Redistributable files

Refer to the file DEPLOY.TXT located in the root directory of your [Product Name] product for a complete list of files that you can distribute in accordance with the JDataStore 6 License Statement and Limited Warranty.

Borland Software Corporation 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 © 2002 Borland Software Corporation. All rights reserved. All Borland brand and product names are trademarks or registered trademarks of Borland Software Corporation in the United States and other countries. Other product names are trademarks or registered trademarks of their respective holders.

Printed in the U.S.A.
dg6
0203040506-9 8 7 6 5 4 3 2 1
PDF
Developer’s Guide 1

Chapter 1
Introduction
When to use JDataStore ............... 1-1
JDataStore and DataStore ............. 1-2
What you should know ................ 1-2
What’s in this book ................... 1-2
Using JDataStore for the first time .. 1-3
Starting with the JDataStore Explorer 1-3
Deploying JDataStore application components 1-3
Contacting Borland developer support . 1-3

Chapter 2
JDataStore fundamentals
JDataStore primer ..................... 2-1
Serializing objects ................... 2-2
Demonstration class: Hello.java .... 2-2
Creating a JDataStore file .......... 2-3
Opening and closing a connection ... 2-3
Handling basic JDataStore exceptions 2-4
Deleting JDataStore 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 JDataStore directory .... 2-8
JDataStore directory contents .... 2-9
Stream details .................... 2-9
Directory sort order ................. 2-10
Reading a JDataStore directory .... 2-10
Closing the JDataStore 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 JDataStore 2-14
Writing to a file stream ........... 2-14
Closing a file stream .............. 2-15
Opening, seeking, and reading a file stream 2-15
Creating a basic JDBC application 2-16
using JDataStore .................. 2-16
Copying streams ................... 2-20
copyStreams parameters ........... 2-21

Chapter 3
JDataStore as an embedded database
Using DataExpress for data access ..., 3-1
Demonstration class: DxTable.java ... 3-1
Connecting to a JDataStore with StorageDataSet ........... 3-2
Creating JDataStore tables with DataExpress 3-3
Using JDataStore tables with DataExpress 3-4
Transactional JDataStores .......... 3-5
Enabling transaction support ..... 3-5
Creating new transactional JDataStores . 3-6
Adding transaction support to existing JDataStores ........... 3-7
Opening a transactional JDataStore . 3-7
Changing transaction settings .... 3-7
Transaction log files ............... 3-8
Moving transaction log files ....... 3-9
Bypassing transaction support .......... 3-9
Removing transaction support ........ 3-10
Deleting transactional JDataStores .... 3-10
Controlling JDataStore transactions ........ 3-10
Understanding the transaction architecture 3-11
Committing and rolling back transactions .. .. 3-11
Tutorial: Controlling transactions via DataExpress ........... 3-11
Step 1: Create a transactional JDataStore with test data .... 3-12
Step 2: Create a data module .......... 3-13
Step 3: Create a GUI for the JDataStore table ........... 3-13
Step 4: Add direct transaction control .......... 3-15
Step 5: Add control over auto-commit .......... 3-16
Using JDBC for data access .......... 3-17
Demonstration class: JdbcTable.java ... 3-17

Naming and renaming the streams
to copy ............... 2-21
Demonstration class: Dup.java .... 2-22
Deleting and undeleting streams .... 2-24
Deleting streams ............... 2-24
How JDataStore reuses deleted blocks 2-24
Undeleting JDataStore streams .... 2-25
Demonstration class: DeleteTest.java ... 2-25
Locating directory entries .......... 2-26
Using individual directory rows .... 2-27
Packing JDataStore files ............ 2-27

Chapter 3
JDataStore as an embedded database
Using DataExpress for data access ..., 3-1
Demonstration class: DxTable.java ... 3-1
Connecting to a JDataStore with StorageDataSet ........... 3-2
Creating JDataStore tables with DataExpress 3-3
Using JDataStore tables with DataExpress 3-4
Transactional JDataStores .......... 3-5
Enabling transaction support ..... 3-5
Creating new transactional JDataStores . 3-6
Adding transaction support to existing JDataStores ........... 3-7
Opening a transactional JDataStore . 3-7
Changing transaction settings .... 3-7
Transaction log files ............... 3-8
Moving transaction log files ....... 3-9
Bypassing transaction support .......... 3-9
Removing transaction support ........ 3-10
Deleting transactional JDataStores .... 3-10
Controlling JDataStore transactions ........ 3-10
Understanding the transaction architecture 3-11
Committing and rolling back transactions .. .. 3-11
Tutorial: Controlling transactions via DataExpress ........... 3-11
Step 1: Create a transactional JDataStore with test data .... 3-12
Step 2: Create a data module .......... 3-13
Step 3: Create a GUI for the JDataStore table ........... 3-13
Step 4: Add direct transaction control .......... 3-15
Step 5: Add control over auto-commit .......... 3-16
Using JDBC for data access .......... 3-17
Demonstration class: JdbcTable.java ... 3-17

Naming and renaming the streams
to copy ............... 2-21
Demonstration class: Dup.java .... 2-22
Deleting and undeleting streams .... 2-24
Deleting streams ............... 2-24
How JDataStore reuses deleted blocks 2-24
Undeleting JDataStore streams .... 2-25
Demonstration class: DeleteTest.java ... 2-25
Locating directory entries .......... 2-26
Using individual directory rows .... 2-27
Packing JDataStore files ............ 2-27

Chapter 3
JDataStore as an embedded database
Using DataExpress for data access ..., 3-1
Demonstration class: DxTable.java ... 3-1
Connecting to a JDataStore with StorageDataSet ........... 3-2
Creating JDataStore tables with DataExpress 3-3
Using JDataStore tables with DataExpress 3-4
Transactional JDataStores .......... 3-5
Enabling transaction support ..... 3-5
Creating new transactional JDataStores . 3-6
Adding transaction support to existing JDataStores ........... 3-7
Opening a transactional JDataStore . 3-7
Changing transaction settings .... 3-7
Transaction log files ............... 3-8
Moving transaction log files ....... 3-9
Bypassing transaction support .......... 3-9
Removing transaction support ........ 3-10
Deleting transactional JDataStores .... 3-10
Controlling JDataStore transactions ........ 3-10
Understanding the transaction architecture 3-11
Committing and rolling back transactions .. .. 3-11
Tutorial: Controlling transactions via DataExpress ........... 3-11
Step 1: Create a transactional JDataStore with test data .... 3-12
Step 2: Create a data module .......... 3-13
Step 3: Create a GUI for the JDataStore table ........... 3-13
Step 4: Add direct transaction control .......... 3-15
Step 5: Add control over auto-commit .......... 3-16
Using JDBC for data access .......... 3-17
Demonstration class: JdbcTable.java ... 3-17
Chapter A
Specifications

JDataStore file capacity .......................... A-1

Chapter B
Troubleshooting

Debugging JDataStore applications ............... B-1
Verifying JDataStore contents .................... B-1
Problems locating and ordering data ............. B-2
Saving log file .................................... B-2

Chapter C
SQL reference

How to access SQL using JDBC: .................. C-1
Data types ......................................... C-2
Literals ........................................... C-4
JDBC Escape sequences ........................... C-5
Escape functions ................................. C-5
String functions ................................ C-6
Numeric functions ................................ C-6
Date and time functions ........................ C-6
System functions ................................ C-6
Keywords .......................................... C-7
Identifiers ........................................ C-9
Expressions ....................................... C-10
Predicates ......................................... C-11
BETWEEN ........................................... C-11
IS .................................................... C-12
LIKE ............................................... C-12
IN ................................................... C-13
Quantified comparisons .......................... C-13
EXISTS ........................................... C-14
Functions .......................................... C-14
ABSOLUTE ......................................... C-14
CHAR_LENGTH and

JOIN ............................................... C-14
CURRENT_DATE, CURRENT_TIME, and

CURRENT_TIMESTAMP ........................... C-15
EXTRACT .......................................... C-15
LOWER and UPPER ............................... C-15
POSITION ......................................... C-15
SQRT .............................................. C-16
SUBSTRING ....................................... C-16
TRIM .............................................. C-16
CAST .............................................. C-17
About statement syntax ........................ C-17
Lists in syntax notation ........................ C-17
Table expressions ............................... C-18
Select expressions .............................. C-19
Join expressions ............................... C-20
Statements ....................................... C-22
Chapter 1

Introduction

JDataStore is a high-performance, small-footprint, all Java™ multifaceted data storage solution. JDataStore provides:

- A zero-administration embedded relational database with both JDBC and DataExpress interfaces that supports transactional multi-user access with crash recovery. For more information, see the JBuilder Database Application Developer's Guide.
- Storage for serialized objects, tables, and other file streams.
- JavaBean components that can be manipulated with visual bean builder tools like JBuilder.

When to use JDataStore

With JDataStore, you can

- Embed SQL-92-compliant database functionality directly into your application without the need for an external database engine. You can access databases through the JDataStore JDBC driver or through the DataExpress components. 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 JavaBean components, JDataStore asynchronously replicates and caches data from a data source, allows access and updates, and resolves changes back into the data source. The data might come from a database server, a CORBA application server, SAP, BAAN, or some other data source.
- Increase the performance of online DataExpress applications with large datasets by using a DataStore instead of the default MemoryStore for caching data.
JDataStore and DataStore

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

What you should know

The JDataStore Developer’s Guide assumes you have a working knowledge of

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

What’s in this book

The JDataStore Developer’s Guide consists of a general guide and tutorial to using JDataStore followed by reference material. It contains these chapters:

- JDataStore fundamentals describes the basic structure of a JDataStore file system. It uses file streams to demonstrate various administrative tasks.
- JDataStore as an embedded database explains how to make a JDataStore transactional and use it as an embedded database with a sample GUI application.
- Using JDataStore’s security features describes the user authentication, user authorization, and encryption that JDataStore provides.
- Multi-user and remote access to JDataStores introduces the JDataStore Server used for remote access. It also discusses multi-user transactional issues.
- Persisting data in a JDataStore explains how to use the JDataStore as a persistent data cache for off-line computing.
- Using the JDataStore Explorer describes the JDataStore Explorer.
- Optimizing JDataStore applications contains a variety of tips on optimizing the performance, reliability, and size of JDataStore applications.
- Troubleshooting explains how to debug JDataStore applications and fix common problems.
- Specifications lists the specifications for the JDataStore file format.
- SQL reference is a reference guide for the SQL-92 dialect supported by the JDataStore JDBC driver.
Using JDataStore for the first time

JDataStore comes with one free development license. When you are ready to deploy JDataStore, you will need to purchase additional deployment license(s). Please contact Borland Customer Service for more information.

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

Starting with the JDataStore Explorer

DataStore Explorer (DSX) is an all-Java visual tool that helps you manage your JDataStores. It is covered in detail in Using the JDataStore Explorer. By using the JDataStore Explorer in conjunction with the sample JDataStore files that ship with JBuilder, you can get an quick idea of what a JDataStore can do.

The JDataStore Explorer provides visual tools for performing many maintenance tasks. The Developer’s Guide explains the fundamentals using the basic JDataStore API. You could begin by going to the JDataStore Explorer chapter first, however.

Throughout the JDataStore Developer’s Guide, you’ll see the following notation, which indicates some task that can be performed visually with the JDataStore Explorer:

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

Deploying JDataStore application components

You can find information on deploying the JDataStore Server for remote access in Deploying the JDataStore Server. For tips on reducing the deployed size of JDataStore client applications, see Pruning deployed resources.

Reminder: JDataStore is provided with a license for development only. For deployment, you must purchase additional licenses. 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. You can also choose from several categories of support, ranging from help on installing the Borland product to fee-based consultant-level support and detailed assistance.

For more information about Borland’s developer support services, 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.
Chapter 2

JDataStore fundamentals

This chapter contains several simple tutorials that demonstrate basic JDataStore concepts. If you haven’t already read the Introduction, please take a moment to do so before beginning the tutorials.

JDataStore primer

A JDataStore file can contain two basic types of data streams: table streams and file streams.

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

File streams can be further broken down into two different categories:

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

Note: All kinds of streams can be stored in the same JDataStore file.

A case-sensitive name referred to as storeName in the API identifies each stream. The name can be up to 192 bytes long. The name is stored along with other information about the stream in the JDataStore’s internal directory. The forward slash (‘/’) is used as a directory separator in the name to provide a hierarchical directory organization. The JDataStore Explorer uses this structure to display the contents of a DataStore in a tree.

The first part of this chapter covers JDataStore fundamentals using file streams. For information about working with table streams, see Creating a basic JDBC application using JDataStore, JDataStore as an embedded database, and Persisting data in a
JDataStore You may also wish to look at the sample which creates a basic JDBC application using JDataStore in /samples/JDataStore/HelloJDBC/.

**Serializing objects**

A DataStore is a component that you can program visually. But when you’re learning about DataStores, it might be easier to write simple code examples that demonstrate how a DataStore works. That’s what this chapter has you do.

The classic first exercise for a new language is how to display “Hello, World!” We’ll carry that tradition on here. (We’ll spare you, however, from performing the classic second exercise, a Fahrenheit to Celsius converter.)

First, create a new project for the dsbasic package, which you’ll use throughout this chapter.

**Important:** Add the JDataStore library to the project so that you can access the JDataStore classes. If you don’t know how to create a project or add a library, See "Adding a required library to a project" in the Database Application Developer’s Guide for instructions on adding a required library.

**Demonstration class: Hello.java**

Add a new file to the project, Hello.java, and type in this code:

```java
// 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();
        }
    }
}
```

Serializing objects After declaring its package, this class imports all the classes in the com.borland.datastore package. That package contains most of the public JDataStore
classes. (The rest of the public JDataStore classes are in the com.borland.datastore.jdbc package, which is needed only for JDBC access. It contains the JDBC driver class, and classes used to implement a JDataStore Server. These classes are covered in JDataStore as an embedded database and Multi-user and remote access to JDataStores) You can also access JDataStore through DataExpress components (packages under com.borland.dx). In this example, these classes are referenced explicitly so that you can see where each class comes from.

Creating a JDataStore file

A new DataStore object is created in the main() method of Hello.java. This object represents a physical JDataStore file and it contains properties and methods that represent its structure and configuration.

Next, the name “Basic.jds” is assigned to the DataStore object’s fileName property. It contains the default file extension “.jds” in lowercase. If the file name doesn’t end with the default extension, the extension is appended to the file name when the property is set.

You can’t create the JDataStore if a file with that name already exists. If the file doesn’t exist, the create() method creates it. If the method fails for any reason (for example, there’s no room on the disk, or someone just created the file in the nanoseconds between this statement and the last), it throws an exception. If the method succeeds, you have an open connection to a new JDataStore file.

DSX: See Creating a new JDataStore file When creating the file, you can also specify options like block size and whether the JDataStore is transactional.

Opening and closing a connection

If the file does exist, a connection opens through the open() method. The open() method is actually a method of the DataStore class’ superclass, DataStoreConnection, which contains properties and methods for accessing the contents of a JDataStore. (The fileName property is also a property of DataStoreConnection, which means that you can and often do access a JDataStore without a DataStore object, as you’ll see shortly.) Because DataStore is a subclass of DataStoreConnection, it has its own built-in connection, which is suitable for simple applications like this. (Note that DataStore can create a new JDataStore file, but DataStoreConnection cannot.)

But the excitement is short-lived. Immediately after opening a connection to the JDataStore file, creating the file in the process if necessary, that connection closes with the close() method. The close() method is also inherited from DataStoreConnection. Because there was only one built-in connection, when all the connections to the JDataStore are closed, the JDataStore file itself shuts down.

You must close any connections that you open before you exit your application (or call the DataStore.shutdown() method, which closes all connections). Opening a connection starts a daemon thread that continues to run and prevents your application from terminating properly. If you don’t close the connections, your application will hang on exit.
Handling basic JDataStore exceptions

Most of the methods in the JDataStore classes can throw a `DataSetException`, or more specifically, one of its subclasses, `DataStoreException`. Most of these exceptions are of the fatal “should never happen” or “don’t do that” variety. For example, you can’t set the `fileName` property if the connection is already open. You can’t create the JDataStore file if one already exists. You can’t open a connection if the named file isn’t really a JDataStore file. You might get an IO exception when writing data when closing a connection.

Therefore, almost all JDataStore code is inside a `try` block. In this case, if an exception is thrown, a stack trace prints.

Deleting JDataStore files

If you run the application now, all it does is create the file `Basic.jds`. If you then run it a second time, it does even less—just opening and closing a connection. Before you go further, you should delete the file.

There is no special function for deleting a JDataStore file. You can use the `java.io.File.delete()` method or anything else that accomplishes the task. As an aside example, if you always want to create a new JDataStore file, you write something like this code fragment:

```java
// store is DataStore with fileName property set
java.io.File storeFile = new java.io.File( store.getFileName() );
if ( storeFile.exists() ) {
    storeFile.delete();
}
store.create();
```

If the JDataStore 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.

DSX: See Deleting the JDataStore file. The JDataStore Explorer automatically deletes any associated transaction log files.

Storing Java objects

Add the boldfaced statements to the `if` block in the `main()` method:

```java
if ( !new java.io.File( store.getFileName() ).exists() ) {
    store.create();
    try {
        store.writeObject( "hello", "Hello, JDataStore! It's " + new java.util.Date() );
    } catch ( java.io.IOException ioe ) {
        ioe.printStackTrace();
    }
} else {
```
The `writeObject()` method attempts to store a Java object as a file stream in the JDataStore using Java serialization. (Note that you can also store objects in a table.) The object to be stored must implement the `java.io.Serializable` interface. A `java.io.IOException` (more specifically, a `java.io.NotSerializableException`) is thrown if it doesn't. Another reason for the exception would be if the write failed (for example, you ran out of disk space).

The first parameter of `writeObject()` specifies the storeName, the name that identifies the object in the JDataStore. 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 boldfaced statements to the `else` block in the `main()` method:

```java
} else {
    store.open();
    try {
        String s = (String) store.readObject( "Hello" );
        System.out.println( s );
    } catch ( com.borland.dx.dataset.DataSetException dse ) {
        dse.printStackTrace();
    } catch ( java.lang.ClassNotFoundException cnfe ) {
        cnfe.printStackTrace();
    } catch ( java.io.IOException ioe ) {
        ioe.printStackTrace();
    }
}
```

The `readObject()` method attempts to retrieve the named object from the JDataStore. Like `writeObject()`, it can throw an IOException for reasons like disk failure. It also can't reconstitute the stored object without the object's class. If that class is not in the classpath, `readObject()` throws a `java.lang.ClassNotFoundException`.

If the named object can't be found, a `DataSetException` with the error code `STORE NOT_FOUND` is thrown. `DataSetException` is a subclass of `DataSetException`. It's important to catch that exception here, even though there's another catch at the bottom of the method, because jumping there would bypass the call to `close()` the JDataStore connection. (The code is structured in this somewhat awkward way to teach certain principles.)

Because `readObject()` returns a `java.lang.Object`, you almost always cast the return value to the expected data type. (If the object isn't actually of that expected type, you get a `java.lang.ClassCastException`.) Here, it's more of a formality, because the `System.out.println` method can take a generic `Object` reference.
Advantages for persistent object storage

You can now run Hello.java. The first time it runs, it creates the JDataStore file and stores the greeting string. When you run it again, the greeting with the date and time displays in the console.

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

- It's simpler, using 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 instead of 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 might also use less storage space, because separate files can waste a lot of space because of how disk clusters are allocated. The default block size in a JDataStore file is small (4KB).
- Because you're not at the mercy of the host file system, your application is more portable. For example, different operating systems have different allowable characters for names. Some systems are case-sensitive, while others are not. Naming rules inside the JDataStore are consistent on all platforms.
- It provides an encryptable file system.

An internal directory system is of little use if you don't have a way to get the contents of the directory.

Using the directory

The DataStoreConnection.openDirectory() method returns the contents of the JDataStore in a searchable structure. But first, add the following program, AddObjects.java, to the project and run it to add a few more objects to the JDataStore:

```java
// AddObjects.java
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");
    
            DataStoreRecord rec = store.add( "intArray", intArray );
            DataStoreRecord rec2 = store.add( "date", date );
            DataStoreRecord rec3 = store.add( "properties", properties );
    
            DataStoreRecord rec4 = store.addDirectory( "myDirectory" );
            rec4.addEntry( "myEntry", rec2 );
            rec4.addEntry( "secondEntry", rec3 );
        }
    }
}
```
store.open();
store.writeObject( "add/create-time", date );
store.writeObject( "add/values", properties );
store.writeObject( "add/array of ints", intArray );
} catch ( com.borland.dx.dataset.DataSetException dse ) {
    dse.printStackTrace();
} catch ( java.io.IOException ioe ) {
    ioe.printStackTrace();
} finally {
    try {
        store.close();
    } catch ( com.borland.dx.dataset.DataSetException dse ) {
        dse.printStackTrace();
    }
}
}

The program does things slightly differently than Hello.java. First, it uses a DataStoreConnection object instead of a DataStore to access the JDataStore file, but it's used in the same way. You set the fileName property, open() the connection, use the writeObject() method to store objects, and close() the connection.

The location of the close() method call is another difference. Because you always want to call close() no matter what happens in the main body of the method, it's placed after the catch blocks inside a finally block. This way, the connection always closes, even if there is an unhandled error. The close() method is safe to call even if the connection never opened. In that case, close() does nothing.

This time, three objects are written to the JDataStore: an array of integers, a Date object (not a Date object converted into a string), and a hashtable. They are named so that they will be in a directory named “add”. The forward slash (/) is the directory separator character. One of the names contains spaces, which is perfectly valid.

**Demonstration class: Dir.java**

Add another file to the project, Dir.java:

```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();
```

```java
```
storeDir = store.openDirectory();
while ( storeDir.inBounds() ) {
    System.out.println( storeDir.getString(
        DataStore.DIR_STORE_NAME ) );
    storeDir.next();
} storeDir.closeDirectory();
} catch ( com.borland.dx.dataset.DataSetException dse ) {
    dse.printStackTrace();
} finally {
    try {
        store.close();
    } catch ( com.borland.dx.dataset.DataSetException dse ) {
        dse.printStackTrace();
    }
}

public static void main( String[] args ) {
    if ( args.length > 0 ) {
        print( args[0] );
    }
}

This class needs a command-line argument, the name of a JDataStore file, which is passed to its print() method. The print() method accesses that JDataStore using code similar to what you've seen before.

Opening a JDataStore directory

Dir.java defines a DataStoreConnection to access the JDataStore and also declares a StorageDataSet. After opening a connection to the JDataStore, the program calls the openDirectory() method of the DataStoreConnection to get the contents of the JDataStore's directory. The directory of a JDataStore is represented by a table.

DSX: See Viewing JDataStore file information
JDataStore directory contents

The JDataStore directory table has nine columns, which means there are nine pieces of information about each stream in the JDataStore, as shown in this table:

Table 2.1 JDataStore directory table columns

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

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

Stream details

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

As with many file systems, when you delete something in a JDataStore, the space it occupied is marked as available, but the contents and the directory entry that points to it are not wiped clean. This means maybe you can undelete something. For more details, see Deleting and undeleting streams.
The Type column indicates whether a stream is a file or table stream, but there are also many internal table stream subtypes (for things like indexes and aggregates). These internal streams are marked with the HIDDEN_STREAM bit to indicate that they should not be displayed. Of course, when you’re reading the directory, you can decide whether they should be hidden or visible.

These internal streams have the same StoreName as the table stream with which they’re associated. This means that the StoreName alone doesn’t always uniquely identify each stream when they interact with the JDataStore at a low level. Often some internal stream types have multiple instances. Therefore, the ID for each stream must guarantee uniqueness at a low level. But the StoreName is unique enough for the storeName parameter used at the API level. For example, when you delete a table stream, all the streams with that StoreName are deleted.

Directory sort order
The directory table is sorted by the first five columns. Because of the values stored in the State column, all active streams are listed first in alphabetical order by name. They are then followed by all deleted streams ordered by their delete time, oldest to most recent. (You can’t use a DataSetView to use a different sort order.)

Reading a JDataStore directory
You manipulate the JDataStore 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. Use the appropriate get<XXX>() method to read the desired information for each stream.

You can’t write to the JDataStore directory because it is read-only.

To run Dir.java, set the runtime parameters in the Project Properties dialog box to the JDataStore file to check, which in this case, is Basic.jds. When it runs, a loop goes through the directory, listing the name of every stream, such as this:

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

You can include a lot more information in the directory listing. The most difficult part is making the formatting decisions for the various bits of information available in all the columns of the JDataStore directory. To display whether the stream is a table or file stream, for example, add the boldfaced statements to the beginning of the loop:

```
while ( storeDir.inBounds() ) {
    short dirVal = storeDir.getShort( DataStore.DIR_TYPE );
    if ( (dirVal & DataStore.TABLE_STREAM) != 0 ) {
        System.out.print( "T" );
    } else if ( (dirVal & DataStore.FILE_STREAM) != 0 ) {
        System.out.print( "F" );
    } else {
        System.out.print( "?" );
    }
```
That addition changes the output to this:

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

The output indicates that all the serialized objects are indeed file streams.

### Closing the JDataStore directory

When you're not using the JDataStore directory, close it by calling the `DataStoreConnection.closeDirectory()` method. Most JDataStore operations modify the directory in some way. If the directory is open, it must be notified, which slows down your application.

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

### Checking for existing streams

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

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

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

It returns `false` because although there is a stream named “hello”, it's a file stream, not a table stream. The same result occurs with this:

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

This time the name doesn’t match case. Here the name and type match:

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

Now it returns `true`. 
Storing arbitrary files

In addition to serializing discrete objects as file streams, you can store and retrieve data streams in a JDataStore through a `com.borland.datastore.FileStream` object. Although `FileStream` is a subclass of `java.io.InputStream`, it has a method for writing to the stream as well so the same object can be used for both read and write access. It also provides random access with a `seek()` method. Because `FileStream` is a subclass of `InputStream`, it’s easy to use streams stored in the JDataStore in generic situations that expect an input stream. You’ll probably read a stream more often than you write one.

DSX: See Importing files

Demonstration class: ImportFile.java

Suppose you have an application that uses boilerplate documents that are modified for individual customers. A field in the customer table contains their personalized copy, but you 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 JDataStore. The following utility program, `ImportFile.java`, does this for you. Add it to the project.

`// ImportFile.java`

```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 );
    fis.close();
}

store.writeObject( streamName + LAST_MOD,
    new Long( file.lastModified() ) );

} catch ( com.borland.dx.dataset.DataSetException dse ) {
    dse.printStackTrace();
} catch ( java.io.FileNotFoundException fnfe ) {
    fnfe.printStackTrace();
} catch ( java.io.IOException ioe ) {
    ioe.printStackTrace();
} finally {
    try {
        store.close();
    } catch ( com.borland.dx.dataset.DataSetException dse ) {
        dse.printStackTrace();
    }
}

public static void main( String[] args ) {
    if ( args.length == 2 ) {
        read( args[0], args[1] );
    } else if ( args.length >= 3 ) {
        read( args[0], args[1], args[2] );
    }
}

The program takes as parameters the name of a JDataStore file, the name of the file to import, and an optional stream name. If you don't specify a file stream name, the file name is used. The main() method calls the appropriate form of the read() method, because the two-argument read() method calls the three-argument read() method.

When the file is imported, the date it was last modified is recorded with it. The ”/modified” suffix appends to the stream name for this date, while the ”/data” suffix appends to the stream name to contain the data from the file. These suffixes are defined as class variables.

The read() method then begins by opening a connection to the JDataStore 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 might 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() throws a DataStoreException with the error code DATASET_EXISTS, because you can't have a table stream and a file stream with the same name.

When createFileStream is successful, it returns a FileStream object that represents the new, empty file stream.

**Referencing the connected JDataStore**

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 can cause performance problems as well. As a conservative start, you can make the size a small multiple of the JDataStore's block size.

The JDataStore's block size is stored in the DataStore object's blockSize property. Whenever you use a DataStoreConnection to access a JDataStore, it automatically creates an instance of DataStore. Other DataStoreConnection objects in the same process that connect to the same JDataStore share that DataStore object. (Access to a JDataStore 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:

```java
byte[] buffer = new byte[4 * store.getDataStore().getBlockSize() * 1024];
```

The getDataStore() method gets the reference to the DataStore object, and from that the getBlockSize() method gets the blockSize property. The property value is in kilobytes so it is multiplied by 1024. The resulting block size is multiplied by four, the arbitrarily-chosen number of blocks to read in each chunk.

**Writing to a file stream**

The FileStream object's write() method takes an array of bytes such as a java.io.OutputStream, although the only form of the method is the one that also specifies the starting offset and length.

The java.io.FileInputStream object reads from a file into an array of bytes. It returns the number of bytes read, or -1 if the end-of-file is reached. In the loop, the number of bytes read is checked for the end-of-file value. If it's not the end-of-file, the number of bytes read are written, starting with the first byte in the array. For every iteration of the loop except the last, the entire array is filled by reading and writing into the FileStream. The last iteration probably won't fill the entire array.
Closing a file stream

Once you're done with a file stream, you should close it. The FileStream object uses the close() method (as does the FileInputStream).

After the file stream is closed, the last-modified date is written using a java.lang.Long object to encapsulate the primitive long value. (You cannot save primitives with serialization.)

To test ImportFile.java, try importing some source code files into Basic.jds.

Opening, seeking, and reading a file stream

Use the openFileStream() method to open an existing file stream by name. Like createFileStream(), it returns a FileStream object at the beginning of the stream. You can then go to any position in the stream with the seek() method, write to the stream, and read from it with the read() method. FileStream also supports InputStream marking with the mark() and reset() methods.

The PrintFile.java program demonstrates opening, seeking, and reading. Add it to the project.

```java
// 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 || LAST_MOD || longValue() ) ) );
        } catch ( com.borland.dx.dataset.DataSetException dse ) {
        }
    }
}
```
dse.printStackTrace();
} catch (java.io.IOException ioe) {
    ioe.printStackTrace();
} catch (java.lang.ClassNotFoundException cnfe) {
    cnfe.printStackTrace();
} finally {
    try {
        store.close();
    } catch (com.borland.dx.dataset.DataSetException dse) {
        dse.printStackTrace();
    }
}

public static void main(String[] args) {
    if (args.length == 2) {
        printBackwards(args[0], args[1]);
    }
}

To demonstrate random access with the seek method (and to make things slightly more interesting), this program prints a file stream backwards. It determines the length of the file stream by calling the FileInputStream's available() method and uses it as a file pointer. When reading from the file, the program moves the file pointer forward. The position of the file pointer decrements and is set for each byte read in the loop. There are two forms of the read() method. The first reads into a byte array (the same form of the method used by the FileInputStream in ImportFile.java). The second returns a single byte. Here the single-byte form is used. Each byte is cast into a character to be printed.

Creating a basic JDBC application using JDataStore

Now that you've learned about creating and manipulating file streams in a JDataStore, it's time to teach you the basics of creating a JDBC application using JDataStore. For more detailed information about creating JDBC applications using JDataStore, see the chapter on JDataStore as an embedded database.

We'll start by creating a new file in the dsbasic package called HelloTX.java. The code for this is very similar to the Hello.java file you created earlier. The differences are shown in boldface:

    // HelloTX.java
    package dsbasic;

    import com.borland.datastore.*;

    public class HelloTX {

        public static void main(String[] args) {
            DataStore store = new DataStore();

2-16  The JDataStore 6 Developer's Guide
try {
    store.setFileName( "BasicTX.jds" );
    store.setUserName("CreateTX");
    store.setTXManager(new TxManager());
    if ( !new java.io.File( store.getFileName() ).exists() ) {
        store.create();
    } else {
        store.open();
    }
    store.close();
} catch ( com.borland.dx.dataset.DataSetException dse ) {
    dse.printStackTrace();
}
} }

The most important difference here is that a TxManager is instantiated and assigned to be the transaction manager for the JDataStore. A JDBC application requires a transactional JDataStore, so a transaction manager is necessary. To create (or open) a transactional JDataStore, you must also set the userName property. If there is no name in particular that you find appropriate, you can set it to a dummy name.

The next step is to write some code that connects to the DataStore. Add a file called HelloJDBC.java to the project. Type the following code into the new file:

```java
//HelloJDBC.java
package dsbasic;
import java.sql.*;
public class HelloJDBC {
    public HelloJDBC() {
    }
    static void main(String args[]) {
        // Both the remote and local JDatastore drivers use the same
        // driver string:
        String DRIVER = "com.borland.datastore.jdbc.DataStoreDriver";

        // Use this string for the local driver:
        String URL = "jdbc:borland:dslocal:";

        // Use this string for the remote driver (and start JDataStore Server):
        // String URL = "jdbc:borland:dsremote://localhost:";

        String FILE = "BasicTX.jds";
        boolean c_open=false;
        Connection con = null;
```
try {
    Class.forName(DRIVER);
    con = DriverManager.getConnection(URL + FILE, "user", "");
    c_open = true;
}

} catch(Exception e) {
    System.out.println(e);
}

// This way the connection will be closed even when exceptions are thrown
// earlier. This is important, because you may have trouble reopening
// a JDatastore file after leaving a connection to it open.
try {
    if(c_open)
        con.close();
}

} catch(Exception e3) {
    System.out.println(e3.toString());
}

Note the boldface lines of code in this program. They are the most important ones to
note. First, the driver string for the JDataStore JDBC driver is specified. This string is
always the same for both the local and remote JDBC drivers. Next, the URL string for
connecting to a local JDataStore is shown. For your information, the code also shows
the remote string, but this is commented out. The last two boldface lines are common
to many JDBC applications, and they're where we actually connect to the JDataStore.

Once you've connected to the JDataStore, you'll probably want to add and
manipulate some data. We'll show you how to do that next. We won't spend a lot of
time on it here, just enough to let you know that you have connected to the
JDataStore, and can add, manipulate, print, and delete data. Add the following
boldfaced lines to the code as shown:

    package dsbasic;
    import java.sql.*;

    public class HelloJDBC {
        public HelloJDBC() {
        }

        public static String formatResultSet(ResultSet rs) {
            // This method formats the result set for printing.
            try {
                ResultSetMetaData rsmd = rs.getMetaData();
                int numberOfColumns = rsmd.getColumnCount();
                StringBuffer ret = new StringBuffer(500);
for (int i = 1; i <= numberOfColumns; i++) {
    String columnName = rsmd.getColumnName(i);
    ret.append(columnName + "," );
}
ret.append("\n");
while (rs.next()) {
    for (int i = 1; i <= numberOfColumns; i++)
        ret.append(rs.getString(i) + "," );
    ret.append("\n");
    return(ret.toString());
}
catch(Exception e) {
    return e.toString();
}
}

static void main(String args[]) {
    // Both the remote and local JDatastore drivers use the
    // same driver string:
    String DRIVER =   "com.borland.datastore.jdbc.DataStoreDriver";
    // Use this string for the local driver:
    String URL    =   "jdbc:borland:dslocal:";
    // Use this string for the remote driver (and start JDataStore Server):
    String URL =  "jdbc:borland:dsremote://localhost/";
    String FILE   =   "BasicTX.jds";
    boolean s_open=false, c_open=false;
    Statement stmt = null;
    Connection con = null;
    try {
        Class.forName(DRIVER);
        con = DriverManager.getConnection(URL + FILE, "user", "");
        c_open = true;
        stmt = con.createStatement();
        s_open = true;
        // The following line creates a table in the JDataStore.
        stmt.executeUpdate("create table HelloJDBC" +
                          "(COLOR varchar(15), " +
                          " NUMBER int, " +
                          " PRICE float)");
        // Values are inserted into the table with
        // the next three statements.
        stmt.executeUpdate("insert into HelloJDBC values('Red', 1, 7.99)");
        stmt.executeUpdate("insert into HelloJDBC values('Blue', 2, 8.99)");
        stmt.executeUpdate("insert into HelloJDBC values('Green', 3, 9.99)");
        // Now we query the table

ResultSet rs = stmt.executeQuery("select * from HelloJDBC");

// Call to formatResultSet() to format the
// printed output.
System.out.println(formatResultSet(rs));

// The next line deletes the table.
stmt.executeUpdate("drop table HelloJDBC");
}
catch(Exception e) {
    System.out.println(e);
}

try {
    // Attempt to clean up by calling the
    // java.sql.Statement.close() method.
    if(s_open)
        stmt.close();
    catch(Exception e2){
        System.out.println(e2.toString());
    }
}

// This way the connection will be closed even when exceptions are thrown
// earlier. This is important, because you may have trouble reopening
// a JDatastore file after leaving a connection to it open.
try {
    if(c_open)
        con.close();
    catch(Exception e3) {
        System.out.println(e3.toString());
    }
}

In the preceding example, the code added to the main() method creates a table and inserts rows in the table. Then it calls the formatResultSet() method and prints the results. Next, it deletes the table from the JDataStore. Finally, it attempts to clean up by calling the close() method of the java.sql.Statement object.

Copying streams

The DataStoreConnection class' copyStreams() method makes a new copy of one or more streams in the same JDataStore or it copies the streams to a different JDataStore. If it encounters an error in an original stream, it attempts to correct that error in the copy. Also, you can use copyStreams() to upgrade an older JDataStore file into the current format.
The `copyStreams` method takes six parameters, as listed in this table:

**Table 2.2 DataStoreConnection.copyStreams method parameters**

<table>
<thead>
<tr>
<th>Parameter name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sourcePrefix</td>
<td>Stream name must begin with this to be pattern-matched; empty string to pattern-match all streams.</td>
</tr>
<tr>
<td>sourcePattern</td>
<td>Stream name pattern to match, with standard * and ? wildcard characters.</td>
</tr>
<tr>
<td>destCon</td>
<td>Connection to destination JDataStore.</td>
</tr>
<tr>
<td>destPrefix</td>
<td>Names of copies of streams have their <code>sourcePrefix</code> replaced with this; should be the same as <code>sourcePrefix</code> if the name should not be changed.</td>
</tr>
<tr>
<td>options</td>
<td>Zero or more of the following <code>DataStore</code> class variables:</td>
</tr>
<tr>
<td></td>
<td>• COPY_CASE_SENSITIVE</td>
</tr>
<tr>
<td></td>
<td>• COPY_IGNORE_ERRORS</td>
</tr>
<tr>
<td></td>
<td>• COPY_OVERWRITE</td>
</tr>
<tr>
<td>out</td>
<td><code>java.io.PrintStream</code> to direct status messages; <code>null</code> to suppress output.</td>
</tr>
</tbody>
</table>

Each of the `options` reverses the default behavior of `copyStreams`. The default behavior

- Ignores case when matching stream names.
- Stops if an unrecoverable error is encountered.
- Stops 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 are output to the designated `PrintStream`.

**DSX:** The JDataStore Explorer provides a UI for copying streams to a new JDataStore file with these parameters. See Copying JDataStore streams

**Naming and renaming the streams to copy**

Forward slashes in stream names are used to simulate a hierarchical directory structure. The `copyStreams()` method is unaware of 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 are copied. `sourcePrefix` is used in combination with the `destPrefix` parameter to rename a stream when it is copied; that is, to change the prefix (the beginning) of the `storeName` of the resulting copy of the stream.
If you specify a `sourcePrefix`, the stream name must start with that string. It's usually used to specify the name of a directory ending with a forward slash. The `destPrefix` is then set to a different directory name also ending with a forward slash. The `sourcePrefix` is stripped from the name, and the `destPrefix` is prepended to the name of the copy. For example, suppose you have the stream named “add/create-time” and you want to create a copy named “tested/create-time”. The effect is to make a copy in a different directory. You would set `sourcePrefix` to “add/” and `destPrefix` to “tested/”.

Although the prefix parameters are usually used for directories, you can rename streams in other ways. For example, you can rename “hello” to “jello” by specifying “h” and “j” for the `sourcePrefix` and `destPrefix` respectively. Or you can change “three/levels/deep” to “not-a-peep” by specifying “three/levels/d” and “not-a-p”. The effect is to move a stream up to the root directory of the JDataStore. You can also do the reverse by making the `destPrefix` longer (with more directory levels) than the `sourcePrefix`. For example, by leaving the `sourcePrefix` blank, but specifying a `destPrefix` that ends with a forward slash, all the streams from the original JDataStore file are placed under a directory in the destination JDataStore.

If you’re not renaming the copy of the stream, there’s no reason to use either prefix parameter, so you should set both of them to an empty string or null. Note that if you’re making a copy of a stream in the same JDataStore file, you must rename the copy.

The `sourcePattern` parameter is matched against everything after the `sourcePrefix`, using the standard wildcard characters “*” (for zero or more characters) and “?” (for a single character). If the `sourcePrefix` is empty, that means that the pattern is matched against the entire string. If you want to copy all the streams in a directory, you can put the directory name in the `sourcePattern`, followed by a forward slash, and leave the `sourcePrefix` empty. For example, if you want to copy everything in the “add” directory, you want to copy everything that starts with “add/”, so the `sourcePattern` would be “add/”. That includes everything in subdirectories, because the `sourcePattern` matches the remainder of the string. (There is no direct way to prevent the copying of streams in subdirectories.)

The `sourcePattern` is matched against names of active streams only. `copyStreams()` doesn’t copy deleted streams.

**Demonstration class: Dup.java**

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

```java
// Dup.java
package dsbasic;

import com.borland.datastore.*;

public class Dup {
    
    public static void copy( String sourceFile, String destFile ) {
        DataStoreConnection store1 = new DataStoreConnection();
```
```java
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. It uses a
DataStoreConnection object to open the source JDataStore. It copies the contents to a
DataStore object so that the JDataStore file can be created if it doesn't already exist.

For the copyStreams method, the sourcePrefix and destPrefix are empty strings, and the
sourcePattern is just "*", which copies everything without renaming anything. The
program ignores unrecoverable errors and displays status messages in the console.

With this program you can combine the contents of more than one JDataStore 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 might not always work and requires a bit more effort. Streams are deleted by name. Understanding what happens when you delete or try to undelete a file stream, whether it’s an arbitrary file or serialized object, is simpler because there’s only one stream with that name. Table streams often have additional internal support streams with the same name as explained in Stream details. 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 a table stream, the main stream and all its support streams are deleted.

Deleting a stream doesn’t 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 might overwrite the space that was formerly occupied by the deleted stream, making the content of the deleted stream unrecoverable.

DSX: See Deleting streams.

How JDataStore reuses deleted blocks

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

- The JDataStore always reclams deleted space before allocating new disk space for its blocks.
- If the JDataStore is transactional, the transaction that deleted the stream must commit before the used space can be reclaimed.
- The oldest deleted streams, the ones with the earliest delete times, are reclaimed first.
- For table streams, the support streams (such as those for indexes and aggregates) are reclaimed first.
- Space is reclaimed from the beginning of the stream to the end of the stream, meaning that you are more likely to recover the end of a file or table than the beginning.
- Because of the way table data is stored in blocks, you never lose or recover a partial row in a table stream, only complete rows.
- When all the space for a stream has been reclaimed, the directory entry for the stream is automatically erased (there is no chance for recovery anyway).
Undeleting JDataStore streams

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

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

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

**DSX:** See Undeleting streams

Demonstration class: DeleteTest.java

The following program, `DeleteTest.java`, demonstrates both deletion and undeletion.

```java
// 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 ) );
```

```
In this program, the name of the JDataStore file and the stream to be deleted are hard-coded, which you would seldom do. The stream is “add/create-time”, which was added to Basic.jds in the Using the directory demonstration program. 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 JDataStore and opening its directory. Next, it locates the directory entry for the stream that is about to be deleted.

Note: Usually 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, which are explained shortly.

To locate the row, a new com.borland.dx.dataset.DataRow is instantiated from the directory dataset, specifying the two columns that are used in the search: 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 `undeleteStream()`, the current row is used. But because of the way the JDataStore directory is sorted (as explained in Directory sort order), when a stream is deleted, its directory entry probably “flies away” to its new position at the bottom of the directory as the most recently deleted stream. The current row is then referencing something else (probably the next stream alphabetically). To undelete the same stream, you could either attempt to relocate the directory entry for the now-deleted stream, or you 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 JDataStore directory dataset, an individual row is a static copy. It’s smaller. After making the copy, you can close the directory dataset to make operations faster. (For this simple demonstration, the overhead for creating the individual row probably outweighs any performance benefit.) You can make static copies of as many directory entries as you want, and manage them any way you want.

To create the individual directory row, another `DataRow` is instantiated from the directory dataset (so that it has the same structure), and the `copyTo()` method copies the data from the current row. And just to prove that it really works, the JDataStore directory is closed.

The file stream is then deleted by name using the plain name string defined at the beginning of the method. Finally, the stream is undeleted using the individual directory entry.

Packing JDataStore files

The only way to shrink a JDataStore file, that is, removing unused blocks and directory entries for deleted streams, is to copy the streams to a new JDataStore file using `copyStreams()`. Only active streams are copied, which results in a packed version of the file.

DSX: See Packing the JDataStore file
JDataStore as an embedded database

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

JDBC access requires that the JDataStore be transactional. DataExpress does not. This chapter begins with DataExpress access, then discusses transactional JDataStores, and finally the local JDBC driver. The remote JDBC driver and JDataStore Server are discussed in Multi-user and remote access to JDataStores.

Using DataExpress for data access

To use a JDataStore as a database file, associate a component that extends from StorageDataSet, such as TableDataSet, to a stream inside a JDataStore. 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 and name it DxTable.java:

```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 the program uses DataExpress, it imports the DataExpress package in addition to the JDataStore 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 JDataStore 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.

If you are defining a completely new table, use TableDataSet.

Each StorageDataSet has a store property that is null when the object is instantiated. If it's still null when the dataset is opens, a com.borland.dx.memorystore.MemoryStore is assigned automatically, which means that the data is stored in memory. If you assign a DataStoreConnection or DataStore to the store property, the data is stored in a persistent JDataStore file instead.

To connect a StorageDataSet to a JDataStore, assign values to these three properties:
• The `fileName` property of the `DataStoreConnection`. This indicates which JDataStore to connect to.

• The `storeName` property of the `StorageDataSet`. This indicates the name of the table stream inside the JDataStore. 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 JDataStore and choose names that don’t conflict.)

• The `store` property of the `StorageDataSet`, which is set to the `DataStoreConnection` (or `DataStore`) object. This connects the two together.

Perform these three steps in any order. Once you've set all three properties, you have a fully qualified connection between a `StorageDataSet` and a JDataStore.

In `DxTable.java`, the JDataStore file is `Basic.jds`, which you created in Creating a JDataStore file. The table stream is named `Accounts`. Think of it as the name of the table. `DxTable.java` assigns `DataStoreConnection` as the value of the `TableDataSet`'s `store` property.

Then the `TableDataSet` opens. Opening a dataset that has a JDataStore attached automatically opens that JDataStore file. If the JDataStore opens successfully, the program creates the named table stream if it doesn’t already exist. If it does exist, that table stream reopening. This establishes an open connection between the dataset and its table stream in the JDataStore. Note that if there is a file stream in the JDataStore with the same name, the program throws an exception because you can’t have a table stream with the same name as a file stream.

### Creating JDataStore tables with DataExpress

Opening a `StorageDataSet` connected to a JDataStore results in an open table stream. For new table streams, `QueryDataSet` and `ProcedureDataSet` then fetch data from their data source and populate the table stream as explained in Tutorial: Offline editing with JDataStore. But `TableDataSet` has no data source. You start with an empty and undefined table stream.

Add the highlighted statements to `DxTable.java`:

```java
table.open();
if ( table.getColumns().length == 0 ) {
    createTable();
}
```  

To detect that a table stream is new, check 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 called `createTable()`. Add it to `DxTable.java`:

```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 to the table.

After you modify the structure of the table by adding these new columns, activate the changes by calling the `StorageDataSet.restructure()` method. The result is an empty but structured table stream, a new table in the JDataStore. (If you know the table doesn’t exist, you can use `addColumns()` to define the structure before opening the `TableDataSet`. Then you won’t need to call `restructure()`.)

You can store as many tables as you want in a single JDataStore file. They must use different table stream names. You can use the same `DataStoreConnection` object in the `store` properties of each `TableDataSet`.

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

---

### Using JDataStore tables with DataExpress

Once the tables in the JDataStore have been defined (no matter how they were created), you can 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 on. In fact, such secondary indexes are also persisted and maintained in the JDataStore file, making the JDataStore a complete embedded database.

To complete the demonstration program, add a smattering of DataExpress functionality with this new method:

```java
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:

```java
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.getInt( "ID" ) + "; ");
        table.getString( "Name" );
        table.getTimestamp( "Update" );
        table.getInt( "ID" );
    }
```
After the program opens the table, creating the table's structure if necessary, it sets a SortDescriptor on the ID field. To add some rows, it calls the appendRow() method.

The appendRow() method begins by going to the last row in the table and obtaining the value of the ID field. Because of the sort order, this value should be the highest ID number used so far. (If the table is empty, the getInt() method returns zero.) The new ID value is one greater than the last. appendRow() inserts a new row and sets its attributes, including the Update field, which is set to the current date and time. Finally appendRow() saves the new row by calling the post() method.

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

If you run the program a few times, you'll see that the new rows get unique ID numbers. This method of generating ID numbers works for a simple single-threaded demonstration program like this that always commits new rows after getting the old ID number. But for more realistic programs, such an approach may not be safe. To use a more robust approach, you must understand locks and transactions.

**Transactional JDataStores**

So far, changes you have made to a JDataStore have been direct and immediate. If you write an object, change some bytes in a file stream, or add a new row to a table, it's done without concern for other connections that might be accessing the same stream. Such changes are immediately visible to 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. It's required for JDBC access.

**Enabling transaction support**

Transaction support is provided by the com.borland.datastore.TxManager class. A JDataStore can be transactional when it is first created, or you can add transaction support later. In either case, you assign a TxManager object as the value of the txManager property of the DataStore object, usually before calling the create() or open() method.

Keep the following points in mind regarding JDataStore transaction support:

- When a JDataStore is made non-transactional, all TxManager property settings are forgotten. If the JDataStore is made transactional again, the TxManager properties will revert to their defaults.
If the `TxManager.ALogDir` and `TxManager.BLogDir` properties are not set, the location of the log files is always assumed to be in the same directory as the location of the JDataStore file. This behavior lets you move the JDataStore file from one directory to another without warnings that the log files exist in the original location and the new location.

If you assign the `txManager` property on an open JDataStore, it causes the `TxManager` to automatically shut down and attempt to reopen the JDataStore so that the new property setting can take effect immediately. If the `DataStoreConnection.userName` property has not been set, the JDataStore will fail to reopen, and an exception will be thrown.

The properties of the `TxManager` object determine various aspects of the transaction manager. When instantiated, the `TxManager` has usable default settings for these properties. If you want to change any of these settings, it's better to do it before creating or opening the JDataStore.

The first time the now-transactional JDataStore opens, it stores its transactional settings internally. The next time you open the JDataStore, you don't have to assign a `TxManager`. Instead the JDataStore automatically instantiates a `TxManager` with its stored settings.

Creating new transactional JDataStores

Here's the minimum code for creating a new transactional JDataStore with default settings:

```java
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 JDataStore is the setting of the `userName` and `txManager` properties. (You can set them in any order.) If you don't want the default settings, the code generally looks something like this:

```java
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 JDataStore file

Adding transaction support to existing JDataStores

The code for making an existing JDataStore transactional is very similar. The main difference is the call to open() instead of create(). For a default TxManager, the code might look like this:

```java
DataStore store = new DataStore();
store.setFileName( "SomeFileName.jds" );
store.setUserName( "AnyNameWouldWork" );
store.setTxManager( new TxManager() );
store.open();
```

Note that even though you are much more likely to use a DataStoreConnection to open an existing JDataStore file, you can’t use one when you are adding transaction support, because txManager is a property of DataStore, not DataStoreConnection.

DSX: See Making the JDataStore transactional

Opening a transactional JDataStore

The only difference when opening a JDataStore that’s transactional and one that’s not is that you must specify a userName. Because it doesn’t hurt to specify a userName for a non-transactional JDataStore (it’s simply ignored), you might want to always specify a userName when opening a JDataStore. The code would look something like this:

```java
DataStoreConnection store = new DataStoreConnection();
store.setFileName( "SomeFileName.jds" );
store.setUserName( "AnyNameWouldWork" );
store.open();
```

Because no TxManager was assigned, when the JDataStore opens, a TxManager is automatically instantiated with its properties set to the values that were persisted in the JDataStore. The TxManager is assigned to the JDataStore’s txManager property. You can get the values of the persisted transaction management properties from there, but you can’t change them directly.

Changing transaction settings

To change a JDataStore’s transaction setting, assign a new TxManager object before opening. The TxManager object knows which properties have been assigned and which ones have been left at their default value. If you assign a TxManager to a transactional JDataStore, only those properties that have been assigned in the new TxManager are changed. All other properties remain as they were; they do not revert to default values (the ones in the new TxManager).

As when adding transaction support, you should assign the TxManager with the new values before you open the JDataStore. For example, suppose you want to change the softCommit property to true. Doing so improves performance by not guaranteeing
recently committed transactions (within approximately one second before a system failure) yet it still guarantees crash recovery:

```java
DataStore store = new DataStore();
TxManager txMan = new TxManager();

// Make changes to TxManager
txMan.setSoftCommit( true );

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

Note that the other properties, such as recordStatus, aren’t set. Although the new TxManager has the default setting when it is assigned to the JDataStore, the setting in the JDataStore isn’t affected, even if it’s not the default.

**DSX:** See Modifying transaction settings

### Transaction log files

The transaction manager works by logging changes made to the JDataStore including the previous values so that the transaction can be rolled back. (The changes aren’t removed from the log file when the changes are committed, so if you archive the log files, it’s possible to extract a complete change log or to reconstruct the contents of the JDataStore.) Most of the TxManager properties control these attributes of the transaction log files:

- Whether to duplex them; that is, whether to keep two separate but identical copies for greater reliability at the expense of some performance.
- Where to put them. When the log files are duplexed, the two copies are usually kept in different locations. Keeping them on different physical drives increases reliability and the chance for recovery even further, and it might also offset some of the performance penalty.
- How big 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 JDataStore file.

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

- The **status file** `MyStore_STATUS_0000000000`,
- The **anchor file** `MyStore_LOGA_ANCHOR`
- The **record file** `MyStore_LOGA_0000000000`.

---

3-8 The JDataStore 6 Developer’s Guide
Duplex logging adds the files `MyStore_LOGB_ANCHOR` and `MyStore_LOGB_0000000000`. These two sets of log files are referred to as the “A” and “B” log files. The `ALogDir` and `BLogDir` properties control the location of these files.

Once a log file reaches the size determined by the `TxManager`'s `maxLogSize` property, additional status and record files are created with the log file number incrementing by one each time. As old log files are no longer needed for active transactions or crash recovery, they are automatically deleted. For information on archiving them, see `Saving log file`.

### Moving transaction log files

If the `ALogDir` and `BLogDir` properties are not set, then the location of the log files is always assumed to be in the same directory as the directory of the JDataStore file. This makes it easier to move the JDataStore from one directory to another. If the `ALogDir` and `BLogDir` properties are set, they include the drive and full path, which means two things:

- If you’re going to create transactional JDataStore files before moving them to another computer, and you’re setting the `ALogDir` and `BLogDir` properties, you should try to create your transactional JDataStore files 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 JDataStore files were created on the C: drive because you don’t have a D: drive on your development computer, you must go through the extra steps of moving the log files when you deploy because the drives are different.

- When you move log files, follow these steps:
  a. Move the log files to the new location. Be sure to remove or rename any copies of the files in the original location.
  b. Create a new `TxManager` with the new location property settings. Assign it to the `txManager` property of the `DataStore`.
  c. Open the JDataStore. The `TxManager` looks in the new location, sees that the log files are there, and changes the persisted settings in the JDataStore.

### Bypassing transaction support

Sometimes you want to access a transactional JDataStore, but need to bypass transaction support. Here are some examples of when this might happen:

- The transaction log files are lost. You can’t open the JDataStore normally.
- You only have read-only access to the JDataStore files. For example, they might 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 JDataStore in read-only mode. Do this through a `DataStore` object. Before opening the JDataStore, set its `readOnly` property to `true`. For example,

```java
DataStore store = new DataStore();
store.setFileName( "SomeReadOnly.jds" );
```
store.setReadOnly( true );
store.open();

Because you are bypassing the TxManager, you don't need to set the userName. If the transaction log files are lost, use the copyStreams() method (or the JDataStore Explorer, see Copying JDataStore streams) to copy the streams to another file.

### Removing transaction support

Make a JDataStore non-transactional by assigning a new TxManager that has its enabled property set to false. (It's default value is true.) If the DataStore's consistent property is false, the JDataStore is internally inconsistent and you won't be allowed to make the change. Because you are disabling the TxManager, you don't need to set the userName. This code removes transaction support:

```java
DataStore store = new DataStore();
TxManager txMan = new TxManager();

// Disable TxManager
txMan.setEnabled( false );
store.setFileName( "SomeFileName.jds" );
store.setTxManager( txMan );
store.open();
```

Disabling the TxManager doesn't remove any existing log files. Disabling the TxManager does cause all TxManager properties to be forgotten. If you make the JDataStore transactional again, the TxManager properties revert to their defaults, so if the ALogDir and BLogDir properties were previously set to a non-default value, you will need to remember to set them again.

**DSX:** See Removing transaction support

### Deleting transactional JDataStores

When you delete a transactional JDataStore file, be sure to delete its log files too. If you don’t, you won’t be allowed to create a new JDataStore file with the same name because the log files won’t match.

### Controlling JDataStore transactions

Once you've made a JDataStore transactional, you can usually ignore the TxManager. The only time you need to reference a TxManager is if you want to examine or change the JDataStore'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 have just one `DataStore` object accessing a particular JDataStore file, while you can have many `DataStoreConnection` objects. When you open a `DataStoreConnection`, it contains a reference to a `DataStore` object, as explained in Referencing the connected JDataStore. If there is no suitable `DataStore` object in memory, the `DataStoreConnection` automatically opens a `DataStore` to satisfy this reference. This means that if you open a `DataStoreConnection` first, subsequent `DataStore` objects accessing the same JDataStore file 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 JDataStore uses stream locks to control access to resources. To read a stream or make a change to any part of a stream (a byte in a file, a row in a table), you must be able to acquire a lock on that stream. Once a connection acquires a lock, it holds on to it until the transaction is committed or rolled back.

In single-connection applications, transactions can be considered primarily as a feature that allows you to undo changes and provide crash recovery. Or you might have made a JDataStore transactional so that it can be accessed through JDBC. If you want to access that JDataStore using DataExpress, you must now deal with transactions. How transactions work has deeper ramifications for multi-connection (multi-user or single-user multi-session) applications. These are discussed in Avoiding blocks and deadlocks along with other multi-user issues in Multi-user and remote access to JDataStores.

Committing and rolling back transactions

Controlling transactions uses three methods of `DataStoreConnection`:

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

When you close `DataStoreConnection`, it attempts to commit any pending transaction. You can control this automatic behavior by listening to the `DataStore`'s `Response` event for a `COMMIT_ON_CLOSE`, as shown in the following tutorial.

Tutorial: Controlling transactions via DataExpress

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

**Figure 3.1** The complete AccountsFrame

![Accounts Frame](image)

**Step 1: Create a transactional JDataStore with test data**

You should already have a JDataStore 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 JDataStore files, transactional and
non-transactional, to play with.

Make a copy of the file, naming it `Tx.jds`. Then add the following program to the
project, `MakeTx.java`:

```java
// MakeTx.java
package dsbasic;
import com.borland.datastore.*;
import com.borland.dx.dataset.DataStoreException dse;

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.DataStoreException dse ) {
                    dse.printStackTrace();
                }
            }
        }
    }
```
This utility program makes any JDataStore file transactional if it isn't already. For JDataStores that are already transactional, nothing happens because no properties are set on the TxManager object.

Set the runtime parameters in the Project Properties dialog box to Tx.jds and run the program. It takes a moment to create the three transaction log files TX_STATUS_0000000000, TX_LOGA_ANCHOR, and TX_LOGA_0000000000.

**Step 2: Create a data module**
The next step creates a data module with the JDataStore 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 check box is not selected. Click OK.
4. Switch to design view for the new file AccountsDM.java.
5. Add a DataStore component from the Data Express tab to the component tree. Change its name to dataStore (easily done by pressing F2 after adding it to the tree).
6. In the Inspector, use the file chooser to set the fileName property of the DataStore to the Tx.jds you created earlier, and set the userName property to some name. (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 see the generated code.
10. Save the file.

**Step 3: Create a GUI for the JDataStore table**
Create a simple table grid to display the data:

1. Select File | New.
2. Select 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 check box is selected; deselect the rest. Click Finish.
5. Switch to design view for the new file AccountsFrame.java.
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() is called when the window is closed. It's important that you close the JDataStore before terminating the program.

In this case, with only one connection, the close() method would work, but because you are calling System.exit(), you want to make sure the JDataStore is closed, no matter how many connections the application is using. You should use DataStore.shutdown() in this situation, which closes the JDataStore file directly. That is why this application uses a DataStore instead of a DataStoreConnection.

You could place the shutdown() method call just before System.exit(), but for reasons that you'll see soon, you want to do this before the window physically closes. Insert the highlighted statements:

```java
// Overriden 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 object as dataStore, which hasn’t been defined. First, add the following import statements:

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

16 Declare a new field (after the components is a good place):

```java
TableScrollPane tableScrollPane1 = new TableScrollPane();
JdbTable jdbTable1 = new JdbTable();
DataStore dataStore;
```
Get a reference to the DataStore from the data module. Add the highlighted statement to the jbInit() method:

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

Run AccountsApp.java. You can navigate through the table and add, edit, and delete rows. All the changes you make are done within the context of a single transaction, although you can’t tell at the moment. When you close the window, the JDataStore closes, and the changes you made are committed. You can verify this by running the application again.

If you didn’t close the JDataStore (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 JDataStore. No changes you made in the application would ever apply. Closing the JDataStore 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 actionPerfomed event handler for the commitButton to:

```java
void commitButton_actionPerformed(ActionEvent e) {
    try {
        dataStore.commit();
    } catch (DataSetException dse) {
        dse.printStackTrace();
    }
}
```
14 Set the actionPerfomed event handler for the commitButton to:

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

These buttons now call commit() or rollback() on the DataStore to commit or roll back any changes made during the current transaction. The current transaction is all that happened 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 JDataStore 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:

```java
public class AccountsFrame extends JFrame implements ResponseListener {

2 Add the response method for the ResponseListener interface:

```java
public void response(ResponseEvent response) {
    if (response.getCode() == ResponseEvent.COMMIT_ON_CLOSE) {
        if (JOptionPane.showMessageDialog(this,
            "Posted changes have not been committed. Do that now?",
            "Commit or rollback",
            JOptionPane.YES_NO_OPTION) == JOptionPane.YES_OPTION) {
            response.ok();
        } else {
            response.cancel();
        }
    }
}
```

This method checks for the COMMIT_ON_CLOSE event. When that occurs, a simple yes/no dialog box appears, asking the user if they want to commit the changes. “Yes” sends the ok response, which signals the JDataStore to commit the changes. “No” sends the cancel response, which signals the JDataStore to roll back the changes.

3 Add the highlighted statement to add the frame as one of the JDataStore's ResponseListeners:

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

With these additions, the user gets a dialog box if there are unsaved changes that asks if the user wants to commit them. Remember that the JDataStore is closed before the window is. If it's not, the dialog box would appear after the window had already disappeared.
You can now run the completed application. In addition to using the buttons to commit and roll back changes, try making some changes and then closing the window to exercise the auto-commit handling.

### Using JDBC for data access

You can access JDataStore tables with JDataStore's Type 4 (direct all Java) JDBC driver, `com.borland.datastore.jdbc.DataStoreDriver`.

You can use this driver for both local and remote access. Remote access requires a JDataStore Server, which is also used for multi-user access. For details on remote access and multi-user issues, see Multi-user and remote access to JDataStores

This is the local connection URL:

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

As with any JDBC driver, you can use the JDBC API or an added-value API such as DataExpress with `QueryDataSet` and `ProcedureDataSet` to access tables.

### Demonstration class: JdbcTable.java

The following program, `JdbcTable.java`, is functionally identical to its DataExpress twin, Demonstration class: DxTable.java. It uses the JDBC API.

```java
// 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"
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. Initialize these prepared statements before calling the appendRow() method. A good place to do this is in the class constructor. Because the constructor is used, the organization of the code is a little different than in DxTable.java.

The first thing that happens in the class constructor is the loading of the JDataStore JDBC driver using Class.forName. Both the driver name and the beginning of the connection URL are defined as class variables for convenience. A Connection toTx.jds is created, and from that, a generic Statement.

The next step is to determine if the table exists. You can do this using 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 you must create it by calling the createTable() method. The createTable() method uses a SQL CREATE TABLE statement. Note that the SQL parser usually converts identifiers to uppercase. To keep the proper casing used by DxTable.java, enclose the identifiers in quotes in this and other SQL statements. Finally, the two prepared statements are created.

The demo() method calls appendRow() to add a couple of test rows. As in DxTable.java, the last/largest ID value is retrieved and incremented for the new row. But instead of using a sort order and going to the last row, the JDBC approach uses an SQL SELECT statement that fetches the maximum value. As a result, the empty table condition, when there is no last value, must be handled specifically.

Finally, the contents of the table are displayed using an SQL SELECT statement to fetch the rows and a loop that’s very similar to the one in DxTable.java. The statement and connection are closed as required by JDBC.

You can run this program interchangeably with DxTable.java. Both of them add two more test rows to the same table.

### Controlling transactions through JDBC

Each JDBC connection actually uses its own internal instance of DataStoreConnection for the connection. That’s how the JDataStore JDBC driver is implemented. But this internal object is not accessible, so you must use the JDBC API.

For control over transactions, disable auto-commit mode by calling Connection.setAutoCommit(false). You can then call commit() and rollback() on the Connection object.
Chapter 4

Using JDataStore's security features

JDataStore provides several built-in security features. This collection of features provides user authentication, user authorization, and encryption for JDataStore databases.

User authentication

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

The SYS/USERS table is read-only if it is accessed by a JDBC query or a StorageDataSet. When users are added, their initial password and database rights are supplied.

There are three methods that can be used to administer users. DataStoreRights.ADMINISTRATOR rights are required to call these methods.

- Users can be added by calling the DataStoreConnection addUser() method.
- A user can be removed by calling the DataStoreConnection removeUser() method.
- A user's rights can be changed by calling the DataStoreConnection changeRights() method.

DSX: See Administering users for an explanation of administering users using the JDataStore Explorer.

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

DSX: For instructions on changing a password through the JDataStore Explorer, see Changing a password.
Authorization

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

- **STARTUP** - the ability to open a database that is shutdown. The user's password is required to add `STARTUP` rights to a user. The `DataStoreConnection.changeRights()` method's `pass` parameter must not be null, and must match the user's password when calling this method to add `STARTUP` rights. `STARTUP` rights can also be specified when the user is added.

- **ADMINISTRATOR** - includes rights to add, remove, and change rights of users, and the ability to encrypt the database. Also includes the four stream rights (WRITE, CREATE, DROP, RENAME). `STARTUP` rights are given to administrators by default when the administrator is added, but `STARTUP` rights can be removed from an administrator. `WRITE`, `CREATE`, `DROP`, and `RENAME` rights cannot be removed from an administrator. Any attempt to remove these rights from an administrator will be ignored.

- **WRITE** - the ability to write to file or table streams in the JDataStore.

- **CREATE** - the ability to create new file or table streams in the JDataStore.

- **DROP** - the ability to remove file or table streams from the JDataStore.

- **RENAME** - the ability to rename file or table streams in the JDataStore.

JDataStore Encryption

A `DataStoreRights.ADMINISTRATOR` user can encrypt an empty database that has a `DataStoreConnection.getVersion()` of `DataStore.VERSION_6_0` or greater. The `DataStoreConnection.encrypt()` method can be used to encrypt databases. `DataStoreConnection.encrypt()` will remove `DataStoreRights.STARTUP` from all users except for the administrator that is adding the encryption. This is because the user's password is required to add `STARTUP` rights to a user. To provide `STARTUP` rights to a user, call `DataStoreConnection.changeRights()`, or drop the user and then add the user back.

*Note:* A database with existing table or file streams will not be encrypted. If you want to encrypt an existing database, create a new database, call `DataStoreConnection.copyUsers()` to copy the existing users to the new database, then go ahead and encrypt the new database. Then call `DataStoreConnection.copyStreams()` to copy the contents of the old database into the encrypted database. For more information on copying streams see Copying JDataStore streams.

**DSX:** See Encrypting a JDataStore for information on how to encrypt a JDataStore using the JDataStore Explorer.
Deciding how to apply JDataStore security

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

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

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

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

- If a JDataStore database file is not password protected, and it is possible for an opponent to write to it with a separate file editing utility or program, the authentication and authorization support can be disabled.

- If it is possible for an opponent to read a JDataStore database file that is not encrypted with a separate file editing utility or program, the opponent might be able to reverse engineer the file format and view its contents.

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

Warning: The cryptographic techniques that JDataStore uses to encrypt data blocks is state of the art. The TwoFish block cypher used by JDataStore has never been defeated. So this means that if you forget your password for an encrypted JDataStore database, you are really out of luck. The best chance of recovering the data would be to have someone guess the password.

There are measures that can be used to guard against forgetting a password for an encrypted database. It is important to note that there is a master password used internally to encrypt data blocks. Any user that has STARTUP rights will have the master password encrypted using their password in the SYS/USERS table. This allows one or more users to open a database that has been shutdown because their password can be used to decrypt a copy of the master password. This feature can be used to create a virgin database that has one secret user in it that has DataStoreRights.ADMINISTRATOR rights (which includes DataStoreRights.STARTUP rights). If this virgin database is used whenever a new empty database is needed, you will always have one secret user that can unlock the encryption.
Encrypting a database will affect performance somewhat. Data blocks are encrypted when they are written from the JDataStore cache to the JDataStore database file. Data blocks are decrypted when they are read from the JDataStore database file into the JDataStore cache. So the cost of encryption is only incurred when file I/O is performed.

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

Note that the remote JDBC driver for JDataStore currently uses Java socket classes to communicate with a JDataStore Server. This communication is not secure. Since the local JDBC driver for JDataStore is in-process, it is secure.
Chapter 5

Multi-user and remote access to JDataStores

JDataStore applications aren't limited to accessing local JDataStore files. Through the use of a JDataStore Server, you can access JDataStores from other machines. The JDataStore Server also enables multi-user access. Access requests from numerous clients are handled by the server process.

Making a Local JDBC Connection

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

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

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

Specifying extended properties

The API documentation for the com.borland.datastore.jdbc.cons.ExtendedProperties interface documents how to specify extended properties and the latest collection of extended properties that can be set for a JDBC connection.
Using the JDBC driver for remote access

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

jdbc:borland:dslocal:<filename>

But the URL for remote connections is:

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

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

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

```java
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 "Driver properties" for the DataStoreDriver component in 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

Because JDataStores accessed through a remote and local connection are potentially JDataStores accessed by multiple users, you must consider some concurrency issues as detailed in Avoiding blocks and deadlocks

Running the JDataStore Server

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

For development purposes, you can start the server from the JBuilder menu. Select Tools | JDataStore Server. This starts the server process, displaying a simple UI:
From the menu, you can get help, terminate the server, or run the JDataStore Explorer (for more information on the JDataStore Explorer, see Using the JDataStore Explorer). From the Options tab, you can change a few items in the server’s configuration.

Reconfiguring the server

You can’t change the server configuration while it is running. The File | Shutdown menu command stops the actual server process and closes all connections in an orderly fashion.

Once the server stops, you can use the Options tab to change the port number, temp directory, and status log directory settings for the server. Then use File | Startup to restart the server process.

Deploying the JDataStore Server

When development ceases on an application and it enters production, you must deploy the JDataStore Server to a server machine.

Packaging the server

The JAR files you need to run the server depend on whether you want the GUI (graphical user interface). Without the GUI, you need just these JAR files:

- jdsserver.jar
- jds.jar
- dx.jar

If you want the GUI, you also need these JAR files:

- dbswing.jar
- dbtools.jar

If you want online help for the GUI to be available, you also need this JAR file:
help.jar

If you want the server's online help to be available, you must also copy the help JAR files from the /doc directory. The help files for the JDataStore Server are in jb_ui.jar. The JDataStore Developer's Guide is in jb_dspg.jar.

Starting the server

To start the server, you must add the necessary JAR files to the classpath. The main class for the JDataStore Server is com.borland.dbtools.dsserver.Server, so the command line for starting the server with default options is this:

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

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

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-port=&lt;number&gt;</td>
<td>The port to listen to. Default: 2508</td>
</tr>
</tbody>
</table>
| -ui=<uiType>  | The look and feel of the UI. One of the following:
|              | 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         |

If you don't use the -ui option (or specify none), the server starts as a console application. Without a UI, you won't be able to reconfigure the server or launch the JDataStore 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.

An executable file for the server is also provided, and it can be started from the command line. The executable uses the classpath and main class name settings in the dss.config file.

Creating custom JDBC servers

The JDataStore Server that comes with JBuilder provides remote access to JDataStore 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 JDataStore. File streams are not accessible through JDBC.

The core of any JDataStore Server is the com.borland.datastore.jdbc.DataStoreServer class. For more information, see the DataStoreServer class in the DataExpress Component Library Reference.
Transactions and connection pooling

Multi-user transaction issues

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

Transaction isolation levels

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

The serializable isolation level designated by java.sql.Connection.TRANSACTION_SERIALIZABLE provides complete isolation for a transaction. An application may choose a weaker isolation level to improve performance or to avoid lock manager deadlocks. Weaker isolation levels are susceptible to one or more of the following isolation violations:

- Dirty reads 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.
SQL92 defines four levels of isolation in terms of the behavior that a transaction running at a particular isolation level is permitted to experience as shown in the following table.

**Table 6.1 SQL Isolation Level Definitions**

<table>
<thead>
<tr>
<th>Isolation level</th>
<th>Dirty Read</th>
<th>NonRepeatable Read</th>
<th>PhantomRead</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read uncommitted</td>
<td>Possible</td>
<td>Possible</td>
<td>Possible</td>
</tr>
<tr>
<td>Read committed</td>
<td>Not Possible</td>
<td>Possible</td>
<td>Possible</td>
</tr>
<tr>
<td>Repeatable read</td>
<td>Not possible</td>
<td>Not possible</td>
<td>Possible</td>
</tr>
<tr>
<td>Serializable</td>
<td>Not possible</td>
<td>Not possible</td>
<td>Not possible</td>
</tr>
</tbody>
</table>

**Setting isolation levels on JDataStore connections**

To set the isolation levels on JDataStore connections:

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

With both of the above methods, the `level` parameter can be one of the four following values:

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

To choose an isolation level, refer to the following table

**Table 6.2 JDataStore connection isolation levels**

<table>
<thead>
<tr>
<th>Isolation level</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRANSACTION_READ_UNCOMMITTED</td>
<td>Suitable for single user applications for reports that do not mind transactionally inconsistent views of the data. Especially useful when browsing JDataStore tables with dbswing and DataExpress DataSet components. This isolation level incurs minimal locking overhead.</td>
</tr>
<tr>
<td>TRANSACTION_READ_COMMITTED</td>
<td>Commonly used for high performance applications. Ideal for data access models that use Optimistic concurrency control. DataExpress QueryDataSet and Borland Application Server use Optimistic concurrency control approaches to data access. In these data access models, read operations are largely performed first. In some cases, read operations are actually performed in a separate transaction from the write operations.</td>
</tr>
</tbody>
</table>
Locks used by the JDataStore lock manager

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

Table 6.3   Locks used by the JDataStore lock manager:

<table>
<thead>
<tr>
<th>Lock</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Critical Section locks</td>
<td>Internal locks used to protect internal data structures. These locks are usually held for a short period of time. They are acquired and released independent of when the transaction is committed.</td>
</tr>
<tr>
<td>Row locks</td>
<td>Shared and exclusive lock modes supported. These locks are released when the transaction commits.</td>
</tr>
<tr>
<td>Table locks</td>
<td>Shared and exclusive lock modes supported. These locks are released when the transaction commits.</td>
</tr>
<tr>
<td>DDL Table locks</td>
<td>Shared and exclusive lock modes supported:</td>
</tr>
<tr>
<td></td>
<td>• Shared DDL locks are held by transactions that have tables opened. Shared DDL locks are not released until first, the transaction commits, and second, the connection closes the table and all statements that refer to the table.</td>
</tr>
<tr>
<td></td>
<td>• Exclusive DDL locks are used when a table must be dropped or structurally modified and are released when a transaction commits.</td>
</tr>
</tbody>
</table>

Extended JDBC Properties that control JDataStore locking behavior

You specify extended JDBC properties by setting them in a java.util.Properties object that is passed when creating a JDBC connection.
Property names are case sensitive and are described in the following table.

Table 6.4  Extended JDBC Properties that control locking behavior

<table>
<thead>
<tr>
<th>Property</th>
<th>Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>tableLockTables</td>
<td>String of semicolon-separated table names. The table names are case-sensitive. Row locking will not be used for tables specified in this list. To specify all tables, set this property to “*”. This property can also be set for DataStoreConnection components by calling the <code>setTableLockTables()</code> method.</td>
</tr>
<tr>
<td>maxRowLocks</td>
<td>Maximum number of row locks per table a transaction should acquire before escalating to a table lock. The default value is 50. This property can also be set for DataStoreConnection components by calling the <code>setMaxRowLocks()</code> method.</td>
</tr>
<tr>
<td>lockWaitTime</td>
<td>Maximum number of milliseconds to wait for a lock to be released by another transaction. When this timeout period expires, an appropriate exception is thrown. This property can also be set for DataStoreConnection components by calling the <code>setLockWaitTime()</code> method.</td>
</tr>
<tr>
<td>readOnlyTxDelay</td>
<td>Maximum number of milliseconds to wait before starting a new read-only view of the database. For more information, see the discussion of the <code>java.sql.Connection.readOnly()</code> property in the section Optimizing JDataStore applications.</td>
</tr>
</tbody>
</table>

**JDataStore lock usage and isolation levels**

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

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

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

Table 6.5  Lock usage and isolation levels

<table>
<thead>
<tr>
<th>Connection isolation level</th>
<th>Row locking behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRANSACTION_READ_UNCOMMITTED</td>
<td>Does not acquire row locks for read operations. It also ignores exclusive row locks held by other connections that have inserted or updated a row.</td>
</tr>
<tr>
<td>TRANSACTION_READ_COMMITTED</td>
<td>Does not acquire row locks for read operations. A transaction using this isolation level will block when reading a row that another transaction has an exclusive row lock due to insert or update operation. This block will terminate when the write transaction commits, a deadlock is detected or the lockTimeOut time limit has expired.</td>
</tr>
</tbody>
</table>
Note that lock escalation from row locks to table locks occurs in some situations for TRANSACTION_SERIALIZABLE as described above, but it also occurs for all isolation levels if the maxRowLocks property is exceeded.

## Debugging lock timeouts and deadlocks

Locks can fail due to a lock timeout or deadlock. Lock timeouts occur when a connection waits to acquire a lock held by another transaction longer than the milliseconds set in the lockWaitTime property. In such cases, an exception is thrown that identifies which connection encountered the timeout and which connection is currently holding the needed lock. The transaction that encounters the lock timeout is not rolled back.

JDataStore has automatic, high speed deadlock detection that should detect almost all deadlocks. It is possible, but unlikely that a deadlock situation will be reported as a locktimeout. An appropriate exception is thrown that identifies which connection encountered the deadlock and which connection it is deadlocked with. Unlike lock timeout exceptions, deadlock exceptions encountered by a java.sql.Connection cause that connection to automatically rollback its transaction. This behavior allows other connections to continue their work.

Use the following guidelines to detect timeout and deadlock situations:

- View the message from timeout and deadlock exceptions. It has information on what table the error occurred and what connections where involved.
- Set the java.sql.DriverManager.SetLogWriter() property to a log writer stream. To minimize the log output to just lock related logging, set the extendedlogFilter property to LOCK_ERRORS.
- Use the DataStoreConnection.dumpLocks() method to report locks held by all connections.
Avoiding blocks and deadlocks

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

- By minimizing the lifespan of transactions that write.
- Using read-only transactions that don’t require locks to read.

Conserving write transactions

Connections should use burst writes. A burst write accumulates changes, and then when necessary, starts a transaction, immediately writes those changes, and commits them. This is the preferred model for most database servers and it’s the model used by DataExpress in its provider/resolver paradigm.

Using read-only transactions

Read-only transactions aren’t blocked by writers or other readers, and because they don’t get locks, they never block other transactions.

To make JDBC connections use read-only transactions, set the `readOnly` property of the `java.sql.Connection` object (returned by the `java.sql.DriverManager.getConnection()` and `com.borland.dx.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 JDataStore. The snapshot sees only data from transactions committed at the point the transaction started (otherwise, the connection would have to see if there are pending changes and roll them back whenever it accesses the data). A snapshot begins when the `DataStoreConnection` opens. It’s refreshed every time its `commit()` method is called.

Concurreny control changes for earlier versions

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

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

Connection pooling and distributed transaction support

JDataStore provides several components for dealing with JDBC 2.0 `DataSource`, connection pooling, and distributed transaction (XA) support. These features require J2EE. If your version of JBuilder does not include `J2EE.jar`, you will need to
Connection pooling

The basic idea behind connection pooling is simple. In an application that opens and closes many database connections, it is efficient to keep unused Connection objects in a pool for future re-use. This saves the overhead of having to open a new physical connection each time a connection is opened.

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

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

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

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

heuristicCompletion

Beginning with version 6.0, JDataStore provides heuristicCompletion, an extended JDBC property that allows you to control the behavior when one or more databases fail during a two-phase commit. When XA transactions are prepared, but not finished (no commit or rollback) you can specify one of three possible string settings for this property:

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

Note that the heuristic commit and rollback options allow for more efficient execution, because locks can be released sooner and less information needs to be written to the transaction log file.
Chapter 7

UDFs and Stored Procedures

Stored procedures are user-defined functions that are designed to handle business logic. These functions serve to hide the complexity of dealing with relational tables. Stored procedures are called directly, and optionally have input and or output parameters. For example:

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

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

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

Language for stored procedures and UDFs

Stored procedures and UDFs for JDataStore must be written in Java. There is no security against bad behaviors in the written Java code. However, the compiled Java classes must be added to the classpath of the JDataStore server process in order to be available for use. This should give the database administrator a chance to control which code is added. Only public static methods in public classes can be made available for use.

Making a stored procedure or UDF available to the SQL engine

After a stored procedure or a UDF has been written and added to the classpath of the JDataStore server process, a function name is associated with that Java method using the following SQL syntax:

```sql
CREATE JAVA_METHOD <method-name> AS <method-definition-string>
```
where the `<method-name>` is a SQL identifier such as `INCREASE_SALARY` and `<method-definition-string>` is a string with a fully qualified method name, such as `com.mycompany.util.MyClass.increaseSalary`.

A stored procedure or a UDF can be dropped from the database by executing:

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

After a method is created, it is ready for use. The following section provides an example of how to define and call a UDF.

### A UDF example

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

CREATE JAVA_METHOD FIND_NEXT_SPACE AS 'com.mycompany.util.MyClass.findNextSpace';
SELECT * FROM TABLE1 WHERE FIND_NEXT_SPACE(FIRST_NAME, CHAR_LENGTH(LAST_NAME)) < 0;
```

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

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

### Input parameters

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

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

The other recognized Java types are:

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

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

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

Output parameters

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

- double[] or Double[]
- float[] or Float[]
- java.math.BigDecimal[]
- long[] or Long[]
- int[] or Integer[]
- short[] or Short[]
- Byte[] (but not byte[] since that is a recognized input parameter by itself)
- boolean[] or Boolean[]
- String[]
- java.sql.Date[]
- java.sql.Time[]
- java.sql.Timestamp[]
- byte[][]
- java.io.InputStream[]
Output parameters can be bound only to variable markers in SQL. All output parameters are essentially INOUT parameters, since any value set before the statement is executed is passed to the Java method. If no value is set, the initial value is arbitrary. If any of the parameters can output a SQL NULL (or have a valid NULL input), use the encapsulated classes instead of the primitive types. For example:

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

```
CREATE JAVA_METHOD MAX AS
    'com.mycompany.util.MyClass.max';
CALL MAX(1,2,3,?);
```

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

### Implicit connection parameter

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

Do not pass anything for this parameter: let JDataStore do it.

An example:

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

```
CREATE JAVA_METHOD INCREASE_SALARY AS
    'com.mycompany.util.MyClass.increaseSalary';
CALL INCREASE_SALARY(20000.00);
```

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

**Overloaded method signatures**

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

An example:

```java
package com.mycompany.util;
public class MyClass {
    public static int abs(int p) {
        return Math.abs(p);
    }
    public static long abs(long p) {
        return Math.abs(p);
    }
    public static BigDecimal abs(java.math.BigDecimal p) {
        return p.abs();
    }
    public static double abs(double p) {
        return Math.abs(p);
    }
}
CREATE JAVA_METHOD ABS_NUMBER AS 'com.mycompany.util.MyClass.abs';
SELECT * FROM TABLE1 WHERE ABS(NUMBER1) = 2.1434;
```

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

**Return type mapping**

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

<table>
<thead>
<tr>
<th>Return type of method</th>
<th>JDataStore SQL type</th>
</tr>
</thead>
<tbody>
<tr>
<td>byte or Byte</td>
<td>SMALLINT</td>
</tr>
<tr>
<td>short or Short</td>
<td>SMALLINT</td>
</tr>
</tbody>
</table>
Return type mappings (continued)

<table>
<thead>
<tr>
<th>Return type of method</th>
<th>JDataStore SQL type</th>
</tr>
</thead>
<tbody>
<tr>
<td>int or Integer</td>
<td>INT</td>
</tr>
<tr>
<td>long or Long</td>
<td>BIGINT</td>
</tr>
<tr>
<td>java.math.BigDecimal</td>
<td>DECIMAL</td>
</tr>
<tr>
<td>float or Float</td>
<td>REAL</td>
</tr>
<tr>
<td>double or Double</td>
<td>DOUBLE</td>
</tr>
<tr>
<td>String</td>
<td>VARCHAR</td>
</tr>
<tr>
<td>boolean or Boolean</td>
<td>BOOLEAN</td>
</tr>
<tr>
<td>java.io.InputStream see (*)</td>
<td>INPUTSTREAM</td>
</tr>
<tr>
<td>java.sql.Date</td>
<td>DATE</td>
</tr>
<tr>
<td>java.sql.Time</td>
<td>TIME</td>
</tr>
<tr>
<td>java.sql.Timestamp</td>
<td>TIMESTAMP</td>
</tr>
<tr>
<td>All other types:</td>
<td>OBJECT</td>
</tr>
</tbody>
</table>

*Note* Any type derived from `java.io.InputStream` is handled as an `INPUTSTREAM`. 
Chapter 8

Persisting data in a JDataStore

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, it's stored in memory through the use of a MemoryStore. You can use alternate storage systems such as the JDataStore by setting the StorageDataSet object's store property. (Currently MemoryStore and DataStore/DataStoreConnection are the only implementations of the Store interface required by the store property.)

The main advantages of DataStore over MemoryStore are transaction semantics, SQL support, and persistence, which enable offline computing. A DataStore remembers the rows fetched in a table, even after the application terminates and restarts. In addition, you can 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. JDataStore stores data and indexes in an extremely compact format, however. As the number of rows in a StorageDataSet increases, using a DataStore provides much better performance and requires considerably less memory than using a MemoryStore.

Whether the data’s storage is MemoryStore or DataStore often doesn’t affect how you work with a StorageDataSet or other data-aware controls connected to the StorageDataSet. Storing Java objects in columns, however, does require you to use Java serialization (java.io.Serializable). If this isn’t possible, you can’t use DataStore components and you should use the default in-memory storage mechanism.
Using a JDataStore with StorageDataSets

You cache and persist StorageDataSets in a JDataStore by setting the values of the three properties discussed in Connecting to a JDataStore with StorageDataSet. 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 JDataStore 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 display the Open dialog box and enter the JDataStore 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 JDataStore.
6. In the Inspector, set the store property of the StorageDataSet to the DataStoreConnection component.

Tutorial: Offline editing with JDataStore

This tutorial presents the steps for creating an application that uses a DataStore component to enable the offline editing of data. The server database is a sample JDataStore file, employee.jds, accessed through the JDataStore Server. Don't confuse this file with the JDataStore used for persistence. Locate the sample file before beginning. It's installed in samples/JDataStore/datastores.

1. Start the JDataStore Server from the menu item Tools|JDataStore Server.
2. Create a new application by selecting File|New from the menu and doubleclicking 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. Click Finish
3. Switch to design view for the newly created PersistFrame.java.
4. Add a Database component from the Data Express tab to the component tree.
5 Open the connection property editor for the Database component in the Inspector. Set the connection properties to the database, using the correct path to the sample employee.jds file in place of <drive letter>:/<jbuilder> in the URL:

<table>
<thead>
<tr>
<th>Property name</th>
<th>Setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>Driver</td>
<td>com.borland.datastore.jdbc.DataStoreDriver</td>
</tr>
<tr>
<td>URL</td>
<td>jdbc:borland:dsremote://localhost/&lt;drive letter&gt;:/&lt;jbuilder&gt;/samples/JDataStore/datastores/employee.jds</td>
</tr>
<tr>
<td>Username</td>
<td>&lt;use any name&gt;</td>
</tr>
<tr>
<td>Password</td>
<td>&lt;leave blank&gt;</td>
</tr>
</tbody>
</table>

6 Click the Test Connection button to check that you’ve set the connection properties correctly. When the connection is successful, click OK.

7 Add a DataStoreConnection component from the Data Express tab to the component tree. Adding a DataStoreConnection component writes an import statement for the datastore package to your code and adds the JDataStore library to your project properties if it wasn’t already listed.

8 Open the fileName property editor for the DataStoreConnection component. Type in the name for a new JDataStore file. Be sure to include the full path. You can use the Browse button to help. You don’t have to specify a file extension because a JDataStore always has the extension .jds. Click OK.

Note: The Designer automatically creates this JDataStore file for you when it’s connected to the StorageDataSet so that the tools work fully. When you run the application, the JDataStore file will already be there. But if you run the application on another computer, the JDataStore file won’t be there. You will have to add extra code to create the JDataStore file if necessary as shown in Creating a JDataStore file

9 Add a QueryDataSet component from the Data Express tab to the component tree.

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

<table>
<thead>
<tr>
<th>Property name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Database</td>
<td>database</td>
</tr>
<tr>
<td>SQL Statement</td>
<td>select * from employee</td>
</tr>
</tbody>
</table>

11 Click Test query to ensure that the query is runnable. When the gray area beneath the button indicates Success, click OK to close the dialog box.

12 Set the storeName property of the QueryDataSet to employeeData.

13 Set the store property to dataStoreConnection1 (the only choice).

14 Add a JdbNavToolBar component from the dbSwing tab to the North position of the frame. Set its dataSet property to queryDataSet1.
15 Add a JdbStatusLabel component from the dbSwing tab to the South position of the frame. Set its dataSet property to queryDataSet1.

16 Add a JScrollPane component from the dbSwing tab to the Center position of the frame.

17 Add a JdbTable component from the dbSwing tab to the JScrollPane. Set its dataSet property to queryDataSet1.

18 Instead of adding code to call DataStore.shutdown() before exiting the application (as you did in an earlier tutorial), you can use a DBDisposeMonitor component to close JDataStore files automatically when you close the frame.

19 Add a DBDisposeMonitor component from the More dbSwing tab to the component tree. Set its dataAwareComponentContainer property to this.

20 Run PersistApp.java.

In the running application, make some changes to the data and click the Post button on the navigator to save the changes to the JDataStore file (the persistence JDataStore specified in step Open the fileName property editor for the DataStoreConnection component. Type in the name for a new JDataStore file. Be sure to include the full path. You can use the Browse button to help. You don’t have to specify a file extension because a JDataStore always has the extension .jds. Click OK.). Changes are also saved to the file when you move off a row, just as they are with an in-memory data set (MemoryStore).

Note: A data set in a JDataStore can have tens or hundreds of thousands of rows. Handling that much data using an in-memory data set would greatly slow application performance.

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

**Understanding how JDataStore manages offline data**

So far in the tutorial, nothing has been saved back to the SQL database on the server. On the JdbNavToolBar, there are several buttons:

- The Post button saves the changes in the current row to the JDataStore file.
- The Save button saves all changes that have been accumulated in the JDataStore 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 JDataStore with the results of the query, including any edits not saved back to the server. In code, the corresponding method is DataSet.executeQuery().

8-4 The JDataStore 6 Developer’s Guide
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 JDataStore 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 JDataStore 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.
- You don't need to write special code to get data into the JDataStore on the first run.
- Once data is in the JDataStore, 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.

When the Execute Query Immediately When Opened option is selected, existing data can't be overwritten (unless you call the `StorageDataSet.refresh()` method explicitly). This means that you can safely close and reopen a data set to change property settings in either a `MemoryStore` or in a `DataStore` without losing editing changes.

Once you've got data in the JDataStore 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. If you do, you'll get an exception because the attempt to connect will fail, but you won't lose any of the changes you have made.

### Restructuring JDataStore StorageDataSets

The Column Designer in the JBuilder UI designer provides support for moving, deleting, and inserting columns. You can also change Column data types. As the data type of a column component in a `StorageDataSet` that is bound to a JDataStore changes, 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.

The Column 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` doesn't convert data values to the new data type. The old values are lost.

JDataStore doesn't perform the move/insert/delete/change type operations on `StorageDataSets` immediately. The structural change is noted inside the JDataStore directory as a pending operation. The `StorageDataSet.getNeedsRestructure()` method returns `true` when there is a pending restructure operation. You can still use a `StorageDataSet` with pending structural changes.
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 force the restructure operation to happen with code by calling the `StorageDataSet.restructure()` method.

You can use the `restructure()` method 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 `JDataStore` 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`.

<table>
<thead>
<tr>
<th>From</th>
<th>To</th>
<th>Big Decimal</th>
<th>Double</th>
<th>Float</th>
<th>Long</th>
<th>Int</th>
<th>Short</th>
<th>Boolean</th>
<th>Time</th>
<th>Date</th>
<th>Timestamp</th>
<th>String</th>
<th>InputStream</th>
<th>Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>Big Decimal</td>
<td>None</td>
<td>Prec</td>
<td>Prec</td>
<td>Prec</td>
<td>Prec</td>
<td>Prec</td>
<td>Prec</td>
<td>Prec</td>
<td>Prec</td>
<td>OK</td>
<td>Loss</td>
<td>Loss</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Double</td>
<td>Prec</td>
<td>None</td>
<td>Prec</td>
<td>Prec</td>
<td>Prec</td>
<td>Prec</td>
<td>Prec</td>
<td>Prec</td>
<td>Prec</td>
<td>OK</td>
<td>Loss</td>
<td>Loss</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Float</td>
<td>OK</td>
<td>None</td>
<td>Prec</td>
<td>Prec</td>
<td>Prec</td>
<td>Prec</td>
<td>Prec</td>
<td>Prec</td>
<td>Prec</td>
<td>OK</td>
<td>Loss</td>
<td>Loss</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Long</td>
<td>OK</td>
<td>Prec</td>
<td>None</td>
<td>Prec</td>
<td>Prec</td>
<td>Prec</td>
<td>Prec</td>
<td>Prec</td>
<td>Prec</td>
<td>OK</td>
<td>Loss</td>
<td>Loss</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Int</td>
<td>OK</td>
<td>OK</td>
<td>Prec</td>
<td>OK</td>
<td>None</td>
<td>Prec</td>
<td>Prec</td>
<td>Prec</td>
<td>Prec</td>
<td>OK</td>
<td>Loss</td>
<td>Loss</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Short</td>
<td>OK</td>
<td>OK</td>
<td>OK</td>
<td>OK</td>
<td>None</td>
<td>Prec</td>
<td>Prec</td>
<td>Prec</td>
<td>Prec</td>
<td>OK</td>
<td>Loss</td>
<td>Loss</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Boolean</td>
<td>OK</td>
<td>OK</td>
<td>OK</td>
<td>OK</td>
<td>OK</td>
<td>None</td>
<td>Prec</td>
<td>Prec</td>
<td>Prec</td>
<td>OK</td>
<td>Loss</td>
<td>Loss</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Time</td>
<td>Prec</td>
<td>Prec</td>
<td>Prec</td>
<td>Prec</td>
<td>Prec</td>
<td>Prec</td>
<td>Prec</td>
<td>Prec</td>
<td>Prec</td>
<td>OK</td>
<td>Loss</td>
<td>Loss</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Date</td>
<td>Prec</td>
<td>Prec</td>
<td>Prec</td>
<td>Prec</td>
<td>Prec</td>
<td>Prec</td>
<td>Prec</td>
<td>None</td>
<td>None</td>
<td>OK</td>
<td>Loss</td>
<td>Loss</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Timestamp</td>
<td>Prec</td>
<td>Prec</td>
<td>Prec</td>
<td>Prec</td>
<td>Prec</td>
<td>Prec</td>
<td>Prec</td>
<td>None</td>
<td>None</td>
<td>OK</td>
<td>Loss</td>
<td>Loss</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Here is a legend to the table:

- **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, such as QueryDataSet and ProcedureDataSet, obtain their structure and row data from a SQL server. But sometimes you need to add columns to these DataSets or to create standalone tables. You can do this with the JBuilder Column Designer. Use it to change, delete, move, and add DataSet columns instantly at design time. The JDataStore uses a mapping table to provide an illusion that each structural operation has occurred. When you’ve made all the structural changes, click Restructure and the restructure occurs.

When the Designer is activated while a StorageDataSet node is selected, the columns for that node are displayed in a grid on the design surface. A toolbar for adding, deleting, moving up, moving down, and restructuring the data set appears 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 JDataStore associated with the data set. While the restructure is running, a dialog box appears that shows the status of the restructure. It has a Cancel button that aborts the restructure. Restructure is available only if the dataset’s store property has been set to a DataStore or DataStoreConnection component (see Using a JDataStore with StorageDataSets).

Understanding structure changes

Regardless of whether a JDataStore is used, columns provided by a query or procedure must be merged with columns defined in code. When a JDataStore 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 JDataStore are the same as those provided by the merge. But when you are developing an application, you might make changes. Even production applications change occasionally. You can continue to use a data set in a JDataStore when the columns in the JDataStore aren’t the same as those from the application/provider merge. The provide operation automatically restructures the table after merging.

If you change a data set’s structure in the Designer instead of editing the source code, the JDataStore tracks the changes. This makes the merge between application/provider columns and JDataStore 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 JDataStore can map the old column name to the new one. If you make the same change in code, the JDataStore can only determine that one column was deleted and another was added. The
JDataStore won't recognize that the column was renamed. So the deleted column won't 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 JDataStore.
- 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 are rebuilt when its needed.
Using the all-Java JDataStore Explorer, you can:

- Examine the contents of a JDataStore. The JDataStore's directory is shown in a tree control with each table and its indexes grouped together. When you select a data stream in the tree, its contents appear (assuming it's a file type such as a text file, GIF image, or table for which the Explorer has a viewer).

- Perform many JDataStore operations without writing code. You can create a new JDataStore file, create tables and indexes, import delimited text files into datasets, import files as file streams, delete tables and indexes, delete datasets or other data streams, and verify the integrity of the JDataStore.

- Manage queries that provide data into datasets in the JDataStore, edit the datasets, and save changes back to server tables.

- Administer security features of a JDataStore, such as users, passwords, and encryption.

Launching the JDataStore Explorer

There are three ways to launch the JDataStore Explorer:

- From JBuilder use the Tools | JDataStore Explorer menu command.

- From the JDataStore Server (see Running the JDataStore Server) use the File | JDataStore Explorer menu command.

- From the command line or with a shortcut.
Starting the JDataStore Explorer from the command line

JDataStore Explorer requires the following JARs:

- jds.jar
- dx.jar
- dbswing.jar
- dbtools.jar

These files are also required if you want the online help to be available:

- jb_ui.jar
- help.jar

An executable file for the JDataStore Explorer is provided, and it can be started from the command line. The executable uses the classpath and main class name settings in the dse.config file.

The main class for the JDataStore Explorer is com.borland.dbtools.dsx.DataStoreExplorer. The command line for starting the server with default options (after setting the classpath) is:

```
java com.borland.dbtools.dsx.DataStoreExplorer
```
You can also specify the options listed in the following table:

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-ui=&lt;uiType&gt;</td>
<td>The look and feel of the UI. One of the following:</td>
</tr>
<tr>
<td></td>
<td>• windows</td>
</tr>
<tr>
<td></td>
<td>• motif</td>
</tr>
<tr>
<td></td>
<td>• metal</td>
</tr>
<tr>
<td></td>
<td>• none</td>
</tr>
<tr>
<td></td>
<td>• &lt;LookAndFeel class name&gt;</td>
</tr>
<tr>
<td>-h=&lt;helpDir&gt;</td>
<td>The directory that contains online help files</td>
</tr>
<tr>
<td>&lt;.jds filename&gt;</td>
<td>A JDataStore file to open on startup</td>
</tr>
<tr>
<td>-?</td>
<td>Displays a message listing these options</td>
</tr>
</tbody>
</table>

**Basic JDataStore operations**

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

**Creating a new JDataStore file**

To create a new JDataStore file,

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

   ![New JDataStore dialog box](Figure 9.2)

2. Enter a name for the new JDataStore file.
3. (Optional) Choose a block size other than the default 4KB.
4. Make sure the Install TxManager check box is set properly:
   - For a non-transactional JDataStore, the check box should be empty
For a transactional JDataStore, the check box should be checked. You can click Properties to set the transaction management properties for the new JDataStore file.

5 Click OK. The store is created and opened in the JDataStore Explorer.

Opening an existing JDataStore file

To open an existing JDataStore file,

1 Select File | Open or click the Open JDataStore toolbar button. This opens the Java version of the standard File Open dialog box.

2 Choose the file to open and click Open.

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

Setting options for opening JDataStore files

You might want to open a JDataStore file in read-only mode to temporarily bypass transaction support or to attempt to open a file that has been damaged. Select View | Options to open the Options dialog box.

Figure 9.3 JDataStore options dialog box

Opening a JDataStore file that was not closed properly

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

Figure 9.4 JDataStore marked open dialog box

If this occurs, follow these steps:

1 Verify that there is no process that might still have the JDataStore file open (in particular, check the Task Manager for any rogue javaw processes):

   • If there is no process that has the JDataStore file open, click Yes to reopen the JDataStore file.
• If you click No, the JDataStore Explorer responds with an error dialog box that
  states that the file is still marked open by another process and the file won’t
  open.

2 Attempting to reopen the file might take several seconds. If the JDataStore file was
  not closed properly, another dialog box informs you of this condition. Click OK to
  attempt to recover the JDataStore file.

3 After you successfully open a JDataStore file that was left marked as open, a dialog
  box appears that gives you the opportunity to verify the contents of the JDataStore
  file. Click Yes to verify the JDataStore contents or No to skip the verification.

**Viewing JDataStore file information**

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

**Figure 9.5** JDataStore Explorer displaying JDataStore file information

![JDataStore Explorer](image)

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

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

Viewing stream contents

The JDataStore directory appears in the tree control on the left. The directory uses forward slashes ("/") in the stream names to synthesize a hierarchical structure. In addition, known stream types such as tables and text files appear under a corresponding node in the tree. You can use the View | Expand All and View | Collapse All menu items to help manage the directory tree.

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

Figure 9.6  JDataStore Explorer displaying table stored in JDataStore

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

File streams are handled by their file-name extension. JDataStore 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:
Renaming streams

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

1. Select the stream to rename/move in the directory tree. (You can’t rename deleted streams. You must undelete them first.)

2. Select Edit | Rename.

3. Type a new name in the Rename dialog box. You can’t use the name of another existing active stream.

4. Click OK.
Deleting streams

When you delete a stream, the blocks it used are marked as available. It’s possible to undelete a stream, although some of the blocks might have been reclaimed by other streams. See How JDataStore reuses deleted blocks 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, you can undelete it. You can’t undelete a stream if there is another active stream with the same name in the same directory, because you can’t have two streams with the same name.

Undeleting a stream doesn’t guarantee all the data in the stream will be recovered. See How JDataStore reuses deleted blocks 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 JDataStore streams

You can use the JDataStore Explorer to copy streams to another JDataStore file, much like the DataStoreConnection.copyStreams() method. While you can’t use this option to make copies of streams within the same JDataStore file, the JDataStore Explorer automatically creates a new JDataStore file for you.

To copy streams,
1. Select Tools | Copy JDataStore. This opens the Copy Streams dialog box:
2 Specify the various Copy Streams options, as summarized in the following table. (For more information, see copyStreams parameters)

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Destination JDataStore</td>
<td>The name of the JDataStore file to copy to.</td>
</tr>
<tr>
<td>Destination directory</td>
<td>Names of copies of streams have their Source directory replaced with this. The destination directory should be the same as Source directory if the name should not be changed.</td>
</tr>
<tr>
<td>Source directory</td>
<td>Stream name must begin with this to be pattern-matched. An empty string pattern-matches all streams.</td>
</tr>
<tr>
<td>Source Pattern</td>
<td>Stream name pattern to match, with standard * and ? wildcard characters.</td>
</tr>
</tbody>
</table>

3 Click OK.

**Verifying the JDataStore**

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

The JDataStore Explorer verifies the entire store and displays the results in the Verifier Log window. After you’ve closed the log window, you can view it again by selecting View | Verifier Log.

**Making the JDataStore transactional**

You can make a non-transactional JDataStore transactional with the JDataStore Explorer:
1 Select TxManager | Install. (If the JDataStore is already transactional, that menu option will be disabled.) This opens the TxManager Properties dialog box:

![TxManager Properties dialog box](image)

2 The dialog box contains default settings for the TxManager object:
   - You can change the default settings for the maximum number of open log files (2), maximum log file size (64MB), and checkpoint frequency (2MB).
   - By default, one set of log files are written in the same directory as the JDataStore 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 can enable Soft commit, which increases performance by not immediately forcing a disk write when a transaction is committed.
   - You can disable status logging for a slight performance improvement.

3 Click OK.

Modifying transaction settings

To modify transaction settings for a JDataStore,
1 Select TxManager | Modify. (If the JDataStore 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 JDataStore is transactional by examining the TxManager menu. If the Enabled menu option is checked, the JDataStore is transactional. If the menu option is disabled, the JDataStore is not transactional.

To make a transactional JDataStore non-transactional, uncheck TxManager | Enabled. This immediately removes transaction support.
Closing JDataStore files

When you are done with the JDataStore file, you should close it to ensure that all changes are written properly. You don't need to close open JDataStore files before exiting the JDataStore Explorer. The files close automatically.

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

JDataStore Explorer as a query console

When the JDataStore 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 can 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 JDataStore 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 JDataStore, edit the datasets, save your changes back to the database, and rerun queries to get the latest server data—all without writing code.

You can use this mechanism to simply import data from another database into a JDataStore. Because the query and connection information used to import the data is saved, you can easily reimport the data if you want.

Using JDataStore Explorer to manage queries

This discussion of ways to store and execute queries might 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 might or might not be saved in a JDataStore). In the former case, the JDataStore provides persistent storage for tables produced by querying a server. In the latter case, the JDataStore 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 JDataStore: as a place to store connection information and queries. These queries can be of either type just mentioned; they can get their data either from server tables or from datasets in a JDataStore. In either case, the result of running a query through the JDataStore Explorer is always another store table.

When you use the Explorer to manage queries, it's important to understand the objects involved and how they are related:

- In a JDataStore, 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 JDataStore.

This is the hierarchy:
- Many connections per JDataStore
- 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.

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

**JDataStore Explorer limitations**

Remember that although the JDataStore Explorer can execute queries and save changes to the dataset back to server tables, it has no application-specific logic. The data is presented in just 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, you won’t be prevented from entering values that violate minimum or maximum constraints, and you might be able to leave required fields empty or violate referential integrity constraints. (Constraints defined in the server are 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 it’s 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 JDataStore Explorer to manage queries or to import a table from another database, you must have an open JDataStore. Then, to define a query, select Tools | Import | Tables. The Import Tables dialog box opens:
The first time you define a query, there won't be any connections to associate it with. The New button next to the Connection field lets you define a new connection through the New JDBC Connection dialog box.

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

Once you have a connection to a database, you'll see a list of available tables. After selecting the desired tables, you can click Finish to simply import those tables. If you want more control over what is imported, click Next to go to the next page.
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 JDataStore, which defaults to the original name. You can change this. It can include a path as in Data/Tutorial/Employee. The Explorer’s tree pane displays 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 you can edit to change the fields, conditions, and grouping.
- Check boxes to control whether indexes should be created, and whether to enable refresh and save for this query. Refresh and save settings for the query are saved to the SYS/DataStore Queries/Queries data set. If you have read-only tables, you should not enable save. If you know the data in a table is not going to change, you should not enable refresh.

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 navigation toolbar or press Ctrl+Del.
- Insert a new definition. To do so, press “+” on the navigation toolbar or press Ctrl+Ins.
Fetching and editing data

Right after saving a new query, the JDataStore Explorer attempts to execute that query to fetch its data. You’ll 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 discards 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 JDataStore

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 JDataStore Explorer, select Tools | Refresh JDataStore or Tools | Save JDataStore 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.
In addition to importing tables from other databases, the JDataStore 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 you export 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. To give you an idea of what one looks like, here is one for a simple three-column dataset:

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

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

Once you have a `SCHEMA` file to accompany the text file, follow these steps to import the text file as a table,
1 Select Tools | Import | Text Into Table. This opens the Import Delimited Text File dialog box.

2 Supply the input text file and the store name of the dataset to be created. Because 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.

**Creating tables**

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

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

2 Choose File | Open in the JDataStore Explorer and select the database for which you want to create a new table.

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

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

5 Select a locale if you are internationalizing the table. Otherwise, leave the value <null>.
6 Click the Insert New Row button on the Navigation bar to create a new row.
7 Click in the Column Name field and type the name of the new column.
8 Click in each of the column property fields on that row you want to define, and select or enter a value. For each column, you must specify at least a column name and data type. You may also specify other properties as needed. (See the Column class for a description of the meaning of the Column properties.)
9 Continue creating the rest of the columns in this manner, rearranging their order in the table as desired. Use the Navigation bar buttons to add or insert additional rows, move to different rows, and rearrange the rows you have added.
10 Click OK when you are finished creating and defining all the columns.

It is also possible to modify an existing table's structure in the JDataStore Explorer. Select a table in the tree on the left, and click the Structure tab. The UI for the Structure tab is the same as the Create Table dialog box.

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

### Creating indexes

You can use the JDataStore Explorer to visually create an index for a JDataStore table. To create an index:
1 Open the JDataStore Explorer. (If you are in JBuilder, choose Tools | DataStore Explorer.)
2 Choose File | Open in the JDataStore Explorer and select a database.
3 Choose Tools | Create Index in the JDataStore Explorer to open the Create Index dialog box.
4 Choose the name of the table for which you want to create an index in the Table Name drop-down list.

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

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

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

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

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

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

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

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

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

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

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

Executing SQL

You can execute arbitrary SQL statements with the currently selected JDataStore file as the database. If the JDataStore file is not transactional, you can only do read-only queries. If a non-transactional JDataStore is not already open in read-only mode, it will automatically be closed and reopened in read-only mode. To execute SQL, select Tools | SQL or click the SQL toolbar button, which opens the SQL dialog box:
You can type in SQL statements directly or execute files containing SQL. Statements that you type are recorded and you can scroll through them with the Previous and Next buttons to modify and re-execute recorded statements. Result sets returned by SQL statements are displayed in the lower half of the dialog box.

**JDataStore file operations**

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

**Packing the JDataStore file**

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

To pack the JDataStore, select Tools | Pack JDataStore.

**Upgrading the JDataStore file**

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

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

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

JDataStore security tasks

The following security-related tasks can be performed using the JDataStore Explorer:

Administrering users

To administer users for a JDataStore, select Tools | Administer Users. If an administrator has not previously been defined for the JDataStore, when you select this menu option, a dialog will prompt for the administrator's user name and password. When you choose an administrator user name and password, the administrator user will automatically be added and will be assigned all rights for the JDataStore by default.

If you are logged in as a user with administrator's rights, the Administer Users dialog appears. If a user without administrator's rights tries to open this dialog, they will be prompted for an administrator's user name and password. The Administer Users dialog allows the administrator to add, edit, and remove users, and assign rights to each user. Here is the Administer Users dialog:

![Administer Users dialog]

The existing users are displayed in a table. Checkboxes in the table columns indicate which rights are assigned to a user. For an explanation of the various rights, see Authorization.
Adding a user
To add a user, click the Add button on the Administering users dialog. You will see the Add User dialog, which looks like this:

![Add User dialog](image)

Enter a name for the new user. Type in the user's password, and then type it again to confirm the password. The user will be able to change their own password when they log in.

Next, select which rights the user will have. For an explanation of the various rights, see Authorization. Click OK when you are done assigning rights to the user.

Editing a user
To edit a user, select the row for the user in the table, then click the Edit button on the Administering users dialog. You can now edit the rights for the selected user. The Edit User dialog appears very similar to the Adding a user dialog, except you cannot edit the user name or password.

Removing a user
To remove a user, select the row for the user in the table, then click the Remove button on the Administering users dialog. You will be prompted to confirm that you want to remove the user. You cannot remove the only administrator.

Changing a password
A user must be currently logged in to change their password. To change the password for the current user, select Tools | Change Password. A Change Password dialog is presented. You must enter the old password, then enter the new password twice for confirmation. Click OK.

Encrypting a JDataStore
To encrypt a JDataStore file, select Encrypt JDataStore from the Tools menu. The JDataStore Explorer will try to encrypt the JDataStore immediately. A message will then indicate success or failure. If the JDataStore is encrypted successfully, a backup
copy of the original file will be made, and the results message will indicated the name of this backup.
This chapter discusses issues that will help improve the performance, reliability, and size of JDataStore applications. Unless otherwise specified, DataStoreConnection refers to either a DataStoreConnection or DataStore object used to open a connection to a JDataStore file.

Loading databases quickly

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

- Use prepared statements whenever possible. If the number of parameters changes from one insert to the next, call PreparedStatement.clearParameters() before setting PreparedStatement parameters.

- Use the DataExpress TableDataSet.addRow() method, which is slightly faster than the JDBC prepared statements. You must set the TableDataSet store to a DataStoreConnection and StoreName property to the name of your table in the JDataStore database.

- Use the DataExpress TextDataFile method to import text files. It has a fast parser and knows how to load data quickly. You must set the TableDataSet store to a DataStoreConnection and set the StoreName property to the name of your table in the JDataStore database.

- When loading a new database, first create the database as non-transactional. Load the database while it is non-transactional using the DataExpress TableDataSet.addRow() or TextDataFile component. After loading is complete, make the database transactional using the DataStore.TxManager property. This technique should make the load operation perform two to three times faster.
General usage recommendations

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

Closing the JDataStore

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

- To guarantee proper closure, don't call System.exit() until you have called DataStore.shutdown().

Closing a JDataStore 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 might not have the opportunity to save the last set of changes. There is a small chance that a non-transactional JDataStore may get corrupted.

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

If a JDataStore file isn't closed properly, it takes several seconds to reset/recover the JDataStore file. Proper closure means that the JDataStore reopens quickly every time.

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

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

Optimizing the JDataStore disk cache

The default maximum cache size for JDataStore is 512 cache blocks. The default block size is 4096 bytes. Therefore, the cache memory reaches its maximum capacity out at approximately about 512*4096 (2MB). Note that this memory is allocated as needed. In some rare situations when all blocks are in use, the cache may grow beyond 512 cache blocks. The minimum cache size can be specified with the DataStore.MinCacheSize property.

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

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

- Modern OS caches are typically high performance. In many cases, increasing the JDataStore cache size does not significantly improve performance and simply uses more memory.
Depending on the Java JVM you are using, allocating large amounts of memory might slow the performance of JVM garbage collection operations.

There is only one JDataStore disk cache for all JDataStore databases open in the same process. When all JDataStore databases are shutdown, the memory for this global disk cache is released.

For handheld devices with small amounts of memory, set the `DataStore.MinCacheSize` property to a smaller number, such as 96.

**Optimizing file locations**

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

- The JDataStore database file itself (file extension is `.jds`) as specified by the `DataStore.FileName` property.
- JDataStore transactional log files. The names of log files that end with an extension of `LOGAnnnnnnnnn`, where 'n' is a numeric digit as specified by the `TxManager.ALogDir` property.
- Temporary files used for large sort operations as specified by the `DataStore.TempDirName` property.
- Temporary `.jds` file used for SQL query results as specified by the `DataStore.TempDirName` property.

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

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

- It is especially important to place the log files on a separate disk drive. Note that the log file is generally written to in sequential order and its contents must be forced to disk in order to complete commit operations. As such, it is advantageous to have a disk drive that can complete write operations quickly.

- On Windows NT, it has been observed that performance can be improved by placing JDataStore log files in a separate directory, and that storing numerous files other than the log files in the log file directory can slow down the performance of commit operations. (Note that this performance tip may also apply to platforms other than Windows NT.)

- Remember to defragment your disk drive file systems periodically on a regular basis. This practice is especially important for the disk drive that stores the log files because JDataStore performs a great deal of sequential read/write operations to this file.
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 JDataStores. 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` can be changed when the connection is open. For example, if you are using a `DataStoreConnection`, you can access the value through the `dataStore` property:

```java
DataStoreConnection store = new DataStoreConnection();
// ...
store.getDataStore().setSaveMode(2);
```

Note that this changes the behavior for all `DataStoreConnection` objects that access that particular JDataStore file.

Tuning memory

You can tune the use of memory in a number of ways. Be aware that asking for too much memory can be as bad as having too little.

- The Java heap tends to resist growing beyond its initial size, forcing frequent garbage collection with an ever-smaller amount of free heap. Use the `-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

Here are some tips that can help performance:

- Setting the `DataStore.tempDirName` property, used by the query engine, to a directory on another (fast) disk drive might help.
- Try setting the `TxManager.checkFrequency` higher. Higher values result in slower recovery, but why be pessimistic?
For simple operations, DataExpress might be faster than JDBC/SQL. JDBC/SQL is probably faster for more complex queries.

Try to write your applications with efficient multithreading. Attempt 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 JDataStore applications.

- **DBDisposeMonitor** (which automatically disposes of data-aware component resources when a container is closed) has a `closeDataStores` property. When `true` (the default), it automatically closes any JDataStores 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 JDataStore, when you close the JFrame, DBDisposeMonitor automatically closes the JDataStore for you. This component is particularly handy when building simple applications to experiment with JDataStore.

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

Using data modules

When using a JDataStore with a StorageDataSet, you should almost always group them all inside data modules. Make any references to these StorageDataSets through DataModule accessor methods such as `businessModule.getCustomer()`. You should do this because much of the functionality surfaced through StorageDataSets is driven by property and event settings. Although most of the important structural StorageDataSet properties are persisted in the JDataStore itself, the classes that implement the event listener interfaces aren't. By instantiating the StorageDataSet with all event listener settings, constraints, calculated fields, and filters implemented with events, they are properly maintained at run time and design time.

Optimizing transactional applications

The increased reliability and flexibility you gain from using transactional JDataStores 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 JDataStore. 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 opens, and it refreshes every time its commit() method is called.

Another benefit of read-only transactions is that they aren't blocked by writers (or other readers). Both reading and writing usually require a stream lock. But because a read-only transaction uses a snapshot, it doesn't need a lock. Therefore it won't be blocked by a writer that has a lock on a stream that it's 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 refreshes, 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 forces disk writes for the purposes of crash recovery, but it doesn't guarantee transaction commits. This improves performance, but makes 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—aren't guaranteed to be written. This isn't the case when soft commit mode is disabled (the default), when committed changes are written immediately.

Transaction log files

The TxManager's ALogDir and BLogDir properties control the location of the transaction log files.

- Don't place the files on slow disk drives, including network drives. (Putting the log files on a floppy disk is a very bad idea.)
You can 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 might reduce some of the performance penalty for writing two separate log files (both drives can write simultaneously), and it 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.

Tuning JDataStore concurrency control performance

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

- Choose the weakest isolation level that your application can function properly with. Lower isolations tend to acquire fewer and weaker locks.
- Set the lockTableTables property discussed above for tables that are infrequently updated. There is less overhead for table locks.
- Set java.sql.Connection.autoCommit() to false to group multiple operations into a single transaction. The java.sql.Connection.commit()/rollback() property can be used for terminating transactions.
- Commit transactions as soon as possible. Most locks are not released until a transaction is committed or rolled back.
- Reuse java.sql.Statements whenever possible and use java.sql.PreparedStatements when possible.
- Close all Statements, PreparedStatements, ResultSets, and Connections when they are no longer needed. Note that single directional ResultSets automatically close when the last row is read.
- Use read only transactions for long running reports or on line backup operations (use the com.borland.datastore.DataStoreConnection.copyStreams() method for online backups). Read only transactions provide a transactionally consistent (serializable), read only view of the tables they access. They acquire no locks, so lock timeout and deadlock are not possible. This property can be set using the java.sql.Connection.readOnly() method for JDBC connections or the com.borland.datastore.DataStoreConnection.setReadOnlyTx() method for DataExpress components.
- There is some overhead for maintaining a read only view. Consequently, multiple transactions can share the same read only view. The readOnlyTxDelay property specifies how old the read only view can be when a read only transaction is started. Commit a read only connection’s transaction to refresh its view of the database. Note that a read only transaction uses the transactional log files to
maintain its views. Consequently, read only connections should be closed as soon as they are no longer needed.

Using Multi-threaded operations

Write transaction throughput can increase as more threads are used to perform operations because each thread can share in the overhead of commit operations via the “group commit” support provided by JDataStore.

Pruning deployed resources

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

- If JDataStore is used without transaction support, exclude these classes:
  - com.borland.datastore.Tx*.class

- If JDataStore is used without using the JDBC driver, exclude these classes:
  - 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 these classes:
  - com.borland.dx.memorystore.*

- If TableDataSet is used, but not QueryDataSet, QueryProvider, StoredProcedureDataSet or StoredProcedureProvider, exclude these classes:
  - com.borland.dx.sql.*

- If DataExpress isn't using any visual components from the JBCL or dbSwing libraries, exclude these classes:
  - com.borland.dx.text.*

- If com.borland.dx.dataset.TextDataFile isn't used, exclude these classes:
  - com.borland.jb.io.*
  - com.borland.dx.dataset.TextDataFile.class
  - com.borland.dx.dataset.SchemaFile.class
AutoIncrement Columns

This version of JDataStore supports a feature called AutoIncrement columns. Specifically, columns of type int and long can now be specified as having AutoIncrement values.

These properties apply to all AutoIncrement column values:

- they are always unique
- they can never be null
- values from deleted rows can never be reused

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

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

**Note** Each table can have only one AutoIncrement column.

Using an AutoIncrement column saves the space of one integer column and one secondary index in your table if you use it as a replacement for your primary key. The JDataStore query optimizer will optimize queries that reference an AutoIncrement column in a where clause.

### AutoIncrement Columns Using DataExpress

To create a table with an AutoIncrement column using DataExpress, set the Column.AutoIncrement property to true before opening a table. If you are modifying an existing table, you will need to call the StorageDataSet.restructure() method. For more information, see JDataStore as an embedded database.

### AutoIncrement Columns Using SQL

To create or modify a table to have an AutoIncrement column using SQL, see SQL reference.
Appendix

Specifications

The following specifications apply to the 6.0 version of the JDataStore file format.

**JDataStore file capacity**

- Minimum block size: 1 KB
- Maximum block size: 32 KB
- Default block size: 4 KB
- Maximum JDataStore 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

**JDataStore 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
- SYS/Users
Troubleshooting

Here are some tips for when problems occur.

Debugging JDataStore applications

Set the `saveMode` property to 2 when debugging a non-transactional JDataStore. The debugger stops all threads when you are single-stepping through code or when breakpoints are hit. If the `saveMode` property isn't set to 2, this keeps the JDataStore daemon thread from saving modified cache data. For more information, see Controlling how often cache blocks are written to disk.

Verifying JDataStore contents

If you suspect that cache contents were not properly saved on a non-transactional JDataStore, you can verify the integrity of the file with the JDataStore Explorer. See Verifying the JDataStore 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 JDataStore. For more information, see the DataExpress Component Library Reference.

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

If problems are encountered, use the JDataStore Explorer (see Copying JDataStore streams) 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 JDataStore 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 manifest themselves in the following ways:

- Locate and query operations might not find records that they should.
- Viewing a table in secondary index order (by setting the `StorageDataSet.sort` property) might not be ordered properly.

Currently, the only way to correct this is to drop the secondary indexes and rebuild them with the current JDK. The `StorageDataSet.restructure` method also drops all the secondary indexes.

Saving log file

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 the log file to another location before it is deleted, or you can `cancel()` the event to prevent the deletion of the log file.
How to access SQL using JDBC:

JDataStore comes with a JDBC driver. Use the following methods to execute a SQL statement:

From Statement.java:
int executeUpdate(String query);
ResultSet executeQuery(String query);

From Connection.java:
Statement createStatement();
PreparedStatement prepare(String query);
CallableStatement prepare(String query);

From PreparedStatement.java and CallableStatement.java:
int executeUpdate();
ResultSet executeQuery();

Each query string must contain exactly one SQL statement. Use executeQuery for statements that return a resultSet. For example:
SELECT * FROM EMPLOYEE.

Use executeUpdate for statements that do not:
CREATE TABLE MYTABLE (COLUMN1 INT, LAST_NAME VARCHAR(20))
INSERT INTO MYTABLE VALUES (1, 'Overbek')
UPDATE MYTABLE SET LAST_NAME='Overbeck' WHERE COLUMN1=1

Note that executeUpdate throws an exception if the statement executed actually produces a resultSet.

If a statement contains output parameters, a CallableStatement must be used.
If a statement contains input parameters, a PreparedStatement or a CallableStatement must be used.
## Data types

In SQL you can specify data types by using the JDataStore names or by using synonyms, which are more portable to other SQL engines. The possible types and their synonyms are listed in the following table.

<table>
<thead>
<tr>
<th>JDataStore datatype</th>
<th>Description</th>
<th>Bytes</th>
<th>Synonyms</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMALLINT</td>
<td>Exact numeric with the range: -32768..32767</td>
<td>1-3</td>
<td>SHORT</td>
</tr>
<tr>
<td>INT</td>
<td>Exact numeric with the range: -2147483648..2147483647</td>
<td>1-5</td>
<td>INTEGER</td>
</tr>
<tr>
<td>BIGINT</td>
<td>Exact numeric with the range: -9223372036854775808..9223372036854775807</td>
<td>1-9</td>
<td>LONG</td>
</tr>
<tr>
<td>DECIMAL(p,d)</td>
<td>Exact numeric with a precision of p digits and d decimals. The precision is limited to 72 digits. If omitted the default value of p is 72 and d is 0.</td>
<td>1-32</td>
<td>BIGDECIMAL</td>
</tr>
<tr>
<td>REAL</td>
<td>Approximate numeric with the range: 1.4E-45..3.4E38 and a precision of 23 bits.</td>
<td>1-5</td>
<td></td>
</tr>
<tr>
<td>DOUBLE</td>
<td>Approximate numeric with the range: 4.9E-324...1.8E308 and a precision of 52 bits.</td>
<td>1-9</td>
<td>DOUBLE_PRECISION</td>
</tr>
<tr>
<td>FLOAT(p)</td>
<td>Approximate numeric with a precision of at least p bits where p&lt;=52. If omitted the default value of p is 52.</td>
<td>1-9</td>
<td></td>
</tr>
<tr>
<td>VARCHAR(p,m)</td>
<td>Variable length string with a maximal length of p characters and a max inline of m bytes. When the string contains more than m bytes, the rest of the string is stored as a BLOB. If omitted the default value of p is unlimited and m is 64.</td>
<td>1-m</td>
<td>STRING</td>
</tr>
</tbody>
</table>
Examples

- A string with a maximum size of 30 bytes and any string over 10 bytes is stored in a separate stream for large objects:
  `VARCHAR(30,10)`

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

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

- 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:
  \texttt{OBJECT}

- Only Java strings using Java serialization:
  \texttt{OBJECT('java.lang.String')}

\section*{Literals}

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

<table>
<thead>
<tr>
<th>JDataStore Datatype</th>
<th>Literal Examples</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMALLINT</td>
<td>8</td>
<td>Exact numerics may contain a decimal point.</td>
</tr>
<tr>
<td>INT</td>
<td>2.</td>
<td></td>
</tr>
<tr>
<td>BIGINT</td>
<td>15.7</td>
<td></td>
</tr>
<tr>
<td>DECIMAL(p,d)</td>
<td>.9233</td>
<td></td>
</tr>
<tr>
<td>REAL</td>
<td>8E0</td>
<td>Approximate numerics is a number followed by the letter E, followed with an optionally signed integer.</td>
</tr>
<tr>
<td>DOUBLE</td>
<td>4E3</td>
<td></td>
</tr>
<tr>
<td>FLOAT(p)</td>
<td>0.3E2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>6.2E-72</td>
<td></td>
</tr>
<tr>
<td>VARCHAR(p,m)</td>
<td>'Hello'</td>
<td>Strings are enclosed in single quotes. The single quote character is represented by two consecutive single quotes.</td>
</tr>
<tr>
<td></td>
<td>'don''t do that'</td>
<td></td>
</tr>
<tr>
<td>VARBINARY(p,m)</td>
<td>B'1011001'</td>
<td>A binary or hexadecimal sequence enclosed in single quotes and preceded by the letter B or X</td>
</tr>
<tr>
<td></td>
<td>X'F08A'</td>
<td></td>
</tr>
<tr>
<td></td>
<td>X'f777'</td>
<td></td>
</tr>
<tr>
<td>OBJECT(t,m)</td>
<td>There are no object literals in JDataStore SQL.</td>
<td></td>
</tr>
<tr>
<td>BOOLEAN</td>
<td>TRUE</td>
<td></td>
</tr>
<tr>
<td></td>
<td>FALSE</td>
<td></td>
</tr>
<tr>
<td>DATE</td>
<td>DATE '2002-06-17'</td>
<td>Displays local time of origin; format is DATE 'yyyy-mm-dd'</td>
</tr>
<tr>
<td>TIME</td>
<td>TIME '15:46:55'</td>
<td>Displays local time of origin; format is TIME 'hh:mm:ss' a 24 hour format</td>
</tr>
<tr>
<td>TIMESTAMP</td>
<td>TIMESTAMP '2001-12-31 13:15:45'</td>
<td>Format is TIMESTAMP 'yyyy-mm-dd hh:mm:ss'</td>
</tr>
</tbody>
</table>
JDBC Escape sequences

JDataStore supports JDBC escape sequences for:

- specifying date and time literals, the escape character for a LIKE clause
- OUTER JOINs
- inserting the results of string or date and time functions into your SQL statement
- calling stored procedures

JDBC escapes must always be enclosed in braces {}. They are used to extend the functionality of SQL.

Examples

Date and time literals:

\{ T 'hh-mm-ss' \}

Specifies a time, which must be entered in the sequence: hours, followed by minutes, followed by seconds.

\{ D 'mm-dd-yy' \}

Specifies a date, which must be entered in the format indicated; month, followed by day, followed by year.

\{ TS 'mm-dd-yy : hh-mm-ss' \}

Specifies a timestamp, which must be entered in the format indicated; month, day, year, colon, hour, minute, second.

OUTER JOINs:

\{ OJ <join table expression> \}

An outer join is performed on the specified table expression.

Escape character for LIKE:

\{ ESCAPE <char> \}

The specified character becomes the escape character in the preceding LIKE clause.

Escape functions

Functions are written in the format:

\{FN <escape function expression> \}

FN indicates that the function following it should be performed.
String functions

CONCAT(string1, string2) concatenates two strings
LCASE(string) returns the string in lowercase
LENGTH(string) returns the length of the string
LOCATE(string1, string2 [, pos] ) locates string1 in string2, starting at position pos in string2
LTRIM(string) trims leading spaces from string
RTRIM(string) trims trailing spaces from string
SUBSTRING(string, start, length) returns a substring of length from the specified string, starting at position start
UCASE(string) returns the string in uppercase

Numeric functions

SQRT(number) returns the square root of a number.
ABS(number) returns the absolute value of a number.

Date and time functions

CURDATE() returns the current date.
CURTIME() returns the current time.
DAYOFMONTH(date) extracts the day of the month from the specified date.
HOUR(time) extracts the hour from the specified time.
MINUTE(time) extracts the minute from the specified time.
NOW() returns the current timestamp.
SECOND(time) extracts the second from the specified time.
YEAR(date) extracts the year from the specified date.

System functions

USER( ) returns the username of the current connection.
CONVERT() converts a scalar expression to one of the following SQL types:

<table>
<thead>
<tr>
<th>Type</th>
<th>Type</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIGINT</td>
<td>BINARY</td>
<td>BIT</td>
</tr>
<tr>
<td>CHAR</td>
<td>DATE</td>
<td>DECIMAL</td>
</tr>
<tr>
<td>DOUBLE</td>
<td>FLOAT</td>
<td>INTEGER</td>
</tr>
</tbody>
</table>
Examples

Time and date literals

```sql
INSERT INTO tablename VALUES(\'10-2-3\', \'2:55:11\')
SELECT \'10:24\' FROM tablename
SELECT \'2000-02-01\' FROM tablename
SELECT \'2000-02-01 10:24:32\' FROM tablename
```

Joins

```sql
SELECT * FROM (a LEFT JOIN b USING(id))
```

Specify escape character for LIKE

```sql
SELECT * FROM a WHERE name LIKE \'%*\%' {ESCAPE '*'}
```

String functions

```sql
SELECT {FN LCASE('Hello')} FROM tablename
SELECT {FN UCASE('Hello')} FROM tablename
SELECT {FN LOCATE('xx', '1xx2')} FROM tablename
SELECT {FN LTRIM('Hello')} FROM tablename
SELECT {FN RTRIM('Hello')} FROM tablename
SELECT {FN SUBSTRING('Hello', 3, 2)} FROM tablename
SELECT {FN CONCAT('Hello ', 'there.')} FROM tablename
```

Time and date functions

```sql
SELECT {FN NOW()} FROM tablename
SELECT {FN CURDATE()} FROM tablename
SELECT {FN CURTIME()} FROM tablename
SELECT {FN DAYOFMONTH(datecol)} FROM tablename
SELECT {FN YEAR(datacol)} FROM tablename
SELECT {FN MONTH(datecol)} FROM tablename
SELECT {FN HOUR(timecol)} FROM tablename
SELECT {FN MINUTE(timecol)} FROM tablename
SELECT {FN SECOND(timecol)} FROM tablename
```

Keywords

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

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>LONGVARBINARY</td>
<td>LONGVARCHAR</td>
<td>REAL</td>
</tr>
<tr>
<td>SMALLINT</td>
<td>TIME</td>
<td>TINYINT</td>
</tr>
<tr>
<td>VARBINARY</td>
<td>VARCHAR</td>
<td></td>
</tr>
</tbody>
</table>
Note that not all SQL-92 keywords are treated as a keyword by the JDataStore SQL engine. For maximum portability, don't use identifiers that are treated as keywords in any SQL dialect.

<table>
<thead>
<tr>
<th>ABSOLUTE</th>
<th>ACTION</th>
<th>ADD</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALL</td>
<td>ALTER</td>
<td>AND</td>
</tr>
<tr>
<td>ANY</td>
<td>AS</td>
<td>ASC</td>
</tr>
<tr>
<td>AUTOCOMMIT</td>
<td>AUTOINCREMENT</td>
<td>AVG</td>
</tr>
<tr>
<td>BETWEEN</td>
<td>BIT</td>
<td>BOTH</td>
</tr>
<tr>
<td>BY</td>
<td>CALL</td>
<td>CASCADE</td>
</tr>
<tr>
<td>CASE</td>
<td>CAST</td>
<td>CHAR</td>
</tr>
<tr>
<td>CHAR_LENGTH</td>
<td>CHARACTER</td>
<td>CHARACTER_LENGTH</td>
</tr>
<tr>
<td>CHECK</td>
<td>COLUMN</td>
<td>COMMIT</td>
</tr>
<tr>
<td>CONSTRAINT</td>
<td>COUNT</td>
<td>CREATE</td>
</tr>
<tr>
<td>CROSS</td>
<td>CURRENT_DATE</td>
<td>CURRENT_TIME</td>
</tr>
<tr>
<td>CURRENT_TIMESTAMP</td>
<td>DATE</td>
<td>DAY</td>
</tr>
<tr>
<td>DEC</td>
<td>DECIMAL</td>
<td>DEFAULT</td>
</tr>
<tr>
<td>DELETE</td>
<td>DESC</td>
<td>DISTINCT</td>
</tr>
<tr>
<td>DOUBLE</td>
<td>DROP</td>
<td>ELSE</td>
</tr>
<tr>
<td>END</td>
<td>ESCAPE</td>
<td>EXCEPT</td>
</tr>
<tr>
<td>EXISTS</td>
<td>EXTRACT</td>
<td>FALSE</td>
</tr>
<tr>
<td>FLOAT</td>
<td>FOR</td>
<td>FOREIGN</td>
</tr>
<tr>
<td>FROM</td>
<td>FULL</td>
<td>GROUP</td>
</tr>
<tr>
<td>HAVING</td>
<td>HOUR</td>
<td>IN</td>
</tr>
<tr>
<td>INDEX</td>
<td>INNER</td>
<td>INSERT</td>
</tr>
<tr>
<td>INT</td>
<td>INTEGER</td>
<td>INTERSECT</td>
</tr>
<tr>
<td>INTO</td>
<td>IS</td>
<td>JOIN</td>
</tr>
<tr>
<td>KEY</td>
<td>LEADING</td>
<td>LEFT</td>
</tr>
<tr>
<td>LIKE</td>
<td>LOCK</td>
<td>LOWER</td>
</tr>
<tr>
<td>MAX</td>
<td>MIN</td>
<td>MINUTE</td>
</tr>
<tr>
<td>MONTH</td>
<td>NATURAL</td>
<td>NO</td>
</tr>
<tr>
<td>NOT</td>
<td>NOWAIT</td>
<td>NULL</td>
</tr>
<tr>
<td>NUMERIC</td>
<td>OFF</td>
<td>ON</td>
</tr>
<tr>
<td>OR</td>
<td>ORDER</td>
<td>OUTER</td>
</tr>
<tr>
<td>POSITION</td>
<td>PRECISION</td>
<td>PRIMARY</td>
</tr>
</tbody>
</table>
Identifiers

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

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

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

Examples

Valid identifiers:

- `customer` // treated as CUSTOMER
- `Help_me` // treated as HELP_ME
- `'Hansen'` // treated as Hansen
- `' '` // treated as a single space

Invalid identifiers:

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

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

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

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

Syntax

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

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

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

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

<concat> ::=   | |

<function> ::=<substring function>
| <position function>
| <trim function>
| <extract function>
| <lower function>
| <upper function>
| <char length function>
| <absolute function>
| <square root function>

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

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

<user defined function reference> ::= <method name> ( [ <expression commalist> ] )

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

<correlation name> ::= <SQL identifier>

Examples
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

Get data using a scalar subquery:

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

Note that it is an error if the subquery returns more than 1 row.

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

BETWEEN

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

expr BETWEEN leftExpr AND rightExpr
is equivalent to the expression:

\[ \text{leftExpr} \leq \text{expr} \text{ AND expr} \leq \text{rightExpr} \]

**Syntax**

\[
\text{<between expression>} ::= \text{<scalar expression>} [\text{NOT}] \text{ BETWEEN } \text{<scalar expression>} \text{ AND } \text{<scalar expression>}
\]

**Example**

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

\[
\text{SELECT } * \text{ from Orders WHERE Amount BETWEEN 3 AND 7}
\]

**IS**

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

**Syntax**

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

**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 can include the special wildcard characters _ and % where:

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

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**

\[
\text{<like expression>} ::= \text{<search item>} [\text{NOT}] \text{ LIKE } \text{<pattern>} [\text{ ESCAPE } \text{<escape char> }]
\]

\[
\text{<search item>} ::= \text{<scalar expression>}
\]
**Example**

Item LIKE `%'shoe%'` evaluates to **TRUE** if Item contains the string “shoe” anywhere inside it.

Item LIKE `S_` evaluates to **TRUE** if Item is exactly three characters long, starting with the letter “S”.

Item Like `%*%` ESCAPE `'*'` evaluates to **TRUE** if Item ends with the percent character. The * is defined to escape the two special characters. If it precedes a special character, it is treated as a normal character in the pattern.

**IN**

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

**Syntax**

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

**Example**

The following expression returns all records where the `name` column matches either “leo” or “aquarius”.

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

The **IN** clause also has a variant where a subquery is used instead of a list of expressions.

**Syntax**

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

**Example**

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

**Quantified comparisons**

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

**Syntax**

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

**Example**

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

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

Syntax

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

Example

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

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

Functions

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

**ABSOLUTE**

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

Syntax

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

Example

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

**CHAR_LENGTH** and **CHARACTER_LENGTH**

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

Syntax

<char length function> ::= CHAR_LENGTH | CHARACTER_LENGTH ( <scalar expression> )
CURRENT_DATE, CURRENT_TIME, and CURRENT_TIMESTAMP

These SQL functions yield the current date and/or time. If the functions are placed more than once in a statement, they 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 can be a DATE, TIME or TIMESTAMP value.

Syntax

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

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

Examples

EXTRACT(MONTH FROM DATE '1999-05-17') yields 5.
EXTRACT(HOUR FROM TIME '18:00:00') yields 18.
EXTRACT(HOUR FROM DATE '1999-05-17') yields an exception.

LOWER and UPPER

The SQL LOWER and UPPER functions yield the given string, converted to the requested case: either 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 evaluate to NULL, the result is NULL.
Syntax
<position function> ::= POSITION ( <string> IN <another> )

Example
POSITION('BCD' IN 'ABCDEFG') yields 2.
POSITION('' IN 'ABCDEFG') yields 1.
POSITION('TAG' IN 'ABCDEFG') yields 0.

SQRT

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

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

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

SUBSTRING

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

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

Examples
SUBSTRING('ABCDEFG' FROM 2 FOR 3) yields 'BCD'
SUBSTRING('ABCDEFG' FROM 4) yields 'DEFG'
SUBSTRING('ABCDEFG' FROM 10) yields ''
SUBSTRING('ABCDEFG' FROM -6 FOR 3) yields 'ABC'
SUBSTRING('ABCDEFG' 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 are removed.
• If the `<trim spec>` is omitted, BOTH is assumed.
• If both `<padding>` and `<trim spec>` are omitted, the FROM keyword must be omitted.

Syntax

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

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

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

Example

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

CAST

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

Syntax

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

Example

```
SELECT * FROM employee WHERE CAST ( salary AS long ) = 86293 yields row where salary equals 86,292.94
```

About statement syntax

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

• Lists
• Table expressions
• Select expressions
• Join expressions

Lists in syntax notation

In the following section you will find element names ending with the words “list” or “commalist” that are not further defined. For example:
These definitions are to be read as a lists with at least one element, comma separated in the case of a commalist:

\[
\text{<select item commalist>} ::= \text{<select item>} \mid , \text{<select item>}^* \\
\text{<column constraint list>} ::= \\
\text{<column constraint>} \mid <\text{column constraint}>^* \\
\]

**Table expressions**

A table expression is a term for an expression that evaluates to an unnamed table. Of the infix operators **INTERSECT**, **UNION**, and **EXCEPT**: **INTERSECT** binds the strongest and **UNION** and **EXCEPT** are equal.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UNION ALL</td>
<td>Creates the union of two tables including all duplicates.</td>
</tr>
<tr>
<td>UNION</td>
<td>Creates the union of two tables. If a row occurs multiple times in both tables, the result has this row exactly twice. Other rows in the result have no duplicates.</td>
</tr>
<tr>
<td>INTERSECTION ALL</td>
<td>Creates the intersection of two tables including all duplicates.</td>
</tr>
<tr>
<td>INTERSECTION</td>
<td>Creates the intersection of two tables. If a row has duplicates in both tables, the result has this row exactly twice. Other rows in the result has no duplicates.</td>
</tr>
<tr>
<td>EXCEPT ALL</td>
<td>Creates a table that has all rows that occur only in the first table. If a row occurs ( m ) times in the first table and ( n ) times in the second, the result holds that row the larger of zero and ( m-n ) times.</td>
</tr>
<tr>
<td>EXCEPT</td>
<td>Creates a table that has all rows that occur only in the first table. If a row occurs ( m ) times in the first table and ( n ) times in the second, the result holds the row exactly twice if ( m &gt; 1 ) and ( n = 0 ). Other rows in the result has no duplicates.</td>
</tr>
</tbody>
</table>

**Syntax**

\[
\text{<table expression>} ::= \\
\text{<table expression>} \text{UNION [ ALL ] <table expression>} \\
| \text{<table expression>} \text{EXCEPT [ ALL ] <table expression>} \\
| \text{<table expression>} \text{INTERSECT [ ALL ] <table expression>} \\
| \text{<join expression>} \\
| \text{<select expression>} \\
| \text{<table expression> } \\
\]

**Examples**

\[
\text{SELECT * FROM T1 UNION SELECT * FROM T2 UNION SELECT * FROM T3}
\]

is executed as:
(SELECT * FROM T1 UNION SELECT * FROM T2) UNION SELECT * FROM T3
SELECT * FROM T1 UNION SELECT * FROM T2 INTERSECT SELECT * FROM T3

is executed as:

SELECT * FROM T1 UNION (SELECT * FROM T2 INTERSECT SELECT * FROM T3)

Select expressions

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

Specify DISTINCT to remove any duplicates in the result.

Specify GROUP BY and HAVING in connection with aggregate functions to calculate summary values from the data in a table. The WHERE clause (if present) limits the number of rows included in the summary. If an aggregate function but no GROUP BY clause is present, a summary for the whole table is calculated. If a GROUP BY clause is present, a summary is computed for each unique set of values for the columns listed in the GROUP BY. Then, if the HAVING clause is present, it filters out complete groups given the conditional expression in the HAVING clause.

Summary queries have additional rules 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 can't nest aggregator functions.

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

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

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

Examples

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 it is not present in the GROUP BY reference list.

## Join expressions

Join expressions in JDataStore give access to a wide variety of join mechanisms. The two most commonly used, inner joins and cross joins, can be expressed with a `SELECT` expression alone, but any kind of outer join must be expressed with a `JOIN` expression.

- **CROSS JOIN**
  
  `A CROSS JOIN B`

  produces the same result set as

  `SELECT A.*, B.* FROM A,B`

- **INNER JOIN**
  
  `A INNER JOIN B ON A.X=B.X`

  produces the same result as

  `SELECT A.*, B.* FROM A,B WHERE A.X=B.X`

- **LEFT OUTER**
  
  `A LEFT OUTER JOIN B ON A.X=B.X`

  produces the rows from the corresponding inner join plus the rows from A that didn’t contribute, filling in the spaces corresponding to columns in B with NULLs.

- **RIGHT OUTER**
  
  `A RIGHT OUTER JOIN B ON A.X=B.X`

  produces the rows from the corresponding inner join plus the rows from B that didn’t contribute, filling in the spaces corresponding to columns in A with NULLs.

- **FULL OUTER**
  
  `A FULL OUTER JOIN B ON A.X=B.X`

  produces the rows from the corresponding inner join plus the rows from A and B that didn’t contribute, filling in the spaces corresponding to columns in B and A with NULLs.
UNION

A UNION JOIN B
produces a result similar to
A LEFT OUTER JOIN B ON FALSE
UNION ALL
A RIGHT OUTER JOIN B ON FALSE

a table with columns for all columns in A and B, with all the rows from A having NULL values for columns from B appended with all the rows from B having NULL values for columns from A.

NATURAL, ON, and USING are mutually exclusive:

ON
ON is an expression that needs to be fulfilled for a JOIN expression.

USING
USING ( C1, C2, C3)
is equivalent to the ON expression
A.C1=B.C1 AND A.C2=B.C2 AND A.C3=B.C3,
except that the resulting table only has columns C1, C2, and C3 once in the table, and those are the three first columns.

NATURAL
NATURAL is the same as a USING clause with the column names that appear in both tables A and B.

Syntax

<join expression> ::=<table reference> CROSS JOIN <table reference>
| <table reference> [NATURAL] [ INNER ] JOIN <table reference>
| [ <join kind> ]
| [ <join kind> ]
| [ <join kind> ]
| [ <join kind> ]
| <table reference> UNION JOIN <table reference>

<table reference> ::=<join expression>
| <table name> [AS] <range variable>
| ( <table expression> )

<range variable> ::= <SQL identifier>

<join kind> ::=ON <conditional expression>
| USING ( <column name commalist> )

Examples

SELECT * FROM Tinvoice FULL OUTER JOIN Titem USING (*InvoiceNumber*)

SELECT * FROM Tinvoice LEFT JOIN Titem ON Tinvoice."InvoiceNumber"
Statements

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

- Data Definition Language for managing tables and indexes, but no schema, domain, views, or security elements.
- Data manipulation and selection with **INSERT**, **UPDATE**, **DELETE**, and **SELECT**; but no cursors.
- Cursor operations are supported through the JDBC version 2.0 ResultSet API.
- Support for general table expressions including **JOIN**, **UNION**, and **INTERSECT**.

Syntax

```sql
<SQL statement> ::= 
  <data definition statement> 
  | <transaction control statement> 
  | <data manipulation statement>

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

<transaction control statement> ::= 
  <commit statement> 
  | <rollback statement> 
  | <set autocommit statement>

<data manipulation statement> ::= 
  <select statement> 
  | <single row select statement> 
  | <delete statement> 
  | <insert statement> 
  | <update statement> 
  | <call statement> 
  | <lock statement>
```
CREATE TABLE

This statement creates a JDataStore table. A column name and data type must be defined for each column.

Optionally, you can specify a default value for each column, along with uniqueness constraints.

You can also optionally specify a foreign key and primary key. JDataStore supports the use of one or more columns as a primary key or foreign key.

Syntax

```sql
<create table statement> ::= 
    CREATE TABLE <table name> ( <table element commalist> )

<table name> ::=  <SQL identifier>

<table element> ::= 
    <column definition>
    | <primary key>
    | <unique key>
    | <foreign key>

<column definition> ::= 
    <column name> <data type>
    [DEFAULT <default value>]
    [[NOT] NULL]
    [AUTOINCREMENT]
    [CONSTRAINT <constraint name> PRIMARY KEY]
    [CONSTRAINT <constraint name> UNIQUE]
    [CONSTRAINT <constraint name> <references definition>]

<column name> ::=  <SQL identifier>

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

<primary key> ::= 
    [CONSTRAINT <constraint name> PRIMARY KEY <column name commalist>]

<unique key> ::= 
    [CONSTRAINT <constraint name> UNIQUE ( <column name commalist>)]

<foreign key> ::= 
    [CONSTRAINT <constraint name> FOREIGN KEY <column name commalist>]

<references definition> ::= 
    REFERENCES <table name> [ ( <column name commalist> )]
    [ON DELETE <action>]
    [ON UPDATE <action>]
    [NO CHECK]
```
<action> ::=  
   NO ACTION  
   | CASCADE  
   | SET DEFAULT  
   | SET NULL  

<constraint name> ::=  <SQL identifier>  

Note: The NO CHECK option creates the foreign key without checking the consistency at creation time. Use this option with caution.

Examples

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

Example of creating a table using two columns for the primary key constraint:

CREATE TABLE t1 {c1 INT, c2 STRING, c3 STRING, primary key {c1, c2}}

Using AutoIncrement Columns with SQL

To create or alter a table to have an AutoIncrement column using SQL, add the AUTOINCREMENT keyword in your <table element> definition.

The following example creates table t1 with an integer AutoIncrement column called c1:

CREATE TABLE t1 {c1 INT AUTOINCREMENT, c2 DATE, c3 CHAR(32)}

To obtain the AutoIncrement value of a newly inserted row using the JDS JDBC driver using JVM version 1.3 or earlier, the JdsStatement.getGeneratedKeys() method can be called. This method is also available in the statement interface of JDBC 3 in JVM 1.4.)

**ALTER TABLE**

This statement adds and removes columns in a JDataStore table, and sets new column defaults and constraints.

**Syntax**

<alter table statement> ::=  
   ALTER TABLE <table name> <change definition commalist>

<change definition> ::=  
   <add column element>  
   | <drop column element>  
   | <alter column element>  
   | <add constraint>  
   | <drop constraint>  

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

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

<alter column element> ::=
ALTER [COLUMN] <column name> SET <default-definition>
| ALTER [COLUMN] <column name> DROP DEFAULT

<add constraint> ::= ADD <base table constraint>

<base table constraint> ::=  
| <primary key> | <unique key> | <foreign key>

<drop constraint> ::= DROP CONSTRAINT <constraint name>

<primary key> ::=  
| [CONSTRAINT <constraint name>] PRIMARY KEY <column name commalist>)

<unique key> ::=  
| [CONSTRAINT <constraint name>] UNIQUE ( <column name commalist>)

<foreign key> ::=  
| [CONSTRAINT <constraint name>] FOREIGN KEY {<column name commalist>}
| references definition

<references definition> ::=  
| REFERENCES <table name> [( <column name commalist> )]
| [ON DELETE <action>]  
| [ON UPDATE <action>]  
| [NO CHECK]

$action> ::=  
| NO ACTION  
| CASCADE  
| SET DEFAULT  
| SET NULL

<constraint name> ::= <SQL identifier>

In ALTER [COLUMN], the optional COLUMN keyword is included for SQL compatibility. It has no effect.

Example

ALTER TABLE Orders Add ShipDate DATE, DROP Amount

DROP TABLE

This statement deletes a table and its indexes from the JDataStore.

Syntax

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

Example

DROP TABLE Orders
**CREATE INDEX**

This statement creates an index for a JDataStore table. Each column can be ordered in ascending or descending order. The default value is ascending order.

**Syntax**

```plaintext
<create index statement> ::= 
  CREATE [UNIQUE] [CASESENSITIVE] INDEX <index name> 
  ON <table name> ( <index element comma list> )
```

```plaintext
@index name ::= <SQL identifier>
@index element ::= <column name> [DESC | ASC]
```

**Example**

This generates a non-unique, case-sensitive, ascending index on the `ITEM` column of the `ORDERS` table:

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

**DROP INDEX**

This statement deletes an index from a JDataStore table.

**Syntax**

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

**Example**

This deletes the index `ORDERINDEX` on the `ORDERS` table:

```sql
DROP INDEX OrderIndex ON Orders
```

**CREATE JAVA_METHOD**

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

**Syntax**

```plaintext
<create java method statement> ::= 
  CREATE JAVA_METHOD <method name> AS <method definition>
```

```plaintext
<method name> ::= <SQL identifier>
```

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

**Example**

```sql
CREATE JAVA_METHOD ABS AS 'java.lang.Math.abs'
```
DROP JAVA_METHOD

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

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

Example
DROP JAVA_METHOD ABS

COMMIT

This statement commits the current transaction. It does not have any effect unless AUTOCOMMIT is turned off.

Syntax
<commit statement> ::= COMMIT [ WORK ]

ROLLBACK

This statement rolls back the current transaction. This statement will not have any effect unless AUTOCOMMIT is turned off.

Syntax
<rollback statement> ::= ROLLBACK [ WORK ]

SET AUTOCOMMIT

This statement changes the autocommit mode. The mode is also controllable directly via the JDBC Connection instance.

Syntax
<set autocommit statement> ::= SET AUTOCOMMIT { ON | OFF }

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, obtains all rows including duplicates. The data can optionally be sorted using ORDER BY. The retrieved rows can optionally be locked for an upcoming UPDATE by specifying FOR UPDATE.

Syntax
<select statement> ::= 
    <table expression> [ ORDER BY <order item list> ] [ FOR UPDATE ]

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

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

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

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

Examples

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 given expression.

SELECT INTO

A SELECT INTO is a SELECT statement that evaluates into exactly one row, whose values are retrieved in output parameters. It is an error if the SELECT evaluates into more than one row or to the empty set.

Syntax

<single row select statement> ::= 
    SELECT [ ALL | DISTINCT ] <select item commalist> 
    INTO <parameter commalist> 
    FROM <table reference commalist> 
    [ WHERE <conditional expression> ] 
    [ GROUP BY <column reference commalist> ] 
    [ HAVING <conditional expression> ]

Example

SELECT CustId, Amount INTO ?, ? FROM Orders WHERE CustId=?
The first two parameter markers indicate output parameters from which the result of the query can be retrieved.

**INSERT**

An `INSERT` statement inserts rows into a table in the JDataStore. A list of columns with associated values are listed in the `INSERT` statement. Columns that aren’t listed in the statement are set to their default values.

**Syntax**

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

```
<insert table expression> ::= 
    <select expression>
    | VALUES (<expression commalist>)
```

**Example**

The following statement should be used in connection with a PreparedStatement in JDBC. It inserts one row each time it is executed. The columns not mentioned are set to their default values. If a column doesn’t have a default value, it’s set to NULL.

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

The following statement finds all the orders from the customer with `CustId = 123` and inserts 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 statement changes are listed explicitly. All the rows for which the `WHERE` clause evaluates to `TRUE` are changed. If no `WHERE` clause is specified, all rows in the table are changed.

**Syntax**

```
<update statement> ::= 
    UPDATE <table name>
    SET <update assignment commalist>
    [ WHERE <conditional expression> ]
```

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

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

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 disposable underwater cameras to 7.25:

```
UPDATE Orders SET Price = 7.25 WHERE Price > 7.25 AND Item = 'UWCameras'
```

DELETE

A DELETE statement deletes rows from a table in the JDataStore. If no WHERE clause is specified, all the rows are deleted. Otherwise only the rows that match the WHERE expression are deleted.

Syntax

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

Example

```
DELETE FROM Orders WHERE Item = 'Shorts'
```

CALL

A stored procedure is called using the CALL statement.

Syntax

```
<call statement> ::= 
    [ ? = ] CALL <method name> ( <expression commalist> )
```

Examples

```
?=CALL ABS(-765)
```

The parameter marker indicates an output parameter position from which the result of the stored procedure can be retrieved.

```
CALL IncreaseSalaries(10)
```

The Java method implementing IncreaseSalaries updates the salaries table with an increase of some percentage for all employees. A java.sql.Connection will implicitly be passed to the Java method. An updateCount of all the rows affected by IncreaseSalaries will be returned from Statement.executeUpdate().
LOCK TABLE

A table can be explicitly locked using the LOCK TABLE statement. The lock ceases to exist when the transaction is committed or rolled back.

Syntax

<lock statement> ::=  
    LOCK <table name commalist>

Example

LOCK Orders, LineItems
Index

A

ABSOLUTE function C-14
access 2-12
JDataStore tables and JDBC 3-17
multiple users and 2-14, 5-1
remote 5-1
accessor methods 10-5
adding data rows
to JDataStore tables C-29
allocation
JDataStore streams 2-24
ALogDir property 3-9, 10-6
ALTER TABLE statement C-24
appendRow method 3-5
applications
creating for offline editing 8-2
AUTOCOMMIT C-27
auto-commit 3-16
automatic crash recovery 3-5

B

backing up JDataStore files 2-22
BETWEEN operator C-11
blocked connections 6-6
blockSize property 2-14
BLogDir property 3-9, 10-6
bypassing transaction support 3-9
byte arrays 2-16

C

cache 10-2
controlling disk writes 10-4
JDataStore databases as 9-11
saving for JDataStore B-1
CALL statement 7-4, C-30
case
string conversions C-15
CAST function C-17
casting to another data type C-17
changing
transaction settings 3-7, 9-10
changing data
in JDataStore database 3-11
in persistent columns 8-7
with SQL statements C-29
CHAR_LENGTH function C-14
CHARACTER_LENGTH function C-14
closeDirectory method 2-11
closing
cache 10-2
connections 2-3
DataStore database 3-14
file streams 2-15
JDataStore database 9-11, 10-2
corrections 8-6
CollationKey classes B-2
commit method 3-10
commit mode 10-6
COMMIT statement C-27
commits 3-11, 10-6
connections
blocked 6-6
DataStore component 2-3, 3-2
implicit parameter 7-4
pooling JDBC 6-6
referencing JDataStore 2-14
remote drivers for 5-2
transaction processing and 3-11, 3-19
constants 2-9
contacting Borland 1-3
controlling disk writes 10-4
Copy Streams dialog box 9-9
copying
JDataStore streams 9-8
copying JDataStore contents 2-12
copying JDataStore streams 2-20
copyStreams method 2-20, 2-27
crash recovery 3-5, 10-6
CREATE INDEX statement C-26
CREATE JAVA_METHOD statement 7-1, C-26
CREATE TABLE statement C-23
createFileStream method 2-23
creating applications
for offline editing 8-2
creating file streams 2-13
creating queries C-1
creating standalone tables 8-7
creating table streams 3-3
creating transaction log files 3-8
Critical Section locks 6-3
CURRENT_DATE function C-15
CURRENT_TIME function C-15
CURRENT_TIMESTAMP function C-15

demon threads 10-2, 10-4
data cache 10-2
controlling disk writes 10-4
persistent 9-11
saving for JDataStore B-1
data modules
  DataStore components and 10-5
data sets
  JDataStores and 8-4
data streams 2-12
data type coercions 8-6
data types
  casting C-17
  portable C-2
  return type mapping 7-5
  type order for Java types 7-2
  types for Java outputs 7-3
database tutorials
  accessing JDataStores with JDBC 3-17
  controlling transactions 3-11
  embedding databases 3-1
  offline editing 8-2
databases 3-1
  providing functionality for 1-1
DataExpress package
  importing 3-2
DATASET_EXISTS error code 2-13
DataSetException class 2-4
DataStore component
  exception handling 2-4
  troubleshooting tips B-1
  tutorial for usage 8-2
  usage overview 1-1
  usage recommendations 10-2
DataStore connections
  referencing 2-14
DataStore streams
  verifying B-1
DataStore tables
  troubleshooting incorrect sorting B-2
DataStoreConnection class 2-3, 3-11
DataStoreDemo sample application 8-2
DataStoreException class 2-4
date and time functions C-6
dates 2-9, C-15
  extracting parts C-15
DDL Table locks 6-3
deadlocks 6-1
  avoiding 6-6
debugging
  JDataStores B-1
Delete JDataStore command 9-22
DELETE statement C-30
deleteStream method 2-24
DeleteTest.java 2-25
deleting
  JDataStore files 2-4, 3-10, 9-22
  JDataStore streams 2-24, 9-8
  JDataStore streams example 2-25
delimited text files 9-16
deploying
  JDataStore Server 5-3
deployment
  minimizing resources for 10-8
developer support 1-3
Dir.java 2-7
  running 2-10
directory listings 2-10
directory sort order 2-10
directory table
  adding entries 2-26
  closing 2-11
  contents described 2-9
  creating 2-27
  reading 2-10
dirty reads 6-1
disk writes 10-4
  soft commit mode and 10-6
DISTINCT keyword C-27
documentation 1-2
drivers
  JDBC connections 5-2
DROP INDEX statement C-26
DROP JAVA_METHOD statement C-27
DROP TABLE statement C-25
Dup.java 2-22
DxTable.java 3-1
directing
in persistent columns 8-7
dir editing tutorial 8-2
embedded databases 3-1
changing data 3-11
opening in read-only mode 3-9
populating 3-4
encryption
  JDataStore file 9-23
escape functions C-5
date and time functions C-6
numeric functions C-6
string functions C-6
system functions C-6
EXCEPT keyword C-27
EXCEPT operator C-18
exceptions
  JDataStore 2-4
EXISTS operator C-14
expressions
  SQL syntax for C-10
EXTRACT function C-15
F
field separators 9-16
file length 2-13
file streams 2-1
closing 2-15
creating 2-13
getting length 2-13
importing files as 9-17
maintaining 2-24
testing for 2-10, 2-11
viewing contents 9-6
writing to 2-14
fileExists method 2-11
files
importing 2-13
marked as opened 9-4
storage capacity A-1
type-specific viewers 9-6
FileStream object 2-12
instantiating 2-14
foreign keys C-23

G
garbage collection 10-2

H
heuristicCompletion 6-7
HIDDEN_STREAM bit 2-10

I
identifiers
JDataStore SQL engine C-9
image files 9-6
implicit connection parameter 7-4
import statements 8-3
importFile.java 2-12
importing data
DataExpress packages 3-2
JDataStore files 2-13
tables 9-12
text files 9-16
IN operator C-13
incorrect sorting B-2
indexes
creating 9-18, C-26
removing from JDataStores C-26
troubleshooting B-2
input parameters for UDFs and stored procedures 7-2
InputStream marking 2-15
INSERT statement C-29
installation support
for JDataStore 1-3
internal streams 2-10
INTERSECT keyword C-27
INTERSECT operator C-18
IS operator C-12
isolation levels 6-1
setting 6-2

J
JAR files
JDBC JDataStore remote server 5-3
Java datatypes 7-2, 7-3
Java objects
getting 2-5
storing 2-4
JDataStore
as embedded database 3-1
connections 2-3
literals supported C-4
restructuring 8-5
security 9-22
security features 4-1
tutorial for serializing objects 2-2
JDataStore connections 3-2
JDataStore database
as data cache 9-11
changing transaction settings 9-10
closing 9-11, 10-2
copying 9-8
enabling transaction support for 9-9
mapping table for 8-7
removing transaction support 3-10, 9-10
JDataStore Explorer 9-1
adding users 9-23
Administer Users 9-22
as query console 9-11
Change Password 9-23
editing users 9-23
Encrypt JDataStore 9-23
file operations with 9-21
hierarchical views 9-5, 9-6
importing with 9-16
limitations as query console 9-12
loading TXManager 9-9
managing queries with 9-12, 9-15
overview 1-3, 9-3
removing a user 9-23
saving queries with 9-15
security features 9-22
starting 9-1
validating contents 9-9
JDataStore Explorer startup options 9-3
JDataStore files 2-1, 9-21
backing up 2-22
bypassing transaction support for 3-9
capacity A-1
changing transaction settings 3-7
copying 2-12, 2-20
creating 2-3, 9-3
deleting 2-4, 9-22
enabling transaction support for 3-5, 3-7
getting contents of 2-6, 2-8
logging changes 3-8
opening 9-4
packing 2-27, 9-21
remote access 5-1
restructuring 8-7
sorting 2-10
upgrading 2-20, 2-22, 9-21
viewing 9-5
JDataStore Server
deploying 5-3
implementation customization 5-4
JDataStore SQL
identifiers C-9
keywords C-7
JDataStore streams
deleting 2-24, 9-8
example for deleting 2-25
formats 2-9
manipulating 9-7
naming conventions 2-21
specifications for A-1
undeleting 2-24, 2-25, 9-8
viewing contents 9-6
working with 2-15
JDataStore tables 3-4
accessing 3-17
adding records C-29
changing data 3-11
creating C-23
date and time literals C-5
deleting C-25
deleting records in C-30
JDBC escape sequences C-5
locking 3-31
modifying C-24
opening in read-only mode 3-9
persistent data in 8-1
populating 3-3, 9-12, 9-15
saving changes to 9-15
selecting records C-27
string functions C-5
updating C-29
JDBC application
creating 2-16
JDBC connections 3-17
transaction control and 3-19
JDBC custom servers 5-4
JDBC drivers 3-17, 5-2

running 5-2
using to execute SQL C-1
JDBC escape sequences C-5
JdbcTable.java 3-17
join expressions C-20
join expressions in SQL syntax C-20

K

keywords
JDataStore SQL engine C-7

L

language for UDFs and stored procedures 7-1
leading characters C-16
length of file streams 2-13
license keys 1-3
LIKE operator C-12
lists in SQL syntax C-17
literal values
JDataStore SQL C-4
literals
date and time literals C-7
literals in JDBC escape sequences C-5
lock manager 6-3
LOCK TABLE statement C-31
locking JDataStores C-31
locks 6-1
Critical Section 6-3
DDL Table 6-3
Row 6-3
log files 10-6
creating for transactions 3-8
moving 3-9
saving B-2
LOWER function C-15

M

maintenance
JDataStore 1-3
mapping return types 7-5
mapping table 8-7
mark method 2-15
maxLogSize property 3-9
moving transaction log files 3-9
multi-user access 2-14, 5-1
transaction processing for 6-1
multi-user connections 3-11

N

naming streams 2-21
New JDataStore dialog box 9-3, 9-4
nonrepeatable reads 6-1
numeric functions C-6

O
offline editing 8-2
openDirectory method 2-6, 2-8
openFileStream method 2-15
opening
  connections 2-3
  transactional JDataStores 3-7
operating system crashes 10-6
  recovering from 3-5
operators
  precedence in SQL queries C-10
  ordinal positions 8-7
output 2-16
  output parameters for UDFs and stored procedures 7-3
overloaded method signatures 7-5

P
Pack JDataStore command 9-21
packaging remote servers 5-3
packing JDataStore files 2-27, 9-21
padding characters C-16
password
  changing 9-23
  persistent columns
    editing 8-7
persistent data 8-1
  persistent data cache 9-11
  persistent storage 2-6
phantom reads 6-1
pooling connections 6-6
populating JDataStore tables 3-3, 9-12, 9-15
POSITION function C-15
power loss 10-6
PrintFile.java 2-15
providers
  DataStore components and 3-4

Q
quantified comparisons C-13
queries
  adding expressions C-10
  case sensitivity C-9
  creating C-1
  running in JDataStore Explorer 9-11, 9-12, 9-15
query console 9-11
  limitations for 9-12

R
random access 2-12
read access 2-12
read method 2-16
read-only mode 3-9
read-only transactions 6-6, 10-6
reconfiguring remote servers 5-3
reconfiguring JDataStore connections 2-14
remote access 5-1
remote editing 8-2
remote servers
  packaging 5-3
  reconfiguring 5-3
  starting 5-2, 5-4
removing transaction support 3-10, 9-10
renaming streams 2-21, 9-7
reserved words
  JDataStore SQL engine C-7
reset method 2-15
resolvers
  DataStore components and 3-4
resources
  freeing 10-2
  minimizing for deployment 10-8
restructuring JDataStore files 8-7
retrieving Java objects 2-5
return type mapping 7-5
rollback method 3-10
ROLLBACK statement C-27
rollbacks 3-8, 3-11
Row locks 6-3
saveMode property 10-4
saving JDataStore tables 9-15
saving log files B-2
scalar literals C-4
SCHEMA files 9-16
secondary indexes
  troubleshooting B-2
security
  features of JDataStore 9-22
  security features
    JDataStore 4-1
seek method 2-15
select method 2-15
select expressions C-19
select expressions in SQL syntax C-19
SELECT INTO statement C-28
SELECT statement C-27
selecting data rows
  from JDataStore tables C-27
separators for delimited text files 9-16
serializable transactions 6-3
serializing objects
  in a JDataStore 2-2
servers
  deploying JDataStore 5-3

Index I-37
JDataStore implementation 5-4
  reconfiguring 5-3
  starting remote 5-2, 5-4
SET AUTOCOMMIT statement C-27
single-connection applications 3-11
soft commit mode 10-6
sort keys
troubleshooting B-2
sorting JDataStore files 2-10
SQL functions C-14
  date and time functions C-6
  escape functions C-5
  numeric functions C-6
  string functions C-6
  system functions C-6
SQL identifiers C-9
SQL keywords C-7
SQL predicates C-11
SQL queries
  case sensitivity C-9
  creating C-1
  running in JDataStore Explorer 9-11, 9-12, 9-15
SQL reference C-1
SQL statements C-18, C-19, C-20, C-22
  adding expressions C-10
  executing 9-20
  executing using JDBC C-1
  lists in syntax notation C-17
SQRT function C-16
standalone tables 8-7
starting
  JDataStore Explorer 9-1
  remote servers 5-2, 5-4
startup options
  JDataStore Explorer 9-3
  storage 1-1, 8-1
  advantages for persistent 2-6
  data streams and 2-12
  Java objects 2-4
  JDataStore file size A-1
StorageDataSet component
  DataStore components and 8-2, 10-5
  overview 3-2
  restructuring for JDataStores 8-5
stored procedures
  adding C-26
  calling C-5, C-30
  declaring 7-1
  definition 7-1
  dropping C-27
  implicit connection parameter 7-4
  input parameters 7-2
  language 7-1
  output parameters 7-3
  overloaded method signatures 7-5
  return type mapping 7-5
stream types 2-10
  determining 2-10, 2-11
streams 2-1
  checking existence of 2-11
  closing file 2-15
  copying 2-20, 9-8
  creating file 2-13
  creating table 3-3
  deleting JDataStore 2-24, 9-8
  example for JDataStore 2-25
  JDataStore formats 2-9
  moving to any position 2-15
  naming/renameing 2-21, 9-7
  specifications for JDataStore A-1
  storing arbitrary files 2-12
  undeleting JDataStore 2-24, 2-25, 9-8
  unique IDs and 2-10
  verifying B-1
  viewing contents for JDataStore 9-6
  writing to 2-14
StreamVerifier class B-1
string delimiters 9-16
string functions C-6
strings C-15
  extracting substrings C-16
  getting position C-15
  removing padding characters C-16
SUBSTRING function C-16
substrings C-15
  getting C-16
  support options 1-3
  system functions C-6
T
  table expressions C-18
  table expressions in SQL syntax C-18
  Table locks 6-3
  table streams 2-1
    creating 3-3
    maintaining 2-24
    testing for 2-10, 2-11
    viewing contents 9-6
  TableDataSet component
    as embedded database 3-1
    tableExists method 2-11
    tables
      adding records C-29
      creating 9-17, C-23
      creating standalone 8-7
      deleting records C-30
      importing 9-12
      importing text files as 9-16
      modifying C-24
      removing from JDataStores C-25
selecting records C-27
technical support 1-3
text files 9-6
  importing 9-16
time C-15
  extracting parts C-15
time formats 2-9
timestamps C-15
trailing characters C-16
transaction
  contexts 3-11
  isolation 3-5
transaction isolation
  levels described 6-1
transaction log files 10-6
  creating 3-8
  moving 3-9
  saving B-2
transactions 10-5
  bypassing 3-9
  changing 3-7, 9-10
  committing C-27
  controlling 3-10, 3-11, 3-19
  distributed 6-6
  enabling for DataStore files 3-5
  enabling for JDataStore files 3-7
  enabling with JDataStore Explorer 9-9
  heuristicCompletion 6-7
  manipulating with TXManager 9-10
  multi-user access and 6-1
  removing for JDataStore database 3-10, 9-10
  rolling back C-27
  setting autocommit mode C-27
  tracking 3-8
  tutorial for 3-11
usage overview 3-11
TRIM function C-16
tutorials 2-1
  accessing JDataStores with JDBC 3-17
  controlling transactions 3-11
  embedding databases 3-1
  offline editing 8-2
   TxManager class 3-5
   TxManager Properties dialog box 9-10
   type coercions 8-6

U

UDFs
  adding C-26
  declaring 7-1
  definition 7-1
  dropping C-27
  example of a UDF 7-2
  implicit connection parameter 7-4
  input parameters 7-2
  language 7-1
  output parameters 7-3
  overloaded method signatures 7-5
  return type mapping 7-5
  undeleteStream method 2-25
  undeleting JDataStore streams 2-24, 2-25
  example 2-25
    with JDataStore Explorer 9-8
UPDATE statement C-29
UNION ALL operator C-18
UNION keyword C-27
UNION operator C-18
unquoted SQL
  identifiers C-9
UPDATE statement C-29
UPPER function C-15
users
  adding 9-23
  administering 9-22
  editing 9-23
  removing 9-23

V

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

W

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

X

XA support 6-6