



Using Caché with JDBC

Version 2010.1
17 February 2010

Using Caché with JDBC

Caché Version 2010.1 17 February 2010

Copyright © 2010 InterSystems Corporation

All rights reserved.

This book was assembled and formatted in Adobe Page Description Format (PDF) using tools and information from the following sources: Sun Microsystems, RenderX, Inc., Adobe Systems, and the World Wide Web Consortium at www.w3c.org. The primary document development tools were special-purpose XML-processing applications built by InterSystems using Caché and Java.



Caché WEBLINK, Distributed Cache Protocol, M/SQL, M/NET, and M/PACT are registered trademarks of InterSystems Corporation.



InterSystems Jalapeño Technology, Enterprise Cache Protocol, ECP, and InterSystems Zen are trademarks of InterSystems Corporation.

All other brand or product names used herein are trademarks or registered trademarks of their respective companies or organizations.

This document contains trade secret and confidential information which is the property of InterSystems Corporation, One Memorial Drive, Cambridge, MA 02142, or its affiliates, and is furnished for the sole purpose of the operation and maintenance of the products of InterSystems Corporation. No part of this publication is to be used for any other purpose, and this publication is not to be reproduced, copied, disclosed, transmitted, stored in a retrieval system or translated into any human or computer language, in any form, by any means, in whole or in part, without the express prior written consent of InterSystems Corporation.

The copying, use and disposition of this document and the software programs described herein is prohibited except to the limited extent set forth in the standard software license agreement(s) of InterSystems Corporation covering such programs and related documentation. InterSystems Corporation makes no representations and warranties concerning such software programs other than those set forth in such standard software license agreement(s). In addition, the liability of InterSystems Corporation for any losses or damages relating to or arising out of the use of such software programs is limited in the manner set forth in such standard software license agreement(s).

THE FOREGOING IS A GENERAL SUMMARY OF THE RESTRICTIONS AND LIMITATIONS IMPOSED BY INTERSYSTEMS CORPORATION ON THE USE OF, AND LIABILITY ARISING FROM, ITS COMPUTER SOFTWARE. FOR COMPLETE INFORMATION REFERENCE SHOULD BE MADE TO THE STANDARD SOFTWARE LICENSE AGREEMENT(S) OF INTERSYSTEMS CORPORATION, COPIES OF WHICH WILL BE MADE AVAILABLE UPON REQUEST.

InterSystems Corporation disclaims responsibility for errors which may appear in this document, and it reserves the right, in its sole discretion and without notice, to make substitutions and modifications in the products and practices described in this document.

For Support questions about any InterSystems products, contact:

InterSystems Worldwide Customer Support

Tel: +1 617 621-0700

Fax: +1 617 374-9391

Email: support@InterSystems.com

Table of Contents

About This Book	1
1 Introduction to Caché JDBC	3
2 Establishing JDBC Connections	5
2.1 Using CacheDataSource to Connect	5
2.2 Using DriverManager to Connect	6
2.3 Using a Connection Pool	6
2.4 Defining a JDBC Connection URL	7
2.4.1 Required Parameters	7
2.4.2 Optional Parameters	7
2.4.3 Username and Password	8
2.4.4 Setting the Port Parameter at the Command Line	8
2.5 Caché JDBC Connection Properties	8
2.5.1 Listing Connection Properties	9
3 Accessing JDBC Databases	11
3.1 A Simple JDBC Application	11
3.2 Using Statements	12
3.2.1 Using Statement to Execute a SELECT	12
3.2.2 Executing a Prepared Statement	13
3.2.3 Using Callable Statements to Execute Stored Procedures	13
3.2.4 Returning Multiple Result Sets	14
3.2.5 Statement Pooling	15
3.3 Inserting and Updating Data	15
3.3.1 Inserting Data and Retrieving Generated Keys	15
3.3.2 Scrolling a Result Set	16
3.3.3 Updating a Scrollable Result Set	17
3.3.4 Using Transactions	17
3.4 Logging for JDBC Applications	18
3.4.1 Enabling Logging for Caché	19
3.4.2 Enabling Logging for JDBC	19
3.4.3 Enabling Logging for the SQL Gateway	19
4 Using the Caché SQL Gateway with JDBC	21
4.1 Creating JDBC SQL Gateway Connections for External Sources	21
4.1.1 Creating a JDBC Gateway Connection	21
4.1.2 Creating a JDBC Connection to Caché via the SQL Gateway	22
4.1.3 Implementation-specific Options	23
4.2 Using the JDBC SQL Gateway Programmatically	24
5 Caché JDBC Compliance	27
5.1 JDBC 4.0 Interfaces and Classes	27
5.2 The JDBC Core API (java.sql)	28
5.2.1 java.sql Interfaces	28
5.2.2 java.sql Classes	29
5.2.3 java.sql Exceptions	29
5.3 The JDBC Optional Package API (javax.sql)	30
5.3.1 javax.sql Interfaces	30
5.3.2 javax.sql Classes	30

5.4 Interfaces with Unsupported Optional Methods	30
5.4.1 CallableStatement — Unsupported Methods	31
5.4.2 Connection — Unsupported or Restricted Methods	32
5.4.3 DatabaseMetaData — Variant Methods	33
5.4.4 PreparedStatement — Unsupported Methods	33
5.4.5 ResultSet — Unsupported Methods	34
5.4.6 Statement — Unsupported or Restricted Methods	35
5.5 Caché JDBC Additions and Extensions	36
5.5.1 CallableStatement Additional Method	36
5.5.2 CacheConnectionPoolDataSource	36
5.5.3 CacheDataSource	37

List of Tables

Table 4–1: Calling JDBC Methods from %Net.Remote.Java.JDBCGateway	25
---	----

About This Book

This book describes how to use the Caché JDBC driver, which enables you to connect to Caché from an external application (such as a development tool or report writer) via JDBC, and allows Caché to access external JDBC data sources.

Who This Book Is For

To use the Caché JDBC driver, you should be familiar with the Java programming language and have some understanding of how Java is configured on your operating system. If you are performing custom configuration of the Caché JDBC driver on UNIX®, you should also be familiar with compiling and linking code, writing shell scripts, and other such tasks.

Organization of This Book

This book is organized as follows:

- The chapter “[Introduction to Caché JDBC](#)” provides basic information about the Caché JDBC driver, and demonstrates how to set up and use a simple Caché JDBC application.
- The chapter “[Establishing JDBC Connections](#)” gives a detailed description of the various ways to establish a JDBC connection to a Caché database.
- The chapter “[Accessing JDBC Databases](#)” provides some simple examples that show how to access and manipulate data in a Caché database from a Java application using the Caché JDBC driver.
- The chapter “[Using the Caché SQL Gateway with JDBC](#)” describes how to access external databases from Caché using JDBC and the Caché SQL Gateway.
- The chapter “[Caché JDBC Compliance](#)” lists all members of the JDBC 4.0 API, describes all Caché-specific additional features of the Caché JDBC driver, and indicates which optional features have been omitted.

Related Information

For related information, see the following sources:

- The book *Using Java with Caché* describes the Caché Java Binding.
- The book *Using Caché with ODBC* describes the Caché ODBC driver.
- The *Caché with Java and J2EE QuickStart Tutorial* is available in the *Caché Tutorials* section of the Caché documentation.

1

Introduction to Caché JDBC

Caché provides a high-performance *type 4* JDBC database driver. The Caché JDBC driver is fully compliant with the JDBC 4.0 API, supporting all required interfaces and adhering to all JDBC 4.0 guidelines and requirements. Intersystems supports the driver for Java JDK releases 1.5 and 1.6.

You can use the Caché JDBC driver and SQL when you want to access your Caché data using a relational model. The [Java Binding](#) mechanism also uses the JDBC driver. You can mix relational and object-oriented database access to provide maximum flexibility to your application.

Using CacheDB.jar

The Caché JDBC driver is contained in CacheDB.jar. In the default Caché installation, two versions of this file are located under <cachsys>\dev\java\lib\, in subdirectories JDK15 and JDK16, corresponding to Java JDK releases 1.5 and 1.6. Use the version that matches your version of Java.

JavaDoc class documentation for the packages in CacheDB.jar is located in <cachsys>\dev\java\doc.

Running the JDBCQuery.java Sample Program

The JDBCQuery.java sample program (located in <cachsys>\dev\java\samples) uses JDBC to establish the connection from client to server, then performs an SQL query. The application is invoked from the command line and assumes that Caché and Java are running on the same machine.

This program's actions are:

- Establishing a connection to a particular namespace on a Caché server using JDBC.
- Invoking an SQL query using a Java Statement object.
- Using a Java ResultSet object to retrieve query metadata and results from the Caché server.

This application uses the Sample.Person class from the precompiled Caché examples in the SAMPLES namespace. You can use Caché Studio to see how this class is implemented in Caché.

2

Establishing JDBC Connections

This chapter provides some detailed examples of Java code that uses the Caché JDBC driver to accomplish basic tasks.

- [Using CacheDataSource to Connect](#) — Use `CacheDataSource` to load the driver and create a `java.sql.Connection` object, which is used to create the `Statement` object.
- [Using DriverManager to Connect](#) — Use the standard `DriverManager` class to create a connection.
- [Using a Connection Pool](#) — Use the `CacheConnectionPoolDataSource` class to control the connection pool for your Java client applications.
- [Defining a JDBC URL](#) — Specify the machine address, port number, and namespace to be accessed. The Caché JDBC driver also allows you to specify several other parameters.
- [Caché JDBC Connection Properties](#) — Specify several standard connection properties.

2.1 Using CacheDataSource to Connect

Use `com.intersys.jdbc.CacheDataSource` to load the driver and then create the `java.sql.Connection` object. This is the preferred method for connecting to a database and is fully supported by Caché.

Note: In earlier versions of JDBC, it was necessary to load the driver before connecting. For example:

```
Class.forName ("com.intersys.jdbc.CacheDriver").newInstance();
Connection dbconnection = DriverManager.getConnection(url,user,password);
```

JDBC 4.0 uses driver autoloading, which makes the `Class.forName()` call unnecessary for either `DriverManager` or `CacheDataSource`.

Here are the steps for using `CacheDataSource`:

1. Import the `java.sql` and `com.intersys.jdbc` packages:

```
import java.sql.*;
import com.intersys.jdbc.*;
```

2. Load the driver, then use `CacheDataSource` to create the connection and specify username and password:

```
try{
    CacheDataSource ds = new CacheDataSource();
    ds.setURL("jdbc:Cache://127.0.0.1:1972/Samples");
    ds.setUser("_system");
    ds.setPassword("SYS");
    Connection dbconnection = ds.getConnection();
}
```

Note: On some systems, Java may attempt to connect via IPv6 if you use `localhost` in the URL rather than the literal address, `127.0.0.1`. This applies on any system where the hostname resolves the same for IPv4 and IPv6.

3. Create a `java.sql.Statement` using the connection. The `Statement` object can be used to execute queries.

```
Statement stmt = dbconnection.createStatement();
```

4. Catch the exceptions thrown by the above methods:

```
catch (SQLException e){
    System.out.println(e.getMessage());
}
catch (ClassNotFoundException e){
    System.out.println(e.getMessage());
}
```

2.2 Using DriverManager to Connect

The standard `DriverManager` class can also be used to create a connection. The following code demonstrates one possible way to do so:

```
Class.forName("com.intersys.jdbc.CacheDriver").newInstance();
String url="jdbc:Cache://127.0.0.1:1972/SAMPLES";
String username = "_SYSTEM";
String password = "SYS";
dbconnection = DriverManager.getConnection(url,username,password);
```

You can also pass connection properties to `DriverManager` in a `Properties` object, as demonstrated in the following code:

```
String url="jdbc:Cache://127.0.0.1:1972/SAMPLES";
java.sql.Driver drv = java.sql.DriverManager.getDriver(url);

java.util.Properties props = new Properties();
props.put("user",username);
props.put("password",password);
java.sql.Connection dbconnection = drv.connect(url, props);
```

See [Caché JDBC Connection Properties](#) for a complete list of the properties used by the Caché JDBC driver.

2.3 Using a Connection Pool

The `com.intersys.jdbc.CacheConnectionPoolDataSource` class implements the `javax.sql.ConnectionPoolDataSource` interface, providing a connection pool for your Java client applications. Here are the steps for using a connection pool with Caché:

1. Import the needed packages:

```
import com.intersys.jdbc.*;
import java.sql.*;
```

2. Instantiate a `CacheConnectionPoolDataSource` object. Use the `restart()` method to close all of the physical connections and empty the pool. Use `setURL()` to set the database URL (see [Defining a JDBC URL](#)) for the pool's connections.

```
CacheConnectionPoolDataSource pds = new CacheConnectionPoolDataSource();
pds.restartConnectionPool();
pds.setURL("jdbc:Cache://127.0.0.1:1972/Samples");
pds.setUser("_system");
pds.setPassword("SYS");
```

3. Initially, `getPoolCount` returns 0.

```
System.out.println(pds.getPoolCount()); //outputs 0.
```

4. Use `CacheConnectionPoolDataSource.getConnection()` to retrieve a database connection from the pool.

```
Connection dbConnection = pds.getConnection();
```

CAUTION: Caché driver connections must always be obtained by calling the `getConnection()` method (inherited from `CacheDataSource`). Do not use the `getPooledConnection()` methods, which are for use only within the Caché driver.

5. Close the connection. Now `getPoolCount` returns 1.

```
dbConnection.close();
System.out.println(pds.getPoolCount()); //outputs 1
```

2.4 Defining a JDBC Connection URL

A `java.sql.Connection` URL supplies the connection with information about the machine address, port number, and namespace to be accessed. The Caché JDBC driver also allows you to use several optional parameters.

2.4.1 Required Parameters

The URL specifies the machine address, port number, and namespace to be accessed.

<i>machine</i>	IP address or host name. For example, both <code>localhost</code> and <code>127.0.0.1</code> indicate the local machine.
<i>port</i>	TCP/IP port number for the connection. For example, <code>56772</code> is the standard Caché Superserver port.
<i>namespace</i>	Namespace to connect to. For example, <code>Samples</code> is the namespace containing Caché sample programs.

The minimal required URL syntax is:

```
jdbc:Cache://<machine>:<port>/<namespace>
```

For example, the following URL specifies *machine* (IP address) as `127.0.0.1`, *port* as `56772`, and *namespace* as `Samples`:

```
jdbc:Cache://127.0.0.1:56772/Samples
```

2.4.2 Optional Parameters

The Caché JDBC driver also allows you to specify several optional parameters. The syntax is:

```
jdbc:Cache://<machine>:<port>/<namespace>/<logfile>:<eventclass>:<nodelay>
```

where the optional parameters are defined as follows:

<i>logfile</i>	specifies a JDBC log file (see Enabling Logging for JDBC).
<i>eventclass</i>	sets the transaction Event Class for this CacheDataSource object. See the CacheDataSource setEventClass() method for a complete description.
<i>nodelay</i>	sets the <code>TCP_NODELAY</code> option if connecting via a CacheDataSource object. Toggling this flag can affect the performance of the application. Valid values are <code>true</code> and <code>false</code> . If not set, it defaults to <code>true</code> . Also see the getNodeDelay() and setNodeDelay() methods of CacheDataSource .

Each of these optional parameters can be defined individually, without setting the others. For example, the following URL just sets the required parameters and the *nodelay* option:

```
jdbc:Cache://127.0.0.1:56772/Samples/::false
```

2.4.3 Username and Password

It is also possible to specify the username and password in the URL, although this is discouraged. If password and username are supplied as part of the URL string, they will be used in order to connect. Otherwise, other mechanisms already in place will be invoked. The syntax is:

```
jdbc:Cache://<machine>:<port>/<namespace>/<options>?username=<string1>&password=<string2>
```

For example, the following URL sets the required parameters, the *nodelay* option, and then the username and password:

```
jdbc:Cache://127.0.0.1:56772/Samples/::false?username="_SYSTEM"&password="SYS"
```

The username and password strings are case sensitive.

2.4.4 Setting the Port Parameter at the Command Line

The `com.intersys.port` property can be used to set the `port` parameter of the URL at the command line. For example, even though sample program `CJTest1.java` sets the port number as a constant equal to 1972, it can be run on port 9523 using the following command:

```
java -cp ../lib/CacheDB.jar -Dcom.intersys.port=9523 CJTest1
```

The current value of this property can be retrieved programmatically with the following code:

```
String myport = java.lang.System.getProperty ("com.intersys.port");
```

2.5 Caché JDBC Connection Properties

The Caché JDBC driver supports several connection properties, which can be set by passing them to `DriverManager` (as described in [Using DriverManager to Connect](#)).

The following properties are supported:

user	Required. String indicating Username. Default = ""
password	Required. String indicating Password. Default = ""
TCP_NODELAY	Optional. Boolean indicating TCP/IP NoDelay Flag. Default = true.
SO_SNDBUF	Optional. Integer indicating TCP/IP SO_SNDBUF value (SendBufferSize). Default = 8760.
SO_RCVBUF	Optional. Integer indicating TCP/IP SO_RCVBUF value(ReceiveBufferSize). Default = 8760.
TransactionIsolationLevel	Optional. java.sql.Connection constant indicating Transaction Isolation Level. Valid values are TRANSACTION_READ_UNCOMMITTED (the default) or TRANSACTION_READ_COMMITTED.
service principal	Optional. String indicating Service Principal Name. Default = null.
connection security level	Optional. Integer indicating Connection Security Level. Valid levels are 0, 1, 2, or 3. Default = 0.

2.5.1 Listing Connection Properties

Code similar to the following can be used to list the available properties for any compliant JDBC driver:

```

java.sql.Driver drv = java.sql.DriverManager.getDriver(url);
java.sql.Connection dbconnection = drv.connect(url, user, password);
java.sql.DatabaseMetaData meta = dbconnection.getMetaData();

System.out.println ("\n\nDriver Info: =====");
System.out.println (meta.getDriverName());
System.out.println ("release " + meta.getDriverVersion() + "\n");

java.util.Properties props = new Properties();
DriverPropertyInfo[] info = drv.getPropertyInfo(url,props);
for(int i = 0; i < info.length; i++) {
    System.out.println ("\n" + info[i].name);
    if (info[i].required) {System.out.print ("    Required");}
    else {System.out.print ("    Optional");}
    System.out.println ("", default = " + info[i].value);
    if (info[i].description != null)
        System.out.println ("    Description: " + info[i].description);
    if (info[i].choices != null) {
        System.out.println ("    Valid values: ");
        for(int j = 0; j < info[i].choices.length; j++)
            System.out.println ("        " + info[i].choices[j]);
    }
}

```


3

Accessing JDBC Databases

This chapter provides examples of Java code using the Caché JDBC driver to query databases and work with the results.

- [A Simple JDBC Application](#) — a complete but very simple application that demonstrates the basic features of JDBC.
- [Using Statements](#) — an overview of JDBC SQL query classes.
- [Inserting and Updating Data](#) — using JDBC result sets to insert and update data in a Caché database.
- [Logging for JDBC Applications](#) — how to enable logging to test and troubleshoot your JDBC applications.

Note: In the examples given in this chapter, several methods throw exceptions of type `SQLException`. The required `try` `catch` blocks are omitted for clarity.

3.1 A Simple JDBC Application

This section describes a very simple JDBC application that demonstrates the use of some of the most common JDBC classes:

- A `CacheDataSource` object is used to create a `Connection` object that links the JDBC application to the Caché database.
- The `Connection` object is used to create a `PreparedStatement` object that can execute a dynamic SQL query.
- The `PreparedStatement` query returns a `ResultSet` object that contains the requested rows.
- The `ResultSet` object has methods that can be used to move to a specific row and read or update specified columns in the row.

All of these classes are discussed in more detail later in the chapter.

The TinyJDBC Application

To begin, import the JDBC packages and open a `try` block:

```
import java.sql.*;
import javax.sql.*;
import com.intersys.jdbc.*;

public class TinyJDBC{
    public static void main() {
        try {
```

Use `CacheDataSource` to open a connection (for details, see [Using CacheDataSource to Connect](#)):

```
Class.forName ("com.intersys.jdbc.CacheDriver").newInstance();
CacheDataSource ds = new CacheDataSource();
ds.setURL("jdbc:Cache://127.0.0.1:1972/SAMPLES");
Connection dbconn = ds.getConnection("_SYSTEM", "SYS");
```

Execute a query and get a scrollable, updatable result set.

```
String sql="Select Name from Sample.Person Order By Name";
int scroll=ResultSet.TYPE_SCROLL_SENSITIVE;
int update=ResultSet.CONCUR_UPDATABLE;

PreparedStatement pstmt = dbconn.prepareStatement(sql,scroll,update);
java.sql.ResultSet rs = pstmt.executeQuery();
```

Move to the first row of the result set and change the name.

```
rs.first();
System.out.println("\n Old name = " + rs.getString("Name"));
rs.updateString("Name", "Bill. Buffalo");
rs.updateRow();
System.out.println("\n New name = " + rs.getString("Name") + "\n");
```

Close objects and catch any exceptions.

```
pstmt.close();
rs.close();
dbconn.close();
} catch (Exception ex) {
System.out.println("TinyJDBC caught exception: "
+ ex.getClass().getName() + ": " + ex.getMessage());
}
} // end main()
} // end class TinyJDBC
```

Important: In the rest of this chapter, examples will be presented as fragments of code, rather than whole applications. These examples are intended to demonstrate some basic features as briefly and clearly as possible. It will be assumed that a connection has already been opened, and that all code fragments are within an appropriate try/catch statement. It is also assumed that the reader is aware of the standard good coding practices that are not illustrated here.

3.2 Using Statements

The `sql.java` package provides three classes used to query databases and return a `ResultSet`: `Statement`, `PreparedStatement`, and `CallableStatement`. All three classes are instantiated by calls to `Connection` methods. The following sections discuss how to use these classes:

- [Using Statement to Execute a SELECT](#) — a very simple statement call.
- [Executing a Prepared Statement](#) — a more detailed example.
- [Executing Stored Procedures with CallableStatement](#) — an example that executes the **ByName** stored procedure from `Sample.Person`.
- [Returning Multiple Result Sets](#) — accessing multiple result sets returned by Caché stored procedures.
- [Statement Pooling](#) — how the Caché driver stores and uses optimized statements.

3.2.1 Using Statement to Execute a SELECT

The following code executes an SQL **SELECT** on Caché using the `Statement` class:

- Create a query string and execute it using the `java.sql.Statement` `execute()` method:

```
String stQuery="SELECT ID, Name from Sample.Person";
java.sql.ResultSet rs = stmt.executeQuery(stQuery);
```

You should always use the fully qualified name `java.sql.ResultSet` to avoid clashes with `com.intersys.classes.ResultSet`

- Process and display the query results:

```
ResultSetMetaData rsmd = rs.getMetaData();
int colnum = rsmd.getColumnCount();
while (rs.next()) {
    for (int i=1; i<=colnum; i++)
        System.out.print(rs.getString(i) + " ");
}
```

3.2.2 Executing a Prepared Statement

The following query uses a prepared statement to return a list of all employees with names beginning in “A” through “E” who work for a company with a name starting in “M” through “Z”:

```
Select ID, Name, Company->Name from Sample.Employee
Where Name < ? and Company->Name > ?
Order By Company->Name
```

Note: This statement uses Implicit Join syntax (the `->` operator), which provides a simple way to access the Company class referenced by `Sample.Employee`.

The prepared statement is implemented just like a regular statement:

- Create the string containing the query and use it to initialize the `PreparedStatement` object, then set the values of the query parameters and execute the query:

```
String sql=
    "Select ID, Name, Company->Name from Sample.Employee " +
    "Where Name < ? and Company->Name > ? " +
    "Order By Company->Name";
PreparedStatement pstmt = dbconnection.prepareStatement(sql);

pstmt.setString(1,"F");
pstmt.setString(2,"L");
java.sql.ResultSet rs = pstmt.executeQuery();
```

- Retrieve and display the result set:

```
java.sql.ResultSet rs = pstmt.executeQuery();
ResultSetMetaData rsmd = rs.getMetaData();
int colnum = rsmd.getColumnCount();
while (rs.next()) {
    for (int i=1; i<=colnum; i++) {
        System.out.print(rs.getString(i) + " ");
    }
    System.out.println();
}
```

3.2.3 Using Callable Statements to Execute Stored Procedures

The following code executes **ByName**, a Caché stored procedure contained in `Sample.Person`:

- Create a `java.sql.CallableStatement` object and initialize it with the name of the stored procedure. The `SqlName` of the procedure is `SP_Sample_By_Name`, which is how it must be referred to in the Java client code:

```
String sql="call Sample.SP_Sample_By_Name(?)"
CallableStatement cs = dbconnection.prepareCall(sql);
```

- Set the value of the query parameter and execute the query, then iterate through the result set and display the data:

```

cs.setString(1,"A");
java.sql.ResultSet rs = cs.executeQuery();

ResultSetMetaData rsmd = rs.getMetaData();
int colnum = rsmd.getColumnCount();
while (rs.next()) {
    for (int i=1; i<=colnum; i++)
        System.out.print(rs.getString(i) + " ");
}
System.out.println();

```

3.2.4 Returning Multiple Result Sets

Caché allows you to define a stored procedure that returns multiple result sets. The Caché JDBC driver supports the execution of such stored procedures. Here is an example of a Caché stored procedure that returns two result sets (note that the two query results have different column structures):

```

// This class method produces two result sets.
ClassMethod DRS(st) [ ReturnResultsets, SqlProc ]
{
    $$$ResultSet("select Name from Sample.Person where Name %STARTSWITH :st")
    $$$ResultSet("select Name, DOB from Sample.Person where Name %STARTSWITH :st")
    Quit
}

```

Note: This stored procedure is not defined in `Sample.Person`. In order to try this example, you must first open `Sample.Person` in Caché Studio and add the class method shown above. You must also add the declaration `include %occResultSet` at the start of the `Sample.Person` file (before the class definition, as shown here):

```

include %occResultSet
Class Sample.Person Extends (%Persistent, %Populate, %XML.Adaptor) [ ClassType = persistent ]
{ ...

```

Remember to recompile `Sample.Person` after making these changes.

The following code executes the stored procedure and iterates through both of the returned result sets:

- Create the `java.sql.CallableStatement` object and initialize it using the name of the stored procedure. Set the query parameters and use **execute** to execute the query:

```

CallableStatement cs = dbconnection.prepareCall("call Sample.Person_DRS(?)");
cs.setString(1,"A");
boolean success=cs.execute();

```

- Iterate through the pair of result sets displaying the data. Note that **getMoreResults** moves to the Statement object's next result set while **getResultSet** retrieves the current result set.

```

if(success) do{
    java.sql.ResultSet rs = cs.getResultSet();
    ResultSetMetaData rsmd = rs.getMetaData();
    for (int j=1; j<rsmd.getColumnCount() + 1; j++)
        System.out.print(rsmd.getColumnName(j)+ "\t\t");
    System.out.println();
    int colnum = rsmd.getColumnCount();
    while (rs.next()) {
        for (int i=1; i<=colnum; i++)
            System.out.print(rs.getString(i) + " \t ");
        System.out.println();
    }
    System.out.println();
} while (cs.getMoreResults());

```

Note: By default **getMoreResults** closes the current result set before moving to the next. The Caché JDBC Driver does not support keeping the current result set open after moving to the next.

3.2.5 Statement Pooling

JDBC 4.0 adds an additional infrastructure, statement pooling, which stores optimized statements in a cache the first time they are used. Statement pools are maintained by connection pools, allowing pooled statements to be shared between connections. All the implementation details are completely transparent to the user, and it is up to the driver to provide the required functionality.

Caché JDBC implemented statement pooling long before the concept became part of the JDBC specification. While the Caché driver uses techniques similar to those recommended by the specification, the actual pooling implementation is highly optimized. Unlike most implementations, Caché JDBC has three different statement pooling caches. One roughly corresponds to statement pooling as defined by the JDBC specification, while the other two are Caché specific optimizations. See *The Query Cache in Using Caché SQL* for an explanation of Caché statement caching. As required, Caché JDBC statement pooling is completely transparent to the user.

The Caché JDBC supports the JDBC 4.0 Statement methods `setPoolable()` and `isPoolable()` as hints to whether the statement in question should be pooled. Caché uses its own heuristics to determine appropriate sizes for all three of its statement pools, and therefore does not support limiting the size of a statement pool by setting the `maxStatements` property in `ConnectionPoolDataSource`. The optional `javax.sql.StatementEventListener` interface is unsupported (and irrelevant) for the same reason.

3.3 Inserting and Updating Data

There are several ways to insert and update Caché data using JDBC:

- [Inserting Data and Retrieving Generated Keys](#) — using `PreparedStatement` and the SQL INSERT command.
- [Scrolling a Result Set](#) — randomly accessing any row of a result set.
- [Updating a Scrollable Result Set](#) — accessing and changing the data in result set rows.
- [Using Transactions](#) — using the JDBC transaction API to commit or roll back changes.

The `generateSSN()` Method

In this section, several examples insert new rows into `Sample.Person`, which requires SSN (Social Security Number) as a unique key. The following method is used in these examples to generate a random SSN:

```
public static String generateSSN() {
    java.util.Random random = new java.util.Random();
    StringBuffer sb = new StringBuffer();
    for (int i=1; i<=9; i++) sb.append(random.nextInt(10));
    sb.insert(5, '-');
    sb.insert(3, '-');
    return sb.toString();
}
```

3.3.1 Inserting Data and Retrieving Generated Keys

The following code inserts a new row into `Sample.Person` and retrieves the generated ID key.

- Create the `PreparedStatement` object, initialize it with the SQL string, and specify that generated keys are to be returned:

```
String sqlIn="INSERT INTO Sample.Person (Name,SSN,DOB) " + "VALUES(?,?,?)";
int keys=Statement.RETURN_GENERATED_KEYS;
PreparedStatement pstmt = dbconnection.prepareStatement(sqlIn, keys);
```

- Set the values for the query parameters and execute the update (see [The `generateSSN\(\)` Method](#) for an explanation of the `generateSSN()` call):

```
String SSN = generateSSN(); // generate a random SSN
java.sql.Date DOB = java.sql.Date.valueOf("1973-02-01");

pstmt.setString(1,"Smith,John"); // Name
pstmt.setString(2,SSN); // Social Security Number
pstmt.setDate(3,DOB); // Date of Birth
pstmt.executeUpdate();
```

- Each time you insert a new row, Caché automatically generates an object ID for the row. The generated ID key is retrieved into a result set and displayed along with the *SSN*:

```
java.sql.ResultSet rsKeys = pstmt.getGeneratedKeys();
rsKeys.next();
String newID=rsKeys.getString(1);
System.out.println("new ID for SSN " + SSN + " is " + newID);
```

Although this code assumes that the ID will be the first and only generated key in *rsKeys*, this is not always a safe assumption in real life.

- Retrieve the new row by ID and display it (*Age* is a calculated value based on *DOB*).

```
String sqlOut="SELECT ID,Name,Age,SSN FROM Sample.Person WHERE ID="+newID;
pstmt = dbconnection.prepareStatement(sqlOut);
java.sql.ResultSet rsPerson = pstmt.executeQuery();

int colnum = rsPerson.getMetaData().getColumnCount();
rsPerson.next();
for (int i=1; i<=colnum; i++)
    System.out.print(rsPerson.getString(i) + " ");
System.out.println();
```

3.3.2 Scrolling a Result Set

The Caché JDBC driver supports scrollable result sets, which allow your Java applications to move both forward and backward through the resultset data. The `prepareStatement()` method uses following parameters to determine how the result set will function:

- The *resultSetType* parameter determines how changes are displayed:
 - `ResultSet.TYPE_SCROLL_SENSITIVE` creates a scrollable result set that displays changes made to the underlying data by other processes.
 - `ResultSet.TYPE_SCROLL_INSENSITIVE` creates a scrollable result set that only displays changes made by the current process.
- The *resultSetConcurrency* parameter must be set to `ResultSet.CONCUR_UPDATABLE` if you intend to update the result set.

The following code creates and uses a scrollable result set:

- Create a `PreparedStatement` object, set the query parameters, and execute the query:

```
String sql="Select ID, Name, SSN from Sample.Person "+
    " Where Name > ? Order By Name";
int scroll=ResultSet.TYPE_SCROLL_INSENSITIVE;
int update=ResultSet.CONCUR_UPDATABLE;

PreparedStatement pstmt = dbconnection.prepareStatement(sql,scroll,update);
pstmt.setString(1,"S");
java.sql.ResultSet rs = pstmt.executeQuery();
```

- The application can scroll backwards as well as forwards through this result set. Use **afterLast** to move the result set's cursor to after the last row. Use **previous** to scroll backwards.

```
rs.afterLast();
int colnum = rs.getMetaData().getColumnCount();
while (rs.previous()) {
    for (int i=1; i<=colnum; i++)
        System.out.print(rs.getString(i) + " ");
    System.out.println();
}
```

- Move to a specific row using **absolute**. This code displays the contents of the third row:

```
rs.absolute(3);
for (int i=1; i<=colnum; i++)
    System.out.print(rs.getString(i) + " ");
System.out.println();
```

- Move to a specific row relative to the current row using **relative**. The following code moves to the first row, then scrolls down two rows to display the third row again:

```
rs.first();
rs.relative(2);
for (int i=1; i<=colnum; i++)
    System.out.print(rs.getString(i) + " ");
System.out.println();
```

3.3.3 Updating a Scrollable Result Set

The following code updates an open result set and saves the changes to the database:

- Create a `PreparedStatement` object, set the query parameters, and execute the query:

```
String sql="Select Name, SSN from Sample.Person "+
    " Where Name > ? Order By Name";
int scroll=ResultSet.TYPE_SCROLL_SENSITIVE;
int update=ResultSet.CONCUR_UPDATABLE;

PreparedStatement pstmt = dbconnection.prepareStatement(sql,scroll,update);
pstmt.setString(1,"S");
java.sql.ResultSet rs = pstmt.executeQuery();
```

A result set that is going to have new rows inserted should not include the Caché ID column. ID values are defined automatically by Caché.

- To update a row, move the cursor to that row and update the desired columns, then invoke **updateRow**:

```
rs.last();
rs.updateString("Name", "Avery. Tara R");
rs.updateRow();
```

- To insert a row, move the cursor to the “insert row” and then update that row's columns. Be sure that all non-nullable columns are updated (see [The generateSSN\(\) Method](#) for an explanation of the `generateSSN()` call). Finally, invoke **insertRow**:

```
rs.moveToInsertRow();
rs.updateString(1, "Abelson,Alan");
rs.updateString(2, generateSSN());
rs.insertRow();
```

3.3.4 Using Transactions

The Caché JDBC driver supports the JDBC transaction API.

- In order to group SQL statements into a transaction, you must first disable autocommit mode using **setAutoCommit()**:

```
dbconnection.setAutoCommit(false);
```

- Use **commit()** to commit to the database all SQL statements executed since the last execution of **commit()** or **rollback()**:

```
pstmt1.execute();
pstmt2.execute();
pstmt3.execute();
dbconnection.commit();
```

- Use **rollback()** to roll back all of the transactions in a transactions. Here the **rollback()** is invoked if **SQLException** is thrown by any SQL statement in the transaction:

```
catch(SQLException ex) {
    if (dbconnection != null) {
        try {
            dbconnection.rollback();
        } catch (SQLException excep){
            // (handle exception)
        }
    }
}
```

3.3.4.1 Transaction Handling Methods

Here is a brief summary of the `java.sql.Connection` methods used for transaction handling:

setAutoCommit()

By default `Connection` objects are in `autocommit` mode. In this mode an SQL statement is committed as soon as it is executed. To group multiple SQL statements into a transaction, first use `setAutoCommit(false)` to take the `Connection` object out of `autocommit` mode. Use `setAutoCommit(true)` to reset the `Connection` object to `autocommit` mode.

commit()

Executing **commit()** commits all SQL statements executed since the last execution of either **commit()** or **rollback()**.

rollback()

Executing **rollback** aborts a transaction and restores any values changed by the transaction back to their original state.

setTransactionIsolation()

Sets the isolation level for a transaction. Caché supports the following JDBC transaction isolation levels:

- `Connection.TRANSACTION_READ_UNCOMMITTED` — Level 1. Permits dirty reads, non-repeatable reads, and phantom reads.
- `Connection.TRANSACTION_READ_COMMITTED` — Level 2. Prevents dirty reads, but allows non-repeatable and phantom reads.

getIsolationLevel()

Returns the current transaction isolation level for the `Connection` object.

3.4 Logging for JDBC Applications

If your applications encounter any problems, you can monitor by enabling the appropriate logging:

- [Enable logging for Caché](#)
- [Enabling Logging for JDBC](#)

- [Enable logging for the JDBC gateway](#)

Run your application, ensuring that you trigger the error condition, then check all the logs for error messages or any other unusual activity. The cause of the error is often obvious.

When using the JDBC Gateway, you should be able to find out more about logging by consulting the documentation for the remote database to which you are connecting.

CAUTION: Enable logging only when you need to perform troubleshooting. You should not enable logging during normal operation, because it will dramatically slow down performance.

3.4.1 Enabling Logging for Caché

To enable logging for Caché, enter a command similar to the following at the Caché Terminal:

```
Set ^%ISCLOG = 2
```

The ^%ISCLOG global logs events in Caché for use in debugging. The full list of log levels is as follows:

0	Caché performs no logging.
1	Caché logs only exceptional events (such as error messages).
2	Caché logs detailed information. For example: 'method ABC invoked with parameters X,Y,Z and returned 1234'.
3	Caché logs raw information such as data received from an HTTP request.

You can turn Caché logging off with either

```
Set ^%ISCLOG = 0
```

or

```
Kill ^%ISCLOG
```

3.4.2 Enabling Logging for JDBC

To enable logging for JDBC when connecting to Caché, add a log file name to the end of your JDBC connection string. When you connect, the driver will save a log file that will be saved to the working directory of the application.

For example, suppose your original connection string is as follows:

```
jdbc:Cache://127.0.0.1:1972/USER
```

To enable logging, change this to the following and then reconnect:

```
jdbc:Cache://127.0.0.1:1972/USER/myjdbc.log
```

This log records the interaction from the perspective of the Caché database.

See [Defining a JDBC Connection URL](#) for a complete list of connection parameters.

3.4.3 Enabling Logging for the SQL Gateway

The [Caché SQL Gateway](#) can also generate a log when used with JDBC. To enable this logging:

- In the System Management Portal, go to **[Home] > [Configuration] > [Object/SQL Gateway Settings] > [JDBC Gateway Settings]**.
- Specify a value for **JDBC Gateway Log**. This should be the name of a log file (for example, jdbcGateway.log) that will record the interaction between the gateway and the database.

4

Using the Caché SQL Gateway with JDBC

The Caché SQL Gateway allows Caché to access external databases via both JDBC and ODBC. For a detailed description of the SQL Gateway, see the chapter on Using the Caché SQL Gateway in *Using Caché SQL*.

This chapter discusses the following topics:

- [Creating JDBC SQL Gateway Connections for External Sources](#) — describes how to create an ODBC logical connection definition for the SQL Gateway.
- [Using the JDBC SQL Gateway Programmatically](#) — briefly discusses how to access a JDBC compliant database programmatically. This option provides more control over the connection than the setup provided by the standard SQL Gateway wizards.

4.1 Creating JDBC SQL Gateway Connections for External Sources

Caché maintains a list of SQL Gateway connection definitions, which are logical names for connections to external data sources. Each connection definition consists of a logical name (for use within Caché), information on connecting to the data source, and a user name and password to use when establishing the connection. These connections are stored in the table %Library.sys_SQLConnection. You can export data from this table and import it into another Caché instance.

Note: If you need an option that is not supported by these wizards, you can instead connect programmatically. See the section on [Using the JDBC Gateway Programmatically](#).

To monitor SQL Gateway problems, you can enable SQL Gateway logging. See the section on [Enabling Logging for the SQL Gateway](#).

4.1.1 Creating a JDBC Gateway Connection

To define a gateway connection for a JDBC-compliant data source, perform the following steps:

1. In the System Management Portal, go to the **[Home] > [Configuration] > [Object/SQL Gateway Settings] > [SQL Gateway Connections]** page.
2. Click **Create New Connection**.
3. On the **Gateway Connection** page, enter or choose values for the following fields:
 - For **Type**, choose **JDBC**.

- **Connection Name** — Specify an identifier for the connection, for use within Caché.
- **User** — Specify the name for the account to serve as the default for establishing connections, if needed.
- **Password** — Specify the password associated with the default account.
- **Driver name** — Full class name of the JDBC client driver.
- **URL** — Connection URL for the data source, in the format required by the JDBC client driver that you are using.
- **Class path** — Specifies a comma-separated list of additional JAR files to load.
- **Properties** — Optional string that specifies vendor-specific connection properties. If specified, this string should be of the following form:

property=value;property=value;...

See [Caché JDBC Connection Properties](#) for more information on connection properties.

For example, a typical connection might use the following values:

Setting	Value
Type	JDBC
Connection Name	ConnectionJDBC1
User	JDBCUser
Password	JDBCPassword
Driver name	oracle.jdbc.driver.OracleDriver
URL	jdbc:oracle:thin:@//oraserver:1521/SID
Class path	/fill/path/to/ojdbc14.jar
Properties	oracle.jdbc.V8Compatibility=true; includeSynonyms=false;restrictGetTables=true

For the other options, see “[Implementation-specific Options.](#)”

4. Optionally test if the values are valid. To do so, click the **Test Connection** button. The screen will display a message indicating whether the values you have entered allow for a valid connection.
5. To create the named connection, click **Save**.
6. Click **Close**.

4.1.2 Creating a JDBC Connection to Caché via the SQL Gateway

Caché provides JDBC drivers and can be used as a JDBC data source. That is, a Caché instance can connect to itself or to another Caché instance via JDBC and the SQL Gateway. Specifically, the connection is from a namespace in one Caché to a namespace in the other Caché. To connect in this way, you need the same information that you need for any other external database: the connection details for the database driver that you want to use. This section provides the basic information.

4.1.2.1 Connecting to Caché as a JDBC Data Source

To configure a Caché instance (Caché 1) to use another Caché instance (Caché 2) as a JDBC data source, do the following:

1. Within Caché 1, use the SQL Gateway to create a JDBC connection to the namespace in Caché 2 that you want to use.

- For **Type**, choose **JDBC**.
- **Connection Name** — Specify an identifier for the connection, for use within Caché 1.
- **User** — Specify the user name needed to access Caché 2, if needed.
- **Password** — Specify the password for this user.
- **Driver name** — Use `com.intersys.jdbc.CacheDriver`
- **URL** — Connection URL for the data source, in the following format:

```
jdbc:Cache://IP_address:port/namespace
```

Here *IP_address:port* is the IP address and port where Caché 2 is running, and *namespace* is the namespace to which you want to connect.

For example, a typical connection might use the following values:

Setting	Value
Type	JDBC
Connection Name	Cache2Samples
User	__SYSTEM
Password	SYS
Driver name	<code>com.intersys.jdbc.CacheDriver</code>
URL	<code>jdbc:Cache://127.0.0.1:1972/SAMPLES</code>

- **Class path** — Leave this blank.
- **Properties** — Optional string that specifies connection properties supported by the Caché JDBC drivers. If specified, this string should be of the following form:
property=value;property=value;...
- **Conversion in composite Row IDs** — The default option should be suitable in most cases, but you can choose any option. The details are given in “[Implementation-specific Options](#).”
- **Do not use delimited identifiers by default** — The default (not to click this check box) should be suitable in most cases, but you can enable this option if desired. The details are given in “[Implementation-specific Options](#).”

2. Click **Save**.
3. Click **Close**.

4.1.3 Implementation-specific Options

Before you define an SQL gateway connection, you should make sure that you understand the requirements of the external database and of the database driver, because these requirements affect how you define the connection.

Do Not Use Delimited Identifiers by Default

The **Do not use delimited identifiers by default** option controls the format of identifiers in the generated routines:

- If this option is selected, all SQL identifiers are created as ordinary identifiers.
- If this option is cleared, all SQL identifiers are created as delimited identifiers.

How you set this depends on whether the external database supports delimited identifiers.

- Select this check box if you are using Sybase or Informix.
- Clear this check box for other databases.

Use COALESCE

The **Use COALESCE** option controls how a query is handled when it includes a parameter (?), and it has an effect only when a query parameter equals null.

- If you do not select **Use COALESCE** and if a query parameter equals null, the query returns only records that have null for the corresponding value. For example, consider a query of the following form:

```
SELECT ID, Name from LinkedTables.Table WHERE Name %STARTSWITH ?
```

If the provided parameter is null, the query would return only rows with null-valued names.

- If you select **Use COALESCE**, the query wraps each parameter within a COALESCE function call, which controls how null values are handled.

Then, if a query parameter equals null, the query essentially treats the parameter as a wildcard. In the previous example, if the provided parameter is null, this query returns all rows, which is consistent with the behavior of typical ODBC clients.

Whether you select this option depends on your preferences and on whether the external database supports the COALESCE function.

To find out whether the external database supports the COALESCE function, consult the documentation for that database.

Conversion in Composite Row IDs

The **Conversion in composite Row IDs** option controls how non-character values are treated when forming a composite ID. Choose an option that is supported by your database:

- **Do not convert non-character values** — This option performs no conversion. This option is suitable only if your database supports concatenating non-character values to character values.
- **Use CAST** — This option uses CAST to convert non-character values to character values.
- **Use {fn convert ...}** — This option uses {fn convert ...} to convert non-character values to character values.

In all cases, the IDs are concatenated with | | between the IDs (or transformed IDs).

Consult the documentation for the external database to find out which option or options it supports.

4.2 Using the JDBC SQL Gateway Programmatically

The following two Caché classes allow you to access the SQL Gateway programmatically:

- %Library.SQLConnection — allows you to access Caché [SQL Gateway connections](#).
- %Net.Remote.Java.JDBCGateway — allows you to connect to and use a JDBC-compliant database. The following methods are provided, each of which corresponds to the JDBC method of the same name:

Table 4–1: Calling JDBC Methods from %Net.Remote.Java.JDBCGateway

cgetString	getColumnType	prepareStatement
closeAll	getColumns	registerOutParameter
cnext	getErrorText	removeConnection
columnCount	getPrecision	removeResultSet
columnTypeName	getPrimaryKeys	removeStatement
commit	getProcedureColumns	rollback
connect	getProcedures	setAutoCommit
connectWithPropString	getScale	setString
execQuery	getString	setValue
execUpdate	getTables	wasError
getColumnCount	next	
getColumnName	prepareCall	

5

Caché JDBC Compliance

The Caché JDBC Driver is a high-performance *type 4* JDBC database driver (pure Java, with no Caché-specific binary code and no JDBC-ODBC bridge). It is fully compliant with the JDBC 4.0 API specification, supporting all required interfaces and adhering to all JDBC 4.0 guidelines and requirements. Of the new functionality introduced in JDBC 4.0, Caché supports all features except SQL Exception handling enhancements, National Character Set conversions, and the XML data type.

This chapter lists all members of the JDBC 4.0 API, indicates which optional features have been omitted from the Caché driver, and describes all Caché-specific features.

The following topics are covered:

- [JDBC 4.0 Interfaces and Classes](#) — gives a brief summary of the major JDBC 4.0 implementation requirements.
- [The JDBC Core API \(java.sql\)](#) — describes the level of Caché driver support for each element in the java.sql package.
- [The JDBC Optional Package API \(javax.sql\)](#) — describes the level of Caché driver support for each element in the javax.sql package.
- [Interfaces with Unsupported Optional Methods](#) — provides a detailed list of optional methods not supported by the Caché driver.
- [Caché JDBC Additions and Extensions](#) — describes additional Caché-specific methods available in some interfaces.

5.1 JDBC 4.0 Interfaces and Classes

This section gives a brief summary of the major JDBC 4.0 implementation requirements. Classes and exceptions not listed here are already fully implemented in the JDBC 4.0 API.

Required Interfaces

The following interfaces must be fully implemented:

java.sql.Driver	java.sql.ParameterMetaData	java.sql Wrapper
java.sql.DatabaseMetaData	java.sql.ResultSetMetaData	javax.sql.DataSource

See [DatabaseMetaData Variant Methods](#) and [CacheDataSource](#) for related information.

Required Interfaces with Optional Methods

The following interfaces must be implemented, but some methods in them are optional if the implementation of those methods depends on a feature that the DBMS does not support:

java.sql.CallableStatement	java.sql.PreparedStatement	java.sql.Statement
java.sql.Connection	java.sql.ResultSet	

See [Interfaces with Unsupported Optional Methods](#) for details. Also see [CallableStatement Additional Method](#).

Optional Interfaces

The following interfaces are optional. Interfaces in *italics* are not supported by the Caché JDBC driver:

<i>java.sql.Array</i>	java.sql.Savepoint	<i>javax.sql.CommonDataSource</i>
java.sql.Blob	<i>java.sql.SQLData</i>	javax.sql.ConnectionEventListener
java.sql.Clob	<i>java.sql.SQLInput</i>	javax.sql.ConnectionPoolDataSource
<i>java.sql.NClob</i>	<i>java.sql.SQLOutput</i>	javax.sql.PooledConnection
java.sql.RowId	<i>java.sql.Struct</i>	<i>javax.sql.StatementEventListener</i>
<i>java.sql.Ref</i>		<i>javax.sql.XAConnection</i>
		<i>javax.sql.XADataSource</i>

For additional information, see [ConnectionPoolDataSource](#).

5.2 The JDBC Core API (java.sql)

This section describes Caché JDBC driver support for each element in the java.sql package.

5.2.1 java.sql Interfaces

The Caché JDBC driver provides the following levels of support for java.sql interfaces:

- **Array** — not supported
- **Blob** — all methods fully supported
- **CallableStatement** — some [optional methods not supported](#); has some Caché-specific [additional methods](#)
- **Clob** — all methods fully supported
- **Connection** — some [optional methods not supported](#)
- **DatabaseMetaData** — all methods supported, some with [restrictions](#)
- **Driver** — all methods fully supported
- **NClob** — not supported
- **ParameterMetaData** — all methods fully supported
- **PreparedStatement** — some [optional methods not supported](#)
- **Ref** — not supported

- **ResultSet** — some [optional methods not supported](#)
- **ResultSetMetaData** — all methods fully supported
- **RowId** — all methods fully supported (added for JDBC 4.0)
- **Savepoint** — all methods fully supported
- **SQLData** — not supported
- **SQLInput** — not supported
- **SQLOutput** — not supported
- **SQLXML** — not supported
- **Statement** — some [optional methods not supported or restricted](#)
- **Struct** — not supported
- **Wrapper** — all methods fully supported

5.2.2 java.sql Classes

The following java.sql classes are already fully implemented in the JDBC 4.0 API:

ClientInfoStatus	DriverPropertyInfo	Time
Date	RowIdLifeTime	Timestamp
DriverManager	SQLPermission	Types

5.2.3 java.sql Exceptions

The Caché JDBC driver throws only the following exceptions:

- **BatchUpdateException**
- **SQLException**
- **SQLWarning**

The following exceptions are listed here for completeness, but are not required and are never used:

DataTruncation	SQLNonTransientException
SQLClientInfoException	SQLRecoverableException
SQLDataException	SQLSyntaxErrorException
SQLFeatureNotSupportedException	SQLTimeoutException
SQLIntegrityConstraintViolationException	SQLTransactionRollbackException
SQLInvalidAuthorizationSpecException	SQLTransientConnectionException
SQLNonTransientConnectionException	SQLTransientException

5.3 The JDBC Optional Package API (javax.sql)

This section describes Caché JDBC driver support for each element in the javax.sql package.

5.3.1 javax.sql Interfaces

The Caché JDBC driver provides the following levels of support for javax.sql interfaces:

- **CommonDataSource** — not implemented (use **DataSource**)
- **ConnectionEventListener** — all methods fully supported
- **ConnectionPoolDataSource** — all methods fully supported, with [additional methods implemented in CacheConnectionPoolDataSource](#)
- **DataSource** — all methods fully supported, with [additional methods implemented in CacheDataSource](#)
- **PooledConnection** — all methods fully supported
- **Rowset** — not supported
- **RowSetInternal** — not supported
- **RowSetListener** — not supported
- **RowSetMetaData** — not supported
- **RowSetReader** — not supported
- **RowSetWriter** — not supported
- **StatementEventListener** — not supported
- **XAConnection** — not supported
- **XADataSource** — not supported

5.3.2 javax.sql Classes

The following javax.sql classes are already fully implemented in the JDBC 4.0 API:

ConnectionEvent	RowSetEvent	StatementEvent
------------------------	--------------------	-----------------------

5.4 Interfaces with Unsupported Optional Methods

The following interfaces have optional methods that the Caché JDBC driver does not support, or methods implemented in a non-standard manner:

- [CallableStatement](#) — **Unsupported Methods**
- [Connection](#) — **Unsupported or Restricted Methods**
- [DatabaseMetaData](#) — **Variant Methods**
- [PreparedStatement](#) — **Unsupported Methods**
- [ResultSet](#) — **Unsupported Methods**

- [Statement — Unsupported or Restricted Methods](#)

5.4.1 CallableStatement — Unsupported Methods

java.sql.CallableStatement does not support the following optional methods:

- **getArray()**

```
Array getArray(int i)
Array getArray(String parameterName)
```

- **getNCharacterStream() and setNCharacterStream()**

```
Reader getNCharacterStream(int parameterIndex)
Reader getNCharacterStream(String parameterName)

void setNCharacterStream(String parameterName, Reader value)
void setNCharacterStream(String parameterName, Reader value, long length)
```

- **getNClob() and setNClob()**

```
java.sql.NClob getNClob(int parameterIndex)
java.sql.NClob getNClob(String parameterName)

void setNClob(String parameterName, Reader reader)
void setNClob(String parameterName, Reader reader, long length)
void setNClob(String parameterName, java.sql.NClob value)
```

- **getNString() and setNString()**

```
String getNString(int parameterIndex)
String getNString(String parameterName)

void setNString(String parameterName, String value)
```

- **getObject()**

```
Object getObject(int i, java.util.Map map)
Object getObject(String parameterName, java.util.Map map)
```

- **getRef()**

```
Ref getRef(int i)
Ref getRef(String parameterName)
```

- **getRowId() and setRowId()**

```
java.sql.RowId getRowId(int i)
java.sql.RowId getRowId(String parameterName)

void setRowId(String parameterName, java.sql.RowId x)
```

- **getURL() and setURL()**

```
java.net.URL getURL(int i)
java.net.URL getURL(String parameterName)

void setURL(String parameterName, java.net.URL val)
```

- **getSQLXML() and setSQLXML()**

```
java.sql.SQLXML getSQLXML(int parameterIndex)
java.sql.SQLXML getSQLXML(String parameterName)

void setSQLXML(String parameterName, java.sql.SQLXML xmlObject)
```

5.4.2 Connection — Unsupported or Restricted Methods

java.sql.Connection does not support the following optional methods:

- **createArrayOf()**

```
java.sql.Array createArrayOf(String typeName, Object[] elements)
```

- **createBlob()**

```
Blob createBlob()
```

- **createClob()**

```
Clob createClob()
```

- **createNClob()**

```
java.sql.NClob createNClob()
```

- **createSQLXML()**

```
java.sql.SQLXML createSQLXML()
```

- **createStruct()**

```
java.sql.Struct createStruct(String typeName, Object[] attributes)
```

- **getTypeMap()** and **setTypeMap()**

```
java.util.Map getTypeMap()
```

```
void setTypeMap(java.util.Map map)
```

Optional Connection Methods with Restrictions

The following optional java.sql.Connection methods are implemented with restrictions or limitations:

- **prepareCall()**

Only TYPE_FORWARD_ONLY is supported for *resultSetType*. Only CONCUR_READ_ONLY is supported for *resultSetConcurrency*.

```
java.sql.CallableStatement prepareCall(String sql, int resultSetType, int resultSetConcurrency)
```

- **setReadOnly()**

A no-op (the Caché driver does not support READ_ONLY mode)

```
void setReadOnly(Boolean readOnly)
```

- **setCatalog()**

A no-op (the Caché driver does not support catalogs)

```
void setCatalog(String catalog)
```

- **setTransactionIsolation()**

Only TRANSACTION_READ_COMMITTED and TRANSACTION_READ_UNCOMMITTED are supported for *level*.

```
void setTransactionIsolation(int level)
```

The following five java.sql.Connection methods do not support CLOSE_CURSORS_AT_COMMIT for *resultSetHoldability*:

- **createStatement()**

```
java.sql.Statement createStatement(int resultSetType, int result, int resultSetHoldability)
```

- **getHoldability()** and **setHoldability()**

```
int getHoldability()
void setHoldability(int holdability)
```

- **prepareCall()**

```
java.sql.CallableStatement prepareCall(String sql, int resultSetType, int resultSetConcurrency,
int resultSetHoldability)
```

- **prepareStatement()**

```
java.sql.PreparedStatement prepareStatement(String sql, int resultSetType, int resultSetConcurrency,
int resultSetHoldability)
```

Caché currently supports only zero or one Auto Generated Keys. An exception is thrown if the `java.sql.Connection` methods below provide `columnIndexes` or `columnNames` arrays whose lengths are not equal to one.

- **prepareStatement()**

```
java.sql.PreparedStatement prepareStatement(String sql, int[] columnIndexes)
java.sql.PreparedStatement prepareStatement(String sql, String[] columnNames)
```

5.4.3 DatabaseMetaData — Variant Methods

`java.sql.DatabaseMetaData` is fully supported, but has methods that vary from the JDBC standard due to Caché-specific handling of their return values. The following methods are affected:

- **supportsMixedCaseQuotedIdentifiers()**

Caché returns `false`, which is not JDBC compliant.

```
boolean supportsMixedCaseQuotedIdentifiers()
```

- **getIdentifierQuoteString()**

If delimited id support is turned on, Caché returns `"` (double quote character), which is what a JDBC compliant driver should return; otherwise Caché returns a space.

```
String getIdentifierQuoteString()
```

5.4.4 PreparedStatement — Unsupported Methods

`java.sql.PreparedStatement` does not support the following optional methods:

- **setArray()**

```
void setArray(int i, Array x)
```

- **setNCharacterStream()**

```
void setNCharacterStream(int parameterIndex, Reader value)
void setNCharacterStream(int parameterIndex, Reader value, long length)
```

- **setNClob()**

```
void setNClob(int parameterIndex, Reader reader, long length)
void setNClob(int parameterIndex, Reader reader)
void setNClob(int parameterIndex, NClob value)
```

- **setNString()**

```
void setNString(int parameterIndex, String value)
```

- **setRef()**

```
void setRef(int i, Ref x)
```

- **setRowId()**

```
void setRowId(int parameterIndex, RowId x)
```

- **setSQLXML()**

```
void setSQLXML(int parameterIndex, SQLXML xmlObject)
```

- **setUnicodeStream()**

Deprecated in Java JDK specification.

```
void setUnicodeStream(int i, InputStream x, int length)
```

- **setURL()**

```
void setURL(int i, java.net.URL x)
```

5.4.5 ResultSet — Unsupported Methods

Caché does not support TYPE_SCROLL_SENSITIVE result set types.

java.sql.ResultSet does not support the following optional methods:

- **getArray()**

```
Array getArray(int i)
Array getArray(String colName)
```

- **getCursorName()**

```
String getCursorName()
```

- **getObject()**

```
Object getObject(int i, java.util.Map map)
Object getObject(String colName, java.util.Map map)
```

- **getRef()**

```
Ref getRef(int i)
Ref getRef(String colName)
```

- **getHoldability()**

```
int getHoldability()
```

- **getNString()**

```
String getNString(int columnIndex)
String getNString(String columnLabel)
```

- **getNCharacterStream()**

```
Reader getNCharacterStream(int columnIndex)
Reader getNCharacterStream(String columnName)
```

- **getUnicodeStream()**

Deprecated in Java JDK specification.

```
java.io.InputStream getUnicodeStream(int i)
java.io.InputStream getUnicodeStream(String colName)
```

- **getURL()**

```
java.net.URL getURL(int i)
java.net.URL getURL(String colName)
```

- **updateArray()**

```
void updateArray(int i, Array x)
void updateArray(String colName, Array x)
```

- **updateNString()**

```
void updateNString(int columnIndex, String nString)
void updateNString(String columnName, String nString)
```

- **updateNCharacterStream()**

```
void updateNCharacterStream(int columnIndex, Reader x)
void updateNCharacterStream(int columnIndex, Reader x, long length)
void updateNCharacterStream(String columnName, Reader reader)
void updateNCharacterStream(String columnName, Reader reader, long length)
```

- **updateNClob()**

```
void updateNClob(int columnIndex, Reader reader)
void updateNClob(int columnIndex, Reader reader, long length)
void updateNClob(String columnName, Reader reader)
void updateNClob(String columnName, Reader reader, long length)
```

- **updateRef()**

```
void updateRef(int i, Ref x)
void updateRef(String columnName, Ref x)
```

5.4.6 Statement — Unsupported or Restricted Methods

java.sql.Statement does not support the following optional method:

- **cancel()**

```
void cancel()
```

Optional Statement Methods with Restrictions

The following optional java.sql.Statement methods are implemented with restrictions or limitations:

- **getResultSetHoldability()**

Only HOLD_CURSORS_OVER_COMMIT

```
int getResultSetHoldability()
```

- **setCursorName()**

A no-op.

```
void setCursorName(String name)
```

- **setEscapeProcessing()**

A no-op (does not apply)

```
void setEscapeProcessing(Boolean enable)
```

Caché currently supports only zero or one auto-generated key. An exception is thrown if the `java.sql.Statement` methods below provide `columnIndexes` or `columnNames` arrays whose lengths are not equal to one:

- **execute()**

```
boolean execute(String sql, int[] columnIndexes)
boolean execute(String sql, String[] columnNames)
```

- **executeUpdate()**

```
int executeUpdate(String sql, int[] columnIndexes)
int executeUpdate(String sql, String[] columnNames)
```

5.5 Caché JDBC Additions and Extensions

The following interfaces have additional Caché-specific methods:

- **CallableStatement** — has an additional method to get binary streams.
- **CacheConnectionPoolDataSource** — is the Caché-specific implementation of the `javax.sql.ConnectionPoolDataSource` interface.
- **CacheDataSource** — is the Caché-specific implementation of `javax.sql.DataSource`.

5.5.1 CallableStatement Additional Method

`java.sql.CallableStatement` has the following additional Caché-only method:

- **getBinaryStream()**

Retrieves the value of the designated parameter (where *i* is the index of the parameter) as a `java.io.InputStream` object.

```
java.io.InputStream getBinaryStream(int i)
```

5.5.2 CacheConnectionPoolDataSource

The `com.intersys.jdbc.CacheConnectionPoolDataSource` class fully implements the `javax.sql.ConnectionPoolDataSource` interface.

Required Methods

- **getPooledConnection()**

```
javax.sql.PooledConnection getPooledConnection()
javax.sql.PooledConnection getPooledConnection(String usr, String pwd)
```

CAUTION: Calling applications should never use the **getPooledConnection()** methods or the `PooledConnection` class. Caché driver connections must always be obtained by calling the **getConnection()** method (which is inherited from **CacheDataSource**). The Caché driver provides pooling transparently through the `java.sql.Connection` object that it returns.

Additional Caché-only Methods

`CacheConnectionPoolDataSource` also supports the following additional Caché-only management methods:

- **restartConnectionPool()**

Restarts a connection pool. Closes all physical connections, and empties the connection pool.

```
void restartConnectionPool()
```

- **getPoolCount()**

Returns the current number of entries in the connection pool.

```
int getPoolCount()
```

- **setMaxPoolSize()**

Sets a maximum connection pool size. If the maximum size is not set, it defaults to 40.

```
void setMaxPoolSize(int max)
```

- **getMaxPoolSize()**

Returns the current maximum connection pool size

```
int getMaxPoolSize()
```

`CacheConnectionPoolDataSource` inherits from **CacheDataSource**, which provides additional Caché-specific methods.

5.5.3 CacheDataSource

The `com.intersys.jdbc.CacheDataSource` class fully implements the `javax.sql.DataSource` interface.

Required Methods

- **getConnection()**

```
java.sql.Connection getConnection()
java.sql.Connection getConnection(String usr,String pwd)
```

Additional Caché-only Methods

In addition to the methods defined by the interface, this class also includes a number of Cache specific methods that can be used to get or set `DataSource` properties.

In order to be able to connect, at least server name and database name properties must be defined, either by using the corresponding setters, or by supplying a valid URL (the same as what would be used when connecting via the `Driver` class). Port number is optional, and defaults to 1972. Username and Password can be set with the optional methods, or can be supplied via the **getConnection()** method.

The following Caché-only management methods are available:

- **getConnectionSecurityLevel()**

Returns an `int` representing the current Connection Security Level setting.

```
int getConnectionSecurityLevel()
```

- **getDatabaseName()**
Returns a String representing the current database (Caché namespace) name.
`String getDatabaseName()`
- **getDataSourceName()**
Returns a String representing the current data source name.
`String getDataSourceName()`
- **getDefaultTransactionIsolation()**
Gets the current default transaction isolation.
`int getDefaultTransactionIsolation()`
- **getDescription()**
Returns a String representing the current description.
`String getDescription()`
- **getEventClass()**
Returns a String representing an Event Class object.
`String getEventClass()`
- **getKeyRecoveryPassword()**
Returns a String representing the current Key Recovery Password setting.
`getKeyRecoveryPassword()`
- **getNodeDelay()**
Returns a boolean representing a current TCP_NODELAY option setting.
`boolean getNodeDelay()`
- **getPassword()**
Returns a String representing the current password.
`String getPassword()`
- **getPortNumber()**
Returns an *int* representing the current port number.
`int getPortNumber()`
- **getServerName()**
Returns a String representing the current server name.
`String getServerName()`
- **getServicePrincipalName()**
Returns a String representing the current Service Principal Name setting.
`String getServicePrincipalName()`
- **getSSLConfigurationName()**

Returns a String representing the current SSL Configuration Name setting.

```
getSSLConfigurationName()
```

- **getURL()**

Returns a String representing a current URL for this CacheDataSource object.

```
String getURL()
```

- **getUser()**

Returns a String representing the current username.

```
String getUser()
```

- **setConnectionSecurityLevel()**

Sets the connection security level

Sets the Connection Security Level for this DataSource object.

- **setDatabaseName()**

Sets the database name (Caché namespace) for this CacheDataSource object.

```
void setDatabaseName(String dn)
```

- **setDataSourceName()**

Sets the data source name for this CacheDataSource object. DataSourceName is an optional setting and is not used by CacheDataSource to connect.

```
void setDataSourceName(String dsn)
```

- **setDefaultTransactionIsolation()**

Sets the default transaction isolation level.

```
void setDefaultTransactionIsolation(int level)
```

- **setDescription()**

Sets the description for this CacheDataSource object. Description is an optional setting and is not used by CacheDataSource to connect.

```
void setDescription(String d)
```

- **setEventClass()**

Sets the Event Class for this CacheDataSource object. The Event Class is a mechanism specific to Cache JDBC. It is completely optional, and the vast majority of applications will not need this feature.

The Caché JDBC server will dispatch to methods implemented in a class when a transaction is about to be committed and when a transaction is about to be rolled back. The class in which these methods are implemented is referred to as the “event class.” If an event class is specified during login, then the JDBC server will dispatch to **%OnTranCommit** just prior to committing the current transaction and will dispatch to **%OnTranRollback** just prior to rolling back (aborting) the current transaction. User event classes should extend **%ServerEvent**. The methods do not return any values and cannot abort the current transaction.

```
void setEventClass(String e)
```

- **setKeyRecoveryPassword()**

Sets the Key Recovery Password for this CacheDataSource object.

```
setKeyRecoveryPassword(java.lang.String password)
```

- **setLogFile()**

Unconditionally sets the log file name for this CacheDataSource object.

```
setLogFile(java.lang.String logFile)
```

- **setNodelay()**

Sets the TCP_NODELAY option for this CacheDataSource object. Toggling this flag can affect the performance of the application. If not set, it defaults to true.

```
void setNodelay(boolean nd)
```

- **setPassword()**

Sets the password for this CacheDataSource object.

```
void setPassword(String p)
```

- **setPortNumber()**

Sets the port number for this CacheDataSource object

```
void setPortNumber(int pn)
```

- **setServerName()**

Sets the server name for this CacheDataSource object.

```
void setServerName(String sn)
```

- **setServicePrincipalName()**

Sets the Service Principal Name for this CacheDataSource object.

```
void setServicePrincipalName(String name)
```

- **setSSLConfigurationName()**

Sets the SSL Configuration Name for this CacheDataSource object.

```
setSSLConfigurationName(java.lang.String name)
```

- **setURL()**

Sets the URL for this CacheDataSource object.

```
void setURL(String u)
```

- **setUser()**

Sets the username for this CacheDataSource object.

```
void setUser(String u)
```