



**IBM**

**Using the IBM JDBC Driver  
for UniData and UniVerse**

Version 10.2  
September, 2006

IBM Corporation  
555 Bailey Avenue  
San Jose, CA 95141

Licensed Materials – Property of IBM

© Copyright International Business Machines Corporation 2006. All rights reserved.

AIX, DB2, DB2 Universal Database, Distributed Relational Database Architecture, NUMA-Q, OS/2, OS/390, and OS/400, IBM Informix®, C-ISAM®, Foundation.2000™, IBM Informix® 4GL, IBM Informix® DataBlade® module, Client SDK™, Cloudscape™, Cloudsync™, IBM Informix® Connect, IBM Informix® Driver for JDBC, Dynamic Connect™, IBM Informix® Dynamic Scalable Architecture™ (DSA), IBM Informix® Dynamic Server™, IBM Informix® Enterprise Gateway Manager (Enterprise Gateway Manager), IBM Informix® Extended Parallel Server™, i.Financial Services™, J/Foundation™, MaxConnect™, Object Translator™, Red Brick® Decision Server™, IBM Informix® SE, IBM Informix® SQL, InformiXML™, RedBack®, SystemBuilder™, U2™, UniData®, UniVerse®, wIntegrate® are trademarks or registered trademarks of International Business Machines Corporation.

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries.

Windows, Windows NT, and Excel are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries.

UNIX is a registered trademark in the United States and other countries licensed exclusively through X/Open Company Limited.

Other company, product, and service names used in this publication may be trademarks or service marks of others.

This product includes cryptographic software written by Eric Young (eay@cryptosoft.com).

This product includes software written by Tim Hudson (tjh@cryptosoft.com).

Documentation Team: Claire Gustafson, Shelley Thompson

#### US GOVERNMENT USERS RESTRICTED RIGHTS

Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

# Table of Contents

<b>Chapter 1</b>	<b>Introduction to IBM JDBC Driver for UniData and UniVerse</b>	
	What is JDBC . . . . .	1-4
	What is a JDBC Driver . . . . .	1-6
	Overview of JDBC. . . . .	1-7
	IBM JDBC for UniData and UniVerse Architecture . . . . .	1-8
<b>Chapter 2</b>	<b>Installing and Setting Up the IBM JDBC Driver for UniData and UniVerse</b>	
	Hardware and Software Requirements . . . . .	2-3
	Installing the IBM JDBC Driver for UniData and UniVerse . . . . .	2-5
	Installing on Windows Platforms. . . . .	2-5
	Installing on UNIX . . . . .	2-6
<b>Chapter 3</b>	<b>Accessing UniData Data</b>	
	Verifying That UniRPC Is Running . . . . .	3-3
	UCI Connection Timeout Configuration . . . . .	3-4
	Making UniData Accounts Accessible . . . . .	3-5
	Tracing Events . . . . .	3-5
	Presenting Data in JDBC–Accessible Format . . . . .	3-7
	Data Types . . . . .	3-7
	Multivalued and Multi-Subvalued Data . . . . .	3-7
<b>Chapter 4</b>	<b>Accessing UniVerse Data</b>	
	Accessing UniVerse Tables and Files . . . . .	4-3
	Tables . . . . .	4-3
	Files. . . . .	4-4
	Multivalued Data . . . . .	4-4
	The TABLES Rowset . . . . .	4-6
	The UniVerse Server Administration Menu . . . . .	4-10
	Making Files Visible to JDBC Applications . . . . .	4-12

Making Files Visible . . . . .	4-12
Restricting File Visibility . . . . .	4-13
Updating the File Information Cache . . . . .	4-14
Removing File Visibility . . . . .	4-16
COLUMNS Rowset . . . . .	4-18
Association Keys . . . . .	4-19
Table and Column Names Containing Special Characters . . . . .	4-22
Delimited Identifiers . . . . .	4-22
Making UniVerse Data Meaningful to JDBC . . . . .	4-23
SQL Data Types . . . . .	4-23
Length of Character String Data . . . . .	4-25
Empty-Null Mapping . . . . .	4-26
Validating and Fixing Tables and Files . . . . .	4-26
Scalar Functions . . . . .	4-29

## Chapter 5      **Programming with JDBC**

Loading the IBM JDBC Driver for UniData and UniVerse . . . . .	5-3
Creating a Connection . . . . .	5-4
Format of Database URLs . . . . .	5-5
Accessing Database MetaData. . . . .	5-7
Querying the Database . . . . .	5-8
Mapping Data Types . . . . .	5-9
Data Conversion . . . . .	5-10
Handling Errors . . . . .	5-11
Handling Transactions . . . . .	5-12
Isolation Levels . . . . .	5-13
JDBC 2.0 Optional Package Support. . . . .	5-14
Tuning the Connection Pool . . . . .	5-18
Restrictions and Limitations . . . . .	5-23
Unsupported Data Types . . . . .	5-23
Unsupported Methods . . . . .	5-23

## Chapter 6      **JDBC Scrollable Cursors**

JDBC Scrollable Cursors . . . . .	6-2
Creating a Scrollable Cursor. . . . .	6-2
Methods for Moving the Cursor. . . . .	6-3
Connection URL Parameters . . . . .	6-4
Example. . . . .	6-5

<b>Chapter 7</b>	<b>IBM for UniData and UniVerse JDBC Code Examples</b>	
	UniJDBC Example Program - List File . . . . .	7-3
	UniJDBC Short Examples . . . . .	7-7
	Select Example . . . . .	7-7
	UniJDBC Prepared Statement Example. . . . .	7-9
	UniJDBC Callable Statement Example . . . . .	7-10
	UniJDBC DatabaseMetaData Example . . . . .	7-11

---

# Introduction to IBM JDBC Driver for UniData and UniVerse

What is JDBC. . . . .	1-4
What is a JDBC Driver . . . . .	1-6
Overview of JDBC . . . . .	1-7
IBM JDBC for UniData and UniVerse Architecture . . . . .	1-8

This manual provides an overview of JDBC technology and how the IBM JDBC Driver for UniData and UniVerse uses it to expose UniData and UniVerse extended relational data. This manual also describes how to configure the IBM JDBC Driver for UniData and UniVerse, and provides information about the JDBC functionality that it supports. From this information, developers can design JDBC-based applications that can access and manipulate data in UniData and UniVerse data stores.

The IBM JDBC Driver for UniData and UniVerse is an implementation of the JDBC 1.0 specification with limited functionality supported for the JDBC 2.0 specification.

This manual is organized in the following way:

- Chapter 1, “[Introduction to IBM JDBC Driver for UniData and UniVerse](#),” introduces you to key concepts about JDBC, and shows you how JDBC fits in with other components in an enterprise network. This chapter also provides you with the hardware and software requirements for clients and servers that use JDBC.
- Chapter 2, “[Installing and Setting Up the IBM JDBC Driver for UniData and UniVerse](#),” describes how to install and configure the driver.
- Chapter 3, “[Accessing UniData Data](#),” describes how to make accounts in UniData databases accessible to JDBC applications and how to normalize multivalued and multi-subvalued data in UniData. Use this manual in conjunction with *Using VSG and the Schema API*. It describes:
  - Visual Schema Generator (VSG) – A Windows-based Graphical User Interface (GUI) tool for making data in UniData accessible to first normal form (1NF) applications.
  - Schema API – An application programming interface (API) that consists of a series of UniBasic subroutines designed to accomplish the same tasks as VSG.

***Note:** IBM recommends that you use VSG to prepare files because it is easier and faster to use than the Schema API. It performs several processes automatically. You might want to use the Schema API if you have a large number of files.*

- Chapter 4, “[Accessing UniVerse Data](#),” describes how to make data in UniVerse schemas and accounts accessible to JDBC applications.
- Chapter 5, “[Programming with JDBC](#),” describes how the IBM JDBC Driver for UniData and UniVerse supports the JDBC API.
- Chapter 6, “[JDBC Scrollable Cursors](#),” describes how to use scrollable cursors with the IBM JDBC Drive for UniData and UniVerse.



- Chapter 7, “[IBM for UniData and UniVerse JDBC Code Examples](#),” contains example programs for the IBM JDBC Driver for UniData and UniVerse.



---

## What is JDBC

Java database connectivity (JDBC) is the JavaSoft specification of a standard application programming interface (API) that allows Java programs to access database management systems. The JDBC API consists of a set of interfaces and classes written in the Java programming language.

Using these standard interfaces and classes, programmers can write applications that connect to databases, send queries written in structured query language (SQL), and process the results.

The JDBC API is consistent with the style of the core Java interfaces and classes, such as `java.lang` and `java.awt`. The following table describes the interfaces, classes, and exceptions that make up the JDBC API.

Interface, Class, or Exception	Description
<code>java.sql.CallableStatement</code>	Interface used to execute stored procedures.
<code>java.sql.Connection</code>	Interface used to establish a connection to a database. SQL statements run within the context of a connection.
<code>java.sql.DatabaseMetaData</code>	Interface used to return information about the database.
<code>java.sql.Driver</code>	Interface used to locate the driver for a particular database management system.
<code>java.sql.PreparedStatement</code>	Interface used to send pre-compiled SQL statements to the database server and obtain results.
<code>java.sql.ResultSet</code>	Interface used to process the results returned from executing an SQL statement.
<code>java.sql.ResultSetMetaData</code>	Interface used to return information about the columns in a <code>ResultSet</code> object.
<code>java.sql.Statement</code>	Interface used to send static SQL statements to the database server and obtain results.
<code>java.sql.Date</code>	Subclass of <code>java.util.Date</code> used for the SQL DATE data type.
<code>java.sql.DriverManager</code>	Class used to manage a set of JDBC drivers.

Interface, Class, or Exception	Description
<code>java.sql.DriverPropertyInfo</code>	Class used to discover and supply properties to a connection.
<code>java.sql.Time</code>	Subclass of <b>java.util.Date</b> used for the SQL TIME data type.
<code>java.sql.TimeStamp</code>	Subclass of <b>java.util.Date</b> used for the SQL TIMESTAMP data type.
<code>java.sql.Types</code>	Class used to define constants that are used to identify standard SQL data types, such as VARCHAR, INTEGER, and DECIMAL.
<code>java.sql.String</code>	Class used to identify long text data types such as LVARCHAR.
<code>java.sql.DataTruncation</code>	Exception thrown or warning reported when data has been truncated.
<code>java.sql.SQLException</code>	Exception that provides information about a database error.
<code>java.sql.SQLWarning</code>	Warning that provides information about a database warning.

Since JDBC is a standard specification, one Java program that uses the JDBC API can connect to any database management system (DBMS), as long as a driver exists for that particular DBMS.

For more information about the JDBC API, visit the JavaSoft Web Site at:

<http://java.sun.com/>

---

## What is a JDBC Driver

This section discusses the JDBC API specification definition of JDBC drivers. For more information, see <http://java.sun.com/docs/books/jdbc/intro.html>.

The JDBC API consists of a set of classes and interfaces written in the Java programming language that provide a standard API for database developers, and makes it possible to write robust database applications using an all-Java API. A JDBC driver implements these interfaces and classes for a particular DBMS vendor.

A Java program that uses the JDBC API loads the specified driver for a particular DBMS before it actually connects to a database. The JDBC Driver-Manager class then sends all JDBC API calls to the loaded driver.

There are four types of JDBC drivers:

1. **JDBC-ODBC bridge plus ODBC driver (also called Type 1):** This type of driver translates JDBC API calls into ODBC calls that are then passed to the ODBC driver. ODBC binary code, and in many cases database client code, must be loaded on each client machine that uses this driver. As a result, this kind of driver is most appropriate on a corporate network where client installations are easily managed, or for application server code written in Java in a three tier architecture.
2. **Native-API, partly java driver (also called Type 2):** This kind of driver converts JDBC API calls into DBMS-specific client API calls. Like the bridge driver, this type of driver requires that some binary code be loaded on each client computer.
3. **JDBC-Net, pure-Java driver (also called Type 3):** This driver translates JDBC calls into a DBMS-independent net protocol, which is then translated to a DBMS protocol by a server.
4. **Native-Protocol, pure-Java driver (also called Type 4):** This type of driver converts JDBC API calls directly into the DBMS-specific network protocol without a middle tier. This allows the client applications to connect directly to the database server.

The IBM JDBC Driver for UniData and UniVerse is a Type 4, Native-Protocol, driver.

---

## Overview of JDBC

The IBM JDBC Driver for UniData and UniVerse enables JDBC applications to connect to the UniData and UniVerse Relational Database Management Systems (RDBMS).

The IBM JDBC Driver for UniData and UniVerse is a Type 4 or Native Protocol type driver and is an implementation of the JDBC 1.0 specification with support for some 2.0 functionality.

A JDBC application sends a connection request to the IBM JDBC Driver for UniData and UniVerse. The driver receives the request, and establishes a connection to the UniData or UniVerse RDBMS. Then the JDBC application can access the RDBMS. When the process is complete, the JDBC application sends a disconnection request to the IBM JDBC Driver for UniData and UniVerse, and disconnects from the RDBMS.

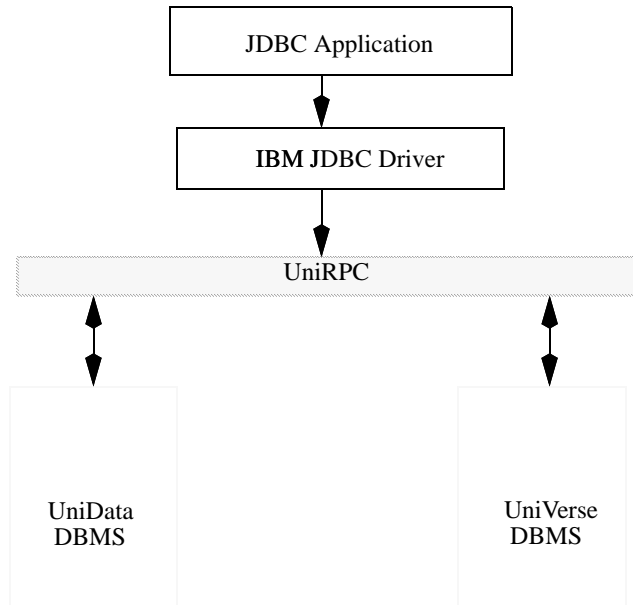
In order to access the data from a JDBC application, the data must conform to the 1NF standard, or be “normalized.”

UniVerse dynamically normalizes its own data. For more information, see Chapter 4, [“Accessing UniVerse Data.”](#)

UniData also normalizes its own data, but you must prepare the data for it to do so. To prepare the data, you must use VSG, a GUI tool that generates schema on UniData files, or the Schema API. For more information about using VSG or the Schema API, see Chapter 3, [“Accessing UniData Data,”](#) and *Using VSG and the Schema API*.

---

## IBM JDBC for UniData and UniVerse Architecture



---

# Installing and Setting Up the IBM JDBC Driver for UniData and UniVerse

Hardware and Software Requirements . . . . .	2-3
Installing the IBM JDBC Driver for UniData and UniVerse . . . . .	2-5
Installing on Windows Platforms . . . . .	2-5
Installing on UNIX. . . . .	2-6

This chapter describes how to prepare both a client and server machine to run IBM JDBC for UniData and UniVerse.

---

## Hardware and Software Requirements

The following lists provide the hardware and software requirements for client and server machines that use the IBM JDBC for UniData and UniVerse.

### *Client Requirements*

#### *Windows Client*

- IBM-compatible personal computer (PC) attached to a network.
- Either of the following operating systems:
  - Windows platforms NT 4.0 (Service Pack #6) or later (for example, Windows 2000, Windows XP, and so forth)
- Approximately 20 MB of free disk space.
- A minimum of 64 MB of random access memory (RAM).
- TCP/IP.
- Java 1.2 or greater must be installed.
- If you wish to use any of the extended options from the JDBC 2.0 Optional API, such as connecting through a data source or the use of connection pooling, you must obtain an implementation of the Java Directory and Naming Interface (JNDI). You need to download and install the JDBC Optional Package from Javasoft, which includes a required import package (javax.sql.\*) for any JDBC application making use of the Optional API. Also required are the JNDI 1.2.1 class libraries (javax.naming.\*), which you can download if you are not using JDK 1.3 or higher. If you are using JDK 1.3 or higher, the JNDI libraries are included.

#### *UNIX Client*

- Java 1.2 or greater must be installed

### *Server Requirements*

- UNIX, Windows NT or higher (for example, Windows 2000, Windows XP, and so forth).
- TCP/IP.



- Either of the following IBM Relational Database servers:
  - UniData 6.0 or later
  - UniVerse 10.0 or later

# Installing the IBM JDBC Driver for UniData and UniVerse

## Installing on Windows Platforms

1. If you are installing the IBM JDBC Driver for UniData and UniVerse from a CD-ROM, load the disc into the CD-ROM drive.
2. A menu appears on the screen. Select UniDK. The installation wizard prompts for several options. When presented with which components of the UniDK you wish to install, make sure you select both the **JDBC** and **UniObjects For Java** options. Click **Next**.

***Note:** If you already have a version of the UniDK installed, the installation process prompts to Repair, Modify or Remove the installation of the UniDK. Select Remove to uninstall the existing version. Once you have uninstalled the older version of the UniDK, return to step 1.*

3. When the installation has completed, you will be prompted to restart your computer. Close all other programs, then click **Yes**.
4. You must now set the CLASSPATH environment variable to include UniJDBC.jar and asjava.zip. Open a Command Prompt session. At the prompt, type:

```
set
CLASSPATH=C:\IBM\UniClient\UniDK\jdbc\lib\unijdbc.jar;C
:\IBM\UniClient\UniDK\uojsdk\lib\asjava.zip;%CLASSPATH%
```

Note that the path used in this example is from a default UniDK installation. If you chose to install the UniDK in a different path, you will need to adjust your CLASSPATH accordingly.

***Note:** If you want to make connections using the DataSource or ConnectionPoolDataSource objects, you must also install the JDBC 2.0 Optional Package. You also need an implementation of the Java Naming and Directory Interface (JNDI). You can acquire these from Sun Microsystems' web site.*

<http://java.sun.com/products/jdbc>

<http://java.sun.com/products/jndi>

See Chapter 5, "[JDBC 2.0 Optional Package Support](#)," for more information.



## Installing on UNIX

The following section describes how to install JDBC on UNIX systems.

### *The JDBC Client and Database Server are on the Same Machine*

If you plan to run your JDBC client on the same machine that is running the UniData or UniVerse database server, follow the instructions below.

A standard installation of either the UniData or UniVerse database servers includes the required files for JDBC. They are located in the *unishared* directory. After the database server has been installed, type the following command.

```
cat /.unishared
```

This command tells you the path to the unishared directory. The JDBC files located in this directory have the following structure:

```
/unishared
  /jdbc/docs/unijdbcjavadoc.zip
  /jdbc/lib/unijdbc.jar
  /jdbc/samples/jdbcsample.zip
  /uojskd/lib/asjava.zip
```

Another required file, asjava.zip is located in the UniObjects for Java installation in the path shown above. The files and their functions are as follows:

- **unijdbc.jar** - The main driver file for the IBM Driver for UniData and UniVerse.
- **asjava.zip** - Contains required libraries for UniJDBC and UniObjects for Java
- **unijdbcjavadoc.zip** - The javadoc for the UniJDBC driver code.
- **jdbcsample.zip** - A sample UniJDBC program for reference.

Set your classpath environment variable to include unijdbc.jar and asjava.zip. For example, enter the following command when using ksh:

```
export
CLASSPATH=/unishared/jdbc/lib/unijdbc.jar:/unishared/uojskd/lib/as
java.zip:$CLASSPATH
```

### ***The JDBC Client and Database Server are on Different Machines***

If you plan to run your JDBC on a UNIX machine other than the one where UniData or UniVerse are installed, you must copy the `unijdbc.jar` and `asjava.zip` files to the client machine, and set the classpath accordingly. Refer to the [“Installing on Windows Platforms”](#) on page 2-5 or [“Installing on UNIX”](#) on page 2-6 for the location of these files on Windows and UNIX installations, respectively.

After you copy the files to your UNIX client machine, you must set your classpath. See the [“Installing the IBM JDBC Driver for UniData and UniVerse”](#) on page 2-5 for further details about setting the classpath.

Once you copy the files and set the classpath, you are ready to get started.

---

# Accessing UniData Data

Verifying That UniRPC Is Running . . . . .	3-3
UCI Connection Timeout Configuration . . . . .	3-4
Making UniData Accounts Accessible . . . . .	3-5
Tracing Events . . . . .	3-5
Presenting Data in JDBC–Accessible Format . . . . .	3-7
Data Types . . . . .	3-7
Multivalued and Multi-Subvalued Data . . . . .	3-7

This chapter describes how to access data in UniData tables and files. You must perform the following tasks:

- Verify that the UniRPC daemon (for UNIX systems) or the UniRPC service (for Windows platforms) is running.
- Make UniData accounts accessible to JDBC applications.
- Present the data in UniData in a format that is accessible to JDBC applications.

---

## Verifying That UniRPC Is Running

When JDBC requests a connection to the UniData UCI server, the UniRPC facility starts the server (udserver) process, which runs on the machine where the database resides. Before you use JDBC, make sure that UniRPC is running.

On UNIX systems, you can verify whether UniRPC is running by entering the following command:

```
ps -ef | grep unirpc
```

If you need to start UniRPC for UNIX, use the UniData *startud* command. For more information, see *Administrative Supplement for APIs* included with both the UniData and UniVerse documentation sets.

On Windows platforms, you can verify whether UniRPC is running by double-clicking the Services icon on the Windows Control Panel. The Services dialog box appears. If UniRPC is running, it appears in the list of services with the status “Started.” If UniRPC is not running, you can start it from the Services dialog box.

UniRPC is installed automatically when you install UniData.

On UNIX systems, the UniRPC services file (unirpcservices) contains an entry that is similar to the following example:

```
udserver /usr/ud71/bin/udsrvd * TCP/IP 0 3600
```

The unirpcservices file is located, by default, on the UCI server in the *.../unishared/unirpc* directory. This directory is located in a path that is parallel with the *udthome* directory. For example, if the path to *udthome* is */disk1/udthome*, the path to the unirpcservices file is */disk1/unishared/unirpc*.

On Windows platforms, the unirpcservices file is located on the UCI server in the following default path:

```
x:\IBM\unishared\unirpc
```

where *x* is the drive on which the software is installed. It contains an entry that is similar to the following example:

```
udserver C:\usr\ud71\bin\udsrvd.exe * TCP/IP 0 3600
```

Your unishared directory might not be located in the default path. To determine its actual location on UNIX systems, enter:

```
cat /.unishared
```

This provides the path for the unishared directory.

For Windows platforms, you can find the path for unishared by looking in the system registry in \HKEY\_LOCAL\_MACHINE\SOFTWARE\IBM\unishared.

When the client system requests a connection to a UCI server, the local UniRPC daemon or service uses the unirpcservices file to verify that the client can start the requested server (in this case, udsrvr).

With the Server edition of UniData, each JDBC connection to a UniData UCI server consumes one UniData license. With the Workstation, Workgroup, or Enterprise edition of UniData, device licensing enables each client system to establish up to ten connections to the UCI server from one device while consuming only one database license — regardless of whether more than one user login is used to make the connections. For more information about device licensing, see *Installing and Licensing UniData Products* and the *Administrative Supplement for APIs*.

## UCI Connection Timeout Configuration

The six-minute default inactivity timeout value (3600) for UCI connections can be too short. If users leave JDBC connections open, but the connections remain inactive for longer than six minutes, they could receive UniRPC error code 81015. To increase this timeout value, use any text editor to modify the unirpcservices file (for the path to this file, see [“Verifying That UniRPC Is Running”](#) on page 3-3).



**Note:** To edit the unirpcservices file, you must have root privileges on UNIX or Administrator privileges on Windows platforms.

For example, to set the connection timeout delay to 24 hours, in the line starting with udsrvr, change the right-most number to 864000 (the number of seconds in 24 hours). The line could appear as follows:

```
udsrvr /usr/ud61/bin/udsvrd * TCP/IP 0 864000
```



---

## Making UniData Accounts Accessible

UniData databases are organized into accounts. A JDBC application connects to a UniData account and can access the files there. You optionally can define the account as a database in the `ud_database` file on the server machine.

You can specify the **account** under the account property of the **connection URL**.

If you want to access an account that has a UDTHOME directory different than the default UDTHOME directory, you must include a definition for that account in the `ud_database` file on the server machine. For UNIX systems, this file is located in the `/usr/udnn/include` path, where *nn* is the release of UniData you are running. For Windows platforms, it is located in `\udthome\include`. You can find the path for *udthome* by looking in the system registry under `\HKEY_LOCAL_MACHINE\SOFTWARE\IBM\UniData\7.1`. Use any text editor to modify the `ud_database` file.

***Note:** To determine your default UniData home directory, use the UNIX `env` command. The results of this command include the default setting for the UDTHOME environment variable.*

The following Windows example shows an entry in the `ud_database` file for a database named `db2`:

```
DATABASE=db2
UDTHOME=d:\disk2\test71
UDTACCT=d:\disk2\test71\testacct.
```

In the `ud_database` file entry, the UDTHOME parameter is optional. You should include it only when the UDTHOME directory is different than the default UDTHOME directory.

## Tracing Events

By using the tracing feature, you can create logs of events between clients and the database through the server. Logs enable IBM support personnel to help troubleshoot problems. You can define trace levels for database entries in the `ud_database` file.



The following table describes the valid trace levels and the associated information that is written to the trace log.

Trace Level	Description
0	Includes all fatal error information.
1	Includes all UCI commands in addition to the information provided by trace level 0.
2	Includes parameter information and column descriptions in addition to the information provided by trace levels 0 and 1.
3	Includes data values in addition to the information provided by trace levels 0, 1, and 2.

#### Trace Levels

The trace log is named `udsrv_database.processID` where *database* is the database name (for example, db2) and *processID* is the process number ID. By default, it is located in your temporary directory (typically, /tmp for UNIX; x:\temp for platforms). For UNIX, you can find the location of the trace log file in the TMP parameter in the `udtconfig` file under `/usr/ud71/include`.

The following UNIX example shows a tracing level setting for a database named `dbase3`:

```
DATABASE=dbase3
UDTHOME=/disk1/ud71
UDTACCT=/home/username/test
TRACE_LEVEL=3
```

---

## Presenting Data in JDBC–Accessible Format

Data in UniData is organized differently from the way JDBC expects it to be organized. Two areas in which the data differs from what JDBC expects are:

- Data types
- Multivalued and multi-subvalued data

### Data Types

UniData does not define data types for data contained in its files. On the other hand, JDBC expects data types for all data. In addition, data in UniData can be of variable length, but JDBC expects data to have either a fixed or a maximum length. To make the data look more like what JDBC expects, you must use VSG or the Schema API.

### Multivalued and Multi-Subvalued Data

JDBC expects data to be organized in first normal form (1NF) format. Although some files could be in 1NF format, which means that only one value is stored in each column of each row, many UniData files have columns that store multiple values in the columns of a row (NF<sup>2</sup> format). To instruct UniData SQL to present data in 1NF format, you must use VSG or the Schema API.

VSG or the Schema API create views and subtables that present multivalued and multi-subvalued data in 1NF format. These views and subtables do not duplicate data, but merely instruct UniData SQL to normalize data before returning it to the application. JDBC passes the 1NF data to the Java application in the form of rowsets.

### *Visual Schema Generator and the Schema API*

There are two ways to create 1NF views and subtables to instruct UniData SQL to present data in UniData in a JDBC–accessible format:

- VSG
- Schema API

VSG is a Windows-based graphical user interface (GUI) tool that makes UniData files accessible to other applications through JDBC. It enables you to create, delete, and modify UniData SQL subtables and views. VSG also enables you to set ANSI privileges (security) for tables, subtables, and views.

VSG guides data administrators through the process of defining the 1NF subtables and views that represent the extended relations by visually presenting all available configuration options. It also translates conversion specifications for UniData to SQL data types. VSG performs logical error checking through every step of the schema generation process.

The Schema API consists of a series of UniBasic subroutines designed to accomplish the same normalization tasks as VSG.

IBM recommends that you use VSG to prepare files because it is easier and faster to use than the Schema API. It performs several processes automatically. You might want to use the Schema API if you have a large number of files (in this case, several VSG processes take several minutes to complete) or you deploy data files with your applications.

For more information about using VSG or the Schema API, see *Using VSG and the Schema API*.

# Accessing UniVerse Data

Accessing UniVerse Tables and Files . . . . .	4-3
Tables . . . . .	4-3
Files . . . . .	4-4
Multivalued Data . . . . .	4-4
The TABLES Rowset . . . . .	4-6
The UniVerse Server Administration Menu . . . . .	4-10
Making Files Visible to JDBC Applications . . . . .	4-12
Making Files Visible . . . . .	4-12
Restricting File Visibility . . . . .	4-13
Updating the File Information Cache . . . . .	4-14
Removing File Visibility . . . . .	4-16
COLUMNS Rowset . . . . .	4-18
Association Keys . . . . .	4-19
Table and Column Names Containing Special Characters . . . . .	4-22
Delimited Identifiers . . . . .	4-22
Making UniVerse Data Meaningful to JDBC . . . . .	4-23
SQL Data Types . . . . .	4-23
Length of Character String Data . . . . .	4-25
Empty-Null Mapping . . . . .	4-26
Validating and Fixing Tables and Files . . . . .	4-26
Scalar Functions . . . . .	4-29

This chapter describes how to access data in UniVerse tables and files. The following data is always accessible to a JDBC application connected to a UniVerse account:

- All UniVerse tables on the system.
- All non-SQL UniVerse files defined in the VOC file of the current account.

In this chapter, the word *tables* refers to UniVerse tables defined by the CREATE TABLE statement, and views defined by the CREATE VIEW statement. The word *files* refers to non-SQL UniVerse files that are created by the CREATE.FILE command. The term *JDBC table* refers to any table or file, real or virtual, that is accessible to JDBC applications.

Before a JDBC application attempts to connect to a UniVerse data source, the UniRPC daemon (for UNIX systems) or the UniRPC service (for Windows platforms) must be running on the server system.

For information about setting up data sources, see Chapter 2, [“Installing and Setting Up the IBM JDBC Driver for UniData and UniVerse.”](#)

---

## Accessing UniVerse Tables and Files

A JDBC application connects to a UniVerse account (which can be a schema) and can access all tables on the system and all files defined in the VOC file of that account.

Sometimes a JDBC application connected to UniVerse gets a list of JDBC tables by requesting a TABLES rowset. This rowset lists:

- All tables on the system.
- Files defined in the current account's VOC file that have been made visible to JDBC.
- Virtual tables (dynamically normalized multivalued data) within the above.

We refer to the JDBC tables listed in the TABLES rowset as “visible” to JDBC applications. Although all SQL tables on the system are, by definition, visible, non-SQL files that have not been made visible are not included in the TABLES rowset. For information about how to make files visible in an account, see [“Making Files Visible to JDBC Applications”](#) on page 4-12.

### Tables

When connected to a UniVerse account, a JDBC application can access all tables on the system. If connected to a UniVerse schema, the JDBC application can reference tables in that schema by using unqualified table names in SQL statements. All other tables on the system can be referenced using table names qualified by the appropriate schema name.

#### *Example 1*

In this example the JDBC application is connected to schema HR. If table EMPLOYEES is in schema HR, you can reference it simply as EMPLOYEES. If view EMPLOYEES is in schema DENVER, you must reference it as DENVER.EMPLOYEES.

### ***Example 2***

In this example, the JDBC application is connected to account SALES that is not a schema. In this case, you must reference table EMPLOYEES (in schema HR) and view EMPLOYEES (in schema DENVER) as HR.EMPLOYEES and DENVER.EMPLOYEES respectively.

## **Files**

A JDBC application can access files that are defined in the VOC file of the account to which the JDBC application is connected. Because some JDBC applications are written to access only tables listed in the TABLES rowset, as a practical matter UniVerse files may not be usable unless they have been made visible for that account.

For example, if a JDBC application connects to an account whose files have been made visible, all files defined in the account's VOC file, except for system files (such as &SAVEDLISTS&) and UV/Net files, are visible. The JDBC application can reference the files using the file names in the VOC. The account does not need to be a schema.

On the other hand, if a JDBC application connects to an account whose files have not been made visible, no files appear in the TABLES rowset. However, the JDBC application can still access files defined in the account's VOC file.

## **Multivalued Data**

JDBC applications expect data to be organized relationally in first normal form (1NF). Although some UniVerse tables and files are in first normal form, which means only one value is stored in each column of each row, many UniVerse tables and files have columns that store multiple values.

UniVerse always presents multivalued data to JDBC in first normal form (1NF), and JDBC passes it to the java applications in the form of rowsets. UniVerse automatically normalizes its tables and files by a process called "dynamic normalization." This means that a file containing multivalued data appears to JDBC as several 1NF tables, each consisting of singlevalued data only.



### *Example*

The table **ORDERS** is defined with columns **ORDERNUM**, **CUSTNUM**, **DATE**, **PART**, and **QTY**. **PART** and **QTY** are multivalued and make up an association called **ITEMS**. The association key is **PART**. Suppose this table contains the following orders.

ORDERNUM	CUSTNUM	DATE	PART	QTY
99101	12345	5/25/99	HINGE	200
			BOLT	650
99102	98765	6/10/99	BOLT	50

**ORDERS Table**

This file appears to JDBC as two 1NF tables called **ORDERS** and **ORDERS\_ITEMS**. The **ORDERNUM** column is the **ORDERS** key.

ORDERNUM	CUSTNUM	DATE
99101	12345	5/25/99
99102	98765	6/10/99

**ORDERS 1NF Table**

The **ORDERNUM** and **PART** columns are the two columns that make up the **ORDERS\_ITEMS** key.

ORDERNUM	PART	QTY
99101	HINGE	200
99101	BOLT	650
99102	BOLT	50

**ORDERS 1NF Table**

## **The TABLES Rowset**

A JDBC application can request a **TABLES** rowset that includes all tables and visible files. For each table and visible file, this rowset includes schema name, table name, and table type. Table type can be **TABLE**, **VIEW**, or **SYSTEM TABLE**.

### ***Example 1***

Suppose the JDBC application is connected to a UniVerse account that is not a schema, and suppose the account contains two files called EMPS and DEPTS. EMPS and DEPTS are not tables. EMPS has one multivalued column called DEPENDENTS, and DEPTS has no multivalued columns. Files in this account have been made visible.

Suppose there is another UniVerse account whose files have been made visible, which contains a file called OTHERFILE.

If the JDBC application requests the TABLES rowset, it includes the following information.

Schema Name	Table Name	Table Type
<null value>	EMPS	TABLE
<null value>	EMPS_DEPENDENTS	TABLE
<null value>	DEPTS	TABLE

**Example 1: TABLES Rowset Information**

This shows that:

- Any files listed in the TABLES rowset must be in the account to which the JDBC application is connected (OTHERFILE does not appear).
- Files listed in the TABLES rowset have a null value for the schema name.
- A multivalued column appears as a separate dynamically-normalized table whose type is TABLE and whose name is composed of the original file's name followed by an underscore and the name of the column.

### ***Example 2***

Suppose the above account is made into a schema named USA. Suppose two tables called CITIES and STATES are then created where CITIES has only singlevalued columns and STATES has an association called COUNTIES.

The TABLES rowset now includes the following returned information.

Schema Name	Table Name	Table Type
<null value>	EMPS	TABLE
<null value>	EMPS_DEPENDENTS	TABLE
<null value>	DEPTS	TABLE
USA	CITIES	TABLE
USA	STATES	TABLE
USA	STATES_COUNTIES	TABLE

**Example 2: TABLES Rowset Information**

This shows that:

- Tables have a schema name, unlike files.
- An association appears as a dynamically-normalized table whose type is TABLE and whose name is composed of the original file's name followed by an underscore and the name of the association.

### ***Example 3***

Suppose an SQL view called STATEVIEW is created in another schema (called JOESCHEMA) with the following SQL statement:

```
CREATE VIEW STATEVIEW AS SELECT * FROM USA.STATES;
```

While still connected to the original account, the JDBC application sees the following information returned in the TABLES rowset.

Schema Name	Table Name	Table Type
<null value>	EMPS	TABLE
<null value>	EMPS_DEPENDENTS	TABLE
<null value>	DEPTS	TABLE
USA	CITIES	TABLE

**Example 3: TABLES Rowset Information**

Schema Name	Table Name	Table Type
USA	STATES	TABLE
USA	STATES_COUNTIES	TABLE
JOESCHEMA	STATEVIEW	VIEW
JOESCHEMA	STATEVIEW_COUNTIES	VIEW

**Example 3: TABLES Rowset Information (Continued)**

This shows that:

- Tables and views in other schemas appear in the TABLES rowset.
- Views and their dynamically-normalized associations and multivalued columns have the table type VIEW.

### ***Example 4***

Suppose the account JOESCHEMA mentioned above contains a UniVerse file called JOEFILE that has not been made visible. Suppose a JDBC application connects to JOESCHEMA and requests the TABLES rowset. This rowset includes the following information.

Schema Name	Table Name	Table Type
USA	CITIES	TABLE
USA	STATES	TABLE
USA	STATES_COUNTIES	TABLE
JOESCHEMA	STATEVIEW	VIEW
JOESCHEMA	STATEVIEW_COUNTIES	VIEW

**Example 4: TABLES Rowset Information**

This shows:

- Files in other accounts (such as EMPS) do not appear in the TABLES rowset.
- Files in the account to which the JDBC application is connected (such as JOEFILE) do not appear if they have not been made visible.

### ***Example 5***

The UniVerse SQL catalog tables also appear in the TABLES rowset. Their table type is SYSTEM TABLE.

Schema Name	Table Name	Table Type
CATALOG	UV_ASSOC	SYSTEM TABLE
CATALOG	UV_COLUMNS	SYSTEM TABLE
CATALOG	UV_COLUMNS_ACOL_NO	SYSTEM TABLE
CATALOG	UV_COLUMNS_AMC	SYSTEM TABLE
CATALOG	UV_SCHEMA	SYSTEM TABLE
CATALOG	UV_TABLES	SYSTEM TABLE

**Example 5: TABLES Rowset Information**

---

## The UniVerse Server Administration Menu

You can use the UniVerse Server Administration menu on the server system to perform certain administrative functions that may facilitate access to your UniVerse data from JDBC applications. These functions include making UniVerse files in an account visible to JDBC, and detecting data anomalies in a table or file. Tasks described later in this chapter use selections that appear on this menu.

To display the menu:

1. Log on to the server system as root on UNIX or as an Administrator on platforms.
2. Invoke UniVerse and log to the HS.ADMIN account. This account is in a subdirectory named HS.ADMIN in the UV account directory.
3. Type the command: HS.ADMIN.

The UniVerse Server Administration menu appears, as shown in the following example.

```
UniVerse Server Administration
```

- ```
1. List activated accounts
2. Show UniVerse ODBC Config configuration for an account
3. Activate access to files in an account
4. Deactivate access to files in an account
5. Run HS.SCRUB on a File/Table
6. Update File Information Cache in an account
```

```
Which would you like? ( 1 - 6 )
```

1. To exit the menu, press ENTER.

The following table describes the selections that appear on the UniVerse Server Administration menu.

| Menu Selection                                            | Description                                                                                                                                                                         |
|-----------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 1. List activated accounts                                | Lists UniVerse accounts whose files have been made visible to JDBC. When a JDBC application is connected to such an account, the account's files are included in the TABLES rowset. |
| 2. Show UniVerse ODBC Config configuration for an account | Does not apply to JDBC.                                                                                                                                                             |
| 3. Activate access to files in an account                 | Makes files in a UniVerse account visible to JDBC so that when a JDBC application is connected to this account, its files are included in the TABLES rowset.                        |
| 4. Deactivate access to files in an account               | Removes JDBC visibility from files in a UniVerse account. This reverses the actions of making files visible.                                                                        |
| 5. Run HS.SCRUB on a File/Table                           | Analyzes the data in a table or file, and lets you optionally correct anomalous data.                                                                                               |
| 6. Update File Information Cache in an account            | Updates this account's .hs_fileinfo file to reflect the current state of all files in the account.                                                                                  |

#### **UniVerse Server Administration Menu Selections**

---

## Making Files Visible to JDBC Applications

For JDBC applications, it is necessary to make files visible unless all of the data of interest is stored in UniVerse tables. This is because files that are not tables are not included in the TABLES rowset unless the JDBC application is connected to an account whose files have been made visible.

### Making Files Visible

Complete the following steps to make files in a UniVerse account visible to JDBC Applications:

1. On the UniVerse Server Administration menu, choose the third selection (Activate access to files in an account).
2. When prompted, enter either the full path of the UniVerse account (on Windows platforms, start with the drive letter, such as D:) or the account name as listed in the UV.ACCOUNT file.
3. Repeat steps 1 and 2 for each account whose files you want to make visible.
4. To exit the UniVerse Server Administration menu, press ENTER.

The process of making files visible does the following:

- Scans the dictionaries of all non-system files named in the VOC, finding all associations and unassociated multivalued columns.
- Writes an @EMPTY.NULL X-record in each file dictionary.
- Writes S or M in field 5 of A- and S-descriptors.
- Creates Q-pointers in the VOC for files comprising multiple data files.
- Creates an HS\_FILE\_A000HS\_FILE\_A000 file in the account.
- Creates a file information cache (.hs\_fileinfo) under the account's directory, which contains a compressed list of all 1NF file names in the account and is used by the JDBC Driver to rapidly construct the TABLES rowset whenever a Java application requests it to do so.
- Updates the UV.ACCOUNT file to indicate that the files in the account have been made visible.



## Restricting File Visibility

When you make an account's files visible to JDBC, the HS\_FILE\_A000 file is created in the account, as stated in the previous section. This file contains records for non-SQL files referenced by F- and Q-pointers in the account's VOC file. The following table shows the format and initial contents of the HS\_FILE\_A000 file.

| FILENAME (Record ID) | ACCESS (Field 1) |
|----------------------|------------------|
| HS_DEFAULT           | READ_WRITE       |
| &DEVICE&             | NONE             |
| .                    | .                |
| .                    | .                |
| .                    | .                |
| VOC                  | NONE             |
| VOCLIB               | NONE             |

**Initial Contents of HS\_FILE\_A000**

You can edit the HS\_FILE\_A000 file to control which files in the account are visible to JDBC applications.

The IDs of records in the HS\_FILE\_A000 file are the names of the files whose access you want to control. Each record has one field (ACCESS), which contains one of the following values:

- READ\_WRITE
- READ
- NONE

If the ACCESS field for a file is set to READ, this is treated the same as READ\_WRITE for JDBC.

The HS\_FILE\_A000 file contains a special record called HS\_DEFAULT which controls default access to all files in the account. When you first make files visible, the HS\_DEFAULT record is set to READ\_WRITE and the records for UniVerse system files (APP.PROGS, BASIC.HELP, ERRMSG, UV.ACCOUNT, DICT.DICT, NEWACC, and so forth) are set to NONE (no access).

To remove visibility from a few selected files and leave the rest visible to JDBC, add a record to HS\_FILE\_A000 for each restricted file, with the ACCESS field set to NONE.

To make just a few files in an account visible to JDBC, change the ACCESS field in the HS\_DEFAULT record from READ\_WRITE to NONE, and then add selected files whose ACCESS field is READ\_WRITE.

If you change the contents of the HS\_FILE\_A000 file, you must then update the file information cache for your changes to take effect. For information about updating the file information cache, see the next section.

## Updating the File Information Cache

After an account's files have been made visible, if the following types of change are made to the account's files, the changes are not automatically reflected in the TABLES rowset:

- Adding, changing, or deleting F- or Q-pointers in the VOC file.
- Creating or deleting UniVerse files.
- Adding, changing, or deleting associations or unassociated multivalued column definitions in file dictionaries.
- Restricting access to selected files by changing the contents of the HS\_FILE\_A000 file.

These types of change are not reflected in the TABLES rowset until the account's file information cache is updated.

You can update the file information cache using the UniVerse Server Administration menu or the HS.UPDATE.FILEINFO command.

Complete the following steps to update the file information cache using the menu:

1. On the UniVerse Server Administration menu, choose the sixth selection (Update File Information Cache in an account).
2. When prompted, enter either the full path of the UniVerse account (on Windows platforms, start with the drive letter, for example, D:) or enter the account name as listed in the UV.ACCOUNT file.
3. Repeat steps 1 and 2 for each account you want to update.
4. To exit the UniVerse Server Administration menu, press ENTER.

To update the file information cache using the HS.UPDATE.FILEINFO command:

1. Log to the desired UniVerse account.



2. At the system prompt, enter the command `HS.UPDATE.FILEINFO`.

The process of updating an account's file information cache executes the following tasks:

- Scans the dictionaries of all non-system files named in the VOC, finding all associations and unassociated multivalued columns.
- Rewrites the file information cache (`.hs_fileinfo`) under the account's directory, based on the above dictionary information and the contents of the `HS_FILE_A000` file.

**Warning:** The file information cache is for internal use only and should not be modified. Any changes to the cache can cause unpredictable behavior and can make UniVerse files in the account inaccessible to JDBC applications.

### *Example*

Suppose account MYACCOUNT contains the following files:

- STATES, which has an association called CITYZIPS.
- EMPS, which has multi-valued (unassociated) columns DEPENDENTS and PHONES.
- OCEANS, which has only singlevalued columns.

After the files in this account are made visible, the TABLES rowset lists the following tables.

| Schema Name  | Table Name      | Table Type |
|--------------|-----------------|------------|
| <null value> | STATES          | TABLE      |
| <null value> | STATES_CITYZIPS | TABLE      |
| <null value> | EMPS            | TABLE      |
| <null value> | EMPS_DEPENDENTS | TABLE      |
| <null value> | EMPS_PHONES     | TABLE      |
| <null value> | OCEANS          | TABLE      |

**TABLES Rowset for MYACCOUNT**

Suppose you now make the following changes in MYACCOUNT:

- Delete file STATES.
- Remove column DEPENDENTS from the dictionary of EMPS.
- Create a VOC entry called EMPLOYEES that refers to the same data file and file dictionary as does EMPS.
- Create a new file CONTINENTS that has only singlevalued columns.
- Add a record to HS\_FILE\_A000, whose key is OCEANS and whose ACCESS field is set to NONE.

These changes are not immediately reflected in the TABLES rowset, but after MYACCOUNT's file information cache is updated, the TABLES rowset lists the following information.

| Schema Name  | Table Name       | Table Type |
|--------------|------------------|------------|
| <null value> | EMPS             | TABLE      |
| <null value> | EMPS_PHONES      | TABLE      |
| <null value> | EMPLOYEES        | TABLE      |
| <null value> | EMPLOYEES_PHONES | TABLE      |
| <null value> | CONTINENTS       | TABLE      |

**Updated TABLES Rowset for MYACCOUNT**

## Removing File Visibility

Complete the following steps to make files in a UniVerse account invisible to JDBC:

1. On the UniVerse Server Administration menu, choose the fourth selection (Deactivate access to files in an account).
2. When prompted, enter either the full path of the UniVerse account (on Windows platforms, start with the drive letter, for example, D:) or the account name as listed in the UV.ACCOUNT file.
3. Repeat steps 1 and 2 for each account whose files you want to make invisible.
4. To exit the UniVerse Server Administration menu, press ENTER.

Making files invisible executes the following tasks:

- Deletes the HS\_FILE\_A000 file in the account.
- Deletes the file information cache (.hs\_fileinfo) under the account's directory.
- Updates the UV.ACCOUNT file to indicate that files in this account are not visible.

---

## Accessing Columns in UniVerse Tables and Files

As explained earlier, NF<sup>2</sup> data in UniVerse appears as 1NF tables to a JDBC application. You can access all columns defined in the dictionaries of these NF<sup>2</sup> files (including tables and views). In addition to stored-data columns, A JDBC application can access I-descriptors and correlatives (as read-only columns).

Some JDBC applications are written as general-purpose programs that determine a list of visible columns by requesting a COLUMNS rowset. The COLUMNS rowset for any visible 1NF table describes all columns returned by the following statement:

```
SELECT * FROM tablename;
```

Columns other than those returned in the COLUMNS rowset may be accessible by JDBC applications, since "SELECT \*" does not always return all columns defined in the dictionary.

### COLUMNS Rowset

A JDBC application can request a COLUMNS rowset, which provides a list of column characteristics. Each row in the COLUMNS rowset includes the following information:

- Schema name
- Table name
- Column name
- Column position
- Data type
- Precision
- Scale

For UniVerse tables, and all multivalued columns within them, columns listed in the COLUMNS rowset are defined by the dictionary's @SELECT phrase (if it exists), or are defined as the columns created by CREATE TABLE (and possibly modified by ALTER TABLE) if there is no @SELECT phrase.

For files, and for associations and unassociated multivalued columns within them, columns listed in the COLUMNS rowset are defined by the dictionary's @SELECT phrase (if it exists). If there is no @SELECT phrase, the contents of the COLUMNS rowset are defined by the dictionary's @ phrase (if it exists). If neither an @SELECT phrase nor an @ phrase exists, only the record ID (@ID) appears in the COLUMNS rowset.

For more information about the @SELECT and @ phrases, see *UniVerse Administration for DBAs*.

## Association Keys

UniVerse tables have primary keys. UniVerse files have record IDs. Primary keys and record IDs are unique identifiers for each row (record) of data in a table or file.

Because JDBC shows associations and unassociated multivalued columns as 1NF tables, they also require a set of unique identifiers that serve as primary keys. These keys are called association keys.

When you create a UniVerse table or file, you can define one or more association columns as the association key, but you do not have to. If you do not, UniVerse SQL generates a virtual column called @ASSOC\_ROW containing unique values that, combined with the primary keys or record IDs of the base table, become the association keys for the 1NF table generated from the association.

For detailed information about defining association keys, see *UniVerse SQL Administration for DBAs* and the *UniVerse SQL Reference*.

### Example

The file EMPS is defined with columns EMPNUM, NAME, DEPTNUM, DEPENDENTS, and PHONES, where DEPENDENTS and PHONES are unassociated multivalued columns. Suppose this file contains the following employees.

| EMPNUM | NAME     | DEPTNUM | DEPENDENTS | PHONES       |
|--------|----------|---------|------------|--------------|
| 4456   | GONZALES | 97      | SUSAN      | 555-876-4041 |
|        |          |         | FREDERICA  |              |
| 4901   | HURLBUT  | 58      |            | 555-245-1000 |
|        |          |         |            | 555-245-1456 |
| 6511   | RICHARDS | 58      | HELEN      | 400-765-4321 |
|        |          |         | ALFRED     | 400-765-9010 |
|        |          |         | SAMUEL     |              |

#### EMPS File

This table will appear to JDBC as three 1NF tables called EMPS, EMPS\_DEPENDENTS, and EMPS\_PHONES.

The EMPS 1NF table has column EMPNUM as its key.

| EMPNUM | NAME     | DEPTNUM |
|--------|----------|---------|
| 4456   | GONZALES | 97      |
| 4901   | HURLBUT  | 58      |
| 6511   | RICHARDS | 58      |

#### EMPS NF1 Table



The EMPS\_DEPENDENTS 1NF table has a key consisting of two columns: EMPNUM and @ASSOC\_ROW.

| EMPNUM | DEPENDENTS | @ASSOC_ROW |
|--------|------------|------------|
| 4456   | SUSAN      | 1          |
| 4456   | FREDERICA  | 2          |
| 6511   | HELEN      | 1          |
| 6511   | ALFRED     | 2          |
| 6511   | SAMUEL     | 3          |

**EMPS\_DEPENDENTS 1NF Table**

The EMPS\_PHONES 1NF table also has a two-column key consisting of EMPNUM and @ASSOC\_ROW.

| EMPNUM | PHONES       | @ASSOC_ROW |
|--------|--------------|------------|
| 4456   | 555-876-4041 | 1          |
| 4901   | 555-245-1000 | 1          |
| 4901   | 555-245-1456 | 2          |
| 6511   | 400-765-4321 | 1          |
| 6511   | 400-765-9010 | 2          |

**EMPS\_PHONES 1NF Table**

---

## Table and Column Names Containing Special Characters

Some UniVerse table and column names can contain special characters such as period (.) or at-sign (@). These names can cause difficulty with some JDBC applications unless they are enclosed in double quotation marks.

### Delimited Identifiers

UniVerse supports an ANSI-SQL feature called "delimited identifiers." This means that any identifier (table name, column name, index name, constraint name, and so forth) can be enclosed in double quotation marks to avoid ambiguity in the syntax of an SQL statement. This is particularly useful in the case of UniVerse table and column names that contain the period (.) character.

The following example shows a SELECT statement that contains column and table names delimited by double-quotation marks:

```
SELECT "MY.COLUMN" FROM SCHEMAX."MY.TABLE";
```

---

## Making UniVerse Data Meaningful to JDBC

This section describes tasks you may need to perform on the UniVerse server to make particular kinds of UniVerse data meaningful to JDBC applications:

- Define SQL data types for data in UniVerse files.
- Define the length of character string data in UniVerse files.
- Set up empty-null mapping.
- Validate and fix data in data files and dictionaries that cause SQL and JDBC problems.

### SQL Data Types

To fine-tune or define data type, precision, and scale values for columns and I-descriptors in files, you can edit the DATATYPE field of the corresponding dictionary entry (field 8 in D- and I-descriptors, field 6 in A- and S-descriptors). This is especially important for character data in which the display width defined in the dictionary may be much larger or smaller than the largest data values in the file. The HS.SCRUB utility can automatically make these adjustments based on the data found. For more information about HS.SCRUB, see [“Validating and Fixing Tables and Files”](#) on page 4-26.

If the DATATYPE field is empty, UniVerse determines a column's SQL data type by examining its conversion code and format specifications. The UniVerse server reports this SQL data type to JDBC applications in the COLUMNS rowset. If the UniVerse-generated SQL data type is inappropriate for the actual data in the column, you can specify the correct SQL data type in the DATATYPE field of the column's dictionary entry. For some data types, the SQL data type syntax is different between dictionary specifications and UniVerse SQL statements (for example, CREATE TABLE) as noted in the following table. Square brackets indicate optional parameters.

| Dictionary Syntax   | UniVerse SQL Syntax  | Notes                    |
|---------------------|----------------------|--------------------------|
| CHAR [ACTER] [,n]   | CHAR [ACTER] [(n)]   | n = number of characters |
| DEC [IMAL] [,p[,s]] | DEC [IMAL] [(p[,s])] | p = precision s = scale  |
| FLOAT [,p]          | FLOAT [(p)]          | p = precision            |
| NUMERIC [,p[,s]]    | NUMERIC [(p[,s])]    | p = precision s = scale  |
| VARCHAR [, n]       | VARCHAR [(n)]        | n = number of characters |

#### SQL Data Type Syntax

Note the syntactic differences regarding the use of parentheses and commas.

The DATE, DOUBLE PRECISION, INT [EGER], REAL, SMALLINT, and TIME data type syntax is identical for dictionaries and UniVerse SQL.

You can specify the SQL data type for any column, real or virtual, in a UniVerse file. You need not specify the SQL data type for any column of a table defined by the CREATE TABLE or CREATE VIEW statement, but you may want to specify the SQL data type for other columns in the table (such as I-descriptors) that are not defined in the SICA. You cannot modify the SQL data type for columns defined in the SICA, and UniVerse ignores the dictionary definitions for these columns. For more information about the SICA and UniVerse tables, see *UniVerse SQL Administration for DBAs* manual and the *UniVerse SQL User Guide*.

## Length of Character String Data

In JDBC, every character column has either a fixed length or a maximum length. This column length is called its precision and is reported in the COLUMNS rowset.

The precision of a character column is determined from one of the following:

- For a column in a UniVerse file, the precision is defined by the data type specified in the DATATYPE field of the column's dictionary definition. If the data type is not defined, the FORMAT field of the dictionary defines the precision.
- For a column defined by a CREATE TABLE or ALTER TABLE statement, the precision is defined by the column definition, which is stored in the table's SICA.
- For a column in a view, the precision is defined by the CREATE VIEW statement or by the precision of the column specified by the SELECT statement that creates the view.
- For a column added to a table's dictionary (such as an I-descriptor), the precision is defined by the data type specified in the DATATYPE field of the column's dictionary definition. If the data type is not defined, the FORMAT field of the dictionary defines the precision.

You can use the HS.SCRUB utility to examine a file's data and write appropriate data types in the DATATYPE field of the dictionary definitions for character-string columns. For information about HS.SCRUB, see [“Validating and Fixing Tables and Files”](#) on page 4-26.

The actual number of characters in a UniVerse character column can be greater than its precision. JDBC retrieves such extra characters, up to a limit. The number of bytes of character data that JDBC retrieves from a character column is the smallest of:

- The number of bytes of data the column actually contains.
- The number of bytes of data that UCI can fetch. This is controlled by the UCI configuration parameters MAXFETCHBUFF and MAXFETCHCOLS, defined in the uci.config file.

If your fetch buffer is not large enough to hold all the character data that the JDBC application retrieves, it generates a truncation warning.

The actual number of characters in a UniVerse character column can be less than its precision. Unlike some DBMSs, UniVerse does not automatically pad CHAR(*n*) columns on the right with spaces. If you insert the value "abc " (with two trailing spaces) into a CHAR(10) column, the column contains only five characters, not 10.

Your application and data source must agree on a consistent way to treat trailing spaces in a CHAR(*n*) column. Generally, it is better to treat CHAR(*n*) columns as if they were VARCHAR columns with no space padding.

## Empty-Null Mapping

UniVerse files use empty strings in much the same way tables use null values. Unfortunately, empty strings in numeric or date columns cause data conversion errors in JDBC, making these columns almost inaccessible to some JDBC applications. To make files with empty values accessible to JDBC applications, the UniVerse server provides empty-null mapping, converting empty values in UniVerse files to null values in JDBC application buffers, and vice versa.

To activate empty-null mapping for a UniVerse table or file, add an X-descriptor named @EMPTY.NULL to the dictionary. To turn off empty-null mapping, delete the @EMPTY.NULL entry from the dictionary.

The third selection (Activate access to files in an account) on the UniVerse Server Administration menu enables empty-null mapping in all files in the account by creating @EMPTY.NULL dictionary entries.

## Validating and Fixing Tables and Files

Use the HS.SCRUB utility to scan data in a table or file and fix data and dictionary problems that can cause SQL and JDBC difficulties. The HS.SCRUB utility performs the following tasks:

- Reports data anomalies.
- Saves a select list of record IDs of problem records.
- Writes appropriate data types in the DATATYPE field of the dictionary's column definitions.
- Adjusts dictionary entries to accommodate bad data, such as nonnumeric values in numeric, date, and time columns, which can lead to adjusting the column type to CHARACTER.
- Adds an @EMPTY.NULL record to the dictionary.
- Adds an @SELECT record to the dictionary.
- Fixes the data in the file, optionally saving a copy of the original file.

### *The @EMPTY.NULL Record*

HS.SCRUB adds an @EMPTY.NULL record to the table or file dictionary if all the following conditions are met:

- The dictionary does not already include an @EMPTY.NULL record.
- The data contains empty values.
- Modification of the dictionary is enabled (FIX, AUTOFIX, AUTOFIX DICT).

For information about empty-null mapping, see [“Empty-Null Mapping”](#) on page 4-26.

### ***The @SELECT Record***

HS.SCRUB adds an @SELECT record to the file dictionary (but not to a table dictionary) if both of the following conditions are met:

- The dictionary does not already include an @ or @SELECT record.
- Modification of the dictionary is enabled (FIX, AUTOFIX, AUTOFIX DICT).

For information about @SELECT records, see *UniVerse SQL Administration for DBAs*.

### ***Running the HS.SCRUB Utility***

To run the HS.SCRUB utility, choose the fifth selection (Run HS.SCRUB on a File/Table) on the UniVerse Server Administration menu, or use the HS.SCRUB command described in the next section.

If you use the HS.SCRUB utility on a File/Table selection, the system prompts you to enter either the full path of the UniVerse account (for Windows platforms, start with the drive letter, such as D:) or the account name as listed in the UV.ACCOUNT file. Press ENTER to see a list of UniVerse accounts whose files have been made visible to JDBC.

Next the system prompts you to enter the name of the table or file you want to analyze or change. After you enter a file name, the system prompts you to enter the mode of operation. Enter one of the following at the Mode prompt:

- To generate a report without modifying the table or file, press ENTER
- FIX
- AUTOFIX
- AUTOFIX DICT

## ■ AUTOFIX DATA

The next subsection describes these modes.

### *Using the HS.SCRUB Command*

To use the HS.SCRUB command, log to the UniVerse account containing the table or file you want to analyze.

The syntax of the HS.SCRUB command is as follows:

HS.SCRUB *file name* [FIX | AUTOFIX [DICT | DATA]]

where *file name* is the name of the file or table to analyze.

The following table describes each parameter of the syntax.

| Parameter | Description                                                                                                                                                                                                                                                             |
|-----------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| FIX       | Puts HS.SCRUB in interactive mode, in which the system prompts you to resolve any anomalies following the analysis.                                                                                                                                                     |
| AUTOFIX   | Puts HS.SCRUB in automatic mode; anomalies are corrected with the default action that would have been presented to the user. If you do not specify DICT or DATA with the AUTOFIX option, HS.SCRUB resolves anomalies in both the data file or table and its dictionary. |
| DICT      | Indicates that HS.SCRUB resolves only those anomalies associated with the file's or table's dictionary.                                                                                                                                                                 |
| DATA      | Indicates that HS.SCRUB resolves only those anomalies associated with the file's or table's data.                                                                                                                                                                       |

#### **HS.SCRUB Parameters**

When neither FIX nor AUTOFIX is specified, HS.SCRUB only reports anomalies. No data or dictionary items are modified.



## Scalar Functions

The following table lists UniVerse JDBC support of scalar functions.

|         | Function            | Supported |
|---------|---------------------|-----------|
| String  | ASCII               | No        |
|         | CHAR                | No        |
|         | CONCAT              | Yes       |
|         | DIFFERENCE          | No        |
|         | INSERT <sup>a</sup> | Yes       |
|         | LCASE               | Yes       |
|         | LEFT                | Yes       |
|         | LENGTH              | Yes       |
|         | LOCATE              | No        |
|         | LTRIM               | Yes       |
|         | REPEAT              | No        |
|         | REPLACE             | No        |
|         | RIGHT <sup>1</sup>  | Yes       |
|         | RTRIM               | Yes       |
|         | SOUNDEX             | No        |
|         | SPACE               | No        |
|         | SUBSTRING           | Yes       |
|         | UCASE               | Yes       |
| Numeric | ABS                 | No        |
|         | ACOS                | No        |

**UniVerse JDBC Support of Scalar Functions**

|               | <b>Function</b> | <b>Supported</b> |
|---------------|-----------------|------------------|
|               | ASIN            | No               |
|               | ATAN            | No               |
|               | ATAN2           | No               |
|               | CEILING         | No               |
|               | COS             | No               |
|               | COT             | No               |
|               | DEGREES         | Yes              |
|               | EXP             | No               |
|               | FLOOR           | No               |
|               | LOG             | No               |
|               | LOG10           | No               |
|               | MOD             | No               |
|               | PI              | Yes              |
|               | POWER           | No               |
|               | RADIANS         | Yes              |
|               | RAND            | No               |
|               | ROUND           | No               |
|               | SIGN            | No               |
|               | SIN             | No               |
|               | SQRT            | No               |
|               | TAN             | No               |
|               | TRUNCATE        | No               |
| Time and Date | CURDATE         | No               |
|               | CURTIME         | No               |

**UniVerse JDBC Support of Scalar Functions (Continued)**

|            | <b>Function</b>      | <b>Supported</b> |
|------------|----------------------|------------------|
|            | DAYNAME              | No               |
|            | DAYOFMONTH           | No               |
|            | DAYOFWEEK            | No               |
|            | DAYOFYEAR            | No               |
|            | HOUR                 | No               |
|            | MINUTE               | No               |
|            | MONTH                | No               |
|            | MONTHNAME            | No               |
|            | NOW                  | No               |
|            | QUARTER              | No               |
|            | SECOND               | No               |
|            | TIMESTAMPADD         | No               |
|            | TIMESTAMPDIFF        | No               |
|            | WEEK                 | No               |
|            | YEAR                 | No               |
| System     | DATABASE             | Yes              |
|            | IFNULL               | No               |
|            | USER                 | Yes              |
| Conversion | CONVERT <sup>b</sup> | Yes              |

---

**UniVerse JDBC Support of Scalar Functions (Continued)**

---

- a. INSERT and RIGHT may evaluate their arguments more than once, consequently you cannot use a parameter marker as an argument to these functions, and you must watch out for side effects if you use an I-descriptor.
- b. UniVerse JDBC supports the same conversions that UniVerse SQL CAST supports.

An example of a call to a scalar function is as follows:

```
SELECT * FROM UV_SCHEMA  
WHERE SCHEMA_NAME = {fn DATABASE()};
```

JDBC implies that no scalar functions can take an arithmetic expression or a parameter marker as an argument. UniVerse JDBC tolerates violations of this rule, but in the future it may optionally trap them and generate an error message.

---

# Programming with JDBC

|                                                                |      |
|----------------------------------------------------------------|------|
| Loading the IBM JDBC Driver for UniData and UniVerse . . . . . | 5-3  |
| Creating a Connection . . . . .                                | 5-4  |
| Format of Database URLs . . . . .                              | 5-5  |
| Accessing Database MetaData . . . . .                          | 5-7  |
| Querying the Database . . . . .                                | 5-8  |
| Mapping Data Types . . . . .                                   | 5-9  |
| Data Conversion . . . . .                                      | 5-10 |
| Handling Errors . . . . .                                      | 5-11 |
| Handling Transactions . . . . .                                | 5-12 |
| Isolation Levels . . . . .                                     | 5-13 |
| JDBC 2.0 Optional Package Support . . . . .                    | 5-14 |
| Tuning the Connection Pool . . . . .                           | 5-18 |
| Restrictions and Limitations . . . . .                         | 5-23 |
| Unsupported Data Types . . . . .                               | 5-23 |
| Unsupported Methods . . . . .                                  | 5-23 |

This chapter explains the IBM-specific information you need to use the IBM JDBC Driver for UniData and UniVerse to connect to your database.

---

## Loading the IBM JDBC Driver for UniData and UniVerse

To load the IBM JDBC Driver for UniData and UniVerse, use the `Class.forName()` method, passing it the value `com.ibm.u2.jdbc.UniJDBCDriver`, as shown in the following example:

```
try {  
    Class.forName("com.ibm.u2.jdbc.UniJDBCDriver");  
} catch (Exception e) {  
}
```

The `Class.forName()` method loads the IBM implementation of the Driver class, `UniJDBCDriver`. The `UniJDBCDriver` class then creates an instance of the driver and registers it with the `DriverManager` class.

Once you load the IBM JDBC Driver for UniData and UniVerse, you are ready to connect to the database.

---

## Creating a Connection

To create a connection to a UniData or UniVerse database, use the `DriverManager.getConnection()` method. This method creates a `Connection` object, which is later used to create and send SQL statements to the database, and then process the results.

The `DriverManager` class keeps track of the available drivers and handles connection requests between appropriate drivers and databases. The `url` parameter of the `getConnection()` method is a database URL that specifies the host name, port number and account path. These are the required properties defined in the table below. An optional second parameter to the `getConnection()` method, *property*, is the property list defined in the following table.

The full syntax of the connection string is defined as:

```
jdbc:ibm-u2://<host>[:<port>]/<account>[[;name=value]...]
```

The *required* properties for the connection string are listed in the following table.

---

| Database<br>URL<br>Variable | Description                                                                                 |
|-----------------------------|---------------------------------------------------------------------------------------------|
| <b>host</b>                 | The host machine where the database server is running.                                      |
| <b>port</b>                 | The port number of the UniRPC server running on the host. The default port number is 31438. |
| <b>account</b>              | The full path to the account.                                                               |

---



The *optional* connection properties for the connection string are listed in the following table.

| Database URL Variable | Description                                                                                                      |
|-----------------------|------------------------------------------------------------------------------------------------------------------|
| <b>user</b>           | User logon name for host system.                                                                                 |
| <b>password</b>       | Password for logon.                                                                                              |
| <b>dbmstype</b>       | The database type to which you will be connecting. Must be UniData or UniVerse. The default setting is UniVerse. |
| <b>network</b>        | Specifies the network used to access the sql server. The default value is TCP/IP.                                |
| <b>proxyhost</b>      | The host name for the proxy server being used.                                                                   |
| <b>proxyport</b>      | The port number of the proxy server. The default value is 31448.                                                 |

## Format of Database URLs

There are two ways to connect to a SQL server through the IBM JDBC Driver for UniData and UniVerse. The first method is defined by the JDBC 1.0 specification, and uses the URL connection string.

The following example shows a database URL that connects to the UniData demo account, demo:

```
try {
    Class.forName("com.ibm.u2.jdbc.UniJDBCdriver");
    String sUrl = "jdbc:ibm-
u2://localhost/d:/ibm/ud60/demo;dbmstype=UNIDATA";
    Connection con = DriverManager.getConnection(sUrl, "user",
"pass");
}
catch (Exception e) {

}
```

Another method of creating a JDBC connection is through the use of a DataSource object and is defined by the JDBC 2.0 Optional Package API. The two DataSource interfaces are **com.ibm.u2.jdbcx.UniJDBCDataSource** and **com.ibm.u2.jdbcx.UniJDBCPoolConnectionDataSource**. For more information on these types of JDBC connections, see [“JDBC 2.0 Optional Package Support”](#) on page 5-14.

---

## Accessing Database MetaData

To access information about a UniData or UniVerse database, use the JDBC API **DatabaseMetaData** interface. The IBM JDBC Driver for UniData and UniVerse supports most of the **DatabaseMetaData** methods.

For some methods returning a **ResultSet** object, for example **DatabaseMetaData.getTables()**, UniData and UniVerse handle system tables differently, and thus will return different results.

In UniVerse, the `getSchemas()` method returns the schema names available in the database information and which accounts are accessible by JDBC. In UniData, the `getSchemas()` method returns the owner of every table in the account to which you are connected.

The `getTables()` method contains SQL tables and database files which are accessible by JDBC.

The `getColumns()` method contains columns of SQL tables and files which are accessible by JDBC.

**Note:** When calling *DatabaseMetaData* methods on UniData Files, null should be passed as the parameter whenever a schema name is requested.

When calling *DatabaseMetaData* methods on UniVerse Files, null should be passed as the parameter whenever a schema name is requested. For UniVerse Tables, where schemas are supported, the name of the schema should be used.

See “[Restrictions and Limitations](#)” on page 5-22 for a list of methods that are not supported by the IBM JDBC Driver for UniData and UniVerse and should not be used in your Java program.



---

## Querying the Database

The IBM JDBC Driver for UniData and UniVerse complies with the JDBC API specification for sending queries to a database and retrieving the results. The driver supports most of the methods of the **Statement**, **PreparedStatement**, **CallableStatement**, **ResultSet**, and **ResultSetMetaData** interfaces.

The **Statement** interface allows most of the eligible SQL statements to be executed, according to the syntax of SQL statements supported by UniData and UniVerse SQL servers. See [“Restrictions and Limitations”](#) on page 5-22 for a list of the unsupported **Statement** methods

The **PreparedStatement** interface allows users to prepare a statement once and subsequently execute it multiple times. The interface can take parameters presented by '?', and must be bound with specific values before execution.

The **CallableStatement** interface is implemented by a UniData or UniVerse BASIC program. Each callable statement can have multiple parameters with type **INPUT**, **OUTPUT**, and **INPUT\_OUTPUT**. The return value can be carried in the **OUTPUT** and **INPUT\_OUTPUT** parameters. All of the parameters are of VARCHAR data type.

The **ResultSetMetaData** interface will return each column's definition. For a map of supported data types between UniData, UniVerse, and JDBC, see [“Mapping Data Types”](#) on page 5-9.

---

## Mapping Data Types

This section discusses mapping issues between data types defined in a Java program, and the data types supported by the UniData and UniVerse database servers. In particular, it covers the following two topics:

- Mapping between JDBC API data types and UniData and UniVerse data types.
- `ResultSet.getXXX()` methods supported by the IBM JDBC Driver for UniData and UniVerse.

The following table shows the UniData and UniVerse data types to which each JDBC API data type maps. See [“Restrictions and Limitations”](#) on page 5-22 for a list of unsupported data types.

| JDBC API Data Type | UniData Data Type | UniVerse Data Type |
|--------------------|-------------------|--------------------|
| BIGINT             | INTEGER           | INTEGER            |
| BIT                | BIT               | BIT                |
| CHAR               | CHAR              | CHAR               |
| DATE               | DATE              | DATE               |
| DECIMAL            | DECIMAL           | DECIMAL            |
| DOUBLE             | DOUBLE PRECISION  | DOUBLE PRECISION   |
| FLOAT              | FLOAT             | FLOAT              |
| LONGVARCHAR        | VARCHAR           | VARCHAR            |
| NUMERIC            | INTEGER           | NUMERIC            |
| REAL               | REAL              | REAL               |
| SMALLINT           | SMALLINT          | INTEGER            |
| TIME               | TIME              | TIME               |
| TINYINT            | INTEGER           | INTEGER            |
| VARCHAR            | VARCHAR           | VARCHAR            |

## Data Conversion

Since UniData and UniVerse store all of their data types as strings, their respective SQL servers will translate any column data that does not match to their data types to NULL. Using a UniData or UniVerse data type that maps to a JDBC data type, such as **BIT**, will work as long as the data stays within the range of the SQL server's data range.

### *Date and Time Data*

When retrieving date and time data with `ResultSet.getDate()` and `ResultSet.getTime()`, the data being retrieved should be stored in the database in internal date or time format. If the data is instead stored as a date or time string, it is safest to retrieve the data with `ResultSet.getString()`, and explicitly convert it to a `Date` or `Time` object as needed.

---

## Handling Errors

IBM JDBC Driver for UniData and UniVerse error and warning messages (exceptions) can be generated from both the server and driver (client) sides.

All of the server side errors and warning messages contain **SQLSTATE** code and original message text, whereas the driver side errors and warning messages may contain vendor code, **SQLSTATE** code, and original message text.

***Note:** Run-time errors occurring in UniBasic and UVBasic subroutines are not automatically propagated to the JDBC application. You can define an out parameter for the subroutine to notify the JDBC application of UniBasic or UVBasic errors. The JDBC application can then check the status of this parameter after calling the subroutine.*



---

## Handling Transactions

When you create a new `Connection` object, `autocommit` is set to `true` by default. This means that when a statement is sent to the server, a `COMMIT` statement is automatically executed. The `autocommit` mode can be set to `true` or `false` by calling the `Connection.setAutoCommit()` method. The syntax is as follows:

```
Connection.setAutoCommit(true)
Connection.setAutoCommit(false)
```

If `autocommit` is set to `false`, the IBM JDBC Driver for UniData and UniVerse starts a new transaction as soon as the next statement is sent to the database server. This transaction lasts until a `COMMIT` or `ROLLBACK` statement is issued by the user. If the user has already started a transaction by executing `setAutoCommit(false)`, and then calls `setAutoCommit(false)` again, the existing transaction continues unchanged. In order for the connection to the database or database server to be dropped, the Java program must explicitly terminate the transaction by issuing either a `COMMIT` or a `ROLLBACK` statement.

While inside a transaction, if the Java program sets `autocommit` mode on, the IBM JDBC Driver for UniData and UniVerse will roll back the current transaction before it actually turns `autocommit` mode on.

If a `COMMIT` statement is sent to a database that has been created with logging, and `autocommit` mode is on, the database server returns the error `-255 : not in transaction`. This is true whether the statement was sent with the `Connection.commit()` method, or directly with an SQL statement. This is true because there is currently no user transaction started.

When you explicitly send a `COMMIT` statement to the database server in a database created in ANSI mode, the statement commits an empty transaction. The system does not return an error since the database server automatically starts a transaction before it executes the statement. This is true if there is no user transaction currently open.



## Isolation Levels

UniData and Universe support the following isolation levels:

- TRANSACTION\_READ\_UNCOMMITTED
- TRANSACTION\_READ\_COMMITTED
- TRANSACTION\_REPEATABLE\_READ
- TRANSACTION\_SERIALIZABLE

When changing the isolation level in manual transaction mode, the new isolation level does not take effect until the current transaction has been committed or rolled back.

---

## JDBC 2.0 Optional Package Support

The IBM JDBC Driver for UniData and UniVerse supports the following interfaces defined in the JDBC Extended Optional Package:

```
javax.sql.DataSource  
javax.sql.ConnectionPoolDataSource
```

Their corresponding implementation class names are:

```
com.ibm.u2.jdbcx.UniJDBCDataSource  
com.ibm.u2.jdbcx.UniJDBCConnectionPoolDataSource
```

### *Connecting Through a DataSource Object*

Connecting through a DataSource object is the preferred means of establishing a connection to a data source. For information about how and why to use a DataSource object, see the JDBC 2.0 Standard Extension Specification provided by Sun Microsystems, available from the following Web site:

```
http://java.sun.com/products/jdk/1.2
```

In order to create a connection with the DataSource object, you must have the appropriate JNDI (Java Naming and Directory Interface) installed and configured for your application. For more information on JNDI, go to:

```
http://java.sun.com/products/jndi
```

In the following steps for creating a DataSource object:

- The variable ds refers to a DataSource object.
- The logical name for the DataSource object is myDS.

## ***Deploying a DataSource Object***

Complete the following steps to deploy a DataSource object:

1. Instantiate a UniJDBCDataSource object.
2. Configure the parameters appropriately:  
  
`ds.setServerHost("myServer");  
ds.setAccount("myAccount");`
3. Now you must register the DataSource object by using the JNDI to map a logical name to the DataSource object.

```
Context ctx = new InitialContext();  
ctx.bind("myDS", ds);
```

Now you can use the DataSource Object as shown in the following example:

```
Connection con = null;  
  
try {  
  
    Context ctx = new InitialContext();  
  
    DataSource ds = (DataSource)ctx.lookup("myDS");  
  
    con = ds.getConnection("user", "pass");  
  
    // use the connection  
  
} catch (Exception e) {  
  
    // exception-handling code  
  
} finally {  
  
    if (con != null) con.close();  
  
}
```

## ***Connecting Through a Pooled Connection DataSource Object***

Certain JDBC applications may demand higher performance and scalability. This can be achieved by obtaining your connection to the database server through a DataSource object that references a ConnectionPoolDataSource object. By using this interface, the IBM JDBC Driver for UniData and UniVerse keeps a connection in a pool to be reused, rather than returning it to the database server to be terminated. Whenever a connection is requested, the driver obtains the connection from the pool, avoiding the overhead of having the server close and reopen a connection.

In situations where the application services frequent connection requests, using the `ConnectionPoolDataSource` can significantly improve performance.

For more information on implementing the `DataSource` or `ConnectionPoolDataSource` objects, see the JDBC 2.0 Standard Extension Specification provided by Sun Microsystems, available at:

<http://java.sun.com/products/jdk/1.2/docs/guide/jdbc/index.html>

In the following steps for creating a `ConnectionPoolDataSource` object:

- The variable `cpds` refers to a `ConnectionPoolDataSource` object.
- The JNDI logical name for the `ConnectionPoolDataSource` object is `myCPDS`.
- The variable `ds` refers to a `DataSource` object.
- The logical name for the `DataSource` object is `DS_Pool`



**Note:** When a UniVerse JDBC connection is closed by an application, there is a slight delay in releasing the license associated with that connection. As a result, brief intervals may occur where there are more "consumed licenses" than there are active connections. On rare occasions, if the UniVerse system is running near the number of total available licenses, this may cause a connection attempt to be refused, if a previously consumed license has not yet been released.

## *Deploying a ConnectionPoolDataSource Object*

Complete the following steps to deploy a ConnectionPoolDataSource object:

1. Instantiate a UniJDBCConnectionPoolDataSource object.
2. Configure the following parameters appropriately:

```
cpds.setServerHost("myServer");
cpds.setAccount("myAccount");
cpds.setUser("myUsername");
cpds.setPassword("myPassword");
cpds.setUniJDBCCPMMaxConnections(-1);
cpds.setUniJDBCCPMInitPoolsize(0);
cpds.setUniJDBCCPMMinPoolsize(2);
cpds.setUniJDBCCPMMaxPoolsize(8);
cpds.setUniJDBCCPMServiceInterval(100);
cpds.setUniJDBCCPMagelimit(7200);
cpds.setUniJDBCCPMMinAgelimit(3600);
```

**Note:** *The username and Password parameters must be set in order for the InitPoolsize parameter to take affect.*

3. Now you must register the ConnectionPoolDataSource object by using the JNDI to map a logical name to the ConnectionPoolDataSource object.

```
Context ctx = new InitialContext();
ctx.bind("myCPDS"cpds,);
```

4. Now instantiate a UniJDBCDataSource object.
5. You must associate this DataSource object with the logical name you registered for the ConnectionPoolDataSource object:

```
ds.setDataSourceName("myCPDS");
```

6. Register the DataSource object using the JNDI interface:

```
Context ctx = new InitialContext();
ctx.bind("DS_Pool",ds);
```



Now you can use the `DataSource` object, as shown in the following example:

```
Connection con = null;

try {
    Context ctx = new InitialContext();
    DataSource ds = (DataSource)ctx.lookup("DS_pool");
    con = ds.getConnection("user", "pass");

    // use the connection
} catch (Exception e) {
    // exception-handling code
} finally {
    if (con != null) con.close();
}
```

Once a pooled `DataSource` is properly registered, as demonstrated in the previous step-by-step example, the syntax for using it is the same as for a nonpooled `datasource`. It is however, important to explicitly close the connection in a final block to ensure that the connection is returned to the pool after use.

## Tuning the Connection Pool

There are eight properties that you can change to adjust how Connection Pooling works in your application. The application programmer or an administrator can change these values in the deployment phase. These properties are defined in the following table.

The following table lists the available parameters for the **ConnectionPoolDataSource** object.

| Parameter               | Description                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 | Default |
|-------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------|
| IBM_CPM_INIT_POOLSIZE   | Allows you to specify the initial number of connections to be allocated for the pool when the <b>ConnectionPoolDataSource</b> object is first instantiated and the pool is initialized. This parameter should be set if your application requires numerous connections when the <b>ConnectionPoolDataSource</b> object is first instantiated. To obtain the value, call <code>getUniJDBCCPMInitPoolSize()</code> .<br><br>To set the value, call <code>setUniJDBCCPMInitPoolSize()</code> . | 0       |
| IBM_CPM_MAX_CONNECTIONS | This parameter enables you to specify the maximum number of physical connections that the DataSource object can obtain from the server, excluding those returned to the server. A setting of -1 allows for an unlimited number of connections. To obtain the value, call <code>getUniJDBCCPMMaxConnections()</code> . To set the value, call <code>setUniJDBCCPMMaxConnections(int limit)</code> .                                                                                          | -1      |
| IBM_CPM_MINPOOLSIZE     | This parameter enables you to specify the minimum number of connections to maintain in the pool. See the <code>UniJDBC_CPM_MIN_AGE_LIMIT</code> parameter for information about what to do when this minimum number of connections kept in the pool has exceeded the age limit. To obtain the value, call <code>getUniJDBCCPMMinPoolSize()</code> . To set the value, call <code>setUniJDBCCPMMinPoolSize(int min)</code> .                                                                 | 0       |

| Parameter            | Description                                                                                                                                                                                                                                                                                                                                         | Default |
|----------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------|
| IBM_CPM_MAX_POOLSIZE | This parameter enables you to specify the maximum number of connections to maintain in the pool. When the pool reaches this size, all connections are returned to the server. To obtain the value, call <code>getUniJDBCCPMMaxPoolSize()</code> . To set the value, call <code>setUniJDBCCPMMaxPoolSize(int max)</code> .                           | None    |
| IBM_CPM_AGE LIMIT    | This parameter enables you to specify the time, in seconds, that a free connection is kept in the free connection pool. A setting of -1 allows the connections to be retained until the client terminates. To obtain the value, call <code>getUniJDBCCPMAgeLimit()</code> . To set the value, call <code>setUniJDBCCPMAgeLimit(long limit)</code> . | -1      |



| Parameter                | Description                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     | Default |
|--------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------|
| IBM_CPM_MIN_AGE LIMIT    | This parameter enables you to specify the additional time, in seconds, that a connection in the free connection pool will be retained when no connection requests have been received. A setting of -1 allows a minimum of connections to be retained until the client terminates. To obtain the value, call <code>getUniJDBCCPMMinAgeLimit()</code> . To set the value, call <code>setUniJDBC-CPMAgeMinLimit(long limit)</code> .                                                                                                                                               | -1      |
| IBM_CPM_SERVICE_INTERVAL | This parameter enables you to specify the pool service frequency, in milliseconds. Pool service activity includes adding free connections if the number of free connections falls below the minimum value and removing free connections. To obtain the value, call <code>getUniJDBCCPMServiceInterval()</code> . To set the value, call <code>getUniJDBC-CPMServiceInterval (long interval)</code> .                                                                                                                                                                            | 50      |
| IBM_CPM_PC_REFRESH_MODE  | <p>This parameter enables you to specify the connection refreshing mode when a pooled connection is returned to the pool. There are 3 settings for this parameter.</p> <p>1 - The connection will be reused. With this setting, the statements on the connection will be closed, and transaction mode will be switched back to "auto". However, be aware that other properties are inherited, such as BASIC "COMMON" variables.</p> <p>3 - The connection will be closed and reestablished.</p> <p>11 - The connection will be closed and reestablished by a helper thread.</p> | 11      |

---

## Restrictions and Limitations

### Unsupported Data Types

The IBM JDBC Driver for UniData and UniVerse does not support the following data types:

- `java.sql.Types.ARRAY`
- `java.sql.Types.BINARY`
- `java.sql.Types.BLOB`
- `java.sql.Types.CLOB`
- `java.sql.Types.DISTINCT`
- `java.sql.Types.JAVA_OBJECT`
- `java.sql.Types.LONGVARBINARY`
- `java.sql.Types.NULL`
- `java.sql.Types.REF`
- `java.sql.Types.STRUCT`
- `java.sql.Types.TIMESTAMP`
- `java.sql.Types.VARBINARY`

### Unsupported Methods

The IBM JDBC Driver for UniData and UniVerse does not support the following JDBC functionality:

The following methods of the **Statement** interface are not supported.

- `Statement.addBatch(string)`
- `Statement.cancel()`
- `Statement.clearBatch()`
- `Statement.executeBatch()`

The following methods of the **PreparedStatement** interface are not supported.

- `PreparedStatement.addBatch()`

- `PreparedStatement.setArray(int, java.sql.Array)`
- `PreparedStatement.setBlob(int, java.sql.Blob)`
- `PreparedStatement.setCharacterStream(int, java.io.Reader)`
- `PreparedStatement.setClob(int, java.sql.Clob)`
- `PreparedStatement.setRef(int, java.sql.Ref)`

The following methods of the **Connection** interface are not supported.

- `Connection.getCatalog`
- `Connection.setCatalog`
- `Connection.getTypeMap`
- `Connection.setTypeMap`

The following methods of the **DatabaseMetaData** interface are not supported.

- `DatabaseMetaData.getProcedures(String catalog, String schemaPattern, String procedureNamePattern)`
- `DatabaseMetaData.getProcedureColumns(String catalog, String schemaPattern, String procedureNamePattern, String columnNamePattern)`
- `DatabaseMetaData.getCatalogs()`
- `DatabaseMetaData.getColumnPrivileges(String catalog, String schema, String table, String columnNamePattern)`
- `DatabaseMetaData.getTablePrivileges(String catalog, String schemaPattern, String tableNamePattern)`
- `DatabaseMetaData.getBestRowIdentifier(String catalog, String schema, String table, int scope, boolean nullable)`
- `DatabaseMetaData.getVersionColumns(String catalog, String schema, String table)`
- `DatabaseMetaData.getImportedKeys(String catalog, String schema, String table)`
- `DatabaseMetaData.getExportedKeys(String catalog, String schema, String table)`
- `DatabaseMetaData.getCrossReference(String primaryCatalog, String primarySchema, String primaryTable, String foreignCatalog, String foreignSchema, String foreignTable)`

- DatabaseMetaData.getIndexInfo(String catalog, String schema, String table, boolean unique, boolean approximate)
- DatabaseMetaData.getUDTs(String catalog, String schemaPattern, String typeNamePattern, int[] types)

The following methods of the **ResultSet** interface are not supported.

- ResultSet.getCharacterStream(int)
- ResultSet.getCharacterStream(String)

The following method of the **ResultSetMetaData** interface is not supported.

- ResultSetMetadata.getColumnClassName(int)

---

# JDBC Scrollable Cursors

|                                        |     |
|----------------------------------------|-----|
| JDBC Scrollable Cursors . . . . .      | 6-2 |
| Creating a Scrollable Cursor. . . . .  | 6-2 |
| Methods for Moving the Cursor. . . . . | 6-3 |
| Connection URL Parameters . . . . .    | 6-4 |
| Example . . . . .                      | 6-5 |

---

## JDBC Scrollable Cursors

When you execute an SQL statement on the server that has the potential of returning multiple rows, these rows are presented to the Java application one row at a time in an object called *ResultSet*. Prior to this release, UniVerse and UniData could only view these rows in a forward, sequential manner, from the first row to the last row.

At this release, scrollable cursors are introduced in UniJDBC. Scrollable cursors allow the program to move forward or backward, or jump directly to a specific row. UniVerse and UniData do not allow updates to the *ResultSet*. The *ResultSet* is a snapshot taken at the time the statement was executed.

### Creating a Scrollable Cursor

The following three UniJDBC statements can return a result set:

```
Statement stmt =  
con.createStatement(resultSetType,resultSetConcurrency)  
  
PreparedStatement pstmt =  
con.prepareStatement(sql,resultSetType,resultSetConcurrency)  
  
CallableStatement cstmt =  
con.prepareCall(sql,resultSetType,resultSetConcurrency)
```

#### *resultSetType*

The *resultSetType* parameter can be one of the types described in the following table.

| resultSetType           | Description                                                             |
|-------------------------|-------------------------------------------------------------------------|
| TYPE_FORWARD_ONLY       | The cursor can only move forward in the result set.                     |
| TYPE_SCROLL_INSENSITIVE | The resultSet is scrollable, but insensitive to changes made by others. |

resultSetType

#### *resultSetConcurrency*

The only valid valid for the *resultSetConcurrency* parameter is CONCUR\_READ\_ONLY, for a resultSet object that can not be updated.

## Methods for Moving the Cursor

You can call one of the methods described in the following table to move the cursor.

| Method                               | Description                                                                                                                 |
|--------------------------------------|-----------------------------------------------------------------------------------------------------------------------------|
| <code>srs.afterLast()</code>         | Move the cursor to the position after the last row, so the following <code>previous()</code> method points to the last row. |
| <code>srs.beforeFirst()</code>       | Moves the cursor to the position before the first row.                                                                      |
| <code>srs.next()</code>              | Moves the cursor to the next row.                                                                                           |
| <code>srs.previous()</code>          | Moves the cursor to the previous row.                                                                                       |
| <code>srs.absolute(<i>n</i>)</code>  | Moves the cursor to the <i>n</i> row.                                                                                       |
| <code>srs.absolute(-<i>n</i>)</code> | Moves the cursor back <i>n</i> rows from the end of the result set.                                                         |
| <code>srs.relative(<i>n</i>)</code>  | Moves the cursor forward <i>n</i> rows from the current cursor position.                                                    |
| <code>srs.relative(-<i>n</i>)</code> | Moves the cursor back <i>n</i> rows from the current cursor position.                                                       |

### Methods for Moving the Cursor

### *Determining the Cursor Position*

Use the following method to determine the cursor position:

```
rowNum = srs.getRow()
```

---

## Connection URL Parameters

The following table describes the connection URL parameters for use with scrollable cursors.

| Parameter                                   | Description                                                                                                                                                                                 |
|---------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| MAX_SCROLLABLE_CACHE_CURSOR_ROWS            | The maximum number of rows the scrollable cache can contain. If you specify -1, there can be unlimited rows.                                                                                |
| MAX_SCROLLABLE_CACHE_CURSOR_SPACE_IN_MEMORY | In kilobytes, the maximum memory space one scrollable cache can consume. If you specify 0, all rows must be stored on disk. If you specify -1, the space is unlimited. The default is 256K. |
| MAX_SCROLLABLE_CACHE_CURSOR_SPACE_ON_DISK   | In kilobytes, the maximum disk space one scrollable cursor can consume. If you specify 0, no disk storing is allowed. If you specify -1, the space is unlimited. The default value is 0.    |
| TEMPORARY_DIRECTORY                         | The temporary directory for the URL parameter. If you do not specify this parameter, UniJDBC uses the value defined in the system properties sheet.                                         |

---

### Connection URL Parameters



---

## Example

The following example illustrates creating a scrollable cursor.

```
=====
Example - Select
=====
import java.sql.*;

public class UniJDBCExSelect {

    public static void main(String[] argv) {

        try {
            String MyHost = "localhost";
            String MyAccount = "HS.SALES";
            String userid = "username";
            String passwd = "password";

            // generate URL
            String url = "jdbc:ibm-u2://" + MyHost + "/" + MyAccount ;

            // Load driver and connect to server
            Class.forName("com.ibm.u2.jdbc.UniJDBCDriver");
            Connection con = DriverManager.getConnection(url, userid, passwd);

            // Execute an SQL Query
            String sql = "select @ID, CITY, STATE, ZIP, PHONE CUSTOMER ORDER";
            Statement stmt = con.createStatement(
                ResultSet.TYPE_SCROLL_INSENSITIVE, ResultSet.CONCUR_READ_ONLY);

            ResultSet rs = stmt.executeQuery ( sql );

            rs.absolute(10);
            System.out.println("pos = " + rs.getRow + "@ID = "
                + rs.getString("@ID"));

            rs.relative(-2) ;
            System.out.println("pos = " + rs.getRow + "@ID = "
                + rs.getString("@ID"));

            rs.next() ;
            System.out.println("pos = " + rs.getRow + "@ID = "
                + rs.getString("@ID"));

            rs.previous() ;
            System.out.println("pos = " + rs.getRow + "@ID = "
                + rs.getString("@ID"));

            rs.close() ;
            stmt.close() ;
            System.out.println( "End of SC query test." ) ;
            con.close();

        }
        catch ( SQLException e ) {
            System.out.println("Ex-Message :" + e.getMessage());
        }
    }
}
```

```
        System.out.println("Ex-Code :" + e.getErrorCode() );
        System.out.println("Ex-SQLState:" + e.getSQLState() );
        System.out.println("Ex-Next :" + e.getNextException() );
        e.printStackTrace() ;
    }
    catch ( Exception e) {
        System.out.println("Exception caught:"+e) ;
        e.printStackTrace() ;
    }
}
```

---

# IBM for UniData and UniVerse JDBC Code Examples

|                                               |      |
|-----------------------------------------------|------|
| UniJDBC Example Program - List File . . . . . | 7-3  |
| UniJDBC Short Examples . . . . .              | 7-7  |
| Select Example . . . . .                      | 7-7  |
| UniJDBC Prepared Statement Example . . . . .  | 7-9  |
| UniJDBC Callable Statement Example . . . . .  | 7-10 |
| UniJDBC DatabaseMetaData Example . . . . .    | 7-11 |

This chapter provides some sample programs with comments throughout, describing the use of some of the methods of the IBM JDBC Driver for UniData and UniVerse.

This Chapter contains the following sections:

- [“UniJDBC Example Program - List File”](#) - This program shows a step-by-step example for connecting to the server, loading the driver and executing several SQL statements.
- [“UniJDBC Short Examples”](#) - This section contains examples for executing an SQL statement, a prepared SQL statement, a Callable statement and how to access DataBase Metadata,.

These examples are available in the \jdbc\samples directory of the UniDK installation.

---

## UniJDBC Example Program - List File

```
import java.sql.*;
import java.io.*;

/**
 * A sample program to test the IBM JDBC Driver for UniData and UniVerse.
 */
public class jdbcsample {

    public static void main(String[] argv)
    {
        try {
            BufferedReader in = new BufferedReader(new
InputStreamReader(System.in));

            System.out.println("\n\nThis is the JDBC sample program");
            System.out.println("~~~~~");
            System.out.println("\nThis program connects to a U2 account (schema");
            System.out.println("and lists the CUSTOMER file.");

            //-----
            // Connect to the U2 server
            //-----

            // Obtain path to the server account
            System.out.println("\nEnter destination account name: ");
            String account = in.readLine();

            // Get user name
            System.out.println("Enter valid server User Name: ");
            String userid = in.readLine();

            // Get password
            System.out.println("Enter password for user: ");
            String passwd = in.readLine();

            // Get host machine name
            System.out.println("Enter host machine name(localhost or machine
name): ");
            String host = in.readLine();

            // Get database type
            System.out.println("Enter the dbmstype: UNIDATA or UNIVERSE");
            String dbmstype = in.readLine();

            // generate URL
            String url = "jdbc:ibm-
u2://" + host + "/" + account + "?dbmstype=" + dbmstype;

            //Load driver and connect to server
            Class.forName("com.ibm.u2.jdbc.UniJDBCDriver");
            Connection con = DriverManager.getConnection(url, userid, passwd);

            System.out.println("\n*--- Connection successful ---*\n");

            //-----

```

```

// First example
//-----
System.out.println("1. Select from CUSTOMER -----
-");
    testQuery( con ) ;

//-----
// Second example
//-----
    System.out.println("2. Transaction test -----
-");
    testRollback( con ) ;

    con.close();
} catch ( SQLException e ) {
    System.out.println("Ex-Message :" + e.getMessage());
    System.out.println("Ex-Code      :" + e.getErrorCode() );
    System.out.println("Ex-SQLState:" + e.getSQLState());
    System.out.println("Ex-Next      :" + e.getNextException());
    e.printStackTrace() ;
    System.gc();
} catch ( Exception e ) {
    System.out.println("Exception caught:"+e) ;
    e.printStackTrace() ;
}
}

/**
 * Select something from CUSTOMER table.
 * @param con The JDBC connection object.
 */
public static void testQuery(Connection con)
    throws SQLException
{
    Statement stmt = con.createStatement();
    String sql = "select @ID, CITY, STATE, ZIP, PHONE from CUSTOMER";

    // Execute the SELECT statement
    ResultSet rs = stmt.executeQuery(sql);

    // Get result of first five records
    System.out.println("list selected columns for the first five
records:");
    int i = 1;
    while (rs.next() && i < 6)
    {
        System.out.println("\nRecord "+ i +" :");
        System.out.println("@ID : " + rs.getString(1));
        System.out.println("CITY : " + rs.getString(2));
        System.out.println("STATE : " + rs.getString(3));
        System.out.println("ZIP : " + rs.getString(4));
        System.out.println("PHONE : " + rs.getString(5));
        i++;
    }

    rs.close();
    stmt.close() ;
    System.out.println("\n*--- QUERY test is done successful ---*\n");
}

```

```

/**
 * Transaction test. It will:
 * (1) begin a transaction
 * (2) update STATE of CUSTOMER to CA
 * (3) re-read that record to show the update
 * (4) roll the transaction back
 * (5) re-read that record to show original value
 * @param con The JDBC connection object.
 */
static public void testRollback(Connection con)
    throws SQLException
{
    System.out.println("\nThis section will:");
    System.out.println("(1) begin a transaction");
    System.out.println("(2) update STATE of CUSTOMER to CA");
    System.out.println("(3) re-read that record to show the update");
    System.out.println("(4) roll the transaction back");
    System.out.println("(5) re-read that record to show original value\n");

    // Set isolation level and start a transaction
    con.setTransactionIsolation(
java.sql.Connection.TRANSACTION_READ_COMMITTED);
    con.setAutoCommit( false );

    // Update the CUSTOMER file
    String sql = "update CUSTOMER set STATE = 'CA' where @ID = '2'";
    Statement stmt = con.createStatement();

    int rows = stmt.executeUpdate(sql);

    System.out.println(rows + " rows updated.");
    stmt.close();

    //Read the record to show the update
    String lsq1 = "SELECT STATE FROM CUSTOMER WHERE @ID = ?";
    PreparedStatement pstmt = con.prepareStatement(lsq1);

    pstmt.setString(1, "2");
    ResultSet rs = pstmt.executeQuery();

    while (rs.next())
    {
        System.out.println("updated STATE is " + rs.getString(1));
    }
    pstmt.close();

    //Rollback the transaction
    con.rollback();
    System.out.println("\nTransaction is rolled back.\n");

    //re-Read the record to show original value
    pstmt = con.prepareStatement(lsq1);

    pstmt.setString(1, "2");
    rs = pstmt.executeQuery();

    while (rs.next())
    {
        System.out.println("original STATE is " + rs.getString(1));
    }
}

```

```
pstmt.close() ;

System.out.println("\n*--- Transaction test is done successful ---
*\n");
}
```



---

# UniJDBC Short Examples

## Select Example

```
=====
Example - Select
=====

import java.sql.*;

public class UniJDBCExSelect {

    public static void main(String[] argv) {

        try {
            String MyHost = "localhost";
            String MyAccount = "HS.SALES";
            String userid = "username";
            String passwd = "password";

            // generate URL
            String url = "jdbc:ibm-u2://" + MyHost + "/" + MyAccount ;

            // Load driver and connect to server
            Class.forName("com.ibm.u2.jdbc.UniJDBCDBDriver");
            Connection con = DriverManager.getConnection(url, userid, passwd);

            // Execute an SQL Query
            Statement stmt = con.createStatement();
            String sql = "select @ID, CITY, STATE, ZIP, PHONE from CUSTOMER";
            ResultSet rs = stmt.executeQuery(sql);
            int i = 1;
            while (rs.next()) {
                System.out.println("\nRecord "+ i + " :");
                System.out.println("@ID : " + rs.getString("@ID"));
                System.out.println("CITY : " + rs.getString("CITY"));
                System.out.println("STATE : " + rs.getString("STATE"));
                System.out.println("ZIP : " + rs.getString("ZIP"));
                System.out.println("PHONE : " + rs.getString("PHONE"));
                i++;
            }
            rs.close();
            stmt.close() ;
            con.close();
        }
        catch ( SQLException e ) {
            System.out.println("Ex-Message : " + e.getMessage());
            System.out.println("Ex-Code : " + e.getErrorCode() );
            System.out.println("Ex-SQLState: " + e.getSQLState() );
            System.out.println("Ex-Next : " + e.getNextException() );
            e.printStackTrace() ;
        }
        catch ( Exception e ) {
            System.out.println("Exception caught:"+e) ;
            e.printStackTrace() ;
        }
    }
}
```

} }

## UniJDBC Prepared Statement Example

```
=====
Example - Prepared Statement
=====

import java.sql.*;

public class UniJDBCExPrepStmt {

    public static void main(String[] argv) {

        try {
            String MyHost = "localhost";
            String MyAccount = "HS.SALES";
            String userid = "username";
            String passwd = "password";

            // generate URL
            String url = "jdbc:ibm-u2://" + MyHost + "/" + MyAccount ;

            // Load driver and connect to server
            Class.forName("com.ibm.u2.jdbc.UniJDBCDriver");
            Connection con = DriverManager.getConnection(url, userid, passwd);

            // Execute prepared SQL statement
            String sql = "update CUSTOMER set CITY= ?,STATE= ?, "
                + "PHONE = '(603)555-3212' WHERE @ID = '1'";
            PreparedStatement stmt = con.prepareStatement(sql);
            stmt.setString(1, "Concord");
            stmt.setString(2, "NH");
            int rows = stmt.executeUpdate() ;
            System.out.println( rows + " rows updated." ) ;
            stmt.close() ;
            con.close();
        }
        catch ( SQLException e ) {
            System.out.println("Ex-Message :" + e.getMessage());
            System.out.println("Ex-Code :" + e.getErrorCode() ) ;
            System.out.println("Ex-SQLState:" + e.getSQLState() ) ;
            System.out.println("Ex-Next :" + e.getNextException() ) ;
            e.printStackTrace() ;
        }
        catch ( Exception e) {
            System.out.println("Exception caught:"+e) ;
            e.printStackTrace() ;
        }
    }
}
```

## UniJDBC Callable Statement Example

```
=====
Example - Callable Statement
=====

import java.sql.*;

public class UniJDBCExCallStmt {

    public static void main(String[] argv) {

        try {
            String MyHost = "localhost";
            String MyAccount = "HS.SALES";
            String userid = "username";
            String passwd = "password";

            // generate URL
            String url = "jdbc:ibm-u2://" + MyHost + "/" + MyAccount ;

            // Load driver and connect to server
            Class.forName("com.ibm.u2.jdbc.UniJBCDriver");
            Connection con = DriverManager.getConnection(url, userid, passwd);

            // Execute Callable statement
            // Note: HS.OLEDBACCTS is a globally cataloged BASIC program,
            // and so requires the asterisk in front of the name.
            // locally cataloged programs do not require the asterisk
            String sql = "call *HS.OLEDBACCTS(?)";
            CallableStatement stmt = con.prepareCall( sql );
            stmt.registerOutParameter(1, -1);
            ResultSet rs = stmt.executeQuery();
            System.out.println( "HS.OLEDBACCTS returns : " + stmt.getString( 1
));
            rs.close();
            stmt.close();
            con.close();
        }
        catch ( SQLException e ) {
            System.out.println("Ex-Message :" + e.getMessage());
            System.out.println("Ex-Code :" + e.getErrorCode() );
            System.out.println("Ex-SQLState:" + e.getSQLState() );
            System.out.println("Ex-Next : " + e.getNextException() );
            e.printStackTrace() ;
        }
        catch ( Exception e) {
            System.out.println("Exception caught:"+e) ;
            e.printStackTrace() ;
        }
    }
}
```

## UniJDBC DatabaseMetaData Example

```
=====
Example - DatabaseMetaData
=====

import java.sql.*;

public class UniJDBCExGetPrimKeys {

    public static void main(String[] argv) {

        try {
            String MyHost = "localhost";
            String MyAccount = "HS.SALES";
            String MySchema = null;
            String MyTable = "CUSTOMER";
            String userid = "username";
            String passwd = "password";

            // generate URL
            String url = "jdbc:ibm-u2://" + MyHost + "/" + MyAccount ;

            // Load driver and connect to server
            Class.forName("com.ibm.u2.jdbc.UniJBCDriver");
            Connection con = DriverManager.getConnection(url, userid, passwd);

            // get Database Metadata
            DatabaseMetaData dbmetadata = con.getMetaData();
            ResultSet rs = dbmetadata.getPrimaryKeys(null, MySchema, MyTable);
            while( rs.next() ) {
                for( int i=1; i<=5; i++ ) {
                    System.out.println("Column["+i+"]="+rs.getString(i)) ;
                }
            }
            rs.close();
            con.close();
        }
        catch ( SQLException e ) {
            System.out.println("Ex-Message :" + e.getMessage());
            System.out.println("Ex-Code :" + e.getErrorCode() );
            System.out.println("Ex-SQLState:" + e.getSQLState() );
            System.out.println("Ex-Next :" + e.getNextException() );
            e.printStackTrace() ;
        }
        catch ( Exception e ) {
            System.out.println("Exception caught:"+e) ;
            e.printStackTrace() ;
        }
    }
}
```

# *Index*

## A

accessing  
     UniData data 3-2  
     UniVerse data 4-2  
 accessing Database MetaData 5-7  
 association keys 4-19  
 associations 4-19

## C

character columns 4-25  
 classes  
     com.ibm.u2.jdbcx.UniJDBCConnect  
       ionPoolDataSource 5-14  
     com.ibm.u2.jdbcx.UniJDBCDataSou  
       rce 5-14  
 classpath  
     setting 2-5, 2-6  
 Class.forName  
     method 5-3  
 clients  
     hardware and software  
       requirements 2-3  
 columns  
     character 4-25  
     multivalued 3-7  
     unassociated multivalued 4-19  
     @ASSOC\_ROW 4-19  
 commands  
     HS.SCRUB 4-28  
 COMMIT statement 5-12  
 com.ibm.u2.jdbcx.UniJDBCConnectio  
     nPoolDataSource 5-14  
 com.ibm.u2.jdbcx.UniJDBCDataSource  
     e 5-14

configuration files  
     unirpcservices file 3-4  
 configuration parameters  
     MAXFETCHBUFF 4-25  
     MAXFETCHCOLS 4-25  
 configurations  
     defining 3-5  
     UCI connection timeout 3-4  
 Connection  
     Connection.getCatalog 5-24  
     Connection.getTypeMap 5-24  
     Connection.setCatalog 5-24  
     Connection.setTypeMap 5-24  
 Connection Pool  
     tuning 5-18  
 connection string  
     properties 5-4  
 ConnectionPoolDataSource object  
     properties 5-19  
 Connection.commit method 5-12  
 Connection.setAutoCommit 5-12  
 conversion errors 4-26

## D

daemon, unirpc 4-2  
 data  
     conversion errors 4-26  
     length 3-7  
     multivalued 3-7  
 data access  
     UniData 3-2  
 data types 3-7  
     defining 4-23  
     mapping 5-9  
 DatabaseMetaData

DatabaseMetaData.getBestRowIdentifier 5-24  
DatabaseMetaData.getCatalogs 5-24  
DatabaseMetaData.getColumnPrivileges 5-24  
DatabaseMetaData.getCrossReferences 5-25  
DatabaseMetaData.getExportedKeys 5-24  
DatabaseMetaData.getImportedKeys 5-24  
DatabaseMetaData.getIndexInfo 5-25  
DatabaseMetaData.getProcedureColumns 5-24  
DatabaseMetaData.getProcedures 5-24  
DatabaseMetaData.getTablePrivileges 5-24  
DatabaseMetaData.getUDTs 5-25  
DatabaseMetaData.getVersionColumns 5-24  
DATATYPE field 4-23, 4-25  
defining configurations 3-5  
Deploying DataSource Object 5-15

## E

empty strings 4-26  
error handling 5-11  
errors  
data conversion 4-26  
trace logs 3-5  
events, tracing 3-5  
Example  
UniJDBC Example 1 7-3  
UniJDBC Example 2 7-7

## F

fields  
DATATYPE 4-23, 4-25  
FORMAT 4-25  
multivalued 3-7  
SQLTYPE 4-23, 4-25  
FORMAT field 4-25  
functionality, UniOLEDB 5-2

## G

getColumns method 5-7  
getSchema method 5-7  
getTables method 5-7

## H

handling errors 5-11  
handling transactions 5-12  
Hardware  
requirements 2-3  
hardware requirements  
client machine 2-3  
server machine 2-3  
HS.SCRUB utility  
adding @EMPTY.NULL record to dictionary 4-27  
adding @SELECT record to dictionary 4-27  
command syntax 4-28  
functional overview 4-26  
running 4-27

## I

Installing  
Windows 2-5  
installing  
UNIX 2-6  
interfaces  
CallableStatement 5-8  
DatabaseMetaData 5-7  
javax.sql.ConnectionPoolDataSource 5-14  
javax.sql.DataSource 5-14  
JDBC 1-4  
PreparedStatement 5-8  
ResultSet 5-8  
ResultSetMetaData 5-8  
Statement 5-8  
Isolation level  
changing 5-13  
Isolation levels 5-13  
TRANSACTION\_READ\_COMMITTED 5-13  
TRANSACTION\_READ\_UTCOMMITTED 5-13

TRANSACTION\_REPEATABLE\_READ 5-13  
TRANSACTION\_SERIALIZABLE 5-13

## J

java.sql.Types.ARRAY 5-23  
java.sql.Types.BINARY 5-23  
java.sql.Types.BLOB 5-23  
java.sql.Types.CLOB 5-23  
java.sql.Types.DISTINCT 5-23  
java.sql.Types.JAVA\_OBJECT 5-23  
java.sql.Types.LONGVARBINARY 5-23  
java.sql.Types.NULL 5-23  
java.sql.Types.REF 5-23  
java.sql.Types.STRUCT 5-23  
java.sql.Types.TIMESTAMP 5-23  
java.sql.Types.VARBINARY 5-23  
JDBC  
class 1-4  
connection string 5-4  
data conversion 5-10  
database connections 5-5  
definition 1-4  
Deploying DataSource Object 5-15  
driver definition 1-6  
driver types 1-6  
exception 1-4  
format of database URL 5-5  
interfaces 1-4  
JNDI 5-14  
loading the driver 5-3  
overview 1-7  
Pooled DataSource Object 5-16  
specification 1-7  
JDBC 2.0 Optional Package 5-14  
JNDI 5-14

## K

keys  
association 4-19  
primary 4-19

## L

length  
     of character columns 4-25  
     of data 3-7  
 logs, trace 3-5

## M

Manager.getConnection  
     method 5-4  
 mapping  
     data types 5-9  
 MAXFETCHBUFF parameter 4-25  
 MAXFETCHCOLS parameter 4-25  
 menus  
     UniVerse Server Administration 4-10  
 methods  
     Class.forName method 5-3  
     Connection.commit 5-12  
     Connection.setAutoCommit 5-12  
     getColumns 5-7  
     getSchema 5-7  
     getTables 5-7  
     Manager.getConnection 5-4  
     unsupported methods 5-23  
 multivalued  
     data 3-7  
 multivalued columns  
     unassociated 4-19

## N

Note  
     If you already have a version of the UniDK installed, the installation process prompts to Repair, Modify or Remove the installation of the UniDK. Select Remove to uninstall the existing version. Once you have uninstalled the older version of the Uni 2-5  
 null value 4-26

## P

Pooled DataSource Object 5-16  
     deploying 5-17  
 precision 4-25  
 PreparedStatement  
     PreparedStatement.setArray(int, java.sql.Array) 5-24  
     PreparedStatement.setBlob(int, java.sql.Blob) 5-24  
     PreparedStatement.setCharacterStream(int, java.io.Reader) 5-24  
     PreparedStatement.setClob(int, java.sql.Clob) 5-24  
     PreparedStatement.setRef(int, java.sql.Ref) 5-24  
 PreparedStatment  
     PreparedStatement.addBatch() 5-24  
 primary keys 4-19  
 properties  
     connection string 5-4  
     ConnectionPoolDataSource object 5-19

## Q

Querying the database 5-8

## R

record IDs 4-19  
 records  
     @ 4-27  
     @EMPTY.NULL 4-26, 4-27  
     @SELECT 4-27  
 Requirements  
     hardware 2-3  
 requirements  
     UniRPC 3-3  
 ResultSet  
     ResultSet.getCharacterStream 5-25  
     ResultSetMetaData  
         ResultSetMetadata.getColumnClassName 5-25  
 ROLLBACK statement 5-12

## S

servers  
     hardware and software requirements 2-3  
 set  
     classpath 2-6  
     setting up UniOLEDB 2-2, 7-2  
 SICA 4-24  
 software requirements  
     client machine 2-3  
     server machine 2-3  
 SQLTYPE field 4-23, 4-25  
 starting  
     HS.SCRUB utility 4-27  
     UniRPC 3-3  
 Statement  
     Statement.addBatch(string) 5-23  
     Statement.cancel() 5-23  
     Statement.clearBatch() 5-23  
     Statement.executeBatch() 5-23

## T

TCP/IP 2-3, 2-4  
 trace levels 3-5  
 trace logs 3-5  
 tracing events 3-5  
 transactions 5-12  
 troubleshooting  
     trace logs 3-5

## U

UCI  
     connection timeout configuration 3-4  
 uci.config file  
     setting up 2-5  
 UDTHOME directory 3-5  
 ud\_database file  
     trace log settings 3-5  
 unassociated multivalued columns 4-19  
 UniData  
     data access 3-2  
 unidata  
     making accounts accessible 3-5



presenting data in JDBC-accessible  
format 3-7

#### UniOLEDB

functionality 5-2  
setting up 2-2, 7-2

#### UniRPC

Requirements 3-3  
requirements 3-3  
starting 3-3  
verifying that it is running 3-3

unirpc daemon 4-2

unirpcservices 3-3

unirpcservices file 3-4

unishared directory 3-4

#### universe

accessing columns in tables and  
files 4-18  
accessing tables and files 4-3  
empty-null mapping 4-26  
fixing tables and files 4-26  
making data meaningful to JDBC 4-  
23  
making files visible 4-12  
scalar functions 4-29  
special characters 4-22  
system administration menu 4-10

UniVerse Server Administration  
menu 4-10

#### UNIX

installing 2-6

#### unsupported

Data Types 5-23

#### utility, HS.SCRUB

adding @EMPTY.NULL record to  
dictionary 4-27  
adding @SELECT record to  
dictionary 4-27  
command syntax 4-28  
functional overview 4-26  
running 4-27

installing 2-5

---

## Symbols

@ record 4-27

@ASSOC\_ROW column 4-19

@EMPTY.NULL record 4-26, 4-27

@SELECT record 4-27

---

## V

verifying that UniRPC is running 3-3

---

## W

Windows