db2db
A Database Migration Tool

December, 2004

Chen, Fu-Hsiang

School of Electrical Engineering and Computer Science
Oregon State University

Abstract

We developed a tool that migrates the database schema and the data from one database to another. As DBMSs store the same data by using slightly different data types, one cannot simply copy all the tables and data from the source DBMS to the target DBMS. When our tool, `db2db`, migrates a database, from Oracle DBMS to PostgreSQL DBMS, for example, it converts such data types as `CLOB`, `NUMBER`, `VARCHAR2` of Oracle to data types `text`, `numberic`, and `varchar`, respectively, of postgreSQL. The program uses JDBC type-2 or type-4 drivers that work with TCP/IP sockets, which allows a user to migrate databases over a network. As a JDBC type 2 driver can connect to a DBMS that support ODBC, `db2db` can migrate a database from any DBMSs that support JDBC or ODBC.

Index

1. Introduction

In order to migrate a database from one database management system (DBMS) to another, we must copy the *database schema* and the *data* from the source database to the target database. There are many DBMSs in the market, such as Oracle, IBM DB2, and Microsoft SQL Server, and they store the same data by using slightly different data types. Therefore, we cannot simply copy all the tables and data from the source DBMS to the target DBMS.

In this project, we developed `db2db`, a tool to migrate the database schema and the data among DBMSs. This tool maps the data types for the source DBMS to those for the target DBMS, creates the schema in the target DBMS, and then copies all the rows in each table from the source database to the target database.

The main reason we developed this tool was to migrate an Oracle database to a PostgreSQL database. PostgreSQL is a free open-source *object-relational* DBMS (ORDBMS) created from the Postgres database management system developed at U.C. Berkley. It is an ideal platform for Geographical Informational System (GIS) applications as it supports *geometric objects*. PostgreSQL is used by two other open source projects: the PostGIS and MapServer. PostGIS adds to PostgreSQL support for geographic objects, and MapServer allows a user to develop Web-enabled GIS applications.

We implemented `db2db` by using Java and Java Database Connectivity (JDBC) API. JDBC is a standardized database interface for a Java program to perform database operations. As JDBC is platform independent, a user can execute `db2db` either on a Linux machine or a Windows machine. The user can also migrate a database over a network that supports TCP/IP sockets. Most database vendors offer good documentation and support for JDBC drivers.

A schema of a database defines the *tables*, the *columns* in each table, the *data type* of each column, and the *relationship types* among the tables. In order to migrate a schema, we must convert the column data types in the source database to the compatible ones in the target DBMS. Once the column data types are converted, the `CREATE TABLE` statements can be executed on the target database to create the tables.

After the database schema is migrated to the target database, data can be migrated. For this purpose, we generate an `INSERT INTO` statement for each row in a table and execute the statement on the target database.

We tested `db2db` with the database for Biotics 4 developed by NatureServe (`http://www.natureserve.org/`). Biotics 4 is a client-server database application for biodiversity data management. The Biotics database stores tabular and geographical information on species distributions. The database consists of 695 tables and is implemented as an Oracle database. The major purpose of `db2db` was, in fact, to migrate the Biotics 4 database to a PostgresSQL database.

Before determining the mapping rules for the column data types, some tests were conducted on two existing migration tools, Data transfer service (DTS) of Microsoft SQL Server and pgAdmin II. pgAdmin is a front end tool to access PostgresSQL database. The purpose of the tests was to see how these tools map data types from one DBMS to another and to use the results as a reference in determining the mapping rules for this project. In Section 2, we will provide details about database migration. We discuss the tests of data type mappings employed by SQL Server DTS and PgAdmin II in Section 3. Section 4 covers the implementation details of `db2db`, and in Section 5, we describe how to use `db2db`.

2. Overview of Database Migration

There are several issues in migrating a database:

1. Different DBMSs use different names for the same type of data. For example, Oracle uses data type `NUMBER` to represent every type of numbers, while MS SQL Server provides `int`, `float` and `long` to represent numbers.

2. Although there are some SQL data type standard, e.g., SQL 92, most DMBS venders support some non-standard data types. In order to migrate data from one database system to another, it is necessary to map the data types used by the source DBMS to those used by the target DBMS.

3. There are also data types that are not available for the target DBMS. For example, `CLOB` (Character Large Object) is a data type supported by Oracle DBMS. Oracle uses this data type to store a large amount of text. When `CLOB` data is retrieved from an Oracle database, it returns a `CLOB` object that has to be saved in a text file or converted to a `string` before it can be used.

4. Different DBMSs support different precisions of numbers. The precision of a

number in Oracle can be up to 38 digits, while the default `numeric` type in PostgreSQL is 30 digits. In order to migrate a 38-digit number from an Oracle database to a PostgreSQL database, a change in precision may be necessary. `db2db` does not support the mapping of precision yet.

In `db2db`, data type of a source is mapped to a standard SQL data type in the target database as much as possible to increase compatibility with other DBMSs However, this is not always possible, and more specialized data-type mapping may be needed.

For example, in `db2db`, the data type mapping rules employed to migrate an Oracle database to a PostgreSQL database are as follows.

1. A data type is mapped to a similar data type from the target DBMS. For example, Oracle type `NUMBER` is mapped to SQL data type `numeric`, although their precisions differ.

2. `varchar` of Oracle supports a very large text. If the size of a varchar exceeds the limit supported by the target DBMS, this type is mapped to `text` in PostgreSQL.

The mapping rules of data types from Oracle to PostgreSQL are summarized in Table 2.1.

Type and Size in Oracle	Type and Size in PostgreSQL	Notes
NUMBER	NUMERIC	
VARCHAR2 (size < 4000)	VARCHAR	
VARCHAR2 (size > 4000)	TEXT	
CLOB	TEXT	
DATE	TIMESTAMP	
LONG	BIGINT/INT8	Only used once in Biotics.

Table 2.1: Data type mapping rules for migrating a database from Oracle to PostgreSQL.

Figure 2.1 summarizes the migration steps taken by `db2db`.

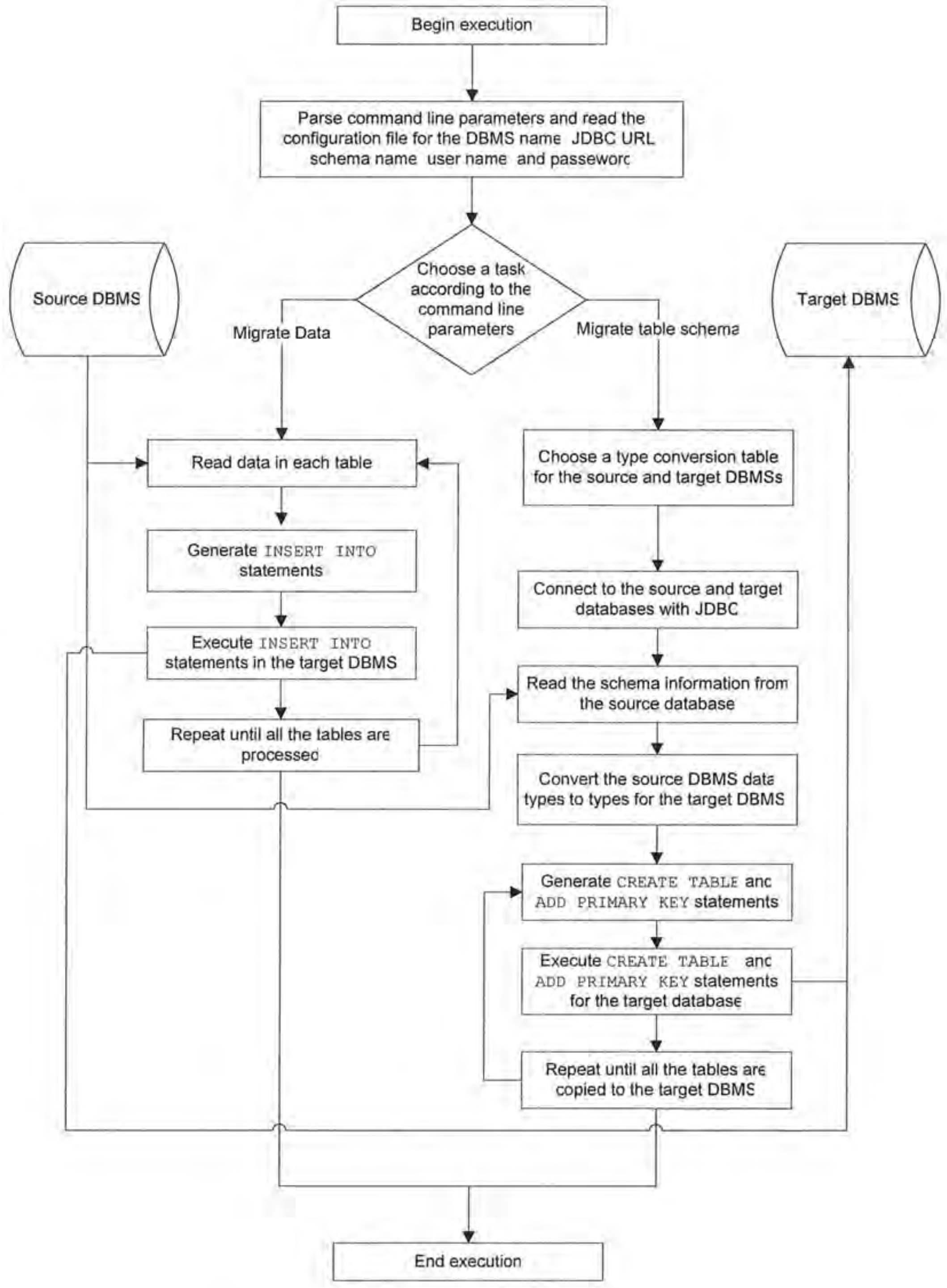

Begin execution
Parse command line parameters and read the configuration file for the DBMS name JDBC URL schema name user name and passeworc
Choose a task according to the command line parameters
Migrate Data
Migrate table schema
Source DBMS
Target DBMS
Read data in each table
Generate INSERT INTO statements
Execute INSERT INTO statements in the target DBMS
Repeat until all the tables are processed
Choose a type conversion table for the source and target DBMSs
Connect to the source and target databases with JDBC
Read the schema information from the source database
Convert the source DBMS data types to types for the target DBMS
Generate CREATE TABLE and ADD PRIMARY KEY statements
Execute CREATE TABLE and ADD PRIMARY KEY statements for the target database
Repeat until all the tables are copied to the target DBMS
End execution

Figure 2.1: Database migration steps taken by `db2db`.

2.1 Schema and Data Migration

The schema and data are migrated according to the following steps:

Migrating Tables

1. Select the schema for the source database to be migrated.
2. Get all the table names in the schema.
3. Get the names, types, sizes, and nullabilities of all the columns in each table.
4. Generate for each table the `CREATE TABLE` statement according to the data type mapping rules.
5. Execute the `CREATE TABLE` statements in the target database.

Adding Primary Keys

1. Read the primary key metadata from the Oracle server.
2. Generate the `ALTER TABLE` statement to add the primary key.
3. Execute the `ALTER TABLE` statements in the target database.

Migrating Data

1. Retrieve all the rows in each table in the source database.
2. Generate an `INSERT INTO` statement for each row.
3. Execute the `INSERT INTO` statement in the target database.

2.2 JDBC Driver

There are four different types of JDBC drivers, and a type 4 driver is used for migrating a database from Oracle to PostgreSQL. Type 4 driver runs as a native Java program and uses TCP/IP sockets to communicate with database servers, while other three types of

JDBC drivers require some middleware or an ODBC bridge to connect to a DBMS.

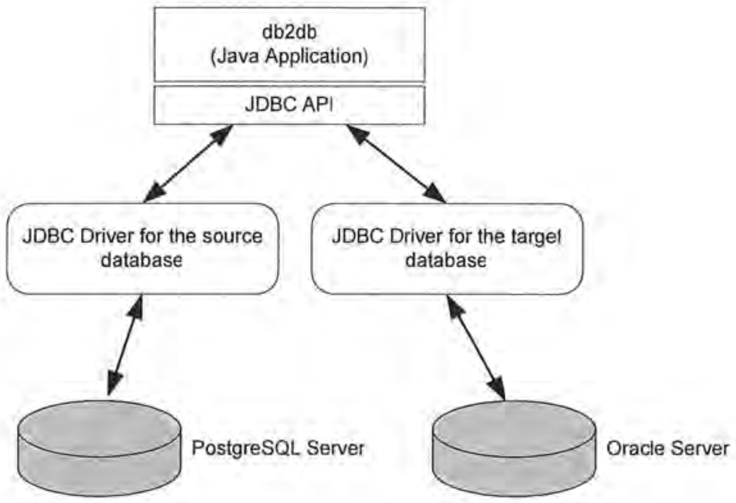

db2db
(Java Application)
JDBC API
JDBC Driver for the source database
JDBC Driver for the target database
PostgreSQL Server
Oracle Server

Figure 2.2: Type 4 JDBC connection.

Type 4 drivers used in this project are provided by Oracle and PostgreSQL development team.

1. **Oracle JDBC driver**

   Oracle9i 9.2.0.3 JDBC Drivers: ojdbc14.jar

   http://otn.oracle.com/software/tech/java/sqlj_jdbc/htdocs/jdbc9201.html

2. **PSQL JDBC driver**

   JDBC driver for PostgreSQL 7.4

   http://jdbc.postgresql.org/download/pg74.214.jdbc3.jar

3. Data Type Mappings

We conducted several tests on the data-type mapping rules employed by SQL Server DTS and pgAdmin II. MS SQL Server provides DTS to migrate a database from Oracle to MS SQL Server. For a PostgresSQL database, the migration wizard in pgAdmin II can be used to migrate the tables and the data. Each test was conducted as follows:

1. Several tables containing columns of different data types and sizes were created in an Oracle database.

2. The data in these tables were then migrated to an MS SQL Server or PostgreSQL database with SQL Server DTS or pgAdmin II respectively.

The results of these tests are summarized in table 3.1

Oracle	JDBC	MS SQL Server	PostgreSQL
`BLOB`	`java.sql.Blob`	`image` (length 16)	N/A
`CLOB`	`java.sql.Clob`	`Text` (length 16)	`text`
`NCLOB`	`No support`	`Text` (length 16)	`char` 4000)
`CHAR(1000)`	`java.lang.String`	`Char` (1000)	`char` (1000)
`VARCHAR2(2000)`	`java.lang.String`	`varchar` (length 2000)	`varchar` (2000)
`DATE`	`java.sql.Timestamp`	`datetime` (length 8)	`timestamp`
`FLOAT(10)`	`Double`	`float` (length 8)	`float8`
`LONG`	`n/a`	`Text` (length 16)	`text`
`RAW(1000)`	`byte[]`	`varbinary` (1000)	N/A
`NUMBER(15, 2)`	`java.math.BigDecimal`	`numeric` (length 9)	`numeric` (15, 2)
`NUMBER(20, 1)`	`java.math.BigDecimal`	`numeric` (length 13)	`numeric` (20, 1)
`NUMBER(38, 5)`	`java.math.BigDecimal`	`numeric` (length 17)	`text`
`NUMBER(10, 21)`	`java.math.BigDecimal`	range must be 0-10	range must be 0-10
`NUMBER(38, 19)`	`java.math.BigDecimal`	`numeric` (length 17)	`numeric` (38, 19)
`NUMBER(38,-2)`	`java.math.BigDecimal`	range must be 0-38	`text`

Table 3.1: Data-type mapping performed by MS SQL Server DTS and pgAdmin II.

4. Implementation

In this section, we discuss the implementation details of `ora2pqsql`, including the use of JDBC classes and their methods. These methods are used for metadata retrieval and SQL statement construction. As `db2db` is an object-oriented program, we also explain its classes and the interactions among them.

4.1 Connecting to a Database with JDBC

In order to communicate with a DBMS, a JDBC *connection* to the DMBS need be created.

1. The JDBC driver located in a directory specified by `classpath` can be loaded by class `DriverManager` as follows

```
DriverManager.registerDriver(new oracle.jdbc.driver.OracleDriver());
DriverManager.registerDriver(new org.postgresql.Driver())
```

2. Once the driver is loaded, a *connection* to each DBMS can be created as:

```
DriverManager.getConnection(String JDBC_URL, String UserName,
                                                    String Password);
```

The database is identified by a URL.

```
    jdbc:[drivertype]:[@][database]
```

The following URLs are used in this project:

```
    jdbc:oracle:thin:@khong.een.orst.edu:1521:khong
    jdbc:postgresql://ganga.een.orst.edu/biotics
```

The second parameter `drivertype` is "`postgresql`" for a PostgreSQL database, and "`oracle`" for an Oracle database. The third parameter identifies a database. The syntax for this paremeter varies among DBMSs, For a PorsgreSQL database, the formats are *`//hostname/databasename`* or *`//hostname:portnumber/databasename`*. For an Oracle database, the format is *`hostname:port:SID`*.

4.2 Retrieving Metadata

There are two types of metadata in a database, namely the *database metadata* and the `ResultSet` *metadata*.

Database Metadata Retrieving

JDBC allows a programmer to access the metadata about the schemas, tables, and columns as well as the data stored in a database. The `DatabaseMetaData` interface of JDBC includes over 150 methods for retrieving information about the data source, features supported, and available data types. Such information is called *database metadata.* In this project, we use the following JDBC methods to access the database metadata.

```
public ResultSet getSchemas()
```

This method retrieves the names of the schema available in the database, returning a `ResultSet` object which includes a description of the schema in each row. The rows are ordered by the schema names.

```
public ResultSet getTables(String catalog, String schemaPattern,
                    String tableNamePattern, String[] types)
```

This method returns a `ResultSet` containing the metadata on the selected tables. The user can select tables by specifying the catalog, schema, table name, and type.

Arguments:

`catalog` - a catalog name, `null` if a catalog does not exist.
`schemaPattern` - a pattern for a schema
`tableNamePattern` - a pattern for tables or views.
`types` - a list of table types (`null` for all types)

```
public ResultSet getColumns(String catalog,String schemaPattern,
                 String tableNamePattern, String columnNamePattern)
```

This method returns the descriptions of table columns. Only the column descriptions for the columns that match the given pattern for catalog, a schema, tables and column names are returned.

Arguments:

`catalog` - a catalog name, `null` if a catalog is not used in the database.
`schemaPattern` - a pattern for a schema
`tableNamePattern` - a pattern for table
`columnNamePattern` - a pattern for column names

The `getColumns()` method in the JDBC driver provided by Oracle returns the following column metadata:

`TABLE_CAT: String`	Catalog (may be null)
`TABLE_SCHEM: String`	Schema (may be null)
`TABLE_NAME: String`	Table name
`COLUMN_NAME: String`	Column name
`DATA_TYPE: int`	SQL type from `java.sql.Types`
`TYPE_NAME: String`	Data source dependent type name
`COLUMN_SIZE: int`	Column size. For the `char` or `date` type, it is the maximum number of characters, and for numeric or decimal type, it is the precision.
`NULLABLE: int`	is NULL allowed ?

`IS_NULLABLE`	"NO" means the column does not allow NULL values. "YES" means the column allows NULL values. An empty string means nobody knows.

Table 4.1: Column metadata used in this project.

Retrieving Metadata as aResultset

A `ResultSet` is an object that stores the data retrieved with a database query, and JDBC provides the `ResultSetMetaData` interface to retrieve metadata as a `ResultSet`. When method `getResultSetMetaData()` is called for a `ResultSet`, it returns a `ResultSetMetaData` object describing the columns. The information on each column includes its name, type display size, and so forth. The following methods of `ResultSetMetaData` are used in `db2db`:

Method Name	Description
`getColumnCount()`	Get the number of columns in a `ResultSet`
`getColumnName(int col)`	Get the name of column `col`
`int getColumnType(int col)`	Get the type of column `col`

4.3 Handling Special Characters

Special characters that have special meanings for a DBMS or Java need be properly handled. For example, a single quote (') is used by SQL to quote a string, and hence, if it occurs in a data string, its special meaning need to be suppressed. Every DBMS has rules for handling special characters. For example, if there is a single quote (') in a character string, we need to add one more ' to escape it. The flowing table lists the notations for special characters:

Literal	Meaning
''	Empty string

''''	A pair of single quote
\t	Tab
\r	Carriage return
\n	Newline
\\	Backslash

4.4 Using Quote symbols

PostgreSQL uses single quote symbols to quote strings. However, single quotes may cause problems.

1. A number does not need be quoted as delimiters. If the number to be inserted is `null`, single quotes are still not needed. We just need to leave the space blank, separated it from the next value by a comma (`,`).

2. If the column data-type is a `text` or `varchar` and if the data is `null`, one can use ' ' (two single quotes), `null`, or '`null`' to represent the null value.

3. If the data type is `timestamp`, and the value is not `null`, single quotes are needed, as '`22/09/74 00:00:00`'. However, if the timestamp is `null`, one can use `null` in the SQL statement, but not '`null`' or ' '.

4.5 Classes Diagram of db2db

The UML class diagram is shown in Figure 4.1:

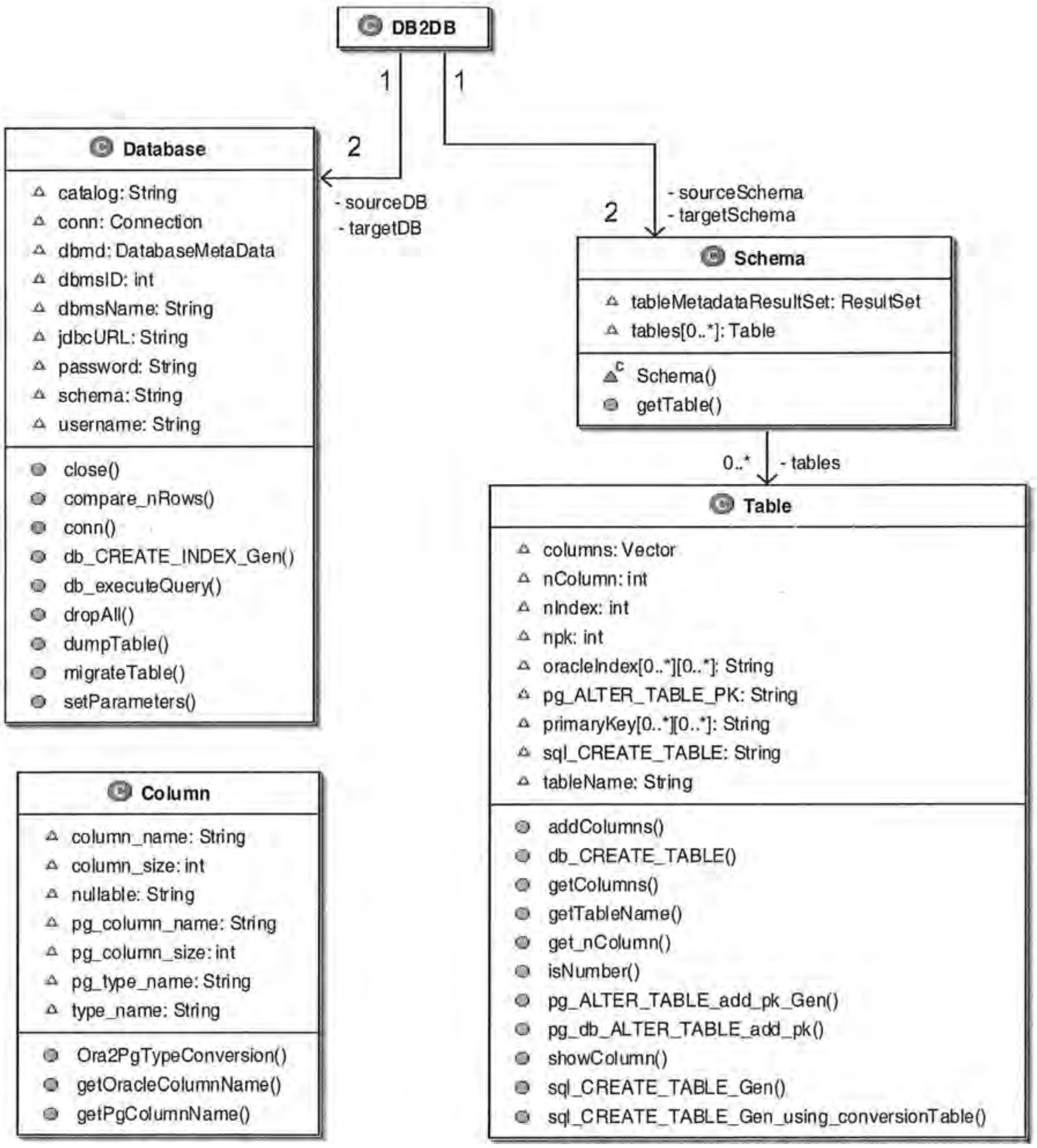


Figure 4.1: UML class diagram of `db2db`.

4.6 Classes and Methods

`public class DB2DB`

Class `DB2DB` is the main class in the program. It is used to parse the command line parameters, to read a configuration file, to choose a column data-type conversion table, and to create connections to the source and the target databases. The column data-type conversion table is selected by the combination of the source and target databases. `DB2DB` contains the following methods:

- `private static void parseParameters(String[] args)` parses the command line parameters.
- `private static void getCnvTable()` uses the source and target DBMS names to choose the conversion table for column data-type mapping.
- `static void connectDB()` creates `Connection`s for the source and target databases and retrieves the database metadata from the source database.
- `private static void readSchema()` creates an instance of `Schema` and sets its reference in `sourceSchema`. The *schema* and the *table metadata* are retrieved by the constructor of `Schema` from the source DBMS.

`public class Database`

This class is used to create an object containing the information about a DBMS. It provides the following methods:

- `public void conn()` establishes JDBC connections to the source and target DBMSs.

- `public void close()` closes the connections to the DBMSs.

- `migrateTable(String tableName)` migrates data in the table designated by `tableName` from the source database to the target database.

- `public void compare_nRows (String tableName, Connection sourceDB_con, Connection targetDB_con)` compares the row numbers of all the tables in the source and target databases. This method provides a simple mechanism to check whether the database has been migrated correctly or not.

- `public void dropAll(String tableName)` deletes all the rows in the table designated by `tableName`. This method is called before copying data from the source database to the target database to prevent duplicated rows from being copied to the target table.

- `public void db_executeQuery(String queryString)` executes a query string on the target database.

`public class Schema`

When a `Schema` object is created, its constructor retrieves the *metadata* on tables in a `ResultSet`, which contains a list of the table names. For each table, the `Schema` creates a `Table` and stores it in `Table tables[]`. Class `Schema` provides the following methods.

- `private static int getMetaDataResultSetRowCount (ResultSet DatabaseMetadataResultSet)` counts the number of rows in a `DatabaseMetadataResultSet`.

- `public Table[] getTable()` returns the reference to all the `Table` objects.

`public class Table`

A `Table` contains information about a table. Class `Table` provides the following methods.

- `public void addColumns()` retrieves the *column metadata* for the current `Table`, that `Columns` in `Vector column[]`.

- `public String getTableName()` returns the table name of the `Table`.

- `public Column[] getColumns()` returns the `Column` vector, which contains all the `columns` of the table.

- `public void showColumn()` prints the name of all the `columns` in the table.

- `public void sql_CREATE_TABLE_Gen()` generates the `CREATE TABLE` SQL statements for the current `Table`.

- `public void db_CreateTable(String pg_CREATE_TABLE)` executes a `CREATE TABLE` statement in the target DBMS.

- `protected void getPrimaryKey()` retrieves the *primary-key metadata* for an Oracle table and store it in `String [][] primaryKey` in the `Table`.

- `public void sql_ALTER_TABLE_add_pk_Gen()` generates `ALTER TABLE ADD PRIMARY KEY` SQL statement for the current `Table`.

- `public void db_ALTER_TABLE_add_pk (String sql_ALTER_TABLE_PK)` executes the `ALTER TABLE ADD PRIMARY KEY` statement in the target DBMS.

`Public class Column`

This class is used to create a `Column` containing the *metadata* about a column, including `string column_name`, `string nullable`, `string type_name`, and `int column_size`. The metadata are used to generate a `CREATE TABLE` statement.

`public class Msg`

This class is used to log execution and error messages to a file and to print time-stamped messages on the console.

- `public static void cout(Object msg)` prints a string form of `msg` on the console with a timestamp.

- `public static void cerr(Object err)` prints a string form of `err` on the console with a timestamp.

- `public static void log (String log)` writes to `exec_log.txt` the string `log` with a timestamp.

- `public static void debug(Object msg)` prints debugging information in `msg` on the console.

- `public static Object[] readFile(String inputFile)` reads every line in a file that lists the name of the tables to be migrated, stores each line in a vector, and returns the vector.

`public class Ora2Pg_string`

This class modifies SQL data string so that the effects of special characters for SQL are suppressed.

- `public String replace(String OriginalString, String toBeReplaced, String newString)` replaces every occurrence of `toBeReplaced` in `OriginalString` with `newString`.

`public class Convert`

This class is used to create a conversion table for a column-data type mapping. How to create a conversion table is described in the next section. This class has the following method:

- `public String convert(String inType, int inSize, int inSize2)` takes the original data type and outputs the converted data type as a string. How to use this method is described in the next section.

`public class CnvOra2Pgsql extends Convert`

As a sub class of `Convert`, this class contains the data type mapping rules to convert column data types for an Oracle database to those for a PostgreSQL database.

4.7 Testing db2db

We tested `db2db` in the following configuration:

Configuration 1:
Source: Biotics database on an Oracle server
Target: PostgreSQL DBMS
Platform: Windows to Linux
Network type: LAN
Driver type: type 4

Configuration 2:
Source: Biotics database on an Oracle server
Target: PostgreSQL DBMS
Platform: Windows to Linux
Network type: a remote site on the internet
Driver type: type 4

Configuration 3:
Source: Fishbase database on MS Access
Target: PostgreSQL DBMS
Platform: Windows to Linux
Network type: LAN
Driver type: type 2

5. Using DB2DB to Migrate Database

5.1 Compiling db2db

As `db2db` is written in Java, it can be compiled and executed on any operating system that supports Java virtual machine version 1.4 or higher. In order to connect to a DBMS, a JDBC driver for it need be specified in the `classpath`. The Oracle driver is `ojdbc14.jar`, and the one for PostgreSQL is `pg74jdbc3.jar`. `db2db` can be compiled on Unix with the following command:

```
javac -classpath .:pg74jdbc3.jar:ojdbc14.jar DB2DB.java
```

Shell scripts `build.sh` is provided to compile `db2db`.

5.2 Running db2db

After `db2db` is compiled, the user can execute it on Unix with the following command.

```
java -Xmx400M -cp .;pg74jdbc3.jar;ojdbc14.jar ora2pgSQL [options]
```

Shell scripts `db2db.sh` is provided to execute `db2db`. When executing the code, the option for the class path is `-cp` instead of `-classpath`. Because some tables in the Biotics database are very large, we need to specify option `-Xmx400M` to reserve 400 MB of virtual memory for the Java virtual machine.

Furthermore, the user can provide several options to specify the details of the task to be performed. The syntax of a `db2db` command line is one of the following:

```
db2db.sh -c configuration_file -s [-d]
db2db.sh -c configuration_file -t tablelist_file -d
db2db.sh -c configuration_file [table1 table2 …]
```

```
db2db.sh -c configuration_file -n [-d]
db2db.sh -c configuration_file -i [-d]
```

- `-c` *`configuration_file`* specifies the configuration file for the JDBC connections
- `-s` migrates a schema from the source DBMS to the target DBMS.
- `-i` reads the metadata on the indexes from the Oracle database, generates `CREATE INDEX` statements, and executes them in the PostgreSQL DBMS.
- `-t` *`table_list_file`* migrates all the data in every Oracle tables to the PostgreSQL database.
- `-n` Compares the number of rows for each table in the source database and the target database.
- `-d` turns on the debug mode. `db2db` prints more debugging information on the console window.

In executing `db2db`, the user can provide the source and the target databases in a *configuration file* with the `-c` command line parameter. The configuration file looks as follows:

```
<JDBCConnections>
        <source_db
            dbms = "DBMS name"
            jdbcUrl="jdbc:oracle:thin:@149.168.72.41:1521:biot"
            schema = "BIOTICS_USER"
            catalog = "null"
            username="username"
            password="password"
        />
       <target_db
            dbms = "DBMS name"
            jdbcUrl="jdbc:postgresql://ganga/biotics_5_chenfu"
            catalog = "null"
            schema = "BIOTICS_USER"
            catalog = "null"
            username="username "
            password="password"
        />
</JDBCConnections>
```

The `source_db` and `target_db` tags are used to identify the source DBMS and target DBMS:

1. `dbms` identifies the name of the DBMS system. The DBMS name is case sensitive. Currently, the following DBMS names are accepted by `db2db`: `Oracle`, `MSSQL`, `PgSQL`, `MySQL`, and `ODBC`.

2. `jdbcURL` is the address of the DBMS.

3. *`catalog`* and *`schema`* are used by the DBMS to identify a database in the DBMS. `catalog` is used by Microsoft SQL Server and `schema` is used by Oracle. PostgreSQL does not use `catalog` and `schema`, instead, a PostgreSQL database is identified by its JDBC connection URL. For example, `jdbc:postgresql://ganga.een.orst.edu/ztest` will create a connection to `ganga.een.orst.edu` and use the database `ztest`. If the DBMS does not need `schema` or `catalog` information to identify a database, `"null"` is used in the configuration file.

4. *`username`* and *`password`* are the login name and password of the user accessing a DBMS.

5.3 Creating a Column Data-Type Conversation Table

Class `CnvOra2Pgsql` shown below maps the column data types used by an Oracle database to those for a PostgreSQL database.

```
public class CnvOra2Pgsql extends Convert {  // from Oracle to PostgreSQL
  CnvOra2Pgsql() {
    CnvEntry[] cnvTable_temp = {
      new CnvEntry("NUMBER",   1, 30, "numeric"),
      new CnvEntry("VARCHAR2", 1, 40, "char"),
      new CnvEntry("CHAR", 1, 40, "char"),
      new CnvEntry("VARCHAR2", 1, 4000, "text"),
      new CnvEntry("VARCHAR2", 1, 8000, "text"),
      new CnvEntry("CLOB", 0 , -1, "text"),
      new CnvEntry("LONG", 0, -1, "bigint"),
      new CnvEntry("DATE", 0, 20, "timestamp")
    };

    cnvTable = cnvTable_temp;
  }
```

Each conversion rule is represented as a `CnvEntry` object in `CnvEntry[] cnvTable_temp` in the default constructor. The constructor of class `CnvEntry` takes the following four parameters:

`String inType`: the column type in the source database.

`int cnvType`: the size parameters associated with the `inType`. Value `0` indicates that the `inType` does not have a size parameter, applicable to type `CLOB`, `DATE` and `LONG`. Value `1` indicates that there is only one size parameter associated with the `inType`, applicable to type `VARCHAR` and `CHAR`.

`int inMaxSize`: the maximum size of the input data to which this rule is applicable. If the `inType` does not have size information, `inMaxSize` is set to `-1`.

`String outType`: the converted data type to be used in the target database.

A new conversion rules can be derived from class `Convert`.

5.4 Error Logging

`DB2DB` logs messages to a file `db2db.log` with a timestamp on each entry, to provide information on its execution and error conditions encountered.

5.5 Using RmiJdbc

RmiJdbc (http://rmijdbc.objectweb.org) is a type 4 JDBC driver that allows a user to connect to an ODBC database on a remote network by using the Java remote method invocation (RMI) interface. RmiJdbc redirects database queries to Sun's JDBC-ODBC Bridge Driver. This bridge driver queries the database via ODBC and returns the result to RmiJdbc. Sun's JDBC-ODBC Bridge Driver is included in a Java JVM distribution.

Before using RmiJdbc, we must start a RmiJdbc server instance as follows.

```
java -jar RmiJdbc.jar [-noreg] [-port regportnum] [-lp portnum] [-sm]
[-ssl] [-passwd passwd] [driverList]
```

`-noreg`

means you launch the RmiJdbc server with an external rmiregistry

`-port regportnum`

specifies the rmiregistry port (optional)

`-lp portnum`

specifies the listener port for remote objects (optional)

`-sm`

uses the standard RMI SecurityManager

`-ssl`

uses RmiJdbc on top of SSL.

`-passwd`

defines an administrative password, used by org.objectweb.rmijdbc.RJAdmin for administrative operations

`-driverList`

lists of JDBC Driver classes available on your server

RmiJdbc.jar is located in the source distribution, under the `dist/lib` directory.

Once the server starts, a remote JDBC application can access your ODBC database. The JDBC URL used for RmiJdbc is

```
jdbc:rmi://<rmihostname[:port]>/<jdbc-url>
```

`rmihostname` is the host name or IP address of the machine where the RmiJdbc server resides.

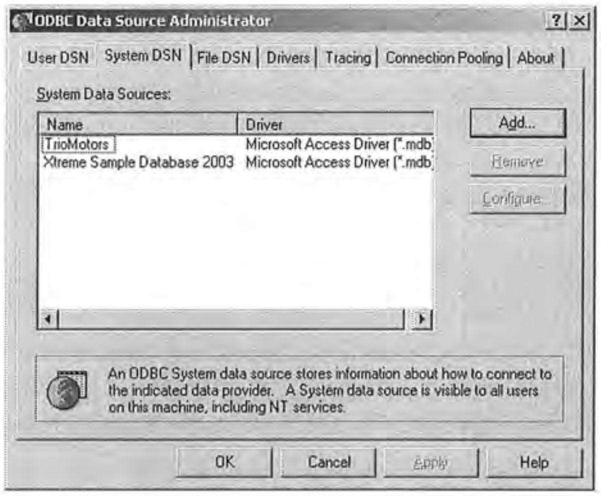
`port` is the port number for the RMI registry and the default port is 1099

`jdbc-url` is the location of the database.

5.6 Setting up a ODBC data source for an Access Database

An ODBC data source for an Access database can be created as follows.

1. In the `Configuration` Panel, go to `Administrative Tools` > `Data Sources (ODBC)`.



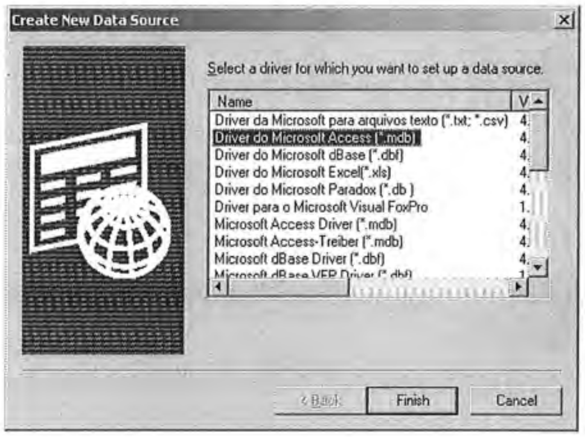
System Data Sources:

Name	Driver
TrioMotors	Microsoft Access Driver (*.mdb)
Xtreme Sample Database 2003	Microsoft Access Driver (*.mdb)

An ODBC System data source stores information about how to connect to the indicated data provider. A System data source is visible to all users on this machine, including NT services.

2. Click `System DSN` > `Add`.

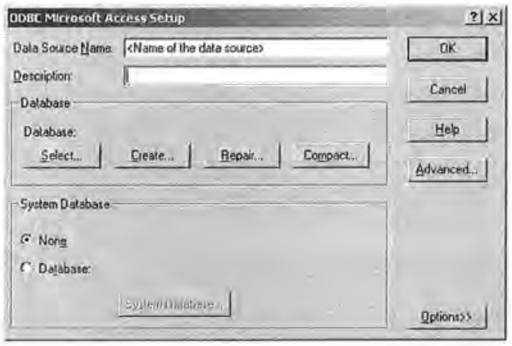
3. Choose `Microsoft Access Driver (*.mdb).`



Select a driver for which you want to set up a data source.

Name	V
Driver da Microsoft para arquivos texto (*.txt; *.csv)	4.
Driver do Microsoft Access (*.mdb)	4.
Driver do Microsoft dBase (*.dbf)	4.
Driver do Microsoft Excel(*.xls)	4.
Driver do Microsoft Paradox (*.db)	4.
Driver para o Microsoft Visual FoxPro	1.
Microsoft Access Driver (*.mdb)	4.
Microsoft Access-Treiber (*.mdb)	4.
Microsoft dBase Driver (*.dbf)	4.
Microsoft dBase VFP Driver (*.dbf)	1.

4. In the `Data Source Name` text box, give a name to the Access Database. This name is the catalog name. Click `Select...` and browse for the Access `mdb` file.


ODBC Microsoft Access Setup

Data Source Name: <Name of the data source>

Description:

System Database: (● None) Database:

5. Setup the user name and password in the `Advanced...`. Click `OK` when done.

5.8 db2db Configuration File

The following configuration file can be used to migrate fishbase database from Acccess to PostgreSQL.

```
<JDBCConnections>

  <source_db
    dbms = "ODBC"
    jdbcUrl="jdbc:rmi://khong.een.orst.edu/jdbc:odbc:fbapp"
    schema = "null"
    catalog = "E:\\Fishbase\\fbapp"
    username="username"
    password="password"
  />

  <target_db
    dbms = "PgSQL"
    jdbcUrl="jdbc:postgresql://ganga/fishbase"
    schema = "null"
    catalog = "null"
    username="username"
    password="password"
  />

</JDBCConnections>
```

6. Conclusion

We developed `db2db` for migrating the database schema and the data from one database to another. It uses JDBC type-2 or type-4 drivers to connect to the source and target DBMSs and allows the database to be migrated over a network. The data type mapping rules for column data-type can be chosen for each specific combination of the source and target DBMSs. The databases we migrated are Biotics and Fishbase. Both of these databases are large database and could not be migrated with such a tool as Microsoft DTS or PgAdminII. `db2db` successfully migrated Biotics and Fishbase. The next step for us is to test `db2db` with other DBMSs and databases and improve its applicability.

7. Reference

1. George Reese, "Database Programming with JDBC and Java", O'Reilly, August 2000.

2. Oracle, "Oracle9i Database Release 2 User, Administrator, and Developer Guides", http://www.oracle.com/technology/documentation/oracle9i.html

3. "PostgreSQL Developers Guide", http://www.postgresql.org/docs/

4. "PostgreSQL Programmers Guide", http://www.postgresql.org/docs/

5. Van Der Lans R., "The SQL Guide to Oracle", Addison-Wesley Professional, December 1991.