- CREATE TABLE statement C-11
- createFileStream method 2-13
- creating applications
 - for offline editing 5-2
- creating file streams 2-13
- creating queries C-1
- creating standalone tables 5-7
- creating table streams 3-3
- creating transaction log files 3-8
- CURRENT_DATE function C-8
- CURRENT_TIME function C-8
- CURRENT_TIMESTAMP function C-8

D

- daemons 7-1, 7-2
- data cache 7-1
 - controlling disk writes 7-2
 - persistent 6-11
 - saving for DataStore 8-1
- data modules
 - DataStore components and 7-3
- data sets
 - DataStores and 5-4

- data streams 2-12
- data type coercions 5-7
- data types C-1
- database tutorials
 - accessing DataStores with JDBC 3-18
 - controlling transactions 3-12
 - embedding databases 3-2
 - offline editing 5-2
- databases 3-1
 - providing functionality for 1-1
- DataExpress package
 - importing 3-2
- DATASET_EXISTS error code 2-14
- DataSetException class 2-4
- DataStore component
 - as data cache 6-11
 - as embedded database 3-1
 - bypassing transaction support for 3-10
 - changing transaction settings 3-8, 6-10
 - closing 7-1
 - copying 2-12, 2-16, 6-8
 - enabling transaction support for 3-6, 3-7, 6-9
 - exception handling 2-4
 - getting contents 2-6, 2-8
 - literals supported C-3
 - logging changes 3-8
 - mapping table for 5-7
 - removing transaction support 3-10, 6-10
 - restructuring 5-6, 5-8
 - supported data types C-1
 - troubleshooting tips 8-1
 - tutorial for serializing 2-2
 - tutorial for usage 5-2
 - usage overview 1-1
 - usage recommendations 7-1
- DataStore connections 2-3, 3-2
 - referencing 2-14
- DataStore Explorer 6-1
 - as query console 6-11
 - file operations with 6-18
 - hierarchical views 6-4, 6-5
 - importing with 6-16
 - limitations as query console 6-12
 - loading TXManager 6-9
 - managing queries with 6-12, 6-14
 - overview 1-3, 6-2
 - saving queries with 6-15
 - starting 6-1
 - validating contents 6-9
- DataStore files 2-1, 6-18
 - backing up 2-18
 - capacity A-1
 - closing 6-11
 - creating 2-3, 6-2
 - deleting 2-4, 6-18
 - opening 6-3
 - packing 2-23, 6-18
 - remote access 4-1
 - sorting 2-10
 - upgrading 2-16, 2-18, 6-18
 - viewing 6-4
- DataStore server deployment 4-3
- DataStore server implementation
 - customizing 4-4
- DataStore SQL identifiers C-5
- DataStore SQL keywords C-4
- DataStore streams
 - deleting 2-20, 6-8
 - example for deleting 2-21
 - formats 2-9
 - manipulating 6-7
 - naming conventions 2-17
 - specifications for A-2
 - undeleting 2-20, 2-21, 6-8
 - verifying 8-1
 - viewing contents 6-5
 - working with 2-15
- DataStore tables 3-4
 - accessing 3-17
 - adding records C-13, C-15
 - changing data 3-11
 - creating C-11
 - deleting C-17
 - deleting records in C-17
 - modifying C-12
 - opening in read-only mode 3-10
 - persistent data in 5-1
 - populating 3-3, 6-12, 6-14
 - saving changes to 6-15
 - sorting C-15
 - summarizing values C-13
 - troubleshooting incorrect sorting 8-2
 - updating C-16
- DataStoreConnection class 2-3
- DataStoreConnection component 3-11
- DataStoreDemo sample application 5-2
- DataSetException class 2-4
- dates 2-9, C-8
 - extracting parts C-9
- deadlock 4-4
- deadlocks
 - avoiding 4-5
- debugging DataStores 8-1
- Delete DataStore command 6-18
- DELETE statement C-17
- deleteStream method 2-20
- DeleteTest.java 2-21
- deleting DataStore files 2-4, 3-11, 6-18
- deleting DataStore streams 2-20, 6-8
 - example 2-21

- deleting records C-17
- delimited text files 6-16
- deployment
 - DataStore JDBC server 4-3
 - minimizing resources for 7-5
- DESC keyword C-15
- descending sort order C-15
- developer support 1-3
- Dir.java 2-7
 - running 2-10
- directory listings 2-10
- directory sort order 2-10
- directory table
 - adding entries 2-22
 - closing 2-11
 - contents described 2-9
 - creating 2-23
 - reading 2-10
- dirty reads 4-5
- disk writes 7-2
 - soft commit mode and 7-5
- DISTINCT keyword C-13
- documentation 1-2
- drivers
 - JDBC connections 4-1
- DROP INDEX statement C-17
- DROP TABLE statement C-17
- Dup.java 2-18
- DxTable.java 3-2

E

- editing in persistent columns 5-7
- editing offline tutorial 5-2
- embedded databases 3-1
 - changing data 3-11
 - opening in read-only mode 3-10
 - populating 3-4
- exceptions 2-4
- expressions
 - SQL syntax for C-5
- EXTRACT function C-9

F

- field separators 6-16
- file length 2-13
- file streams 2-1
 - closing 2-15
 - creating 2-13
 - getting length 2-13
 - importing files as 6-17
 - maintaining 2-20
 - testing for 2-10, 2-11
 - viewing contents 6-5
 - writing to 2-14
- fileExists method 2-11

- files
 - importing 2-13
 - marked as opened 6-4
 - storage capacity A-1
 - type-specific viewers 6-6
- FileStream object 2-12
 - instantiating 2-14

G

- garbage collection 7-2
- GROUP BY function C-13

H

- HAVING function C-13
- HIDDEN_STREAM bit 2-9

I

- identifiers
 - DataStore SQL engine C-5
- image files 6-6
- import statements 5-3
- ImportFile.java 2-12
- importing DataExpress packages 3-2
- importing DataStore files 2-13
- importing tables 6-12
- importing text files 6-16
- incorrect sorting 8-2
- indexes
 - creating C-12
 - removing from DataStores C-17
 - troubleshooting 8-2
- InputStream marking 2-15
- INSERT statement C-15
- installation support 1-3
- internal streams 2-9
- IS operator C-7
- isolation levels 4-5

J

- JAR files
 - JDBC DataStore remote server 4-3
- Java objects
 - getting 2-5
 - storing 2-4
- JDBC connections 3-18
 - transaction control and 3-20
- JDBC custom servers 4-4
- JDBC DataStore server deployment 4-3
- JDBC drivers 3-17, 4-1
 - running 4-2
- JDBC server startup options 4-4
- JdbcTable.java 3-18

K

keywords

 DataStore SQL engine C-4

L

leading characters C-10

length of file streams 2-13

LIKE operator C-7

literal values C-3

locks 4-5

log files 7-5

 creating for transactions 3-8

 moving 3-9

 saving 8-2

LOWER function C-9

M

maintenance 1-3

mapping table 5-7

mark method 2-15

maxLogSize property 3-9

moving transaction log files 3-9

multi-user access 2-14, 4-1

 transaction processing for 4-4

multi-user connections 3-11

N

naming streams 2-17

New DataStore dialog box 6-3, 6-4

nonrepeatable reads 4-5

O

offline editing 5-2

openDirectory method 2-6, 2-8

openFileStream method 2-15

opening connections 2-3

opening transactional DataStores 3-7

operating system crashes 7-5

 recovering from 3-6

operators

 precedence in SQL queries C-5

ORDER BY clause C-15

ordinal positions 5-8

output 2-16

P

Pack DataStore command 6-18

packaging remote servers 4-3

packing DataStore files 2-23, 6-18

padding characters C-10

parameterized queries

 guidelines for C-1

persistent column editing 5-7

persistent data 5-1

persistent data cache 6-11

persistent storage 2-6

phantom reads 4-5

populating DataStore tables 3-3, 6-12, 6-14

POSITION function C-9

power loss 7-5

PrintFile.java 2-15

providers

 DataStore components and 3-4

Q

queries

 adding expressions C-5

 case sensitivity C-5

 creating C-1

 running in DataStore Explorer 6-11, 6-12, 6-14

query console 6-11

 limitations for 6-12

R

random access 2-12

read access 2-12

read method 2-16

read-only mode 3-10

read-only transactions 4-5, 7-4

reconfiguring remote servers 4-3

referencing DataStore connections 2-14

remote access 4-1

remote editing 5-2

remote server

 startup options 4-4

remote servers

 packaging 4-3

 reconfiguring 4-3

 starting 4-2, 4-3

removing transaction support 3-10, 6-10

renaming streams 2-17, 6-7

reserved words

 DataStore SQL engine C-4

reset method 2-15

resolvers

 DataStore components and 3-4

resources

 freeing 7-2

 minimizing for deployment 7-5

restructuring DataStore components 5-8

retrieving Java objects 2-5

rollback method 3-11

rollbacks 3-8, 3-12

S

- saveMode property 7-2
- saving DataStore tables 6-15
- saving log files 8-2
- scalar literals C-3
- SCHEMA files 6-16
- secondary indexes
 - troubleshooting 8-2
- seek method 2-15
- SELECT statement C-13
- separators for delimited text files 6-16
- serializable transactions 4-5
- serializing DataStore component 2-2
- server startup options 4-4
- servers
 - DataStore implementation 4-4
 - deploying JDBC DataStore 4-3
 - reconfiguring 4-3
 - starting remote 4-2, 4-3
- single-connection applications 3-11
- soft commit mode 7-5
- sort keys
 - troubleshooting 8-2
- sort order C-15
- sorting data C-15
- sorting DataStore files 2-10
- SQL functions C-8
- SQL identifiers C-5
- SQL keywords C-4
- SQL predicates C-7
- SQL queries
 - case sensitivity C-5
 - creating C-1
 - running in DataStore Explorer 6-11, 6-12, 6-14
- SQL statements C-1, C-11
 - adding expressions C-5
 - executing 6-17
- SQL syntax C-1
- standalone tables 5-7
- starting DataStore Explorer 6-1
- starting remote servers 4-2, 4-3
- startup options
 - JDBC remote server 4-4
- storage 1-1, 5-1
 - advantages for persistent 2-6
 - data streams and 2-12
 - DataStore file size A-1
 - Java objects 2-4
- StorageDataSet class
 - overview 3-2
- StorageDataSet component
 - DataStore components and 5-2, 7-3
 - restructuring for DataStores 5-6

- stream types 2-9
 - determining 2-10, 2-11
- streams 2-1
 - checking existence of 2-11
 - closing file 2-15
 - copying 2-16, 6-8
 - creating file 2-13
 - creating table 3-3
 - DataStore formats 2-9
 - deleting DataStore 2-20, 6-8
 - example for DataStore 2-21
 - moving to any position 2-15
 - naming/renaming 2-17, 6-7
 - specifications for DataStore A-2
 - storing arbitrary files 2-12
 - undeleting DataStore 2-20, 2-21, 6-8
 - unique IDs and 2-10
 - verifying 8-1
 - viewing contents for DataStore 6-5
 - writing to 2-14
- StreamVerifier class 8-1
- string delimiters 6-16
- strings C-9
 - extracting substrings C-10
 - getting position C-9
 - removing padding characters C-10
- SUBSTRING function C-10
- substrings C-9
 - getting C-10
- summary values C-13
- support options 1-3

T

- table streams 2-1
 - creating 3-3
 - maintaining 2-20
 - testing for 2-10, 2-11
 - viewing contents 6-5
- TableDataSet component
 - as embedded database 3-1
- tableExists method 2-11
- tables
 - adding records C-13, C-15
 - creating C-11
 - creating standalone 5-7
 - deleting records C-17
 - importing 6-12
 - importing text files as 6-17
 - modifying C-12
 - removing from DataStores C-17
- technical support 1-3
- text files 6-6
 - importing 6-16
- time C-8
 - extracting parts C-9

- time formats 2-9
- timestamps C-8
- trailing characters C-10
- transaction contexts 3-11
- transaction isolation 3-6
 - levels described 4-5
- transaction log files 7-5
 - creating 3-8
 - moving 3-9
 - saving 8-2
- transactions 7-4
 - bypassing 3-10
 - changing 3-8, 6-10
 - controlling 3-11, 3-12, 3-20
 - enabling for DataStore components 3-6, 3-7
 - enabling with DataStore Explorer 6-9
 - manipulating with TXManager 6-10
 - multi-user access and 4-4
 - removing for DataStore components 3-10, 6-10
 - tracking 3-8
 - tutorial for 3-12
 - usage overview 3-11
- TRIM function C-10
- tutorials 2-1
 - accessing DataStores with JDBC 3-18
 - controlling transactions 3-12
 - embedding databases 3-2
 - offline editing 5-2
- TxManager class 3-6
- TxManager Properties dialog box 6-10
- type coercions 5-7

U

- undeleteStream method 2-21
- undeleting DataStore streams 2-20, 2-21
 - example 2-21
 - with DataStore Explorer 6-8
- unquoted SQL identifiers C-5
- UPDATE statement C-16
- updating DataStores C-16
- Upgrade DataStore command 6-18
- upgrading DataStore files 2-16, 2-18
 - with DataStore Explorer 6-18
- UPPER function C-9

V

- Verifier Log window 6-9
- Verify DataStore command 6-9
- viewers 6-6

W

- write access 2-12
- writing to file streams 2-14

Y

- Y2K issues 1-4
- year 2000 issues 1-4