UnityJDBC User Documentation

UnityJDBC User Documentation

Table of Contents

1. General Information 1
    1. Overview of UnityJDBC 1
    2. History of UnityJDBC 1
    3. Further Information and Support 1
2. Installation 2
    1. General Issues 2
        1.1. System Requirements 2
    2. Quick Setup and Installation 2
    3. Installation on Windows 3
    4. Installation on Linux/UNIX 3
    5. Installation on Other Platforms 4
    6. Configuring Data Sources 4
    7. Testing the Installation 4
3. Tutorial 5
    1. Using the Sample Programs 5
        1.1. Using ExampleQuery.java 5
        1.2. Using ExampleUpdate.java 7
        1.3. Using ExampleMetadata.java 8
        1.4. Using ExampleMerge.java 8
        1.5. Using ExampleEngine.java 10
    2. Configuring Your Data Sources 10
    3. Writing Your Own Java Programs 12
4. Supported SQL Syntax 14
    1. Overview 14
    2. Data Types 14
    3. Identifiers 14
    4. Functions and Operators 14
        4.1. Logical Operators 14
        4.2. Comparison Operators 14
        4.3. Arithmetic Functions and Operators 15
        4.4. String Functions 15
        4.5. Pattern Matching Operators 16
        4.6. Data Type Conversion Functions 16
        4.7. Date/Time Functions and Operators 16
        4.8. Aggregate Functions 17
        4.9. User-Defined Functions and Support for Other Functions 17
        4.10. Function Translation 18
        4.11. Non-parsed Functions 18
    5. SELECT Statement 19
    6. INSERT Statement 20
    7. UPDATE Statement 20
    8. DELETE Statement 20
    9. By-Pass Statement 21
5. Supported JDBC Methods 22
    1. Overview 22
6. Driver Specific Capabilities 23
    1. MERGE Operator 23
        1.1. MATCH Functions 23
        1.2. MERGE Operator Examples 24
7. UnityJDBC Driver Internals 27
    1. Overview 27

2. Embedded Relational Database Engine 27
8. Limitations and Planned Features 28
1. Limitations 28
2. Planned Features 28
3. Feature List 28
4. Contacts and Support 29

List of Tables

1.1. UnityJDBC Release History 1
4.1. Comparison Operators 14
4.2. Mathematical Operators 15
4.3. Mathematical Functions 15
4.4. String Functions 15
4.5. Date Functions 16
4.6. Aggregate Functions 17
6.1. MATCH Functions 24
6.2. Employee Database 1 referred to as EmpDB1 25
6.3. Employee Database 2 referred to as EmpDB2 25
6.4. MERGE Full Outer Join Result 25
6.5. Query Final Result 25
6.6. Query 2 Final Result 26
8.1. Planned Features in Coming Versions 28
8.2. Planned Features in Coming Versions 28

Chapter 1. General Information

1. Overview of UnityJDBC

The UnityJDBC driver is a Type 4 JDBC driver capable of querying multiple databases in a single SQL query. The driver can be used similar to other JDBC drivers including in application and web servers, Java applets, or standalone Java programs. Internally, UnityJDBC contains a database engine and optimizer allowing it to efficiently merge and join data from source databases to produce a single ResultSet. UnityJDBC supports updating data using results produced from cross-database queries and performs automatic dialect translation to convert queries into the proper dialect. A brief list of the major supported features is below:

- supports cross-database joins of any number or type of JDBC-accessible sources (Microsoft SQL Server, Oracle, DB2, Postgres, MySQL, Sybase, etc.)
- allows merging and matching of data across databases to detect data inconsistencies, errors, or allow for synchronization of data between databases
- performs SQL dialect translation and automatically executes functions and features internally in the driver if the data source does not support them
- contains an advanced optimizer and query processor that performs efficient query processing by having each source process as much of the query as possible
- cross-database queries can be used to create new tables (INSERT INTO ... SELECT)
- supports cross-database PreparedStatements
- has a driver by-pass feature to allow direct access to individual sources
- supports connection pools and connection properties
- allows for user-defined functions (basic, aggregate, and matching)
- works with any data source that has a JDBC driver and will run on any Java supported platform

2. History of UnityJDBC

UnityJDBC is the result of over 10 years of work on database integration. UnityJDBC was first commercially released in 2006 and continues to be actively supported and extended.

Table 1.1. UnityJDBC Release History

Release Version and Date	Major Features
UnityJDBC v1.0 - May 2006	Cross-database join support, match functions, full optimizer, query by-pass
UnityJDBC v2.0 - May 2007	Connection pools, DataSource connections, more functions
UnityJDBC v3.0 - May 2008	Native INSERT/UPDATE/DELETE, INSERT INTO...SELECT across databases, PreparedStatements, user-defined functions
UnityJDBC v4.0 - August 2011	Database dialect translation, paging using LIMIT/OFFSET, single database subqueries, result caching

3. Further Information and Support

Additional information about UnityJDBC can be found at http://www.unityjdbc.com.

Chapter 2. Installation

1. General Issues

The free evaluation version of UnityJDBC is a fully functioning system. The only constraint with the evaluation version is that it will only produce the first 100 results in a ResultSet. The full version has no restrictions. The evaluation version can be downloaded at http://www.unityjdbc.com/download.html. You are free to distribute the evaluation version of the software.

1.1. System Requirements

UnityJDBC requires a JRE of 1.6 or higher. UnityJDBC will run on a J2SE or J2EE platform.

2. Quick Setup and Installation

UnityJDBC can be downloaded, installed, tested, and configured for your environment in less than 30 minutes. Here are the 5 easy steps:

1. **Downloading** - First, download the UnityJDBC package to your computer from http://www.unityjdbc.com/download.html.

2. **Installation** - The UnityJDBC package contains the driver and some test programs. Unzip the package into a temporary directory. Copy the `UnityJDBC.jar` into your classpath. A common place to install it is in your `ext` directory under your JDK or JRE installation. At this point, copy the JDBC drivers of the databases that you require into this directory as well. Some JDBC drivers for common databases are available in the directory `OtherJDBCDrivers` included in the package.

3. **Testing** - Go to the directory where you unzipped the package and into the subdirectory `code`. Compile and run the test program called `test/ExampleQuery.java`. This program connects to a local HSQLDB that can be started by using the script `startDB.bat` or `startDB.sh` in the directory `sampleDB/hsqldb`. Here are the two commands (executed from the `code` directory) to compile and run the sample code:

   ```
   javac test/ExampleQuery.java

   java test.ExampleQuery
   ```

   If you have issues with compiling or running, try to explicitly indicate the location of the UnityJDBC JAR:

   ```
   javac -cp .;../UnityJDBC.jar;../sampleDB/hsqldb/hsqldb.jar test/ExampleQuery.java

   java -cp .;../UnityJDBC.jar;../sampleDB/hsqldb/hsqldb.jar test.ExampleQuery
   ```

4. **Setting up your environment** - There are three things you must do to setup querying your own databases. First, you must make sure the JDBC driver for your source database system and the UnityJDBC driver are correctly installed in your classpath (Step #2). Second, you must create an XML sources file that contains your source(s) connection information. The example sources file `UnityDemo.xml` file in the `test/xspec` directory is shown below. Third, you must build a XML schema description of each source (called an XSpec or schema file). The graphical program `com.unityjdbc.sourcebuilder.SourceBuilder.java` will help you build these files. You may run the program directly or use the script files `initsources.bat` (Windows) or `initsources.sh` (Linux).

   ```
   <SOURCES>
      <DATABASE>
         <URL>jdbc:hsqldb:hsql://localhost/tpch</URL>
         <USER>sa</USER>
   ```

```
      <PASSWORD></PASSWORD>
      <DRIVER>org.hsqldb.jdbcDriver</DRIVER>
      <XSPEC>test/xspec/UnityDemoPart.xml</XSPEC>
   </DATABASE>
   <DATABASE>
      <URL>jdbc:hsqldb:hsql://localhost/tpch</URL>
      <USER>sa</USER>
      <PASSWORD></PASSWORD>
      <DRIVER>org.hsqldb.jdbcDriver</DRIVER>
      <XSPEC>test/xspec/UnityDemoOrder.xml</XSPEC>
   </DATABASE>
   <DATABASE>
      <URL>jdbc:hsqldb:hsql://localhost/emptydb</URL>
      <DRIVER>org.hsqldb.jdbcDriver</DRIVER>
      <XSPEC>test/xspec/emptydb.xml</XSPEC>
   </DATABASE>
   <DATABASE>
      <URL>jdbc:hsqldb:hsql://localhost/xdb</URL>
      <DRIVER>org.hsqldb.jdbcDriver</DRIVER>
      <XSPEC>test/xspec/mydb.xml</XSPEC>
      <USER>test</USER>
      <PASSWORD>test</PASSWORD>
   </DATABASE>
</SOURCES>
```

5. **Writing your own query program** - Copy the file `ExampleQuery.java` to `MyQuery.java`. There are 2 lines that you must modify. The first line indicates where your new sources XML file is located on your machine. You may specify an absolute or relative path from the current directory. The second line you must modify is to change the SQL query to reference fields and tables in your data source(s). Compile and run the program. Then add more data sources to your XML sources file as in Step #4. Queries can reference any table or field in any data source in your XML sources file as long as you prefix a table or field with the database name such as `MyDB.MyTable.MyField`.

3. Installation on Windows

Here are the steps to install UnityJDBC on a Windows platform:

1. Download and unzip the UnityJDBC package.

2. Copy `UnityJDBC.jar` into your `CLASSPATH`. Usually `C:\Program Files\Java\jdk1.6.0\jre\lib\ext.`

3. Compile and run the test program `ExampleQuery.java`. Make sure to start the local DB using the command `startDB.bat` in the directory `sampleDB/hsqldb`.

   ```
   javac test/ExampleQuery.java
   ```

   ```
   java test.ExampleQuery
   ```

4. Run `initsources.bat` to configure your own sources and use in your Java/JDBC program.

4. Installation on Linux/UNIX

Here are the steps to install UnityJDBC on a Linux/UNIX platform:

1. Download and unzip the UnityJDBC package.

2. Copy `UnityJDBC.jar` into your `CLASSPATH`.

3. Compile and run the test program `ExampleQuery.java`. Make sure to start the local DB using the command `startDB.sh` in the directory `sampleDB/hsqldb`.

```
javac test/ExampleQuery.java
```

```
java test.ExampleQuery
```

4. Run `initsources.sh` to configure your own sources and use in your Java/JDBC program.

5. Installation on Other Platforms

Here are the steps to install UnityJDBC on any Java-enabled platform with JRE 1.6 or higher pre-installed:

1. Download and unzip the UnityJDBC package.

2. Copy `UnityJDBC.jar` into your `CLASSPATH`.

3. Compile and run the test program `ExampleQuery.java`. Make sure to start the local DB using the command `startDB.sh` in the directory `sampleDB/hsqldb`.

   ```
   javac test/ExampleQuery.java
   ```

   ```
   java test.ExampleQuery
   ```

4. Run `SourceBuilder.java` to configure your own sources and use in your Java/JDBC program.

6. Configuring Data Sources

UnityJDBC requires information about the data sources being queried in order to validate, optimize, and execute queries against those data sources. All source information is stored in XML files. There are two types of source information files: *source list files* and *XSpec schema files*. A *source list file* provides information on all the sources that could be potentially queried. This file is referred to inside your Java code via a URL when initializing the driver. Inside the file is information on each source including its connection URL and parameters, JDBC driver, and XSpec schema file location. The sample source list file `code\test\xspec\UnityDemo.xml` is provided in the distribution package. Each data source requires an XSpec schema file. The *XSpec schema file* is an XML encoding of the schema information including table and field names, keys, joins, and relation sizes. It is used for validating queries and during optimization. Two of the XSpec files provided in the distribution are: `code\test\xspec\UnityDemoOrder.xml` and `code\test\xspec\UnityDemoPart.xml`.

There are two ways to create your own source list file and associated XSpec files:

1. The easiest way is to use the `SourceBuilder.java` program. This is a GUI that will guide you through creating the files. It will automatically extract source information and build the necessary files.

2. You can manually build a source list file using a text editor. To produce an XSpec for a source, open up the program called `ExtractorXSpec.java` in the `code/com/unityjdbc/sourcebuilder` directory. Modify the JDBC URL, driver path, and output directory accordingly and run the program. The account that you connect with must have read access to the database and associated tables that you want to access. After the XML file has been produced, move it to the directory where you want it and update the source list XML file to reference the correct location.

7. Testing the Installation

The easiest way to test the installation of the driver is to run the `ExampleQuery.java` program. This will connect to the local HSQL database provided. If you encounter errors, make sure that `UnityJDBC.jar` and `hsqldb.jar` are in your `CLASSPATH`.

Chapter 3. Tutorial

1. Using the Sample Programs

Sample programs are provided in the directory `code`. Here is a list of the programs and the features they demonstrate:

1. `ExampleQuery.java` - a query example that joins data across two databases
2. `ExampleUpdate.java` - demonstrates `INSERT/UPDATE/DELETE` and how to store a cross-database query result into a tables
3. `ExampleMetaData.java` - query example showing how to extract metadata information
4. `ExampleMerge.java` - query example showing `MERGE` functionality (cross-database validation and data comparison)
5. `ExampleEngine.java` - an advanced example that shows how users can use the UnityJDBC database engine directly

All of these examples use a local HSQL database that can be started using the script `startDB.bat` or `startDB.sh` in the directory `sampleDB/hsqldb`.

To compile and run any of these sample programs make sure you are in the `code` directory and execute the following commands:

```
javac test/ExampleQuery.java
```

```
java test.ExampleQuery
```

If you have CLASSPATH issues, you can explicitly indicate the location of the HSQL JDBC driver and the UnityJDBC driver by:

```
javac -cp.;../UnityJDBC.jar;../sampleDB/hsqldb/hsqldb.jar test/ExampleQuery.java
```

```
java -cp.;../UnityJDBC.jar;../sampleDB/hsqldb/hsqldb.jar test.ExampleQuery
```

1.1. Using ExampleQuery.java

The `ExampleQuery.java` demonstrates the basic features of the UnityJDBC driver. The code is below.

```
import java.sql.*;

public class ExampleQuery
{
// URL for sources.xml file specifying what databases to integrate.
// This file must be locally accessible or available over the Internet.
static String url="jdbc:Unity://test/xspec/UnityDemo.xml";

public static void main(String [] args) throws Exception
{
Connection con = null;
Statement stmt = null;
ResultSet rst;

try{
   // Create new instance of UnityDriver and make connection
   System.out.println("\nRegistering driver.");
   Class.forName("unity.jdbc.UnityDriver");

   System.out.println("\nGetting connection:  "+url);
   con = DriverManager.getConnection(url);
   System.out.println("\nConnection successful for "+ url);
```

```
   System.out.println("\nCreating statement.");
   stmt = con.createStatement();
   // Unity supports scrollable resultsets, but performance is better with FORWARD_ONLY
   // stmt = con.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
   ResultSet.CONCUR_READ_ONLY);

   // A query is exactly like SQL.
   // Attributes should be FULLY qualified: databaseName.tableName.fieldName
   // Statement must end with a semi-colon ;
   // This example query performs two joins on the client-side (across databases).
   String sql = "SELECT PartDB.Part.P_NAME, OrderDB.LineItem.L_QUANTITY, "
             + " OrderDB.Customer.C_Name, PartDB.Supplier.s_name "
             + " FROM OrderDB.CUSTOMER, OrderDB.LINEITEM, OrderDB.ORDERS, "
             + "      PartDB.PART, PartDB.Supplier "
             + " WHERE OrderDB.LINEITEM.L_PARTKEY = PartDB.PART.P_PARTKEY AND "
             + "       OrderDB.LINEITEM.L_ORDERKEY = OrderDB.ORDERS.O_ORDERKEY "
             + "       AND OrderDB.ORDERS.O_CUSTKEY = OrderDB.CUSTOMER.C_CUSTKEY "
             + "       AND PartDB.supplier.s_suppkey = OrderDB.lineitem.l_suppkey "
             + " AND OrderDB.Customer.C_Name = 'Customer#000000025';";

   // Note: Client's local JVM is used to process some operations.
   // For large queries, this may require setting a large heap space.
   // JVM command line parameters: 0 -Xms500m -Xmx500m
   // These parameters set heap space to 500 MB. Do not set higher than 80% of memory size.
   rst = stmt.executeQuery(sql);

   System.out.println("\n\nTHE RESULTS:");
   int i=0;
   long timeStart = System.currentTimeMillis();
   long timeEnd;
   ResultSetMetaData meta = rst.getMetaData();

   System.out.println("Total columns: " + meta.getColumnCount());
   System.out.print(meta.getColumnName(1));
   for (int j = 2; j <= meta.getColumnCount(); j++)
      System.out.print(", " + meta.getColumnName(j));
   System.out.println();

   while (rst.next()) {
      System.out.print(rst.getObject(1));
      for (int j = 2; j <= meta.getColumnCount(); j++)
         System.out.print(", " + rst.getObject(j));
      System.out.println();
      i++;
   }

   timeEnd = System.currentTimeMillis();
   System.out.println("Query took: "+((timeEnd-timeStart)/1000)+" seconds");
   System.out.println("Number of results printed: "+i);
   stmt.close();
   System.out.println("\nOPERATION COMPLETED SUCCESSFULLY!");
}
catch (SQLException ex)
{   System.out.println("SQLException: " + ex);
}
finally
{
   if (con != null)
   try{   con.close();   }
   catch (SQLException ex)
   {   System.out.println("SQLException: " + ex); } // end try-catch-finally block
} //end main()
} // end UnityDemo
```

The UnityJDBC driver behaves exactly like other JDBC drivers. The basic steps for querying a database with a JDBC driver are:

1. **Load the driver** - This is accomplished by the line `Class.forName("unity.jdbc.UnityDriver");`

2. **Make a connection** - A connection is made to a database by providing the database URL and other properties including user ids and passwords. This example is using the `DriverManager` to make the connection (`con = DriverManager.getConnection(url);`). Note that the URL is of the form `jdbc://unity//<path_to_source_list_file>`. In this case, the URL is `jdbc:Unity://test/xspec/UnityDemo.xml`.This path may be an absolute or relative path on the machine. It is also possible to retrieve encrypted XML files from a network source for use in Applets and other network applications. The source list file provides the connection information for the individual data sources for use by UnityJDBC.

3. **Execute a statement** - UnityJDBC follows the JDBC API for creating statements and executing queries and updates. There are some methods unique to UnityJDBC which are covered in a later section. **Note that all statements in UnityJDBC must be terminated with a semi-colon.** Standard SQL syntax is supported. The major difference is that tables in different databases can be referenced in the same query. This is accomplished using the syntax `DBName.TableName` to refer to tables and `DBName.TableName.FieldName` to refer to fields. (Note that aliasing using `AS` is supported.) If full names are not provided, UnityJDBC will attempt to match as appropriate, but it will generate errors if the provided names are not unique.

This file is a good one to modify to start your own program. Simply change the class and file name, the URL to the location of your source list file, and the query executed, and you are done.

1.2. Using ExampleUpdate.java

UnityJDBC natively supports `INSERT`, `UPDATE`, and `DELETE` statements on a single database. These statements can be executed in by-pass mode in which case UnityJDBC does not parse or validate the statement and passes it straight to the JDBC driver for the corresponding database. In native mode, UnityJDBC will parse and validate the statement before passing it to the data source. Note that the basic `INSERT`, `UPDATE`, and `DELETE` statements operate only on a single table in SQL, so no cross-database query processing is necessary. A sample of the code in `ExampleUpdate.java` is below.

```
Class.forName("unity.jdbc.UnityDriver");
con = DriverManager.getConnection(url);
stmt = con.createStatement();

// Example #1: Basic query
String sql = "SELECT * FROM mydb.Customer;";
rst = stmt.executeQuery(sql);
ExampleHSQL.printResult(rst);

// Example #2: DELETE using native parsing
String databaseName = "mydb";
sql = "DELETE FROM mydb.customer WHERE mydb.customer.id = 51 or mydb.customer.id=52;";
stmt.executeUpdate(sql);

// Example #3: INSERT (by-pass method)
sql = "INSERT INTO Customer (id,firstname,lastname,street,city) "
          + " VALUES (51,'Joe','Smith','Drury Lane', 'Detroit')";
((UnityStatement) stmt).executeByPassQuery(databaseName,sql);

// Example #4: INSERT - Unity Parsed
sql = "INSERT INTO mydb.Customer (mydb.Customer.id, mydb.Customer.firstname, "
          + " mydb.Customer.lastname, mydb.Customer.street, mydb.Customer.city) "
          + " VALUES (52,'Fred','Jones','Smith Lane', 'Chicago');";
stmt.executeUpdate(sql);

// Example #5: INSERT INTO (SELECT...) across databases
sql = "INSERT INTO emptydb.customer (SELECT * FROM mydb.customer);";
stmt.executeUpdate(sql);
```

```
// Prove that we transferred the data
sql = "SELECT * FROM emptydb.Customer;";
rst = stmt.executeQuery(sql);
ExampleHSQL.printResult(rst);
```

Note that you can use the by-pass feature to execute any statement on a source database that UnityJDBC does not natively support. Experimental results show that the by-pass features adds insignificant overhead compared to calling the source JDBC driver directly. Thus, client code only needs to load and use the UnityJDBC driver directly. This results in more portable code that can be more easily moved between database systems.

When UnityJDBC parses the SQL, you can use table and field references that are prefixed with the database name. This is optional if the table and field names are unique across all databases, otherwise the database name is required. The database name is assigned in the XSpec describing the source and does not have to be the same as the system name used by the database system itself. That is, the name can be set by the developer using UnityJDBC.

Since UnityJDBC 3.0 users have the ability to take UnityJDBC queries and feed them into an INSERT INTO statement and populate a table in the database. This allows a user to write a cross-database query to collect information from multiple sources and then insert the result back into a table in any one of the sources. Currently, the only restriction is that the table that will be inserted into must exist and must be present in the XSpec schema file describing the source.

1.3. Using ExampleMetadata.java

`ExampleMetadata.java` demonstrates UnityJDBC's support for the `DatabaseMetaData` interface. This interface functions exactly according to the standard with the major difference that metadata is returned for all databases in the source list file rather than from a single database. That is, all your "integrated" databases really do appear as a single database to your application.

1.4. Using ExampleMerge.java

A truly distinctive feature of UnityJDBC is the ability to compare information across databases not just to perform cross-database joins. UnityJDBC uses a `MERGE` syntax that has no equivalent in SQL. The basic idea is that often the reason to integrate databases is to compare the differences in the data they store either to detect inconsistencies, for use with data synchronization, or to identify areas of overlap between data sources. A `MERGE` query takes two or more subqueries (which may be on a single source or multiple sources) and combines them using a full outer join and attribute match functions. The full outer join will relate data tuples based on attribute values, and the match functions allow the programmer to compare data between sources on a per field basis and decide which data to keep. A sample of the code in `ExampleMerge.java` is below.

```
Class.forName("unity.jdbc.UnityDriver");
con = DriverManager.getConnection(url);
stmt = con.createStatement();

// Show the data in the two tables that we will be integrating
System.out.println("Data in Person table in database 1:");
ResultSet rst = stmt.executeQuery("SELECT * FROM mydb.person;");
printResult(rst);
System.out.println("\nData in Users table in database 2:");
rst = stmt.executeQuery("SELECT * FROM emptydb.users;");
printResult(rst);

// Query #2: All users in database 1 but not database 2
System.out.println("\nAll people in database 1 but not in database 2:");
queryString = "SELECT * FROM mydb.person "
           + " MERGE ON Q1.C1=Q2.C1 EXTRACT ALL FILTER Q2.C1 IS NULL "
           + " SELECT * FROM emptydb.users;";
rst = stmt.executeQuery(queryString);
printResult(rst);
```

```
// Query #4: Find the users whose salary is different between the two databases
System.out.println("\nPeople where the salary field is different in the two databases:");
queryString = "SELECT * FROM mydb.person "
            + " MERGE ON userid=id EXTRACT userid, id, firstname, "
            + "             Q1.C4 AS salary_db1, Q2.C4 AS salary_db2 "
            + " FILTER userid IS NOT NULL and id IS NOT NULL and salary_db1!=salary_db2 "
            + " SELECT * FROM emptydb.users;";
rst = stmt.executeQuery(queryString);
printResult(rst);

// Query #5: Produce an "integrated" result of user id, full name, age (max), salary (both)
System.out.println("\nIntegrated table with user id, full name, age (max), salary (both)");
queryString = "SELECT * FROM mydb.person "
            + " MERGE ON userid=id EXTRACT CHOOSENN(userid,id,1) as userid, "
            + "        CHOOSENN(firstname+' '+lastname,fname+' '+lname,1) as fullname, "
            + " MAX(DATEDIFF('y',date(),birthdate),age) as age, "
            + "   GROUPREF(Q1.C4,Q2.C4) as salary"
            + " SELECT * FROM emptydb.users;";
rst = stmt.executeQuery(queryString);
printResult(rst);

// Query #6: Take result produced with previous query and insert into an existing table
System.out.println("\nInsert into a table called AllUsers results from both sources:");
queryString = "INSERT INTO AllUsers (SELECT * FROM mydb.person "
           + " MERGE ON userid=id EXTRACT CHOOSENN(userid,id,1) as userid,"
           + " CHOOSENN(firstname,fname,1) as fname, CHOOSENN(lastname,lname,1) as lname,"
           + " MAX(mydb.person.salary,emptydb.users.salary), birthdate, yearHired,deptName"
           + " SELECT * FROM emptydb.users);";
stmt.executeUpdate(queryString);
```

The `MERGE` clause matches two subqueries using a join condition. A field in either subquery is referred to by its query number (`Q1` or `Q2`) and its column index starting from 1 (`C1`, `C2`, ...). `GROUPREF()` is an example of a `MATCH` function. This particular function will group the two values into an ArrayList and keep a reference on what source the values were from. Thus, the `MERGE` feature allows you to integrate data and track the source (provenance) of the integrating data. The `FILTER` clause behaves essentially like a `HAVING` clause in SQL and allows the filtering of rows produced after the integration step. More details on the syntax and use of `MERGE` queries is in Chapter 6.

These example `MERGE` queries involve two different people databases. Query #2 shows how you can to determine when records are in one database but not the other. Query #4 shows how records that have different field values in two different databases can be identified. Query #5 shows how to integrate the data into a single query result using `MATCH` functions. The function `CHOOSENN(field1,field2,defaultSource)` will select the field value from the defaultSource unless it is `NULL`. Functions like `MAX()` and `MIN()` are also supported. The `GROUPREF()` function will create an ArrayList of the two field values while also tracking the source of the value. This can be used to display differences to the user as well as tracking the source of the data. Query #6 uses `INSERT INTO` to take an integrated query result and insert the records into another table. Some of the output generated by this sample code is below.

```
Data in Person table in database 1:
USERID  FIRSTNAME  LASTNAME  SALARY     BIRTHDATE  ISACTIVE   YEARHIRED
id1     Joe        Smith     55000      1970-01-13 true       2000
id2     Fred       Anders    45000      1963-03-08 true       2001
id3     Steve      Batner    75000      1955-10-23 false      1998
id5     Perry      Crusus    65000      1988-11-11 true       2003
id7     Jason      Deed      35000      1942-05-09 true       2006

Data in Users table in database 2:
ID       FNAME     LNAME     SALARY     AGE        DEPTNAME
id1      Joe       Smithers  53000.0    38         Accounting
id2      Fred      Anders    45000.0    45         Sales
id3      Steve     Batner    85000.0    53         Sales
id4      New       Guy       21000.0    21         Sales
id5      Pery      Crustes   55000.0    20         Owner
```

```
All people in database 1 but not in database 2:
USERID FIRSTNAME LASTNAME SALARY BIRTHDATE  ISACTIVE YEARHIRED ID    FNAME  LNAME  SALARY AGE
id7    Jason     Deed     35000  1942-05-09 true     2006      (null)(null) (null) (null) (null)

People where the salary field is different in the two databases:
USERID     ID       FIRSTNAME                 salary_db1 salary_db2
id1        id1      Joe                       55000      53000.0
id3        id3      Steve                     75000      85000.0
id5        id5      Perry                     65000      55000.0

Integrated table with user id, full name, age (keep max), salary (show both with source):
userid     fullname             age   salary
id1        Joe Smith            38    [55000 (mydb), 53000.0 (emptydb)]
id2        Fred Anders          45    [45000 (mydb), 45000.0 (emptydb)]
id3        Steve Batner         53    [75000 (mydb), 85000.0 (emptydb)]
id5        Perry Crusus         20    [65000 (mydb), 55000.0 (emptydb)]
id4        New Guy              21    [<null> (mydb), 21000.0 (emptydb)]
id7        Jason Deed           66    [35000 (mydb), <null> (emptydb)]

Insert into a table called AllUsers integrated results from both sources:
USERID   FIRSTNAME  LASTNAME  SALARY     BIRTHDATE  YEARHIRED  DEPTNAME
id1      Joe        Smith     55000      1970-01-13 2000       Accounting
id2      Fred       Anders    45000.0    1963-03-08 2001       Sales
id3      Steve      Batner    85000.0    1955-10-23 1998       Sales
id5      Perry      Crusus    65000      1988-11-11 2003       Owner
id4      New        Guy       21000.0    (null)     (null)     Sales
id7      Jason      Deed      35000      1942-05-09 2006       (null)
```

1.5. Using ExampleEngine.java

Embedded in the UnityJDBC driver is a complete relational engine. This is required to process cross-database join queries. Most users will not interact with the engine directly, and their only contact with the engine may be to increase the JVM heap sizes for processing large cross-database queries. However, all of the relational operators of selection, projection, and join are available for direct use in your programs. The join algorithms support sources larger than main memory, and one particular algorithm, Early Hash Join, is a new research algorithm that was submitted for US patent. It is also possible to explicitly track global query progress on a per operator basis or perform your own optimization of queries after the UnityJDBC optimizer has built an execution tree.

2. Configuring Your Data Sources

UnityJDBC requires information about the data sources being queried in order to validate, optimize, and execute queries against those data sources. All source information is stored in XML files. There are two types of source information files: *source list files* and *XSpec schema files*. A s*ource list file* provides information on all the sources that could be potentially queried. This file is referred to inside your Java code via a URL when initializing the driver. Inside the file is information on each source including its connection URL and parameters, JDBC driver, and XSpec file location. The sample source list file `code\test\xspec\UnityDemo.xml` is provided in the distribution package. Each data source requires an XSpec schema file. The *XSpec schema file* is an XML encoding of the schema information including table and field names, keys, joins, and relation cardinalities. It is used for validating queries and during optimization. Four XSpec schema files are provided in the distribution: `code\test\xspec\UnityDemoOrder.xml`, `code\test\xspec\UnityDemoPart.xml`, `code\test\xspec\mydb.xml`, and `code\test\xspec\emptydb.xml`.

There are two ways to create your own source list file and associated XSpec files:

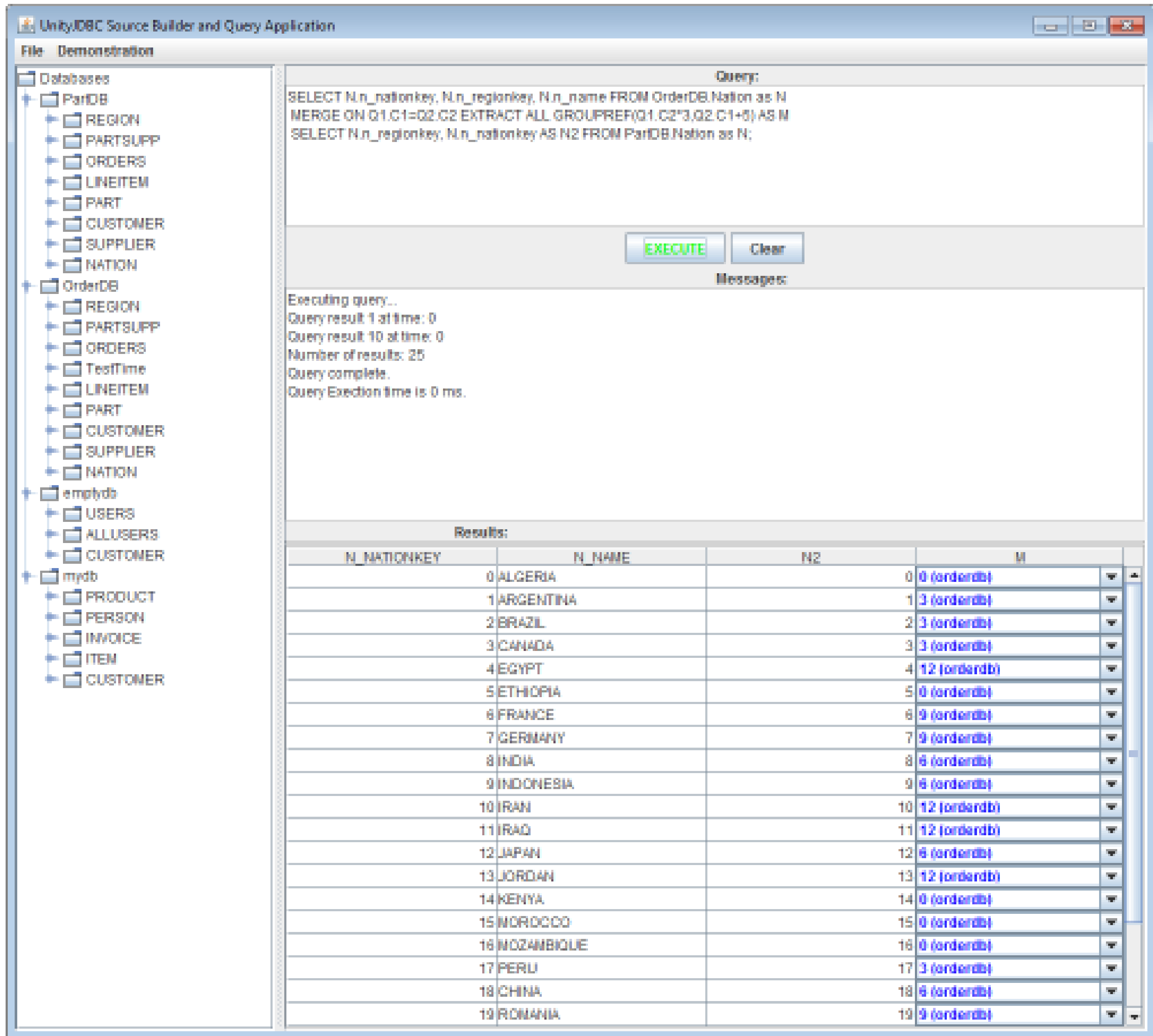
1. The easiest way is to use the `SourceBuilder.java` program. This is a GUI that will guide you through creating the files. It will automatically extract source information and build the necessary files.

2. You can manually build a source list file using a text editor. To produce an XSpec for a source, open up the program called `ExtractorXSpec.java` in the `com/unityjdbc/sourcebuilder` directory. Modify the JDBC URL, driver path, and output directory accordingly and run the program. The account that you connect with must have read

access to the database and associated tables that you want to access. After the XML file has been produced, move it to the directory where you want it and update the source list XML file to reference the correct location.

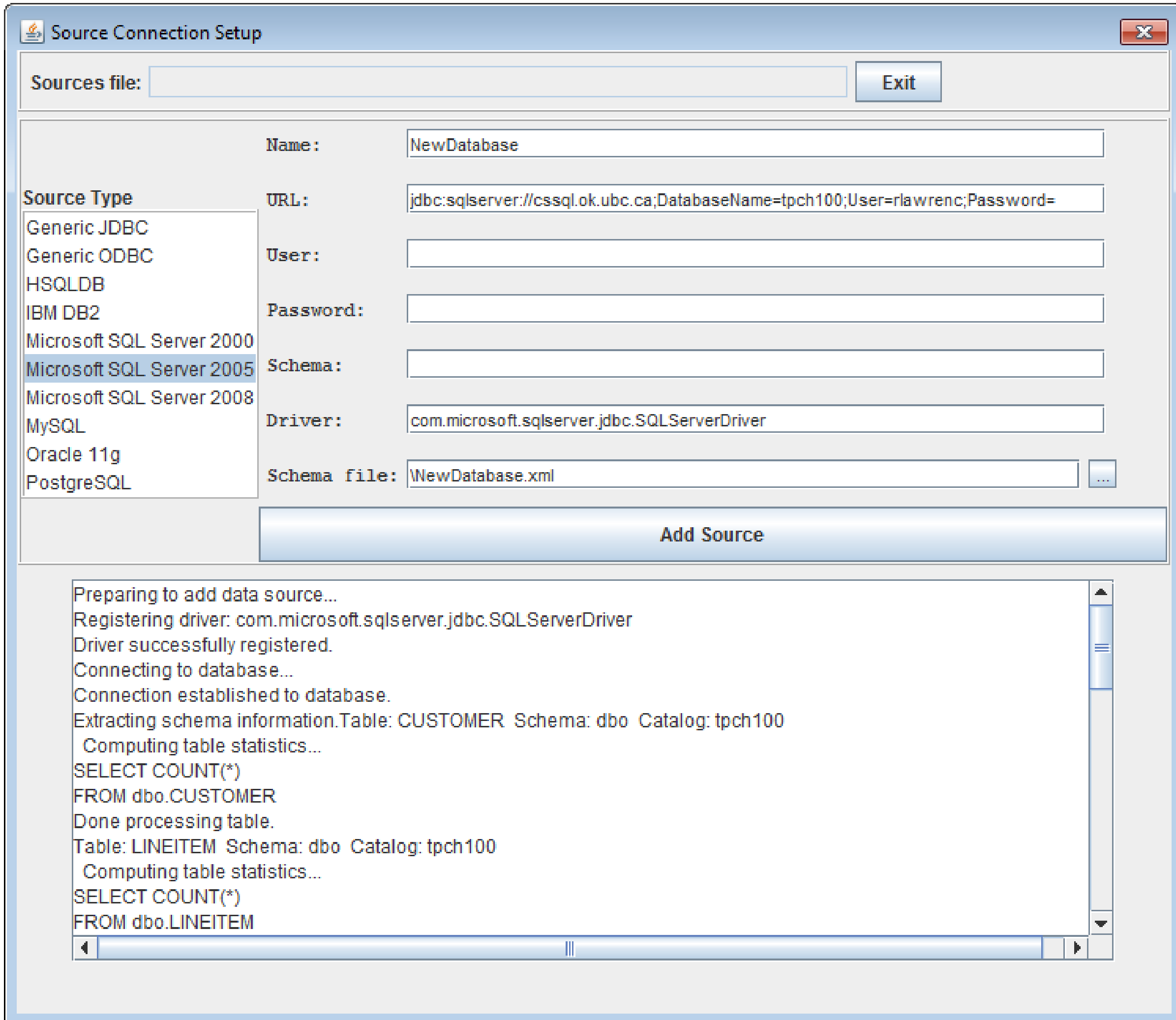
The SourceBuilder utility allows you to define both files.

Before creating your own files, select from the Demonstration menu, Sources then UnityDemo. This will open up the sources file for the demonstration database. Then, select from the Demonstration Menu, Queries, then Merge and Match test. Text should be filled in the query box then click Execute. The query should execute. If it does not, make sure you have setup your environment properly including puttiing the JARs in your CLASSPATH. The result is shown below.



To create your own sources file, select from the File Menu the item New Source Group. Select the location of the file. We recommend the file name end in XML as it is an XML file (but that is not required). Then select from the File Menu, New Source. A new screen pops up asking for information on the source and the location to store the XSpec schema file. Provide your JDBC connection information and click Add Source. If extraction is successful, an XSpec schema file will be created and the source added to your source list file. You can then exit the add source screen and test a query. A screenshot of adding an Microsoft SQL Server 2005 database is below. Note that you can specify the

user id and password in the URL or in the individual text fields. The tables in the database will be extracted based on the privileges of the provided user. Thus, if user X only has privilege to see 5 of the 10 tables, only those tables will be extracted and put in the XSpec. It is recommended that you create your database user first that will access the tables, provide the correct privileges, then perform the extraction. Extraction using an administration account will extract schema data for all tables the administrator can access. By default, all schemas are scanned. This may cause problems if system tables are encountered when connecting using administrator accounts. In this case, use the schema field to select only the schema that should be extracted using a pattern match similar to the DatabaseMetaData JDBC API.



More examples of using the SourceBuilder utility can be found on the web site. Constructing the sources and XSpec schema files is typically the most time-consuming feature to get going with your own project, but often can be completed in less than an hour.

3. Writing Your Own Java Programs

Writing a program that uses UnityJDBC is almost no different than using other JDBC drivers. The differences to remember are:

1. **Must configure sources** - You must configure your sources by creating a source list file and XSpec schema file for each source.
2. **Connection URL refers to source list file** - When connecting to the UnityJDBC driver, using the method `con = DriverManager.getConnection(url);` the `url` must refer to the location of your source list file. An example

URL is "`jdbc:Unity://Tests/xspec/UnityDemo.xml`". The file can be specified using an absolute or relative path from the current directory. The URL may also be a web URL.

3. **Identifiers ofter require a database prefix** - When referring to tables or fields in your query, you often require specifying the database name where they come from such as `MyDB.MyTable` or `MyDB.MyTable.MyField`. For more information, see SQL identifiers.

Chapter 4. Supported SQL Syntax

1. Overview

UnityJDBC supports a cross-database `SELECT` statement. The `SELECT` statement has the standard SQL-92 syntax and supports `WHERE`, `ORDER BY`, `GROUP BY`, and `HAVING`. UnityJDBC supports most subqueries if the entire query is on a single database. UnityJDBC does NOT support cross-database subqueries or `SELECT INTO`. SQL functions are supported using a function syntax with parameters rather than using SQL keywords and syntax. Table and fields often should be prefixed with the database name they originate from. This database name is provided in the XSpec for the data source.

2. Data Types

The standard SQL data types are supported. Since UnityJDBC uses the JDBC drivers provided by database vendors, non-standard data types may not be universally supported.

3. Identifiers

An *identifier* is a string used to reference a database, table, or field. Identifiers follow the standard SQL rules. Since a UnityJDBC query may span multiple databases, table and field identifiers defined in a data source may not be unique across all data sources. In which case, the database name should be appended onto the identifier to create a unique system-wide identifier. For instance, consider an order database given the name `OrderDB` with a table called `Orders` and fields `id` and `orderDate`. The `Orders` table may be referred to using only `Orders` or `OrderDB.Orders`. Similarly, the field `id` may be referred to as `Orders.id` or `OrderDB.Orders.id`. Standard aliasing using `AS` in the `FROM` and `SELECT` clauses is supported. Delimited identifiers are supported by enclosing in double quotes (e.g. "from" or "my field with spaces"). Delimited identifiers must be used for SQL reserved words. It is recommended to avoid delimited identifiers when possible.

4. Functions and Operators

Arithmetic operators +, -, /, %, * are supported as well as generic expressions. Functions are not specified according to SQL keyword syntax but rather as a function identifier with parameters similar to programming languages. The format of functions is: `function (param1, param2, ...).`

4.1. Logical Operators

The logical operators of `AND`, `OR`, `NOT`, and `XOR` are available.

4.2. Comparison Operators

The following comparison operators are available:

Table 4.1. Comparison Operators

Operator	Description
<	less than
>	greater than
<=	less than or equal to
>=	greater than or equal to
=	equal
!=	not equal

Operator	Description
IS [NOT] NULL	tests if value is NULL
IS [NOT] [TRUE	FALSE]	tests if value is true or false

4.3. Arithmetic Functions and Operators

The following mathematical operators are supported:

Table 4.2. Mathematical Operators

Operator	Description
+	addition (and string concatenation for strings)
-	subtraction
/	division
%	modulus (remainder of integer division)
*	multiplication

The following are a few of the mathematical functions supported. A complete list of functions is available on the web site.

Table 4.3. Mathematical Functions

Function	Return Type	Example	Result	Description
abs(x)	Same as x	abs(-17.4)	17.4	Absolute value
ceil(dp or numeric)	Same as input	ceil(-42.8)	-42	Smallest integer not less than argument
exp(dp or numeric)	Same as input	exp(1.0)	2.71828182845905	exponential
floor(dp or numeric)	Same as input	ln(2.0)	0.693147180559945	natural logarithm
log(dp or numeric)	Same as input	log(100.0)	2	base 10 logarithm
power(a dp, b dp)	double precision	power(9,3)	729	a raised to the power of b
random()	double precision	random()		random value between 0.0 and 1.0
sqrt(dp or numeric)	double precision	sqrt(2.0)	1.4142135623731	square root

4.4. String Functions

The following are a few of the string functions supported. A complete list of functions is available on the web site.

Table 4.4. String Functions

Function	Return	Example	Result	Description
<str> + <str>	String	'Unity' + 'JDBC'	UnityJDBC	String concatenation
ascii(string)	int	ascii('xyz')	120	ASCII code of the first character of the input string
length(string)	int	length('UnityJDBC')	9	Length of string in characters

Function	Return	Example	Result	Description
lower(string)	String	lower('JDBC')	jdbc	Convert string to lower case
position(searchstring, string)	int	position('J','UnityJDBC')	5	Location of searchstring in string (indexed from 1)
replace(sourcestring, searchstring, replacestring)	String	replace('abUnityabJDBC', 'ab', 'XX')	XXUnityXXJDBC	Replace all occurrences of searchstring in sourcestring with replacestring
substring(string, start)	String	substring('UnityJDBC',6)	JDBC	substring starting at position start
substring(string, start, count)	String	substring('UnityJDBC,6,2)	JD	substring starting at position start and taking count characters
trim(string)	String	trim(' UnityJDBC ')	UnityJDBC	remove leading and trailing spaces from string
ltrim(string) OR trim(string, 'LEADING')	String	trim(' UnityJDBC ')	'UnityJDBC '	remove leading spaces from string
rtrim(string) OR trim(string, 'TRAILING')	String	trim(' UnityJDBC ')	' UnityJDBC'	remove trailing spaces from string
trim(string, ['BOTH', 'LEADING', 'TRAILING'], [<chars>])	String	trim('aaaUnityJDBCbbb', 'BOTH', 'ab')	UnityJDBC	remove leading, trailing or both from string where characters removed may be optionally specified in <chars>
upper(string)	String	upper('jdbc')	JDBC	Convert string to upper case

4.5. Pattern Matching Operators

Pattern matching is supported using the `LIKE` operator.

For example, `'abcdef' LIKE 'ab%'` is true. The '`%`' is used to match one or more characters, and '_' is used to match a single character.

4.6. Data Type Conversion Functions

Data type conversions are performed using the `CAST(x,y)` function. The `CAST` function takes any object as the first parameter and takes a string literal representation of the type to cast to as the second parameter. Note that the type must be put in single quotes as a string literal. Example:

`CAST(45, 'VARCHAR')` creates `'45'`

Possible type names are: `'VARCHAR'`, `'CHAR'`, `'INT'`, `'FLOAT'`, `'DOUBLE'`, `'DATE'`, `'TIMESTAMP'`, `'TIME'`.

4.7. Date/Time Functions and Operators

The following are a few of the date functions supported. A complete list is on the website.

Table 4.5. Date Functions

Function	Return Type	Example	Result	Description
CURRENT_TIMESTAMP	TIMESTAMP	CURRENT_TIMESTAMP	2011-07-06 12:53:45	Returns the current date. Format: "yyyy-MM-dd HH:mm:ss"

Function	Return Type	Example	Result	Description
CURRENT_TIME	TIME	CURRENT_TIME	12:53:45	Returns the current time. Format: "HH:mm:ss"
CURRENT_DATE	DATE	CURRENT_DATE	2011-07-06	Returns the current date. Format: "yyyy-MM-dd"
YEAR	INT	YEAR('2011-07-06)	2011	Returns the year of the given date expression.
MONTH	INT	MONTH('2011-07-06)	7	Returns the month of the given date expression.
DAY	INT	DAY('2011-07-06)	6	Returns the day of the given date expression.
DATEADD	TIMESTAMP	DATEADD('2011-07-06', INTERVAL 3 days)	2006-07-06 12:53:45	Allows the addition of a given date field to a datetime expression. Intervals are supported and are translated as necessary for systems that do not support them.

4.8. Aggregate Functions

The following aggregate functions are supported:

Table 4.6. Aggregate Functions

Function	Argument Type	Return Type	Description
avg(expression)	int, float, double precision type	int for integer types, double precision for float/double types	Average of all input values
count(*)	N/A	int	Count of number of input values
count(expression)	any	int	Count of number of non-null input values
group_concat(expr)	any	varchar	Returns a comma-separated list of all input values.
max(expression)	any comparable type	same as input	Maximum of all input values
min(expression)	any comparable type	same as input	Minimum of all input values
sum(expression)	int, float, double precision type	int for integer types, double precision for float/double types	Sum of all input values

4.9. User-Defined Functions and Support for Other Functions

For queries on a single database, UnityJDBC parses functions and passes them directly to the database engine for execution. Thus, all functions that can be executed at the source are available. UnityJDBC and user-defined functions are used only when applying functions to data **after** it is extracted from the sources. UnityJDBC will parse queries containing functions that it itself cannot process in its internal database engine. These functions are passed down to the

database engine and executed locally. Only functions that require inputs from more than one database are processed in the UnityJDBC database engine. All other functions are passed down to the sources.

UnityJDBC supports user-defined functions (UDFs). Adding your own user-defined function is easy. There are three different types of functions: tuple functions, aggregate functions, and MATCH functions. A tuple function operates on one tuple at a time for its data and includes functions like `SUBSTRING()` and `ABS()`. An aggregate function is used in `GROUP BY` queries and aggregates an expression (usually a column) across multiple rows in a group to produce a single value. Examples include `MAX()` and `COUNT()`. MATCH functions are unique to UnityJDBC. They allow expressions/columns to be related across data sources. MATCH functions include `MAX()` and `MIN()` but also `GROUP()`, `CHOOSE()`, and others. More information on these functions can be found in the section on MATCH functions.

To create a tuple function, you must create a Java class that extends the Function class. A template example is in the file `F_Function_Template.java`. This class must implement a constructor, an evaluate() method, and provide information on the parameters it requires. Once completed, as long as this function is available in the `CLASSPATH`, UnityJDBC will search for it when called. Similar templates are available for aggregate functions, `A_Aggregrate_Template.java`, and MATCH functions, `M_Merge_Template.java`. Sample code is provided in the directory `unity/functions`.

4.10. Function Translation

UnityJDBC has a database of known functions. This database contains information on what functions are supported on each data source. This is how UnityJDBC processes functions:

1. **UnityJDBC does not support function** - If a function is not in the UnityJDBC database, it is passed down as-is to the underlying source. If the source is able to execute it successfully, the query continues. If not, an error is thrown.

2. **UnityJDBC supports function, data source requires translation** - If the function requested in the query is not directly supported by the data source (different name, different parameters, etc.), but UnityJDBC contains a mapping in its database, the function is translated to the correct form on the data source and executed on the data source.

3. **UnityJDBC supports function, data source does not support function** - If UnityJDBC supports the function but not the data source, then the query is optimized to perform as much of the processing as possible on the source, but the function execution is performed internally in UnityJDBC. This way your query can execute on data sources with the help of UnityJDBC that do not support the required functions.

4. **UnityJDBC is running with local execution** - If the local execution flag is set for the UnityStatement object executing the query, all functions except aggregate functions are executed by UnityJDBC. This setting may be useful to reduce load on the source or to guarantee absolute consistency of function execution across different sources.

The UnityJDBC function database is encrypted and stored in the unityjdbc.jar. To add user-defined functions to the function database, create a mapping.xml file in the JRE classpath (execution directory, etc.) that stores the information on the function. An example is included in the release and more information is available on the web site.

4.11. Non-parsed Functions

UnityJDBC attempts to support most of the SQL standard. If there is a function or feature not supported, it is possible to use the NP() function to pass the query string directly to the data source by-passing UnityJDBC validation. This may be used to support subqueries or non-standard functions or SQL syntax. Below are several examples.

```
Query:
SELECT N1.n_nationkey, NP('OrderDB','n_name','varchar')
FROM OrderDB.Nation N1 WHERE N1.n_nationkey = 1;

Result: (n_name is substituted directly into the query)
SELECT n_nationkey, n_name
FROM Nation N1 WHERE N1.n_nationkey = 1

Query:
```

```
SELECT N1.n_nationkey,  NP('OrderDB','(select n_name from nation n2
where N1.n_nationkey = N2.n_nationkey)','varchar') as name
FROM OrderDB.Nation N1 WHERE N1.n_nationkey = 1

Result:
SELECT N1.N_NATIONKEY,
(select n_name from nation n2 where N1.n_nationkey = N2.n_nationkey) name
FROM NATION N1 WHERE N1.N_NATIONKEY = 1

Query:
SELECT N2.*
FROM NP('OrderDB','(select n_name,n_nationkey from nation)','n_name,n_nationkey') N1,
NP('PartDB','(select n_name,n_nationkey from nation)','n_name,n_nationkey') as N2
where N2.n_nationkey < 2 and N1.n_nationkey = N2.n_nationkey;

Result:
// Substitutes subquery expression for each of the two data sources (OrderDB and PartDB).
// The result of the two subqueries is then joined at the UnityJDBC level.
// OrderDB:
SELECT N2.n_name, N2.n_nationkey
FROM (select n_name,n_nationkey from nation) N2\n WHERE N2.n_nationkey < 2
// PartDB:
SELECT N1.n_nationkey FROM (select n_name,n_nationkey from nation) N1
```

More information on non-parsed functions is available on the web site.

5. SELECT Statement

The SELECT statement supported by UnityJDBC has the following syntax. **Note that all statements MUST be terminated with a semi-colon.**

```
SELECT [ALL | DISTINCT ] <exprList>
      FROM <tableList>
      [WHERE <condition>]
      [GROUP BY <exprList>
      [HAVING <condition>]
      [ORDER BY <expr> [ASC | DESC],...]
      [LIMIT <expr> [OFFSET <expr>]]
```

- An `<exprList>` is a list of expressions. Each individual expression `<expr>` may be a column identifier, a literal constant, or some expression consisting of operators, functions, constants, and column identifiers. Recall that a column identifier may often need to be prefixed by its database name and table name.
- A `<tableList>` is a list of table references. Each table reference can be aliased using the `AS` operator.
- A `<condition>` is a boolean condition that may contain multiple subconditions related using `AND`, `OR`, and `XOR`.
- If the `GROUP BY` clause is used, no attributes should be present in the `SELECT <exprList>` that are not in an aggregate function or are `GROUP BY` attributes.
- The `HAVING <condition>` filters groups and typically should contain only aggregate functions.
- The `ORDER BY` clause can order results on any number of attributes in either ascending or descending order.
- The `LIMIT` clause allows paging of results. The `OFFSET` clause determines the first row of the result with the first row numbered as 1.

Some examples using the TPC-H schema follow. The database name for these examples is '`OrderDB`'.

Return all nations with their key and name:

```
SELECT OrderDB.Nation.n_nationkey, OrderDB.Nation.n_name
FROM   OrderDB.Nation;
```

Return the nations and their regions. Only return nations in the region name of 'AMERICA'. Note the use of table aliasing using AS.

```
SELECT N.n_nationkey, N.n_name, R.r_regionkey, R.r_name
FROM OrderDB.Nation as N, OrderDB.Region as R
WHERE N.n_regionkey = R.r_regionkey AND R.r_name = 'AMERICA';
```

Calculate the number of countries in each region. Only return a region and its country count if it has more than 4 countries in it. Order by regions with most countries.

```
SELECT R.r_regionkey, R.r_name, COUNT(N.n_nationkey)
FROM OrderDB.Nation as N, OrderDB.Region as R
WHERE N.n_regionkey = R.r_regionkey
GROUP BY R.r_regionkey, R.r_name
HAVING COUNT(N.n_nationkey) > 4
ORDER BY COUNT(N.n_nationkey) DESC;
```

6. INSERT Statement

The `INSERT` statement supported by UnityJDBC has the following syntax:

```
INSERT INTO <tbl_name> [(<col_name>,...)] VALUES <exprList>;
```

Specifying column names is optional. An example is below:

```
INSERT INTO mydb.Customer (id,firstname,lastname,street,city)
       VALUES (52,'Fred','Jones','Smith Lane', 'Chicago');
```

UnityJDBC also supports `INSERT INTO ... SELECT` with the following syntax:

```
INSERT INTO <tbl_name> [(<col_name>,...)] VALUES <exprList>
(SELECT <query>);
```

This is useful for storing query results into another table. Note that this table and all its column must exist in the XSpec or an error will be returned. Here is an example:

```
INSERT INTO emptydb.customer (SELECT * FROM mydb.customer);
```

7. UPDATE Statement

The `UPDATE` statement supported by UnityJDBC has the following syntax:

```
UPDATE <tbl_name> SET col1=expr1, col2=expr2, ... [WHERE <condition>];
```

An example is below:

```
UPDATE Employee SET salary=salary*1.10 WHERE age > 50;
```

8. DELETE Statement

The `DELETE` statement supported by UnityJDBC has the following syntax:

```
DELETE FROM <tbl_name> [WHERE <condition>];
```

An example is below:

```
DELETE FROM Employee WHERE salary > 100000;
```

9. By-Pass Statement

You can use methods to by-pass or flow through the driver to execute an untranslated query directly on a single source. In the `UnityStatement` class are these two methods:

```
public ResultSet executeByPassQuery(String dbName, String sql) throws SQLException
public int executeByPassUpdate(String dbName, String sql) throws SQLException
```

These methods will execute a query or update on a single source (given by name). The SQL statement provided is not parsed or validated and passed directly to the source driver. There is no overhead in this type of query as it is equivalent to invoking the source's JDBC driver directly.

Chapter 5. Supported JDBC Methods

1. Overview

UnityJDBC supports the majority of the methods in the `Driver`, `Connection`, `Statement`, `ResultSet`, and `ResultSetMetaData` interfaces. UnityJDBC supports the `PreparedStatement` interface but not the `CallableStatement` interface. UnityJDBC supports native updates using `INSERT`, `DELETE`, and `UPDATE`. It is also possible to use `INSERT INTO` to insert query results into another table. UnityJDBC does not support transactions across databases. Support for other JDBC methods is also limited by the underlying support of the JDBC driver for each data source. UnityJDBC requires a JDK of 1.6 or higher.

Chapter 6. Driver Specific Capabilities

1. MERGE Operator

Since UnityJDBC is designed to integrate data across sources, it has a special operator to make it easier to merge and compare data from different sources. This is important because data sources may contain inconsistent data that must be reconciled. The `MERGE` operator is a special operator that takes the result of two SQL queries (specified using a `SELECT` statement) and merges them. The `MERGE` syntax is as follows:

```
SELECT ...
MERGE [ON <joinCondition>] [EXTRACT [ALL] <matchList>] [FILTER <filterList>]
SELECT ...
```

- All clauses in the `MERGE` clause are optional. If no clauses are specified, `MERGE` performs a cross-product of the two input tables. The input tables are produced using regular `SELECT` expressions which can contain joins across databases.
- Attributes can be specified by their name in any of the clauses. However, to avoid naming conflicts, we recommend specifying attributes by their source query (`Q1` or `Q2`) and their column position indexed from 1 (`C1`, `C2`, `C3`). For example, to specify the 3rd attribute of query 2 use `Q2.C3`.
- The `ON <joinCondition>` specifies a `FULL OUTER JOIN` condition. Only equi-joins are supported. Typically, it is used to specify a join on two common attributes between different sources (such as ISBN or SSN) while keeping records from both sources regardless if they match with the other source.
- The `EXTRACT` clause and corresponding `<matchList>` is used to merge and compare attributes that are deemed equivalent in two sources. Specifying `[ALL]` will put all attributes in the final result even if they are not specified in the `<matchList>`, otherwise only attributes specified are in the final result. An element in the `<matchList>` may either be an attribute in the two input tables or the result of a `MATCH` function.
- `MATCH` functions are applied after the outer join or cross-product. A `MATCH` function takes two attributes in a single row and merges them using a defined function. Example functions include `AVG`, `MAX`, `MIN`, or `GROUP`. `MATCH` functions are also capable of recording the source where each attribute to allow for tracking data provenance. `MATCH` functions may be nested such as `SUM(A,SUM(B,C))`.
- The `FILTER` clause is similar to the `HAVING` clause in a regular `SELECT`. It allows filtering of records after the `MATCH` operator has been performed.
- You can have multiple `MERGE` operators such as:

  ```
  SELECT ...
  MERGE [ON <joinCondition>] [EXTRACT [ALL] <matchList>] [FILTER <filterList>]
  SELECT ...
  MERGE ...
  SELECT ...
  ...
  ```

1.1. MATCH Functions

The available `MATCH` functions are below. For illustration, consider that the match function will be applied to a single tuple with attributes `(C,A,B)` with values `(c,1,2)`. Attribute `A` comes from data source 1 and attribute `B` comes from data source 2. Attribute `C` is a common attribute used to relate (join) the records across sources. In the result column is shown the tuple produced after the match function is applied (and assuming the common attribute `C` is also requested in the `MATCH` list).

Table 6.1. MATCH Functions

Function	Return Type	Example	Result	Description
avg(x,y)	int or double depending on type of x	avg(A,B)	(c,1.5)	Returns average of two attributes.
choose(x,y,src)	Same as input	choose(A,B,2)	(c,2)	Always selects from source src (1 or 2). Attribute in other source is ignored.
chooseNN(x,y)	Same as input	chooseNN(A,B,1)	(c,1)	Always selects from source src (1 or 2) unless requested source value is NULL, then take from other source. Attribute in other source is ignored.
group(x,y)	ArrayList	group(A,B)	(c, [1,2])	Produce an array of two values in a single attribute.
groupref(x,y)	ArrayList	groupref(A,B)	(c, [(1,src1), (2,src2)])	Produce an array of AttributeValueSource objects.
max(x,y)	Same as input	max(A,B)	(c,2)	Returns maximum of two attributes.
maxref(x,y)	Attribute-ValueSource	maxref(A,B)	(c, (2,src2))	Returns maximum of two attributes and the source.
min(x,y)	Same as input	min(A,B)	(c,1)	Returns minimum of two attributes.
minref(x,y)	Attribute-ValueSource	minref(A,B)	(c, (1,src1))	Returns minimum of two attributes and the source.
sum(x,y)	int or double depending on type of x	sum(A,B)	(c,3)	Returns sum of two attributes.

There are two distinct types of `MATCH` functions. The regular type is similar to aggregate functions and can compute maximum, minimum, sum, and averages. The second type records the source of the value selected and have '`ref`" in their name. The result of these functions is a Java object called `AttributeValueSource`. This object contains two publicly accessible values: an `Object value` representing the value returned from the database and a `GQDatabaseRef source` object representing information on the database source where the attribute was obtained. In Java code, this attribute is retrieved by calling `getObject(columnId)` and then casting to an `AttributeValueSource` object. Knowing the source is valuable in many comparison situations when relating data across sources. Note that the regular functions do not track the source of their values, so it is recommended that you always use the `ref` functions when source is important. However, mixing the function types is allowed with the caveat that you may get source '`UNKNOWN`' returned by functions that do not track their source.

The group functions return an `ArrayList` of values that can be retrieved and manipulated using the `getObject(columnId)` call on the `ResultSet`.

1.2. MERGE Operator Examples

The following is some examples on how to use the MERGE operator. We will consider merging two employee databases. Each employee database has an Employee table with an id and a name. Not all employees are in both databases, and we want to create a master employee database. Some fields are different between the two databases. Our two databases are as follows:

Table 6.2. Employee Database 1 referred to as EmpDB1

empId	empName	job
1	Joe	Programmer
3	Steve	Accountant
4	Frank	CEO
6	Smith	Janitor

Table 6.3. Employee Database 2 referred to as EmpDB2

id	name	age
1	Joesph	30
2	Stacy	45
5	Carrie	36
6	<null>	27

Here is our query:

```
SELECT E.empId, E.empName, E.job  FROM EmpDB1.Employee AS E
MERGE ON Q1.C1=Q2.C2 EXTRACT chooseNN(Q1.C1,Q2.C1,1), chooseNN(Q1.C2,Q2.C2,2), Q1.C3, Q2.C3
SELECT Emp.id, Emp.name, Emp.age FROM EmpDB2.employee AS Emp
```

The `ON Q1.C1=Q2.C2` will perform a full outer join of the two inputs. The result before applying `MATCH` functions will be:

Table 6.4. MERGE Full Outer Join Result

empId	empName	job	id	name	age
1	Joe	Programmer	1	Joseph	30
<null>	<null>	<null>	2	Stacy	45
3	Steve	Accountant	<null>	<null>	<null>
4	Frank	CEO	<null>	<null>	<null>
<null>	<null>	<null>	5	Carrie	36
6	Smith	Janitor	6	<null>	27

In our `EXTRACT` clause, we have indicated we want to keep the `empId` attribute (`Q1.C1`) unless it is null then use the `id` attribute (`Q2.C1`), the job attribute (`Q1.C3`), the age attribute (`Q2.C3`), and combine the two name attributes using `chooseNN` with default source to 1 (always take the name in the second source (`name`) unless it is null). The result after applying `EXTRACT` is:

Table 6.5. Query Final Result

empId (derived)	name (derived)	job	age
1	*Joseph*	Programmer	30
2	Stacy	<null>	45
3	Steve	Accountant	<null>
4	Frank	CEO	<null>

empId (derived)	name (derived)	job	age
5	Carrie	<null>	36
6	Smith	Janitor	27

Note that the name of employee id 1 is '`Joseph`' instead of '`Joe`' because we indicated to the system we wanted to use the value in attribute `name` instead of `empName` whenever possible.

Let's modify the query so that we keep both names and record what source they come from. In that case, we will use `groupref` instead of `chooseNN`:

```
SELECT E.empId, E.empName, E.job  FROM EmpDB1.Employee AS E
MERGE ON Q1.C1=Q2.C2 EXTRACT chooseNN(Q1.C1,Q2.C1,1), groupref(Q1.C2,Q2.C2), Q1.C3, Q2.C3
SELECT Emp.id, Emp.name, Emp.age FROM EmpDB2.employee AS Emp
```

The result is:

Table 6.6. Query 2 Final Result

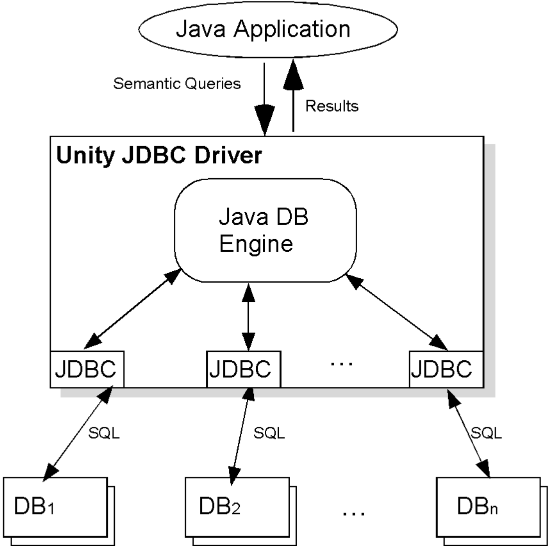
empId (derived)	name (derived)	job	age
1	[(Joe,src1), (Joseph,src2)]	Programmer	30
2	[(Stacy,src2)]	<null>	45
3	[(Steve,src1), (Steve,src2)]	Accountant	<null>
4	[(Frank,src1), (Frank, src2)]	CEO	<null>
5	[(Carrie,src2)]	<null>	36
6	[(Smith,src1), (Smith,src2)]	Janitor	27

Now as a programmer you have access to an `ArrayList` in the `name` attribute containing all the values from the sources and information on each source. You can then use that information to selectively display your required view.

Chapter 7. UnityJDBC Driver Internals

1. Overview

UnityJDBC contains an embedded database engine to join the results produced by executing queries on other JDBC-accessible sources. It requires a JDBC driver for each source to be accessed. The system diagram is below. The UnityJDBC architecture is the result of years of research and development and has been published in numerous technical and research publications.



2. Embedded Relational Database Engine

Embedded in UnityJDBC is a relational database engine and associated operators of selection, projection, grouping, ordering, and join. We have allowed public access to these classes which would allow users to by-pass the Unity parser, validator, and optimizer completely. In effect, you can build your own global query spanning data sources by combining these operators into an execution tree. In the distribution is a file called `ExampleEngine.java` which demonstrates how to use the engine to build an execution tree. Also in this file is an example on how you can have Unity parse but not execute a global query. UnityJDBC will return its global query and execution plan which you can later execute. This feature gives you the opportunity to modify the global execution plan before execution if desired. It also allows you to track the progress of a global query at the operator level.

Chapter 8. Limitations and Planned Features

1. Limitations

UnityJDBC supports global queries and single source updates. It does not currently support cross-database updates or transactions.

2. Planned Features

The following features are planned to be added in coming versions. Version 5.0 will be released in August 2012 with following versions released approximately one per year. If you have any feature requests, please e-mail support@unityjdbc.com.

Table 8.1. Planned Features in Coming Versions

Version	Feature Description
5.0	Non-correlated subqueries across multiple datasources
5.0	Transactional support

3. Feature List

The following table summarizes the features of UnityJDBC and the version where they were first introduced.

Table 8.2. Planned Features in Coming Versions

Version	Feature Description
1.0	Cross-database SQL queries for any JDBC source
1.0	Query by-pass
1.0	MERGE feature with MATCH functions
1.0	Embedded relational database engine
1.0	XSpec/SourceFile encryption
1.0	Support for Applets
1.0	Support for query results/databases larger than main memory
2.0	DataSource connections
2.0	Pooled connections
3.0	Prepared Statements
3.0	User-defined Functions
3.0	INSERT, DELETE, UPDATE on a single source
3.0	INSERT, DELETE, UPDATE across sources
3.0	INSERT INTO across sources

Version	Feature Description
4.0	Paging using LIMIT/OFFSET
4.0	Query and ResultSet caching
4.0	Universal dialect and function translation (support for DBMS missing, non-standard functions)
4.0	Single source subqueries
5.0	Subqueries across data sources
5.0	Transactional support for a single source

4. Contacts and Support

Please contact support@unityjdbc.com or post a message in the forums at http://www.unityjdbc.com/support.html if you encounter any bugs, issues, or have feature requests.

This document is Copyright 2011 by UnityJDBC. All rights reserved.