



# Guía del desarrollador

---



VERSIÓN 6

**Borland®**  
**JDataStore™**

Borland Software Corporation  
100 Enterprise Way, Scotts Valley, CA 95066-3249  
[www.borland.com](http://www.borland.com)

### **Archivos redistribuibles**

En el archivo `deploy.html` ubicado en el directorio raíz del producto JBuilder encontrará una lista completa de archivos que se pueden distribuir de acuerdo con la licencia de JBuilder y la limitación de responsabilidad.

Borland Software Corporation puede tener patentes concedidas o en tramitación sobre los temas tratados en este documento. Dirijase al CD del producto o al cuadro de diálogo Acerca de para la lista de patentes. La modificación de este documento no le otorga derechos sobre las licencias de estas patentes.

COPYRIGHT © 1997-2003 Borland Software Corporation. Reservados todos los derechos. Todos los nombres de productos y marcas de Borland son marcas comerciales o registradas de Borland Software Corporation en Estados Unidos y otros países. Las otras marcas pertenecen de sus respectivos propietarios.

Si desea más información acerca de las condiciones de contrato de terceras partes y acerca de la limitación de responsabilidades, consulte las notas de esta versión en su CD de instalación de JBuilder.

Impreso en EE.UU.

dg6

0203040506-9 8 7 6 5 4 3 2 1

PDF

# Índice de materias

## Capítulo 1

### Introducción

Cuándo utilizar JDataStore . . . . .	1-1
JDataStore y DataStore . . . . .	1-2
Conocimientos necesarios. . . . .	1-2
Contenido del manual. . . . .	1-2
Utilización de JDataStore por primera vez . . .	1-3
Introducción al Explorador de JDataStore . . .	1-3
Distribución de componentes de aplicaciones JDataStore. . . . .	1-4

## Capítulo 2

### Fundamentos de JDataStore

Introducción a JDataStore. . . . .	2-1
Serialización de objetos . . . . .	2-2
Demostración de clase: Hola.java. . . . .	2-2
Creación de archivos JDataStore . . . . .	2-3
Apertura y cierre de una conexión . . . . .	2-3
Tratamiento de las excepciones básicas de JDataStore . . . . .	2-4
Eliminación de archivos JDataStore . . . . .	2-4
Almacenamiento de objetos Java . . . . .	2-5
Recuperación de objetos Java . . . . .	2-6
Ventajas del almacenamiento persistente de objetos . . . . .	2-6
Utilización del directorio de JDataStore. . . . .	2-7
Demostración de clase: Dir.java . . . . .	2-8
Apertura de directorios JDataStore. . . . .	2-9
Contenido del directorio JDataStore . . . . .	2-9
Detalles del flujo . . . . .	2-10
Orden de clasificación en directorios. . . .	2-11
Lectura de directorios JDataStore. . . . .	2-11
Cierre del directorio JDataStore. . . . .	2-12
Búsqueda de flujos . . . . .	2-12
Almacenamiento de archivos arbitrarios . . . .	2-13
Demostración de clase: ImportFile.java . . .	2-13
Creación de flujos de archivo . . . . .	2-15
Referencias al archivo JDataStore conectado . . . . .	2-15
Escritura en un flujo de archivo. . . . .	2-16
Cierre de flujos de archivo . . . . .	2-16
Apertura, búsqueda y lectura de flujos de archivo. . . . .	2-16
Creación de una aplicación JDBC básica mediante JDataStore. . . . .	2-18
Copia de flujos . . . . .	2-22

Parámetros de copyStreams . . . . .	2-22
Asignación y cambio del nombre de los flujos copiados . . . . .	2-23
Demostración de clase: Dup.java . . . . .	2-24
Eliminación y recuperación de flujos . . . . .	2-26
Eliminación de flujos . . . . .	2-26
Utilización de los bloques eliminados en JDataStore . . . . .	2-26
Recuperación de flujos JDataStore. . . . .	2-27
Demostración de clase: DeleteTest.java. . .	2-27
Búsqueda de elementos del directorio . . .	2-28
Filas de directorio independientes . . . . .	2-29
Empaquetado de archivos JDataStore. . . .	2-29

## Capítulo 3

### JDataStore como base de datos incrustada

Utilización de DataExpress para acceder a datos . . . . .	3-1
Demostración de clase: DxTable.java . . . .	3-2
Conexión con JDataStore mediante StorageDataSet. . . . .	3-2
Creación de tablas de JDataStore con DataExpress . . . . .	3-4
Utilización de tablas de JDataStore con DataExpress . . . . .	3-5
JDataStore transaccional . . . . .	3-6
Activación de admisión de transacciones. . .	3-6
Creación de archivos JDataStore transaccionales . . . . .	3-7
Activación de transacciones en archivos JDataStore creados previamente . . . .	3-8
Apertura de archivos JDataStore transaccionales . . . . .	3-8
Modificación de la configuración de transacciones . . . . .	3-9
Históricos de transacciones . . . . .	3-10
Desplazamiento de históricos de transacciones . . . . .	3-11
Desviación de admisión de transacciones .	3-11
Desactivación de la característica de transacciones. . . . .	3-12
Eliminación de archivos JDataStore transaccionales. . . . .	3-12
Control de las transacciones de JDataStore . .	3-12
Arquitectura de las transacciones . . . . .	3-13

Confirmación y cancelación de transacciones . . . . .	3-14
Tutorial: Control de transacciones por medio de DataExpress . . . . .	3-14
Paso 1: Creación de un archivo JDataStore transaccional con datos de prueba . . . . .	3-15
Paso 2: Creación de un módulo de datos. . . . .	3-15
Paso 3: Creación de una interfaz gráfica de usuario para la tabla JDataStore . . . . .	3-16
Paso 4: Adición del control directo de transacciones . . . . .	3-18
Paso 5: Adición del control de confirmación automática. . . . .	3-19
Utilización de JDBC para acceder a los datos. . . . .	3-20
Demostración de clase: JdbcTable.java. . . . .	3-20
Control de transacciones por medio de JDBC . . . . .	3-23

## Capítulo 4

### Utilización de las características de seguridad de JDataStore

Autenticación de usuario . . . . .	4-1
Autorización . . . . .	4-2
Encriptación en JDataStore . . . . .	4-2
Cómo aplicar la seguridad de JDataStore . . . . .	4-3

## Capítulo 5

### Acceso remoto y multiusuario a DataStore

Realización de una conexión con un JDBC local . . . . .	5-1
Especificación de propiedades ampliadas . . . . .	5-2
Controlador JDBC para acceso remoto . . . . .	5-2
Ejecución del Servidor de JDataStore . . . . .	5-3
Cambio de configuración del servidor. . . . .	5-3
Distribución del servidor de JDataStore. . . . .	5-4
Empaquetado del servidor . . . . .	5-4
Inicio del servidor . . . . .	5-4
Creación de servidores JDBC personalizados . . . . .	5-5

## Capítulo 6

### Transacciones y agrupación de conexiones

Problemas de las transacciones multiusuario. . . . .	6-1
Niveles de aislamiento de transacciones. . . . .	6-1
Configuración de niveles de aislamiento en conexiones JDataStore . . . . .	6-2

Bloqueos utilizados por el gestor de bloqueos de JDataStore . . . . .	6-3
Propiedades JDBC ampliadas que controlan el comportamiento de bloqueo de JDataStore . . . . .	6-4
Utilización de bloqueos en JDataStore y niveles de aislamiento. . . . .	6-5
Depuración de problemas de expiración del tiempo de espera para bloqueos y conflictos . . . . .	6-6
Cómo evitar bloqueos y conflictos. . . . .	6-6
Conservación de las transacciones de escritura. . . . .	6-7
Transacciones de sólo lectura. . . . .	6-7
Cambios en el control de concurrencia para versiones anteriores . . . . .	6-7
Agrupación de conexiones y soporte de transacciones distribuidas . . . . .	6-8
Agrupación de conexiones . . . . .	6-8
heuristicCompletion . . . . .	6-9

## Capítulo 7

### Las UDF y los procedimientos almacenados

Lenguaje para los procedimientos almacenados y las UDF. . . . .	7-1
Procedimientos almacenados o UDF disponibles para el motor SQL . . . . .	7-2
Un ejemplo de UDF. . . . .	7-2
Parámetros de entrada . . . . .	7-3
Parámetros de salida . . . . .	7-3
Parámetro implícito de conexión . . . . .	7-5
Firmas de métodos sobrecargados . . . . .	7-5
Asignación de tipos devueltos. . . . .	7-6

## Capítulo 8

### Persistencia de datos en un JDataStore

Utilización de DataStore en lugar de MemoryStore . . . . .	8-1
Utilización de JDataStore con objetos StorageDataSets . . . . .	8-2
Tutorial: Edición fuera de línea con JDataStore . . . . .	8-3
Gestión de datos fuera de línea con JDataStore . . . . .	8-5
Reestructuración de los StorageDataSet de JDataStore . . . . .	8-6
Conversión de tipos de datos . . . . .	8-7

Editor de columnas persistentes . . . . .	8-8
Cambios en la estructura . . . . .	8-9
<b>Capítulo 9</b>	
<b>El Explorador de JDataStore</b>	
Inicio del Explorador de JDataStore . . . . .	9-1
Inicio desde la línea de comandos . . . . .	9-2
Operaciones básicas de JDataStore. . . . .	9-3
Creación de archivos JDataStore . . . . .	9-3
Apertura de un archivo JDataStore. . . . .	9-4
Definición de las opciones de apertura de los archivos JDataStore . . . . .	9-4
Apertura de un archivo JDataStore cerrado inadecuadamente . . . . .	9-4
Visualización de la información en archivos JDataStore . . . . .	9-5
Visualización del contenido del flujo . . . . .	9-7
Redenominación de flujos. . . . .	9-8
Eliminación de flujos. . . . .	9-9
Recuperación de flujos. . . . .	9-9
Copia de flujos JDataStore. . . . .	9-9
Comprobación del JDataStore . . . . .	9-10
Conversión del JDataStore en transaccional . . . . .	9-11
Modificación de los valores de transacción . . . . .	9-11
Desactivación de la característica de transacciones . . . . .	9-12
Cierre de archivos JDataStore. . . . .	9-12
El Explorador de JDataStore como consola de consultas. . . . .	9-12
Utilización del Explorador de JDataStore para gestionar consultas . . . . .	9-13
Limitaciones del Explorador de JDataStore . . . . .	9-13
Creación y mantenimiento de consultas y conexiones. . . . .	9-14
Captura y edición de datos . . . . .	9-16
Actualización de datos y almacenamiento de los cambios . . . . .	9-17
Importación de tablas y archivos. . . . .	9-18
Importar archivos de texto como tablas . . . . .	9-18
Importación de archivos. . . . .	9-19
Creación de tablas . . . . .	9-19
Creación de índices . . . . .	9-20
Ejecución de SQL . . . . .	9-22
Operaciones con archivos JDataStore . . . . .	9-23
Empaquetado de archivos JDataStore . . . . .	9-23
Actualización de archivos JDataStore . . . . .	9-24
Eliminación de archivos JDataStore . . . . .	9-24
Tareas de seguridad de JDataStore . . . . .	9-24
Gestión de usuarios . . . . .	9-24
Adición de usuarios . . . . .	9-25
Modificación de los derechos de usuario . . . . .	9-26
Eliminación de usuarios. . . . .	9-26
Cambio de la contraseña . . . . .	9-26
Encriptación de un JDataStore . . . . .	9-26
<b>Capítulo 10</b>	
<b>Optimización de aplicaciones JDataStore</b>	
Carga rápida de bases de datos . . . . .	10-1
Recomendaciones generales. . . . .	10-2
Cierre del archivo JDataStore . . . . .	10-2
Optimización de la caché de disco de JDataStore . . . . .	10-3
Optimización de ubicaciones de archivos . . . . .	10-3
Control de la frecuencia de la escritura en disco de los bloques de la memoria caché . . . . .	10-4
Ajuste de la memoria . . . . .	10-5
Consejos para mejorar el rendimiento . . . . .	10-5
Componentes adicionales de JDataStore . . . . .	10-6
Módulos de datos. . . . .	10-6
Optimización de aplicaciones transaccionales. . . . .	10-7
Transacciones de sólo lectura . . . . .	10-7
Modo de confirmación por software . . . . .	10-8
Históricos de transacciones . . . . .	10-8
Desactivación del registro de estado. . . . .	10-8
Optimización del rendimiento del control de concurrencia de JDataStore . . . . .	10-8
Las operaciones multihilo . . . . .	10-9
Depuración de recursos publicados . . . . .	10-10
Columnas con autoincremento . . . . .	10-10
Columnas con autoincremento en DataExpress . . . . .	10-11
Columnas con autoincremento en SQL. . . . .	10-11
<b>Apéndice A</b>	
<b>Especificaciones</b>	
Capacidad del archivo JDataStore . . . . .	A-1
Nombres de flujo de JDataStore. . . . .	A-2
<b>Apéndice B</b>	
<b>Resolución de problemas</b>	
Depuración de aplicaciones JDataStore . . . . .	B-1

Comprobación del contenido de JDataStore . . .	B-1
Problemas en la búsqueda y la ordenación de datos . . . . .	B-2
Almacenamiento de históricos . . . . .	B-2

## Apéndice C

### Referencia de SQL

Acceso a SQL mediante JDBC . . . . .	C-1
Tipos de datos . . . . .	C-2
Literales . . . . .	C-4
Secuencias de escape JDBC . . . . .	C-5
Funciones de escape . . . . .	C-6
Funciones de cadena . . . . .	C-6
Funciones numéricas. . . . .	C-7
Funciones de fecha y hora. . . . .	C-7
Funciones del sistema . . . . .	C-7
Palabras clave . . . . .	C-8
Identificadores . . . . .	C-10
Expresiones . . . . .	C-11
Predicados . . . . .	C-12
BETWEEN. . . . .	C-13
IS . . . . .	C-13
LIKE . . . . .	C-13
IN . . . . .	C-14
Comparaciones cuantificadas. . . . .	C-15
EXISTS. . . . .	C-15
Funciones . . . . .	C-16
ABSOLUTE . . . . .	C-16
CHAR_LENGTH y CHARACTER_LENGTH . . . . .	C-16
CURRENT_DATE, CURRENT_TIME y CURRENT_TIMESTAMP . . . . .	C-16

EXTRACT . . . . .	C-17
LOWER y UPPER . . . . .	C-17
POSITION . . . . .	C-17
SQRT . . . . .	C-18
SUBSTRING . . . . .	C-18
TRIM . . . . .	C-18
CAST . . . . .	C-19
La sintaxis de las sentencias . . . . .	C-19
Listas en la sintaxis . . . . .	C-20
Expresiones de tabla . . . . .	C-20
Expresiones de selección . . . . .	C-21
Expresiones de unión . . . . .	C-23
Sentencias . . . . .	C-25
CREATE TABLE. . . . .	C-25
ALTER TABLE. . . . .	C-27
DROP TABLE . . . . .	C-29
CREATE INDEX. . . . .	C-29
DROP INDEX . . . . .	C-29
CREATE JAVA_METHOD. . . . .	C-30
DROP JAVA_METHOD . . . . .	C-30
COMMIT . . . . .	C-30
ROLLBACK . . . . .	C-31
SET AUTOCOMMIT . . . . .	C-31
SELECT. . . . .	C-31
SELECT INTO . . . . .	C-32
INSERT . . . . .	C-32
UPDATE . . . . .	C-33
DELETE . . . . .	C-34
CALL . . . . .	C-34
LOCK TABLE . . . . .	C-35

## Índice

I-1

## Introducción

*JDataStore* es una polifacética solución de almacenamiento de datos, de altas prestaciones y bajo consumo de recursos, desarrollada enteramente para Java™. *JDataStore* proporciona:

- Una base de datos relacional incrustada sin administración, con interfaces DataExpress y JDBC, que acepta acceso con transacciones, sin bloqueo, para varios usuarios, y cuenta con recuperación de detención del sistema. Para obtener más información, consulte la *Guía del desarrollador de aplicaciones de base de datos de JBuilder*.
- Un almacén de objetos serializados, tablas y otros flujos de archivo.
- Los componentes JavaBean que pueden manipularse con herramientas visuales de creación de Beans, como JBuilder.

### Cuándo utilizar JDataStore

---

*JDataStore* permite:

- Incrustar funciones de base de datos compatibles con SQL-92 directamente en la aplicación, sin necesidad de un motor de base de datos externo. Se puede acceder a las bases de datos por medio del controlador JDBC de *JDataStore* o los componentes de DataExpress. *JDataStore* acepta la mayoría de tipos de datos JDBC, incluido el objeto Java.
- Serializar todos los objetos de la aplicación y los flujos de archivo en un solo archivo físico, para mayor comodidad y portabilidad.

- Permitir las aplicaciones móviles y fuera de línea. Mediante los componentes JavaBean de DataExpress, JDataStore duplica y guarda en caché de manera asíncrona los datos que provienen de una fuente de datos, permite el acceso y actualización de datos, y actualiza los cambios en la fuente de datos. Los datos pueden proceder de un servidor de base de datos, un servidor de aplicaciones CORBA, SAP, BAAN, o de otra fuente de datos.
- Aumentar el rendimiento de las aplicaciones en línea de DataExpress con conjuntos de datos voluminosos utilizando un DataStore en vez del MemoryStore por defecto.

## JDataStore y DataStore

---

JDataStore es el nombre del producto, herramientas y formato de archivo. Dentro de este producto se encuentra un paquete DataStore que incluye una clase datastore, así como varias clases que tienen "datastore" parte de su nombre.

## Conocimientos necesarios

---

La *Guía del desarrollador de JDataStore* supone que tiene un conocimiento previo acerca de:

- Programación en Java
- La interfaz de usuario de JBuilder (cómo crear, gestionar y ejecutar proyectos y cómo utilizar las herramientas de diseño)
- DataExpress (nociones básicas)
- JDBC (nociones básicas)
- SQL básico

## Contenido del manual

---

La *Guía del Desarrollador de JDataStore* es una guía general y un tutorial para usar JDataStore, con material de referencia. Incluye los siguientes capítulos:

- [Fundamentos de JDataStore](#) describe la estructura básica de un sistema de archivos JDataStore. Se sirve de flujos de archivo para mostrar el funcionamiento de diferentes tareas administrativas.
- [JDataStore como base de datos incrustada](#) explica cómo crear una base de datos de transacciones de JDataStore y utilizarla como base de datos incrustada con un ejemplo de aplicación de interfaz gráfica.



- [Utilización de las características de seguridad de JDataStore](#) describe la autenticación de usuarios y el cifrado que proporciona JDataStore.
- [Acceso remoto y multiusuario a DataStore](#) introduce el servidor de JDataStore utilizado para el acceso remoto. También trata aspectos de las transacciones referentes multiusuario.
- [Persistencia de datos en un JDataStore](#) explica la forma de utilizar JDataStore para el almacenamiento de datos persistentes en caché, para la ejecución fuera de línea.
- [El Explorador de JDataStore](#) describe el Explorador de JDataStore.
- [Optimización de aplicaciones JDataStore](#) contiene varias sugerencias para mejorar el rendimiento, la fiabilidad y el tamaño de las aplicaciones JDataStore.
- [Resolución de problemas](#) explica cómo depurar las aplicaciones JDataStore y solucionar problemas comunes.
- [Especificaciones](#) enumera las especificaciones del formato de archivos de JDataStore.
- [Referencia de SQL](#) es una guía de referencia del sublenguaje SQL-92 del controlador JDBC de JDataStore.

## Utilización de JDataStore por primera vez

---

JDataStore incluye una licencia de desarrollo gratuita. Cuando esté preparado para distribuir JDataStore debe adquirir licencias de distribución adicionales. Si desea más información, póngase en contacto con el servicio al cliente de Borland.

También puede obtener licencias en el almacén de Borland en línea en <http://shop.borland.com>.

## Introducción al Explorador de JDataStore

---

El Explorador de JDataStore es una herramienta visual creada enteramente con Java que ayuda a gestionar los JDataStores. Esto se detalla en el [El Explorador de JDataStore](#). Puede hacerse una idea de las posibilidades de JDataStore si utiliza el Explorador de JDataStore y los archivos JDataStore de muestra incluidos con JBuilder.

El Explorador de JDataStore proporciona herramientas visuales para efectuar muchas tareas de mantenimiento. La guía del desarrollador explica los fundamentos de la utilización de una API básica de JDataStore. Sin embargo, se recomienda empezar por el capítulo dedicado al Explorador de JDataStore.

En la *Guía del desarrollador de JDataStore* aparece la siguiente anotación cuando una tarea no se puede efectuar de forma visual con el Explorador de JDataStore:

DSX: Esta anotación está acompañada por una referencia a la tarea en el Explorador de JDataStore.

## Distribución de componentes de aplicaciones JDataStore

---

Puede encontrar información acerca de la distribución del servidor de JDataStore para el acceso remoto en [Distribución del servidor de JDataStore](#). Para sugerencias acerca de la reducción del tamaño de la distribución de aplicaciones cliente de JDataStore, consulte [Depuración de recursos publicados](#).

Recuerde: JDataStore se suministra únicamente con una licencia de desarrollo. Deberá adquirir licencias adicionales si desea distribuir programas. Si desea más información, póngase en contacto con el servicio al cliente de Borland.

# Fundamentos de JDataStore

Este capítulo contiene varios tutoriales que muestran los conceptos básicos de JDataStore. Si aún no ha leído la [Introducción](#), dedíquese unos minutos antes de empezar los tutoriales.

## Introducción a JDataStore

---

Los archivos JDataStore pueden contener dos tipos básicos de flujos de datos: de *tabla* y de *archivo*.

Los flujos de tabla pueden ser tablas de base de datos completas, creadas por las API de JDBC o de DataExpress. También incluyen datos de las tablas almacenadas en caché de una fuente de base de datos externa, como un servidor de base de datos. La configuración de la propiedad `store` de un `StorageDataSet` al `DataStore` crea los datos de tabla caché.

Los archivos de flujo pueden subdividirse en dos categorías:

- Archivos arbitrarios, creados con `DataStoreConnection.createFileStream( )`. En estos flujos se puede leer, escribir y efectuar búsquedas.
- Los objetos Java serializados se almacenan como flujos de archivo.

**Nota:** Se pueden almacenar todo tipo de flujos en el mismo archivo JDataStore.

Un nombre (existe diferencia entre mayúscula y minúscula) llamado `storeName` en la API identifica cada flujo. Este nombre puede exceder de 192 bytes, y se almacena junto con otra información sobre el flujo, en el directorio interno del JDataStore. La barra normal ("/") se utiliza en el nombre como separador de directorios, para proporcionar una organización jerárquica de los directorios.

El Explorador de JDataStore utiliza esta estructura para mostrar el contenido de un DataStore en un árbol.

En la primera parte de este capítulo se tratan los fundamentos de JDataStore mediante flujos de archivo. Para obtener más información acerca de trabajar con flujos de tabla, consulte [“Creación de una aplicación JDBC básica mediante JDataStore” en la página 2-18](#) y [Persistencia de datos en un JDataStore](#). Tal vez desee consultar el ejemplo que crea una aplicación sencilla de JDBC utilizando JDataStore en `/samples/JDataStore/HelloJDBC/`.

## Serialización de objetos

---

Un DataStore es un componente que se puede programar visualmente. Sin embargo, durante el proceso de aprendizaje de los DataStores, resulta más conveniente escribir ejemplos de código sencillos que muestren el funcionamiento de un DataStore. Este tema se trata en el presente capítulo.

El clásico primer ejercicio para aprender a utilizar un lenguaje de programación consiste en mostrar el mensaje “¡Hola a todos!”. Aquí vamos a respetar la tradición. (Sin embargo, no tendrá que realizar el clásico segundo ejercicio: el conversor de Fahrenheit a Celsius).

En primer lugar, cree un proyecto nuevo para el paquete `dsbasic` que se va a utilizar en diversas partes de este capítulo.

**Importante:** Añada al proyecto la biblioteca JDataStore para poder acceder a las clases de JDataStore. Si no sabe cómo crear un proyecto o añadir una biblioteca consulte [“Cómo añadir una biblioteca a un proyecto” en Guía del desarrollador de aplicaciones de base de datos](#).

## Demostración de clase: Hola.java

---

Añada un nuevo archivo al proyecto, `Hola.java`, y escriba este código:

```
// Hola.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();
    }
}
}

```

Tras declarar el paquete, esta clase importa todas las clases del paquete `com.borland.datastore`. Ese paquete contiene la mayoría de las clases públicas de `JDataStore`. (El resto se encuentran en el paquete `com.borland.datastore.jdbc`, que sólo es necesario para el acceso a JDBC. Contiene la clase del controlador de JDBC y las clases que se utilizan para implementar el servidor de `JDataStore`. Si desea más información acerca de las clases, consulte [JDataStore como base de datos incrustada y Acceso remoto y multiusuario a DataStore](#)) Es posible acceder a `JDataStore` a través de los componentes DataExpress (paquetes en `com.borland.dx`). En este ejemplo, las referencias son explícitas de manera que pueda ver de dónde procede cada una.

## Creación de archivos `JDataStore`

---

En el método `main()` de `Hello.java`, se crea un objeto `DataStore`. Este objeto representa un archivo `DataStore` que contiene propiedades y métodos que representan su estructura y configuración.

A continuación, se asigna el nombre “Basic.jds” a la propiedad `fileName` del objeto `DataStore`. Contiene la extensión “.jds” por defecto, en minúsculas. Si el nombre de archivo no tiene esta extensión, se le añade automáticamente cuando se configura esta propiedad.

No es posible crear el archivo `JDataStore` si ya existe un archivo con ese nombre. Si el archivo no existe, el método `create()` lo crea. Si este método falla por algún motivo (por ejemplo porque no hay espacio en el disco o porque alguien ha creado un archivo con este nombre en el momento comprendido entre la ejecución de esta sentencia y la de la última) se lanza una excepción. Si el método tiene éxito, tiene una conexión abierta a un nuevo archivo `JDataStore`.

**DSX:** Consulte [“Creación de archivos `JDataStore`” en la página 9-3](#). Cuando se crea el archivo también se pueden elegir opciones como el tamaño del bloque y la condición de transaccional del `JDataStore`.

## Apertura y cierre de una conexión

---

Si el archivo no existe, se abre una conexión a través del método `open()`. `open()` es un método de la superclase de la clase `DataStore`, `DataStoreConnection`, que tiene propiedades y métodos para acceder a los contenidos de `JDataStore`. (La propiedad `fileName` también se encuentra en

`DataStoreConnection`, lo que significa que a menudo se puede acceder a un archivo `JDataStore` sin un objeto `DataStore`, como se verá más adelante). Dado que `DataStore` es una subclase de `DataStoreConnection`, tiene su propia conexión incorporada adecuada para aplicaciones sencillas como ésta. (`DataStore` puede crear un archivo `JDataStore`, pero `DataStoreConnection` no puede).

Pero la emoción dura poco. Inmediatamente después de abrir una conexión a un archivo `JDataStore` se crea un archivo en el proceso, si es necesario. Esta conexión se cierra con el método `close()`. El método `close()` es heredado de `DataStoreConnection`. Dado que sólo existía una conexión incorporada, cuando se cierran las conexiones con el archivo `JDataStore`, éste también se cierra.

Se deben cerrar todas las conexiones que se hayan abierto antes de salir de la aplicación (o llamar al método `DataStore.shutdown()` que cierra todas las conexiones). La apertura de una conexión inicia un hilo de utilidad de conexión que sigue ejecutándose e impide a la aplicación cerrarse correctamente. Es necesario cerrar estas conexiones o la aplicación quedará bloqueada cuando intente salir.

## Tratamiento de las excepciones básicas de `JDataStore`

---

La mayoría de los métodos de las clases `JDataStore` puede lanzar una excepción `DataSetException`, o concretamente una de sus subclases, `DataStoreException`. Casi todas estas excepciones se producen cuando se hace algo indebido. Por ejemplo, no se puede asignar valores a la propiedad `fileName` si la conexión ya está abierta. No se puede crear el archivo `JDataStore` si ya existe. No se puede abrir una conexión si el archivo mencionado no es realmente un `JDataStore`. Se puede producir una excepción de entrada y salida cuando se escriben datos en el cierre de una conexión.

Por tanto, casi todo el código de `JDataStore` se encuentra dentro de un bloque `try`. En este caso, si se lanza una excepción se imprime un seguimiento de la pila.

## Eliminación de archivos `JDataStore`

---

Si ejecuta la aplicación ahora, lo único que se consigue es crear el archivo `Basic.jds`. Si se ejecuta por segunda vez, hace menos aún; se limita a abrir y cerrar una conexión. Antes de continuar es necesario borrar el archivo.

No existe ninguna función especial para borrar archivos `JDataStore`. Se puede utilizar el método `java.io.File.delete()` o cualquier otro que sirva el propósito. Como ejemplo de ayuda, si se desea crear siempre un archivo `JDataStore`, se puede utilizar un fragmento de código como el siguiente:

```
// almacenar es un archivo DataStore con la propiedad fileName configurada
java.io.File storeFile = new java.io.File( store.getFileName() );
if ( storeFile.exists() ) {
    storeFile.delete();
}
store.create();
```

Si el archivo `JDataStore` es transaccional, está acompañado de archivos históricos de transacciones que también se deben borrar. Si desea más información sobre los históricos de transacciones, consulte [Históricos de transacciones](#).

**DSX:** Consulte [Eliminación de archivos JDataStore](#). El Explorador de `JDataStore` borra automáticamente todos los históricos de transacciones asociados.

## Almacenamiento de objetos Java

---

Añada las sentencias en negrita al bloque `if` del método `main()`:

```
if ( !new java.io.File( store.getFileName() ).exists() ) {
    store.create();
    try {
        store.writeObject( "hola", "¡Hola, DataStore! Es " + new
java.util.Date() );
    } catch ( java.io.IOException ioe ) {
        ioe.printStackTrace();
    }
    } else {
```

El método `writeObject()` intenta almacenar un objeto Java como flujo de archivo en `JDataStore` empleando la serialización de Java. (En las tablas también se pueden almacenar objetos). El objeto que se va a guardar debe implementar la interfaz `java.io.Serializable`. Si no es así, se lanza una `java.io.IOException` (más concretamente, `java.io.NotSerializableException`). Otro motivo para que se lance la excepción es que falle la escritura (por ejemplo, si se acaba el espacio en disco).

El primer parámetro de `writeObject()` establece el `storeName`, nombre que identifica el objeto en el archivo `JDataStore`. Este nombre distingue entre mayúsculas y minúsculas. El segundo parámetro es el objeto que se va a almacenar. En este caso, se trata de una cadena con un saludo y la fecha y la hora actuales. La clase `java.lang.String` implementa `java.io.Serializable`, por lo que la cadena se puede almacenar con `writeObject`.

## Recuperación de objetos Java

---

Añada las sentencias en negrita al bloque `else` del método `main()`:

```

    } else {
        store.open();

        try {
            String s = (String) store.readObject( "hola" );
            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();
        }
    }
}

```

El método `readObject()` recupera el objeto del `JDataStore`. Igual que `writeObject()`, puede lanzar una `IOException` por motivos como un fallo del disco. Tampoco puede reconstruir el objeto almacenado sin su clase. Si esta clase no se encuentra en la vía de acceso a clases, `readObject()` lanza una `java.lang.ClassNotFoundException`.

Si no se puede encontrar el objeto mencionado, se lanza una `DataStoreException` con el código de error `STORE_NOT_FOUND`.

`DataStoreException` es una subclase de `DataSetException`. Es importante identificar esta excepción aquí, aunque haya otro bloque `catch` al final del método, porque si se pasa directamente a él se saltaría la llamada al método `close()` de la conexión `JDataStore`. (El código se estructura de esta forma atípica para enseñar unos principios determinados).

Dado que `readObject()` devuelve un objeto `java.lang.Object`, casi siempre se cambia el valor devuelto al del tipo de datos esperado. (Si el objeto no es del tipo esperado, se obtiene una `java.lang.ClassCastException`). En este caso se trata más de una formalidad, ya que el método `System.out.println` puede tomar referencia de un objeto genérico.

## Ventajas del almacenamiento persistente de objetos

---

Ahora, ya puede ejecutar `Hello.java`. La primera vez que se ejecuta, crea el archivo `JDataStore` y almacena la cadena de saludo. En las ejecuciones posteriores se mostrará en la consola el saludo, con la fecha y la hora.

Para el sencillo almacenamiento persistente de objetos, los archivos `JDataStore` tienen una serie de ventajas respecto al uso de las clases JDK del paquete `java.io`:



- Es más sencillo, utiliza una sola clase en vez de cuatro (`FileOutputStream`, `ObjectOutputStream`, `FileInputStream` y `ObjectInputStream`).
- Es posible almacenar todos los objetos en un solo archivo y acceder a ellos con un nombre lógico, en lugar de recorrer todos los objetos de un archivo de flujo.
- Con un solo archivo es imposible perder de forma accidental un objeto o dos, como ocurriría con archivos independientes. Puede que también se reduzca el espacio utilizado, porque los archivos independientes suelen desperdiciar mucho espacio debido al método de asignación de clústeres en el disco. El tamaño por defecto del bloque en un archivo `JDataStore` es pequeño (4KB).
- Dado que no se depende del sistema de archivos del anfitrión, la aplicación es más portable. Por ejemplo, cada sistema operativo permite diferentes caracteres en los nombres. Algunos sistemas distinguen entre mayúsculas y minúsculas y otros no. Las convenciones internas de nomenclatura de los archivos `JDataStore` son compatibles con todas las plataformas.
- Proporciona un sistema de archivo cifrado.

Un sistema de directorio interno tendría poca utilidad si no permite acceder a su contenido.

## Utilización del directorio de JDataStore

---

El método `DataStoreConnection.openDirectory()` devuelve el contenido de `JDataStore` en una estructura. Pero primero añada el siguiente programa, `AddObjects.java`, y ejecútelo para añadir más objetos al archivo `JDataStore`:

```
// AddObjects.java
package dsbasic;
import com.borland.datastore.*;
public class AddObjects {
    public static void main( String[] args ) {
        DataStoreConnection store = new DataStoreConnection();
        int[] intArray = { 5, 7, 9 };
        java.util.Date date = new java.util.Date();
        java.util.Properties properties = new java.util.Properties();
        properties.setProperty( "una propiedad", "un valor" );
    try {
        store.setFileName( "Basic.jds" );
        store.open();
        store.writeObject( "add/create-time", date );
        store.writeObject( "add/values", properties );
        store.writeObject( "add/array of ints", intArray );
    } catch ( com.borland.dx.dataset.DataSetException dse ) {
        dse.printStackTrace();
    } catch ( java.io.IOException ioe ) {
        ioe.printStackTrace();
    }
}
```

```

        } finally {
    try {
        store.close();
    } catch ( com.borland.dx.dataset.DataSetException dse ) {
        dse.printStackTrace();
    }
}
}
}
}

```

El programa funciona de forma ligeramente diferente a `Hello.java`. En primer lugar, utiliza un objeto `DataStoreConnection` en vez de un `DataStore` para acceder al archivo `JDataStore`, aunque se utiliza de la misma forma. Se asigna un valor a la propiedad `fileName`, se abre la conexión `open()`, se utiliza el método `writeObject()` para almacenar los objetos y, por último, se cierra la conexión `close()`.

Otra diferencia está en el lugar que ocupa la llamada al método `close()`. Dado que siempre hay que llamar a `close()`, independientemente de lo que ocurra en la parte principal del método, se coloca detrás de los bloques `catch`, dentro de un bloque `finally`. De este modo, la conexión siempre se cierra aunque se produzca un error sin tratar. El método `close()` es seguro para llamar incluso si la conexión nunca se ha abierto. En este caso, `close()` no hace nada.

Esta vez se escriben tres objetos en el archivo `JDataStore`: una matriz de enteros, un objeto `Date` (no un objeto `Date` convertido en cadena) y una tabla con direccionamiento calculado. Se les asigna el nombre de forma que se encuentren en un directorio llamado “add”. La barra (/) es el carácter de separación de directorios. Uno de los nombres contiene espacios, que son caracteres válidos.

## Demostración de clase: Dir.java

---

Añada al proyecto otro archivo, `Dir.java`:

```

// Dir.java
package dsbasic;
import com.borland.datastore.*;
public class Dir {
    public static void print( String storeFileName ) {
        DataStoreConnection store = new DataStoreConnection();
        com.borland.dx.dataset.StorageDataSet storeDir;
    try {
        store.setFileName( storeFileName );
        store.open();
        storeDir = store.openDirectory();
        while ( storeDir.inBounds() ) {
            System.out.println( storeDir.getString(
                DataStore.DIR_STORE_NAME ) );
            storeDir.next();
        }
    }
}

```

```

        store.closeDirectory();
    } catch ( com.borland.dx.dataset.DataSetException dse ) {
        dse.printStackTrace();
    } finally {
try {
        store.close();
    } catch ( com.borland.dx.dataset.DataSetException dse ) {
        dse.printStackTrace();
    }
    }
}
public static void main( String[] args ) {
    if ( args.length > 0 ) {
        print( args[0] );
    }
}
}

```

Esta clase necesita un argumento de línea de comandos, el nombre de un archivo JDataStore, que se pasa a su método `print()`. Este método accede al archivo JDataStore con un código parecido al que se ha visto antes.

## Apertura de directorios JDataStore

`Dir.java` define un `DataStoreConnection` para acceder al JDataStore y declara un `StorageDataSet`. Después de abrir una conexión al JDataStore, el programa llama a un método `openDirectory()` de `DataStoreConnection` para obtener el contenido del directorio JDataStore. El directorio del archivo JDataStore está representado por una tabla.

**DSX:** Consulte [Visualización de la información en archivos JDataStore](#).

## Contenido del directorio JDataStore

La tabla del directorio JDataStore tiene nueve columnas, lo que significa que cada una de ellas contiene información concreta sobre cada uno de los flujos del archivo JDataStore, como se muestra en esta tabla:

**Tabla 2.1** Columnas de la tabla del directorio JDataStore

#	Nombre	Constante	Tipo	Contenido
1	State	DIR_STATE	short	Indica si el flujo está activo o borrado
2	DeleteTime	DIR_DEL_TIME	long	Si se ha borrado, cuándo; de lo contrario, cero
3	StoreName	DIR_STORE_NAME	String	El valor de <code>storeName</code>
4	Tipo	DIR_TYPE	short	Campos de bit que indican el tipo de flujos
5	Id	DIR_ID	int	Número identificador ID no repetido

**Tabla 2.1** Columnas de la tabla del directorio JDataStore (continuación)

#	Nombre	Constante	Tipo	Contenido
6	Properties	DIR_PROPERTIES	String	Propiedades y sucesos del flujo DataSet
7	ModTime	DIR_MOD_TIME	long	Hora de la última modificación del flujo
8	Length	DIR_LENGTH	long	Longitud en bytes del flujo
9	BlobLength	DIR_BLOB_LENGTH	long	Longitud en bytes de los BLOB de un flujo de tabla

Pueden referenciarse las columnas por nombre y número. Existe una constante definida como variable de la clase `DataStore` para cada nombre de columna. La mejor forma de hacer referencia a una columna es mediante estas constantes. Esto es porque incluyen una comprobación durante la compilación para garantizar que se está refiriendo a una columna válida. Las constantes cuyos nombres terminan en `_STATE` se utilizan para los distintos valores de la columna State. También existen constantes para los valores y máscaras de bits de la columna Type, con nombres que acaban en `_STREAM`.

## Detalles del flujo

La hora del directorio JDataStore se establece con el método UTC (Hora universal coordinada, una mezcla de las siglas francesas [TUC] e inglesas [CUT]). Están disponibles para la creación de fechas con `java.util.Date(long)`.

Como ocurre con muchos sistemas de archivo, cuando se borra algo de un JDataStore, el espacio que ocupaba se marca como disponible pero no se borra el contenido ni el elemento del directorio que lo señala. Esto permite deshacer las acciones de borrado. Para obtener más detalles, consulte [Eliminación y recuperación de flujos](#).

La columna Type indica si un flujo es de archivo o de tabla, pero también hay muchos subtipos internos de flujo de tabla (para los índices y los totalizadores, por ejemplo). Estos flujos internos se marcan con el bit `HIDDEN_STREAM`, que indica que no se deben mostrar. Por supuesto, cuando se lee el directorio se puede decidir si han de estar ocultos o visibles.

Estos flujos internos tienen el mismo nombre StoreName que el flujo de tabla al que están asociados. Esto significa que el StoreName no identifica cada uno de los flujos cuando interactúan a bajo nivel con el JDataStore. A menudo algunos tipos de flujo interno tienen múltiples instancias. Por tanto, el número de ID de cada flujo debe garantizar que no se produzcan repeticiones a bajo nivel. Sin embargo, el nombre StoreName basta como identificación para el parámetro `storeName` que se utiliza en la API. Por ejemplo, cuando se borra un flujo de tabla se borran todos los flujos que llevan el mismo nombre StoreName.

## Orden de clasificación en directorios

La tabla del directorio se ordena por las cinco primeras columnas. Los valores almacenados en la columna State hacen que los flujos activos se muestren en primer lugar por orden alfabético del nombre. Estos van seguidos por los flujos eliminados, ordenados por orden cronológico de eliminación, del más antiguo al más reciente. (No es posible utilizar `DataSetView` para cambiar el orden.)

## Lectura de directorios JDataStore

---

La tabla de directorios JDataStore se gestiona como cualquier otra tabla con la API de DataExpress. Utilice los métodos `next()` `inBounds()` para desplazarse en el directorio de cada entrada. Utilice el método `get<XXX>()` apropiado para leer la información deseada para cada flujo.

No se puede escribir en el directorio JDataStore ya que es de sólo lectura.

Para ejecutar `Dir.java`, establezca los parámetros de ejecución del cuadro de diálogo Propiedades del proyecto en el archivo JDataStore que se va a comprobar; en este caso, `Basic.jds`. Cuando se ejecuta, el directorio se recorre con un bucle y se enumeran los nombres de los flujos, como en:

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

Es posible añadir mucha más información a la lista del directorio. Lo más difícil es decidir el formato de la información disponible en todas las columnas del directorio JDataStore. Por ejemplo, para mostrar si el flujo es de tabla o de archivo, añada las sentencias en negrita al principio del bucle:

```
while ( storeDir.inBounds() ) {
    short dirVal = storeDir.getShort( DataStore.DIR_TYPE );
    if ( (dirVal & DataStore.TABLE_STREAM) != 0 ) {
        System.out.print( "T" );
    } else if ( (dirVal & DataStore.FILE_STREAM) != 0 ) {
        System.out.print( "F" );
    } else {
        System.out.print( "?" );
    }
    System.out.print( " " );
    System.out.println( storeDir.getString( DataStore.DIR_STORE_NAME ) );
    storeDir.next();
}
```

Esta adición cambia la vía de acceso a archivos generados a:

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

La vía de acceso a archivos generados indica que todos los objetos serializados son flujos de archivo.

## Cierre del directorio `JDataStore`

---

Cuando no se necesita el directorio `JDataStore`, resulta aconsejable cerrarlo llamando al `DataStoreConnection.closeDirectory()`. La mayoría de las operaciones efectuadas en `JDataStore` modifican de alguna forma el directorio. Si el directorio está abierto es absolutamente necesario comunicarle los cambios, lo que reduce la velocidad de ejecución de la aplicación.

Si se intenta acceder al directorio `StorageDataSet` cuando está cerrado, se obtiene una `DataSetException` con el código de error `DATASET_NOT_OPEN`.

## Búsqueda de flujos

---

Aunque es posible efectuar búsquedas manuales en el directorio `JDataStore`, `DataStoreConnection` cuenta con dos métodos para comprobar si un flujo existe sin necesidad de abrir el directorio. El método `tableExists()` comprueba los flujos de tabla y el método `fileExists()` comprueba los flujos de archivo. Los dos métodos utilizan el parámetro `storeName` y pasan por alto los flujos borrados. Devuelven `true` si en el archivo `JDataStore` hay un flujo activo del tipo correspondiente con ese nombre y `false` si no es así. Recuerde que los nombres de flujo distinguen entre mayúsculas y minúsculas y que no es posible tener un flujo de tabla y otro de archivo con el mismo nombre.

Por ejemplo, si se ejecuta el siguiente fragmento de código en `Basic.jds` tal y como se encuentra en este momento del tutorial:

```
store.tableExists( "hola" )
```

devuelve `false` porque, aunque hay un flujo llamado “hola”, es de archivo y no de tabla. Se obtiene el mismo resultado de la siguiente manera:

```
store.fileExists( "Hola" )
```

Esta vez no coinciden las mayúsculas y minúsculas en el nombre. Aquí coinciden el nombre y el tipo:

```
store.fileExists( "hola".)
```

devuelve `true`.

## Almacenamiento de archivos arbitrarios

---

Además de serializar objetos discretos como flujos de archivo es posible almacenar y recuperar flujos de datos en un archivo `DataStore` por medio de un objeto `com.borland.datastore`. Aunque `FileStream` es una subclase de `java.io.InputStream`, también tiene un método para escribir en el flujo, por lo que se puede utilizar el mismo objeto para acceso de lectura y escritura. También permite acceder de forma aleatoria con del método `seek()`. El hecho de que `FileStream` sea una subclase de `InputStream` facilita el uso de flujos almacenados en el archivo `JDataStore` en situaciones que precisan un flujo de entrada. Lo más habitual es leer más flujos de los que se escriben.

**DSX:** Consulte [“Importación de archivos” en la página 9-19](#).

### Demostración de clase: `ImportFile.java`

---

Supongamos que una aplicación utiliza documentos de uso frecuente que se modifican según el cliente. En la tabla de clientes hay un campo que contiene su copia personalizada, pero también es necesario almacenar el original para poder hacer más copias para los clientes nuevos. El original se puede almacenar como flujo de archivo en `JDataStore`. El siguiente programa, `ImportFile.java`, lo hace por usted. Añádalo al proyecto.

```
// ImportFile.java
package dsbasic;

import com.borland.datastore.*;

public class ImportFile {

    private static final String DATA    = "/data";
    private static final String LAST_MOD = "/modified";

    public static void read( String storeFileName,
                           String fileToImport ) {
        read( storeFileName, fileToImport, fileToImport );
    }

    public static void read( String storeFileName,
                           String fileToImport,
                           String streamName ) {
        DataStoreConnection store = new DataStoreConnection();

        try {

            store.setFileName( storeFileName );
            store.open();

            FileStream fs = store.createFileStream( streamName + DATA );

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

```

        java.io.File file = new java.io.File( fileToImport );
        java.io.FileInputStream fis = new java.io.FileInputStream( file );

        int bytesRead;
        while ( (bytesRead = fis.read( buffer )) != -1 ) {
            fs.write( buffer, 0, bytesRead );
        }
        fs.close();
        fis.close();

        store.writeObject( streamName + LAST_MOD,
                           new Long( file.lastModified() ) );
    } catch ( com.borland.dx.dataset.DataSetException dse ) {
        dse.printStackTrace();
    } catch ( java.io.FileNotFoundException fnfe ) {
        fnfe.printStackTrace();
    } catch ( java.io.IOException ioe ) {
        ioe.printStackTrace();
    } finally {
        try {
            store.close();
        } catch ( com.borland.dx.dataset.DataSetException dse ) {
            dse.printStackTrace();
        }
    }
}

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

```

Los parámetros del programa son el nombre de un archivo JDataStore, el nombre del archivo que se desea importar y un nombre de flujo optativo. Si no se indica otro nombre, se utilizará el nombre de archivo. El método `main()` llama a la forma apropiada del método `read()`, ya que el método `read()` de dos argumentos llama al método `read()` de tres argumentos.

Cuando se importa el archivo, se importa también la fecha en la que se ha modificado por última vez. El sufijo `"/modified"` se añade al nombre del flujo para esta fecha y el sufijo `"/data"` se añade para contener los datos del archivo. Estos sufijos se definen como variables de clase.

El método `read()` empieza con una conexión a un archivo JDataStore con un objeto `DataStoreConnection`.



## Creación de flujos de archivo

---

Como ocurre con la mayoría de las API de flujo de archivo, existen métodos independientes para crear flujos de archivo y acceder a los existentes. El método para crear un nuevo archivo de flujo es `createFileStream()` y su único parámetro es la propiedad `storeName` del flujo que se crea.

Si ya existe un flujo de archivo con ese nombre, incluso si se trata de un objeto serializado, se borrará sin previo aviso. Puede querer comprobar primero si ese archivo de flujo existe con el método `fileExists()` (`ImportFile.java` no realiza esta comprobación). Si existe un flujo de tabla con ese nombre, `createFileStream()` lanza una `DataStoreException` con el código de error `DATASET_EXISTS`, porque no puede tener un flujo de tabla y un flujo de archivo con el mismo nombre.

Si `createFileStream` se ejecuta correctamente, devuelve un objeto `FileStream` que representa el nuevo flujo de archivo, vacío.

## Referencias al archivo JDataStore conectado

---

Una sencilla operación de copia como ésta utiliza un bucle para leer y escribir el archivo. La cuestión es el tamaño de estas partes. Está el problema evidente de hacerlas demasiado pequeñas, pero si se hacen muy grandes también pueden ocasionar problemas de rendimiento. Un comienzo prudente consiste en elegir tamaños de partes de archivos múltiples del tamaño de bloque de `JDataStore`.

El tamaño de bloque de `JDataStore` se almacena en la propiedad `blockSize` del objeto `DataStore`. Cuando se utiliza `DataStoreConnection` para acceder a un archivo `JDataStore`, se crea automáticamente una instancia de `DataStore`. Los demás objetos `DataStoreConnection` del mismo proceso que se comparten al mismo archivo `JDataStore` comparten este objeto `DataStore`. (El acceso a un archivo `JDataStore` es exclusivo de un solo proceso, el acceso de varios usuarios se consigue por medio de un solo proceso del servidor). El objeto `DataStoreConnection` tiene una propiedad de sólo lectura, llamada `dataStore`, que contiene una referencia al objeto `DataStore` conectado.

El objeto `FileStream` escribe una matriz de bytes, que se declara en esta sentencia:

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

El método `getDataStore()` obtiene la referencia a un objeto `DataStore` y a partir de este método `getBlockSize()` obtiene la propiedad `blockSize`. El valor de la propiedad se especifica en kilobytes, por lo que se multiplica por 1024. El número de bloque resultante se multiplica por cuatro, es el número de bloques elegidos de forma arbitraria que se lee cada vez.

## Escritura en un flujo de archivo

---

El método `write()` del objeto `FileSteam` utiliza una matriz de bytes, como `java.io.OutputStream`, aunque la única forma de este método es la que también especifica la desviación y la longitud iniciales.

El objeto `java.io.FileInputStream` lee un archivo y lo copia en una matriz de bytes. Devuelve el número de bytes leídos o -1 cuando se llega al final del flujo. En el bucle se comprueba el número de bytes leídos para el valor de fin de archivo. Si no se trata del final del archivo, se escribe en la matriz el número de bytes leídos, empezando por el primero. En cada ejecución del bucle, excepto la última vez, se llena toda la matriz con los datos leídos y este contenido se escribe en el objeto `FileSteam`. La última ejecución del bucle probablemente no llene la matriz.

## Cierre de flujos de archivo

---

Cuando se termina de trabajar con un flujo de archivo es necesario cerrarlo. El objeto `FileSteam` utiliza el método `close()` (igual que `FileInputStream`).

Después de cerrar el flujo de archivo se escribe la fecha de la última modificación, empleando un objeto `java.lang.Long` para encapsular el valor `long` primitivo. (No es posible guardar valores primitivos con serialización).

Para comprobar `ImportFile.java`, intente importar varios archivos de código fuente a `Basic.jds`.

## Apertura, búsqueda y lectura de flujos de archivo

---

Utilice el método `openFileSteam()` para abrir un flujo de archivo por medio de su nombre. Igual que `createFileSteam()`, devuelve un objeto `FileSteam` al principio del flujo. Puede desplazarse a cualquier posición en el flujo con el método `seek()`, escribir en él y leerlo con el método `read()`. `FileSteam` es compatible con la marca `InputStream` con los métodos `mark()` y `reset()`.

El programa `PrintFile.java` permite apertura, búsqueda y lectura. Añádalo al proyecto.

```
// PrintFile.java
package dsbasic;

import com.borland.datastore.*;

public class PrintFile {

    private static final String DATA      = "/data";
    private static final String LAST_MOD = "/modified";
```

```

public static void printBackwards( String storeFileName,
                                   String streamName ) {
    DataStoreConnection store = new DataStoreConnection();

    try {
        store.setFileName( storeFileName );
        store.open();

        FileStream fs = store.openFileStream( streamName + DATA );
        int streamPos = fs.available();

        while ( --streamPos >= 0 ) {
            fs.seek( streamPos );
            System.out.print( (char) fs.read() );
        }
        fs.close();

        System.out.println( "Last modified: " + new java.util.Date(
            ((Long) store.readObject( streamName
                                     + LAST_MOD )).longValue() ) );
    } catch ( com.borland.dx.dataset.DataSetException dse ) {
        dse.printStackTrace();
    } catch ( java.io.IOException ioe ) {
        ioe.printStackTrace();
    } catch ( java.lang.ClassNotFoundException cnfe ) {
        cnfe.printStackTrace();
    } finally {
        try {
            store.close();
        } catch ( com.borland.dx.dataset.DataSetException dse ) {
            dse.printStackTrace();
        }
    }
}

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

```

Para demostrar el acceso aleatorio con el método `seek` (y para hacer las cosas ligeramente más interesantes) este programa imprime un flujo de archivo al revés. La longitud del flujo de archivo se determina por medio de una llamada del método `available()` de `FileStream` y se utiliza como puntero del archivo. Cuando se lee del archivo, el puntero se desplaza hacia delante. La posición del puntero retrocede y su ubicación se modifica por cada byte leído en el bucle. El método `read()` tiene dos formas: El primero lee en una matriz de byte (la misma forma del método usado por el `FileInputStream` en `ImportFile.java`). La segunda devuelve un solo byte. Aquí se utiliza la forma de un solo byte. Cada byte representa el carácter que se imprime.

## Creación de una aplicación JDBC básica mediante JDataStore

---

Ahora que ya sabe cómo crear y tratar flujos de archivos en un JDataStore, se abordarán los conceptos básicos de aplicaciones JDBC utilizando JDataStore. Para obtener información más detallada acerca de la creación de aplicaciones JDBC con JDataStore, consulte [JDataStore como base de datos incrustada](#)

El primer paso será crear un archivo en el paquete dsbasic llamado HelloTX.java. El código de este archivo es muy similar al del archivo Hello.java que creó anteriormente. Las diferencias aparecen en negrita:

```
// HolaTX.java
package dsbasic;

import com.borland.datastore.*;

public class HelloTX {

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

La diferencia más importante en este caso es que se crea una instancia TxManager que se asigna al gestor de transacciones del archivo JDataStore. Las aplicaciones JDBC precisa un JDataStore, transaccional, lo cual requiere un gestor de transacciones. Para abrir o crear un archivo JDataStore transaccional, también hay que asignar valores a la propiedad userName. Si no hay ningún nombre adecuado, se puede escribir cualquiera.

El siguiente paso es escribir código para una conexión con el DataStore. Añada un archivo con el nombre HolaJDBC.java al proyecto. Escriba el siguiente código en el archivo nuevo:

## Creación de una aplicación JDBC básica mediante JDataStore

```
//HolaJDBC.java

package dsbasic;

import java.sql.*;

public class HelloJDBC {

    public HelloJDBC() {
    }

    static void main(String args[]) {
        // Los controladores remotos y locales de JDataStore utilizan la misma
        // cadena de controlador:
        String DRIVER = "com.borland.datastore.jdbc.DataStoreDriver";

        // Utilice esta cadena para el controlador local:
        String URL = "jdbc:borland:dslocal:";

        // Utilice esta cadena para el controlador remoto (e inicie el servidor
        // JDataStore):
        // 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);
        }

        // De esta manera la conexión se cerrará incluso cuando se lancen
        // excepciones con anterioridad. Esto es importante, porque puede tener
        // problemas volviendo a abrir un archivo JDataStore después de dejar una
        // conexión a la misma abierta.
        try {
            if(c_open)
                con.close();
        }
        catch(Exception e3) {
            System.out.println(e3.toString());
        }
    }
}
```

Preste especial atención a las líneas de código en **negrita** de este programa. Son las más importantes en este ejemplo. Primero se indica el nombre del controlador JDBC JDataStore. Esta cadena es la misma para los controladores locales y remotos JDBC. A continuación aparece la cadena URL para conectarse con un JDataStore local. Para su información, el código también muestra la cadena remota, pero aparece comentada. Las dos últimas líneas en **negrita** son comunes a muchas aplicaciones JDBC y representan el momento de conexión al JDataStore.

Una vez realizada la conexión con el archivo JDataStore, deseará añadir y modificar datos. A continuación se mostrará cómo realizar estas operaciones. Por ahora no se analizarán, pero se explicará lo suficiente para cerciorarse de haber conectado con el JDataStore y que puede añadir, manipular, imprimir y borrar datos. Añada las siguientes líneas en **negrita** al código de la siguiente manera:

```
package dsbasic;

import java.sql.*;

public class HelloJDBC {

    public HelloJDBC() {
    }

    public static String formatResultSet(ResultSet rs) {
    // Este método formatea el conjunto de resultados para la impresión.

    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[]) {
    // Los controladores JDataStore remotos y locales utilizan la
    // misma cadena de controlador:
    String DRIVER = "com.borland.datastore.jdbc.DataStoreDriver";
```

```

// Utilice esta cadena para el controlador local:
String URL    = "jdbc:borland:dslocal: ";

// Utilice esta cadena para el controlador remoto (e inicie el servidor
// JDataStore):
// 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;

    // la siguiente línea crea una tabla en el JDataStore.
    stmt.executeUpdate("create table HelloJDBC" +
        "(COLOR varchar(15), " +
        " NUMBER int, " +
        " PRICE float)");

    // Los valores están insertados en la tabla con
    // las tres sentencias siguientes.
    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)");

    // Ahora realizamos una consulta a la tabla
    ResultSet rs = stmt.executeQuery("select * from HelloJDBC");

    // Llame a formatResultSet() para formatear
    // el resultado de la impresión.
    System.out.println(formatResultSet(rs));

    // La siguiente línea borra la tabla.
    stmt.executeUpdate("drop table HelloJDBC");
}

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

try {
    // Intenta limpiar llamando a
    // método java.sql.Statement.close().
    if(s_open)
        stmt.close();
}

```

```

        catch(Exception e2){
            System.out.println(e2.toString());
        }

        // De esta manera la conexión se cerrará incluso con excepciones lanzadas
        // con anterioridad. Esto es importante, porque puede tener problemas
        // volviendo a abrir un archivo JDatastore después de dejar una conexión
        // a la misma abierta.
    try {
        if(c_open)
            con.close();
    }
    catch(Exception e3) {
        System.out.println(e3.toString());
    }
}
}

```

En el ejemplo anterior, el código que se añade al método `main()` crea una tabla e inserta filas en dicha tabla. Entonces llama al método `formatResultSet()` e imprime los resultados. A continuación, borra las tablas desde el `JDataStore`. Finalmente, intenta limpiar llamando al método `close()` del objeto `java.sql.Statement`.

## Copia de flujos

---

El método `copyStreams()` de la clase `DataStoreConnection` hace una nueva copia de los flujos en el mismo `JDataStore` o en uno diferente. Si encuentra un error en un flujo original, intenta corregirlo en la copia. También, es posible utilizar `copyStreams()` para actualizar un archivo `JDataStore` antiguo en un formato actualizado.

### Parámetros de `copyStreams`

---

El método `copyStreams` tiene los seis parámetros que se enumeran en esta tabla:

**Tabla 2.2** Parámetros del método `DataStoreConnection.copyStreams`

Nombre del parámetro	Descripción
<code>sourcePrefix</code>	El nombre del flujo debe comenzar con esta indicación de directorio para que coincida con los patrones de concordancia; déjelo vacío para que concuerde con todos los flujos.
<code>sourcePattern</code>	Modelo del nombre de flujo con el que se debe concordar, en el que se incluyen los comodines estándar <code>*</code> y <code>?</code>
<code>destCon</code>	Conexión al archivo <code>JDataStore</code> de destino.



**Tabla 2.2** Parámetros del método `DataStoreConnection.copyStreams` (continuación)

Nombre del parámetro	Descripción
<code>destPrefix</code>	En el nombre de las copias de flujos, <code>sourcePrefix</code> se sustituye por este parámetro; debe tener el mismo valor que <code>sourcePrefix</code> si no se desea cambiar el nombre.
<code>options</code>	Ninguna o las siguientes variables de clase de <code>DataStore</code> : <ul style="list-style-type: none"> <li>• <code>COPY_CASE_SENSITIVE</code></li> <li>• <code>COPY_IGNORE_ERRORS</code></li> <li>• <code>COPY_OVERWRITE</code></li> </ul>
<code>out</code>	<code>java.io.PrintStream</code> para dirigir los mensajes de estado; <code>null</code> para suprimir la vía de archivos generados.

Los parámetros `options` invierten el comportamiento por defecto de `copyStreams`. Comportamiento por defecto:

- Pasa por alto el uso de mayúsculas y minúsculas al comparar nombres de flujos.
- Se detiene si se encuentra un error irrecuperable.
- Se detiene si ya existe un flujo con el nombre de destino.

Si `copyStreams()` se detiene a causa de uno de los dos motivos anteriores, lanza una excepción `DataSetException`. Los mensajes de estado de los flujos que se copian se sacan al flujo de impresión `PrintStream` elegido.

**DSX:** El Explorador de `JDataStore` cuenta con una interfaz de usuario que permite copiar flujos en un nuevo archivo `JDataStore`, con estos parámetros. Consulte [Copia de flujos JDataStore](#).

## Asignación y cambio del nombre de los flujos copiados

La barra en los nombres de flujos se utilizan para simular una estructura hierárquica de directorios. El método `copyStreams()` no tiene en cuenta la estructura de directorios. Simplemente trata los nombres como cadenas. Debe utilizar la barra cuando sea necesario imponer una estructura.

Los dos primeros parámetros, `sourcePrefix` y `sourcePattern`, determinan qué flujos se copian. `sourcePrefix` se utiliza en combinación con el parámetro `destPrefix` para asignar un nuevo nombre al flujo copiado, es decir, cambiar el prefijo (el principio) del `storeName` de la copia resultante del flujo.

Si se especifica `sourcePrefix`, el nombre del flujo debe empezar por esta cadena. Normalmente se utiliza para especificar el nombre de un directorio, que termina con una barra. Después se establece para `destPrefix` un nombre distinto, que también termina con una barra. En el nombre de la copia, `sourcePrefix` se sustituye por `destPrefix`. Por ejemplo, supongamos que desea crear en otro directorio una copia del flujo llamado “añadir/crear-tiempo” y asignarle el nombre

“probado/crear-tiempo”. El resultado será crear una copia en un directorio diferente. Se establece `sourcePrefix` en “add/” y `destPrefix` en “tested/”.

Aunque los parámetros de prefijo se utilizan normalmente para los directorios, hay otras formas de cambiar el nombre de los flujos. Por ejemplo, se puede cambiar “hello” por “jello” indicando “h” y “j” como `sourcePrefix` y `destPrefix`, respectivamente. O también puede cambiar “tres/niveles/más” por “no-hacer-más”, introduciendo “tres/niveles/” y “no-hacer-”. Como resultado el flujo se mueve al directorio raíz del `JDataStore`. También se puede hacer lo contrario: ampliar `destPrefix`, con más niveles de directorio que `sourcePrefix`. Por ejemplo, si se deja `sourcePrefix` en blanco pero se termina `destPrefix` con una barra, todos los flujos del archivo `JDataStore` de origen se colocan en un directorio en el archivo de destino.

Si no se desea cambiar el nombre de la copia del flujo no hay motivos para utilizar ningún parámetro de prefijo, por lo que se debe asignar a los dos una cadena vacía o el valor `null`. Tenga en cuenta que si hace una copia de un flujo en el mismo archivo `JDataStore` debe cambiarla de nombre.

El parámetro `sourcePattern` se compara con todo lo que sigue a `sourcePrefix`, con los caracteres estándar habituales “\*” (para ninguno o para varios caracteres) y “?” (para un solo carácter). Si `sourcePrefix` no tiene ningún valor, el modelo se compara con toda la cadena. Si desea copiar todos los flujos en un directorio, se puede colocar su nombre en `sourcePattern`, seguido por una barra y dejar `sourcePrefix` vacío. Por ejemplo, si desea copiarlo todo en el directorio “add” se copia todo lo que empieza por “add/”, por lo que `sourcePattern` debe ser “add/\*”. Esto incluye el contenido de todos los subdirectorios, porque `sourcePattern` coincide con el resto de la cadena. (No existe una forma directa de evitar la copia de flujos en subdirectorios.)

El valor de `sourcePattern` se compara sólo con el nombre de los flujos activos. `copyStreams()` no copia los flujos borrados.

## Demostración de clase: Dup.java

---

Se puede utilizar el siguiente programa, `Dup.java`, para hacer una copia de seguridad de un archivo `DataStore` o actualizar un archivo antiguo:

```
// Dup.java
package dsbasic;

import com.borland.datastore.*;

public class Dup {

    public static void copy( String sourceFile, String destFile ) {
        DataStoreConnection store1 = new DataStoreConnection();
```

```

        DataStore      store2 = new DataStore();

    try {
        store1.setFileName( sourceFile );
        store2.setFileName( destFile );
        if ( !new java.io.File( store2.getFileName() ).exists() ) {
            store2.create();
        } else {
            store2.open();
        }
        store1.open();

        store1.copyStreams( "",      // Desde el directorio raiz
                           "**",    // Todos los flujos
                           store2,
                           "",      // Al directorio raiz
                           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] );
        }
    }
}

```

Este programa copia el contenido de un almacén en otro. Para abrir el archivo `JDataStore` de origen se utiliza un objeto `DataStoreConnection`. Este copia el contenido a un objeto `DataStore` para que el archivo `JDataStore` se pueda crear si es que aún no existe.

Para el método `copyStreams`, `sourcePrefix` y `destPrefix` son cadenas vacías y `sourcePattern` es sólo un asterisco ("`*`"), que lo copia todo sin cambiar el nombre. El programa pasa por alto los errores no recuperables y muestra en la consola mensajes de estado.

Este programa permite combinar el contenido de varios archivos `JDataStore` en uno solo, siempre que no haya flujos con el mismo nombre (`COPY_OVERWRITE` no se especifica como opción).

## Eliminación y recuperación de flujos

---

Borrar flujos es fácil y seguro. Sin embargo, la recuperación de flujos eliminados no siempre funciona y resulta más difícil. Los flujos se borran por nombre. Es fácil entender lo que ocurre cuando se elimina o se intenta recuperar un flujo de archivo, sea un archivo arbitrario o un objeto serializado, porque sólo hay un flujo con ese nombre. Los flujos de tabla a menudo tienen flujos internos adicionales con el mismo nombre como se explica en [Detalles del flujo](#). Estos son un poco más complicados.

### Eliminación de flujos

---

El método `DataStoreConnection.deleteStream()` toma el nombre del flujo que hay que borrar. En el caso de los flujos de archivo, se borra el flujo elegido. Cuando se trata de un flujo de tabla, se borra el flujo principal así como todos sus flujos auxiliares.

Cuando se elimina un flujo no se sustituye ni se borra su contenido. Como ocurre en la mayoría de los sistemas de archivo, el espacio que utiliza el flujo se marca como disponible y la entrada de directorio que apunta a ese espacio se considera borrada. La hora de eliminación se registra. Con el tiempo, los flujos nuevos pueden ocupar el espacio en el que estaba antes el flujo eliminado, por lo que es imposible recuperar su contenido.

**DSX:** Consulte [Eliminación de flujos](#).

### Utilización de los bloques eliminados en JDataStore

---

Los bloques del archivo JDataStore que antes ocupaban los flujos eliminados se vuelven a utilizar en conformidad con las siguientes reglas:

- El archivo JDataStore siempre busca espacio de flujos eliminados antes de asignar más espacio del disco a sus bloques.
- Si el archivo JDataStore es transaccional es necesario enviar la transacción que ha borrado el flujo para poder reutilizar el espacio que ocupa.
- Se utilizan primero los flujos eliminados más antiguos, aquellos que se eliminaron primero.
- Para los flujos de tabla, se sustituyen en primer lugar los flujos auxiliares (índices, totalizadores, etc.).
- El espacio se ocupa desde el principio del flujo hasta el final; es decir, es más difícil recuperar los datos que se encuentran al principio archivo o la tabla, porque es más probable que se hayan borrado.

- Dado que los datos de las tablas se almacenan en bloques, nunca se pierden ni se recuperan trozos de filas, sólo filas completas.
- Cuando todo el espacio de un flujo se ha reasignado, se elimina automáticamente del directorio, ya que es imposible recuperarlo.

## Recuperación de flujos JDataStore

---

Dado que los flujos de tabla tienen varios flujos con el mismo nombre, no basta con este dato sobre el flujo que se desea eliminar. Se debe utilizar una fila del directorio JDataStore, contiene suficiente información para identificar un flujo determinado sin lugar a dudas.

El método `DataStoreConnection.undeleteStream()` toma esta fila como parámetro. Se puede pasar el mismo conjunto de datos del directorio. La fila actual del conjunto de datos del directorio se entiende como fila que hay que recuperar.

Se puede crear un flujo nuevo con el nombre de uno eliminado. No es posible recuperar un flujo eliminado mientras un flujo actual utiliza su nombre.

**DSX:** Consulte [Recuperación de flujos](#).

## Demostración de clase: DeleteTest.java

---

El siguiente programa, `DeleteTest.java`, muestra la eliminación y la no eliminación.

```
// DeleteTest.java
package dsbasic;
import com.borland.datastore.*;
public class DeleteTest {
    public static void main( String[] args ) {
        DataStoreConnection store = new DataStoreConnection();
        com.borland.dx.dataset.StorageDataSet storeDir;
        com.borland.dx.dataset.DataRow locateRow, dirEntry;
        String storeFileName = "Basic.jds";
        String fileToDelete = "add/create-time";

        try {
            store.setFileName( storeFileName );
            store.open();

            storeDir = store.openDirectory();
            locateRow = new com.borland.dx.dataset.DataRow( storeDir,
                new String[] { DataStore.DIR_STATE,
                    DataStore.DIR_STORE_NAME } );
            locateRow.setShort( DataStore.DIR_STATE, DataStore.ACTIVE_STATE );
            locateRow.setString( DataStore.DIR_STORE_NAME, fileToDelete );
            if ( storeDir.locate( locateRow,
                com.borland.dx.dataset.Locate.FIRST ) ) {
```

```
        System.out.println( "Deleting " + fileToDelete );
        dirEntry = new com.borland.dx.dataset.DataRow( storeDir );
        storeDir.copyTo( dirEntry );
        store.closeDirectory();
        System.out.println( "Before delete, fileExists: "
            + store.fileExists( fileToDelete ) );

        store.deleteStream( fileToDelete );
        System.out.println( "After delete, fileExists: "
            + store.fileExists( fileToDelete ) );

        store.undeleteStream( dirEntry );
        System.out.println( "After undelete, fileExists: "
            + store.fileExists( fileToDelete ) );
    } else {
        System.out.println( fileToDelete
            + " not found or already deleted" );
        store.closeDirectory();
    }
} catch ( com.borland.dx.dataset.DataSetException dse ) {
    dse.printStackTrace();
} finally {
    try {
        store.close();
    } catch ( com.borland.dx.dataset.DataSetException dse ) {
        dse.printStackTrace();
    }
}
}
```

En este programa, el nombre del archivo `JDataStore` y el flujo que se van a eliminar se encuentran en el código, algo que, por lo general, no ocurre. El flujo es “añadir/crear-tiempo”, que se ha añadido a `Basic.jds` en el programa de demostración `AddObjects.java`. Se trata de un flujo de archivo sobre todo porque el método `fileExists()` se utiliza para comprobar si la eliminación y la recuperación han funcionado.

## Búsqueda de elementos del directorio

---

El programa empieza por abrir una conexión con el archivo `JDataStore` y abrir su directorio. A continuación, busca el elemento del directorio correspondiente al flujo que se va a eliminar.

**Nota** Generalmente encontrará la entrada de directorio para el flujo después de que se haya borrado y utilizará el conjunto de datos de directorio para deshacer la eliminación del flujo. En esta ocasión se hace de forma diferente para mostrar en concepto de filas de directorio, como se explica en breve.

Para buscar una fila se crea una instancia de `com.borland.dx.dataset.DataRow` desde el conjunto de datos del directorio y se establece que en la búsqueda

se utilizan las dos columnas: State y StoreName. El programa intenta buscar en el directorio el elemento correspondiente al flujo especificado, que debe estar activo. Cuando se busca la fila se coloca el directorio en el elemento deseado y además se indica que el flujo existe y está activo, de forma que el programa puede efectuar el siguiente paso.

## Filas de directorio independientes

---

Cuando se pasa un conjunto de datos de directorio a un método como `undeleteStream()` se utiliza la fila actual. Sin embargo, a causa del orden del directorio `JDataStore` (tal y como se explica en [“Orden de clasificación en directorios” en la página 2-11](#)), cuando se borra un flujo, es probable que desaparezca el elemento que le corresponde en el directorio y se coloque al final, en calidad de último flujo borrado. Por lo tanto, la fila actual apunta ahora a otro objeto (probablemente al siguiente flujo por orden alfabético). Para recuperar el mismo flujo se puede intentar volver a buscar el elemento del directorio del flujo ahora eliminado o copiar los datos del directorio correspondientes al flujo en una fila distinta antes de efectuar la eliminación.

El uso de filas de directorio independientes tiene varias ventajas. A diferencia del conjunto de datos activo del directorio `JDataStore`, las filas independientes son copias estáticas: Son más pequeñas. Después de hacer la copia, se puede cerrar el conjunto de datos de directorio para agilizar las operaciones (para esta sencilla demostración es posible que la pérdida de tiempo que supone crear la fila no justifique la mejora del rendimiento), es posible hacer copias estáticas de todos los elementos del directorio que se desee y gestionarlas como se quiera.

Para crear la fila independiente del directorio se crea otra instancia de `DataRow` desde el conjunto de datos del directorio (para que tenga la misma estructura), y el método `copyTo()` copia los datos de la fila actual. Para demostrar que funciona, el directorio `JDataStore` se cierra.

El flujo de archivo se elimina por su nombre, definido en una cadena al principio del método. Por último, el flujo se recupera por medio del elemento del directorio.

## Empaquetado de archivos `JDataStore`

---

La única forma de reducir el tamaño de un archivo `JDataStore`, eliminando los bloques sin utilizar y los elementos del directorio correspondientes a los flujos eliminados, consiste en copiar los flujos a un nuevo archivo `JDataStore` por medio de `copyStreams()`. Sólo se copian los flujos activos, con lo que se obtiene una versión compacta del archivo.

**DSX:** Consulte [Empaquetado de archivos `JDataStore`](#).





## JDataStore como base de datos incrustada

JDataStore proporciona a las aplicaciones funciones de base de datos incrustada con un solo archivo JDataStore y el controlador JDBC de JDataStore (y las clases que lo aceptan). No se necesita ningún proceso de servidor para las conexiones locales. Además de la aceptación de la norma industrial JDBC, se puede aprovechar el aumento de la comodidad y la flexibilidad que supone el acceso a JDataStore directamente desde la API de DataExpress. Es posible utilizar los dos tipos de acceso en la misma aplicación.

El acceso a JDBC necesita que el JDataStore sea transaccional. Con DataExpress no ocurre lo mismo. Este capítulo se ocupa en primer lugar del acceso por DataExpress, después trata los archivos JDataStore transaccionales y, por último, el controlador JDBC local. El controlador JDBC remoto y el servidor JDataStore se tratan en [Acceso remoto y multiusuario a DataStore](#)

### Utilización de DataExpress para acceder a datos

---

Para utilizar un JDataStore como archivo de base de datos, asocie un componente que se amplía desde `StorageDataSet`, como `TableDataSet`, a un flujo dentro de un JDataStore. El objeto `StorageDataSet` representa una tabla de la base de datos incrustada y proporciona todos los métodos necesarios para desplazarse por los datos: buscarlos, añadirlos, modificarlos y borrarlos.

## Demostración de clase: DxTable.java

---

Inicie un nuevo archivo en el proyecto o paquete `dsbasic` y nómbrelo `DxTable.java`:

```
// DxTable.java
package dsbasic;

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

public class DxTable {

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

    public void demo() {
    try {
        store.setFileName( "Basic.jds" );
        table.setStoreName( "Contabilidad" );
        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();
    }
}
```

Puesto que el programa utiliza DataExpress, importa el paquete DataExpress además del paquete JDataStore. Esta clase tiene dos campos: `DataStoreConnection` y `TableDataSet`. El método `main()` crea una instancia de la clase y ejecuta su método `demo()`.

## Conexión con JDataStore mediante StorageDataSet

---

El punto crucial de la semántica de DataExpress es la clase `StorageDataSet`. Sus tres subclases se utilizan para distintos tipos de fuentes de datos:

- `QueryDataSet` se utiliza para los datos de las consultas SQL.
- `ProcedureDataSet` se utiliza para los datos de los procedimientos SQL almacenados.
- `TableDataSet` no tiene un proveedor de datos predefinido.

Si está definiendo una nueva tabla, utilice `TableDataSet`.

Todos los objetos `StorageDataSet` tienen una propiedad `store`, que adopta el valor **null** cuando se crea una instancia. Si su valor sigue siendo **null** cuando se abre el conjunto de datos se asigna

`com.borland.dx.memorystore.MemoryStore` automáticamente, lo que significa que los datos se almacenan en la memoria. Sin embargo, si se asigna `DataStoreConnection` o `DataStore` a la propiedad `store`, se utiliza el almacenamiento persistente en un archivo `JDataStore`.

Para conectar un `StorageDataSet` a un `JDataStore`, asigne valores a estas tres propiedades:

- La propiedad `fileName` del componente `DataStoreConnection`. Indica con qué archivo `JDataStore` se debe establecer la conexión.
- La propiedad `storeName` del componente `StorageDataSet`. Indica el nombre del flujo de tabla dentro del archivo `JDataStore`. Se puede utilizar un nombre ya existente si no es para el mismo conjunto de datos. De lo contrario, se debe usar un nombre nuevo. (Del usuario depende gestionar el contenido del archivo `DataStore` y elegir los nombres que no creen conflictos).
- La propiedad `store` de `StorageDataSet`, se establece en el objeto `DataStoreConnection` (o `DataStore`). Crea una conexión entre los dos.

Realice estos tres pasos en cualquier orden. Cuando las tres propiedades están establecidas se obtiene una conexión completa entre `StorageDataSet` y `JDataStore`.

En `DxTable.java`, el archivo `JDataStore` es `Basic.jds`, creado en [Creación de archivos JDataStore](#). El flujo de tabla se llama “Contabilidad”. A todos los efectos, éste es el nombre de la tabla. `DxTable.java` asigna

`DataStoreConnection` como el valor de la propiedad `store` de `TableDataSet`.

A continuación se abre el objeto `TableDataSet`. Cuando se abre un conjunto de datos que tiene un archivo `JDataStore` asociado, éste se abre automáticamente. Si el archivo `JDataStore` se abre correctamente, el programa crea el flujo de tabla mencionado en caso de que no exista. En caso de que exista, se limita a abrirlo. Esto establece una conexión abierta entre el conjunto de datos y su flujo de tabla en el archivo `JDataStore`. Si en el archivo `JDataStore` existe un flujo de archivo con el mismo nombre, el programa lanza una excepción, porque no es posible que un flujo de tabla y otro de archivo tengan el mismo nombre.

## Creación de tablas de JDataStore con DataExpress

---

La apertura de un objeto `StorageDataSet` conectado con un archivo `JDataStore` tiene como resultado un flujo de tabla abierto. Para los nuevos flujos de tabla, `QueryDataSet` y `ProcedureDataSet` capturan los datos de su fuente de datos y llenan con ellos el flujo de tabla, como se explica en [Tutorial: Edición fuera de línea con JDataStore](#). Pero `TableDataSet` no tiene fuente de datos por lo que se empieza con un flujo de tabla vacío, sin definir.

Añada las sentencias resaltadas a `DxTable.java`:

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

Se puede comprobar si un flujo de tabla es nuevo comprobando el número de columnas del objeto `TableDataSet`. Si es cero, es posible definir las columnas de la tabla. En este caso, se hace por un método llamado `createTable()`. Añádalo a `DxTable.java`:

```
public void createTable() throws DataSetException {
    table.addColumn( "Nombre" , Variant.STRING );
    table.addColumn( "Nombre" , Variant.STRING );
    table.addColumn( "Actualización", Variant.TIMESTAMP );
    table.addColumn( "Texto" , Variant.INPUTSTREAM );
    table.restructure();
}
```

En este programa de demostración, el método `createTable()` utiliza la forma más sencilla del método `StorageDataSet.addColumn()` para añadir columnas una a una, por nombre y tipo. Las columnas no tienen restricciones. Las columnas de caracteres, definidas como `Variant.STRING`, pueden contener cadenas de cualquier longitud. Para definir columnas con restricciones se deben configurar las propiedades de los objetos `Column`, asignando las propiedades apropiadas como `precision` y añadirlos con los métodos `addColumn()` o `setColumns()` a la tabla.

Después de modificar la estructura de la tabla al añadir estas nuevas columnas, active los cambios con el método `StorageDataSet.restructure()`. El resultado es un flujo de tabla vacío pero estructurado, que constituye una nueva tabla del archivo `JDataStore`. (Si sabe que la tabla no existe, puede utilizar `addColumns()` para definir la estructura antes de abrir el objeto `TableDataSet`, No necesitará llamar a `restructure()`.)

En un archivo `JDataStore` se pueden almacenar tantas tablas como se desee. Los flujos de tabla no pueden tener nombres repetidos. Se puede utilizar el objeto `DataStoreConnection` en la propiedad `store` de los distintos objetos `TableDataSet`.

Existen dos formas más de crear tablas en un archivo `JDataStore`. En concreto, se puede utilizar una sentencia SQL `CREATE TABLE` por medio del controlador `JDBC` de `JDataStore`.

## Utilización de tablas de `JDataStore` con `DataExpress`

---

Cuando las tablas del archivo `JDataStore` están definidas (independientemente de cómo se hayan creado) se puede utilizar el resto de la API de `DataExpress` por medio de un objeto `TableDataSet`, como se haría con cualquier conjunto de datos. Es posible crear filtros, índices, vínculos maestro detalle, etc. De hecho, estos índices secundarios, son persistentes y mantenidos en el archivo `JDataStore`, haciendo de `JDataStore` una base de datos incrustada.

Para terminar el programa de demostración, añade funciones de `DataExpress` con este nuevo método:

```
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( "Nombre", name );
    table.setTimestamp( "Actualización", new java.util.Date().getTime() );
    table.post();
}
```

Añade las sentencias resaltadas al método `demo()`:

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

table.setSort( new SortDescriptor( new String[] { "ID" } ) );
    appendRow( "Temporada de conejos" );
    appendRow( "Temporada de patos" );
table.first();
while ( table.inBounds() ) {
    System.out.println( table.getInt( "ID" ) + ": "
        + table.getString( "Nombre" ) + ", "
        + table.getTimestamp( "Actualización" ) );
    table.next();
}

} catch ( DataSetException dse ) {
```

Después de que el programa abra la tabla, creando la estructura de tabla si fuera necesario, asigna un `SortDescriptor` al campo ID. Para añadir algunas filas, llama al método `appendRow()`.

`appendRow()` empieza por ir a la última fila de la tabla y obtiene el valor del campo ID. A causa del orden de clasificación de la tabla, este valor debe tratarse del número de ID más elevado utilizado hasta el momento. (Si la tabla está vacía, el método `getInt()` devuelve cero). El nuevo valor ID es mayor en un punto que el anterior. `appendRow()` inserta una nueva fila y configura los atributos, incluido el campo Actualizar, que contiene el día y la hora actual. Finalmente `appendRow()` guarda la siguiente fila llamando al método `post()`.

Después de añadir unas líneas al final, la tabla se recorre con un bucle y su contenido se muestra en la consola. Por último, se cierran los objetos `JDataStore` y `TableDataSet`.

Si se ejecuta el programa unas cuantas veces se puede observar que las filas nuevas obtienen números de ID no repetidos. Este método de generación de números ID funciona para programas de demostración de un solo hilo como éste; las filas nuevas siempre se envían después de obtener el antiguo número de ID. Sin embargo, para los programas reales se debe utilizar un método más seguro. Para ello, es necesario comprender los bloqueos y las transacciones.

## JDataStore transaccional

---

Hasta el momento, los cambios efectuados en el archivo `JDataStore` han sido directos e inmediatos. Si se escribe un objeto, se cambian unos cuantos bytes de un flujo de archivo o se añade una fila a una tabla, no se tienen en cuenta las otras conexiones que puedan estar accediendo al mismo flujo. Estos cambios son visibles automáticamente para las otras conexiones.

Aunque este comportamiento es seguro para las aplicaciones sencillas, las más complejas requieren el aislamiento de las transacciones. De este modo, además de evitar que se lean datos modificados o “fantasmas”, se pueden deshacer los cambios efectuados durante una transacción. La aceptación de transacciones permite la recuperación de detenciones automática y es necesaria para el acceso mediante JDBC.

### Activación de admisión de transacciones

---

La clase `com.borland.datastore.TxManager` proporciona la aceptación de transacciones. `JDataStore` puede ser transaccional cuando se crea o se puede añadir más adelante la aceptación de transacciones. En cada caso,

asigna un objeto `TxManager` como el valor de la propiedad `txManager` del objeto `DataStore`, generalmente antes de llamar al método `create()` o `open()`.

Tenga en cuenta los siguientes puntos relativos a la aceptación de transacciones de `JDataStore`:

- Cuando un `JDataStore` deja de ser transaccional, el `TxManager` pierde todas sus propiedades. Si el `JDataStore` se convierte en transaccional de nuevo, las propiedades de `TxManager` tendrán su valor por defecto.
- Si no se ha asignado ningún valor a las propiedades `TxManager.ALogDir` y `TxManager.BLogDir`, se entenderá que los archivos de registro se encuentran en el mismo directorio que el archivo `JDataStore`. Este comportamiento permite mover el archivo `JDataStore` de un directorio a otro sin obtener advertencias de que los archivos de registro ya existen en la ubicación original y en la nueva ubicación.

Si se asigna un valor a la propiedad `txManager` de un archivo `JDataStore` abierto, `TxManager` se cierra inmediatamente e intenta abrir de nuevo el `JDataStore` a fin de que surta efecto la nueva configuración. Si la propiedad `DataStoreConnection.userName` aún no ha recibido ningún valor, el `JDataStore` no podrá volver a abrirse y se producirá una excepción.

Las propiedades del objeto `TxManager` determinan varios aspectos del administrador de transacciones. Cuando se crea una instancia de `TxManager`, éste tiene estas propiedades configuradas con las opciones por defecto. Si desea modificar esta configuración, es mejor hacerlo antes de crear o abrir el archivo `JDataStore`.

La primera vez que se abre el archivo `JDataStore`, que ahora es transaccional, almacena internamente la configuración transaccional. A partir de ahora, no será necesario asignar un `TxManager` cuando se abre el `JDataStore`. El `JDataStore` automáticamente instancia un `TxManager` con las configuraciones almacenadas.

Para abrir o crear un archivo `DataStore` transaccional también hay que asignar valores a la propiedad `DataStoreConnection.userName`. La propiedad `userName` se utiliza para identificar a los usuarios en entornos en los que hay varios, por ejemplo durante la contención de bloqueo. Si no hay ningún nombre adecuado, se puede escribir cualquiera.

## Creación de archivos `JDataStore` transaccionales

Este es el código mínimo para crear un `JDataStore` transaccional sin configuraciones por defecto:

```
DataStore store = new DataStore();

store.setFileName( "AlgúnArchivo.jds" );
store.setUserName( "CualquierNombreFunciona" );
store.setTxManager( new TxManager() );
store.create();
```

Las dos diferencias entre este código y el de los archivos JDataStore no transaccionales estriban en la configuración de las propiedades `userName` y `txManager`. (Se pueden configurar en cualquier orden). Si no se desea utilizar la configuración por defecto, el código podría ser el siguiente:

```

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

// Modificar TxManager
txMan.setRecordStatus( false );

store.setFileName( "AlgunArchivo.jds" );
store.setUserName( "CualquierNombreFunciona" );
store.setTxManager( txMan );
store.create();

```

En este ejemplo, la propiedad `recordStatus`, que controla si se escriben mensajes de estado, tiene el valor **false**.

**DSX:** Consulte [Creación de archivos JDataStore](#).

## Activación de transacciones en archivos JDataStore creados previamente

El código para que un JDataStore sea transaccional es muy similar. La diferencia principal es la llamada a `open()` en vez de a `create()`. Para un TxManager por defecto, el código sería, aproximadamente:

```

DataStore store = new DataStore();

store.setFileName( "AlgunArchivo.jds" );
store.setUserName( "CualquierNombreFunciona" );
store.setTxManager( new TxManager() );
store.open();

```

Aunque es mucho más probable que se utilice un objeto `DataStoreConnection` para abrir un archivo JDataStore, no se puede utilizar para añadir aceptación de transacciones, porque `txManager` es una propiedad de `DataStore`, no de `DataStoreConnection`.

**DSX:** “[Conversión del JDataStore en transaccional](#)” en la página 9-11.

## Apertura de archivos JDataStore transaccionales

La única diferencia entre la apertura de un archivo JDataStore transaccional y otro que no lo es estriba en que se debe establecer `userName`. Dado que no ocurre nada si se especifica `userName` para los archivos no transaccionales (simplemente se pasa por alto), puede ser conveniente establecer siempre `userName` cuando se abre un archivo JDataStore. El código podría ser:

```

DataStoreConnection store = new DataStoreConnection();
store.setFileName( "AlgunArchivo.jds" );

```



```
store.setUserName( "CualquierNombreFunciona" );
store.open();
```

Debido a que no había `TxManager` asignado, cuando el `JDataStore` se abre, un `TxManager` se instancia automáticamente con sus propiedades configuradas a los valores que eran persistentes en `JDataStore`. `TxManager` se asigna a la propiedad `txManager` de `JDataStore`. Aquí es posible obtener los valores de las propiedades persistentes de la administración de transacciones, pero no cambiarlos de forma directa.

## Modificación de la configuración de transacciones

Para cambiar la configuración de transacciones de un archivo `JDataStore`, asígnele un nuevo objeto `TxManager` antes de abrirlo. El objeto `TxManager` sabe qué propiedades se han asignado y cuáles conservan el valor por defecto. Si se asigna un `TxManager` a un archivo `JDataStore` transaccional, sólo se modifican las propiedades asignadas en el nuevo objeto `TxManager`. Todas las demás propiedades conservan la configuración; no adoptan los valores por defecto (procedentes de `TxManager`).

Al añadir compatibilidad con transacciones, deberá asignar a `TxManager` nuevos valores antes de abrir `JDataStore`. Por ejemplo, suponga que desea cambiar la propiedad `softCommit` a **true**. Esto mejora el rendimiento ya que no se garantizan las transacciones enviadas recientemente (aproximadamente en el segundo anterior al fallo del sistema) y sin embargo se garantiza la recuperación de errores:

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

// Modificar TxManager
txMan.setSoftCommit( true );

store.setFileName( "AlgunArchivo.jds" );
store.setUserName( "CualquierNombreFunciona" );
store.setTxManager( txMan );
store.open();
```

Observe que las demás propiedades, como `recordStatus`, no tienen asignados valores. Aunque el nuevo objeto `TxManager` tiene la configuración por defecto cuando se asigna al archivo `JDataStore`, no influye sobre su configuración, aunque no sea la predeterminada.

DSX: Consulte [Modificación de los valores de transacción](#).

## Históricos de transacciones

---

El administrador de transacciones registra los cambios efectuados en el archivo DataStore y los valores anteriores, para que sea posible anular la transacción. (Los cambios no se eliminan del histórico cuando se envían, por lo que si se activan estos históricos es posible extraer un registro de cambios completo o reconstruir el contenido del archivo JDataStore.) La mayoría de las propiedades de `TxManager` controla estos atributos de los históricos de transacciones:

- Si duplicarlos, es decir, si es necesario mantener dos copias independientes pero exactas para aumentar la fiabilidad, a costa de reducir el rendimiento.
- Dónde se deben colocar. Normalmente, cuando se duplican los históricos se guardan en ubicaciones distintas. El hecho de guardarlos en unidades físicas distintas aumenta la fiabilidad y la posibilidad de recuperación, además de paliarse parcialmente el deterioro del rendimiento.
- El tamaño que deben alcanzar antes de iniciar otro archivo.

Por defecto, los históricos sencillos se colocan en el mismo directorio que el archivo JDataStore.

La primera vez que se habilitan las transacciones para un archivo JDataStore, éste crea sus históricos. El nombre de los históricos coincide con el del archivo JDataStore, pero no lleva extensión. Por ejemplo, si el almacén de datos `MyStore.jds` utiliza el registro de transacciones sencillo, se crean estos históricos:

- Archivo de estado `MyStore_STATUS_0000000000`,
- Archivo de ancla `MyStore_LOGA_ANCHOR`
- El archivo de registro `MyStore_LOGA_0000000000`.

La duplicación de registro añade los archivos `MyStore_LOGB_ANCHOR` y `MyStore_LOGB_0000000000`. Estos dos conjuntos de archivos de registro se denominan “A” y “B”. Las propiedades `ALogDir` y `BLogDir` controlan la ubicación de estos archivos.

Cuando un histórico alcanza el tamaño establecido en la propiedad `maxLogSize` de `TxManager`, se crean archivos de estado y de registro adicionales, en los que el número del histórico aumenta en uno cada vez. Dado que los históricos antiguos ya no son necesarios para las transacciones activas ni para la recuperación ante las detenciones del sistema, se borran automáticamente. Para obtener más información sobre la forma de comprimirlos, consulte [“Almacenamiento de históricos” en la página B-2](#).

## Desplazamiento de históricos de transacciones

Si las propiedades `ALogDir` y `BLogDir` no están configuradas, se supone que la ubicación de los archivos históricos es la misma que el directorio del archivo `JDataStore`. Esto hace más fácil mover el `JDataStore` de un directorio a otro. Si se definen las propiedades `ALogDir` y `BLogDir`, estas incluyen la ruta de acceso completa, con la letra de unidad. Esto significa dos cosas:

- Si va a crear archivos `JDataStore` transaccionales antes de moverlos a otro ordenador y configurar las propiedades `ALogDir` y `BLogDir`, deberá crear los archivos `JDataStore` transaccionales en una ubicación con el mismo nombre de unidad y de directorio. Si intenta distribuir los archivos a la unidad D:, pero el archivo `JDataStore` fue creado en C:, porque no tiene una unidad D: en su equipo de desarrollo, debe efectuar los pasos adicionales para trasladar los históricos cuando distribuya la aplicación, porque las unidades son distintas.
- Cuando mueva los archivos históricos, siga estos pasos:
  - 1 Desplace los históricos a la nueva ubicación. No olvide eliminar o cambiar de nombre las copias de los archivos de la ubicación original.
  - 2 Cree un nuevo archivo `TxManager` con la nueva configuración de las propiedades. Asígnelo a la propiedad `txManager` de `DataStore`.
  - 3 Abra el `JDataStore`. `TxManager` busca en la nueva ubicación, comprueba que los históricos se encuentran en ella y cambia la configuración persistente del archivo `JDataStore`.

## Desviación de admisión de transacciones

---

En ocasiones puede ser conveniente acceder a un archivo `JDataStore` transaccional de forma que no se active la aceptación de transacciones. Estos son algunos ejemplos de cuando puede ocurrir esto:

- Se han perdido los históricos de transacciones. Puede abrir normalmente el `JDataStore`.
- Se tiene acceso de sólo lectura a los archivos `JDataStore`. Por ejemplo, pueden estar en un CD-ROM o en un directorio de red para el que no se tenga acceso de escritura.

En cualquiera de los casos, se puede pasar por alto provisionalmente la aceptación de transacciones si se abre el archivo `JDataStore` en modo de sólo lectura. Haga esto con un objeto `DataStore`. Antes de abrir el `JDataStore`, asigne a su propiedad `readOnly` el valor `true`. Por ejemplo:

```
DataStore store = new DataStore();

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

Como se pasa por alto `TxManager`, no es necesario configurar `userName`. Si se pierden los históricos de transacción, utilice el método `copyStreams()` (o el Explorador de `JDataStore`; consulte [Copia de flujos JDataStore](#)) para copiar los flujos a otro archivo.

## Desactivación de la característica de transacciones

---

Haga un `JDataStore` no transaccional asignando un `TxManager` que tenga su propiedad `enabled` con el valor **false**. (El valor por defecto es **true**.) Si la propiedad `consistent` de `DataStore` tiene el valor **false**, el `JDataStore` es internamente inconsistente y no tendrá permiso para realizar el cambio. Como se desactiva `TxManager`, no es necesario configurar `userName`. Así se desactiva esta característica.

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

// Desactivar TxManager
txMan.setEnabled( false );

store.setFileName( "AlgúnArchivo.jds" );
store.setTxManager( txMan );
store.open();
```

Desactivar `TxManager` no borra los archivos de histórico. Al desactivar `TxManager` no se guardan las propiedades de `TxManager`. Si vuelve a hacer transaccional el `JDataStore`, las propiedades `TxManager` recuperan los valores por defecto, por lo que si las propiedades `ALogDir` y `BLogDir` no tenían estos valores, necesitará recordarlas para poder configurarlas.

**DSX:** Consulte [Desactivación de la característica de transacciones](#).

## Eliminación de archivos JDataStore transaccionales

---

Cuando se borra un archivo `JDataStore` transaccional es necesario borrar sus históricos. De lo contrario, no se permite crear un nuevo archivo `JDataStore` con el mismo nombre, porque los históricos no coinciden.

# Control de las transacciones de JDataStore

---

Una vez hecho un JDataStore transaccional, puede generalmente ignorar TxManager. El único momento en el que necesita hacer referencia a TxManager es al querer examinar o cambiar las configuraciones de transacción de JDataStore. La interfaz que controla las transacciones se encuentra en el objeto DataStoreConnection, sobre todo por medio de los métodos commit y rollback.

## Arquitectura de las transacciones

---

Cada objeto DataStoreConnection constituye un contexto de transacciones distinto. Esto significa que todos los cambios efectuados por medio de una DataStoreConnection se consideran un grupo independiente de los cambios efectuados por las otras conexiones.

Una subclase de DataStoreConnection, el objeto DataStore, también se puede emplear con contexto de transacciones independiente. La diferencia consiste en que sólo un objeto DataStore puede acceder a un archivo JDataStore, aunque es posible tener varios objetos DataStoreConnection. Cuando abre una DataStoreConnection, contiene una referencia a un objeto DataStore, como se explica en [“Referencias al archivo JDataStore conectado” en la página 2-15](#). Si no hay un objeto DataStore adecuado en la memoria, DataStoreConnection abre automáticamente un DataStore para cumplir esta referencia. Por tanto, si se abre en primer lugar un objeto DataStoreConnection, la funcionalidad de los objetos DataStore subsiguientes que acceden al mismo archivo JDataStore está reducida, de modo que se comportan como un DataStoreConnection.

La duración de una transacción empieza con cualquier operación de lectura o escritura realizada por medio de una conexión. JDataStore utiliza bloqueos de flujo para gestionar el acceso a los recursos. Para leer o modificar un flujo (cambiando cualquier parte, como un byte de un archivo o una fila de una tabla) es necesario establecer un bloqueo. Una vez establecido, la conexión conserva el bloqueo hasta que la transacción se envía o se cancela.

En las aplicaciones de una sola conexión, las transacciones se pueden considerar sobre todo una función que permite deshacer cambios y permitir la recuperación de detenciones. O debería haber hecho un JDataStore transaccional para que sea accesible a través de un JDBC. Si quiere tener acceso a ese JDataStore con DataExpress, deberá tratar con transacciones. El funcionamiento de las transacciones influye en gran medida sobre las aplicaciones de conexiones múltiples (multiusuario o varias sesiones de un solo usuario). Estos aspectos se tratan en [Cómo evitar bloqueos y conflictos](#) y en el [Capítulo 5, “Acceso remoto y](#)

[multiusuario a DataStore](#)” junto con otros temas relacionados con el aspecto multiusuario.

## Confirmación y cancelación de transacciones

El control de transacciones utiliza estos tres métodos de `DataStoreConnection`:

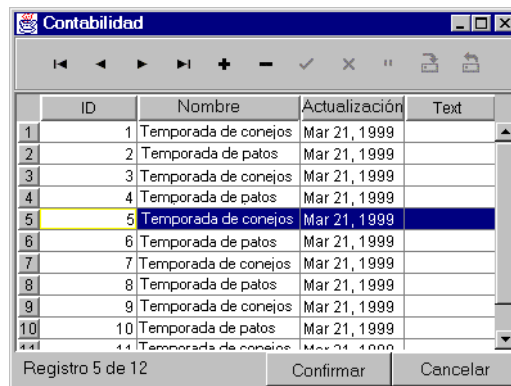
- Para ver si una transacción se ha iniciado, llame a `transactionStarted()`.
- Para enviar una transacción, llame a `commit()`.
- Para cancelar una transacción, llame a `rollback()`.

Cuando cierra `DataStoreConnection`, intenta confirmar cualquier transacción pendiente. Este comportamiento automático se puede controlar esperando a que se produzca el suceso `Response` de `DataStore` para `COMMIT_ON_CLOSE`, como se muestra en el siguiente tutorial.

## Tutorial: Control de transacciones por medio de DataExpress

Este tutorial crea una sencilla aplicación basada en Swing que puede confirmar o cancelar transacciones. También detecta la confirmación automática que se efectúa en el cierre, y permite al usuario decidir cuándo confirmar los datos. Muestra también importantes detalles acerca del uso de un `JDataStore` en una aplicación de interfaz de usuario. El resultado final tiene el siguiente aspecto:

**Figura 3.1** AccountsFrame terminado



## Paso 1: Creación de un archivo JDataStore transaccional con datos de prueba

En este momento ya debe disponer de un archivo JDataStore, `Basic.jds`, con datos. En vez de hacer transaccional este archivo, cópielo y haga transaccional la copia. De este modo es posible utilizar los dos tipos de archivos JDataStore, transaccionales y no transaccionales, para experimentar con ellos.

Haga una copia del archivo y asígnele el nombre `Tx.jds`. A continuación, añada el siguiente programa al proyecto, `MakeTx.java`:

```
// MakeTx.java
package dsbasic;
import com.borland.datastore.*;
public class MakeTx {
    public static void main( String[] args ) {
        if ( args.length > 0 ) {
            DataStore store = new DataStore();
            try {
                store.setFileName( args[0] );
                store.setUserName( "MakeTx" );
                store.setTxManager( new TxManager() );
                store.open();
                store.close();
            } catch ( com.borland.dx.dataset.DataSetException dse ) {
                dse.printStackTrace();
            }
        }
    }
}
```

Este programa de utilidad hace transaccionales los archivos JDataStore que no lo sean ya. Con los archivos JDataStore transaccionales no ocurre nada, porque no se establece ninguna propiedad del objeto `TxManager`.

Establezca los parámetros de ejecución del cuadro de diálogo Propiedades de proyecto en `Tx.jds` y ejecute el programa. Se tarda cierto tiempo en crear los tres históricos de transacción `Tx_STATUS_0000000000`, `Tx_LOGA_ANCHOR` y `Tx_LOGA_0000000000`.

## Paso 2: Creación de un módulo de datos

El siguiente paso consiste en crear un módulo de datos con el archivo JDataStore y un objeto `TableDataSet`:

- 1 Seleccione Archivo | Nuevo.
- 2 Seleccione el módulo de datos en el cuadro de diálogo y pulse Aceptar.
- 3 Compruebe en el Asistente para módulos de datos que el nombre de paquete es `dsbasic` y asigne a la clase el nombre `AccountsDM`. Compruebe

que la casilla de selección Llamar al Modelador de datos no se encuentra activada. Pulse Aceptar.

- 4 Pase a la vista diseño del nuevo archivo `AccountsDM.java`.
- 5 Añada un componente `DataStore` desde la pestaña `DataExpress` al árbol de componentes. Cambie su nombre a `dataStore` (basta con pulsar *F2* después de añadirlo al árbol).
- 6 Asigne a la propiedad `fileName` de `DataStore` el valor `Tx.jds` creado antes, por medio del selector de archivos del Inspector, y asigne un nombre a la propiedad `userName`. (Si no se le ocurre ninguno, escriba `Chuck`).
- 7 Añada un componente `TableDataSet` de la pestaña `Data Express` al árbol de componente.
- 8 En el Inspector, asigne a la propiedad `storeName` el valor `Accounts`, y a la propiedad `store`, el valor `dataStore`.
- 9 Vuelva a la vista de código fuente para observar el código generado.
- 10 Guarde el archivo.

### Paso 3: Creación de una interfaz gráfica de usuario para la tabla JDataStore

Cree una simple rejilla de tabla para mostrar los datos:

- 1 Seleccione Archivo | Nuevo.
- 2 Seleccione la aplicación en el cuadro de diálogo y pulse Aceptar.
- 3 En la primera ficha del Asistente para aplicaciones, introduzca en Nombre de clase el `AccountsApp`. Pulse Siguiente.
- 4 En la segunda ficha del Asistente para aplicaciones, introduzca en Clase de marco `AccountsFrame` y en Título, `Contabilidad`. Compruebe que la casilla de selección `Centrar marco` está activada y las demás están desactivadas. Pulse el botón Finalizar.
- 5 Pase a la vista diseño para el nuevo archivo `AccountsFrame.java`.
- 6 Elija la opción Usar módulo de datos en el menú Asistentes.
- 7 El asistente debe buscar y seleccionar el módulo de datos `AccountsDM`. En Nombre de campo asigne `dataModule`. Seleccione la opción de utilizar una instancia compartida (estática). Pulse Aceptar.
- 8 Añada un componente `JdbNavToolBar` de la pestaña `dbSwing` a la ubicación Norte del marco.
- 9 Añada un componente `JdbStatusLabel` de la pestaña `dbSwing` a la ubicación Sur del marco.
- 10 Añada un componente `TableScrollPane` de la pestaña `dbSwing` a la ubicación Centro del marco.



- 11 Añada un componente `JdbTable` de la pestaña `dbSwing` a `TableScrollPane`.
- 12 En el Inspector, asigne a la propiedad `dataSet` de los tres componentes `jdb` el valor `dataModule.TableDataSet1` (la única opción).
- 13 Vuelva a la vista de código fuente.
- 14 Vaya al método `processWindowEvent()`. Este método se genera de forma que se llame a `System.exit()` cuando se cierre la ventana. Es importante que cierre el `JDataStore` antes de finalizar el programa.

En este caso, con sólo una conexión, el método `close()` funciona, pero como se llama a `System.exit()`, es necesario que se cierre el archivo `JDataStore`, independientemente del número de conexiones que utiliza la aplicación. En este caso se debe utilizar `DataStore.shutdown()`, que cierra directamente el archivo `JDataStore`. Éste es el motivo por el que la aplicación utiliza `DataStore` en lugar de `DataStoreConnection`.

Se puede colocar la llamada al método `shutdown()` justo antes de `System.exit()`, pero por los motivos que se verán más adelante, es conveniente hacerlo antes de que se cierre la ventana. Inserte las sentencias resaltadas:

```
//Se redefine para que pueda llegar a 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 Este código hace referencia al objeto `DataStore` como `dataStore`, que no se ha definido. En primer lugar, añada las siguientes sentencias **import**:

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

- 16 Declare un nuevo campo (cuando el componente se encuentre en una buena ubicación):

```
TableScrollPane tableScrollPane1 = new TableScrollPane();
JdbTable jdbTable1 = new JdbTable();
DataStore dataStore;
```

- 17 Obtenga una referencia al `DataStore` desde el módulo de datos. Añada la sentencia resaltada al método `jdbInit()`:

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

Ejecute `AccountsApp.java`. Puede recorrer la tabla y añadir, modificar y borrar filas. Todos los cambios se efectúan dentro de una sola transacción, aunque aún no se puede observar. Cuando cierra la ventana, se cierra el `JDataStore` y los cambios que hizo se confirman. Esto se puede comprobar ejecutando de nuevo la aplicación.

Si no se ha cerrado el archivo `JDataStore` ni se ha enviado la transacción actual antes de poner fin a la ejecución de aplicación, en el histórico de transacciones aparece una transacción sin enviar, por lo que los cambios se consideran huérfanos y no se escriben en el archivo. Ninguno de los cambios efectuados en la aplicación se aplican. Cuando se cierra el archivo `JDataStore`, estos cambios se confirman de forma automática.

### Paso 4: Adición del control directo de transacciones

En este paso se añade el control directo de la transacción, al permitir que el usuario confirme o cancele la transacción actual:

- 1 Cambie al modo de diseño de `AccountsFrame.java`.
- 2 Borre el objeto `JdbStatusLabel`.
- 3 Añada un componente `JPanel` de la pestaña Contenedores Swing a la ubicación South del marco.
- 4 Asigne a su propiedad `layout` el valor `GridLayout`.
- 5 Añada un componente `JdbStatusLabel` de la pestaña `dbSwing` a `JPanel`.
- 6 Asigne a su propiedad `dataSet` el valor `dataModule.TableDataSet1` (la única opción).
- 7 Añada otro componente `JPanel` de la pestaña Contenedores Swing al primer componente `JPanel`. Aparece a la derecha de `JdbStatusLabel`.
- 8 Asigne a su propiedad `layout` el valor `GridLayout`.
- 9 Añada un componente `JButton` de la pestaña Contenedores Swing al componente anidado `JPanel`.
- 10 Póngale el nombre `commitButton` y asigne a su propiedad `text` el valor `Commit`.
- 11 Añada otro componente `JButton` de la pestaña Contenedores Swing al componente anidado `JPanel`.
- 12 Póngale el nombre `rollbackButton` y asigne a su propiedad `text` el valor `Rollback`.
- 13 Defina el manejador del suceso `actionPerfomed` de `commitButton`:

```

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

```

**14 Defina el manejador del suceso actionPerformed de commitButton de la siguiente forma:**

```

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

```

Estos botones llaman a `commit()` o `rollback()` en el `DataStore` para confirmar o cancelar cualquier cambio que se hizo durante la actual transacción. La transacción actual comprende todo lo ocurrido desde la última confirmación o cancelación.

## Paso 5: Adición del control de confirmación automática

Este último paso permite a la aplicación detectar la confirmación automática cuando se cierra el archivo `JDataStore` y permite al usuario decidir si prefiere confirmar o anular los cambios:

**1 En `AccountsFrame.java`, modifique la definición de clase para que implemente `ResponseListener`:**

```

public class AccountsFrame extends JFrame implements ResponseListener {

```

**2 Añada el método `response` de la interfaz `ResponseListener`:**

```

    public void response( ResponseEvent response ) {
        if ( response.getCode() == ResponseEvent.COMMIT_ON_CLOSE ) {
            if ( JOptionPane.showConfirmDialog( this,
                "No se han confirmado los cambios,. ¿Desea hacerlo ahora?",
                "Confirmar o anular",
                JOptionPane.YES_NO_OPTION ) == JOptionPane.YES_OPTION ) {
                response.ok();
            } else {
                response.cancel();
            }
        }
    }
}

```

Este método busca el suceso `COMMIT_ON_CLOSE`. Cuando esto ocurre aparece un sencillo cuadro de diálogo con las opciones Sí y No para que el usuario confirme si desea los cambios. Si se elige "Sí", se envía la respuesta `ok`, que indica al archivo `JDataStore` que debe confirmar los

cambios. Si se elige “No”, se envía la respuesta `cancel`, que indica al archivo `JDataStore` que debe cancelar los cambios.

Añada las sentencias resaltadas para incluir el marco como uno de los `ResponseListeners` de `JDataStore`.

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

De este modo, se presenta al usuario un cuadro de diálogo si hay cambios sin guardar y se le pregunta si desea confirmarlos. Recuerde que el archivo `JDataStore` se cierra antes que la ventana. De lo contrario, el cuadro de diálogo aparecería cuando la ventana ya no fuera visible.

Ahora se puede ejecutar la aplicación completa. Además de utilizar los botones para confirmar o cancelar los cambios, intente efectuar cambios y cerrar la ventana para practicar con la gestión de confirmación automática.

## Utilización de JDBC para acceder a los datos

---

Es posible acceder a las tablas `JDataStore` por medio del controlador JDBC Tipo 4 de `JDataStore` (100% Java),  
`com.borland.datastore.jdbc.DataStoreDriver`.

Este controlador se puede utilizar para acceso local y remoto. El acceso remoto requiere un servidor `JDataStore`, que se utiliza también para el acceso multiusuario. Si desea conocer más detalles sobre los asuntos relacionados con el acceso remoto y multiusuario, consulte el [Capítulo 5, “Acceso remoto y multiusuario a DataStore”](#).

La dirección URL para conexiones locales es la siguiente:

```
jdbc:borland:dslocal:<nombrearchivo>
```

Como ocurre con cualquier controlador JDBC, se puede utilizar la API de JDBC o una API complementaria como `DataExpress` con `QueryDataSet` y `ProcedureDataSet` para acceder a las tablas.

### Demostración de clase: `JdbcTable.java`

---

El siguiente programa, `JdbcTable.java`, es idéntico a su gemelo `DataExpress`, `DxTable.java`. Utiliza la API de JDBC.

```

// 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, "Contabilidad", null );
        if ( !rs.next() ) {
            createTable();
        }
        appendPStmt = con.prepareStatement("INSERT INTO \"Contabilidad\"
VALUES"
            + "(?, ?, CURRENT_TIMESTAMP, NULL)" );
        getIdPStmt = con.prepareStatement(
            "SELECT MAX(ID)FROM \"Contabilidad\"");
    } catch ( SQLException sqle ) {
        sqle.printStackTrace();
    } catch ( ClassNotFoundException cnfe ) {
        cnfe.printStackTrace();
    }
    }

    public void createTable() throws SQLException {
        stmt.executeUpdate( "CREATE TABLE \"Contabilidad\" ( "
            + "ID INTEGER,"
            + "\"Nombre\" VARCHAR,"
            + "\"Actualización\" TIMESTAMP,"
            + "\"Texto\" 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( "Temporada de conejos" );
        appendRow( "Temporada de patos" );

        rs = stmt.executeQuery( "SELECT * FROM \"Contabilidad\"" );
        while (rs.next()) {
            System.out.println( rs.getInt( "ID" ) + ": "
                                + rs.getString( "Nombre" ) + ", "
                                + rs.getTimestamp( "Actualización" ) );
        }
        stmt.close();
        con.close();
    } catch ( SQLException sqle ) {
        sqle.printStackTrace();
    }
    }

    public static void main( String[] args ) {
        new JdbcTable().demo();
    }
}

```

Esta aplicación JDBC utiliza dos sentencias preparadas: una para añadir filas y la otra para obtener el último valor de ID para ellas. Inicie estas sentencias preparadas antes de llamar al método `appendRow()`. Un buen lugar para hacerlo es el constructor de clases. Como se utiliza el constructor, la organización del código es ligeramente distinta de la de `DxTable.java`.

Lo primero que ocurre en el constructor de la clase es la carga del controlador JDBC de `JDataStore` mediante `Class.forName`. Tanto el nombre del controlador como el principio de la dirección URL de conexión se definen como variables de clase, para su mayor comodidad. Se crea una conexión con `Tx.jds` y, a partir de ella, una sentencia genérica.

El paso siguiente consiste en averiguar si existe la tabla. Puede hacerlo utilizando `DatabaseMetaData.getTables()`. El código pide una lista de tablas llamada “Contabilidad”. Si esa lista esta vacía, significa que no existe la tabla y debe crearla llamando al método `createTable()`. El método `createTable()` utiliza una sentencia SQL `CREATE TABLE`. El analizador sintáctico de SQL suele convertir los identificadores en mayúsculas. Para conservar las mayúsculas y minúsculas que utiliza `DxTable.java`, los identificadores se colocan entre comillas entre ésta y otras sentencias SQL. Finalmente, se crean las dos sentencias preparadas.

El método `demo()` llama a `appendRow()` para añadir un par de filas de prueba. Como ocurre en `DxTable.java`, se recupera el último número de ID, que es el mayor, y se aumenta para la fila siguiente. Sin embargo, en lugar de utilizar una ordenación y acudir a la última fila, JDBC utiliza una

sentencia SQL `SELECT` que recoge el valor máximo. Por tanto, es necesario gestionar expresamente la condición de tabla vacía cuando no existe un último valor.

Por último, el contenido de la tabla recogido mediante una sentencia SQL `SELECT` y un bucle muy similar al de `DxTable.java`. La sentencia y la conexión se cierran correctamente.

Este programa y `DxTable.java` se pueden utilizar indistintamente. Ambos añaden otras dos filas de prueba a la misma tabla.

## Control de transacciones por medio de JDBC

---

Cada conexión JDBC utiliza su propia instancia `DataStoreConnection`. Así se implementa el controlador JDBC de `JDataStore`. Sin embargo, no es posible acceder a este objeto interno, se debe utilizar la API de JDBC.

Para controlar las transacciones, desactive el modo de confirmación automática llamando a `Connection.setAutoCommit(false)`. Puede llamar a `commit()` y `rollback()` en el objeto `Connection`.





# Utilización de las características de seguridad de JDataStore

JDataStore incorpora varias características de seguridad. Entre ellas se incluye la autenticación y autorización de usuarios, y la encriptación de bases de datos JDataStore.

## Autenticación de usuario

---

Por defecto, las bases de datos JDataStore no exigen autenticación de usuario para acceder a la base de datos. La autenticación de un JDataStore se activa añadiendo al menos un usuario a la tabla de sistema `SYS/USERS` en una base de datos JDataStore. Esto puede llevarse a cabo desde código o desde el Explorador de JDataStore.

La tabla `SYS/USERS` es de sólo lectura si se accede a ella mediante una consulta JDBC o un `StorageDataSet`.

Cuando se añade un usuario, se proporciona la contraseña inicial y sus derechos de acceso a la base de datos.

Existen tres métodos para gestionar usuarios. Los derechos de `DataStoreRights.ADMINISTRATOR` son necesarios para llamar a estos métodos.

- Los derechos de un usuario pueden modificarse llamando al método `DataStoreConnection.changeRights()`.
- Los usuarios pueden eliminarse llamando al método `DataStoreConnection.removeUser()`.
- Se pueden añadir usuarios con el método `DataStoreConnection.addUser()`.

**DSX:** Consulte [Gestión de usuarios](#) para una explicación acerca de cómo administrar usuarios con el explorador de JDataStore.

El método `DataStoreConnection.changePassword()` se puede utilizar para cambiar la contraseña. Cualquier usuario puede cambiar su propia contraseña. Aunque esta acción requiere conocer la contraseña actual, no requiere derechos `DataStoreRights.ADMINISTRATOR`.

**DSX:** Para obtener instrucciones acerca de cómo cambiar una contraseña a través del Explorador de JDataStore, consulte [Cambio de la contraseña](#).

## Autorización

---

Los derechos de una base de datos dependen de la especificación de constantes en la interfaz `com.borland.datastore.DataStoreRights`. Los derechos indicados por `DataStoreRights` incluyen:

- **STARTUP**—la facultad de abrir una base de datos. Se requiere una contraseña para añadir los derechos **STARTUP** a un usuario. El parámetro `pass` del método `DataStoreConnection.changeRights()`, no puede tener el valor null, y debe coincidir con la contraseña del usuario al llamar a este método para añadir derechos **STARTUP**. Los derechos **STARTUP** pueden indicarse en el momento de la adición del usuario.
- **ADMINISTRATOR**—incluye los derechos para añadir, borrar y cambiar los derechos de los usuarios, así como la posibilidad de cifrar la base de datos. también incluye los cuatro derechos de flujos (**WRITE**, **CREATE**, **DROP**, **RENAME**). Los derechos **STARTUP** se conceden por defecto a los administradores cuando se añaden, pero los derechos **STARTUP** pueden ser eliminados. Los derechos **WRITE**, **CREATE**, **DROP**, y **RENAME** no se le pueden quitar a un administrador. Cualquier intento de retirar dichos permisos a un administrador será obviado.
- **WRITE**—la facultad de escribir en los archivos o flujos de tabla en el JDataStore.
- **CREATE**—la facultad de crear archivos o flujos de tabla en el JDataStore.
- **DROP**—la facultad de borrar archivos o flujos de tabla desde JDataStore.
- **RENAME**—facultad de renombrar archivos o flujos de tabla en el JDataStore.

## Encriptación en JDataStore

---

Un usuario `DataStoreRights.ADMINISTRATOR` puede cifrar una base de datos vacía que tenga versión `DataStore.VERSION_6_0` de `DataStoreConnection.getVersion()` o posterior. El método `DataStoreConnection.encrypt()` puede utilizarse para cifrar bases de datos.

`DataStoreConnection.encrypt()` borrará `DataStoreRights.STARTUP` de todos los usuarios exceptuando al administrador que está añadiendo el cifrado. Esto se debe a que se necesita la contraseña de usuario para añadir derechos `STARTUP` a un usuario. Para proporcionar derechos `STARTUP` a un usuario, llame a `DataStoreConnection.changeRights()` o elimínelo y vuelva a añadirlo.

**Nota:** No se encriptará una base de datos que cuente con flujos de tabla o de archivo. Si desea cifrar una base de datos existente, cree una nueva base de datos, llame a `DataStoreConnection.copyUsers()` para copiar los usuarios existentes a la nueva base de datos, continúe y cifre la nueva base de datos. Después, debe llamar al método `DataStoreConnection.copyStreams()` para copiar el contenido de la anterior base de datos a la encriptada. Para obtener más información acerca de como copiar flujos, consulte [Copia de flujos JDataStore](#).

**DSX:** Consulte [Encriptación de un JDataStore](#) para obtener información acerca de como cifrar un JDataStore utilizando el explorador de JDataStore.

## Cómo aplicar la seguridad de JDataStore

---

En este caso un *adversario* es alguien que intenta romper el sistema de seguridad de JDataStore.

Las funciones de autenticación y autorización son seguras en las aplicaciones del servidor en las que el adversario no tiene acceso a los archivos físicos de JDataStore. La tabla `SYS/USERS` almacena contraseñas, ID de usuario y derechos de manera encriptada. También almacena el ID y los derechos de usuario en una columna no encriptada, pero sólo para su presentación. Los valores encriptados del ID y los derechos de usuario se utilizan por motivos de seguridad.

Las contraseñas almacenadas están cifradas utilizando un algoritmo *TwoFish*. Se utiliza un generador de números pseudo-aleatorio para asignar valores ficticios a la encriptación de la contraseña. Esto hace que los ataques de tipo *diccionario* para encontrar la contraseña sean mucho más difíciles. En un ataque de diccionario, el adversario realiza intentos de adivinar la contraseña hasta que acierta. Este proceso resulta más sencillo si el adversario cuenta con información personal sobre el usuario y si el usuario ha elegido una contraseña obvia. No hay mejor defensa contra los ataques de diccionario que una buena contraseña (ininteligible). Cuando se introduce una contraseña errónea, el hilo actual pasa a estado inactivo durante 500 milisegundos.

Si la base de datos de un JDataStore no está encriptada es importante restringir el acceso físico al archivo debido a las siguientes razones:

- Si un adversario puede escribir en un archivo de base de datos de un JDataStore no protegido con contraseña mediante un programa o utilidad de edición, es posible desactivar las opciones de autenticación y autorización.
- Si un adversario puede leer un archivo de base de datos de un JDataStore no encriptado mediante un programa o utilidad de edición, es posible que el adversario pueda hacer ingeniería inversa sobre el archivo y ver su contenido.

Para entornos en los que un adversario peligroso puede acceder a copias físicas de los JDataStore, la base de datos y los archivos históricos deberían estar cifrados, además de protegidos con contraseña.

**Advertencia:** Las técnicas criptográficas que utiliza un JDataStore para encriptar bloques de datos son de última generación. El algoritmo TwoFish que utiliza un JDataStore nunca ha sido descifrado. Esto quiere decir que si olvida la contraseña de una base de datos JDataStore encriptada, tiene problemas. La única posibilidad que tiene de recuperar los datos es que alguien adivine la contraseña.

Se pueden tomar algunas medidas para no olvidar la contraseña de una base de datos encriptada. Es importante recordar que se utiliza una contraseña maestra de manera interna para encriptar los bloques de datos. Cualquier usuario que tiene derechos `STARTUP` tendrá la contraseña maestra encriptada utilizando su contraseña en la tabla `SYS/USERS`. Esto permite que más de un usuario pueda abrir una base de datos, gracias a que pueden desencriptar una copia de la contraseña maestra con su contraseña. Esta característica se puede utilizar para crear una base de datos virgen que cuenta con un usuario secreto con derechos `DataStoreRights.ADMINISTRATOR` (que incluyen derechos `DataStoreRights.STARTUP rights`). Si se utiliza esta base de datos virgen cada vez que se precisa una nueva base de datos vacía, siempre habrá un usuario secreto que pueda desbloquear la encriptación.

La encriptación de una base de datos afecta ligeramente el rendimiento. Los bloques de datos se encriptan cuando se escriben del caché de JDataStore al archivo de base de datos JDataStore. Los bloques de datos se desencriptan cuando se leen desde el archivo de base de datos JDataStore al caché JDataStore. Por tanto, sólo se aprecia la reducción de rendimiento por la encriptación cuando se realizan acciones de E/S.

JDataStore encripta todo los bloques de datos al completo de los archivo `.jds`, excepto los primeros 16 bytes de cada uno de ellos. Los primeros 16 bytes de un bloque de datos no contienen información de usuario. Algunos bloques no se encriptan. Por ejemplo, los bloques de mapas de bits de asignación, el bloque de cabecera, los bloques de ancla del histórico y los bloques de tabla `SYS/USERS`. Los campos con información importante de la tabla `SYS/USERS` se encriptan mediante la contraseña del usuario. Los bloques de archivo del histórico se encriptan por completo. Los archivos

de ancla del histórico y de estado del histórico no se encriptan. La base de datos temporal que utiliza el motor de consultas se encripta. Los archivos de clasificación utilizados en la clasificación de fusiones de gran envergadura no se encriptan, pero se eliminan una vez finaliza la clasificación.

Observe que el controlador remoto de JDBC para JDataStore utiliza actualmente las clases socket de Java para comunicar con el servidor JDataStore. Esta comunicación no es segura. Mientras que el controlador de JDBC local de JDataStore está en ejecución, es segura.



## Acceso remoto y multiusuario a DataStore

Las aplicaciones JDataStore no están limitadas al acceso a archivos locales de JDataStore. Es posible acceder a JDataStore de otros ordenadores por medio de un servidor JDataStore. Este servidor permite también el acceso multiusuario. El proceso del servidor gestiona las solicitudes de acceso de los clientes.

### Realización de una conexión con un JDBC local

---

La conexión con un JDBC local permite que la aplicación se ejecute el mismo proceso que el motor de JDataStore. Las aplicaciones que realizan un gran número de llamadas a métodos en la API de JDBC tendrán una ventaja significativa en el desarrollo mediante el controlador local de JDataStore.

El siguiente código proporciona un ejemplo sencillo de cómo establecer una conexión con un JDBC local:

```
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");
```

## Especificación de propiedades ampliadas

---

La documentación API de la interfaz `com.borland.datastore.jdbc.cons.ExtendedProperties` documenta cómo especificar las propiedades ampliadas y la colección más reciente de propiedades ampliadas que pueden configurarse con una conexión JDBC.

## Controlador JDBC para acceso remoto

---

Es posible utilizar el controlador JDBC Tipo 4 de `JDataStore` (100% Java), `com.borland.datastore.jdbc.DataStoreDriver` para acceder a archivos `JDataStore` remotos y locales. La dirección URL para conexiones locales es:

```
jdbc:borland:dslocal:<nombrearchivo>
```

La dirección URL para conexiones remotas es:

```
jdbc:borland:dsremote://<nombreanfitrión>/<nombrearchivo>
```

Tenga en cuenta que en Unix los nombres de archivo con referencia a la vía de acceso empiezan con una barra inclinada, por lo que las URL de estos archivos tienen dos barras entre el nombre de anfitrión y el del archivo.

Es necesario que se esté ejecutando un proceso de servidor `JDataStore` en el ordenador <anfitrión>. Las comunicaciones entre la aplicación cliente y el servidor `JDataStore` utilizan por defecto el puerto 2508. Puede cambiar el número de puerto cuando inicie o reinicie el servidor. Se puede cambiar en el cliente si cuenta con una conexión con propiedades ampliadas. Por ejemplo, si desea acceder al archivo `JDataStore` `c:\someApp\ecom.jds` del sistema `mobile.mycompany.com` a través del puerto 9876 puede hacer algo como:

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

Encontrará más información sobre las propiedades de conexión en “Driver properties” del componente `DataStoreDriver`, en la *DataExpress Component Library Reference*.

Al margen de estas diferencias, las conexiones remotas por JDBC funcionan, desde la perspectiva del cliente, de forma parecida a las conexiones locales de JDBC. Para obtener más información, consulte [Utilización de JDBC para acceder a los datos](#).



Dado que los almacenes de datos a los que se accede por conexión remota y local pueden estar a disposición de varios usuarios, se deben considerar posibles conflictos, como se explica en [Cómo evitar bloqueos y conflictos](#).

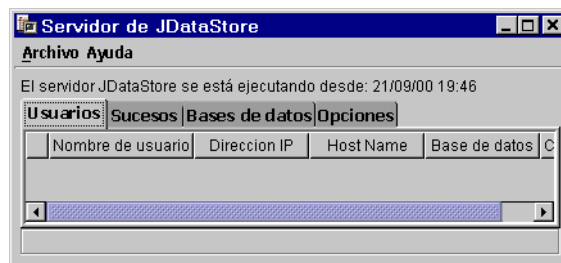
## Ejecución del Servidor de JDataStore

---

Para acceder a archivos de JDataStore por medio de una conexión JDBC remota es necesario que un servidor de JDataStore se esté ejecutando en un ordenador con acceso local a estos archivos. Este ordenador puede ser el que contiene los archivos en realidad o puede tener acceso directo por red a los archivos de JDataStore de otra unidad.

Para el desarrollo, el servidor se puede iniciar desde el menú de JBuilder. Seleccione Herramientas | Servidor de JDataStore. Esto inicia el proceso del servidor y muestra una sencilla interfaz de usuario:

**Figura 5.1** Servidor de JDataStore



Desde el menú puede obtener ayuda, finalizar la conexión con el servidor y ejecutar el Explorador de JDataStore (si desea obtener más información sobre el Explorador de JDataStore, consulte el [El Explorador de JDataStore](#)). En la pestaña Opciones se pueden modificar algunos elementos de la configuración del servidor.

## Cambio de configuración del servidor

---

No es posible modificar la configuración del servidor mientras se ejecuta. El comando de menú Archivo | Detener detiene el proceso actual del servidor y cierra todas las conexiones ordenadamente.

Una vez detenido el servidor, puede utilizar la pestaña Opciones para cambiar el número de puerto, el directorio temporal y la configuración de estado del directorio del histórico del servidor. A continuación, seleccione Iniciar en el menú Archivo para reiniciar el proceso del servidor.

## Distribución del servidor de JDataStore

---

Cuando termina la fase de desarrollo de una aplicación y comienza la fase de producción, debe distribuir el Servidor de JDataStore a una máquina servidor.

### Empaquetado del servidor

---

Los archivos JAR necesarios para ejecutar el servidor dependen de si se desea una interfaz gráfica de usuario. Sin la interfaz gráfica sólo se necesitan los siguientes archivos JAR:

- `jdserver.jar`
- `jds.jar`
- `dx.jar`

Si quiere la GUI, necesita estos archivos JAR:

- `dbswing.jar`
- `dbtools.jar`

Si necesita ayuda en línea para la GUI, necesita también este archivo JAR:

- `help.jar`

Si desea que la ayuda en pantalla del servidor esté disponible también deberá copiar los archivos JAR de ayuda del directorio `/doc`. Los archivos de ayuda para el servidor de JDataStore se encuentran en `jb_ui.jar`. La *Guía del desarrollador de JDataStore* está en `jb_dspg.jar`.

### Inicio del servidor

---

Para iniciar el servidor es necesario añadir los archivos JAR necesarios a la vía de acceso a clases. La clase principal del servidor JDataStore es `com.borland.dbtools.dserver.Server`, por lo que la línea de comandos para iniciar el servidor con las opciones por defecto es:

```
java com.borland.dbtools.dserver.Server
```

También se pueden especificar las opciones que se enumeran en la tabla siguiente:

**Tabla 5.1** Opciones de inicio del servidor de JDataStore

Opción	Descripción
-port=<número>	Puerto por el que se espera la conexión. Por defecto: 2508
-ui=<tipo de interfaz de usuario>	Aspecto de la interfaz de usuario. Uno de los siguientes: <ul style="list-style-type: none"> <li>• windows</li> <li>• motif</li> <li>• metal</li> <li>• none</li> <li>• &lt;nombre de clase LookAndFeel&gt;</li> </ul>
-temp=<nombre de directorio>	Directorio que se debe utilizar para todos los archivos temporales.
-doc=<directorio de ayuda>	El directorio que contiene los archivos de ayuda en pantalla.
-?	Muestra un mensaje que enumera estas opciones.
-help	

Si no se utiliza la opción `-ui` (o si se elige `none`), el servidor se inicia como aplicación de consola. Sin la interfaz de usuario no es posible volver a configurar el servidor ni abrir el Explorador de JDataStore. Para detener el servidor utilice la acción correspondiente del sistema operativo o el shell. Por ejemplo, cuando se esté ejecutando en primer plano, pulse `Ctrl+C`. Cuando se ejecute en segundo plano en Unix, utilice el comando `kill`.

Está también disponible un archivo ejecutable para el servidor, y puede iniciarse desde la línea de comandos. El ejecutable utiliza las configuraciones de la vía de acceso a clases y la clase principal en el archivo `dss.config`.

## Creación de servidores JDBC personalizados

El servidor JDataStore que acompaña a JBuilder proporciona acceso remoto a los archivos JDataStore. Es posible crear servidores personalizados con funciones adicionales. Por ejemplo, dado que es posible que el servidor se ejecute ininterrumpidamente, se puede añadir un hilo de mantenimiento que efectúe una copia de seguridad de los archivos todas las noches a la misma hora. Otro ejemplo podría ser añadir la posibilidad de eliminar flujos de archivo en un JDataStore. No es posible acceder a los flujos de archivo por medio de JDBC.

El centro del servidor JDataStore es la clase `com.borland.datastore.jdbc.DataStoreServer`. Si desea más información, examine el paquete `datastore.jdbc` de la *DataExpress Component Library Reference*.



# Transacciones y agrupación de conexiones

## Problemas de las transacciones multiusuario

---

El acceso de varios usuarios puede provocar distintos problemas en las transacciones, desde la reducción del rendimiento hasta los conflictos. (Estos problemas pueden presentarse también en aplicaciones complejas de un solo usuario que utilizan varias conexiones de transacción). Para evitar o reducir al mínimo estos problemas es necesario comprender los mecanismos de transacción que utiliza JDataStore.

### Niveles de aislamiento de transacciones

---

JDataStore admite los cuatro niveles de aislamiento especificados por los estándares JDBC y ANSI/ISO SQL (SQL/92).

El nivel de aislamiento serializable designado por `java.sql.Connection.TRANSACTION_SERIALIZABLE` proporciona un aislamiento completo para una transacción. Una aplicación puede elegir un nivel de aislamiento más débil para mejorar el rendimiento o evitar conflictos del gestor de bloqueos. Los niveles de aislamiento más débiles pueden sufrir las siguientes vulneraciones del aislamiento:

- Lecturas de datos no confirmados, en las que una conexión puede leer los datos escritos por otra conexión antes de que se hayan confirmado.
- Lecturas no repetibles, en las que una conexión lee una fila confirmada, otra conexión cambia y confirma la fila y la primera conexión obtiene un valor distinto cuando vuelve a leerla.

- Lecturas fantasma, en las que una conexión lee todas las filas que cumplen la condición WHERE, una segunda conexión añade otra fila que también cumple esta condición y la primera conexión, al volver a efectuar la lectura, ve la nueva fila que no estaba la primera vez.

SQL92 define cuatro niveles de aislamiento en términos del comportamiento que una transacción que se ejecuta en un determinado nivel de aislamiento puede experimentar, tal como se muestra en la siguiente tabla.

**Tabla 6.1** Definiciones de niveles de aislamiento en SQL

Nivel de aislamiento	Lectura de datos no confirmados	Lectura no repetible	Lectura fantasma
Lectura no confirmada	Posible	Posible	Posible
Lectura confirmada	No es posible	Posible	Posible
Lectura repetible	No es posible	No es posible	Posible
Serializable	No es posible	No es posible	No es posible

## Configuración de niveles de aislamiento en conexiones JDataStore

Para configurar los niveles de aislamiento en conexiones JDataStore:

- Utilice `java.sql.Connection.setTransactionIsolation(int level)` para especificar el nivel de aislamiento en conexiones JDBC de JDataStore.
- Utilice `com.borland.datastore.DataStoreConnection.setTxIsolation(int level)` para conexiones DataExpress de JDataStore.

Con ambos métodos, el parámetro `nivel` puede adoptar uno de los siguientes cuatro valores:

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

Para elegir un nivel de aislamiento, consulte la tabla siguiente.

**Tabla 6.2** Los niveles de aislamiento de una conexión JDBC

Nivel de aislamiento	Descripción
TRANSACTION_READ_UNCOMMITTED	Apropiada para aplicaciones utilizadas por un único usuario con el fin de realizar informes que no exigen vistas transaccionalmente incoherentes de los datos. Especialmente útil para examinar tablas JDBC con dbswing y componentes DataSet de DataExpress. Este nivel de aislamiento conlleva una mínima sobrecarga de datos adicionales para gestionar bloqueos.
TRANSACTION_READ_COMMITTED	Se utiliza habitualmente para aplicaciones de alto rendimiento. Ideal para modelos de acceso a datos que utilizan un control de concurrencia optimista. QueryDataSet de DataExpress y Borland Application Server utilizan enfoques de control de concurrencia optimista para el acceso a datos. En estos modelos de acceso a datos, las operaciones de lectura se realizan, en gran medida, en primer lugar. En algunos casos, las operaciones de lectura realmente se ejecutan en una transacción independiente de las operaciones de escritura.
TRANSACTION_REPEATABLE_READ	Proporciona una mayor protección para acceso a datos transaccionalmente coherentes sin la concurrencia reducida de TRANSACTION_SERIALIZABLE. No obstante, este nivel de aislamiento produce una mayor sobrecarga de datos adicionales para gestionar bloqueos, ya que los bloqueos de filas deben obtenerse y mantenerse durante toda la transacción.
TRANSACTION_SERIALIZABLE	Proporciona una completa capacidad de serialización para las transacciones con los inconvenientes de una menor concurrencia y una mayor posibilidad de conflictos. Aunque con este nivel de aislamiento aún es posible utilizar los bloqueos de filas para las operaciones habituales, algunas operaciones hacen que el gestor de bloqueos de JDBC tenga que utilizar bloqueos de tabla.

## Bloqueos utilizados por el gestor de bloqueos de JDBC

La tabla siguiente describe los bloqueos utilizados por el administrador de bloqueos de JDBC:

**Tabla 6.3** Bloqueos utilizados por el gestor de bloqueos de JDBC

Bloqueo	Descripción
Bloqueos de sección crítica	Bloqueos internos utilizados para proteger estructuras de datos internas. Estos bloqueos se mantienen, por lo general, durante un corto período de tiempo. Se adquieren y liberan independientemente de si la transacción está o no confirmada.
Bloqueos de fila	Admiten modos de bloqueo exclusivo y compartido. Estos bloqueos se liberan al confirmar la transacción.

**Tabla 6.3** Bloqueos utilizados por el gestor de bloqueos de JDataStore (continuación)

Bloqueo	Descripción
Bloqueos de tabla	Admiten modos de bloqueo exclusivo y compartido. Estos bloqueos se liberan al confirmar la transacción.
Bloqueos de tabla DDL	Admiten modos de bloqueo exclusivo y compartido: <ul style="list-style-type: none"> <li>Los bloqueos DDL compartidos se utilizan en transacciones que tienen tablas abiertas. Estos bloqueos no se liberan hasta que, en primer lugar, la transacción se confirma, y, en segundo lugar, la conexión cierra la tabla y todas las sentencias que hacen referencia a la tabla.</li> <li>Los bloqueos DDL exclusivos se utilizan para eliminar tablas o modificar su estructura, y se liberan al confirmar la transacción.</li> </ul>

## Propiedades JDBC ampliadas que controlan el comportamiento de bloqueo de JDataStore

Las propiedades extendidas de JDBC se especifican asignándolas a un objeto `java.util.Properties` que se pasa al crear una conexión JDBC.

Los nombres de las propiedades distinguen entre mayúsculas y minúsculas y se describen en la siguiente tabla.

**Tabla 6.4** Propiedades JDBC ampliadas que controlan el comportamiento de bloqueo

Propiedad	Comportamiento
<code>tableLockTables</code>	Cadena de nombres de tablas separados por punto y coma. Los nombres de tabla distinguen entre mayúsculas y minúsculas. El bloqueo de fila no se utiliza para las tablas especificadas en esta lista. Para especificar todas las tablas, asigne a esta propiedad el valor <code>""</code> . Esta propiedad también se puede utilizar para componentes <code>DataStoreConnection</code> mediante una llamada al método <code>setTableLockTables()</code> .
<code>maxRowLocks</code>	Número máximo de bloqueos de fila por tabla que una transacción debería obtener antes de pasar a un bloqueo de tabla. El valor por defecto es 50. Esta propiedad también se puede utilizar con componentes <code>DataStoreConnection</code> mediante una llamada al método <code>setMaxRowLocks()</code> .
<code>lockWaitTime</code>	Número máximo de milisegundos que se debe esperar para que un bloqueo sea liberado por otra transacción. Cuando este tiempo de espera termina, se lanza la excepción apropiada. Esta propiedad también se puede utilizar con componentes <code>DataStoreConnection</code> mediante una llamada al método <code>setLockWaitTime()</code> .
<code>readOnlyTxDelay</code>	Número máximo de milisegundos que se debe esperar antes de iniciar una nueva vista de sólo lectura de la base de datos. Para obtener más información, consulte la información sobre la propiedad <code>java.sql.Connection.readOnly()</code> en el apartado <a href="#">Optimización de aplicaciones JDataStore</a> .



## Utilización de bloqueos en JDataStore y niveles de aislamiento

La utilización de bloqueos de tabla y bloqueos de fila varía entre los diversos niveles de aislamiento. La propiedad de conexión `tableLockTables` afecta a todos los niveles de aislamiento (desactiva el bloqueo de fila). Los bloqueos de sección crítica y DDL se aplican de la misma forma para todos los niveles de aislamiento.

Todos los niveles de aislamiento obtienen, al menos, un bloqueo de fila exclusivo para las operaciones de actualización, eliminación o inserción de filas. En algunos casos de aumento del nivel de bloqueo, en su lugar se puede utilizar un bloqueo de tabla exclusivo.

La siguiente tabla describe el comportamiento del bloqueo de fila para los niveles de aislamiento de la conexión de JDataStore:

**Tabla 6.5** Utilización de bloqueos y niveles de aislamiento

Nivel de aislamiento	Comportamiento del bloqueo de fila
TRANSACTION_READ_UNCOMMITTED	No obtiene bloqueos de filas para operaciones de lectura. Además, pasa por alto los bloqueos de fila exclusivos utilizados por otras conexiones que han insertado o actualizado una fila.
TRANSACTION_READ_COMMITTED	No obtiene bloqueos de filas para operaciones de lectura. Una transacción que utilice este nivel de aislamiento se bloqueará cuando lea una fila sobre la que otra transacción posee un bloqueo de fila exclusivo debido a una operación de inserción o actualización. Este bloqueo terminará cuando la transacción de escritura se confirme, se detecte un conflicto o el límite de tiempo de <code>lockTimeOut</code> haya expirado.
TRANSACTION_REPEATABLE_READ	Obtiene bloqueos de fila compartidos para operaciones de lectura.
TRANSACTION_SERIALIZABLE	Obtiene bloqueos de fila compartidos para consultas que seleccionan una fila según un conjunto de valores de columna unívocos tales como una clave principal o la columna <code>INTERNALROW</code> . En SQL, la cláusula <code>where</code> determina si se seleccionan o no valores de columna unívocos. Los bloqueos de fila exclusivos también se utilizan para operaciones de inserción, actualización y eliminación en filas identificadas por un conjunto unívoco de valores de columna. Las operaciones de lectura que no se seleccionan según un conjunto unívoco de valores de columna utilizarán un bloqueo de tabla compartido. Las operaciones de lectura que no consigan encontrar ninguna fila utilizarán un bloqueo de tabla compartido. Las operaciones de inserción y actualización ejecutadas sobre una fila especificada de forma no unívoca utilizarán un bloqueo de tabla exclusivo.

Observe que el aumento del nivel de bloqueo desde los bloqueos de fila hasta los bloqueos de tabla se produce en algunas situaciones de `TRANSACTION_SERIALIZABLE` según se ha descrito anteriormente, pero también ocurre en todos los niveles de aislamiento si se supera el valor de la propiedad `maxRowLocks`.

## Depuración de problemas de expiración del tiempo de espera para bloqueos y conflictos

---

Los bloqueos pueden fallar debido a situaciones de conflicto o de expiración del tiempo de espera para un bloqueo. La expiración del tiempo de espera para un bloqueo se produce cuando una conexión espera para obtener un bloqueo utilizado por otra transacción durante un tiempo superior al expresado (en milisegundos) en la propiedad `lockWaitTime`. En esos casos, se lanza una excepción que identifica cuál es la conexión para la que expiró el tiempo de espera y cuál es la que está utilizando actualmente el bloqueo que se necesita. La transacción para la que ha expirado el tiempo de espera para el bloqueo no se cancela.

`JDataStore` dispone de una detección automática de alta velocidad de conflictos que debería ser capaz de detectar cualquier conflicto. Es posible, aunque improbable, que una situación de conflicto se notifique como una situación de `locktimeout` (expiración del tiempo de espera para un bloqueo). Se lanza una excepción apropiada que identifica qué conexión generó el conflicto y con qué conexión se encuentra en conflicto. A diferencia de las excepciones de expiración del tiempo de espera para bloqueo, las excepciones de conflicto detectadas por una conexión `java.sql.Connection` hacen que esa conexión cancele automáticamente su transacción. Este comportamiento permite que otras conexiones puedan continuar su trabajo.

Utilice las siguientes directrices para detectar situaciones de expiración de tiempo de espera y conflictos:

- Examine el mensaje que ofrecen estas excepciones. Contiene información sobre cuál era la tabla donde se produjo el error y cuáles eran las conexiones implicadas.
- Asigne a la propiedad `java.sql.DriverManager.SetLogWriter()` un flujo del escritor de registros. Para limitar los resultados del registro a sólo la información relativa al bloqueo, asigne a la propiedad `extendedlogFilter` el valor `LOCK_ERRORS`.
- Utilice el método `DataStoreConnection.dumpLocks()` para informar sobre los bloqueos utilizados por otras conexiones.

## Cómo evitar bloqueos y conflictos

---

Generalmente, una conexión necesita un bloqueo para leer o escribir en un flujo o una fila. Una conexión puede verse bloqueada por otra conexión que esté leyendo o escribiendo. Hay dos formas de evitar las paradas:

- Reducir al mínimo la duración de las transacciones de escritura.

- Utilizar transacciones de sólo lectura que no requieran bloqueos para leer.

## Conservación de las transacciones de escritura

Las conexiones deberían utilizar la escritura en ráfagas. La escritura en ráfagas acumula los cambios y, cuando es necesario, inicia una transacción, escribe los cambios inmediatamente y los confirma. Ésta es la forma de trabajo habitual de la mayoría de los servidores de base de datos, y es el modelo que utiliza DataExpress en sus mecanismos de suministro y resolución.

## Transacciones de sólo lectura

Las transacciones de sólo lectura no pueden quedar bloqueadas por conexiones de escritura u otras conexiones de lectura y, como no utilizan bloqueos, no detienen nunca las demás transacciones.

Para que las conexiones de JDBC utilicen transacciones de sólo lectura, asigne el valor **true** a la propiedad `readOnly` del objeto `java.sql.Connection` (devuelto por los métodos `java.sql.DriverManager.getConnection()` y `com.borland.dx.dataset.sql.Database.getJdbcConnection()`). Cuando se utilizan objetos `DataStoreConnection`, se debe asignar el valor **true** a la propiedad `readOnlyTx` antes de abrir la conexión.

Las transacciones de sólo lectura simulan una instantánea del archivo `JDataStore`. En esta instantánea sólo aparecen los datos de las transacciones confirmadas en el momento de su inicio (de lo contrario, la conexión debe comprobar si hay cambios pendientes y cancelarlos cuando accede a los datos). La instantánea comienza cuando se abre `DataStoreConnection`. Esta se actualiza cada vez que se llama a su método `commit()`.

## Cambios en el control de concurrencia para versiones anteriores

---

Los archivos de base de datos de `JDataStore` creados con versiones anteriores del software `JDataStore` continuarán utilizando bloqueos de tabla para el control de conflictos de concurrencia. No obstante, existen algunas pequeñas mejoras en el control de concurrencia para archivos de bases de datos antiguos:

- Compatibilidad con `TRANSACTION_READ_UNCOMMITTED` y `TRANSACTION_SERIALIZABLE`.
- Utilización de bloqueos de tabla compartidos para operaciones de lectura (las versiones anteriores de `JDataStore` utilizaban bloqueos de tabla exclusivos para las operaciones de lectura y escritura).

## Agrupación de conexiones y soporte de transacciones distribuidas

---

JDataStore proporciona varios componentes para las operaciones con `DataSources` de JDBC, con grupos de conexiones y la compatibilidad con transacciones distribuidas (XA). Estas características necesitan J2EE. Si su versión de JBuilder no incluye el archivo `J2EE.jar`, deberá obtenerlo en la web de Sun y añadirlo a su proyecto como biblioteca necesaria. En “Adición de una biblioteca requerida a un proyecto” de la *Guía del desarrollador de aplicaciones de base de datos* encontrará instrucciones sobre cómo añadir una biblioteca necesaria.

### Agrupación de conexiones

---

El principal propósito de los grupos de conexiones es muy sencillo. En una aplicación en la que se abren y cierran muchas conexiones con bases de datos resulta eficiente mantener en un grupo los objetos `Connection` que queden libres para poder reutilizarlos posteriormente. De esta manera, desaparece la necesidad de abrir una nueva conexión física cada vez que se abre una conexión.

El objeto `JdbcConnectionPool` permite las transacciones XA agrupadas. Esta característica permite a un `JDataStore` participar en una transacción distribuida, actuando como gestor de recursos. `JDataStore` proporciona soporte XA mediante la implementación de tres interfaces estándar, indicadas por Sun en la especificación API de transacciones Java (JTA):

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

Para obtener una conexión distribuida a un `JDataStore` desde `JdbcConnectionPool`, puede llamar a `JdbcConnectionPool.getXAConnection()`. La conexión que devuelve este método sólo funciona con el controlador JDBC de `JDataStore`. La compatibilidad con XA sólo resulta útil si se combina con un gestor de transacciones distribuidas, como Borland Enterprise Server.

Normalmente es necesario confirmar o cancelar todas las transacciones globales antes de cerrar la `XAConnection` asociada. Si se utiliza una conexión en una transacción global que aún no está en estado preparado pero sí en estado iniciado con éxito o suspendido, la transacción se cancela.

## heuristicCompletion

---

A partir de la versión 6.0, JDataStore ofrece `heuristicCompletion`, una propiedad JDBC ampliada que permite controlar el funcionamiento cuando fallan una o más bases de datos durante una confirmación de dos fases. Cuando se preparan transacciones XA pero no se finalizan (ni se confirman ni se cancelan), se puede especificar una entre tres posibles configuraciones de cadenas para esta propiedad:

- **commit**: hace que la transacción se confirme heurísticamente cuando JDataStore vuelve a abrir la base de datos.
- **rollback**: hace que la transacción se cancele heurísticamente cuando JDataStore vuelve a abrir la base de datos.
- **none**: hace que JDataStore mantenga el estado de la transacción cuando vuelve a abrir una base de datos. Si se utiliza esta opción, los bloqueos que se mantuvieron cuando se preparó la transacción se recuperan y se mantienen hasta que la transacción se confirma o se cancela por parte de un gestor de transacciones compatible con JTA/JTS.

La configuración por defecto para esta propiedad es `commit`.

Observe que las opciones heurísticas `commit` y `rollback` permiten una ejecución más eficaz, ya que los bloqueos se pueden liberar antes, y se necesita escribir menos información en el archivo de registro de la transacción.



# Las UDF y los procedimientos almacenados

Los procedimientos **almacenados** son funciones definidas por el usuario que están diseñadas para gestionar lógica empresarial. Estas funciones sirven para ocultar la complejidad de manejar tablas relacionales. A los procedimientos almacenados se les llama directamente y, de forma opcional, disponen de parámetros de salida y/o de entrada. Por ejemplo:

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

Las **UDF** (Funciones definidas por el usuario) son funciones definidas por el usuario y diseñadas para su uso en subexpresiones de una sentencia SQL. Normalmente, una sentencia `SELECT` puede utilizar una UDF en su cláusula `WHERE`. Por ejemplo:

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

## Lenguaje para los procedimientos almacenados y las UDF

---

Los procedimientos almacenados y las UDF para JDataStore se deben escribir en Java. No existe seguridad frente al mal funcionamiento en el código escrito Java. Sin embargo, las clases compiladas Java deben añadirse a la vía de acceso a clases del proceso del servidor JDataStore para que se puedan utilizar. Esto da la oportunidad al administrador de la base de datos de controlar el código que se añade. Sólo se pueden utilizar los métodos estáticos públicos de las clases públicas.

## Procedimientos almacenados o UDF disponibles para el motor SQL

---

Después de que se haya escrito o añadido un procedimiento almacenado o una UDF a la vía de acceso a clases del proceso del servidor JDataStore, se asocia un nombre de función a ese método Java mediante la siguiente sintaxis SQL:

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

donde el *<method-name>* es un identificador SQL como `INCREASE_SALARY` y *<method-definition-string>* es una cadena con un nombre completo de método, como `com.mycompany.util.MyClass.increaseSalary`.

Se puede eliminar un procedimiento almacenado o una UDF de la base de datos si se ejecuta:

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

Una vez que se ha creado un método, ya se puede utilizar. En el siguiente apartado se ofrece un ejemplo de cómo definir y llamar a una UDF.

## Un ejemplo de UDF

---

```
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;
```

**Nota** Este ejemplo define un método que coloca el primer carácter de espacio después de un determinado índice en una cadena. El primer fragmento SQL define la UDF y el segundo muestra un ejemplo de cómo se utiliza.

Se supone que `TABLE1` cuenta con dos columnas `VARCHAR` : `FIRST_NAME` y `LAST_NAME`. La función `CHAR_LENGTH` es una función SQL integrada.



## Parámetros de entrada

---

Cuando se llama al método Java se lleva a cabo una comprobación final de tipos de parámetros. Los tipos numéricos se convierten en un tipo mayor si es necesario, para que coincidan con los tipos de parámetros de un método Java. El orden de tipos numéricos para los tipos Java es:

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

Los demás tipos Java reconocidos son:

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

Tenga en cuenta que si pasa valores NULL al método Java, no puede utilizar los tipos primitivos, como `short` y `double`. Utilice las clases de encapsulación equivalentes en su lugar (`Short`, `Double`). Un valor NULL de SQL se pasa como valor `null` Java.

Si un método Java cuenta con un parámetro de un tipo (o una matriz de un tipo) que no se encuentra recogido en las tablas anteriores, se gestiona como un tipo `OBJECT` de SQL.

## Parámetros de salida

---

Si un parámetro de método Java es una matriz de uno de los tipos de entrada reconocidos (excepto para `byte[]`), se espera que sea un parámetro de salida. `JDataStore` pasa una matriz de longitud 1 a la llamada del método, y se espera que el método rellene el primer elemento de la matriz con el valor de salida. Los tipos reconocidos Java para los parámetros de salida son:

## Parámetros de salida

- double[] o Double[]
- float[] o Float[]
- java.math.BigDecimal[]
- long[] o Long[]
- int[] o Integer[]
- short[] o Short[]
- Byte[] (pero no byte[], ya que se trata de un parámetro de entrada reconocido por sí mismo)
- boolean[] o Boolean[]
- String[]
- java.sql.Date[]
- java.sql.Time[]
- java.sql.Timestamp[]
- byte[][]
- java.io.InputStream[]

Los parámetros de salida sólo se pueden asociar a marcadores de variables en SQL. Todos los parámetros de salida son esencialmente parámetros `INOUT`, ya que cualquier valor establecido antes de que se ejecute la sentencia se pasa al método Java. Si no se establece ningún valor, el valor inicial es arbitrario. Si alguno de los parámetros da como salida un valor `NULL` de SQL (o tiene un valor de entrada `NULL` válido), utilice las clases encapsuladas en lugar de los tipos primitivos. Por ejemplo:

```
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,?);
```

La sentencia `CALL` debe prepararse con una sentencia `Callable` para conseguir el valor de salida. Consulte la documentación de JDBC para obtener más información acerca del uso de `java.sql.CallableStatement`. Observe la asignación de `result[0]` en el método Java. La matriz pasada al método tiene exactamente un elemento.

## Parámetro implícito de conexión

---

Si el primer parámetro de un método Java es del tipo `java.sql.Connection`, `JDataStore` pasa un objeto de conexión que comparte el contexto de conexión transaccional utilizado para llamar al procedimiento almacenado en primer lugar. Este objeto de conexión se puede utilizar para ejecutar sentencias SQL mediante la API JDBC.

No pase nada para este parámetro: deje que lo haga `JDataStore`.

Por ejemplo:

```
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);
```

- Notas**
- `INCREASE_SALARY` sólo requiere un parámetro: la cantidad en la que se incrementan los salarios. El método Java correspondiente tiene dos parámetros.
  - No llame a `commit`, `rollback`, `setAutoCommit` ni `close` en el objeto de conexión pasado a los procedimientos almacenados. Debido a razones de funcionamiento, no es recomendable utilizar esta función para una UDF, aunque es posible.

## Firmas de métodos sobrecargados

---

Los métodos Java se pueden sobrecargar para evitar pérdidas numéricas de precisión.

## Asignación de tipos devueltos

Por ejemplo:

```
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;
```

**Nota** El método sobrecargado `abs` sólo se registra una vez en el motor SQL. Suponga que el método `abs` que toma un `BigDecimal` no se ha implementado. Si `NUMBER1` fuera un `NUMERIC` con decimales, se llamaría al método `abs` con un `double`, que tiene la probabilidad de perder precisión cuando se convierte el número de un `BigDecimal` a un `double`.

## Asignación de tipos devueltos

---

El valor devuelto del método se asigna a un tipo SQL equivalente. A continuación aparece la tabla de asignación de tipos:

### Asignación de tipos devueltos

Tipo devuelto del método	Tipo SQL de JDataStore
byte o Byte	SMALLINT
short o Short	SMALLINT
int o Integer	INT
long o Long	BIGINT
java.math.BigDecimal	DECIMAL
float o Float	REAL
double o Double	DOUBLE
String	VARCHAR
boolean o Boolean	BOOLEAN

## Asignación de tipos devueltos (continuación)

Tipo devuelto del método	Tipo SQL de JDataStore
java.io.InputStream <b>consulte (*)</b>	INPUTSTREAM
java.sql.Date	DATE
java.sql.Time	TIME
java.sql.Timestamp	TIMESTAMP
<b>Todos los demás tipos:</b>	OBJECT

*Nota* Cualquier tipo derivado de java.io.InputStream se maneja como un INPUTSTREAM.



## Persistencia de datos en un JDataStore

Los controles enlazados a datos `dbSwing` y `JBCL` se asocian a conjuntos de datos `DataExpress`. El paradigma `DataExpress` establece una separación clara entre las fuentes de datos y los conjuntos de datos, por medio de proveedores y almacenadores, por lo que `DataExpress` es ideal para las aplicaciones de varios niveles.

Después de proporcionar los datos a una aplicación o a un módulo de datos, es posible trabajar con ellos, en el sistema local, en controles enlazados a datos. Puede almacenar los datos en la memoria local (`MemoryStore`) o en un archivo local (`DataStore`). Para guardar los datos en la base de datos es preciso "resolver" las modificaciones.

### Utilización de `DataStore` en lugar de `MemoryStore`

---

Por defecto, cuando los datos se cargan en un componente de `JBuilder`, se almacenan en la memoria a través de un componente `MemoryStore`. Puede usar sistemas de almacenamiento alternativos, como `JDataStore`, si define la propiedad `store` del objeto `StorageDataSet`. (Actualmente, `MemoryStore` y `DataStore/DataStoreConnection` son las únicas implementaciones de la interfaz `Store` requeridas por la propiedad `store`.)

Las principales ventajas de `DataStore` respecto a `MemoryStore` son la semántica de transacciones, la aceptación de SQL y la persistencia, que permite la ejecución fuera de línea. `DataStore` recuerda las filas capturadas en una tabla, incluso después de haber cerrado y reiniciado la aplicación. Además, es posible aumentar el rendimiento de cualquier aplicación con objetos `StorageDataSet` voluminosos. Los `StorageDataSet` que utilizan `MemoryStore` tienen un rendimiento ligeramente mejor que `DataStore` para

un número pequeño de filas. JDataStore almacena datos e índices en un formato muy reducido. A medida que aumenta el número de filas del objeto `StorageDataSet`, la utilización de `DataStore` proporciona un mayor rendimiento y requiere mucha menos memoria que `MemoryStore`.

Tanto si el sistema de almacenamiento de datos es `MemoryStore` como si se trata de `DataStore`, la forma de trabajar con objetos `StorageDataSet` u otros controles enlazados a datos conectados a `StorageDataSet` es la misma. Sin embargo, el almacenamiento de objetos Java en columnas requiere el uso de serialización de Java (`java.io.Serializable`). Si esto no es posible, no se pueden utilizar los componentes `DataStore` y se debe emplear el mecanismo de almacenamiento en memoria por defecto.

## Utilización de JDataStore con objetos StorageDataSets

---

Guarda en caché y guarda `StorageDataSets` en un `JDataStore` cambiando los valores de las tres propiedades que se tratan en [Conexión con JDataStore mediante StorageDataSet](#). Normalmente, los datos persistentes de un proveedor incluyen las dos subclases de `StorageDataSet` con proveedores predefinidos: `QueryDataSet` (que utiliza `QueryProvider`) y `ProcedureDataSet` (que utiliza `ProcedureProvider`).

Para almacenar y conservar los datos de uno de estos `StorageDataSet` en un `JDataStore` con las herramientas de diseño:

- 1 Seleccione el archivo Marco de la aplicación y cambie a vista de diseño.
- 2 Añada un componente `DataStoreConnection` de la pestaña `DataExpress` de la paleta de componentes al árbol de componentes.
- 3 Desde el Inspector, seleccione la propiedad `fileName` del componente `DataStoreConnection`. Pulse el botón Examinar para que aparezca el cuadro de diálogo Abrir e introduzca el nombre del archivo `JDataStore`. Pulse Abrir.
- 4 En el árbol de componentes, seleccione `QueryDataSet` o `QueryDataSet`. (Tenga en cuenta que también se puede utilizar un objeto `TableDataSet` con el valor `QueryProvider` o `ProcedureProvider` en la propiedad `provider`).
- 5 En el Inspector, asigne a la propiedad `storeName` el nombre que desea utilizar para el flujo de tabla del `JDataStore`.
- 6 En el Inspector, asigne la propiedad `store` del componente `StorageDataSet` al componente `DataStoreConnection`.



## Tutorial: Edición fuera de línea con JDataStore

---

Este tutorial explica los pasos que deben seguirse para crear una aplicación que utiliza un componente `DataStore` para activar la modificación de los datos fuera de línea. La base de datos de servidor es un archivo JDataStore de ejemplo, `employee.jds`, al que se accede desde el servidor JDataStore. No confunda este archivo JDataStore con el que se utiliza para la persistencia. Encuentre el archivo de muestra antes de empezar. Está instalado en `samples/JDataStore/datastores`.

- 1 Inicie el servidor de JDataStore del elemento de menú Herramientas | Servidor de JDataStore.
- 2 Cree una aplicación nueva. Para ello, seleccione Nuevo en el menú Archivo y haga doble clic en el icono Aplicación. En el Asistente para aplicaciones:
  - En la primera ficha, utilice el nombre de clase `PersistApp`.
  - En la segunda ficha, elija el nombre de clase de marco `PersistFrame`. Pulse el botón Finalizar.
- 3 Pase a la vista de diseño para el nuevo archivo `PersistFrame.java`.
- 4 Añada un componente `Database` desde la pestaña Data Express al árbol de componentes.
- 5 En el Inspector, abra el editor de la propiedad `connection` del componente `Database`. Defina las propiedades de conexión con la base de datos escribiendo la vía de acceso correcta al archivo `employee.jds` en lugar de `<letra de la unidad:/<jbuilder>` en la URL:

Nombre de la propiedad	Valor
Controlador	<code>com.borland.datastore.jdbc.DataStoreDriver</code>
URL	<code>jdbc:borland:dsremote://localhost/d:/jbuilder4/samples/JDataStore/datastores/employee.jds</code>
Nombre de usuario	<code>&lt;utilice cualquier nombre&gt;</code>
Contraseña	<code>&lt;déjela en blanco&gt;</code>

- 6 Haga clic sobre el botón Probar conexión para comprobar que las propiedades de conexión se han establecido correctamente. Cuando la conexión sea satisfactoria, pulse Aceptar.
- 7 Añada un componente `DataStoreConnection` de la pestaña DataExpress al árbol de componentes.

Cuando se añade un componente `DataStoreConnection` se escribe en el código una sentencia **import** para el paquete `datastore` y la biblioteca JDataStore se añade a las propiedades del proyecto si no estaba ya en ellas.

- 8** Abra el editor de la propiedad `fileName` del componente `DataStoreConnection`. Escriba el nombre de un nuevo archivo `JDataStore`. No olvide incluir la vía de acceso. Para hacerlo, puede utilizar el botón Examinar. No es necesario incluir la extensión ya que los archivos `JDataStore` siempre tienen la extensión `.jds`. Pulse Aceptar.

**Nota:** El diseñador crea automáticamente el archivo `JDataStore` cuando se conecta con el objeto `StorageDataSet`, por lo que las herramientas funcionan a pleno rendimiento. Cuando se ejecuta la aplicación, el archivo `JDataStore` ya está en su sitio. Sin embargo, si la aplicación se ejecuta en un ordenador distinto, el archivo `JDataStore` no se encontrará en su lugar. Tendrá que añadir más código para crear el archivo `JDataStore` si fuera necesario, tal y como se muestra en [Creación de archivos JDataStore](#).

- 9** Añada un componente `QueryDataSet` de la pestaña Data Express en el árbol de componente.
- 10** Abra el editor de la propiedad `query` del componente `QueryDataSet` en el Inspector y defina las siguientes propiedades:

Nombre de la propiedad	Valor
Base de datos	<code>database1</code>
Sentencia SQL	<code>select * from employee.</code>

- 11** Haga clic en Probar consulta para cerciorarse de que la consulta se ejecuta correctamente. Una vez que el área gris debajo del botón indique `Correcto`, pulse Aceptar y se cerrará el cuadro de diálogo.
- 12** Asigne a la propiedad `storeName` de `QueryDataSet` el valor `employeeData`.
- 13** Asigne a la propiedad `store` el valor `dataStoreConnection1` (la única opción).
- 14** Añada un componente `JdbNavToolBar` de la pestaña `dbSwing` a la ubicación Norte del marco. Asigne a la propiedad `dataSet` el valor `queryDataSet1`.
- 15** Añada un componente `JdbStatusLabel` de la pestaña `dbSwing` a la ubicación Sur del marco. Asigne a la propiedad `dataSet` el valor `queryDataSet1`.
- 16** Añada un componente `TableScrollPane` de la pestaña `dbSwing` a la ubicación Centro del marco.
- 17** Añada un componente `JdbTable` de la pestaña `dbSwing` a `TableScrollPane`. Asigne a su propiedad `dataSet` el valor `queryDataSet1`.

- 18** En lugar de añadir código que llame a `DataStore.shutdown()` antes de salir de la aplicación, como se ha hecho en un tutorial anterior, se puede utilizar un componente `DBDisposeMonitor` para cerrar automáticamente los archivos `JDataStore` cuando se cierra el marco.
- 19** Añada al árbol de componentes un componente `DBDisposeMonitor` de la pestaña Más dbSwing. Asigne a la propiedad `dataAwareComponentContainer` el valor `this`.
- 20** Ejecute `PersistApp.java`.

En la aplicación que se está ejecutando, modifique los datos y pulse el botón Registrar del navegador para guardar los cambios efectuados en el archivo `JDataStore` (la persistencia `JDataStore` especificada en el paso 8). Los cambios también se guardan en el archivo cuando se sale de una fila, tal y como ocurre con los conjuntos de datos que se encuentran en la memoria (`MemoryStore`).

**Nota:** Los conjuntos de datos de los archivos `JDataStore` pueden tener cientos de miles de filas. La gestión de todos estos datos en la memoria reduciría de forma considerable el rendimiento de la aplicación.

Cierre la aplicación y vuelva a ejecutarla. Los datos aparecen como se dejaron en la ejecución anterior. Este comportamiento es distinto del que tienen los conjuntos de datos que se encuentran en la memoria. Si lo desea, salga de la aplicación, cierre el servidor `JDataStore` y vuelva a ejecutar la aplicación. Aunque no exista una conexión con la base de datos SQL sigue siendo posible ver y modificar los datos del archivo `JDataStore`. Esto resulta especialmente útil si se desea trabajar con los datos fuera de línea, por ejemplo en un ordenador portátil.

## Gestión de datos fuera de línea con JDataStore

---

Hasta el momento no se ha vuelto a guardar nada en la base de datos SQL del servidor. En un `JdbNavToolBar` hay distintos botones:

- El botón Registrar guarda los cambios efectuados en la fila actual del archivo `JDataStore`.
- El botón Guardar guarda en el servidor todos los cambios acumulados en el archivo `JDataStore`. `DataExpress` busca automáticamente la forma de devolver los cambios al servidor SQL. En el código, el método correspondiente es `DataSet.saveChanges()`.
- El botón Actualizar devuelve la consulta y sobrescribe los datos del archivo `JDataStore` incluidos los cambios que no están guardados en el servidor con el resultado de la consulta. En el código, el método correspondiente es `DataSet.executeQuery()`.

Las opciones establecidas en `queryDescriptor` también influyen en cómo se almacenan, guardan y actualizan los datos. En el `queryDescriptor` de este ejemplo se selecciona la opción Ejecutar consulta inmediatamente al abrir. Esta opción indica la forma en que se cargan los datos en el archivo `JDataStore` cuando la aplicación se ejecuta por primera vez. En las ejecuciones posteriores se elimina la ejecución de la consulta, porque el conjunto de datos se encuentra en el archivo `JDataStore` y no en el servidor. Como resultado:

- Los cambios que no se han guardado en el servidor se conservan cuando se sale de la aplicación y se reinicia.
- No necesita escribir ningún código especial para obtener datos en el `JDataStore` en la primera ejecución.
- Cuando los datos se encuentran en el archivo `JDataStore` es posible trabajar fuera de línea. De hecho, ni siquiera se establece una conexión con la base de datos hasta que se efectúa alguna operación que la requiera, como guardar cambios.

Cuando la opción Ejecutar consulta al abrirla está seleccionada, los datos existentes no pueden sobrescribirse (a menos que llame al método `StorageDataSet.Métodorefresh()` explícitamente). Esto significa que se pueden cerrar y volver a abrir los conjuntos de datos para cambiar la configuración de las propiedades en `MemoryStore` o `DataStore` sin correr el riesgo de perder datos.

Cuando los datos se encuentran en el archivo `JDataStore`, es posible ejecutar esta aplicación y modificar los datos aunque el servidor de base de datos no esté disponible. Cuando se trabaja fuera de línea no se deben pulsar los botones Guardar y Actualizar del navegador. Si se pulsan, se producirá una excepción al fallar el intento de conexión, aunque no se perderán los cambios efectuados.

## Reestructuración de los `StorageDataSet` de `JDataStore`

---

El editor de columnas persistentes del diseñador de interfaces de `JBuilder` permite trasladar, eliminar y añadir columnas. También se pueden cambiar los tipos de datos de las columnas. Cuando se modifica el tipo de datos de un componente `Column` de un objeto `StorageDataSet` enlazado a un archivo `JDataStore`, se producen conversiones de tipo de datos al pasar de un tipo a otro.

Para activar el diseñador de columnas:

- 1 Seleccione el archivo Módulo de datos en el panel de desplazamiento.
- 2 Seleccione la pestaña Diseño del `DataModule`.
- 3 Haga clic con el botón derecho sobre un `StorageDataSet` en el panel de estructura de `JBuilder`.

#### 4 Seleccione Activar diseñador.

El diseñador de columnas funciona con los `StorageDataSet` que estén utilizando un `MemoryStore` o un `DataStore`. `MemoryStore` realiza todas las operaciones de forma instantánea. Cuando se cambia un tipo de datos `Column`, `MemoryStore` no convierte los valores de los datos al nuevo tipo. Se pierden los valores anteriores.

`JDataStore` no realiza de forma inmediata las operaciones trasladar/añadir/eliminar/cambiar tipo en los `StorageDataSets`. El cambio estructural se registra dentro del directorio `JDataStore` como operación pendiente. El método `StorageDataSet.getNeedsRestructure()` devuelve **true** cuando hay una operación de reestructuración pendiente. Puede seguir utilizando `StorageDataSet` con cambios estructurales pendientes:

- Se puede leer y escribir en las columnas desplazadas.
- No se pueden ver las columnas eliminadas.
- Es posible leer las columnas insertadas, pero no escribir en ellas.
- Es posible leer las columnas con el tipo de datos cambiado, pero no escribir en ellas.

Para llevar a cabo una operación de reestructuración pendiente, haga clic en el botón Reestructurar de la barra de herramientas del diseñador de columnas. También se puede provocar una operación de reestructura por medio del código, con una llamada al método `StorageDataSet.restructure()`.

El método `restructure()` puede utilizarse aunque no haya cambios estructurales pendientes para reparar o compactar un `StorageDataSet` y sus índices asociados. (En el método `DataStoreConnection.copyStreams()` se explica otra forma de reparar los flujos dañados).

## Conversión de tipos de datos

Cuando se modifica el tipo de datos de un componente `Column` de un objeto `StorageDataSet` enlazado a un archivo `JDataStore` se producen conversiones de tipo de datos al pasar de un tipo a otro. En la tabla siguiente se describe lo que ocurre cuando un tipo de datos se convierte en otro. A la izquierda se muestra el tipo de datos original del componente `Column`, y en la parte superior de la tabla se encuentran los tipos de datos nuevos de `Column`.

Desde\Hasta	Big Decimal	Double	Float	Long	int	Short	boolean	Hora	Fecha	Time stamp	Cadena	Input Stream	Object
Big Decimal	Ning	Prec	Prec	Prec	Prec	Prec	Prec	Prec	Prec	Prec	Bien	Perd	Perd
Double	Prec	Ning	Prec	Prec	Prec	Prec	Prec	Prec	Prec	Prec	Bien	Perd	Perd
Float	Prec	Bien	Ning	Prec	Prec	Prec	Prec	Prec	Prec	Prec	Bien	Perd	Perd

Desde/ Hasta	Big Decimal	Double	Float	Long	int	Short	boolean	Hora	Fecha	Time stamp	Cadena	Input Stream	Object
<b>Long</b>	Bien	Prec	Prec	Ning	Prec	Prec	Prec	Prec	Prec	Prec	Bien	Perd	Perd
<b>int</b>	Bien	Bien	Prec	Bien	Ning	Prec	Prec	Prec	Prec	Prec	Bien	Perd	Perd
<b>Short</b>	Bien	Bien	Bien	Bien	Bien	Ning	Prec	Prec	Prec	Prec	Bien	Perd	Perd
<b>boolean</b>	Bien	Bien	Bien	Bien	Bien	Bien	Ning	Prec	Prec	Prec	Bien	Perd	Perd
<b>Hora</b>	Prec	Prec	Prec	Prec	Prec	Prec	Prec	Prec	Prec	Prec	Bien	Perd	Perd
<b>Fecha</b>	Prec	Prec	Prec	Prec	Prec	Prec	Prec	Ning	Ning	Prec	Bien	Perd	Perd
<b>Hora marca</b>	Prec	Prec	Prec	Prec	Prec	Prec	Prec	Prec	Prec	Ning	Bien	Perd	Perd

Leyenda:

- Perd: se han perdido todos los datos en esta conversión.
- Ning: no es necesaria la conversión.
- Prec: posible pérdida de precisión en esta conversión.
- Bien: no se han perdido datos en la conversión.

## Editor de columnas persistentes

Muchos DataSet, como QueryDataSet y ProcedureDataSet, obtienen de un servidor SQL sus datos de estructura y filas. Pero a veces es necesario añadir columnas a estos DataSet o crear tablas independientes. Puede hacerlo con el diseñador de columnas de JBuilder. Puede utilizarlo para modificar, eliminar, desplazar y añadir columnas DataSet de forma instantánea en la fase de diseño. El JDataStore utiliza una tabla de asignación para simular que se han realizado las operaciones estructurales. Cuando haya realizado todos los cambios estructurales, haga clic sobre Reestructurar.

Si se activa el diseñador y hay un nodo StorageDataSet seleccionado, las columnas de dicho nodo se visualizan en una rejilla sobre la superficie de diseño. Por encima de la rejilla hay una barra de herramientas para añadir, eliminar, desplazar arriba o abajo y reestructurar el conjunto de datos.

- Arriba y Abajo manipulan la propiedad de orden de preferencia de las columnas.
- Añadir añade una columna en el orden preferido con respecto a la columna seleccionada en la rejilla.
- Borrar elimina la columna del conjunto de datos.

- Reestructurar compila el componente **this** y ejecuta una MV independiente para realizar una reestructuración del JDataStore asociado al conjunto de datos. El estado de la reestructuración se muestra dinámicamente en un cuadro de diálogo. Este cuadro incluye un botón Cancelar que permite suspender la operación. Sólo es posible reestructurar si se ha asignado a la propiedad `store` del conjunto de datos un componente `DataStore` o `DataStoreConnection` (consulte [Utilización de JDataStore con objetos StorageDataSets](#)).

## Cambios en la estructura

---

Independientemente de que se utilice o no un JDataStore, las columnas facilitadas por una consulta o un procedimiento se deben combinar con las definidas en el código. Cuando se utiliza un archivo JDataStore, esta combinación debe producirse y, a continuación, se combina la lista de columnas resultante con las que se encuentran en el conjunto de datos.

En teoría, las columnas del archivo JDataStore son las mismas que las que resultan de la combinación. Sin embargo, es frecuente que se produzcan cambios durante el desarrollo. Incluso pueden producirse ocasionalmente en las aplicaciones ya finalizadas. Es posible seguir utilizando los datos establecidos en JDataStore cuando las columnas que contiene no son las mismas que las resultantes de la combinación de aplicación/proveedor. La operación de provisión reestructura la tabla tras la combinación.

Si se modifica la estructura de un conjunto de datos en el diseñador en vez de cambiar el código fuente, JDataStore efectúa un seguimiento de los cambios. Esto facilita la combinación entre las columnas aplicación / proveedor y las columnas del archivo JDataStore. Por ejemplo, si se modifica el nombre de una columna calculada por medio del diseñador, se puede seguir presentando y modificando los valores de esta columna porque el archivo JDataStore puede asignar el nombre de columna antiguo al nuevo. Si se efectúa el mismo cambio en el código, el archivo JDataStore sólo puede establecer que se ha borrado una columna y se ha añadido otra, pero no reconoce el cambio de nombre. Por tanto, la columna borrada no se presenta y la columna añadida está vacía y no es posible modificarla.

En la lista siguiente se proporciona más información sobre los distintos tipos de cambios de estructura de los conjuntos de datos:

- Insertar columna: Hasta que se realiza la reestructuración, la columna es visible pero aparece vacía y no es posible modificarla.
- Eliminar columna: La columna no es visible, pero sigue existiendo en el almacén hasta que se efectúa la reestructuración.
- Cambiar nombre de columna: El cambio de nombre entra en vigor de inmediato.

- Cambiar orden de columna: a columna sigue siendo visible y es posible modificarla. Hasta que se efectúa la reestructuración, el rendimiento se ve afectado ligeramente cuando se efectúa la asignación del orden de la columna establecido en la aplicación y el que se encuentra realmente en el archivo JDataStore.
- Cambiar tipo de datos de columna: la columna sigue siendo visible, pero no es posible modificarla hasta que se efectúa la reestructura.

Este proceso también compacta el conjunto de datos y borra los índices. Los índices se generan cuando se necesitan.



## El Explorador de JDataStore

Mediante el Explorador de JDataStore completamente escrito en Java, es posible:

- Examinar el contenido de un JDataStore. El directorio de JDataStore se muestra en un control arborescente, donde se agrupan cada tabla con sus respectivos índices. Cuando se selecciona un flujo de datos en la arborescencia, se muestra su contenido (suponiendo que sea un tipo de archivo como archivo de texto, imagen GIF o una tabla, para los cuales el Explorador cuenta con un visualizador).
- Desarrollar muchas operaciones JDataStore sin escribir código fuente. Es posible crear archivos JDataStore, crear tablas e índices, importar archivos de texto delimitados a conjuntos de datos, importar archivos como flujos de archivo, eliminar tablas e índices, conjuntos de datos u otros flujos de datos y comprobar la integridad del JDataStore.
- Gestionar consultas que proporcionen datos a conjuntos de datos en el JDataStore, editar los conjuntos de datos y guardar las modificaciones en las tablas del servidor.
- Gestione las características de seguridad de un JDataStore, como usuarios, contraseñas y cifrado.

### Inicio del Explorador de JDataStore

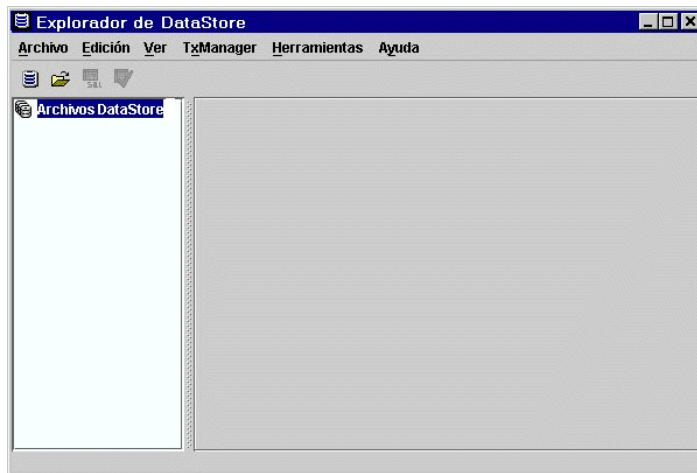
---

Existen tres maneras de abrir el Explorador de JDataStore:

- En JBuilder utilice el comando de menú Herramientas | Explorador de JDataStore.
- Desde la línea de comandos o con un atajo de teclado.

- En el servidor de JDataStore (consulte [Ejecución del Servidor de JDataStore](#)), utilice el comando de menú Archivo | Explorador de JDataStore.

**Figura 9.1** Explorador de JDataStore abierto



## Inicio desde la línea de comandos

---

El Explorador de JDataStore requiere los siguientes archivos JAR:

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

Estos archivos también son necesarios si desea que esté disponible la ayuda en línea:

- jb\_ui.jar
- help.jar

Un archivo ejecutable para el servidor también está disponible, y puede iniciarse desde la línea de comandos. El ejecutable utiliza la vía de acceso y las configuraciones de los nombres de clases principales en el archivo `dss.config`.

La clase principal del Explorador de JDataStore es `com.borland.dbtools.dsx.DataStoreExplorer`. La línea de comandos que inicia el servidor con las opciones por defecto una vez definida la vía de acceso a clases es:

```
java com.borland.dbtools.dsx.DataStoreExplorer
```

También se pueden especificar las opciones que se enumeran en la tabla siguiente:

**Tabla 9.1** Opciones de inicio del Explorador de JDataStore

Opción	Descripción
-ui=<tipo de interfaz de usuario>	Aspecto de la interfaz de usuario. Uno de los siguientes: <ul style="list-style-type: none"> <li>• windows</li> <li>• motif</li> <li>• metal</li> <li>• none</li> <li>• &lt;nombre de clase LookAndFeel&gt;</li> </ul>
-h=<directorio de ayuda>	El directorio que contiene los archivos de ayuda en pantalla.
<.jds filename>	Un archivo JDataStore para abrir al inicio
-?	Muestra un mensaje que enumera estas opciones.

## Operaciones básicas de JDataStore

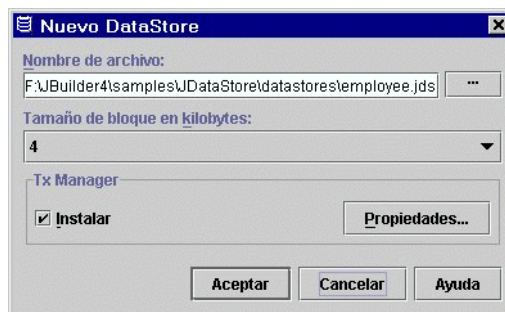
La mayoría de las operaciones de JDataStore en el Explorador de JDataStore necesitan un archivo JDataStore. Puede crear uno o abrir un archivo JDataStore disponible. Se pueden abrir varios archivos JDataStore a la vez.

### Creación de archivos JDataStore

Para crear un archivo JDataStore:

- 1 Seleccione Nuevo en el menú Archivo o pulse el botón Nuevo JDataStore de la barra de herramientas. Se abre el cuadro de diálogo Nuevo DataStore:

**Figura 9.2** Cuadro de diálogo Nuevo DataStore



- 2 Escriba un nombre para el archivo JDataStore.

- 3 (Optativo) Seleccione un tamaño de bloque distinto a los 4 KB por defecto.
- 4 Asegúrese de que la casilla de selección de Install TxManager está seleccionada o no, según corresponda:
  - Para un JDataStore no transaccional, la casilla de selección debe estar vacía.
  - Para un JDataStore transaccional, la casilla de selección debería estar seleccionada. Puede pulsar Propiedades para asignar las propiedades de gestión de la transacción en el nuevo archivo JDataStore.
- 5 Pulse Aceptar. El almacén se crea y se abre en el Explorador de JDataStore.

## Apertura de un archivo JDataStore

---

Para abrir un archivo JDataStore:

- 1 Seleccione Archivo | Abrir o pulse el botón Abrir JDataStore de la barra de herramientas. De este modo se abre la versión Java del cuadro de diálogo estándar Abrir archivos.
- 2 Seleccione el archivo deseado y pulse Abrir.

El Explorador de JDataStore conserva referencias de los cinco últimos archivos abiertos. Se pueden abrir directamente desde el menú Archivo.

## Definición de las opciones de apertura de los archivos JDataStore

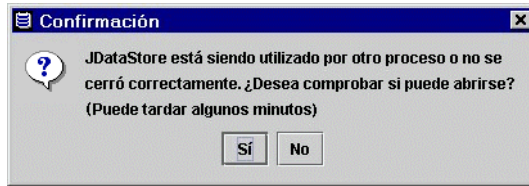
También puede ser conveniente abrir un archivo JDataStore en modo de sólo lectura para omitir provisionalmente la aceptación de transacciones o intentar abrir un archivo dañado. Seleccione Ver | Opciones para abrir el cuadro de diálogo Opciones.

**Figura 9.3** Cuadro de diálogo de Opciones de JDataStore



## Apertura de un archivo JDataStore cerrado inadecuadamente

Si el archivo JDataStore está marcado como abierto, lo que ocurre si el archivo JDataStore no se cerró correctamente, aparece un cuadro de dialogo preguntando si desea intentar la apertura de JDataStore de todas formas.

**Figura 9.4** Cuadro de diálogo de un JDataStore marcado como abierto

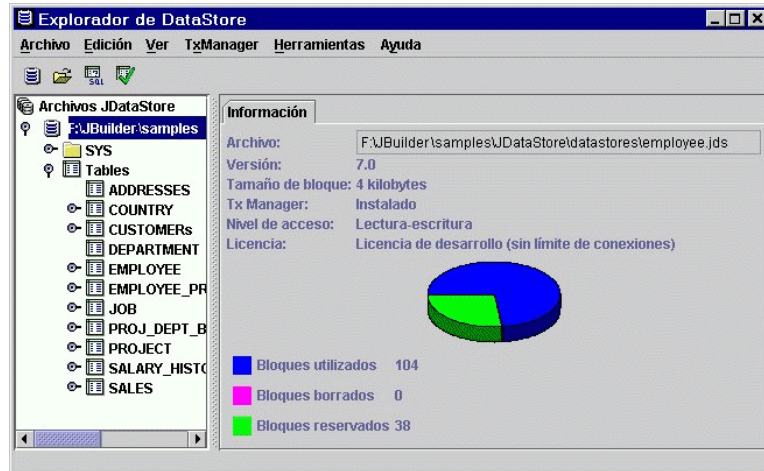
Si ocurre, siga estos pasos:

- 1 Compruebe que no se están desarrollando procesos que pudieran tener abierto el archivo JDataStore (en particular, compruebe en el TxManager que no hay procesos sueltos de `javaw`):
  - Si no hay procesos que mantengan abierto el archivo JDataStore, pulse Sí para reabrirlo.
  - Si pulsa No, el Explorador de JDataStore responde con un cuadro de diálogo de error que dice que el archivo aún permanece marcado como abierto por otro proceso y el archivo no se abre.
- 2 El intento de reapertura del archivo puede llevar varios segundos. Si el archivo JDataStore no se cerró adecuadamente, otro cuadro de diálogo informa de ello. Pulse Aceptar para intentar la recuperación del archivo.
- 3 Tras haber abierto con éxito un archivo JDataStore que estaba marcado como abierto, se abre un cuadro de diálogo que le da la oportunidad de comprobar su contenido. Pulse Sí para comprobar el contenido del archivo JDataStore o pulse No si desea pasar por alto dicha comprobación.

## Visualización de la información en archivos JDataStore

---

Cuando se abre un archivo JDataStore, su directorio aparece a la izquierda en un control arborescente. Todos los archivos JDataStore abiertos son nodos que salen directamente del nodo de raíz. La información sobre el archivo JDataStore aparece a la derecha en el área de visualización:

**Figura 9.5** Explorador de JDataStore con información de archivos JDataStore

Esta información incluye:

- El nombre del archivo JDataStore.
- El número de versión del formato de archivo JDataStore.
- El tamaño del bloque.
- Si el componente TxManager se encuentra instalado, esto es, si el archivo JDataStore es transaccional.
- Si el archivo JDataStore se ha abierto como lectura y escritura o sólo lectura.
- El tipo de licencia que se está utilizando.
- Una representación gráfica y un recuento de cómo están asignados los bloques:
  - Bloques en uso.
  - Bloques anteriormente ocupados por datos que ahora figuran como eliminados y están disponibles para su reutilización.
  - Bloques reservados asignados para usos futuros (los archivos JDataStore transaccionales asignan espacio de disco para mejorar la fiabilidad).

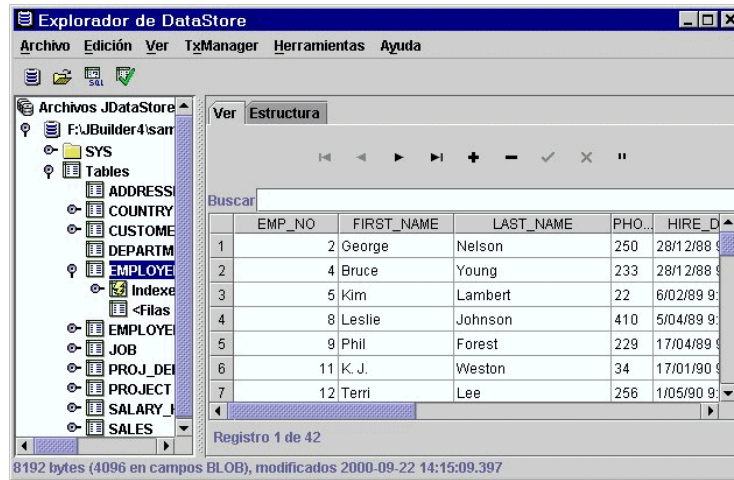
Esta información se puede ver en cualquier momento seleccionando en nodo en la arborescencia que contenga el nombre del archivo JDataStore.

## Visualización del contenido del flujo

El directorio JDataStore aparece en el árbol de directorios a la izquierda. El directorio utiliza barras (“/”) en los nombres de flujos para sintetizar una estructura jerárquica. Además, los tipos de flujo conocidos, como las tablas y los archivos de texto, aparecen en la arborescencia bajo el nodo correspondiente. Puede utilizar las opciones Ampliar todo y Contraer todo del menú Ver para situarse mejor en el árbol de directorios.

Cuando selecciona un flujo en el árbol, el contenido del flujo se verá si hay un visualizador apropiado. Existe un visualizador incorporado para flujos de tabla.

**Figura 9.6** Explorador de JDataStore con una tabla almacenada en JDataStore



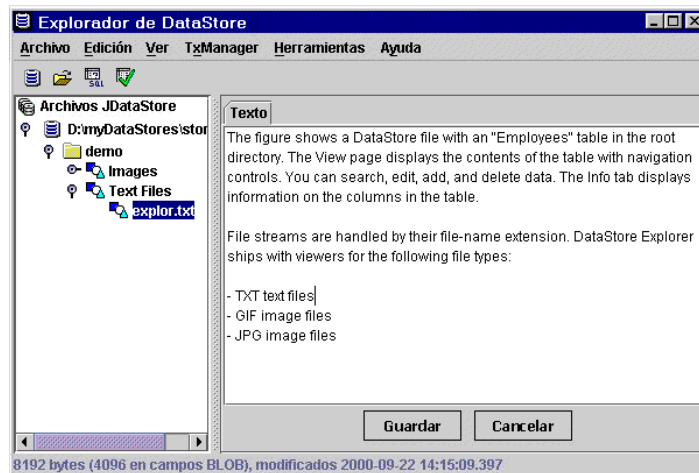
La figura muestra un archivo JDataStore con una tabla “Employees” en el directorio raíz. La ficha Ver presenta el contenido de la tabla, con controles de desplazamiento. Puede buscar, editar, añadir y eliminar datos. La pestaña Información muestra la información de las columnas en la tabla.

Los flujos de archivo son gestionados por su extensión de nombre de archivo. El Explorador de JDataStore incluye visualizadores para los siguientes tipos de archivo:

- Archivos de texto TXT.
- Archivos de imagen GIF.
- Archivos de imagen JPG.

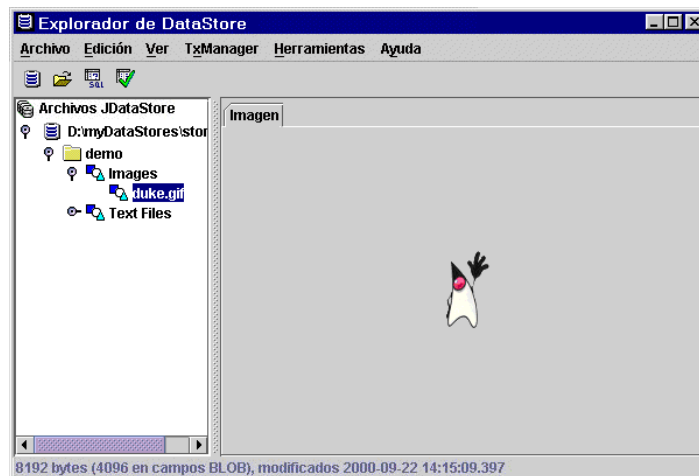
Por ejemplo, éste es el archivo de texto “demo/explor.txt”:

**Figura 9.7** Explorador de JDataStore mostrando un archivo de texto almacenado en JDataStore



Éste es el archivo de imagen "demo/duke.gif":

**Figura 9.8** Explorador de JDataStore mostrando una imagen almacenada en JDataStore



## Redenominación de flujos

Utilice esta operación para renombrar un flujo o trasladar el flujo a otro directorio:

- 1 Seleccione el flujo que debe renombrar / trasladar en el árbol de directorios. (No se puede cambiar el nombre de los flujos que hayan sido eliminados. Primero se deben recuperar).
- 2 Seleccione Renombrar en el menú Edición.



- 3 Escriba el nuevo nombre en el cuadro de diálogo Renombrar. No es posible utilizar el nombre de otro flujo activo.
- 4 Pulse Aceptar.

## Eliminación de flujos

---

Cuando borra un flujo, los bloques utilizados se marcan como disponibles. Es posible recuperar un flujo borrado, si bien algunos bloques pueden haber sido ocupados por otros flujos. Consulte [Utilización de los bloques eliminados en JDataStore](#) para obtener más información.

Para borrar un flujo:

- 1 Seleccione en el árbol de directorios el flujo que desea eliminar.
- 2 Seleccione Borrar en el menú Edición. El flujo queda marcado en el árbol como eliminado.

## Recuperación de flujos

---

Si un flujo eliminado es visible en el árbol de directorios, puede ser recuperado. No se puede recuperar un flujo si existe otro flujo activo con un nombre idéntico en el mismo directorio, ya que no puede haber dos flujos con un mismo nombre.

No borrar un flujo no garantiza que todos los datos de ese flujo se recuperarán. Consulte [Utilización de los bloques eliminados en JDataStore](#) para obtener más información.

Para recuperar un flujo:

- 1 Seleccione en el árbol de directorios el flujo borrado que debe recuperar.
- 2 Seleccione Recuperar borrado en el menú Edición. Se elimina la marca de borrado del icono del flujo.

## Copia de flujos JDataStore

---

Puede utilizar el explorador de JDataStore para copiar flujos a otro archivo JDataStore, como ocurre con el método `DataStoreConnection.copyStreams()`. Mientras no pueda utilizar esta opción para realizar copias de flujos en el mismo archivo JDataStore, el explorador de JDataStore crea automáticamente un nuevo archivo JDataStore.

Para copiar flujos:

- 1 Seleccione Copiar JDataStore en el menú Herramientas. Se abre el cuadro de diálogo Copiar flujos:

Figura 9.9 Cuadro de diálogo Copiar flujos



- 2 Especifique las diversas opciones de Copiar flujos, tal como se presentan en la siguiente tabla. Para obtener más información, consulte [Parámetros de copyStreams](#).

Tabla 9.2 Opciones de Copiar flujos

Nombre	Descripción
JDataStore destino	Nombre del archivo JDataStore destino.
Directorio destino	Nombres de copias de los flujos tienen el directorio raíz reemplazado por. Si no se desea cambiar el nombre, el directorio de destino y el original deben coincidir.
Directorio fuente	El nombre del flujo debe empezar con esto para que haya concordancia con el patrón. Si está vacío se entiende que coincide con todos los flujos.
Modelo de fuente	Modelo del nombre de flujo con el que se debe concordar, en el que se incluyen los comodines estándar * y ?

- 3 Pulse Aceptar.

## Comprobación del JDataStore

Para comprobar la estructura y el contenido del archivo DataStore, seleccione Verificar DataStore en el menú Herramientas o pulse el botón Verificar DataStore de la barra de herramientas.

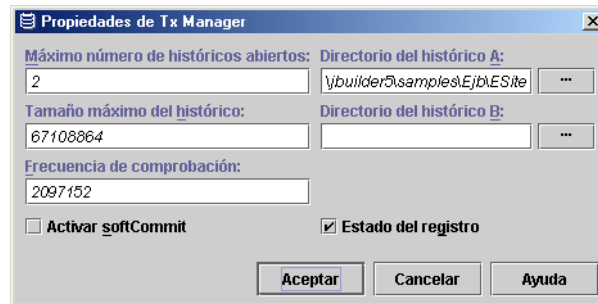
El Explorador de JDataStore verifica el almacén y muestra los resultados en la ventana Histórico de verificación. Una vez haya cerrado la ventana Histórico, puede volver a abrirla seleccionando Ver | Histórico de verificación.

## Conversión del JDataStore en transaccional

Es posible convertir un archivo JDataStore no transaccional en transaccional con el Explorador de JDataStore:

- 1 Seleccione Instalar en el menú TxManager. (Si el archivo JDataStore ya es transaccional, esa opción de menú está desactivada.) Esta opción abre el cuadro de diálogo Propiedades de TxManager.

**Figura 9.10** Cuadro de diálogo Propiedades de TxManager



- 2 El cuadro de diálogo incluye los valores por defecto del objeto TxManager:
  - Puede cambiar los valores por defecto para el número máximo de históricos abiertos (2), el tamaño máximo del histórico (64 MB) y la frecuencia de comprobación (2 MB).
  - Por defecto, en el mismo directorio del archivo JDataStore se escribe un conjunto de históricos. Para seleccionar otro directorio, especifique un Directorio de histórico A distinto.
  - Para mantener un conjunto redundante de históricos, especifique un Directorio de histórico B.
  - Se puede activar el envío por software, lo que aumenta el rendimiento, ya que no fuerza la escritura en disco de forma automática cuando se envía una transacción.
  - Puede desactivar estado de la grabación para obtener una leve mejora del rendimiento.
- 3 Pulse Aceptar.

## Modificación de los valores de transacción

Para modificar los valores de transacción de un archivo JDataStore:

- 1 Seleccione Modificar en el menú TxManager. (Si el archivo JDataStore no es transaccional, esa opción de menú está desactivada). Se abre el cuadro de diálogo Propiedades de TxManager.

- 2 Cambie los valores en función de sus objetivos.
- 3 Pulse Aceptar.

## Desactivación de la característica de transacciones

---

Puede comprobar si un archivo JDataStore es transaccional si selecciona el menú TxManager. Si la opción de menú Activado está seleccionada, el archivo JDataStore es transaccional. Si la opción está desactivada, no lo es.

Para que el JDataStore transaccional deje de serlo, desactive TxManager | Activado. Así se desactiva inmediatamente esta característica.

## Cierre de archivos JDataStore

---

Cuando haya terminado de utilizar el archivo JDataStore, debe cerrarlo para asegurarse de que todos los cambios se escriben adecuadamente. No necesita cerrar archivos JDataStore antes de salir del explorador de JDataStore. Los archivos se cierran automáticamente.

Para cerrar el archivo JDataStore, seleccione Archivo | Cerrar. Para cerrar todos los archivos JDataStore, seleccione Archivo | Cerrar todo.

## El Explorador de JDataStore como consola de consultas

---

Cuando el archivo JDataStore se utiliza como caché de datos persistentes (que significa que los datos no se pierden cuando se cierra la aplicación), le corresponde a la aplicación implementar la lógica para gestionar los datos. Esta lógica puede incluir código fuente para conectarse con un servidor, sentencias SQL para cargar conjuntos de datos con datos procedentes del servidor y código fuente para guardar las actualizaciones en el servidor, además de todas las opciones de presentación y gestión de datos que permita la aplicación.

El Explorador de JDataStore puede sustituir a una aplicación sencilla. Dentro del Explorador, y sin escribir código fuente adicional, puede definir una conexión con una base de datos, definir consultas SQL hechas a tablas de dicha base de datos, ejecutar las consultas para producir los conjuntos de datos que se almacenan en el archivo JDataStore, editar los conjuntos de datos, guardar los cambios en la base de datos y volver a ejecutar consultas para obtener los datos más recientes del servidor.

Este procedimiento también se puede utilizar simplemente para importar datos de otra base de datos a un archivo JDataStore. Dado que se guarda la información de consulta y conexión utilizada para importar los datos, éstos se pueden volver a importar fácilmente si se desea.

## Utilización del Explorador de JDataStore para gestionar consultas

---

Este apartado sobre las formas de almacenar y ejecutar consultas trae a colación la diferencia existente entre consultas hechas al servidor para producir conjuntos de datos de almacén y consultas hechas a éstos que producen nuevas tablas (que pueden o no almacenarse en un archivo JDataStore). En el caso anterior, el JDataStore proporciona almacén persistente para las tablas creadas al consultar el servidor. En el caso siguiente, el JDataStore funciona como un servidor en sí mismo, y como otros servidores, se accede a él ejecutando sentencias SQL a través de una conexión JDBC.

Aquí se presenta un tercer uso complementario de JDataStore: como un lugar donde almacenar consultas e información de conexiones. Estas consultas pueden ser de cualquiera de los tipos recién mencionados; pueden obtener sus datos en tablas de un servidor o en conjuntos de datos de un JDataStore. En cualquier caso, el resultado de ejecutar una consulta a través del Explorador de JDataStore siempre es otra tabla de almacén.

Cuando se utiliza el Explorador para gestionar consultas, es importante conocer los objetos implicados y cómo se relacionan entre sí:

- En un JDataStore, puede definir varias conexiones a bases de datos.
- Una misma conexión puede servir para muchas consultas. Estas seleccionan datos de las tablas de la misma base de datos.
- Todas las consultas crean una tabla. Cuando se define y se ejecuta la consulta en el Explorador, la tabla resultante siempre se almacena en el JDataStore.

Esta es la jerarquía:

- Muchas conexiones por JDataStore.
- Muchas consultas por conexión.
- Una tabla por consulta.

La interfaz de usuario para guardar los cambios y actualizar los datos refleja esta organización, como se describe en [Actualización de datos y almacenamiento de los cambios](#).

La información de conexión y las sentencias SQL de consulta, que habitualmente están incrustadas en el código fuente de la aplicación, se almacenan en el JDataStore en dos conjuntos de datos especiales denominados “SYS/Conexiones” y “SYS/Consultas”.

### Limitaciones del Explorador de JDataStore

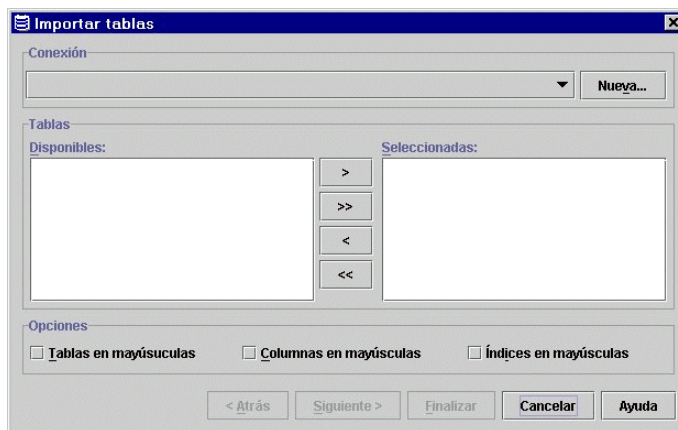
Si bien el Explorador de JDataStore puede ejecutar consultas y guardar los cambios de los conjuntos de datos en las tablas del servidor, no posee la lógica

propia de una aplicación. Los datos se presentan en una tabla sencilla. Tampoco se llevará a cabo la validación de datos y la posibilidad de introducirlos, habitualmente incluidos en el código fuente de un módulo de datos. Si edita datos en el Explorador, no se encontrará con máscaras de edición o listas de selección, no encontrará obstáculos para introducir valores que incumplan las restricciones de mínimos y máximos e incluso puede llegar a dejar vacíos campos imprescindibles o incumplir restricciones de integridad referencial. (Las restricciones definidas en el servidor se ponen de manifiesto, pero no antes de que intente guardar los cambios). El Explorador resulta útil para trabajar con datos de prueba o para realizar pequeños cambios a los datos de producción, pero no sustituye a un módulo de datos escrito con requisitos de integridad específicos de los conjuntos de datos.

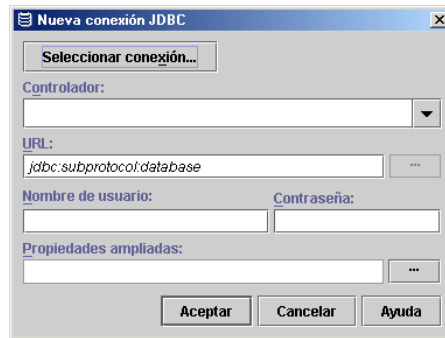
## Creación y mantenimiento de consultas y conexiones

Para gestionar consultas o importar tablas de otra base de datos con el Explorador de JDataStore es necesario tener un archivo JDataStore abierto. Después, para definir una consulta, elija Herramientas | Importar | Tablas. Se abre el cuadro de diálogo Importar tablas:

**Figura 9.11** Primera ficha del cuadro de diálogo Importar tablas

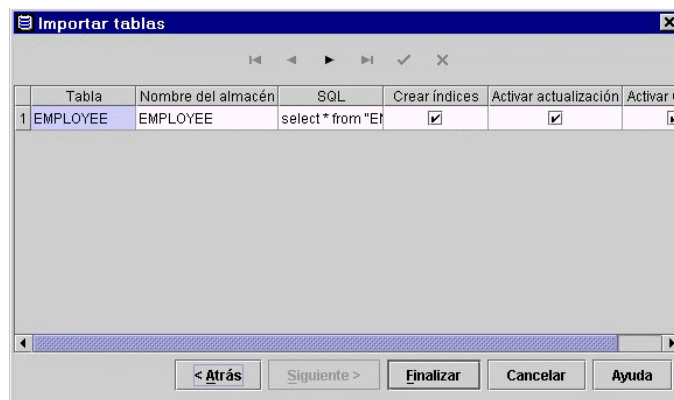


La primera vez que define una consulta, no dispone de conexiones a las que asociarla. El botón Nueva junto al campo Conexión permite definir una nueva conexión mediante el cuadro de diálogo Nueva conexión JDBC.

**Figura 9.12** Cuadro de diálogo Nueva conexión JDBC

Introduzca los mismos parámetros que introduciría en el editor de la propiedad Connection de una base de datos: URL, nombre de usuario, contraseña y nombre del controlador JDBC. También se pueden especificar propiedades ampliadas para la conexión. Cuando más tarde defina consultas, puede elegir una conexión existente o definir una nueva.

Cuando tenga una conexión a una base de datos, verá una lista de las tablas disponibles. Después de seleccionar las tablas deseadas, puede pulsar Finalizar para importarlas. Si desea tener más control sobre los datos que se importan, pulse Siguiente para pasar a la segunda ficha.

**Figura 9.13** Segunda ficha del cuadro de diálogo Importar tablas

Aquí se muestran todas las tablas que se van a importar:

- El nombre original de la tabla en la base de datos que se va a importar.
- El nombre del flujo de tabla que se va a crear en el archivo JDataStore, que es el nombre original. Esto puede cambiarse. Puede incluir una vía de acceso, como en "Datos/Tutorial/Empleado". El panel de árbol del Explorador lo muestra como un conjunto de datos denominado "Empleado" en una carpeta denominada "Tutorial" que se encuentra en una carpeta denominada "Datos".

- La sentencia SQL utilizada para recuperar los datos, que se puede modificar para cambiar los campos, las condiciones y la agrupación.
- Casillas de selección para permitir la creación de índices, y para permitir guardar o actualizar la consulta. Esta configuración se guarda en el conjunto de datos “SYS/DataStore Queries/Queries”. Si tiene tablas de sólo lectura, no deberá permitir el guardado. Si sabe que los datos de una tabla no van a cambiar, no debería permitir la actualización.

Pulse Finalizar para importar los datos y guardar las consultas.

La primera vez que se abre el cuadro de diálogo Importar tablas, se crean dos flujos de tabla vacíos, denominados “SYS/Conexiones” y “SYS/Consultas”. Las consultas creadas van a “SYS/Consultas” y las conexiones a “SYS/Conexiones”. Cuando termine de definir la primera consulta pulsando Aceptar, estas tablas tendrán una fila.

Para mantener las conexiones o consultas, seleccione la tabla “Conexiones” o “Consultas” bajo la rama “SYS/DataStore” del árbol del Explorador. Se puede:

- Ver y modificar consultas y conexiones existentes.
- Eliminar la definición de una consulta o una conexión. Para hacerlo, selecciónela y pulse “-” en la barra de herramientas de desplazamiento o pulse *Ctrl+Supr*.
- Insertar una definición. Para hacerlo, pulse “+” en la barra de herramientas de desplazamiento o pulse *Ctrl+Insert*.

## Captura y edición de datos

---

Inmediatamente después de guardar una nueva consulta, el Explorador de JDataStore intenta ejecutarla para capturar sus datos. El panel de árbol del Explorador se actualiza para mostrar la tabla recién creada, con el nombre de almacén especificado. Acto seguido, puede volver a ejecutar la consulta para actualizar manualmente los datos. Tenga en cuenta que la actualización de datos descarta los cambios no guardados.

Para ejecutar una consulta, selecciónela en la tabla “SYS/Consultas”, pulse Actualizar tabla y responda “Sí” a la advertencia referida a cambios no guardados.

Visualice una tabla seleccionándola en el árbol del Explorador. En el lado derecho del Explorador se ve la tabla en una rejilla. Seleccione la ficha Info para ver los nombres de las columnas del conjunto de datos y sus tipos de datos. Puede editar la tabla en la ficha Ver, pero sea consciente del riesgo que corre respecto de la integridad de los datos.

Tras la edición, puede almacenar los cambios o descartarlos. Los cambios se descartan actualizando el conjunto de datos y respondiendo “Sí” a la advertencia referida a cambios no guardados.



## Actualización de datos y almacenamiento de los cambios

Se pueden guardar los cambios y actualizar los datos en tres niveles diferentes:

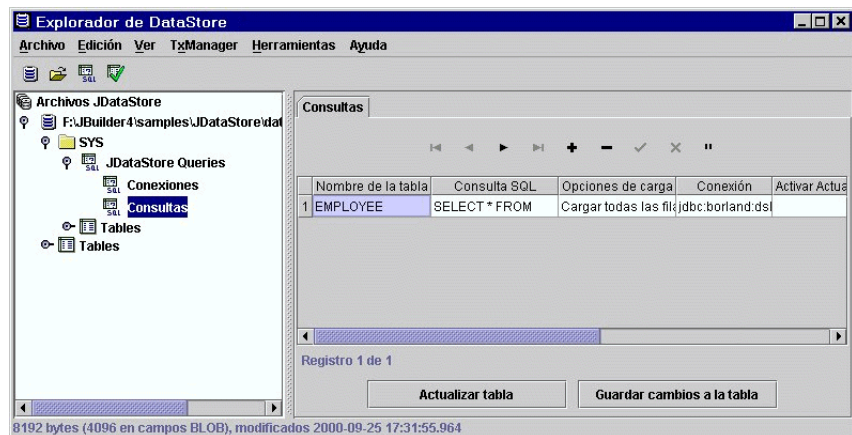
- Consultas individuales.
- Todas las consultas de una conexión determinada.
- Todas las consultas de todas las conexiones almacenadas en el JDataStore.

Para guardar o actualizar los cambios de una tabla sencilla, seleccione la fila “SYS/Consultas” para la consulta que crea esa tabla. Dispone de los botones Actualizar tabla y Guardar cambios a la tabla, que indican que sólo se verá afectada la tabla suministrada por dicha consulta.

Para actualizar o guardar cambios en todas las tablas de una conexión con una base de datos, seleccione la fila de la conexión en el conjunto de datos “SYS/Conexiones”. Dispone de los botones Actualizar consultas de conexión y Guardar cambios a la conexión, que indican que se verán afectadas todas las tablas producidas por consultas realizadas a la base de datos de dicha conexión.

Para actualizar o guardar cambios en todos los conjuntos de datos para los cuales se definieron consultas a través del Explorador de JDataStore, seleccione Herramientas | Actualizar JDataStore o bien Herramientas | Guardar cambios del JDataStore. Estos comandos vuelven a ejecutar todas las consultas o guardan los cambios de todos los conjuntos de datos con una consulta asociada, en aquellas consultas que tengan activadas las opciones Enlazar a la actualización en el menú Herramientas y Enlazar a “guardar” en el menú Herramientas. Estos valores se pueden cambiar en la tabla “SYS/Consultas”.

**Figura 9.14** Entradas de la tabla SYS/Consultas



# Importación de tablas y archivos

---

Además de importar tablas desde otras bases de datos, el Explorador de JDataStore simplifica la importación de archivos de texto delimitados como flujos de tabla y de archivos arbitrarios como flujos de archivo.

## Importar archivos de texto como tablas

---

El contenido del archivo de texto debe estar en el formato delimitado que exporta DataExpress y debe haber un archivo SCHEMA con el mismo nombre en el directorio para definir la estructura del conjunto de datos destino.

Los archivos SCHEMA (cuyo nombre termina con una extensión `.schema`) se crean al exportar un dataset a un archivo de texto con el método `com.borland.dx.TextDataFile.save()`. Es recomendable que exporte los datos desde el conjunto de datos para generar el archivo SCHEMA. Para dar una idea de cuál es su aspecto, éste es uno de un conjunto de datos de tres columnas:

```
[ ]
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,
```

Este archivo SCHEMA define la comilla doble como delimitador de cadenas y el carácter de tabulación como separador de campos. Hay tres columnas: de enteros, de cadena y de fecha y hora.

Cuando ya se tiene el archivo SCHEMA que acompaña al de texto, siga estos pasos para importar el archivo de texto como una tabla:

- 1** Seleccione Herramientas | Importar | Texto en Tabla. Se abre el cuadro de diálogo Importar archivo de texto delimitado.
- 2** Dé nombre al archivo de texto de entrada y al almacén del conjunto de datos que se crea. Dado que la operación crea un conjunto de datos, no un flujo de archivo, debería omitir la extensión del nombre del almacén.
- 3** Pulse Aceptar.

## Importación de archivos

Para importar un archivo como flujo de archivo,

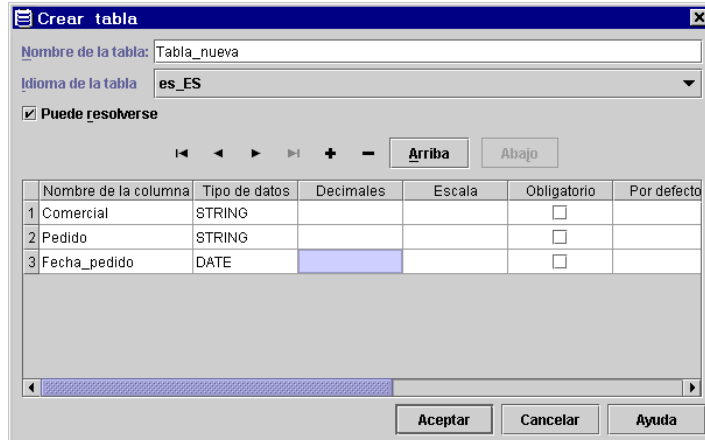
- 1 Seleccione Herramientas | Importar | Archivo.
- 2 Dé nombre al archivo de texto de entrada y al almacén del flujo de archivo que se crea.
- 3 Pulse Aceptar.


## Creación de tablas

Puede utilizar el explorador de JDataStore para crear visualmente nuevas tablas para una base de datos JDataStore.

Para crear una tabla:

- 1 Abra el Explorador de JDataStore. (Si está en JBuilder, seleccione Herramientas | Explorador de JDataStore).
- 2 Seleccione Archivo | Abrir en el Explorador de JDataStore y elija la base de datos en la cual desea crear una tabla.
- 3 Seleccione Herramientas | Crear tabla en el Explorador de JDataStore, lo cual abrirá el cuadro de diálogo Crear tabla.



- 4 Escriba un nombre para la tabla nueva en el campo Nombre de tabla.
- 5 Seleccione un idioma si desea internacionalizar la tabla. Si no, deje el valor <null>.
-  6 Haga clic en el botón Insertar nueva fila en la barra de desplazamiento para crear una nueva fila.

- 7 Haga clic en el campo Nombre de la columna y escriba el nombre de la nueva columna.
- 8 Haga clic en cada uno de los campos de propiedades de columna de la fila que desee definir y seleccione o asigne un valor. Cada columna debe contar al menos con un nombre y el tipo de datos. También se pueden especificar otras propiedades, según sea necesario. (Consulte la clase `Column` para una descripción del significado de las propiedades `Column`.)
- 9 Cree de igual manera el resto de las columnas, ordenándolas a su gusto dentro de la tabla. Utilice los botones de la barra de desplazamiento para añadir o insertar filas, desplazarse a otras y organizar las filas que ha añadido.
- 10 Haga clic en Aceptar cuando haya terminado de crear y definir todas las columnas.

El Explorador de `JDataStore` también permite modificar la estructura de las tablas. Seleccione una tabla en el árbol de la izquierda y haga clic sobre la pestaña Estructura. La interfaz de usuario de la pestaña Estructura es similar al cuadro de diálogo Crear tabla.

Si desea más información sobre cómo utilizar el cuadro de diálogo Crear tabla, haga clic sobre el botón Ayuda en el cuadro de diálogo.

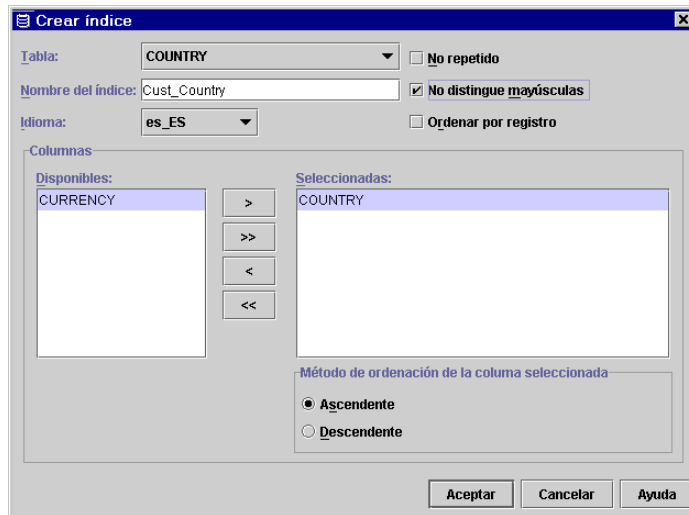
## Creación de índices

---

Puede utilizar el Explorador de `JDataStore` para crear de forma visual índices para las tablas `JDataStore`. Para crear un índice:

- 1 Abra el Explorador de `JDataStore`. (Si está en `JBuilder`, elija Herramientas | Explorador de `JDataStore`.)
- 2 Seleccione Archivo | Abrir en el Explorador de `JDataStore` y elija una base de datos.

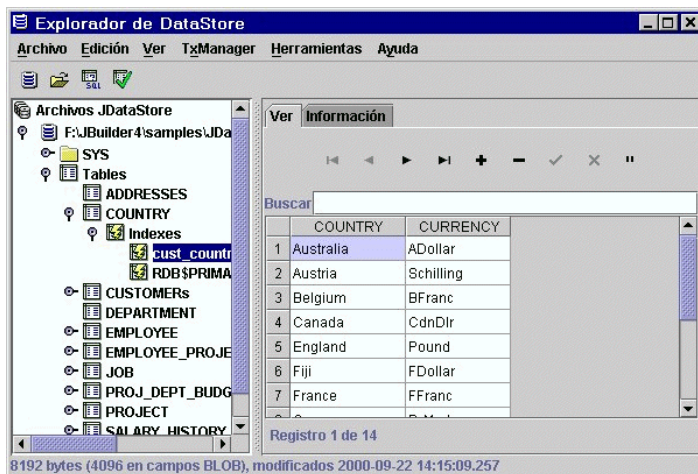
- 3 Seleccione Herramientas | Crear índice en el Explorador de JDataStore, lo que abrirá el cuadro de diálogo Crear índice.



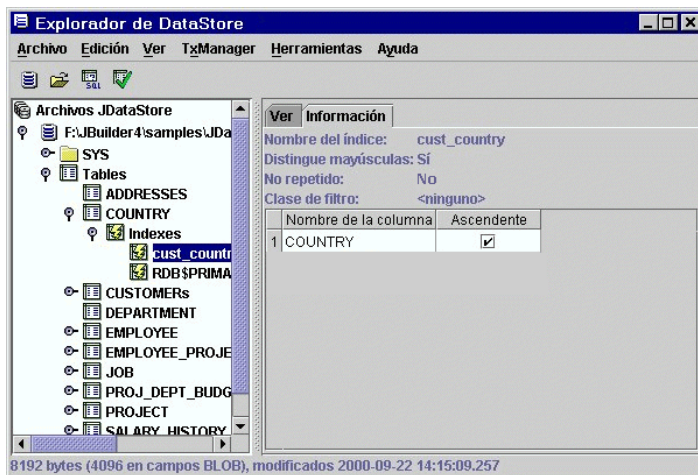
- 4 Elija la tabla para la que quiere crear el índice en la lista desplegable Tabla.
- 5 Escriba el nombre del índice en el campo Nombre del índice.
- 6 Especifique el idioma que desea utilizar para determinar cómo realizar la clasificación, si procede. Si no, deje el valor <null>.
- 7 Seleccione No repetido si desea que las filas no se repitan.
- 8 Seleccione No distingue mayúsculas si no desea tener este aspecto en cuenta.
- 9 Seleccione Ordenar por registro si desea que las filas nuevas permanezcan en la posición donde se colocaron inicialmente.
- 10 Seleccione las columnas a ordenar. Utilice los botones de flecha para moverlas de la lista Disponibles a la lista Seleccionadas y viceversa.
- 11 Especifique el orden de clasificación Ascendente o Descendente en cada columna de la lista Seleccionadas.
- 12 Cuando haya terminado de especificar los criterios de clasificación. El índice se añade a la lista de índices de esa tabla en el árbol a la izquierda del Explorador de JDataStore.

Ahora aparecen dos pestañas en el lado derecho del Explorador de JDataStore: Ver e Información.

- La pestaña Ver muestra los resultados de la clasificación mediante el índice seleccionado.



- La pestaña Información muestra las propiedades del índice seleccionado.



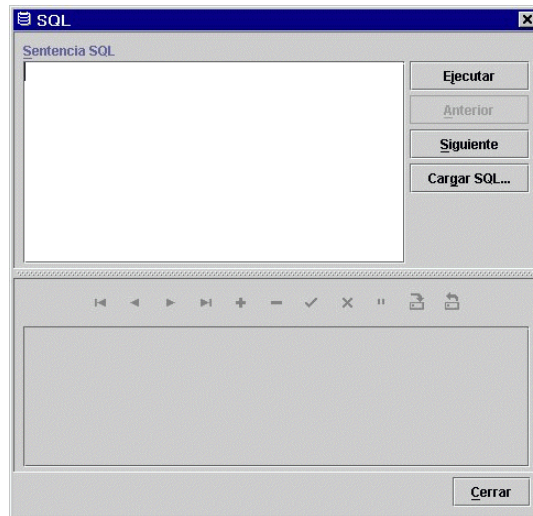
Si desea obtener más información sobre cómo utilizar el cuadro de diálogo Crear índice, haga clic sobre el botón Ayuda en el cuadro de diálogo.

## Ejecución de SQL

Es posible ejecutar sentencias SQL arbitrarias con el archivo JDataStore seleccionado como base de datos. Si el archivo JDataStore no es transaccional, sólo puede hacer consultas de sólo lectura. Si un JDataStore no transaccional no está actualmente abierto en modo de sólo lectura, se cerrará automáticamente y se volverá a abrir en modo de sólo lectura.

Para ejecutar la sentencia SQL, elija Herramientas | SQL o pulse el botón SQL de la barra de herramientas, que abre el cuadro de diálogo SQL:

**Figura 9.15** Cuadro de diálogo SQL



Se pueden escribir directamente las sentencias SQL o ejecutar archivos que las contengan. Se registran las sentencias escritas, y es posible recorrerlas con los botones Anterior y Siguiente para modificarlas y volver a ejecutarlas. El resultado de los conjuntos devueltos por las sentencias SQL se muestra en la mitad inferior del cuadro de diálogo.

## Operaciones con archivos JDataStore

---

El Explorador de JDataStore también proporciona algunas funciones que no tienen un análogo directo en la API de JDataStore.

### Empaquetado de archivos JDataStore

---

Cuando se empaqueta el archivo JDataStore cambia el nombre del archivo existente (y se antepone "BackupX\_of\_" donde X es un número que se incrementa automáticamente) y copia todos los flujos del archivo anterior a la copia nueva del archivo actual.

Para empaquetar el archivo JDataStore, elija Herramientas | Compactar JDataStore.

## Actualización de archivos JDataStore

---

El Explorador de JDataStore puede abrir versiones anteriores del formato de archivo JDataStore. La única operación disponible es actualizar el archivo a la versión actual. Cuando se actualiza el archivo JDataStore cambia el nombre del archivo existente (y se antepone “BackupX\_of\_” donde X es un número que se incrementa automáticamente) y copia todos los flujos del archivo anterior a la copia nueva del archivo actual.

Para actualizar el JDataStore, seleccione Herramientas | Actualizar JDataStore. Cuando el archivo JDataStore actual pertenece a la versión actual, esta opción de menú está desactivada.

## Eliminación de archivos JDataStore

---

Puede utilizar el Explorador de JDataStore para eliminar un archivo JDataStore y sus históricos de transacción. Para utilizar esta función, debe abrir el archivo JDataStore que desea eliminar. A continuación, elija Herramientas | Eliminar JDataStore. Se abre un cuadro de diálogo de confirmación. Pulse Sí para eliminar el archivo o archivos JDataStore.

## Tareas de seguridad de JDataStore

---

Las siguientes tareas relacionadas con la seguridad se pueden llevar a cabo mediante el Explorador de JDataStore:

### Gestión de usuarios

---

Si desea gestionar los usuarios de un JDataStore, seleccione Herramientas | Gestionar usuarios. Si no se ha definido un administrador de JDataStore anteriormente, se solicitará el nombre de usuario y la contraseña del administrador cuando se seleccione esta opción de menú. Cuando se elige un nombre de usuario y contraseña de administrador, dicho administrador se añade automáticamente y se le asignan todos los derechos sobre el JDataStore por defecto.

Si se ha conectado como usuario con derechos de administrador, aparece el cuadro de diálogo Gestionar usuarios. Si un usuario sin derechos de administrador intenta abrir este diálogo, se le pide un nombre de usuario y contraseña de administrador. Dicho cuadro de diálogo permite al administrador añadir, modificar y eliminar usuarios, así como asignar



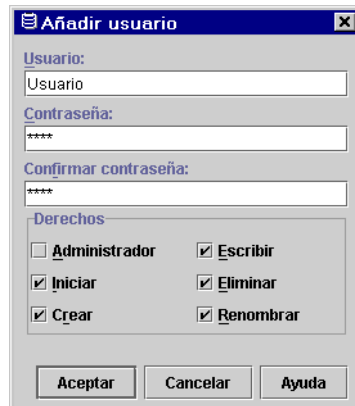
derechos a cada uno. El cuadro de diálogo Gestionar usuarios ofrece el siguiente aspecto:



Los usuarios existentes se muestran en una tabla. Las casillas de selección en las columnas de la tabla indican los derechos asignados a cada usuario. Para una explicación acerca de los distintos derechos, consulte [Autorización](#).

## Adición de usuarios

Para añadir un usuario, haga clic en el botón Añadir en el cuadro de dialogo Gestionar usuarios. A continuación aparece el cuadro de diálogo Añadir usuario, que tiene el siguiente aspecto:



Escriba un nombre para el nuevo usuario. Escriba la contraseña de usuario dos veces para confirmarla. El usuario podrá cambiar su contraseña cuando se conecte.

A continuación, seleccione los derechos que se asignan al usuario. Para una explicación acerca de los distintos derechos, consulte [Autorización](#). Haga clic en Aceptar cuando haya finalizado.

## **Modificación de los derechos de usuario**

Para cambiar de usuario, seleccione la fila del usuario en la tabla, haga clic en la botón modificar en el cuadro de dialogo Gestionar usuarios. Ya puede modificar los derechos del usuario seleccionado. El cuadro de diálogo Modificar derechos de usuario es muy similar al cuadro de diálogo Añadir usuario, con la diferencia que no se puede modificar el nombre de usuario ni la contraseña.

## **Eliminación de usuarios**

Para borrar un usuario, seleccione la fila de usuario en la tabla, haga clic en el botón Eliminar del cuadro de dialogo Gestionar usuarios. Se pedirá que confirme que desea eliminar el usuario. No se podrá eliminar un usuario si es el único administrador que existe.

## **Cambio de la contraseña**

---

Para cambiar la contraseña, el usuario debe estar conectado. Si desea cambiar la contraseña del usuario actual, seleccione Herramientas | Cambiar contraseña. A continuación aparece el cuadro de diálogo Cambiar contraseña. Escriba la contraseña antigua y, después, la nueva dos veces para confirmarla. Pulse Aceptar.

## **Encriptación de un JDataStore**

---

Si desea encriptar un archivo JDataStore, seleccione Encriptar JDataStore del menú Herramientas. El Explorador de JDataStore intenta encriptar el JDataStore. Un mensaje indica si lo ha logrado o no. Si el JDataStore logra realizar la encriptación, se efectuará una copia de seguridad del archivo original y aparecerá un mensaje en el que conste el nombre de la copia.

# Optimización de aplicaciones JDataStore

En este capítulo se trata una serie de temas que ayuda a mejorar el rendimiento, la fiabilidad y el tamaño de las aplicaciones JDataStore. Si no se indica lo contrario, `DataStoreConnection` se refiere tanto a un objeto `DataStoreConnection` como a un objeto `DataStore`, utilizados para abrir una conexión con un archivo JDataStore.

## Carga rápida de bases de datos

---

A continuación, se indican algunas sugerencias que pueden mejorar el rendimiento de la aplicación al cargar bases de datos:

- Utilice sentencias preparadas siempre que sea posible. Si el número de parámetros cambia entre una inserción y la siguiente, realice una llamada al método `PreparedStatement.clearParameters()` antes de configurar los parámetros `PreparedStatement`.
- Utilice el método `TableDataSet.addRow()` de `DataExpress`, que resulta ligeramente más rápido que las sentencias preparadas de `JDBC`. Deberá asignar al almacén de `TableDataSet` un `DataStoreConnection` y asignar a la propiedad `StoreName` el nombre de la tabla de la base de datos JDataStore.
- Utilice el método `TextDataFile` de `DataExpress` para importar archivos de texto. Dispone de un analizador rápido y sabe cómo cargar los datos rápidamente. Deberá asignar al almacén de `TableDataSet` un `DataStoreConnection` y asignar a la propiedad `StoreName` el nombre de la tabla de la base de datos JDataStore.

- Cuando cargue una nueva base de datos, cree primero la base de datos como no transaccional. Cargue la base de datos no transaccional mediante el método `TableDataSet.addRow()` de `DataExpress` o el componente `TextDataFile`. Tras completar la carga, convierta la base de datos en transaccional mediante la propiedad `DataStore.TxManager`. Esta técnica debería bastar para acelerar la operación de carga de dos a tres veces.

## Recomendaciones generales

---

Estas recomendaciones son validas para todos los tipos de aplicaciones `JDataStore`.

### Cierre del archivo `JDataStore`

---

No olvide hacer una llamada a `DataStoreConnection.close()` cuando `DataStoreConnection` deje de ser necesario o llame a `DataStore.shutdown()` para cerrar el archivo `JDataStore` independientemente del número de conexiones que haya. En particular:

- Para garantizar el cierre apropiado, no llame a `System.exit()` antes que a `DataStore.shutdown()`.

El cierre de un `JDataStore` garantiza que todas las modificaciones se han guardado en el disco. Existe un hilo de utilidad para todas las instancias abiertas de `DataStoreConnection` que guarda continuamente los datos modificados de la memoria caché. (Por defecto, los datos modificados se guardan cada 100 milésimas de segundo). Si se sale directamente de la máquina virtual de Java, es posible que el hilo de utilidad no tenga la oportunidad de guardar los últimos cambios. Existe la posibilidad de que los archivos `JDataStore` no transaccionales se dañen.

Los archivos `JDataStore` transaccionales no pueden perder datos, pero el administrador de transacciones cancela todos los cambios sin confirmar.

Si un archivo `JDataStore` se cierra de forma incorrecta, se tardan varios segundos en reiniciarlo o recuperarlo. Si el cierre se efectúa correctamente, el archivo `JDataStore` se abre siempre rápidamente.

Otra ventaja de cerrar los objetos `DataStoreConnection` es que cuando todos están cerrados, se libera la memoria asignada a la caché.

Cierre todos los objetos `StorageDataSet` cuya propiedad `store` tenga el valor `DataStoreConnection` cuando haya terminado con ellos. De esta forma se liberan los recursos de `JDataStore` asociados a `StorageDataSet` y permite que se libere la memoria no utilizada de `StorageDataSet`.

## Optimización de la caché de disco de JDataStore

---

El tamaño de caché máximo por defecto para JDataStore es de 512 bloques. El bloque por defecto es de 4096 bytes. Por lo tanto, la memoria caché alcanza su capacidad máxima a unos 512\*4096 bytes (2 MB). Observe que esta memoria se asigna según se va necesitando. En algunas ocasiones especiales, cuando se utilizan todos los bloques, la memoria caché puede crecer por encima de los 512 bloques. El tamaño mínimo para la caché se puede definir en la propiedad `DataStore.MinCacheSize`.

**Nota** No cambie el tamaño de la caché de JDataStore de forma arbitraria. Compruebe de antemano que eso mejorará el rendimiento de la aplicación.

A la hora de cambiar la caché de JDataStore, tenga en cuenta las siguientes consideraciones:

- El almacenamiento en caché que utilizan los sistemas operativos modernos ofrece un gran rendimiento. En muchas ocasiones, incrementar el tamaño de caché de JDataStore no mejora significativamente el rendimiento y sí consume más memoria.
- Dependiendo de la máquina virtual de Java utilizada, la asignación de grandes cantidades de memoria podría reducir el rendimiento de las operaciones de liberación de memoria no utilizada de la máquina virtual.
- Sólo existe una caché de disco de JDataStore para todas las bases de datos JDataStore abiertas en el mismo proceso. Al cerrar todas las bases de datos JDataStore, la memoria para esta caché global de disco se libera.
- Para dispositivos de mano con poca memoria, asigne a la propiedad `DataStore.MinCacheSize` un valor más pequeño, como 96.

## Optimización de ubicaciones de archivos

---

Las bases de datos JDataStore desarrollan la mayoría de las operaciones de lectura y escritura contra los cuatro siguientes tipos de archivos:

- El propio archivo de base de datos de JDataStore (la extensión de archivo es `.jds`), según se especifica en la propiedad `DataStore.FileName`.
- Archivos de registro de transacciones de JDataStore. Son los nombres de archivos de registro que terminan con la extensión `LOGAnnnnnnnnnnn`, donde 'n' es un dígito numérico según se especifica en la propiedad `TxManager.ALogDir`.
- Archivos temporales utilizados para grandes operaciones de ordenación según se especifica en la propiedad `DataStore.TempDirName`.

- Archivo temporal `.jds` utilizado para resultados de consultas SQL según se especifica en la propiedad `DataStore.TempDirName`.

El rendimiento puede mejorarse indicando a `JDataStore` que coloque los archivos mencionados anteriormente en diferentes unidades de disco.

A continuación, se indican algunas otras sugerencias sobre almacenamiento de archivos que pueden ayudar a mejorar el rendimiento de la aplicación:

- Es especialmente importante colocar los archivos de registro en una unidad de disco independiente. Observe que el archivo de registro se escribe generalmente en orden secuencial y su contenido se debe volcar al disco para poder completar las operaciones de confirmación. Por ello, resulta favorable disponer de una unidad de disco que pueda completar las operaciones de escritura rápidamente.
- En Windows NT, se ha observado que el rendimiento se puede mejorar colocando los archivos de registro de `JDataStore` en un directorio independiente y que almacenar numerosos archivos distintos de los de registro en el directorio puede ralentizar el rendimiento de las operaciones de confirmación. (Observe que esta sugerencia de rendimiento también se puede aplicar a plataformas distintas de Windows NT.)
- Acuérdesse de desfragmentar los sistemas de archivo de la unidad de disco periódicamente. Esta práctica es especialmente importante para la unidad de disco que almacena los archivos de registro, ya que `JDataStore` realiza una gran cantidad de operaciones secuenciales de lectura y escritura en estos archivos.

## Control de la frecuencia de la escritura en disco de los bloques de la memoria caché

---

La propiedad `saveMode` del componente `DataStore` permite controlar la frecuencia con la que los bloques de la memoria caché se escriben en el disco. Esta propiedad sólo es aplicable a los archivos `JDataStore` no transaccionales. El método puede tener los siguientes valores:

- **0:** Hace que el hilo de utilidad gestione toda la escritura de caché. Esta configuración aumenta al máximo el rendimiento, pero el riesgo de deterioro también es mayor.
- **1:** Guarda inmediatamente cuando se añaden o se borran bloques, el hilo de utilidad se encarga de los otros cambios. Éste es el modo establecido por defecto. El rendimiento es casi tan bueno como el de `saveMode(0)`.
- **2:** Guarda inmediatamente todos los cambios. Esta configuración es útil para depurar aplicaciones que utilizan un archivo `DataStore`.

A diferencia de otras propiedades de `DataStore`, `saveMode` se puede modificar cuando la conexión está abierta. Por ejemplo, si se utiliza `DataStoreConnection` se puede acceder al valor por medio de la propiedad `dataStore`:

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

Esto modifica el comportamiento de todos los objetos `DataStoreConnection` que acceden a este archivo `JDataStore`.

## Ajuste de la memoria

---

El uso de memoria se puede ajustar de varias formas. Tenga en cuenta que pedir demasiada memoria puede ser tan malo como tener muy poca.

- La memoria dinámica de Java se resiste a superar su tamaño original, por lo que se obliga a liberarse frecuentemente de los recursos no utilizados con un espacio libre cada vez menor. La opción `-Xms` permite establecer un tamaño de memoria dinámica mayor desde el principio.
- Se puede probar a aumentar el valor de la propiedad `DataStore.minCacheBlocks` que controla el número mínimo de bloques que se guarda en la memoria caché.
- `DataStore.maxSortBuffer` controla el tamaño máximo del búfer que se utiliza para ordenar la memoria. Las ordenaciones que sobrepasan este tamaño de búfer utilizan un orden más lento, por medio del disco.

## Consejos para mejorar el rendimiento

---

A continuación se dan algunos consejos para mejorar el rendimiento:

- Asigne a la propiedad `DataStore.tempDirName`, utilizada por el motor de consulta, un directorio de otra unidad más rápida.
- Pruebe a asignar a la propiedad `TxManager.checkFrequency` un valor más alto. Los valores altos tienen como resultado una recuperación más lenta, pero no hay motivos para alarmarse.
- Para las operaciones sencillas, `DataExpress` puede ser más rápido que `JDBC/SQL`. Sin embargo, es probable que `JDBC/SQL` sea más rápido para las consultas complejas.
- Cuando escriba sus aplicaciones, tenga en cuenta el aumento de eficacia que proporciona el multihilo. Reduzca al mínimo el número de objetos de monitor, para aprovechar los sistemas con varios procesadores.

## Componentes adicionales de JDataStore

---

La biblioteca de componentes dbSwing proporciona dos componentes (en la ficha Más dbSwing de la paleta de componentes) que hace más fácil la creación de aplicaciones JDataStore estables.

- `DBDisposeMonitor` (que desecha automáticamente los recursos de componentes enlazados a datos cuando se cierra un contenedor) posee una propiedad `closeDataStores`. Cuando su valor es **true** (predefinido), cierra automáticamente los archivos JDataStore enlazados a los componentes que limpia.

Por ejemplo, si se coloca un objeto `DBDisposeMonitor` en un `JFrame` que tiene componentes dbSwing enlazados a un archivo JDataStore, cuando se cierra `JFrame`, `DBDisposeMonitor`, cierra el archivo automáticamente. Este componente resulta muy útil en la creación de aplicaciones sencillas para experimentar con JDataStore.

- `DBExceptionHandler` tiene un botón Salir. Este botón se puede ocultar modificando su configuración, pero es visible por defecto. Hacer clic en este botón cierra automáticamente cualquier JDataStore que pueda encontrar. `DBExceptionHandler` es el cuadro de diálogo que presentan por defecto los componentes dbSwing cuando se produce una excepción.

## Módulos de datos

---

Al utilizar un JDataStore con un `StorageDataSet`, debería siempre agruparlos dentro de los módulos de datos. Haga referencia a estos `StorageDataSet` con los métodos de acceso `DataModule` como `businessModule.getCustomer()`. La razón es que gran parte de la funcionalidad ofrecida a través de los `StorageDataSet` depende de las configuraciones de los sucesos y de las propiedades. Aunque la mayor parte de las más importantes propiedades estructurales de `StorageDataSet` persisten en el directorio JDataStore, no sucede lo mismo con las clases que implementan las interfaces de monitores de sucesos. Si se instancia el `StorageDataSet` con todas las configuraciones del monitor de sucesos, las restricciones, los campos calculados y los filtros implementados con los sucesos, se mantendrán correctamente tanto en fase de diseño como durante la ejecución.



# Optimización de aplicaciones transaccionales

---

El aumento de la fiabilidad y la flexibilidad que proporcionan los archivos JDataStore transaccionales deteriora el rendimiento. Esto se puede paliar de varias formas.

## Transacciones de sólo lectura

---

Cuando se leen los datos pero no se escriben se puede aumentar considerablemente el rendimiento por medio de las transacciones de sólo lectura. La propiedad `readOnlyTx` de `DataStoreConnection` controla si una transacción es de sólo lectura. Se debe asignar el valor **true** a esta propiedad antes de que se abra la conexión. Para las conexiones de JDBC, éste está controlado por la propiedad `readOnly` del objeto

```
java.sql.Connection (devuelta por los métodos  
java.sql.DriverManager.getConnection() y  
com.borland.dx.dataset.sql.Database.getJdbcConnection()).
```

Las transacciones de sólo lectura simulan una instantánea del archivo JDataStore. En esta instantánea sólo aparecen los datos de las transacciones confirmadas en el momento de su inicio (de lo contrario, la conexión debe comprobar si hay cambios pendientes y cancelarlos cuando accede a los datos). Cuando se abre la conexión `DataStoreConnection` se toma una instantánea, que se actualiza cada vez que se llama a su método `commit()`.

Otra ventaja de las transacciones de sólo lectura es que las demás transacciones no las detienen. La escritura y la lectura requieren normalmente un bloqueo de flujo, Pero dado que una transacción de sólo lectura utiliza una instantánea, no necesita bloqueo. Por lo tanto, no se verá afectada por una transacción de escritura que tenga un bloqueo en el flujo que está modificando.

Es posible mejorar más aún el funcionamiento de la aplicación configurando `readOnlyTxDelay`. El valor por defecto de esta propiedad es cero, lo que significa que cuando se inicia o se actualiza una transacción de sólo lectura se toma una nueva instantánea. La propiedad `readOnlyTxDelay` establece la duración máxima en milésimas de segundo para las instantáneas que puede compartir la conexión. Si el valor de esta propiedad es distinto de cero, las instantáneas se buscan desde la más reciente hasta la más antigua. Si se ha tomado alguna propiedad después del momento establecido en `readOnlyTxDelay`, se utiliza y no se toma otra.

## Modo de confirmación por software

---

Si se activa el modo de confirmación por software por medio de la propiedad `softCommit` de `TxManager`, el administrador de transacciones fuerza la escritura en disco con vistas a la recuperación de errores, pero no garantiza la confirmación de las transacciones. Esto mejora el rendimiento, pero aumenta las probabilidades de perder datos en caso de corte del suministro eléctrico o fallo del sistema operativo o el hardware.

No se garantiza la escritura de las transacciones confirmadas recientemente. El plazo depende del sistema operativo, pero por lo general es de un segundo. Esto no ocurre cuando el modo de confirmación por software se encuentra desactivado (la opción por defecto), ya que los cambios enviados se escriben inmediatamente.

## Históricos de transacciones

---

Las propiedades `ALogDir` y `BLogDir` de `TxManager` controlan la ubicación de los archivos históricos de transacciones.

- Los archivos no se deben colocar en unidades lentas, como las de red. (Tampoco es recomendable colocarlos en disquetes).
- Puede duplicar los archivos de históricos especificando una propiedad `BLogDir` además de una propiedad `ALogDir` para aumentar la fiabilidad. Si una de las copias se daña o se pierde, se conserva la otra.
- Es aconsejable elegir para `BLogDir` una unidad de disco distinta de la de `ALogDir`. Esto puede reducir los problemas de rendimiento que ocasiona la escritura de dos históricos independientes (las dos unidades pueden escribir simultáneamente), y aumenta la fiabilidad porque es poco probable que fallen los dos discos.

## Desactivación del registro de estado

También es posible aumentar el rendimiento desactivando el registro de los mensajes de estado. Para ello, asigne a la propiedad `recordStatus` de `TxManager` el valor `false`.

## Optimización del rendimiento del control de concurrencia de JDataStore

---

Utilice las siguientes directrices para optimizar el rendimiento de las operaciones de control de concurrencia (conflictos) de `JDataStore`:

- Elija el nivel de aislamiento más débil con el que la aplicación pueda funcionar correctamente. Los niveles de aislamiento más bajos tienden a adquirir bloqueos más débiles y escasos.

- Configure la propiedad `lockTableTables` explicada anteriormente para tablas que se actualizan con poca frecuencia. Existe un menor espacio adicional de datos para gestionar bloqueos de tabla.
- Asigne a la propiedad `java.sql.Connection.setAutoCommit()` el valor `false` para agrupar varias operaciones en una única transacción. La propiedad `java.sql.Connection.commit()/rollback()` se puede utilizar para terminar transacciones.
- Confirme las transacciones tan pronto como sea posible. La mayoría de los bloqueos no se liberan hasta que una transacción se confirma o se cancela.
- Reutilice `java.sql.Statements` siempre que sea posible y utilice `java.sql.PreparedStatements` cuando sea posible.
- Cierre todos los objetos `Statements`, `PreparedStatements`, `ResultSets` y `Connections` cuando ya no se necesiten. Observe que los objetos individuales direccionales `ResultSets` se cierran automáticamente después de leer la última fila.
- Utilice transacciones de solo lectura para informes largos y operaciones de copia de seguridad en línea (utilice el método `com.borland.datastore.DataStoreConnection.copyStreams()` para copias de seguridad en línea). Las transacciones de sólo lectura proporcionan una vista de sólo lectura transaccionalmente congruente (serializable) de las tablas a las que acceden. No obtienen bloqueos, por tanto no se producen conflictos ni expiración de los tiempos de espera de los bloqueos. Esta propiedad se puede definir mediante el método `java.sql.Connection.setReadOnly()`, para conexiones JDBC, o el método `com.borland.datastore.DataStoreConnection.setReadOnlyTx()`, para componentes `DataExpress`.
- Se necesita cierto espacio de datos adicional para mantener una vista de sólo lectura. En consecuencia, varias transacciones pueden compartir la misma vista de sólo lectura. La propiedad `readOnlyTxDelay` especifica cuánto tiempo se puede mantener la vista de sólo lectura cuando se inicia una transacción de este tipo. Confirme una transacción de conexión de sólo lectura para actualizar la vista de la base de datos. Observe que una transacción de sólo lectura utiliza los históricos de transacciones para mantener sus vistas. En consecuencia, las conexiones de sólo lectura se deberían cerrar cuando ya no se necesiten.

## Las operaciones multihilo

---

La producción de transacciones de escritura se puede ver incrementada a medida que se utilizan más hilos de proceso para realizar operaciones, ya que cada hilo puede incluir los datos adicionales de las operaciones de confirmación procedentes de la capacidad de “confirmación de grupo” suministrada por `JDataStore`.

## Depuración de recursos publicados

---

Cuando se distribuye una aplicación JDataStore es posible excluir determinados archivos de clases y gráficos que no se utilizan. En particular:

- Si JDataStore se utiliza sin capacidad de transacción, excluya esta clase:
  - `com.borland.datastore.Tx*.class`
- Si JDataStore se utiliza sin el controlador JDBC, excluya estas clases:
  - `com.borland.datastore.Sql*.class`
  - `com.borland.jdbc.*`
  - `com.borland.sql.*`
- Si se utiliza DataExpress y la propiedad `StorageDataSet.store` tiene asignada una instancia de `DataStore` o `DataStoreConnection`, excluya la clase:
  - `com.borland.dx.memorystore.*`
- Si se utiliza `TableDataSet`, pero no `QueryDataSet`, `QueryProvider`, `StoredProcedureDataSet` ni `StoredProcedureProvider`, excluya la clase:
  - `com.borland.dx.sql.*`
- Si DataExpress no utiliza componentes visuales de las bibliotecas JBCL y dbSwing, excluya la clase:
  - `com.borland.dx.text.*`
- Si no se utiliza `com.borland.dx.dataset.TextDataFile`, excluya estas clases:
  - `com.borland.jb.io.*`
  - `com.borland.dx.dataset.TextDataFile.class`
  - `com.borland.dx.dataset.SchemaFile.class`

## Columnas con autoincremento

---

Esta versión de JDataStore es compatible con una función denominada columnas con autoincremento. En concreto, las columnas de tipo `int` y `long` pueden especificarse de modo que presenten valores con autoincremento.

Estas propiedades se aplican a todos los valores de las columnas con autoincremento:

- Son siempre únicos.
- Nunca pueden ser nulos.
- Los valores procedentes de filas borradas no se pueden volver a utilizar.

Estas propiedades hacen que las columnas con autoincremento sean ideales para claves principales de tipo `integer` o `long` formadas por una sola columna.

Una columna con autoincremento proporciona la vía de acceso aleatorio más rápida a una determinada fila de una tabla `JDataStore`, ya que constituye el valor de “fila interna” para una fila.

**Nota** Una tabla sólo puede contener una columna con autoincremento.

Una columna con autoincremento permite ahorrar el espacio de la columna entera y el índice secundario que se necesitarían para actuar como clave principal. El optimizador de consultas de `JDataStore` optimiza las consultas que hacen referencia a una columna con autoincremento en una cláusula “where”.

## Columnas con autoincremento en DataExpress

---

Para crear una tabla con una columna con autoincremento en `DataExpress`, asigne a la propiedad `Column.AutoIncrement` el valor `true` antes de abrir una tabla. Si va a modificar una tabla existente, deberá realizar una llamada al método `StorageDataSet.restructure()`. Para obtener más información, consulte [Creación de tablas de JDataStore con DataExpress](#).

## Columnas con autoincremento en SQL

---

Si desea crear o modificar una tabla con columnas con autoincremento en `SQL`, consulte [Referencia de SQL](#).





## Especificaciones

Las siguientes especificaciones son aplicables a la versión 6.0 del formato de archivo JDataStore.

### Capacidad del archivo JDataStore

---

Tamaño mínimo de bloque: 1 KB.

Tamaño máximo de bloque: 32 KB.

Tamaño de bloque por defecto: 4 KB.

Tamaño máximo del archivo JDataStore: 2.000 millones de bloques. Para el tamaño de bloque por defecto, esto equivale a un máximo de 8,796,093,022,208 bytes.

Número máximo de filas por flujo de tabla: 4.000 millones.

Longitud máxima de fila: 1/3 del tamaño del bloque. Las cadenas, objetos y flujos de entrada que sobrepasen el tamaño de línea (64 bytes por defecto) se almacenan como BLOB (objetos grandes binarios) para no ocupar espacio en la fila.

Tamaño máximo de BLOB: 2 GB cada uno.

Tamaño máximo del flujo de archivo: 2 GB cada uno.

## Nombres de flujo de JDataStore

---

Carácter de separación de directorios: /

Longitud máxima: 192 bytes.

- Equivalencia con caracteres de un solo byte: 192 caracteres.
- Equivalencia con caracteres de dos bytes: 95 (se pierde un byte para indicar el juego de caracteres de dos bytes).

Nombres reservados:

- SYS/Conexiones
- SYS/Consultas
- SYS/Usuarios



# Resolución de problemas

He aquí algunos consejos para cuando surjan problemas.

## Depuración de aplicaciones JDataStore

---

Para depurar archivos JDataStore no transaccionales, asigne a la propiedad `saveMode` el valor 2. El depurador detiene todos los hilos cuando se avanza línea a línea por el código y cuando se alcanza un punto de interrupción. Si el valor de la propiedad `saveMode` no es 2, el hilo de utilidad de JDataStore no guarda los datos modificados de la memoria caché. Para obtener más información, consulte [Control de la frecuencia de la escritura en disco de los bloques de la memoria caché](#).

## Comprobación del contenido de JDataStore

---

Si existen motivos para pensar que es posible que el contenido de la memoria caché no se guarda correctamente en un JDataStore no transaccional, se puede comprobar la integridad del archivo con el Explorador de JDataStore. Consulte [Comprobación del JDataStore](#) para obtener más información.

También existe una clase denominada `borland.datastore.StreamVerifier` con métodos públicos estáticos `verify()` que puede verificar uno o todos los flujos de JDataStore. Para obtener más información, consulte *DataExpress Component Library Reference*.

Se debe tener en cuenta que todos los JDataStore transaccionales cuentan con recuperación automática por detención del sistema cuando se abren. No es necesario comprobarlos.

Si tiene problemas, utilice el Explorador de JDataStore (consulte [Copia de flujos JDataStore](#)) o el método `DataStoreConnection.copyStreams()` para reparar los daños.

## Problemas en la búsqueda y la ordenación de datos

---

Sun Microsystems cambia periódicamente sus clases `java.text.CollationKey` para solucionar los problemas. Los índices secundarios de las tablas guardadas en un JDataStore utilizan estas clases `CollationKey` para generar claves de clasificación si se utiliza una configuración local distinta a la estadounidense. En ocasiones, cuando Sun cambia el formato de estas clases `CollationKeys`, los índices secundarios creados por los JDK de Sun antiguos pueden no funcionar correctamente con los nuevos. Los problemas provocados por esta situación se ponen de manifiesto de las siguientes formas:

- Las operaciones de búsqueda y consulta pueden no encontrar registros que cumplen los requisitos.
- Cuando se muestra una tabla ordenada por el índice secundario (configurando la propiedad `StorageDataSet.sort`) es posible que el orden no sea el correcto.

Por el momento, la única forma de corregir estos problemas consiste en desechar los índices secundarios y volver a crearlos con el JDK actual. El método `StorageDataSet.restructure` también desecha todos los índices secundarios.

## Almacenamiento de históricos

---

Dado que los históricos antiguos ya no son necesarios para las transacciones activas ni para la recuperación ante las detenciones del sistema, se borran automáticamente. Es posible guardar los históricos monitorizando el suceso `DataStore.response` en espera de una notificación `ResponseEvent.DROP_LOG`. Entonces es posible copiar el histórico en otro lugar antes de que se borre o puede introducir el método `cancel()` para cancelar el suceso y evitar que se borre.



## Referencia de SQL

### Acceso a SQL mediante JDBC

JDataStore incorpora un controlador JDBC. Utilice los siguientes métodos para ejecutar una sentencia SQL:

Desde `Statement.java`:

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

Desde `Connection.java`:

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

Desde `PreparedStatement.java` y `CallableStatement.java`:

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

Cada cadena de consulta debe contener exactamente una cadena SQL.

Utilice `executeQuery` para las sentencias que devuelvan `resultSet`. Por ejemplo:

```
SELECT * FROM EMPLOYEE
```

Utilice `executeUpdate` para las sentencias que no lo devuelvan:

```
CREATE TABLE MYTABLE (COLUMN1 INT, LAST_NAME VARCHAR(20))
INSERT INTO MYTABLE VALUES (1, 'Overbek')
UPDATE MYTABLE SET LAST_NAME='Overbeck' WHERE COLUMN1=1
```

Observe que `executeUpdate` lanza una excepción si la sentencia ejecutada produce en realidad un `resultSet`.

Si la sentencia contiene parámetros de salida, se debe utilizar un `CallableStatement`.

Si una sentencia contiene parámetros de entrada, se debe utilizar un `PreparedStatement` o un `CallableStatement`.

## Tipos de datos

En SQL, estos tipos se deben establecer por medio de estos nombres `JDataStore` o sinónimos, que son más fáciles de transportar a otros motores SQL. Los tipos posibles y sus sinónimos se establecen en la tabla siguiente:

Tipo de datos de <code>JDataStore</code>	Descripción	Bytes	Sinónimos
SMALLINT	Número exacto con el rango: -32768..32767	1-3	SHORT
INT	Número exacto con el rango: -2147483648..2147483647	1-5	INTEGER
BIGINT	Número exacto con el rango: -9223372036854775808..9223372036854775807	1-9	LONG
DECIMAL(p,d)	Número exacto con una precisión de p dígitos y d decimales. La precisión se limita a 72 dígitos. Si se omite, el valor por defecto para p es 72 y para d es 0.	1-32	BIGDECIMAL
REAL	Número aproximado con el rango: 1.4E-45...3.4E38 y una precisión de 23 bits.	1-5	
DOUBLE	Número aproximado con el rango: 4.9E-324...1.8E308 y una precisión de 52 bits.	1-9	DOUBLE_PRECISION
FLOAT(p)	Número aproximado con una precisión de al menos p bits donde $p \leq 52$ . Si se omite, el valor por defecto para p es 52.	1-9	

VARCHAR(p,m)	<p>Cadena de longitud variable con una longitud máxima de p caracteres y un inline máximo de m bytes.</p> <p>Si la cadena contiene más de m bytes, el resto de la cadena se guarda como un BLOB.</p> <p>Si se omite, el valor por defecto para p es ilimitado, y para m es 64.</p>	1-m	STRING
VARBINARY(p,m)	<p>Binario de longitud variable con una longitud máxima de p bytes y un inline máximo de m bytes.</p> <p>Si los datos contienen más de m bytes, el resto de los datos se guardan como un BLOB.</p> <p>Si se omite, el valor por defecto para p es ilimitado, y para m es 64.</p>	1-m	INPUTSTREAM BINARY
OBJECT(t,m)	<p>Binario de longitud variable que contiene un objeto Java de tipo t y un inline máximo m.</p> <p>El objeto Java se guarda mediante la serialización Java.</p> <p>Si se omite, se puede guardar cualquier tipo, y m es 64.</p>	1-m	
BOOLEAN	Tipo de dos valores: true/false	1	BIT
DATE	Una fecha local posterior al año 0.	1-9	
TIME	Una hora local dentro del rango: 00:00:00..23:59:59.	1-9	
TIMESTAMP	Una marca de fecha y hora GMT. Al contrario que DATE y TIME, el valor depende de la zona horaria del equipo en el que se lee. TIMESTAMP puede contener fechas anteriores al año 0.	1-9	

Ejemplos

- Las cadenas con un tamaño máximo de 30 bytes y todas las cadenas que tengan más de 10 bytes se almacenan en un flujo independiente para objetos grandes:  
`VARCHAR(30,10)`
- Las cadenas con un tamaño máximo de 30 bytes no tienen nunca una longitud de línea establecida (la precisión es menor que el valor de longitud por defecto, de 64):  
`VARCHAR(30)`
- Las cadenas sin límite de longitud utilizan el tamaño por defecto:  
`VARCHAR`
- Tipo `BigDecimal` sin decimales, con espacio para 5 dígitos significativos por lo menos:  
`DECIMAL(5,2)`
- Tipo `BigDecimal` sin decimales, con espacio para 4 dígitos significativos por lo menos:  
`NUMERIC(4)`
- Tipo `BigDecimal` sin decimales, con espacio para 72 dígitos significativos por lo menos:  
`NUMERIC`
- Cualquier objeto Java que sea serializable:  
`OBJECT`
- Sólo las cadenas Java utilizan serialización de Java:  
`OBJECT('java.lang.String')`

Literales

En la tabla siguiente se enumeran los tipos de valores literales escalares aceptados:

Tipo de datos de <code>JDataStore</code>	Ejemplos de literales	Descripción
<code>SMALLINT</code> <code>INT</code> <code>BIGINT</code> <code>DECIMAL(p,d)</code>	8 2. 15.7 .9233	Los números exactos deben contener una coma decimal.
<code>REAL</code> <code>DOUBLE</code> <code>FLOAT(p)</code>	8E0 4E3 0.3E2 6.2E-72	El número aproximado es un número seguido por la letra E, seguida por un entero que puede llevar signo

VARCHAR(p,m)	'Hola' 'don't' (negación en inglés)	Las cadenas se encuentran entre comillas sencillas. El carácter de comilla sencilla (apóstrofo) se representa por medio de dos comillas sencillas consecutivas. El apóstrofo no es de uso común en español, pero sí en otros idiomas
VARBINARY(p,m)	B'1011001' X'F08A' X'f777'	Secuencia binaria o hexadecimal entre comillas sencillas, precedida por la letra B o X
OBJECT(t,m)		No hay literales de objetos en SQL de JDataStore.
BOOLEAN	TRUE FALSE	
DATE	DATE '2002-06-17'	Muestra la hora local del origen; el formato es DATE 'aaaa-mm-dd'
TIME	TIME '15:46:55'	Muestra la hora local del origen; el formato es de 24 horas, TIME 'hh:mm:ss'
TIMESTAMP	TIMESTAMP '2001-12-31 13:15:45'	El formato es TIMESTAMP aaaa-mm-dd hh:mm:ss'

## Secuencias de escape JDBC

JDataStore admite secuencias de escape de procedimiento de JDBC para:

- especificar los literales de fecha y hora, el carácter de escape para una cláusula LIKE
- OUTER JOINS
- insertar el resultado de la cadena o de las funciones de fecha y hora en la sentencia SQL
- llamar a procedimientos almacenados

Las secuencias de escape de JDBC siempre aparecen entre llaves {}. Se usan para ampliar el funcionamiento de SQL.

## Ejemplos

### Literales de fecha y hora:

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

Indica una hora, que debe escribirse en el siguiente orden: horas, minutos y, a continuación, segundos.

```
{ D 'mm-dd-aa' }
```

Indica una fecha que debe escribirse en el formato indicado; mes, día, y año.

```
{ TS 'mm-dd-aa : hh-mm-ss' }
```

Indica una marca temporal que debe ser introducida en el formato indicado; mes, día, año, dos puntos, hora, minutos, segundos.

### Expresiones OUTER JOIN:

```
{ OJ <expresión de unión de tabla> }
```

Una unión externa se realiza en la expresión de tabla indicada.

### Carácter de escape para LIKE:

```
{ ESCAPE <char> }
```

El carácter indicado se convierte en el carácter de escape en la precedente cláusula LIKE.

## Funciones de escape

---

Las funciones se escriben en el siguiente formato:

```
{ FN <expresión de función de escape> }
```

FN indica que la función siguiente debe ser realizarse.

## Funciones de cadena

---

CONCAT(string1, string2) concatena dos cadenas

LCASE(string) devuelve la cadena en minúsculas

LENGTH(string) devuelve la longitud de la cadena

LOCATE(string1, string1 [, pos] ) busca string1 en string2, comenzando en la posición pos de string2

LTRIM(string) // quita los espacios iniciales de la cadena

RTRIM(string) // elimina los espacios finales de la cadena

SUBSTRING(string, start, length) devuelve una subcadena de length de la string especificada, comenzando en la posición start



`UCASE(string)` devuelve la cadena en mayúsculas

## Funciones numéricas

---

`SQRT(number)` devuelve la raíz cuadrada de un número

`ABS(number)` devuelve el valor absoluto de un número

## Funciones de fecha y hora

---

`CURDATE()` devuelve la fecha actual.

`CURTIME()` devuelve la hora actual.

`DAYOFMONTH(date)` extrae el día del mes de la fecha indicada.

`HOURL(time)` extrae la hora del tiempo indicado.

`MINUTE(time)` extrae el minuto del tiempo indicado.

`NOW()` devuelve la marca temporal actual.

`SECOND(time)` extrae el segundo del tiempo indicado.

`YEAR(date)` extrae el año del tiempo especificado.

## Funciones del sistema

---

`USER()` devuelve el nombre de usuario de la conexión actual.

`CONVERT()` convierte una expresión escalar en uno de los siguientes tipos SQL:

BIGINT	BINARY	BIT
CHAR	DATE	DECIMAL
DOUBLE	FLOAT	INTEGER
LONGVARBINARY	LONGVARCHAR	REAL
SMALLINT	TIME	TINYINT
VARBINARY	VARCHAR	

## Ejemplos

### *Literales de fecha y hora*

```
INSERT INTO tablename VALUES({D '10-2-3'}, {T '2:55:11'})
SELECT {T '10:24'} FROM tablename
SELECT {D '2000-02-01'} FROM tablename
SELECT {TS '2000-02-01 10:24:32'} FROM tablename
```

*Uniones*

```
SELECT * FROM {OJ a LEFT JOIN b USING(id)}
```

*Especificar un carácter de escape para LIKE*

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

*Funciones de cadena*

```
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
```

*Funciones de fecha y hora*

```
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
```

# Palabras clave

---

Esta lista contiene todos los identificadores reservados para las palabras clave de esta versión del motor SQL de JDataStore. Las palabras clave no distinguen entre mayúsculas y minúsculas. Por ejemplo, `select`, `SELECT` y `SeLeCT` se gestionan como la palabra clave `SELECT`.

Se debe tener en cuenta que el motor SQL de JDataStore no trata todas las palabras clave de SQL-92 como tales. Para aumentar al máximo la portabilidad no se deben utilizar identificadores que se consideren palabras clave en ningún sublenguaje de SQL.

ABSOLUTE	ACTION	ADD
ALL	ALTER	AND
ANY	AS	ASC
AUTOCOMMIT	AUTOINCREMENT	AVG

BETWEEN	BIT	BOTH
BY	CALL	CASCADE
CASE	CAST	CHAR
CHAR_LENGTH	CHARACTER	CHARACTER_LENGTH
CHECK	COLUMN	COMMIT
CONSTRAINT	COUNT	CREATE
CROSS	CURRENT_DATE	CURRENT_TIME
CURRENT_TIMESTAMP	DATE	DAY
DEC	DECIMAL	DEFAULT
DELETE	DESC	DISTINCT
DOUBLE	DROP	ELSE
END	ESCAPE	EXCEPT
EXISTS	EXTRACT	FALSE
FLOAT	FOR	FOREIGN
FROM	FULL	GROUP
HAVING	HOURL	IN
INDEX	INNER	INSERT
INT	INTEGER	INTERSECT
INTO	IS	JOIN
KEY	LEADING	LEFT
LIKE	LOCK	LOWER
MAX	MIN	MINUTE
MONTH	NATURAL	NO
NOT	NOWAIT	NULL
NUMERIC	OFF	ON
OR	ORDER	OUTER
POSITION	PRECISION	PRIMARY
REAL	REFERENCES	RESTRICT
RIGHT	ROLLBACK	SECOND
SELECT	SET	SMALLINT
SOME	SQRT	SUBSTRING
SUM	TABLE	THEN
TIME	TIMESTAMP	TIMEZONEHOUR

TIMEZONEMINUTE	TRAILING	TRIM
TRUE	UNION	UNIQUE
UNKNOWN	UPDATE	UPPER
USER	USING	VALUES
VARCHAR	VARYING	WHEN
WHERE	WORK	YEAR

## Identificadores

---

Los identificadores SQL sin comillas no distinguen entre mayúsculas y minúsculas, y se tratan como si estuvieran en mayúsculas. Los identificadores se pueden colocar entre comillas dobles, y en ese caso se considera que distinguen entre mayúsculas y minúsculas. Los identificadores sin comillas deben seguir estas reglas:

- El primer carácter debe ser una letra reconocida por la clase `java.lang.Character`.
- Cada uno de los caracteres siguientes debe ser una letra, dígito, guión bajo (\_) o símbolo de dólar (\$).
- Las palabras clave no pueden emplearse como identificadores.

Los identificadores entre comillas pueden contener cualquier cadena de caracteres, que puede incluir espacios, símbolos y palabras clave.

### Ejemplos

Identificadores válidos:

```
customer    // tratado como CUSTOMER
Help_me     // tratado como HELP_ME
"Hansen"    // tratado como Hansen
" "         // tratado como espacio
```

Identificadores no válidos:

```
_order      // debe comenzar con un carácter
date        // date es palabra clave
borland.com // no se permiten los puntos
```

Los elementos de la siguiente lista son todos el mismo identificador, y todos se tratan como `LAST_NAME`:

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

# Expresiones

---

Las expresiones se utilizan en todo el lenguaje SQL. Contienen varios operadores de infijo y algunos de prefijo. La prioridad de los operadores, desde el que tiene mayor valor hasta el que tiene el menor, es:

- prefijo + -
- infijo \* /
- infijo + - ||
- infijo = <> < > <= >=
- prefijo NOT
- infijo AND
- infijo OR

## Sintaxis

```

<expresión> ::=
    <expresión escalar>
    | <expresión condicional>

<expresión condicional> ::=
    <expresión condicional> OR <expresión condicional>
    | <expresión condicional> AND <expresión condicional>
    | NOT <expresión condicional>
    | <expresión escalar> <operador de comparación> <expresión escalar>
    | <expresión escalar> <operador de comparación> { ANY | SOME | ALL } (
<expresión de tabla> )
    | <expresión escalar> [NOT] BETWEEN <expresión escalar>
    | <expresión escalar> [NOT] LIKE <expresión escalar> [ ESCAPE <expresión
escalar> ]
    | <expresión escalar> [NOT] IS { NULL | TRUE | FALSE | UNKNOWN }
    | <expresión escalar> IN ( <expresiones escalares separadas por comas> )
    | <expresión escalar> IN ( <expresión de tabla> )
    | EXISTS ( <expresión de tabla> )

<operador de comparación> ::=
    = | <> | < | > | <= | >=

<expresión escalar> ::=
    <expresión escalar> { + | - | * | / | <concat> } <expresión escalar>
    | { + | - } <expresión escalar>
    | ( <expresión> )
    | ( <expresión de tabla> )
    | <referencia a columna>
    | <referencia a función definida por el usuario>
    | <literal>
    | <función aggregator>
    | <función>
    | <marca de parámetro>

```

```
<concatenación> ::=      | |

<función> ::=
    <función substring>
  | <función position>
  | <función trim>
  | <función extract>
  | <función lower>
  | <función upper>
  | <función char lenght>
  | <función absolute>
  | <función square root>

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

<referencia a columna> ::= [<calificador de tabla> .] <nombre de columna>

<referencia a función definida por el usuario> ::= <nombre del método> ([
<expresiones separadas por comas> ])

<calificador de tabla> ::= <nombre de tabla> | <nombre de correlación>

<nombre de correlación> ::= <identificador SQL>
```

### Ejemplos

Seleccione el valor resultado de calcular la cantidad por el precio en la tabla de pedidos para un cliente que se facilitará más adelante, para los pedidos de enero:

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

Obtenga los datos mediante una subconsulta escalar:

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

Observe que hay un error si la subconsulta devuelve más de 1 fila.

## Predicados

---

Se aceptan los siguientes predicados, utilizados en expresiones condicionales.

## BETWEEN

---

El predicado `BETWEEN` define un intervalo de valores. El resultado de:

```
expr BETWEEN leftExpr AND rightExpr
```

equivale a la expresión:

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

### Sintaxis

```
<expresión between> ::=
    <expresión escalar> [NOT] BETWEEN <expresión escalar> AND <expresión
    escalar>
```

### Ejemplo

Seleccione todos los pedidos en los que un cliente ha encargado entre tres y siete artículos iguales:

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

## IS

---

El predicado `IS` se define para evaluar expresiones. Todas las expresiones evaluadas pueden presentar el valor `NULL`, pero las condicionales pueden presentar uno de los tres valores: `TRUE`, `FALSE`, `UNKNOWN`. En las expresiones condicionales, `UNKNOWN` equivale a `NULL`. Tenga en cuenta que en una consulta `SELECT` con una cláusula `WHERE` sólo se incluyen las filas cuyo valor sea `TRUE`. Si el valor de la expresión es `FALSE` o `UNKNOWN`, no se incluye. El predicado `IS` puede tener dos resultados: `TRUE` (verdadero) o `FALSE` (falso).

### Sintaxis

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

### Ejemplo

`TRUE IS TRUE` da como resultado `TRUE`.

`FALSE IS NULL` da como resultado `FALSE`.

## LIKE

---

El predicado `LIKE` proporciona a SQL una comparación sencilla de modelos de cadena. El elemento buscado, el modelo y el caracter de escape (si existe) deben ser cadenas. El modelo puede incluir los caracteres comodín `_` y `%`:

- El guión bajo ( `_` ) corresponde a un solo carácter

- El símbolo de porcentaje (%) representa una secuencia de  $n$  caracteres donde  $n \geq 0$

El carácter de escape, si lo hay, permite que se incluyan en el patrón de búsqueda los dos caracteres comodín. El patrón distingue entre mayúsculas y minúsculas. Si desea efectuar una búsqueda sin distinguir entre mayúsculas y minúsculas, utilice la función `LOWER` o `UPPER`.

### Sintaxis

```
<expresión like> ::=
    <elemento buscado> [NOT] LIKE <modelo> [ ESCAPE <carácter escape> ]

<elemento buscado> ::= <expresión escalar>

<patrón> ::= <expresión escalar>

<carácter escape> ::= <expresión escalar>
```

### Ejemplo

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

da como resultado `TRUE` si `Item` contiene la cadena “shoe”.

```
Item LIKE 'S_'
```

da como resultado `TRUE` si `Item` tiene exactamente tres caracteres y empieza por la letra “S”.

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

da como resultado `TRUE` si `Item` termina con el carácter de porcentaje. El asterisco `*` se define como carácter de escape para los dos caracteres especiales. Si precede a uno de ellos, este carácter se considera normal y no un comodín.

## IN

---

La cláusula `IN` indica una lista de valores que deben comprobarse. Cualquiera de los valores en la lista se considerará comprobado para la sentencia `SELECT` que contiene la cláusula `IN`.

### Sintaxis

```
<expresión in> ::= <expresión escalar> IN ( <expresiones escalares separadas
por comas> )
```

### Ejemplo

La siguiente expresión devuelve todos los registros en los que la columna `name` coincide con “leo” o con “aquarius”.

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

La cláusula `IN` también cuenta con una variante en la que se utiliza una subconsulta en lugar de una lista de expresiones.



**Sintaxis**

```
<expresión in> ::= <expresión escalar> IN ( <expresión de tabla> )
```

**Ejemplo**

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

**Comparaciones cuantificadas**

---

Una expresión se puede comparar con uno o todos los elementos de una tabla resultante.

**Sintaxis**

```
<comparación cuantificada> ::=
    <expresión escalar> <operador de comparación> { ANY | SOME | ALL } (
    <expresión de tabla> )
```

**Ejemplo**

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

**EXISTS**

---

Una expresión, que da como resultado tanto TRUE como FALSE dependiendo de si hay elementos en una tabla resultante.

**Sintaxis**

```
<predicado exists> ::= EXISTS ( <expresión de tabla> )
```

**Ejemplo**

La siguiente sentencia busca todo el equipo de buceo, siendo el nombre el principio de un nombre de una pieza diferente del equipo.

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

# Funciones

---

Se debe observar que todas las funciones que influyen sobre cadenas funcionan para las cadenas de cualquier longitud. Las cadenas grandes se guardan como BLOB, por lo que puede ser conveniente definir los campos de texto grandes como VARCHAR para activar las búsquedas.

## ABSOLUTE

---

La función ABSOLUTE funciona sólo con expresiones numéricas, y produce el valor absoluto del número pasado.

### Sintaxis

<función absolute> ::= ABSOLUTE( <expresión> )

### Ejemplo

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

## CHAR\_LENGTH y CHARACTER\_LENGTH

---

Las funciones SQL CHAR\_LENGTH y CHARACTER\_LENGTH devuelven la longitud de una cadena.

### Sintaxis

```
<función char length> ::=  
    CHAR_LENGTH ( <expresión escalar> )  
    CHARACTER_LENGTH ( <expresión escalar> )
```

## CURRENT\_DATE, CURRENT\_TIME y CURRENT\_TIMESTAMP

---

Estas funciones de SQL devuelven la fecha y la hora actuales. Si se colocan más de una vez en una sentencia, cuando ésta se ejecuta todas dan el mismo resultado.

### Ejemplo

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

## EXTRACT

---

La función `EXTRACT` de SQL extrae partes de valores de fecha y hora. La expresión puede ser un valor `DATE`, `TIME` o `TIMESTAMP`.

### Sintaxis

```
<función extract> ::=
    EXTRACT ( <campo de extracción> FROM <expresión escalar> )

<campo de extracción> ::=
YEAR
| MONTH
| DAY
| HOUR
| MINUTE
| SECOND
```

### Ejemplos

`EXTRACT(MONTH FROM DATE '1999-05-17')` da como resultado 5

`EXTRACT(HOUR FROM TIME '18:00:00')` da como resultado 18

`EXTRACT(HOUR FROM DATE '1999-05-17')` da como resultado una excepción

## LOWER y UPPER

---

Las funciones `LOWER` y `UPPER` de SQL dan como resultado la cadena dada, convertida en el formato solicitado: o todo en minúsculas o todo en mayúsculas.

### Sintaxis

```
<función lower> ::=
    LOWER ( <expresión escalar> )

<función upper> ::=
    UPPER ( <expresión escalar> )
```

## POSITION

---

La función `POSITION` de SQL devuelve la posición de una cadena dentro de otra. Si alguno de los argumentos tiene el valor `NULL`, el resultado es `NULL`.

### Sintaxis

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

### Ejemplo

`POSITION('BCD' IN 'ABCDEFGF')` da como resultado 2

`POSITION(' ' IN 'ABCDEFGH')` da como resultado 1

`POSITION('TAG' IN 'ABCDEFGH')` da como resultado 0.

## SQRT

---

La función `SQRT` funciona sólo con expresiones numéricas, y devuelve la raíz cuadrada del número pasado.

### Sintaxis

`<función sqrt> ::= SQRT(<expresión>)`

### Ejemplo

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

## SUBSTRING

---

La función `SUBSTRING` de SQL extrae una subcadena de una cadena. Si alguno de los operandos tiene el valor `NULL`, el resultado es `NULL`. La posición `start` indica el primer carácter de la posición de la que se extrae la subcadena, donde 1 indica el primer carácter. Si la parte `FOR` se encuentra presente, indica la longitud de la cadena resultante.

### Sintaxis

```
<función substring> ::=
    SUBSTRING ( <expresión string> FROM <posición inicial> [ FOR <longitud> ]
    )
```

### Ejemplos

`SUBSTRING('ABCDEFGH' FROM 2 FOR 3)` da como resultado 'BCD'

`SUBSTRING('ABCDEFGH' FROM 4)` da como resultado 'DEFG'

`SUBSTRING('ABCDEFGH' FROM 10)` da como resultado '' (comillas)

`SUBSTRING('ABCDEFGH' FROM -6 FOR 3)` da como resultado 'ABC'

`SUBSTRING('ABCEDFG' FROM 2 FOR -1)` genera una excepción

## TRIM

---

La función `TRIM` de SQL elimina los caracteres de relleno, iniciales o finales, de una cadena. `<relleno>` debe ser una cadena de longitud 1, que es el carácter eliminado de la cadena.

- Si `<relleno>` se omite, los caracteres de espacio se eliminan.
- Si se omite `<especificaciones de recorte>` se adopta `BOTH`.

- Si se omiten <relleno> y <especificaciones de recorte> es necesario omitir la palabra clave FROM.

### Sintaxis

```
<función trim> ::=
    TRIM ( [<especificaciones de recorte>] [<relleno>] [FROM] <expresión
escalar> )
```

```
<especificaciones de relleno> ::=
LEADING
| TRAILING
| BOTH
```

```
<relleno> ::= <expresión escalar>
```

### Ejemplo

```
TRIM(' Hello world ') da como resultado 'Hola a todos'
TRIM(LEADING '0' FROM '00000789.75') devuelve '789.75'
```

## CAST

---

La función CAST cambia un tipo de dato por otro tipo de dato.

### Sintaxis

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

### Ejemplo

```
SELECT * FROM employee WHERE CAST ( salary AS long ) = 86293 devuelve row en
la que salary es igual a 86,292.94
```

## La sintaxis de las sentencias

---

Esta sección recoge un número de convenciones que se utilizan en las siguientes referencias a sentencias. Y más en concreto:

- Listas
- Expresiones de tabla
- Expresiones de selección
- Expresiones de unión

## Listas en la sintaxis

---

En la siguiente sección se encuentran nombres de elementos que comienzan por las palabras “lista” o “lista separada por comas”, que no se definen más adelante. Por ejemplo:

```
<lista separada por comas de elementos seleccionados>
<lista de restricciones de columna>
```

Estas definiciones se deben leer como listas separadas por comas con un elemento por lo menos, en el caso de las listas separadas por comas:

```
<lista separada por comas de elementos seleccionados> ::= <elemento
seleccionado> [ , <elemento seleccionado> ]*

<lista de restricciones de columna> ::=
  <restricción de columna> [ <restricción de columna> ] *
```

## Expresiones de tabla

---

Una expresión de tabla es el término para una expresión que da como resultado una tabla sin nombre. De los operadores de infijo `INTERSECT`, `UNION` y `EXCEPT`: `INTERSECT` obliga en mayor grado y `UNION` y `EXCEPT` son iguales.

UNION ALL	Crea la unión de dos tablas, incluidos todos los duplicados.
UNION	Crea la unión de dos tablas. Si una fila aparece varias veces en ambas tablas, el resultado tiene esa fila dos veces. Otras filas del resultado no tienen duplicados.
INTERSECTION ALL	Crea la intersección de las dos tablas, incluidos todos los duplicados.
INTERSECTION	Crea la intersección de dos tablas. Si una fila tiene duplicados en ambas tablas, el resultado tiene esa fila dos veces. Otras filas del resultado no tienen duplicados.
EXCEPT ALL	Crea una tabla que contiene todas las filas que sólo aparecen en la primera tabla. Si una fila aparece $m$ veces en la primera tabla y $n$ veces en la segunda, el resultado mantiene el valor de la fila por encima de cero y en $m-n$ veces.
EXCEPT	Crea una tabla que contiene todas las filas que sólo aparecen en la primera tabla. Si una fila aparece $m$ veces en la primera tabla y $n$ veces en la segunda, el resultado tiene dos filas si $m > 1$ y $n = 0$ . Otras filas del resultado no tienen duplicados.

## Sintaxis

```
<expresión de tabla> ::=
    <expresión de tabla> UNION [ ALL ] <expresión de tabla>
  | <expresión de tabla> EXCEPT [ ALL ] <expresión de tabla>
  | <expresión de tabla> INTERSECT [ ALL ] <expresión de tabla>
  | <expresión de unión>
  | <expresión de selección>
  | ( <expresión de tabla> )
```

## Ejemplos

```
SELECT * FROM T1 UNION SELECT * FROM T2 UNION SELECT * FROM T3
```

se ejecuta como:

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

se ejecuta como:

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

## Expresiones de selección

---

Una expresión de selección es la expresión de tabla que se utiliza más a menudo en una sentencia `SELECT`.

Especifique `DISTINCT` para eliminar los duplicados del resultado.

Especifique `GROUP BY` y `HAVING` junto con las funciones de totalización para calcular los valores resumen de los datos de la tabla. La cláusula `WHERE` (si se utiliza) limita el número de filas incluidas en el resumen. Si hay una función agregada pero no una cláusula `GROUP BY` se calcula el resumen de toda la tabla. Si hay cláusula `GROUP BY` se calcula un resumen para cada conjunto de valores no repetidos de las columnas enumeradas en `GROUP BY`. Si hay cláusula `HAVING`, se filtran los grupos completos establecidos en su expresión condicional.

Las consultas de resumen tienen reglas adicionales para las partes de las expresiones que pueden ocupar las columnas:

- En la cláusula `WHERE` no puede haber funciones de totalización.
- Las referencias de columna que aparecen fuera del totalizador se deben encontrar en la cláusula `GROUP BY`.
- No es posible anidar funciones de totalización.

```
<expresión select> ::=
    SELECT [ ALL | DISTINCT ] <lista de elementos seleccionados separados por
comas>
    FROM   <lista referencia de tabla separados por comas>
          [ WHERE <expresión condicional> ]
          [ GROUP BY <lista de referencias a columnas separadas por comas> ]
          [ HAVING <expresión condicional> ]
```

```
<función aggregator> ::=
    <nombre del totalizador> ( <expresión> )
    | COUNT ( * )

<nombre del totalizador> ::=
AVG
    | SUM
    | MIN
    | MAX
    | COUNT
```

### Ejemplos

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

da como resultado una sola fila con el valor total de todos los pedidos.

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

tiene como resultado una sola fila con el número de pedidos en los que el valor de Amount para el cliente 123 no es NULL.

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

da como resultado un conjunto de filas con la suma del valor de todos los pedidos agrupado por los clientes para aquéllos cuyo número de ID sea inferior a 200.

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

da como resultado un conjunto de clientes importantes con el valor de todos sus pedidos.

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

No válido: hay totalizadores anidados.

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

No válido: la columna CustId se menciona en la lista elemento seleccionado, pero no en la lista referencia GROUP BY.



## Expresiones de unión

---

Las expresiones de unión en JDataStore proporcionan acceso a una gran variedad de mecanismos de unión. Los dos utilizados con más frecuencia, las uniones internas y cruzadas, se pueden expresar sólo con la expresión `SELECT`, pero cualquier tipo de unión externa se debe expresar mediante una expresión `JOIN`.

CROSS JOIN	<p><code>A CROSS JOIN B</code>  <b>produce el mismo resultado que</b>  <code>SELECT A.*, B.* FROM A,B</code></p>
INNER JOIN	<p><code>A INNER JOIN B ON A.X=B.X</code>  <b>produce el mismo resultado que</b>  <code>SELECT A.*, B.* FROM A,B WHERE A.X=B.X</code></p>
LEFT OUTER	<p><code>A LEFT OUTER JOIN B ON A.X=B.X</code>  <b>produce las filas de la correspondiente unión interna junto con las filas de A que no contribuyen, rellenando los espacios correspondientes a las columnas de B con valores NULL.</b></p>
RIGHT OUTER	<p><code>A RIGHT OUTER JOIN B ON A.X=B.X</code>  <b>produce las filas de la correspondiente unión interna junto con las filas de B que no contribuyen, rellenando los espacios correspondientes a las columnas de A con valores NULL.</b></p>
FULL OUTER	<p><code>A FULL OUTER JOIN B ON A.X=B.X</code>  <b>produce las filas de la correspondiente unión interna junto con las filas de A y B que no contribuyen, rellenando los espacios correspondientes a las columnas de A y B con valores NULL.</b></p>
UNION	<p><code>A UNION JOIN B</code>  <b>produce un resultado parecido a</b></p> <p><code>A LEFT OUTER JOIN B ON FALSE  UNION ALL  A RIGHT OUTER JOIN B ON FALSE</code></p> <p><b>una tabla con columnas de todas las columnas de A y B, con todas las filas de A que tengan valores NULL para las columnas de B añadidas, con todas las filas de B con valores NULL para las columnas de A.</b></p> <p><b>NATURAL, ON y USING se excluyen mutuamente:</b></p>
ON	<p><b>ON es una expresión que necesita cumplirse para una expresión JOIN.</b></p>

**USING** USING ( C1, C2, C3)  
**es equivalente a la expresión** ON  
A.C1=B.C1 AND A.C2=B.C2 AND A.C3=B.C3,  
**excepto que en la tabla resultante las columnas C1, C2 y C3 sólo aparecen una vez, y son las tres primeras columnas.**

**NATURAL** NATURAL **es lo mismo que una cláusula USING con los nombres de las columnas que aparecen en las columnas A y B.**

## Sintaxis

```
<expresión de unión> ::=
    <referencia de tabla> CROSS JOIN <referencia de tabla>
  | <referencia de tabla> [NATURAL] [ INNER ] JOIN <referencia de tabla>
    [ <tipo de unión> ]
  | <referencia de tabla> [NATURAL] LEFT [OUTER] JOIN <referencia de tabla>
    [ <tipo de unión> ]
  | <referencia de tabla> [NATURAL] RIGHT [OUTER] JOIN <referencia de tabla>
    [ <tipo de unión> ]
  | <referencia de tabla> [NATURAL] FULL [OUTER] JOIN <referencia de tabla>
    [ <tipo de unión> ]
  | <referencia de tabla> UNION JOIN <referencia de tabla>

<referencia de tabla> ::=
    <expresión de unión>
  | <nombre de tabla> [AS] <variable de rango>
  | ( <expresión de tabla> )

<variable de rango> ::= <identificador SQL>

<tipo de unión> ::=
    ON <expresión condicional>
  | USING ( <lista nombres de columnas separados por comas> )
```

## Ejemplos

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

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

SELECT * FROM Tinvoice NATURAL RIGHT OUTER JOIN Titem

SELECT * FROM Tinvoice INNER JOIN Titem USING ("InvoiceNumber")

SELECT * FROM Tinvoice JOIN Titem ON Tinvoice."InvoiceNumber"
    = Titem."InvoiceNumber"
```

# Sentencias

---

El controlador JDBC de JDataStore acepta un subconjunto de la norma ANSI/ISO SQL-92. En líneas generales, proporciona:

- Lenguaje de definición de datos para la gestión de tablas e índices, pero no de esquemas, dominios, vistas ni elementos de seguridad.
- Manipulación y selección de datos con `INSERT`, `UPDATE`, `DELETE` y `SELECT`; sin cursores.
- Las operaciones de cursor se admiten mediante la JDBC versión 2.0 `ResultSet` API.
- Admite las expresiones de tabla generales, incluidas `JOIN`, `UNION` y `INTERSECT`.

## Sintaxis

```
<sentencia SQL> ::=
    <sentencia de definición de datos>
    | <sentencia de control de transacciones>
    | <sentencia de manipulación de datos>

<sentencia de definición de datos> ::=
    <sentencia create table>
    | <sentencia alter table>
    | <sentencia drop table>
    | <sentencia create index>
    | <sentencia drop index>
    | <sentencia create java method>
    | <sentencia drop java method>

<sentencia de control de transacciones> ::=
    <sentencia commit>
    | <sentencia rollback>
    | <sentencia set autocommit>

<sentencia de manipulación de datos> ::=
    <sentencia select>
    | <sentencia single row select>
    | <sentencia delete>
    | <sentencia insert>
    | <sentencia update>
    | <sentencia call>
    | <sentencia lock>
```

## CREATE TABLE

---

Esta sentencia crea una tabla JDataStore. Es necesario definir un nombre de columna y un tipo de datos para cada columna.

Opcionalmente, puede especificar un valor por defecto para cada una de las columnas, únicas para las restricciones.

Si lo desea, puede también especificar una clave ajena y una clave principal. En un JDataStore pueden definirse una o más columnas como clave principal o ajena.

### Sintaxis

```

<sentencia create table> ::=
    CREATE TABLE <nombre de tabla> ( <lista elemento de tabla separados por
    coma> )

<nombre de tabla> ::= <identificador SQL>

<elemento de tabla> ::=
    <definición de columna>
    | <clave principal>
    | <clave única>
    | <clave ajena>

<definición de columna> ::=
    <nombre de columna> <tipo de datos>
    [ DEFAULT <valor por defecto> ]
    [[NOT] NULL]
    [AUTOINCREMENT]
    [[CONSTRAINT <nombre de restricción>] PRIMARY KEY]
    [[CONSTRAINT <nombre de restricción>] UNIQUE]
    [[CONSTRAINT <nombre de restricción>] <definición de referencias>]

<nombre de columna> ::= <identificador SQL>

<valor por defecto> ::=
    <literal>
    | <función current date>

<clave principal> ::=
    [CONSTRAINT <nombre de restricción>] PRIMARY KEY <lista de nombres de
    columnas separados por comas>

<clave principal> ::=
    [CONSTRAINT <nombre de restricción>] UNIQUE ( <lista de nombres de columnas
    separados por comas> )

<clave ajena> ::=
    [CONSTRAINT <nombre de restricción>] FOREIGN KEY <lista de nombres de
    columnas separados por comas>
    <definición de referencias>

<definición de referencias> ::=
    REFERENCES <nombre de tabla> [( <lista de nombres de columnas separados por
    comas> )]
    [ON DELETE
    <acción>]
    [ON UPDATE <acción>]

```

```

[NO CHECK]

<acción> ::=
    NO ACTION
  | CASCADE
  | SET DEFAULT
  | SET NULL

<nombre de restricción> ::= <identificador SQL>

```

**Nota:** La opción `NO CHECK` crea la clave ajena sin comprobar la coherencia en el momento de la creación. Utilice esta opción con precaución.

## Ejemplos

```

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

```

**Ejemplo de la creación de una tabla definiendo dos columnas como restricción de clave principal:**

```

CREATE TABLE t1 (c1 INT, c2 STRING, c3 STRING, primary key (c1, c2))

```

### *Columnas con autoincremento en SQL*

Para crear o modificar una tabla mediante SQL de modo que disponga de una columna con autoincremento, añada la palabra clave `AUTOINCREMENT` en la definición del <elemento de tabla>.

El siguiente ejemplo crea la tabla `t1` con una columna entera con autoincremento denominada `c1`:

```

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

```

Para obtener el valor de autoincremento de una fila recién insertada mediante el controlador JDBC de JDS, con la versión 1.3 de JVM, se puede utilizar el método `JdsStatement.getGeneratedKeys()`. Este método también está disponible en la interfaz de la sentencia de JDBC 3 en JVM 1.4.)

## ALTER TABLE

---

Esta sentencia añade y elimina columnas en una tabla `JDataStore`, y establece nuevas restricciones y valores por defecto para la columna.

### Sintaxis

```

<sentencia alter table> ::=
    ALTER TABLE
    nombre de tabla <lista de cambios de columna separados por comas>

<definición de cambio> ::=
    <elemento de columna añadido>
  | <elemento de columna colocado>
  | <elemento de columna alternado>
  | <restricción añadida>
  | <restricción colocada>

```

```
<elemento de columna añadido> ::= ADD [COLUMN] <definición de columna>

<elemento de columna colocado> ::= DROP [COLUMN] <nombre de columna>

<elemento de columna alternado> ::=
    ALTER [COLUMN] <nombre de columna> SET <definición por defecto>
    | ALTER [COLUMN] <nombre de columna> DROP DEFAULT

<restricción añadida> ::= ADD <restricción de tabla base>

<restricción de tabla base> ::=
    <clave principal> | <clave única> | <clave ajena>

<restricción colocada> ::= DROP CONSTRAINT <nombre de restricción>

<clave principal> ::=
    [CONSTRAINT <nombre de restricción>] PRIMARY KEY <lista de nombres de
    columnas separados por comas>

<clave principal> ::=
    [CONSTRAINT <nombre de restricción>] UNIQUE ( <lista de nombres de columnas
    separados por comas>)

<clave ajena> ::=
    [CONSTRAINT <nombre de restricción>] FOREIGN KEY <lista de nombres de
    columnas separados por comas>
    <definición de referencias>

<definición de referencias> ::=
    REFERENCES <nombre de tabla> [( <lista de nombres de columnas separados por
    comas> )]
    [ON DELETE
    <acción>]
    [ON UPDATE <acción>]
    [NO CHECK]

<acción> ::=
    NO ACTION
    | CASCADE
    | SET DEFAULT
    | SET NULL

<nombre de restricción> ::= <identificador SQL>
```

En `ALTER [COLUMN]`, la palabra clave optativa `COLUMN` se incluye para la compatibilidad con SQL. Sin embargo, no tiene ningún efecto.

### Ejemplo

```
ALTER TABLE Orders Add ShipDate DATE, DROP Amount
```

## DROP TABLE

---

Esta sentencia borra una tabla y sus índices del archivo JDataStore.

### Sintaxis

```
<sentencia drop table> ::=  
    DROP TABLE <nombre de tabla>
```

### Ejemplo

```
DROP TABLE Orders
```

## CREATE INDEX

---

Esta sentencia crea un índice para una tabla JDataStore. Cada columna se puede ordenar de forma ascendente o descendente. El valor predefinido es el orden ascendente.

### Sintaxis

```
<sentencia create index> ::=  
    CREATE [UNIQUE] [CASEINSENSITIVE] INDEX <nombre de índice>  
    ON <nombre de tabla> ( <lista elemento de tabla separados por coma> )  
  
<nombre de índice> ::= <identificador SQL>  
  
<elemento de índice> ::= <nombre de columna> [DESC | ASC]
```

### Ejemplo

Esta sentencia genera un índice ascendente que puede repetirse y que distingue entre mayúsculas y minúsculas en la columna `ITEM` de la tabla `ORDERS`:

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

## DROP INDEX

---

Esta sentencia elimina un índice de una tabla JDataStore.

### Sintaxis

```
<sentencia drop index> ::=  
    DROP INDEX <nombre de índice> ON <nombre de tabla>
```

### Ejemplo

Esto elimina el índice `ORDERINDEX` de la tabla `ORDERS`:

```
DROP INDEX OrderIndex ON Orders
```

## CREATE JAVA\_METHOD

---

Esta sentencia coloca un procedimiento almacenado o una UDF (función definida por el usuario) escritos en Java, disponibles para utilizarlos en SQL de JDataStore. Los archivos de clase del código deben añadirse a la vía de acceso a clases del proceso del servidor JDataStore antes de utilizarlos. Consulte [Las UDF y los procedimientos almacenados](#), si desea obtener más detalles sobre la implementación de procedimientos almacenados y UDF para JDataStore.

### Sintaxis

```
<sentencia create java method > ::=  
    CREATE JAVA_METHOD <nombre de método> AS <definición de método>  
  
<nombre de método> ::= <identificador SQL>  
  
<definición de método> ::= <literal de cadena>
```

### Ejemplo

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

## DROP JAVA\_METHOD

---

Esta sentencia coloca un procedimiento almacenado o una UDF, haciendo que sea posible utilizarlo en SQL de JDataStore.

### Sintaxis

```
<sentencia drop java method > ::=  
    DROP JAVA_METHOD <nombre de método>
```

### Ejemplo

```
DROP JAVA_METHOD ABS
```

## COMMIT

---

Esta sentencia confirma la transacción actual. No tiene ningún efecto a menos que se desactive AUTOCOMMIT.

### Sintaxis

```
<sentencia commit> ::= COMMIT [ WORK ]
```



## ROLLBACK

---

Esta sentencia cancela la transacción actual. No tiene ningún efecto a menos que se desactive `AUTOCOMMIT`.

### Sintaxis

```
<sentencia rollback> ::= ROLLBACK [ WORK ]
```

## SET AUTOCOMMIT

---

Esta sentencia cambia el modo de confirmación automática. Este modo también se puede controlar directamente a través de la instancia de conexión JDBC.

### Sintaxis

```
<sentencia set autocommit> ::= SET AUTOCOMMIT { ON | OFF }
```

## SELECT

---

Las sentencias `SELECT` se utilizan para recuperar datos de una o más tablas. La palabra clave optativa `DISTINCT` elimina las filas duplicadas del resultado de las sentencias `SELECT`. La palabra clave `ALL`, que es la predeterminada, obtiene todas las filas, incluidas las duplicadas. Opcionalmente, los datos se pueden ordenar mediante `ORDER BY`. Las filas recuperadas se pueden bloquear de forma opcional para el siguiente `UPDATE` especificando `FOR UPDATE`.

### Sintaxis

```
<sentencia select> ::=
    <expresión de tabla> [ ORDER BY <lista elementos de pedido> ] [ FOR UPDATE
    ]

<expresión de tabla> ::=
    <expresión de tabla> UNION [ ALL ] <expresión de tabla>
    | <expresión de tabla> EXCEPT [ ALL ] <expresión de tabla>
    | <expresión de tabla> INTERSECT [ ALL ] <expresión de tabla>
    | <expresión de unión>
    | <expresión de selección>
    | ( <expresión de tabla> )

<elemento de pedido> ::= <parte de pedido> [ ASC | DESC ]

<parte de pedido> ::=
    <literal entero> | <nombre de columna> | <expresión>

<expresión select> ::=
    SELECT [ ALL | DISTINCT ] <lista de elementos seleccionados separados por
    comas>
    FROM    <lista referencia de tabla separados por comas>
```

```
[ WHERE <expresión condicional> ]  
[ GROUP BY <lista de referencias a columnas separadas por comas> ]  
[ HAVING <expresión condicional> ]
```

### Ejemplos

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

efectúa la ordenación a partir de Item, la primera columna.

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

ordena a partir de la columna calculada CALC.

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

ordena a partir de la expresión facilitada.

## SELECT INTO

---

Una sentencia `SELECT INTO` es una sentencia `SELECT` que da como resultado exactamente una fila, cuyos valores se recuperan en parámetros de salida. Se produce un error si `SELECT` da como resultado más de una fila o en un conjunto vacío.

### Sintaxis

```
<sentencia single row select> ::=  
    SELECT [ ALL | DISTINCT ] <lista de elementos seleccionados separados por  
comas>  
        INTO < lista de parámetros separados por comas>  
FROM    <lista referencia de tabla separados por comas>  
        [ WHERE <expresión condicional> ]  
        [ GROUP BY <lista de referencias a columnas separadas por comas> ]  
        [ HAVING <expresión condicional> ]
```

### Ejemplo

```
SELECT CustId, Amount INTO ?, ? FROM Orders WHERE CustId=?
```

Los dos primeros marcadores de parámetros indican los parámetros de salida de los que se puede recuperar el resultado de la consulta.

## INSERT

---

La sentencia `INSERT` inserta filas en una tabla del archivo `JDataStore`. Contiene una lista de columnas con valores asociados. Las columnas que no se mencionan en la sentencia adoptan el valor por defecto.

### Sintaxis

```
<sentencia insert> ::=  
    INSERT INTO <nombre de tabla> ( <lista nombre de columna> )]  
    [<expresión de inserción tabla> | DEFAULT VALUES ]  
  
<expresión de inserción tabla> ::=
```

```
<expresión de selección>
| VALUES (<lista de expresiones separadas por comas>)
```

## Ejemplo

La sentencia siguiente se debe utilizar en combinación con `PreparedStatement` en JDBC. Inserta una fila cada vez que se ejecuta. Las columnas que no se mencionan adoptan su valor por defecto. Si una columna no tiene un valor por defecto, se le asigna `NULL`.

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

Esta sentencia busca todos los pedidos del cliente cuyo ID es `CustId = 123`, e inserta la columna `Item` de estos pedidos en la tabla `ResTable`.

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

## UPDATE

---

La sentencia `UPDATE` se utiliza para modificar datos. Las columnas que modifica esta sentencia se recogen de forma explícita. Todas las filas para las que la cláusula `WHERE` da como resultado `TRUE` se modifican. Si no se especifica la cláusula `WHERE`, todas las filas de la tabla se modifican.

### Sintaxis

```
<sentencia update> ::=
    UPDATE <nombre de tabla>
    SET <lista de asignaciones de actualización separadas por comas>
    [ WHERE <expresión condicional> ]

<asignación de actualización> ::=
    <referencia a columna> = <expresión de actualización>

<expresión de actualización> ::=
    <expresión escalar>
    | DEFAULT
    | NULL
```

### Ejemplos

Todos los pedidos del cliente 123 se asignan al cliente 500:

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

Aumenta la cantidad de todos los pedidos de la tabla:

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

Cambie el precio de todas las cámaras sumergibles a 7.25:

```
UPDATE Orders SET Price = 7.25 WHERE Price > 7.25 AND Item = 'UWCamaras'
```

## DELETE

---

La sentencia `DELETE` borra las filas de una tabla del archivo `JDataStore`. Si no se especifica la cláusula `WHERE` se borran todas las filas. De lo contrario, se borran sólo las filas que coinciden con la expresión `WHERE`.

### Sintaxis

```
<sentencia delete> ::=
    DELETE FROM <nombre de tabla>
    [ WHERE <expresión condicional> ]
```

### Ejemplo

```
DELETE FROM Orders WHERE Item = 'Shorts'
```

## CALL

---

Se llama a un procedimiento almacenado mediante la sentencia `CALL`.

### Sintaxis

```
<sentencia call> ::=
    [ ? = ] CALL <nombre de método> ( <lista de expresiones separadas por
    comas> )
```

### Ejemplos

```
?=CALL ABS(-765)
```

El marcador del parámetro indica la posición del parámetro de salida desde la que se puede recuperar el resultado del procedimiento almacenado.

```
CALL IncreaseSalaries(10)
```

El método Java que implementa `IncreaseSalaries` actualiza la tabla `salaries` con un aumento de un porcentaje para todos los empleados. Se pasa `java.sql.Connection` de forma implícita al método Java. Un `updateCount` de todas las filas afectadas por `IncreaseSalaries` se devuelve desde `Statement.executeUpdate()`.

## LOCK TABLE

---

Una tabla se puede bloquear explícitamente mediante la sentencia `LOCK TABLE`. El bloqueo deja de existir cuando la transacción se confirma o se cancela.

### Sintaxis

```
<sentencia lock> ::=  
    LOCK <lista referencia de tabla separados por comas>
```

### Ejemplo

```
LOCK Orders, LineItems
```



# Índice

## A

---

- abrir
  - archivos JDataStore transaccionales 3-8
  - conexiones 2-3
- ABSOLUTE (función) C-16
- accesos 2-13
  - aleatorio 2-13
  - de escritura 2-13
  - de lectura 2-13
  - multiusuario 2-15, 5-1
    - proceso de transacciones 6-1
  - remotamente 5-1
  - remoto 5-1
  - tablas JDataStore y JDBC 3-20
  - y varios usuarios 2-15, 5-1
- actualizar
  - archivos JDataStore 2-22, 2-24, C-33
  - con Explorador de JDataStore 9-24
- agrupación de conexiones 6-8
- aislamiento de transacciones 3-6
  - niveles (descripción) 6-1
- almacenadores y componentes DataStore 3-5
- almacenamiento 1-1, 8-1
  - flujos de datos 2-13
  - objetos Java 2-5
  - persistente 2-6
  - tamaño del archivo JDataStore A-1
  - ventajas del persistente 2-6
- ALogDir (propiedad) 3-11, 10-8
- ALTER TABLE (sentencia) C-27
- añadir filas de datos
  - a tablas JDataStore C-32
- aplicaciones
  - crear para edición fuera de línea 8-3
  - de ejemplo DataStoreDemo 8-3
  - de una sola conexión 3-13
- aplicaciones (JDBC)
  - crear 2-18
- appendRow (método) 3-6
- archivos
  - capacidad de almacenamiento A-1
  - importar 2-14
  - marcado como abierto 9-4
  - SCHEMA 9-18
  - visualizadores específicos 9-7
- archivos de imagen 9-7
- archivos de texto 9-7
  - delimitados 9-18
  - importar 9-18

- archivos JAR
  - servidor remoto JDBC de JDataStore 5-4
- archivos JDataStore 2-1, 9-23
  - abrir 9-4
  - accesos remoto 5-1
  - activar admisión de transacciones 3-6, 3-8
  - actualizar 2-22, 2-24, 9-24
  - borrar 2-4, 9-24
  - cambiar valores de transacción 3-9
  - capacidad A-1
  - clasificar 2-11
  - copia de seguridad 2-24
  - copiar 2-13, 2-22
  - crear 2-3, 9-3
  - empaquetar 2-29, 9-23
  - evitar admisión de transacciones 3-11
  - obtener contenido 2-7, 2-9
  - reestructurar 8-9
  - registrar cambios 3-10
  - visualizar 9-5
- asignar
  - flujos JDataStore 2-26
  - nombre a flujos 2-23
  - tipos devueltos 7-6
- AUTOCOMMIT C-31

## B

---

- base de datos JDataStore
  - activar admisión de transacciones 9-11
  - cambiar valores de transacción 9-11
  - cerrar 9-12, 10-2
  - como caché de datos 9-12
  - copiar 9-9
  - eliminar característica de transacciones 3-12, 9-12
  - redefinir tabla 8-8
- bases de datos 3-1
  - incrustadas 3-1
    - abrir en modo de sólo lectura 3-11
    - cargar 3-5
    - modificar datos 3-13
    - proporciona funcionalidad para 1-1
- BETWEEN (operador) C-13
- bit HIDDEN\_STREAM 2-10
- blockSize (propiedad) 2-15
- BLogDir (propiedad) 3-11, 10-8
- bloquear JDataStores C-35
- bloqueos 6-1
  - de fila 6-3
  - de sección crítica 6-3

- de tabla 6-3
- de tabla DDL 6-4
- borrar
  - archivos JDataStore 2-4, 3-12, 9-24
  - flujos JDataStore 2-26, 9-9
    - ejemplo 2-27
  - registros en tablas JDataStore C-34

## C

- caché 10-2
  - base de datos JDataStore como 9-12
  - controlar escritura de disco 10-4
  - guardar para JDataStore B-1
- caché de datos 10-2
  - controlar escritura de disco 10-4
  - guardar para JDataStore B-1
  - persistentes 9-12
- cadena (funciones) C-6
- cadenas
  - eliminar caracteres de relleno C-18
  - extraer subcadenas C-18
  - obtener posición C-17
- CALL(sentencia) 7-4, C-34
- cambiar
  - la configuración de servidores remotos 5-3
  - valores de transacción 3-9, 9-11
- cancelación 3-10, 3-14
- caracteres de relleno C-18
- caracteres finales C-18
- caracteres iniciales C-18
- CAST (función) C-19
- cerrar
  - base de datos DataStore 3-17
  - base de datos JDataStore 9-12, 10-2
  - conexiones 2-3
  - flujos de archivos 2-16
- CHAR\_LENGTH (función) C-16
- CHARACTER\_LENGTH (función) C-16
- clases
  - CollationKey B-2
  - DataSetException 2-4
  - DataStoreConnection 2-4
  - DataStoreException 2-4
  - StorageDataSet 3-2
  - StreamVerifier B-1
  - TxManager 3-6
- clasificar
  - archivos JDataStore 2-11
- claves
  - de licencia 1-3
  - de ordenación
    - resolución de problemas B-2
- claves ajenas C-25
- closeDirectory (método) 2-12
- CollationKey (clases) B-2
- columnas
  - autoincremento 10-10
  - utilizando DataExpress 10-11
- comandos
  - actualizar versión de JDataStore 9-24
  - compactar el JDataStore 9-23
  - eliminar JDataStore 9-24
  - verificar JDataStore 9-10
- commit (métodos) 3-13
- COMMIT (sentencia) C-30
- comparaciones cuantificadas C-15
- componentes
  - DataStore 1-1, 2-4, 8-3, 10-2, B-1
  - DataStoreConnection 3-13
  - StorageDataSet 8-2, 8-6, 10-6
  - TableDataSet 3-1
- conexiones
  - agrupación de conexiones JDBC 6-8
  - bloqueadas 6-6
  - componentes DataStore 2-3, 3-2
  - controladores remotos 5-2
  - DataStore
    - hacer referencia 2-15
  - hacer referencia a JDataStore 2-15
  - JDataStore 2-3, 3-2
  - JDBC 3-20
    - y control de transacciones 3-23
  - multiusuario 3-13
  - parámetro implícito 7-5
  - y proceso de transacciones 3-13, 3-23
- confirmaciones 3-14, 10-8
  - automática 3-19
- conflictos 6-1
  - evitar 6-6
- conjuntos de datos
  - JDataStores y 8-5
- consola de consultas 9-12
  - limitaciones 9-13
- constantes 2-10
- consultas
  - añadir expresiones C-11
  - crear C-1
  - distinción entre mayúsculas y minúsculas C-10
  - ejecutar en el Explorador de JDataStore 9-13, 9-14, 9-16
  - SQL
    - crear C-1
    - distinción entre mayúsculas y minúsculas C-10
    - ejecutar en el Explorador de JDataStore 9-13, 9-14, 9-16
- contextos de transacción 3-13



- contraseña modificación 9-26
- controladores
  - conexiones JDBC 5-2
  - JDBC 3-20, 5-2
  - ejecutar 5-3
- controladores JDBC
  - usar para ejecutar SQL C-1
- controlar escritura de disco 10-4
- conversión en otro tipo de datos C-19
- copia de seguridad de archivos JDataStore 2-24
- copiar
  - contenidos de JDataStore 2-13
  - flujos JDataStore 2-22, 9-9
- copyStreams (método) 2-22, 2-29
- creación de aplicaciones para edición fuera de línea 8-3
- crear
  - consultas C-1
  - flujos de archivos 2-15
  - flujos de tabla 3-4
  - históricos de transacciones 3-10
  - tablas independientes 8-8
- CREATE JAVA\_METHOD (sentencia) 7-2, C-30
- CREATE TABLE (sentencia) C-25
- createFileStream (método) 2-15
- cuadros de diálogo
  - copiar flujos 9-10
  - Nuevo JDataStore 9-3, 9-4
  - Propiedades de TxManager 9-11
- CURRENT\_DATE (función) C-16
- CURRENT\_TIME (función) C-16
- CURRENT\_TIMESTAMP (función) C-16

## D

---

- Dataexpress (paquete)
  - importar 3-2
- DATASET\_EXISTS (código de error) 2-15
- DataSetException (clase) 2-4
- DataStore (componente) 2-3, 3-2
  - aspectos generales de su utilización 1-1
  - recomendaciones de uso 10-2
  - solución de problemas B-1
  - tratamiento de excepciones 2-4
  - tutorial de utilización 8-3
- DataStoreConnection (clase) 2-4
- DataStoreConnection (componente) 3-13
- DataStoreException (clase) 2-4
- datos persistentes 8-1
- DELETE (sentencia) C-34
- deleteStream (método) 2-26
- DeleteTest.java 2-27
- delimitadores de cadena 9-18
- depurar JDataStores B-1

- Dir.java 2-8
  - ejecutar 2-11
- distribución del servidor de JDataStore 5-4
- distribuir
  - minimizar recursos 10-10
  - Servidor de JDataStore 5-4
- documentación 1-2
- documentos de acceso frecuente 2-13
- DROP INDEX (sentencia) C-29
- DROP JAVA\_METHOD (sentencia) C-30
- DROP TABLE (sentencia) C-29
- Dup.java 2-24
- DxTable.java 3-2

## E

---

- ediciones
  - fuera de línea 8-3
  - remota 8-3
- editar en columnas persistentes 8-8
- editor de columnas persistentes 8-8
- eliminar
  - característica de transacciones 3-12, 9-12
- empaquetar
  - archivos JDataStore 2-29, 9-23
  - servidores remotos 5-4
- en secuencias de escape JDBC C-5
- encriptar archivo JDataStore 9-26
- escribir en flujos de archivos 2-16
- escrituras de disco 10-4
  - modo de confirmación por software 10-8
- evitar admisión de transacciones 3-11
- excepciones 2-4
- Explorador de JDataStore 9-1
  - añadir usuarios 9-25
  - cambiar contraseña 9-26
  - características de seguridad 9-24
  - cargar TxManager 9-11
  - como consola de consultas 9-12
  - descripción general 1-3, 9-3
  - eliminar un usuario 9-26
  - encriptar un JDataStore 9-26
  - gestionar consultas 9-14, 9-16
  - gestionar usuarios 9-24
  - guardar consultas 9-17
  - importar con 9-18
  - iniciar 9-1
  - limitaciones como consola de consultas 9-13
  - modificar los derechos de usuarios 9-26
  - operaciones de archivo 9-23
  - validar contenido 9-10
  - vistas jerárquicas 9-5, 9-7
- expresiones
  - sintaxis SQL C-11

- expresiones de selección C-21
- expresiones de selección en la sintaxis SQL C-21
- expresiones de tabla C-20
- expresiones de tabla en sintaxis SQL C-20
- expresiones de unión C-23
- expresiones de unión en la sintaxis SQL C-23
- EXTRACT (función) C-17

## F

---

- fallos
  - del sistema operativo 10-8
    - recuperar 3-6
  - eléctrico 10-8
- fecha y hora C-16
- fechas 2-10, C-16
  - extraer partes C-17
- fileExists (método) 2-12
- FileStream (objeto) 2-13
  - instanciar 2-15
- firmas de métodos sobrecargados 7-5
- flujos 2-1
  - almacenar archivos arbitrarios 2-13
  - borrar JDataStore 2-26, 9-9
  - cerrar archivo 2-16
  - comprobar B-1
    - su existencia 2-12
  - copiar 2-22, 9-9
  - crear
    - archivos 2-15
    - tabla 3-4
  - desplazar a una ubicación 2-16
  - e identificadores no repetidos 2-10
  - ejemplo para JDataStore 2-27
  - especificaciones de JDataStore A-2
  - formatos de JDataStore 2-10
  - nombrar/renombrar 2-23, 9-8
  - recuperar JDataStore 2-26, 2-27, 9-9
  - ver contenido de JDataStore 9-7
- flujos DataStore
  - comprobar B-1
- flujos de archivos 2-1
  - cerrar 2-16
  - comprobar 2-11, 2-12
  - crear 2-15
  - escritura 2-16
  - importar archivos como 9-19
  - mantener 2-26
  - obtener longitud 2-14
  - ver contenido 9-7
- flujos de datos 2-13
- flujos de escritura 2-16
- flujos de tabla 2-1
  - comprobar 2-11, 2-12

- crear 3-4
- mantener 2-26
  - ver contenido 9-7
- flujos internos 2-10
- flujos JDataStore
  - borrar 2-26, 9-9
  - convenciones de nomenclatura 2-23
  - ejemplo de borrado 2-27
  - especificaciones A-2
  - formatos 2-10
  - manipular 9-8
  - recuperar 2-26, 2-27, 9-9
  - utilizar 2-16
    - ver contenido 9-7
- formatos de hora 2-10
- funciones
  - ABSOLUTE C-16
  - CAST C-19
  - CHAR\_LENGTH C-16
  - CHARACTER\_LENGTH C-16
  - CURRENT\_DATE C-16
  - CURRENT\_TIME C-16
  - CURRENT\_TIMESTAMP C-16
  - EXTRACT C-17
  - LOWER C-17
  - POSITION C-17
  - SQRT C-18
  - SUBSTRING C-18
  - TRIM C-18
  - UPPER C-17
- funciones de escape C-6
  - cadena (funciones) C-6
  - funciones de fecha y hora C-7
  - funciones del sistema C-7
  - funciones numéricas C-7
- funciones de fecha y hora C-7
- funciones de SQL C-16
  - cadena (funciones) C-6
  - funciones de escape C-6
  - funciones de fecha y hora C-7
  - funciones del sistema C-7
  - funciones numéricas C-7
- funciones del sistema C-7
- funciones numéricas C-7

## G

---

- gestor de bloqueos 6-3
- guardar
  - históricos B-2
  - tablas JDataStore 9-17

## H

---

heuristicCompletion 6-9  
históricos 10-8

- crear para transacciones 3-10
- de transacciones 10-8
  - crear 3-10
  - guardar B-2
  - trasladar 3-11
- guardar B-2
- trasladar 3-11

hora C-16

- extraer partes C-17

## I

---

identificadores

- motor SQL de JDataStore C-10
- SQL C-10

implementación del servidor JDataStore

- personalizar 5-5

import (sentencias) 8-3importar

- archivos
  - de texto 9-18
  - JDataStore 2-14
- paquetes DataExpress 3-2
- tablas 9-14

ImportFile.java 2-13IN

- operador C-14

INDEXTABLE (sentencia) C-29índices

- crear 9-20, C-29
- eliminar de archivos JDataStore C-29
- resolución de problemas B-2
- secundarios
  - resolución de problemas B-2

iniciar

- el Explorador de JDataStore 9-1
- servidores remotos 5-3, 5-4

INSERT (sentencia) C-32IS (operador) C-13

## J

---

JDataStore

- como base de datos incrustada 3-1
- literales aceptados C-4
- reestructurar 8-6
- tutorial para serializar objetos 2-2

JdbcTable.java 3-20

## L

---

lecturas de datos no confirmados 6-1  
lecturas fantasma 6-2  
lecturas no repetibles 6-1  
lenguaje para los procedimientos almacenados y las UDF 7-1  
liberar memoria no utilizada 10-2  
LIKE (operador) C-13  
listas del directorio 2-11  
listas en sintaxis SQL C-19  
literales

- de fecha y hora C-7
- escalares C-4

LOCK TABLE (sentencia) C-35longitud

- de archivo 2-14
- de flujos de archivo 2-14

LOWER (función) C-17

## M

---

mantenimiento 1-3  
marca InputStream 2-16  
mark (método) 2-16  
matrices de bytes 2-17  
maxLogSize (propiedad) 3-10  
métodos

- appendRow 3-6
- closeDirectory 2-12
- commit 3-13
- copyStreams 2-22, 2-29
- createFileStream 2-15
- de acceso 10-6
- deleteStream 2-26
- fileExists 2-12
- mark 2-16
- openDirectory 2-7, 2-9
- openFileStream 2-16
- read 2-17
- reset 2-16
- rollback 3-13
- seek 2-16
- tableExists 2-12
- undeleteStream 2-27

modificación de tipos 8-7

- de datos 8-7

modificar datos

- con sentencias SQL C-33
- en base de datos JDataStore 3-13
- en columnas persistentes 8-8

modo

- de confirmación 10-8
- por software 10-8

de sólo lectura 3-11  
módulos de datos y componentes DataStore 10-6

## N

---

niveles de aislamiento 6-1  
definir 6-2

## O

---

objetos  
  FileStream 2-13, 2-15  
  Java  
    almacenar 2-5  
    obtener 2-6  
opciones  
  de inicio  
    de JDataStore Explorer 9-3  
    Explorador de JDataStore 9-3  
openDirectory (método) 2-7, 2-9  
openFileStream (método) 2-16  
operador (IN) C-14  
operador EXCEPT C-20  
operador EXISTS C-15  
operador INTERSECTION C-20  
operador UNION C-20  
operador UNION ALL C-20  
operadores  
  BETWEEN C-13  
  IS C-13  
  LIKE C-13  
  prioridad en consultas SQL C-11  
orden de clasificación  
  en directorios 2-11  
ordenación incorrecta B-2

## P

---

palabras clave  
  DISTINCT C-31  
  EXCEPT C-31  
  INTERSECT C-31  
  motor SQL de JDataStore C-8  
  UNION C-31  
palabras clave SQL C-8  
palabras reservadas  
  motor SQL de JDataStore C-8  
paquetes DataExpress 3-2  
parámetro implícito de conexión 7-5  
parámetros de entrada para procedimientos  
  almacenados y UDF 7-3  
parámetros de salida para UDF y procedimientos  
  almacenados 7-3  
posiciones ordinales 8-8  
POSITION (función) C-17

predicados SQL C-12  
PrintFile.java 2-16  
procedimientos almacenados  
  añadir C-30  
  asignación de tipos devueltos 7-6  
  colocar C-30  
  declaración 7-2  
  definición 7-1  
  firmas de métodos sobrecargados 7-5  
  lenguaje 7-1  
  llamar C-5, C-34  
  parámetro implícito de conexión 7-5  
  parámetros de entrada 7-3  
  parámetros de salida 7-3  
propiedades  
  ALogDir 3-11, 10-8  
  blockSize 2-15  
  BLogDir 3-11, 10-8  
  maxLogSize 3-10  
  saveMode 10-4  
proveedores y componentes DataStore 3-5

## R

---

read (método) 2-17  
recuperación de detenciones 3-6, 10-8  
  automática 3-6  
recuperar  
  flujos JDataStore 2-26, 2-27  
  con Explorador de JDataStore 9-9  
  ejemplo 2-27  
  objetos Java 2-6  
recursos  
  liberar 10-2  
  minimizar para distribuir 10-10  
redefinir tabla 8-8  
reestructurar archivos JDataStore 8-9  
Referencia de SQL C-1  
referencias a conexiones JDataStore 2-15  
rellenar tablas JDataStore 3-4, 9-14, 9-16  
renombrar flujos 2-23, 9-8  
reset (método) 2-16  
rollback (métodos) 3-13  
ROLLBACK (sentencia) C-31

## S

---

saveMode (propiedad) 10-4  
secuencias de escape de procedimiento de  
  JDBC C-5  
seek (método) 2-16  
seguridad  
  características de JDataStore 9-24  
  de JDataStore 9-24  
seleccionar filas de datos

- de tablas JDataStore C-31
- SELECT (sentencia) C-31
- SELECT INTO(sentencia) C-32
- sentencias
  - ALTER TABLE C-27
  - CALL 7-4, C-34
  - COMMIT C-30
  - CREATE INDEX C-29
  - CREATE JAVA\_METHOD 7-2, C-30
  - CREATE TABLE C-25
  - DELETE C-34
  - DROP INDEX C-29
  - DROP JAVA\_METHOD C-30
  - DROP TABLE C-29
  - import 8-3
  - INSERT C-32
  - LOCK TABLE C-35
  - ROLLBACK C-31
  - SELECT C-31
  - SELECT INTO C-32
  - SET AUTOCOMMIT C-31
  - SQL 9-22, C-1, C-11, C-19, C-20, C-21, C-23, C-25
  - UPDATE C-33
- separadores
  - de archivos de texto delimitados 9-18
  - de campo 9-18
- serialización de objetos en un JDataStore 2-2
- servidores
  - cambiar configuración 5-3
  - distribuir JDataStore 5-4
  - implementación de JDataStore 5-5
  - iniciar remotos 5-3, 5-4
  - JDBC personalizados 5-5
- servidores remotos
  - cambiar configuración 5-3
  - empaquetar 5-4
  - iniciar 5-3, 5-4
- SET AUTOCOMMIT(sentencia) C-31
- soporte XA 6-8
- SQL (sentencias) C-20, C-21, C-23, C-25
  - añadir expresiones C-11
  - ejecutar 9-22
  - ejecutar mediante JDBC C-1
  - listas en la sintaxis C-19
- SQL de JDataStore
  - identificadores C-10
  - palabras clave C-8
- SQL sin comillas
  - identificadores C-10
- SQRT (función) C-18
- StorageDataSet (clase)
  - descripción general 3-2
- StorageDataSet (componente)

- reestructurar para JDataStore 8-6
- y componentes DataStore 8-2, 10-6
- StreamVerifier (clase) B-1
- subcadenas C-17
  - obtener C-18
- SUBSTRING (función) C-18

## T

---

- tabla de directorios
  - añadir elementos 2-28
  - cerrar 2-12
  - crear 2-29
  - descripción del contenido 2-9
  - leer 2-11
- tablas
  - añadir registros C-32
  - borrar registros C-34
  - cambiar C-27
  - crear 9-19, C-25
    - autónomas 8-8
  - eliminar de archivos JDataStore C-29
  - importar 9-14
    - archivos de texto 9-18
  - independientes 8-8
  - seleccionar registros C-31
- tablas de DataStore
  - solución de problemas de ordenación incorrecta B-2
- tablas JDataStore 3-5
  - abrir en modo de sólo lectura 3-11
  - acceder 3-20
  - actualizar C-33
  - añadir registros C-32
  - bloquear C-35
  - borrar C-29
  - borrar registros C-34
  - cadena (funciones) C-5
  - cambiar C-27
  - cargar 3-4, 9-14, 9-16
  - crear C-25
  - datos persistentes 8-1
  - guardar cambios 9-17
  - literales de fecha y hora C-5
  - modificar datos 3-13
  - secuencias de escape de procedimiento de JDBC C-5
  - seleccionar registros C-31
- TableDataSet (componente)
  - como base de datos incrustada 3-1
- tableExists (método) 2-12
- tipos
  - de flujos 2-10
  - determinar 2-11, 2-12

## tipos de datos

- asignación de tipos devueltos 7-6
- clasificación de tipos para tipos Java 7-3
- conversión C-19
- portable C-2
- tipos para salidas Java 7-3

## tipos de datos Java 7-3

### transacciones 10-7

- activar con Explorador de JDataStore 9-11
- activar para archivos JDataStore 3-6, 3-8
- aspectos generales de su utilización 3-13
- confirmación C-30
- controlar 3-12, 3-14, 3-23
- de sólo lectura 6-7, 10-7
- distribuidas 6-8
- eliminar para base de datos JDataStore 3-12, 9-12
- establecer modo de confirmación
  - automática C-31
- evitar 3-11
- gestionar con TxManager 9-11
- heuristicCompletion 6-9
- modificación 3-9, 9-11
- rolling back C-31
- seguimiento 3-10
- serializables 6-3
- tutorial 3-14
- y acceso multiusuario 6-1

## trasladar históricos de transacciones 3-11

## TRIM (función) C-18

## tutoriales 2-1

- acceder a archivos JDataStore con JDBC 3-20
- bases de datos
  - acceder a archivos JDataStore con JDBC 3-20
  - bases de datos incrustadas 3-2
  - controlar transacciones 3-14
  - edición fuera de línea 8-3
  - incrustadas 3-2
- controlar transacciones 3-14
- edición fuera de línea 8-3

## TxManager (clase) 3-6

# U

---

## UDF

- añadir C-30
- asignación de tipos devueltos 7-6
- colocar C-30
- declaración 7-2
- definición 7-1
- ejemplo de una UDF 7-2
- firmas de métodos sobrecargados 7-5
- lenguaje 7-1
- parámetro implícito de conexión 7-5
- parámetros de entrada 7-3
- parámetros de salida 7-3
- undeleteStream (método) 2-27
- UPDATE (sentencia) C-33
- UPPER (función) C-17
- usuarios
  - añadir 9-25
  - editar 9-26
  - eliminar 9-26
  - gestionar 9-24
- utilidades 10-2, 10-4

# V

---

## valores literales

- SQL de JDataStore C-4
- ventana Histórico 9-10
- vía de archivos generados 2-17
- visualizadores 9-7