A Project Report

On

Querying Heterogeneous Databases

BY

Aditya Lohana

2016A7PS0048H

and

Keval Morabia

2015A7PS0143H

Under the supervision of

Prof. R. Gururaj

SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS OF

BITS F366: LAB ORIENTED PROJECT

BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE PILANI (RAJASTHAN)

HYDERABAD CAMPUS

(NOVEMBER 2018)

CONTENTS

Acknowledgements

I would like to express my deepest appreciation to all those who provided me the opportunity to complete this project. Some special thanks go to my teammate, Aditya Lohana (and Keval Morabia for vice versa) who gave feedback and suggestion about the task “Querying Heterogeneous Databases”. Last but not least, many thanks go to the head of the project, Prof. R. Gururaj who has invested his full effort in guiding the team in achieving the goal.

Introduction

Generally, organisations and institutions which interact with a large number of people have to deal with a large amount of data. These institutions need to maintain this data in a structured manner so that it can be processed and retrieved when needed. These organisations used Database Management Systems to solve their problem of data storage. THe case with such systems is that in contrast with other software solutions, DBMS have historically been implemented in a diverse manner with different software solution providers developing their own flavours. This makes integrating databases from multiple sources a challenging task. Many organisations which use DBMS may use different versions of these, and we need to maintain uniformity in the storage of data. Also, many organisations may use different logical schemas to store their data, such as the tables and their column names. Such inconsistency might lead to problems when this data needs to be retrieved centrally or merged to produce a uniform structure.

Approach

The problem of integrating heterogeneous databases requires the use of data structures to handle multiple tables and column names for the same logical structure. Such a task requires a platform which can provide us with the functionality we need. This ruled out using a web based solution as most backend languages lack these features and aren't robust enough. Also, a visual feedback of the queries was important to check the results in a unified manner. A GUI was also necessary as the organisations or people handling the databases might not be familiar with programming. Development of GUI in Python is cumbersome as tkinter is known for its lack of layout options and there isn't a development environment suited for the purpose. This made JAVA the most viable option for us to use. JAVA has excellent options to include database management such as the JDBC, ODBC and sqlite drivers. In addition to this, development of GUI in JAVA is very simplified due to its Object Oriented nature and WIndowBuilder. Hence, we decided to proceed with JAVA.

Development Environment

- MySQL DBMS
- ObjectDB - an object database for Java
- Netbeans Eclipse IDEs
- GIT version Control

Progress

The entire project work is divided into 5 stages:

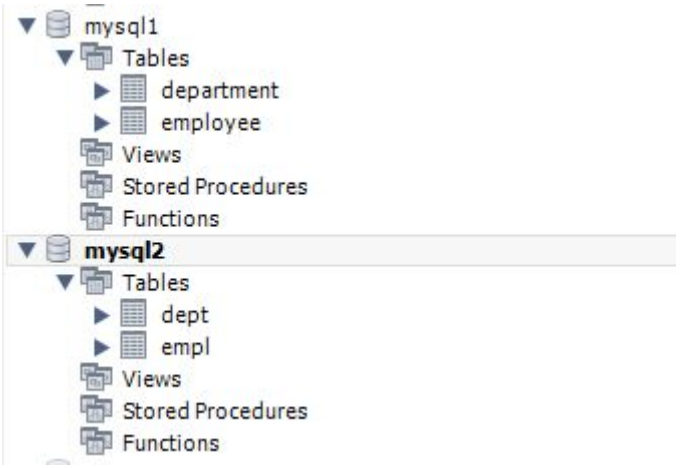
Stage 1:

The idea for the solution was to first to build a system to handle a single database and build it in a bottom up fashion.This would allow us to formulate the idea behind the solution clearly in a structured manner. The first step was to build a GUI which could run a query on a single database. This would later become our unified database. We developed a GUI window that allowed the user to enter the fields required and the table, along with any conditional statements. The result was displayed in a separate window as a table or list of tuples. The query being executed was printed to the console so that we could check what was happening under the hood of the system. This would come in handy when dealing with multiple databases.
Here is the code to launch a new window to display the results of the query.

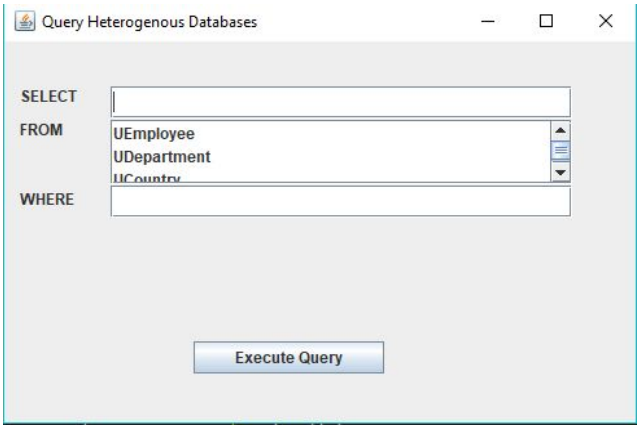
```
// display result table on a new window
public static void JTableDisplay(Object[][] rowData, Object[] columnName) {
    JFrame resultFrame = new JFrame("Unified Data");

    JPanel panel = new JPanel();
    panel.setLayout(new BorderLayout());

    JTable table = new JTable(rowData, columnName);
    table.setEnabled(false);

    JScrollPane tableContainer = new JScrollPane(table);

    panel.add(tableContainer, BorderLayout.CENTER);
    resultFrame.getContentPane().add(panel);

    resultFrame.pack();
    resultFrame.setVisible(true);
}
```

Stage 2:

After successfully implementing a retrieval system for querying a single database, we proceeded with the task of integrating multiple databases. The databases were chosen in such a manner that they had different table names. The structure of the databases was as shown below.



The databases in question were mysql1 and mysql2. To query on a homogeneou database, all the user had to do was to select the unified table on which he wished to perform the query. The user was given the option to choose the fields needed and the conditions for retrieval. The column names in the tables were identical and hence this simplified our task. THe GUI for the retrieval system described above was as shown below.



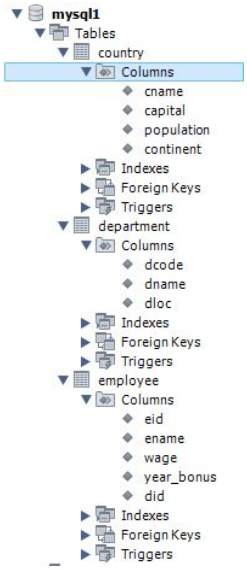
Due to similar column names, the query was run on both the databases and the result was combined by performing a set union operation on the tuples returned from both the tables. The unified tables were built using a hashmap which mapped the corresponding tables from each database. The JAVA code built for it is as follows:

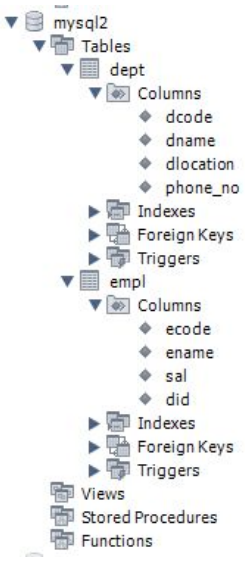
```
HashMap<String, ResultSet> queryResultSets = new HashMap<>();
Statement stmt = conn.createStatement();
ResultSet rs = stmt.executeQuery(query);
queryResultSets.put(dbURL, rs);
```

```
// combine results
ArrayList<Object[]> queryResult = new ArrayList<>();
for(String dbURL : unifiedDB.keySet()) {
    Database db = unifiedDB.get(dbURL);
    try {
        ResultSet rs = queryResultSets.get(dbURL);
        if(rs == null)  continue; // current db doesnt contain selected tables
        ResultSetMetaData metaData = rs.getMetaData();
        int noOfColumns = metaData.getColumnCount();
        while (rs.next()) {
            Object[] row = new Object[totalCols];
            for (int i = 0; i < noOfColumns; i++) {
                row[colIndex] = rs.getObject(i+1);
            }
            queryResult.add(row);
        }

    } catch(SQLException e1) {
        e1.printStackTrace();
    }
}
```

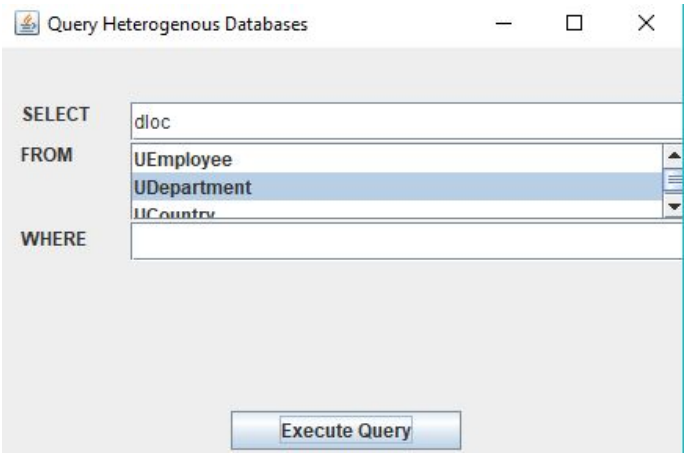
Stage 3:

Now that we were comfortable with the task of handling multiple databases, we tried to tackle a problem that was evident in our previous solution. The table columns or the field names had to be the same for us to run the queries. So, we built a mapping from the actual column names to their names in the unified tables. The user could now select the table(s) he wanted to run the query on using a checklist, and the query would be split across the databases. The structure for the test databases is displayed in the following picture.





Check list enabled GUI for running queries



A HashMap was used for the mappings(shown below)

```
static HashMap<String, Database> unifiedDB = new HashMap<>();
static String[] unifiedTables = new String[] {"UEmployee", "UDepartment", "UCountry"};
static {
    // DATABASE 1
    Database mySQLDB1 = new Database();
    mySQLDB1.addU2AMapping("UEmployee", "employee");
    mySQLDB1.addTableColMapping("UEmployee", "ecode", "eid"); // ecode in UEmployee = eid in employee of mysql1
    mySQLDB1.addTableColMapping("UEmployee", "sal", "wage");
    mySQLDB1.addU2AMapping("UDepartment", "department");
    mySQLDB1.addU2AMapping("UCountry", "country");
    unifiedDB.put("jdbc:mysql://localhost:3306/mysql1", mySQLDB1);

    // DATABASE 2
    Database mySQLDB2 = new Database();
    mySQLDB2.addU2AMapping("UEmployee", "empl");
    mySQLDB2.addU2AMapping("UDepartment", "dept");
    mySQLDB2.addTableColMapping("UDepartment", "dloc", "dlocation");
    unifiedDB.put("jdbc:mysql://localhost:3306/mysql2", mySQLDB2);
}
```

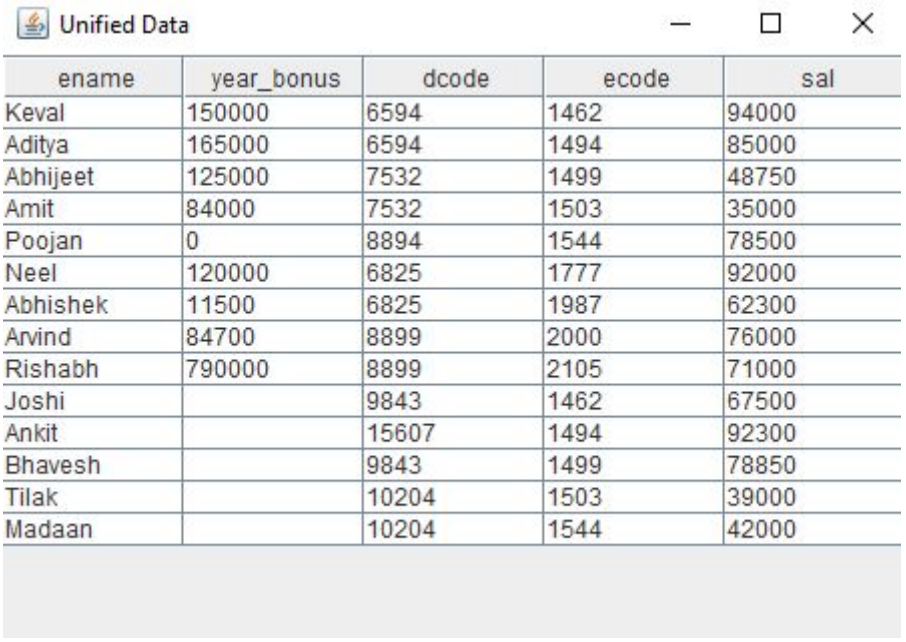
The input query was split appropriately according the field names in the tables; and depending on whether the columns and tables were present in the database, the results were merged. Here are few snippets from the console window which show the query being run on the individual databases.

```
SELECT * FROM country
jdbc:mysql://localhost:3306/mysql2 doesn't contain table: UCountry
```

As seen in the snippet above, the database mysql2 doesn't have a table corresponding to the unified table UCountry and hence the final result would come from the first database.

Another sample query was run on the unified UDepartment joined with UEmployee table and the result window was as follows. Note that the year_bonus field was not present in the dept table in mysqldb2 and hence those fields are empty for some of the result tuples.

```
<terminated> Main (1) [Java Application] C:\Program Files\Java\jre1.8.0_111\bin\javaw.exe (01-Nov-2018, 9:19:14 PM)
SELECT eid,ename,wage,year_bonus,dcode FROM employee,department WHERE did=dcode
SELECT ecode,ename,sal,dcode FROM empl,dept WHERE did=dcode
```



ename	year_bonus	dcode	ecode	sal
Keval	150000	6594	1462	94000
Aditya	165000	6594	1494	85000
Abhijeet	125000	7532	1499	48750
Amit	84000	7532	1503	35000
Poojan	0	8894	1544	78500
Neel	120000	6825	1777	92000
Abhishek	11500	6825	1987	62300
Arvind	84700	8899	2000	76000
Rishabh	790000	8899	2105	71000
Joshi		9843	1462	67500
Ankit		15607	1494	92300
Bhavesh		9843	1499	78850
Tilak		10204	1503	39000
Madaan		10204	1544	42000

Stage 4:

The previous version of the Database Management System only provided support for relational databases. This is a gross simplification of the complexity and variety of databases we may encounter in the real world. Hence, the system was extended to support object oriented databases, which form a major chunk of database solutions as they are easy to implement and do not need setting up of servers. As JAVA has native support for object databases, it was natural for us to extend the functionality to support object databases. We are using ObjectDB as an object-oriented database.

This database contains 3 classes: Employee, Department, and Country which have the following instance variables:

```
@Entity
public class Employee implements Serializable {
    private static final long serialVersionUID = 1L;

    @Id int ecode;
    String ename;
    int sal;
    int yearBonus;
    int did;
```

```
@Entity
public class Employee implements Serializable {
    private static final long serialVersionUID = 1L;

    @Id int ecode;
    String ename;
    int sal;
    int yearBonus;
    int did;
```

```
@Entity
public class Country implements Serializable{
    private static final long serialVersionUID = 1L;

    @Id String ccode;
    String cname;
    String capital;
    long population;
    String continent;
```

Notice that ‘year_bonus’ and ‘phone_no’ in unified database refers to ‘yearBonus’ in Employee class and ‘phoneNo’ in Department class. Also note that Country class has an instance variable ‘continent’ which is not present in the mysql database. This shows that the object-oriented database can have more number of columns and different column names.

The database was initialised as follows, with extra care needed to add all columns to the mapping individually as object databases do not support the ‘select * from’ notion of relational databases. Each column retrieval query must be explicitly entered. Note that we had to do this process only for columns that had different names in the case of relational databases. This mapping is maintained in the hashmap and will be used later to split the query and querying individual databases.

```
// ORDBMS
static Database ORDBMS = new Database();
// TableColMapping and U2AMapping for all cols bcz select * doesnt work, individual col names have to be mentioned
ORDBMS.addU2AMapping("UEmployee", "Employee e");
ORDBMS.addTableColMapping("UEmployee", "ecode", "e.ecode");
ORDBMS.addTableColMapping("UEmployee", "ename", "e.ename");
ORDBMS.addTableColMapping("UEmployee", "sal", "e.sal");
ORDBMS.addTableColMapping("UEmployee", "year_bonus", "e.yearBonus"); // different than unifiedTableColumn
ORDBMS.addTableColMapping("UEmployee", "did", "e.did");

ORDBMS.addU2AMapping("UDepartment", "Department d");
ORDBMS.addTableColMapping("UDepartment", "dcode", "d.dcode");
ORDBMS.addTableColMapping("UDepartment", "dname", "d.dname");
ORDBMS.addTableColMapping("UDepartment", "dloc", "d.dloc");
ORDBMS.addTableColMapping("UDepartment", "phone_no", "d.phoneNo"); // different than unifiedTableColumn

ORDBMS.addU2AMapping("UCountry", "Country c");
ORDBMS.addTableColMapping("UCountry", "ccode", "c.ccode"); // this col is not in rdbms
ORDBMS.addTableColMapping("UCountry", "cname", "c.cname");
ORDBMS.addTableColMapping("UCountry", "capital", "c.capital");
ORDBMS.addTableColMapping("UCountry", "population", "c.population");
ORDBMS.addTableColMapping("UCountry", "continent", "c.continent");
```

After the mapping of the column, we proceeded to solve the problem of querying two separate database models.

Querying:

Object databases require us(the developer) to query each column individually and we must develop additional code to take care of conditions when the user inputs “*” in the unified interface to query all columns. The solution for it is shown below, which includes extracting actual column names in the table and their

unified database names so that these can be included as headers in the result table.

```
// ordbms query creation
String ordbmsSelectText = tfSelect.getText().replace(" ", "");
ArrayList<String> ordbmsSelectedActualCols = new ArrayList<>();
ArrayList<String> ordbmsSelectedUnifiedCols = new ArrayList<>();
List<String> selectedUnifiedTables = listFrom.getSelectedValuesList();
if(ordbmsSelectText.equals("") || ordbmsSelectText.equals("*")) { // get all cols for selected tables
    selectedUnifiedTables.forEach(unifiedTableName -> {
        TableDescription td = ORDBMS.getTable(unifiedTableName);
        td.getAllUnifiedColNames().forEach(unifiedColName -> {
            ordbmsSelectedActualCols.add(ORDBMS.getActualColName(unifiedColName));
            ordbmsSelectedUnifiedCols.add(unifiedColName);
        });
    });
}else {// replace unified col names by actual and discard cols not present in db
    for(String unifiedColName : ordbmsSelectText.split(",")) {
        String actualColName = ORDBMS.getActualColName(unifiedColName);
        if(actualColName.equals(unifiedColName)) continue;
        ordbmsSelectedActualCols.add(actualColName);
        ordbmsSelectedUnifiedCols.add(unifiedColName);
    }
}
unifiedColumnHeaders.addAll(ordbmsSelectedUnifiedCols);
```

The mappings for the tables are also handled similarly(code shown below).

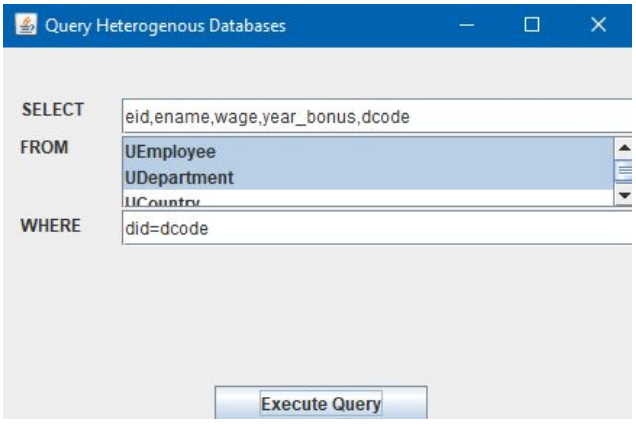
```
    List<String> selectedActualTables = new ArrayList<>();
    selectedUnifiedTables.forEach(unifiedTableName -> {
        selectedActualTables.add(ORDBMS.getActualTableName(unifiedTableName));
    });
    String ordbmsQuerySelectText = String.join(",", ordbmsSelectedActualCols);

    String ordbmsQuery = "SELECT " + ordbmsQuerySelectText + " FROM " + String.join(",", selectedActualTables);

    if(!tfWhere.getText().replace(" ", "").equals("")) { // replace unified col names by actual col names
        ArrayList<String> ordbmsWhereConditions = new ArrayList<>();
        for(String condition : tfWhere.getText().replace(" ","").split(",")) {
            String[] uCols = condition.split("=");
            String modifiedCondition = ORDBMS.getActualColName(uCols[0]) + "=" + ORDBMS.getActualColName(uCols[1]);
            ordbmsWhereConditions.add(modifiedCondition);
        }

        ordbmsQuery += " WHERE " + String.join(",", ordbmsWhereConditions);
    }
```

Now, the only part left is to actually run the query on the database, and merge the results. The results obtained from the object database need to be displayed along with the results of the relational database in tabular form. Hence, the field values are placed under the appropriate unified column headers, leaving the columns that do not appear in the table as blank.

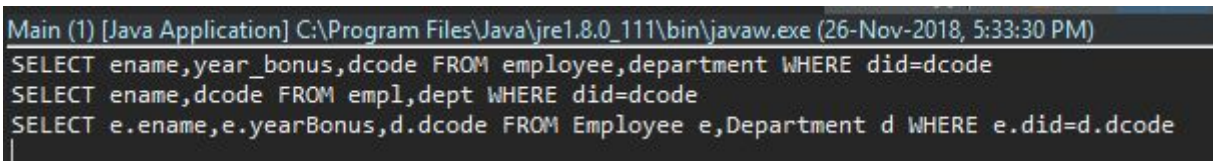
```java
// Combined result
ArrayList<Object[]> queryResult = new ArrayList<>();

// ordbms query execution
EntityManagerFactory emf = Persistence.createEntityManagerFactory(ordbmsFile);
EntityManager em = emf.createEntityManager();

TypedQuery<Object[]> q = em.createQuery(ordbmsQuery, Object[].class);
List<Object[]> results = q.getResultList();

int noOfColumns = ordbmsSelectedUnifiedCols.size();
for(Object[] r : results) {
    Object[] row = new Object[totalCols];
    for (int i = 0; i < noOfColumns; i++) {
        int colIndex = unifiedColHeaderToIndex.get(ordbmsSelectedUnifiedCols.get(i));
        row[colIndex] = r[i];
    }
    queryResult.add(row);
}
```

Results:

The query run earlier was tried again, this time with added support for object databases. The various windows are displayed below along with the command line output showing how the query was split. We can clearly see that the columns were appropriately split and the results were combined correctly.



The last line in the command line output shows the query being run on the object database.



The following was the final result displayed in tabular form, with empty columns left out.

Unified Data

ename	year_bonus	dcode
Emp3	3000	1
Emp1	1000	2
Emp4	4000	2
Emp2	2000	3
Emp5	5000	3
Keval	150000	6594
Aditya	165000	6594
Abhijeet	125000	7532
Amit	84000	7532
Poojan	0	8894
Neel	120000	6825
Abhishek	11500	6825
Arvind	84700	8899
Rishabh	790000	8899
Joshi		9843
Ankit		15607
Bhavesh		9843
Tilak		10204
Madaan		10204

Stage 5:

The crux of the motivation behind the project was completed in the previous stage with the development of a software solution that could query from heterogeneous databases. But the databases were still physically on the same machines. The final step was to make sure that the solution would work no matter where the database was hosted. The existing platform was a simple modification away from this functionality. The usernames and passwords for the various databases can be stored in separate hashmaps with the urls of the databases. hashmaps allow for quick retrieval as well as greater flexibility than arrays. The modified implementation is as shown below.

```
static HashMap<String,String> mysqlDBUserName = new HashMap<>();
static HashMap<String,String> mysqlDBPass = new HashMap<>();
```

Adding credentials to hashmaps:

```
unifiedRDBMS.put("jdbc:mysql://sql12.freemysqlhosting.net/sql12267109", mySQLDB1);
mysqlDBUserName.put("jdbc:mysql://sql12.freemysqlhosting.net/sql12267109","sql12267109");
mysqlDBPass.put("jdbc:mysql://sql12.freemysqlhosting.net/sql12267109","lQqPGciqCi");
```

Extracting username and password from hashmap before connecting:

```
ArrayList<Connection>  connections = new ArrayList<>(); // end all connection in the end
for(String dbURL : unifiedRDBMS.keySet()) {
    try {

        String user= mysqlDBUserName.get(dbURL);
        String passwd= mysqlDBPass.get(dbURL);
        Connection conn = DriverManager.getConnection(dbURL, user, passwd);
```

Note that the database has been hosted remotely at ‘sql12.freemysqlhosting.net’. This is a free service used for testing and hence the performance is not reliable with frequent connection timeouts. There are numerous commercial services available which provide reliable hosting solutions.

Challenges

The current solution assumes that the user has some idea about the logical construct of the database system. If the user is not familiar with the data, he may not be aware about the semantic equivalence of columns in different databases. If there are no logical connections between the two databases, running queries on the simultaneously doesn't make much sense. Also, the current system supports just the functionality of retrieval. We cannot update the values in the database from the unified interface but they can be modified in their individual databases.

Further Possible Enhancements

1. Support for querying XML Database
2. Querying aggregates
3. Data entry and editing functionality
4. Automatic mapping of column names in the tables using statistical analysis

References

- https://en.wikipedia.org/wiki/Heterogeneous_database_system
- https://www.mysql.com/
- https://www.tutorialspoint.com/mysql/
- https://www.tutorialspoint.com/jdbc/
- https://www.objectdb.com/
- https://en.wikipedia.org/wiki/Distributed_database
- https://www.freemysqlhosting.net/