A VIRTUAL DATABASE FOR STANDARD REPORTING TOOLS

Computer Science 380 Project

Haoxiang Zhu

2006

Supervised by: Dr. Santokh Singh & Dr. Xinfeng Ye

Department of Computer Science

University of Auckland

ABSTRACT

Traditionally, reporting tools connect to a data source directly. In a service-oriented architecture, the data model might change from time to time. Therefore, any changes made to the data model will result in changes to the reporting tool's configuration.

In this paper, we are going to address this problem by investigating the implementation of a middle tier, which locates in between of the reporting tool and the data source. After the middle tier is in place, every change made to the data model does not influence the reporting tool's configuration in any way.

CONTENTS

1. Introduction

A reporting tool is a software tool that allows end users to build professional business reports based on data in a database. These reports can then be scheduled to be run automatically at particular times or based on business rules and distributed a group of end consumers who are interested in the information.

Most of the reporting tools have a direct connection to a data source. However, in a service oriented architecture, the data mode in the underlying data source may change from time to time. The data model change can be caused by, for example, a change of service provide, a change of the business logic etc. Therefore, in order for the reporting tool to produce a consistent result, the reporting tool's setting also has to be changed to cope with the data model changes.

In this project, we are going to focus on the development of a middle tier that is to be used by a standard reporting tool. It is aiming to improve the usability of standard reporting tools. After the middle tier is in place, the reporting tool should be able to work independently to any data model changes (including database location) in the underlying database. This middle tier locates between the standard reporting tool and the database server, so that whenever there are changes made in the data model, we only need to make minimum amount of changes in the middle tier without changing the setting of the standard reporting tools.

Besides, we will also compare the performances of different implementations of the middle tier, in order to find out a best solution which not only satisfy the requirements but also work as efficiently as possible.

2. Background

As Internet plays a more and more important role in our life, many big organizations prefer to have their customer invoices, monthly billing information in electronic form which is automatically generated by a reporting tool, rather than the old paper form.

Account Summary

Account History

Billing Month	Total Charge	Amount Paid	Amount Owing
03 - Sep - 2006	$54.95	$54.95	$0.00
03 - Aug - 2006	$17.72	$17.72	$0.00
23 - Jul - 2006	$49.95	$49.95	$0.00
23 - Jun - 2006	$49.95	$49.95	$0.00
23 - May - 2006	$49.95	$49.95	$0.00
23 - Apr - 2006	$49.95	$49.95	$0.00
23 - Mar - 2006	$49.95	$49.95	$0.00
23 - Feb - 2006	$49.95	$49.95	$0.00
23 - Jan - 2006	$49.95	$49.95	$0.00
23 - Dec - 2005	($74.02)	($74.02)	$0.00
23 - Nov - 2005	$49.95	$49.95	$0.00

Figure 2.1 Slingshot Online Account History (https://www.slingshot.co.nz/)

By doing so, not only it provides the customers more controls over the data that they are interested in, it also eliminates the costs of distributing traditional paper-form mails.

Most of the reporting tools have a direct connection to a database. The reporting tool generates different results according to the different requests it receives from the end users of the reporting tools.

Let us, first of all, look at the design of traditional reporting tools illustrated in Figure 2.2

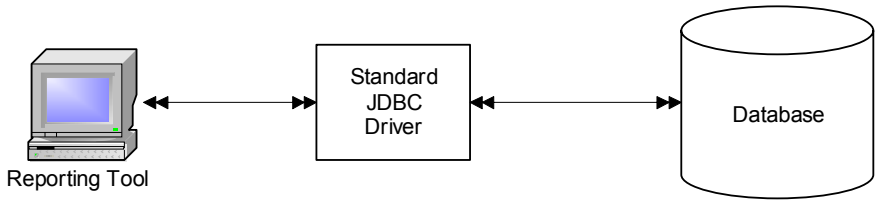


Figure 2.2 Architecture Diagram of Traditional Reporting Tool

From Figure 2.2, we observed the following:

- A reporting tool issues a query (i.e. SQL statement), requesting the data that the end user is interested in.
- The query is intercepted by the "Standard JDBC Driver" (e.g. MySQL Connector/J)

- The JDBC Driver executes the query against the target database.
- The JDBC Driver constructs a *ResultSet* Object which contains all the data that the reporting tool is requesting for.
- The *ResultSet* is sent back to the reporting tool.
- The reporting tool gets the *ResultSet,* format the result according to user's requirements, and display the report.

Let us use an example to illustrate the behavior of the standard reporting tool. We illustrate this by looking at a widely-used reporting tool "BIRT". "BIRT" is an open source, Eclipse-based reporting system that integrates with applications to produce compelling reports for both web and PDF. BIRT provides core reporting features such as report layout, data access and scripting.

Similar to what we have mentioned earlier, for BIRT, every report has a direct connection to a data source through a JDBC driver. All the data presented in that particular report is obtained from the data source it connects to.

Figure 2.3 shows a dialog box prompts the data source the particular report connects to:

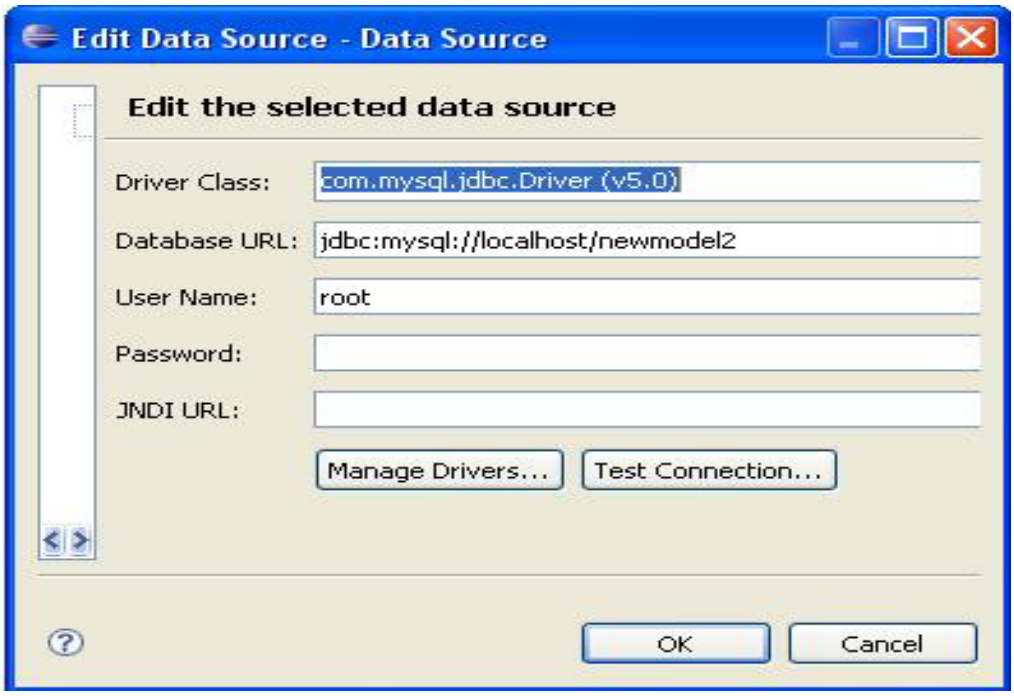


Figure 2.3 BIRT data source connection

After this connection is established, we can then start configuring the report by selecting a dataset from the data source. The dataset is specified by an SQL statement executed against the data source, as shown in Figure 2.4.

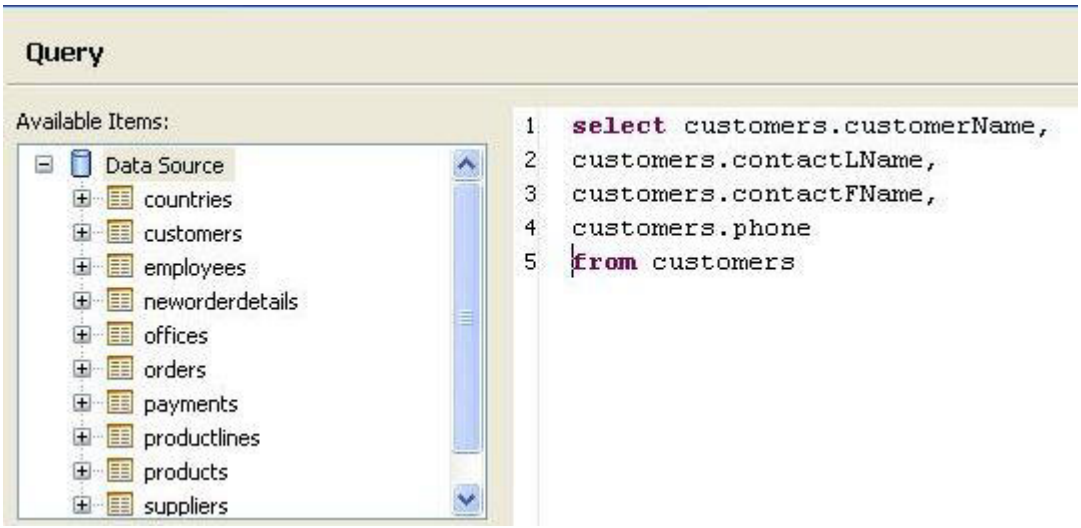


Figure 2.4 BIRT select dataset

In this example, our dataset is *customerName*, *contactLName*, *contactFName*, *phone* from *customer* table of the data source.

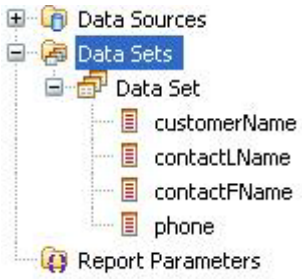


Figure 2.5 Selected dataset

When we have the dataset ready, we can design the layout of the report and use the data from the dataset to populate the report, as shown in Figure 2.6.

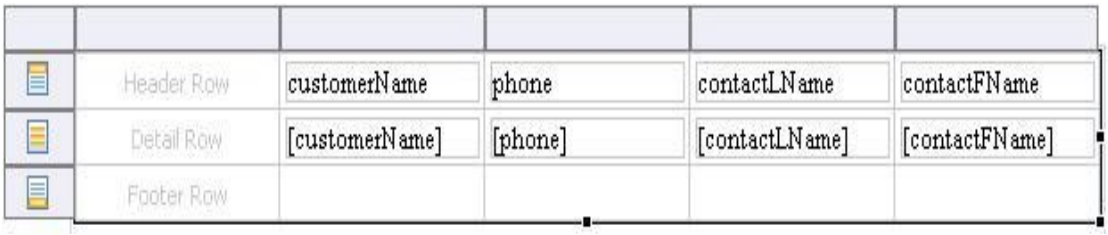


Figure 2.6 Report design view

Up to this point, we have all the setting ready in order to generate a report, we need to keep in mind that all the data present in this report is obtained from the data source this particular report connects to. Figure 2.7 shows a report generated by the above report setting:

customerName	phone	contactLName	contactFName
13 Collectables, Ltd.	(171) 555-2282	Devon	Elizabeth
5 Gifts By Mail, Co.	+47 2212 1555	Klaeboe	Jan
ANG Resellers	(91) 745 6555	Camino	Alejandra
AV Stores, Co.	(171) 555-1555	Ashworth	Rachel
Alpha Cognac	61.77.6555	Roulet	Annette
American Souvenirs Inc	2035557845	Franco	Keith
Amica Models & Co.	011-4988555	Accorti	Paolo
Anna's Decorations, Ltd	02 9936 8555	O'Hara	Anna
Anton Designs, Ltd.	+34 913 728555	Anton	Carmen
Asian Shopping Network, Co	+612 9411 1555	Walker	Brydey
Asian Treasures, Inc.	2967 555	McKenna	Patricia

Figure 2.7 A simple report generated by BIRT

This is how BIRT uses its underlying data source to generate a report. In this example, the reporting tool (BIRT) has a direct connection to its data source with no data model changes made.

However, in some cases, the data model, at the server side, may change from time to time, it is therefore impossible for the reporting tool to work as correctly as before without changing the configurations of the reporting tool. This may not be desired from the business prospective since they do not want their customers to reconfigure the reporting tool every time they change the data model.

Let us once again, use a simple example to illustrate how this can happen: Image the reporting tool has a connection to a data source which contains only one table "MyTable", this table has two columns: ColumnX and ColumnY. The reporting tool's configurations contain a query which extracts ColumnX from this table: *select ColumnX from MyTable*, as shown in figure 2.8:

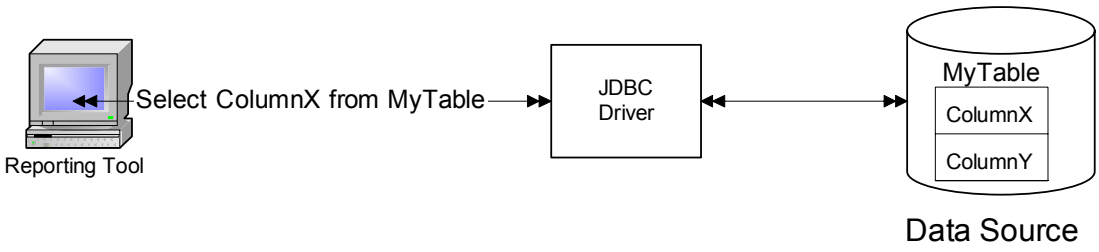


Figure 2.8 Reporting tool extract data from data source

This setting works fine because the query is valid according to the current data model. But if now the data model in the data source is changed by renaming ColumnX to NewColumnX. In this case, if we do not change the reporting tool's setting, i.e. do not change the query: *select ColumnX from MyTable*, it will cause an error, since

ColumnX does not exist in the data model any more. This is shown in Figure 2.9:

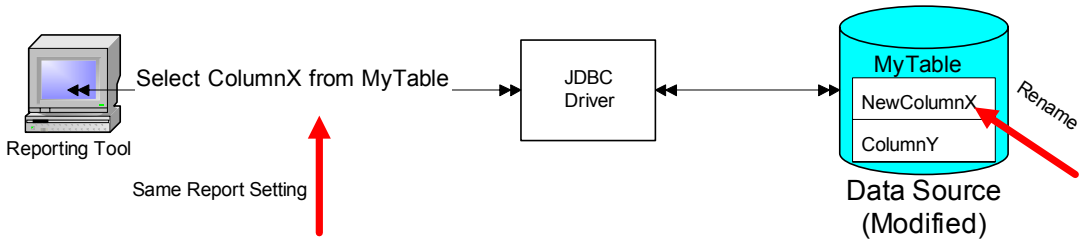


Figure 2.9 Reporting tool causing error

Clearly, we can see how this problem can be raised from the above example. It is therefore useful to make the reporting tool's setting independent to the data model change in the data source.

In the remaining chapters of this report, we are going to address this problem and provide a feasible solution to this problem.

3. System Architecture & Design

In this chapter, we start by an overview of our system design in Section 3.1. In Section 3.2, we explain why we need a customized JDBC driver in our underlying system. Then we present a more detailed architecture diagrams for the underlying system in Section 3.3. Finally, in Section 3.4, we conclude this chapter by explaining the design of the middle tier in more detail.

3.1 System Overview

We have already discussed the potential problems the traditional reporting tool may have (refer to Chapter 2, Figure 2.8 and Figure 2.9), basically, when the data model changes, the reporting tool will not work as correctly as before without modifying its configuration. Our design is aiming to solve this problem, so that the changes made in the data model are independent to the query generated by the reporting tool.

One possible way of solving this problem is: when the reporting tool sends the query defined in its setting, we can have another layer (a middle tier) which acts like a ***virtual database*** for the reporting tools to connect. What it actually does is that it modifies the query before it gets executed against the modified database at runtime. The modified version of the query should be executed against the modified database, but still provide the reporting tool with the desired result. This can be illustrated in Figure 3.1:

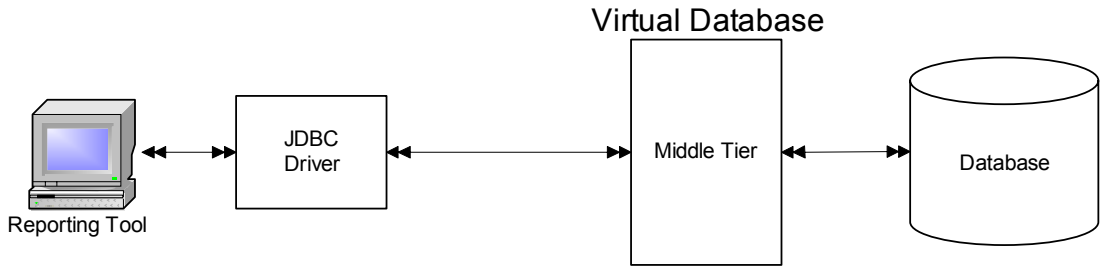


Figure 3.1 A bird eye view of the system

In Figure 3.1, unlike traditional reporting tools, we make the reporting tool connect to a middle tier (***a virtual database***) instead of a database directly. However, the JDBC driver that the traditional reporting tool uses only provide the facility to connect to a real database, in order to achieve our goal to make the reporting tool connect to the ***virtual database***, we therefore need to modify the JDBC driver.

3.2 Customized JDBC Driver Design

According to the initial investigation, most reporting tools acquire data by querying the relational databases. Most of them, particularly Java written reporting tools, connect to such relational databases by a type 3 JDBC driver. Almost every RDMS has a JDBC driver that is compatible to its database, e.g. MySQL provides the Connector/J as its database connector.

Standard JDBC provides methods that do these basic things:

- Create and manage connections to data sources based on a URL or DataSource object registered with a Java Naming and Directory Interface (JNDI) naming service. Thus, no client-side configuration is required.
- Compose and send SQL statements to the data sources.
- Retrieve and process result sets that are returned to the Java application or applet.

Since we do not want a direct connection from the reporting tools to the relational database, we therefore need to define our own JDBC driver that satisfy our interests, the customized JDBC driver should,

EITHER:

- Connect to a middle tier service instead of a database.
- Pass the request from the reporting tool to the middle tier service, so that the middle tier will process the request and return the result back to the JDBC driver.
- Send the result that obtained from the middle tier service back to the reporting tool as if the result is obtained directly from the database.

OR:

- Connect to a middle tier service instead of a database.
- Pass the request from the reporting tool to the middle tier service, so that the middle tier will process the request and return a new request that can be executed on the modified database, giving the same result as if the data model has not been changed.
- Act like a standard JDBC driver, but process the new request obtained from the middle tier service.

3.3 System Architecture Diagrams

After we have a customized JDBC driver (refer to Section 3.2) in place, we can now modify the behavior of the reporting tool. The customized JDBC driver gives the reporting tool the ability to connect to a middle tier (contains a set of programs), as supposed to a real database. By doing so, we can use the programs in the middle tier to achieve our goal.

Figure 3.2 illustrates one possible solution:

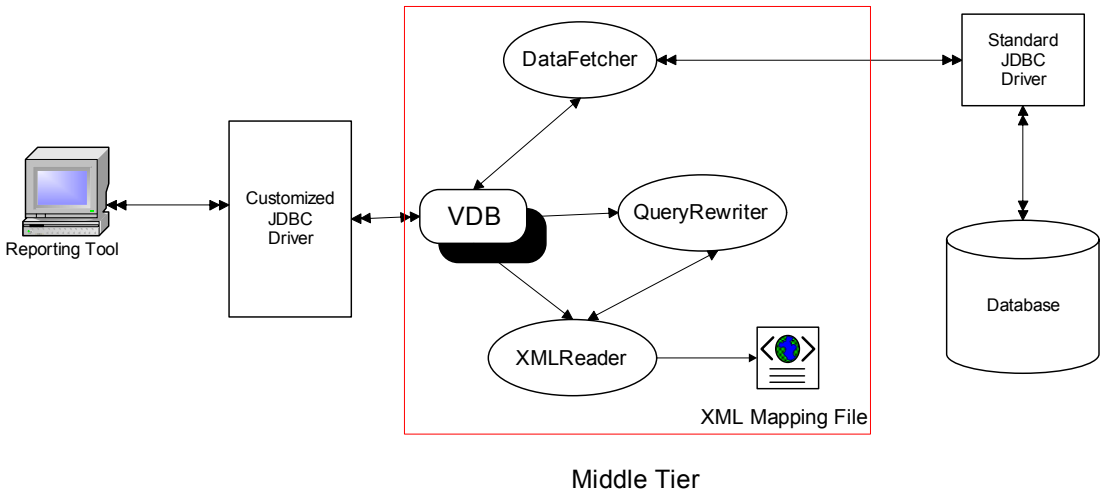


Figure 3.2 System Architecture Diagram (Approach One)

Let us look at this design in more detail. The reporting tool uses a customized JDBC driver to connect to this middle tier instead of the database directly. In essence, the customized JDBC driver passes whatever requests coming from the reporting tool to the middle tier. The middle tier consists of the following components:

- VDB (***Virtual Database***): It is called a ***virtual database***, since it is not a real database. However, it provides the reporting tool an interface to connect to as if it is a real database. It is the central controller in the middle tier as it dispatches the requests coming from the reporting tool to the appropriate elements within the middle tier for processing and returns the results back to the reporting tool.

- XML Mapping file: When there are data model changes in the real database, we do not want these changes to be visible to the reporting tool, another word, the reporting tool maintains an old (consistent) view of the database that is never changed. In order to achieve this, we need to have some methods of mapping the data model in the real database to the data model that the reporting tool sees. Hence, this XML file specifies the mapping from the actual database to reporting tool's view of the database.(The mapping is

explained in Chapter 4 of this report)

- XML Reader: The XML Reader reads the XML Mapping files and extracts the useful information from the XML Mapping file. It is generally used by the QueryRewriter when rewriting the query generated from the reporting tool.

- QueryRewriter: Since the requests (i.e. SQL query) coming directly from the reporting tool will not be able to be executed correctly against the real database after the changes have been made. It is therefore necessary to reconstruct the incoming query into a new version which is compatible to the current database. QueryRewriter does this by accessing the XML Mapping file through the XML Reader. Generally, executing the query constructed by the QueryRewriter against the current database produces the same result as executing the initial query against the old database (i.e. the database before the data model changes).

- DataFetcher: As its name suggests, this is the object that accesses the real database in the middle tier. It uses a standard JDBC driver to access the database, but executes the re-written query generated by the QueryRewriter. Upon receiving the *ResultSet*, it forwards the *ResultSet* back to VDB. VDB will then pass this ResultSet back to the reporting tool which initialized the request. (However, this class is not required in our second approach, which is going to be explained shortly.)

To sum up, this design achieves our goal by performing the following steps:

1. The reporting tool generates a request. This request is sent to the customized JDBC driver.
2. The customized JDBC driver forwards this request to VDB which sits in the middle tier.
3. VDB dispatches the same request to *QueryRewriter*, expecting a rewritten query which can be executed against the current database.
4. In order to rewrite the query, the *QueryRewriter* calls the XML Reader to extract the relevant information from the XML Mapping file.
5. *XMLReader* reads the XML Mapping file and returns the mapping information back to the *QueryRewriter*.
6. According to the mapping information, the QueryRewriter reconstructs the query to be compatible to the current database.
7. *QueryRewriter* returns the rewritten query back to VDB.
8. VDB forwards this rewritten query to the *DataFetcher*. Upon receiving the written query, the *DataFetcher* executes this query against the current database through a standard JDBC Driver (e.g. MySQL Connector/J).
9. *DataFetcher* receives the *ResultSet* and send it back to the VDB.

10. VDB transfers this *ResultSet* back to the reporting tool through the customized JDBC driver as if this *ResultSet* is from the database directly.

The following sequence diagram illustrates the interactions among the reporting tool the middle tier and the data source for this approach:

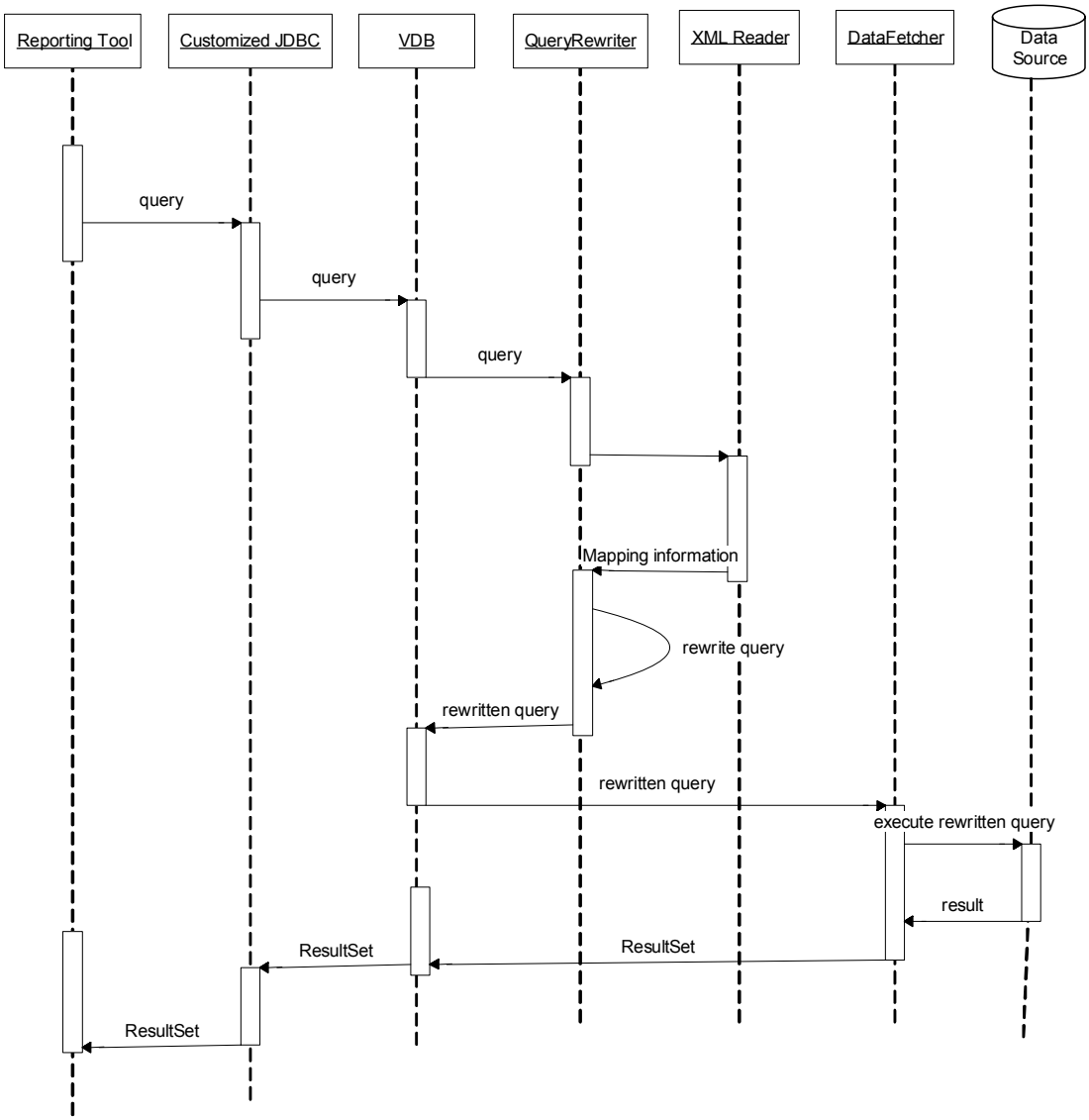


Figure 3.3 Sequence Diagram for Approach One

In this design, we maintain a connection from the middle tier to the current database through the DataFetcher. However, it is not necessary for us to maintain such a connection from the middle tier. This is because all we need is an updated version of the query statement which can be executed against the current database but yield the same results as if it is executing the old query against the database before the data model changes. Hence, we have a second approach to solve the problem. The system architecture of the second approach is illustrated in Figure 3.4

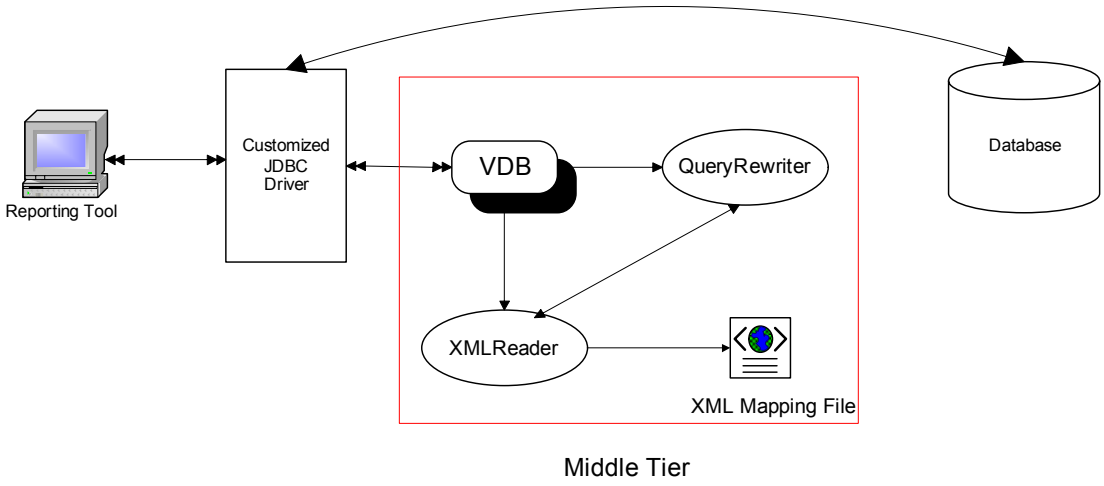


Figure 3.4 System Architecture Diagram (Approach Two)

This approach does not differ much to the previous approach except that no connection is maintained from the middle tier to the database. Instead, more like the traditional reporting tool, the connection is maintained between the JDBC driver and the database. By doing so, we can therefore avoid the need of using a *DataFetcher* in the middle tier, since now the results are returned from the database back to the JDBC directly bypassing the middle tier.

Nevertheless, we still need a customized JDBC driver to connect to the middle tier due to the fact that before getting the *ResultSet* from the database, we still need the middle tier to reconstruct the query which can be executed against the current database.

This design achieves our goal by performing the following steps (Step 1-7 are the same as the previous approach):

1. The reporting tool generates a request. This request is sent to the customized JDBC driver.
2. The customized JDBC driver forwards this request to VDB which sits in the middle tier.
3. VDB dispatches the same request to *QueryRewriter*, expecting a rewritten query which can be executed against the current database.
4. In order to rewrite the query, the *QueryRewriter* calls the XML Reader to extract the relevant information from the XML Mapping file.
5. *XMLReader* reads the XML Mapping file and returns the mapping information back to the *QueryRewriter*.
6. According to the mapping information, the *QueryRewriter* reconstructs the query to be compatible to the current database.
7. QueryRewriter returns the rewritten query back to VDB.

8. VDB transfers the rewritten query to the customized JDBC driver
9. Upon receiving the rewritten query, the JDBC driver executes the rewritten query in the current database which yield the same results as if it is executing the old query against the database before the data model changes.

The following sequence diagram illustrates the interactions among the reporting tool the middle tier and the data source for this approach:

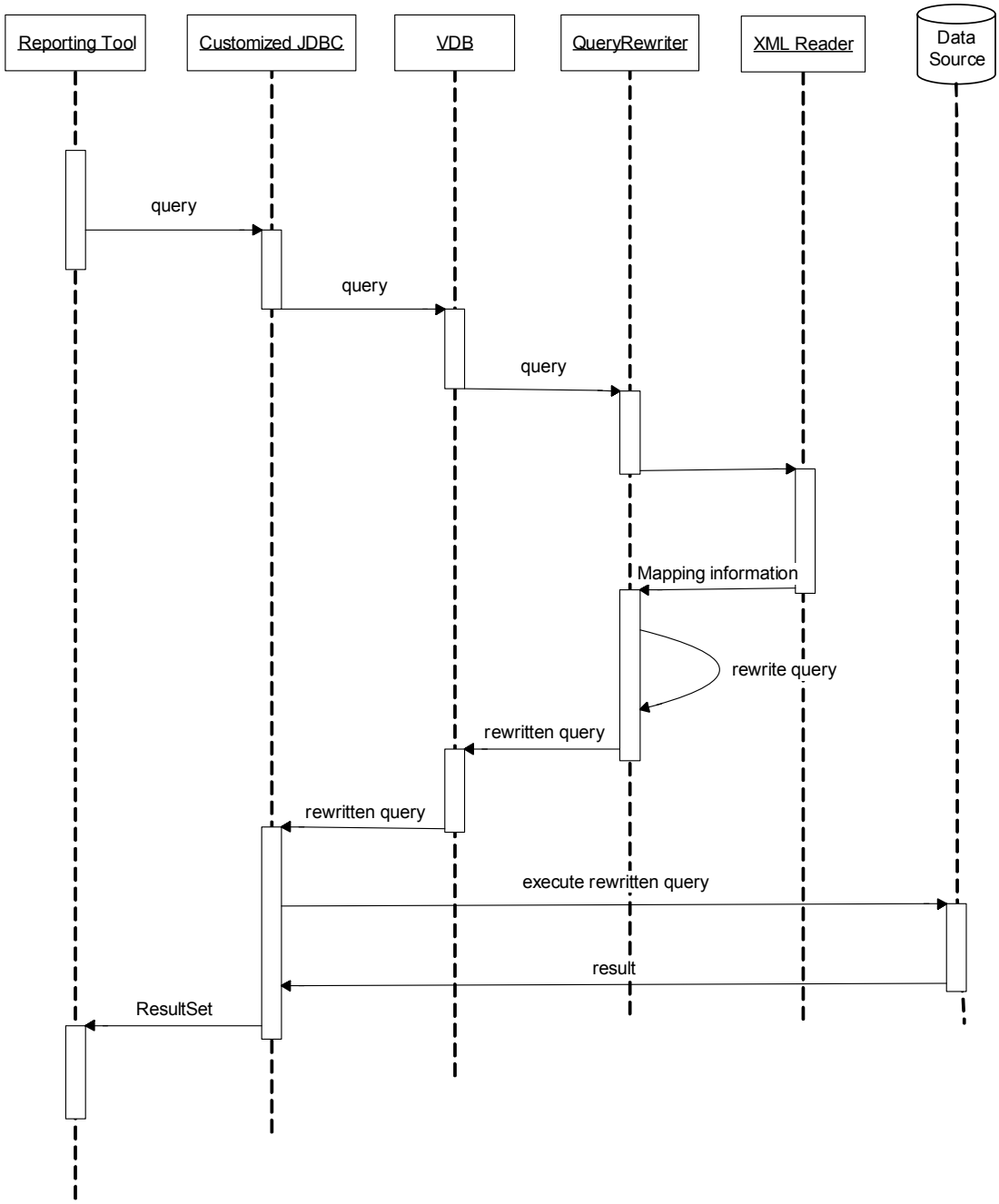


Figure 3.5 Sequence Diagram for Approach Two

3.4 The Middle Tier

3.4.1 Distributing the Middle Tier Service Using RMI Technology

As we can see from both architectures in Section 3.3, in order to achieve our goal, it is necessary to have a middle tier service that sits between the reporting tool and the database. From the business prospective, it is possible that the middle tier service is either hosted in the same server where the database is hosted, or, more commonly, in a dedicated server machine.

In our proposed system, it is the JDBC driver (refer to Section 3.2), which sits on the client side, connects to the middle tier service. In order to invoke the server methods remotely from the client side (i.e. the reporting tool), it is therefore feasible to use JAVA RMI technology.

Figure 3.6 illustrates how RMI is integrated into our system:

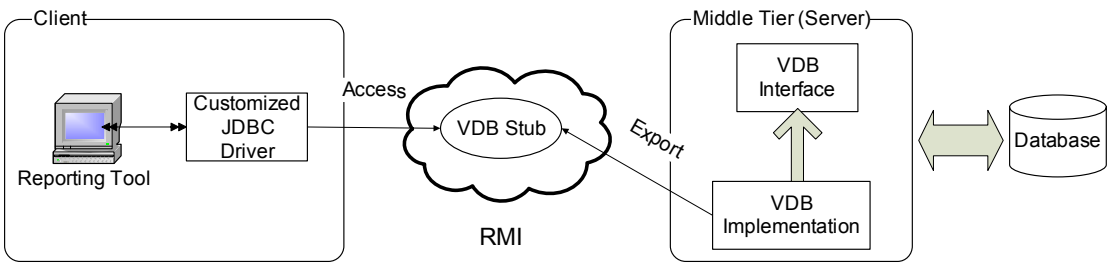


Figure 3.7 RMI in the system

"Java Remote Method Invocation (Java RMI) enables the programmer to create distributed Java technology-based to Java technology-based applications, in which the methods of remote Java objects can be invoked from other Java virtual machines, possibly on different hosts. RMI uses object serialization to marshal and unmarshal parameters and does not truncate types, supporting true object-oriented polymorphism" [Java API].

It is worth mentioning that "RMI uses object serialization to marshal and unmarshal parameters", it is therefore important to note that everything we passed though RMI should either be primitive types (such as *String*, *int*) or serializable *Object*.

3.4.2 Middle Tier Design

In Section 3.3, we mentioned that VDB (***Virtual Database***) is a central controller in the middle tier as it provides the reporting tool an interface to connect to as if it is a real database. The customized JDBC driver (from the client side) needs to access this middle tier remotely. A solution to this problem is to make the *VDB* a remote object which can be executed from the client side. Since *VDB* is the central controller, if we can access the *VDB* remotely, we can therefore access the middle tier remotely.

Here, we use the standard way to achieve the remote method invocation: we define an interface called *VDB,* and this interface exists in both client (Customized JDBC driver) and the server side (middle tire). The server side provides the actual implementation (*VDBImpl*) to the *VDB* interface.

Figure 3.7 is a class diagram illustrates the structure of the middle tier, and how the middle tier is distributed as RMI services to the client.

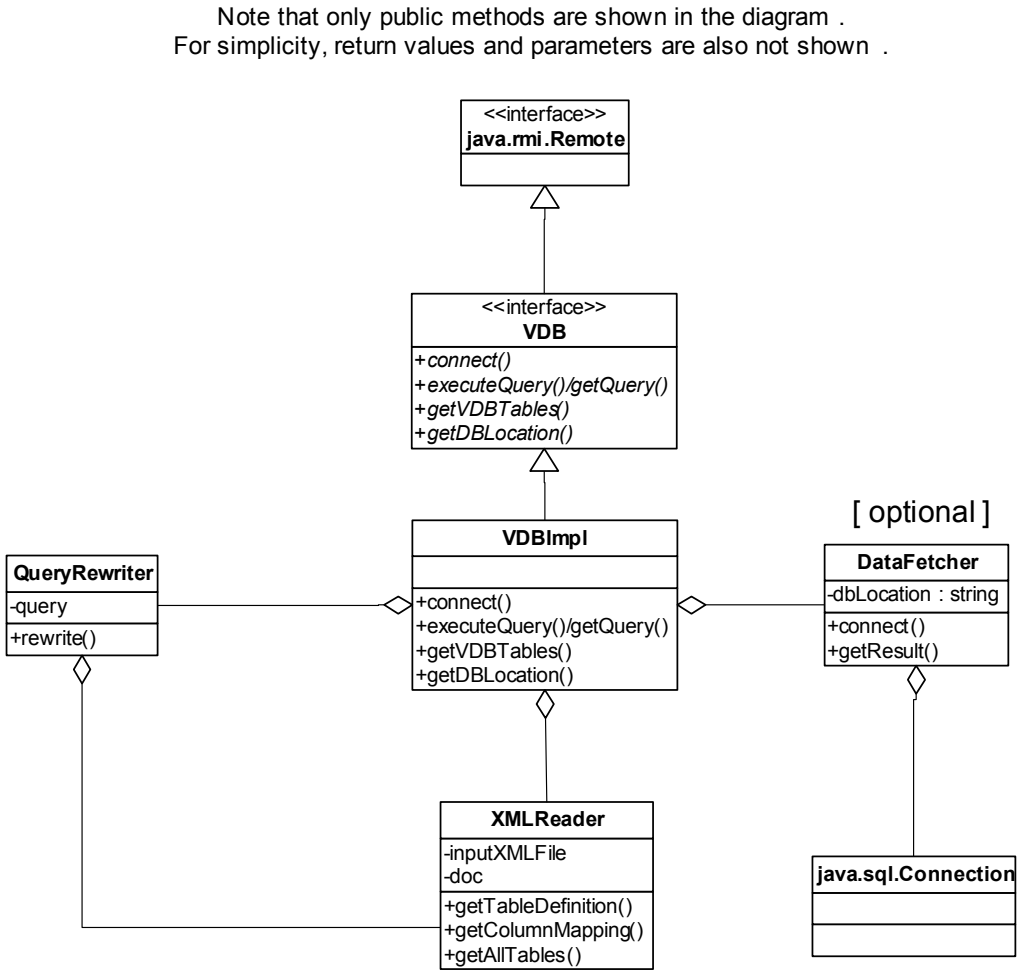


Figure 3.7 Class Diagram of the Middle Tier

As we can see in this diagram, every class in the middle tier represents one of the components in the middle tier architecture (refer to Section 3.3), we have already discussed them in previous sections. Now, let us look at each of the classes in more detail.

- We make *VDB* the remote object by implementing the *java.rmi.Remote* interface, and it is itself an interface. The client side (Customized JDBC driver) must have the same interface in order to access the remote object.

- *VDBImpl* is the actual implementation of the *VDB* interface, and it provides the functionalities of the following methods:
  - *connect*: When this method is invoked, the VDB calls the connect method on DataFetcher, which establishes an actual database connection between *DataFetcher* and the actual (real) database.
  - *executeQuery / getQuery*: This method is called differently depending on different implementations. *executeQuery* executes the rewritten query against the current database, while *getQuery* only returns the rewritten query as a *String*. These two methods are used mutually exclusively. They are used in different implementations (more detail in Section 5.3).
  - *getVDBTables*: This method returns all the tables in the ***Virtual Database***, another word, the tables that the reporting tool is supposed to see. This is because as the data model changes, the tables that actually exist in the current database may not exist in the reporting tool's view of the database (e.g. adding a table in the data model).
  - *getDBLocation*: This method returns the URL string of the current database(e.g. "*jdbc:mysql://www.myDBServer/myDBName*;")

- *DataFetcher:* This class maintains a connection (*java.sql.Connection*) to the current database. The location of the database (*dbLocation*) is obtained through *VDB*. The class is optional in the middle tier design because it is only used in the design showed in Figure 3.2 where the middle tier keeps a connection to the database. The design showed in Figure 3.4 does not require a persistent connection between the middle tier and the real database. Hence this class is not needed. The class diagram for the latter implementation is the same except that there is no such DataFetcher class.
  - *connect*: Connect to the real database through a standard JDBC Driver (e.g. MySQL Connector/J).
  - *getResult*: Obtain the resultset from the real database. Note that the return type cannot be *java.sql.ResultSet* due to the fact that we can only transport serializable object through RMI. *java.sql.ResultSet* is, unfortunately, not serializable. The actual return type of this method

may vary depending on different implementations. This will be discussed in more detail in Section 5.3 of this report.

- *XMLReader*: This class is defined to read an XML mapping file.
  - *getTableDefinition*: This method takes in a table name as the parameter and returns an SQL select statement which generates the table in the reporting tool's prospective from the current underlying database.
  - *getColumnMapping*: This method also takes a table name as input parameter and returns a *Hashtable* object. This object contains the column mapping from the new version of the table to old version of the table. The *Hashtable* contains the <*key, value*> pair as: <*newColumnName, oldColumnName*>. If the table has not been modified, *newColumnName* is the same as *oldColumnName.*
  - *getAllTables:* This method returns all the table names that the reporting tool should see. That is, all the tables in the Virtual Database.

- QueryRewriter: This class is defined to rewrite the query generated by the reporting tool to the form that is compatible to the current database. Refer to Section 5.2 for detail description of query rewriting.

4 XML Database Mapping

In this chapter, we are going to see how the database mapping file is defined using XML, in particular, how to achieve the goal of mapping the data model in the current database to an old view of the data model that the reporting tool consistently sees.

Since the reporting tool maintains a constant view of the data model before any changes have been made, we therefore must have some way of mapping the current data model in the database to the reporting tool' s constant view. In achieving this, we define an XML file which contains all the necessary mapping information.

The XML mapping file can generally cope with all the possible data model changes in a database such as:

- Table rename
- Table deletion/creation
- Table conjoin/split
- Column rename
- Column deletion/creation
- Column relocation (move from one table to another)

The change of physical location of the database is not considered as a data model change, hence it is not dealt with in the XML mapping file. However, the location of the database is transparent to the reporting tool. This is done by way of a database location service provided by the middle tier. It is described in Section 5.1 of this report.

Before we go into the detail of such an XML mapping file, it is important to know that we have made the following assumptions in defining the XML mapping file:

- The XML mapping file must be defined manually. This means that this XML mapping file must be written by a human-being who is in charge of the database (e.g. a database administrator). At the current stage, this process cannot be automated.
- The person who defines the XML mapping file must have perfect knowledge of both the data model before changes (i.e. reporting's view to the data model) and the current data model in the database (i.e. the database with data model modified).
- The mapping file only contains the mapping information from the most-recent version of data model to the reporting tool's view of the data model. No information in between the two states of the data model (if any) is available in the mapping file (at the current stage of our design).

An simple example of such a mapping file is shown in Figure 4.1:

```
<VDB>
    <table name = “tableName”>
        <def>table definition in the database after data model is changed</def>

        <columns>
            <col name = "columnA">newColumnA </col>
            <col name = " columnB "></col>
            ...
        </columns>
    </table>
    ...
    <!-- more table follows -->
</VDB>
```

Figure 4.1 Simple example of XML mapping file

This XML mapping file is defined according to the following roles:

1. The XML mapping file defines the data model in the reporting tool’s view. It only defines all the tables/columns that exist in the reporting tool’s view.
2. The “name” attribute in the table tag defines the name of the table from the reporting tool’s point of view
3. The value of the “def” tag must be an SQL query that is executed against the current database. The execution of the query produces a *ResultSet,* with its contents equivalent to the table of the name defined in the name attribute (i.e. the old version of this table) as if the data model has not been changed. The value can be empty when the table has not been changed.
4. The “columns” element contains “col” sub-elements, which defines all the columns inside the current table, from the reporting tool’s view.
5. The “name” attribute of the “col” tag defines the name of the corresponding column, in the reporting tool’s point of view.
6. The value of the “col” tag specifies the new name of the column in the current data model (if any). If the column name has not been changed, the value should be empty, or same as the “name” attribute.

Strictly speaking, except defining the data model mapping, the XML mapping file also provides the reporting tool with the metadata information. This is because the data model in the current database is not a correct version of the data model that reporting tool wants, of course, provided that the data model has been changed at least once and not the same as the original version. For this reason, the standard JDBC implementation provided by the database vendor (e.g. MySQL Connector/J) must be modified. An example of such modification may be *getMetaData()* method from the *Connection* class, since the meta-data information cannot be obtained from the database directly. The only way to obtain the meta-data for the reporting tool (after any changes are made) is from the XML mapping file.

We are now going to present some concrete examples on how the database mapping is done.

Assume there is a brand new database, which has only one table defined: Customers. This data model (with only one Customers table) is the data model that the reporting tool sees, and the reporting tool's view remains the same irrespective to any further changes in the data model. Therefore, the data model with one Customer table is the one we are going to define in our ***Virtual Database*** (middle tier). Figure 4.2 illustrates the data mode of the ***Virtual Database***, i.e. the data model from the reporting tool's view.

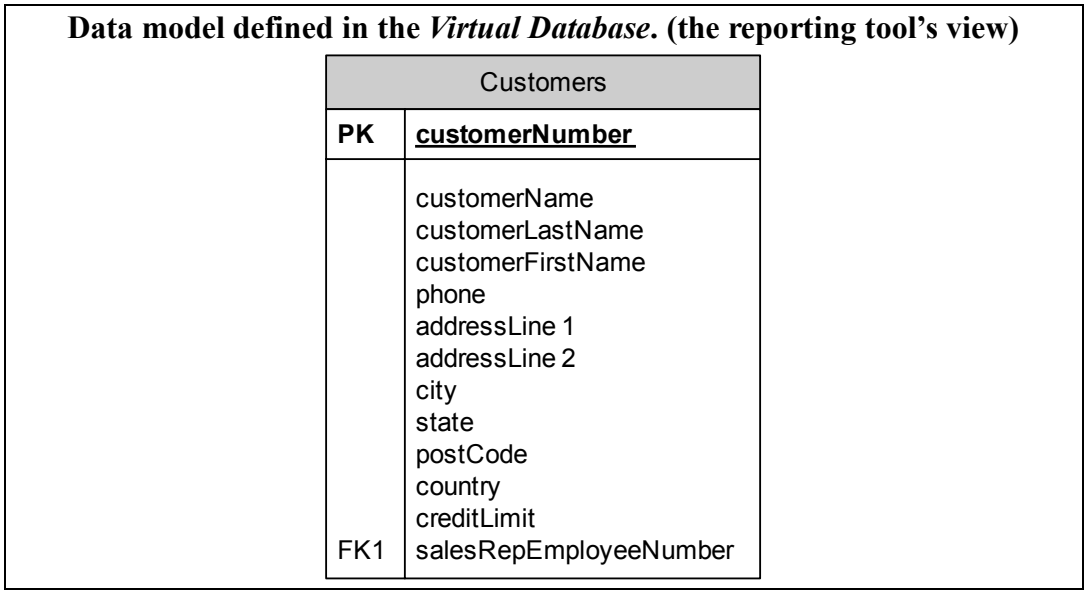


Figure 4.2 Data model (before change), reporting tool's data model, VDB data model (Example 1)

After a period of time, the data model is changed. The after-change data model in the real database is shown in Figure 4.3.

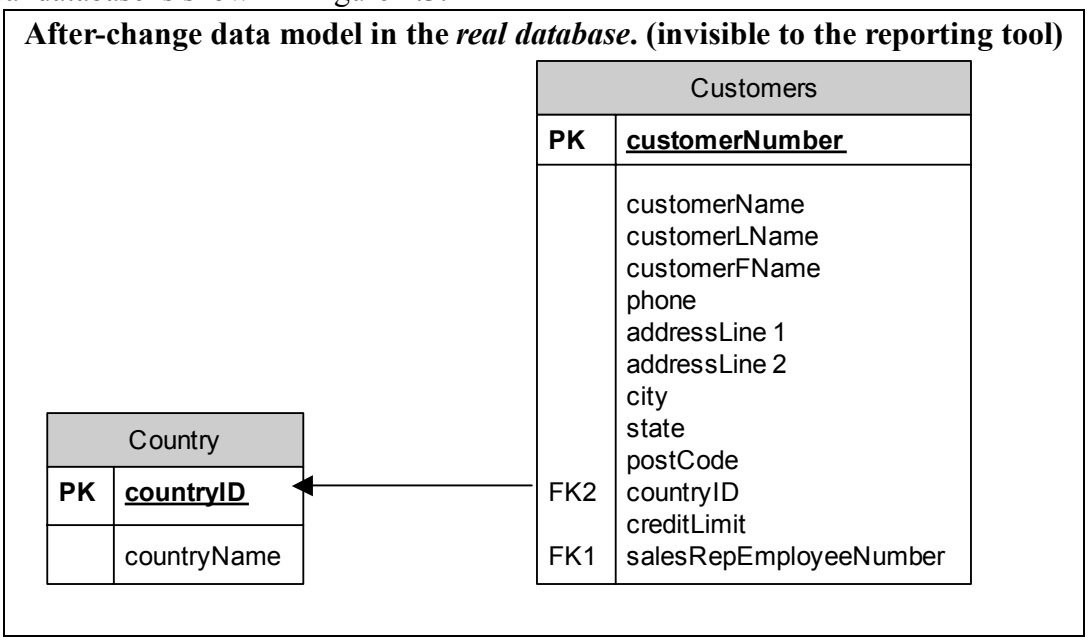


Figure 4.3 Data model (after change), data model in the real database (Example 1)

Comparing the above data models, we have observed that the following changes have been made in the new data mode (Figure 4.3 compared to Figure 4.2):

- A new Country table has been added with two columns: “countryID” and “countryName”.
- The old Customers table has been split into two tables: Customers and Country, with the “countryID” the foreign key linking them.
- The “country” column in the old data model has been moved to the new Country table in the new data model, and has been renamed to “countryName”. (Here, our assumptions applies: one must have perfect knowledge that “country” in the old data model is now being called “countryName” in the new data model)
- “customerLastName” and “customerFirstName” in the Customers table of the old model have now been renamed to “customerLName” and “customerFName” respectively.

Now, let us look at the XML file that maps these two data models:

```
1 <VDB>
2    <table name = "customers">
3        <def>
4            SELECT c.customerNumber, c.customerName, c.contactLName,
5            c.contactFName, c.phone, c.addressLine1,
6            c.addressLine2, c.city, c.state, c.postalCode,
7            cou.countryName, c.salesRepEmployeeNumber, c.creditLimit
8            FROM customers c, countries cou
9            WHERE cou.countryID = c.countryID
10       </def>
12       <columns>
13           <col name = "customerNumber"></col>
14           <col name = "customerName"></col>
15           <col name = "contactLastName">contactLName</col>
16           <col name = "contactFirstName">contactFName</col>
17           <col name = "phone"></col>
18           <col name = "addressLine1"></col>
19           <col name = "addressLine2"></col>
20           <col name = "city"></col>
21           <col name = "state"></col>
22           <col name = "postalCode"></col>
23           <col name = "countryName"></col>
24           <col name = "salesRepEmployeeNumber"></col>
25           <col name = "creditLimit"></col>
26       </columns>
27   </table>
28 </VDB>
```

a

b

Figure 4.4 XML Mapping File (Example 1)

The following issues should be emphasized in particular:

- Since the XML mapping file only defines the data model that the reporting tool presumes, there is no "Country" in the XML mapping file.
- As indicated by "a" in Figure 4.4, the "def" element defines an SQL query, that is to be executed against the new database (new data model). If this query is to be solely executed, it is going to produce a *ResultSet* that contains the exactly equivalent content to the Customers table before the data model has been modified (i.e. equivalent to the Customers table in Figure 4.2) except the fact that "contactLastName" and "contactFirstName" have been renamed, as a result, the column names in the result is not the same as the column names in the original data model. For now, let us just assume that it is not a problem. This will soon be fixed when we actually rewrite the query (refer to Section 5.2 later in this report).
- As indicated by "b" in Figure 4.4, since "contactLastName" and "contactFirstName" in the original data model have been renamed in the new data model. We map the changes by specifying the new name of the columns as the value of the "col" element.

The following is another more complicated example showing how the XML mapping file is defined. As in the previous example, the reporting tool's version of the data model is shown in Figure 4.5. And the data model in the real database is shown in Figure 4.6:

Data model defined in the *Virtual Database*. (the reporting tool's view)

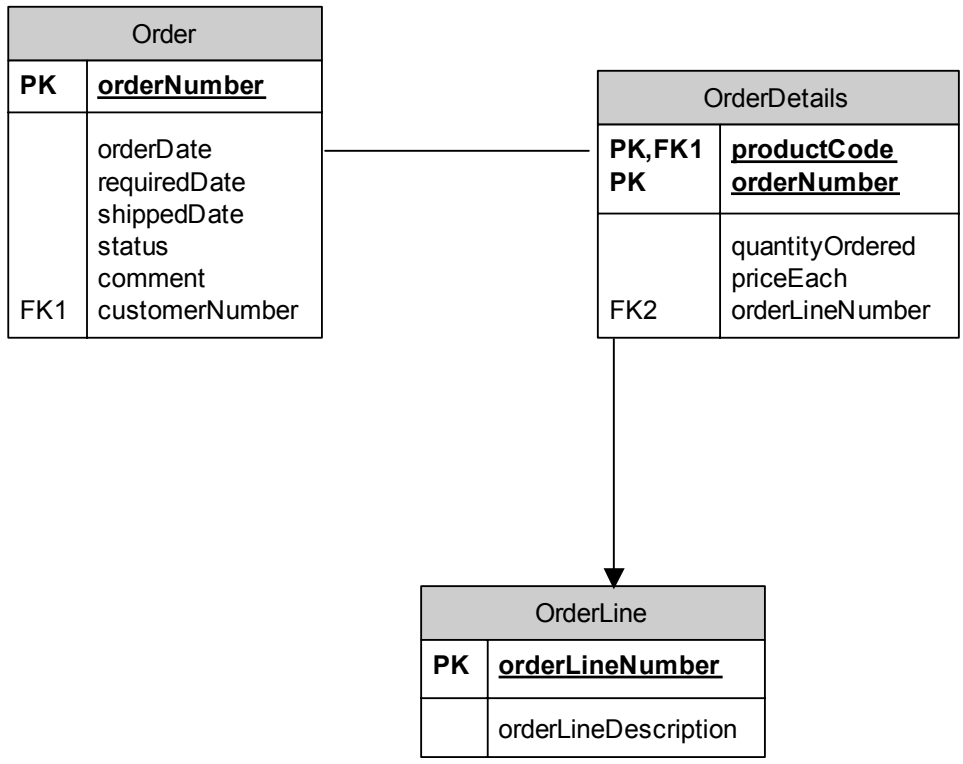


Figure 4.5 Data model (before change), reporting tool's data model, VDB data model (Example 2)

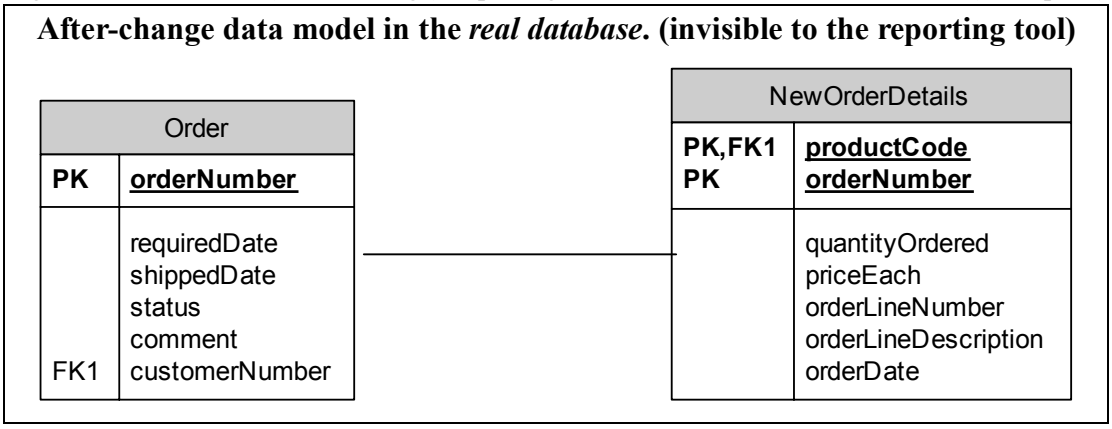


Figure 4.6 Data model (after change), data model in the real database (Example 2)

Again, we can observe that the following changes are made:

- The "OrderLine" table does not exist in the current data model any more.
- The table "OrderDetails" has been renamed to "NewOrderDetails"
- The table "OrderLine" has been combined into the table "NewOrderDetails" by moving the column "orderLineDescription" to table "NewOrderDetails".
- The column "orderDate" in table "Order" has been moved to table "NewOrderDetails"

The XML file that maps these changes of the two data models is illustrated in Figure 4.7:

```
1<VDB>
2    <table name = "orders">
3        <def>
4            SELECT DISTINCT orders.orderNumber,orderDate,
5            requiredDate,shippedDate,status,comments,customerNumber
6            from orders, neworderdetails
7            where orders.orderNumber = neworderdetails.orderNumber
8        </def>
9
10       <columns>
11           <col name = "orderNumber"></col>
12           <col name = "orderDate"></col>
13           <col name = "requiredDate"></col>
14           <col name = "shippedDate"></col>
15           <col name = "status"></col>
16           <col name = "comments"></col>
17           <col name = "customerNumber"></col>
18       </columns>
19   </table>
20
21   <table name = "orderdetails">
22       <def>
23           SELECT productCode,orderNumber,
24           quantityOrdered,priceEach,orderLineNumber
```

```
25              from neworderdetails
26          </def>
27
28          <columns>
29              <col name = "orderNumber"></col>
30              <col name = "productCode"></col>
31              <col name = "quantityOrdered"></col>
32              <col name = "priceEach"> </col>
33              <col name = "orderLineNumber"></col>
34          </columns>
35      </table>
36
37      <table name = "orderlines">
38          <def>
39              SELECT DISTINCT orderLineNumber,
40              orderLineDescription
41              From neworderdetails
42          </def>
43
44          <columns>
45              <col name = "orderLineNumber"></col>
46              <col name = "orderLineDescription"> </col>
47          </columns>
48      </table>
49</VDB>
```

a

Figure 4.7 XML Mapping File (Example 2)

Once again, let us look at the following issues in particular:

- Since the XML mapping file defines the data model that the reporting tool presumes. Although the table "OrderLine" has been removed from the current data model, is it still defined in the XML mapping file. The table still "virtually" exists if we execute the query inside the "def" tag under "orderline" (indicated by "a" in Figure 4.7). However, this table is now constructed by extracting information from table NewOrderDetails
- The table name change does not affect the XML definition, the only thing needs to be changed is to replace the new table name with the old table name whenever it appears inside the "def" tag, i.e. appears in the query that is going to executed against the current database. Such an example can be found in line 6, 7, 25, 41 from the XML mapping file where "orderdetails" has been replaced by "neworderdetails".

The above two examples showed how the XML mapping file is defined. In general, the syntax of the XML mapping file should conform to the following DTD:

```
<!DOCTYPE VDB[
    <!ELEMENT VDB (table*)>
    <!ELEMENT table (def, columns)>
    <!ATTLIST table name CDATA #REQUIRED>
    <!ELEMENT def (#PCDATA)>
    <!ELEMENT columns (col+)>
    <!ELEMENT col (#PCDATA)>
    <!ATTLIST col name CDATA #REQUIRED>
]>
```

Figure 4.8 DTD of the XML mapping file

5 Implementation

In this chapter, we are going to look at the implementation of the system, with a primary focus on the middle tier. In Section 5.1, we discuss the Database Location Service provided by the middle tier to achieve database location transparency between the reporting tool and the database. In Section 5.2, we are going to look at how the SQL query generated by the reporting tool is converted into a new query that produces the desired result for the reporting tool as if the data model is not changed. And finally, in Section 5.3, we are going to find out how exactly does the reporting tool obtain the data it requests, and how do different implementations differ in the way of passing the *ResultSet* to the reporting tool.

5.1 Database Location Service

As we have mentioned earlier, most reporting tool connects to a database directly in order to fetch the data of interest, meaning that the reporting tool must have the knowledge of the physical location of the database in order to connect. This physical location is specified by a database URL. It has the following syntax:

protocol:driver://host:port/databaseName

- Protocol specifies the protocol used to connect to a database. In our case, it should be "jdbc".
- Driver is the driver that enables the client to connect to a database. It is usually a vendor provided software, such as (MySQL Connector/J), but not limited to.
- Host and port, similar to HTTP protocol, specifies the database host IP address and port number to connect to.
- Database name, as its name suggests, is the name of the underlying database.

A concrete example can be:

jdbc:com.mysql.jdbc.Driver://192.168.23.21:80/northwind

By looking at the database URL, we observed that if the database location was changed, we must make change to the URL in order to connect to the desired database. In our system, however, we want to achieve the goal that any changes in the database side, is transparent to the reporting tool, including the database location. Hence, the traditional way of specifying database URL is not robust enough for our system.

Nevertheless, this problem can be easily solved by adding an extra service to the existing RMI service in the middle tier. Such service is a database location service. As its name suggested, this service provides the location of the current database, since the reporting tool connects to the middle tier instead of the database, it is therefore possible to reveal the location of the database to the reporting tool at runtime, as long

as the reporting tool finds the location of the middle tier. Since the middle tier acts as a ***Virtual Database*** interface for the reporting tool, it is implemented in such a way that the when the reporting tool is connected to the middle tier, it is very similar that the reporting tool is connecting to a real database except a little modification to the database URL.

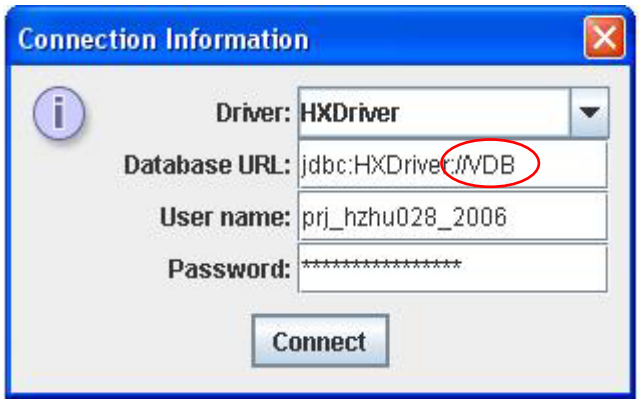


Figure 5.1 Connect the reporting tool to Virtual Database

Figure 5.1 is a typical dialog box that reporting tool prompts for database connection through JDBC Driver. In the example above, it is using a JDBC Driver called: HXDriver.

We also notice that instead of specifying the physical address of the database, we have made the database server name to be "VDB". In here, "VDB" is a virtual address of the database, namely, the ***Virtual Database***. As we mentioned in Section 3.4.1, the middle tier is expressed as a remote object by the interface of "VDB", "VDB" is in fact the name of the remote object that existed in the RMI registry.

The following code snippet shows how "VDB" is registered as a remote object from the server side, it exports the stub for the client to use at the client side (refer to Figure 3.7 in Section 3.4.1 for detail):

```
public class RegVDB{
    public static void main(String [] args){
        VDB stub = null;
        try{
            LocateRegistry.createRegistry(8081);
            VDBImpl vdb = new VDBImpl();
            stub = (VDB)UnicastRemoteObject.exportObject(vdb, 0);
            Registry reg = LocateRegistry.getRegistry(8081);
            reg.rebind("VDB",stub);
        }catch(Exception e){……}
    }
}
```

Figure 5.1 Code snippet of the *RegVDB* class

By doing so, the reporting tool can access the middle tier through the remote object, we can therefore reveal the physical location to the reporting tool at run-time by providing the following service:

```
/**
  * This method tells the RT where the actual DB is located at run time as
  * a service the RT connects to this DB at run time
  * By doing so, the location of the actual DB is transparent to the RT
  */
public String getDBInfo() throws RemoteException{
    return "192.168.23.43/northwind";    // DBServerIP / DBname
}
```

Figure 5.2 Code snippet of database location service

This service, in essence, is a remote method. The return value of this method is a *String* containing the database server IP address and the database name. That is, the second half of the database URL.

This method is designed to be called by the JDBC driver at the reporting tool's side (client) in order to find out the current physical location of the database. By providing this service, whenever the location of the database changes, the reporting tool's setting remains the same since all it needs to do is to call this remote method to find out the up-to-date location of the database and connect to that address. The server side, however, need to change the current database location it is pointing to in the cases of database location changes (e.g. if the server has been moved to a new location with the IP address: 203.231.32.34, the return value should be: 203.231.32.34/northwind).

The following code snippet shows how, the JDBC driver, in the reporting tool's side, find out where the current location of the database:

```
//@modify: modify url, so that VDB maps to a real DB address
int startInx = url.indexOf("//") + 2;
String vdbName = url.substring(startInx);
try{
    //Look for virtual DB in RMI registry
    Registry reg = LocateRegistry.getRegistry("localhost",8081);
    vdb = (VDB)reg.lookup(vdbName);
    url = vdb.getDBInfo(); //find out the current database location
}
catch(Exception e){
    return null;
}
//@end modify
```

Figure 5.3 Customized JDBC driver looks for the current DB location

5.2 Automatic Query Rewriting

Let us once again review what our system needs to achieve: When the reporting tool generates a request (SQL statement), the middle tier gets hold of the query through the JDBC Driver. The middle tier cannot execute this query against the database due to the fact that the data model resides in the real database is possibly not the same as the data model that the reporting tool presumes.

Therefore, the middle tier has a query rewriting facility which rewrites the query, originates from the reporting tool, to a new form of query which is compatible to the current data model but produces the results that the reporting tools wants as if the data model is not modified. Another word, we need to execute such a new query, to extract the old information from the "after-change" version of the data model.

This is done by having a *QueryRewriter* class inside the middle tier implementation. The only assumption made in the current implementation is that the SQL statement can only contain "SELECT" statement with "FROM" and "WHERE" clauses. This is a reasonable assumption due to the fact that we are dealing with the requests issued by the reporting tool. It is unlikely to have "UPDATE", "INSERT" in the query whatsoever. Keyword such as "ORDERBY" is not dealt with at the current stage of implementation.

The *QueryRewriter* class has a single *public* method *rewrite(),* it reconstruct the incoming query by rewriting "SELECT", "FROM", and "WHERE" clauses respectively, the return value is the reconstructed SQL statement of type *String* .

The *QueryRewriter* class has an instance of *XMLReader*, it extracts the information from the XML mapping file as the content to be rewritten. Figure 5.4 shows the code snippet of the *rewrite()* method:

```
/* This method rewrite the query originated from the reporting
 * tool to a query which is compatible to the current DB
 * @param: old query
public String rewrite(String query){
    this.query = query;
    String newQuery = "";
    if(query.toLowerCase().startsWith("select")){
        newQuery += rewriteSelectClause(getSelectClause()) + " ";
        newQuery += rewriteFromClause(getTableNames()) + " ";
        newQuery += rewriteWhereClause(getWhereClause());
        return newQuery.trim();
    } else return query;
}
```

Figure 5.4 Code snippet of the *rewrite()* method.

The query rewriting is done according to the following roles:

- First it checks whether this query is a SELECT statement, if it is not, the returned query is the incoming query without any modifications made. This is not avoidable since we only want to deal with the "real" query that requests for concrete data from the database. It is very common for most reporting tools to query for meta data, typically at its first connection to the database, by issuing query such as "SHOW TABLES", this kind of query is out of our interest and hence it is necessary to avoid them in order not to produce unnecessary complications and mistakes.

- Secondly, it processes the "SELECT" clause of the SQL statement. For each of the fields (columns) appears in the SELECT clause, if the column name is not specified in the form: *tableName.columnName,* it looks through the XML mapping file to find out which table (specified in the FROM clause) this column belongs to, and rewrite the column name into the format of: *tableName.columnName.* It also checks to see weather the name of the column is changed. If not change is made, keep it the same, otherwise change the column name into the new column name of the current data model and add an "AS" keyword followed by the old column name before the data model is modified.

  The following pseudocode illustrates how this is done:

```
for each column col in select clause
   if tableName is not specified
      tableName := findTableNameFromXml(col);
      col := tableName + "." + col;
   end if

   if col is renamed
      newName := findNewColumnNameFromXml(col);
      col := tableName + "." + newName;
      col += "AS" + oldName;
   end if

end for
```

Figure 5.5 Pseudocode: rewriting SELECT clause

- Thirdly, it processes the FROM clause of the originated query. The rewriting is done by replacing each of the table name specified in the FROM clause by a sub-query. The sub-query is obtained from the "def" sub-element within the table element, from the XML mapping file. This works because the query in the "def" tag generates a virtual form of the table as if the model is not changed.

The following pseudocode illustrates how this is done:

```
for each tableName is the FROM clause:
   String subQuery = “”;
   def := lookForDefInXml(tableName);
   subQuery += def;
   subQuery += “AS” + tableName; // AS the old table name
end for
```

Figure 5.6 Pseudocode: rewriting FROM clause

- Lastly, it will processes the WHERE clause. The rewriting of WHERE clause follows a similar role to rewriting SELECT clause. It basically finds all the column names in the WHERE condition and replace them with the new column names (if any) in the current data model. However, it requires more work since there are unknown number of conditions separated by keyword “AND” or “OR”. We need to extract all the conditions out before further process them.

The following figure shows the pseudocode of rewriting WHERE clause:

```
String[] conditions = split the WHERE clause according to {AND, OR};

for each condition cond in conditions
   String sign := find the sign in cond // can be <, >, =, <=, >= etc
   String left := cond.left(sign); // left hand side of condition
   String right := cond.right(sign); // right hand side of condition

   for col is {left, right}
      if tableName is not specified
         tableName := findTableNameFromXml(col);
         if(tableName != null)
            col := tableName + “.” + col;
        else // no table name matching
            col := col // do not change, it is an assignment such as
                       // table. col =’hello’
      end if
      if col is renamed
         newName := findNewColumnNameFromXml(col);
         col := tableName + “.” + newName;
      end if

   end for
end for
```

Figure 5.7 Pseudocode: rewriting WHERE clause

After the above steps, we will have a new query which is ready to be executed against the current database, but produce the desired result that the reporting tool requests, i.e. if this rewritten query is executed against the current database, we will then have a *ResultSet* object that contains the data that the reporting tool requests.

The content of the ResultSet will be exactly the same if we were to execute the original query against the database before any changes are made.

In the next section, we are going to look at how can we, use different implementations, to transport this *ResultSet* back to the reporting tool.

5.3 Getting The ResultSet

In Section 5.2, we have discussed how the incoming query can be rewritten to be compatible to the current data model. But one question can be raised: who is going to execute this rewritten query?

This question has already been answered in Section 3.3, where we present the architecture of the middle tier. Different architectures have different way of handling the rewritten query, and hence different ways of obtaining the final *ResultSet* containing the data that the reporting tool is interested in.

Basically, there are two scenarios:

1. **The middle tier is responsible of obtaining the *ResultSet* and transporting it back to the reporting tool.**
2. **The middle tier is not responsible of obtaining the *ResultSet*. The reporting tool itself obtains the *ResultSet*.**

5.3.1 Approach One – Execute the rewritten query from middle tier

Let us first of all look at the first scenario.

When the rewritten query is in place, since the DataFetcher in the middle tier maintains a connection to the current database, we can therefore execute the rewritten query from the DataFetcher class. After the execution, a *ResultSet* Object is returned, containing the result the reporting tool wants. This *ResultSet* Object must be transported to the reporting tool's side for further processing. However, we need to keep in mind that the middle tier is in fact a remote object. Every method we called from the reporting tool's side is a remote method invocation. RMI requires that both parameter and return type must be either primitive type or object that is serializable. In our case, the *ResultSet* object obtained from the remote side, is not serializable, hence it cannot be transported to the client side.

We therefore, need to process the *ResultSet* object at the remote side (i.e. middle tier), and pack the data into another object that is serializable for it to be transported through RMI to the reporting tool, more strictly, to the customized JDBC driver used by the reporting tool. Upon receiving this serializable "*ResultSet*", the customized JDBC driver needs to re-process those data encapsulated in the object to reconstruct a "real" *ResultSet* object for the reporting tool to use.

Figure 5.8 shows a detail diagram of how this approach works:

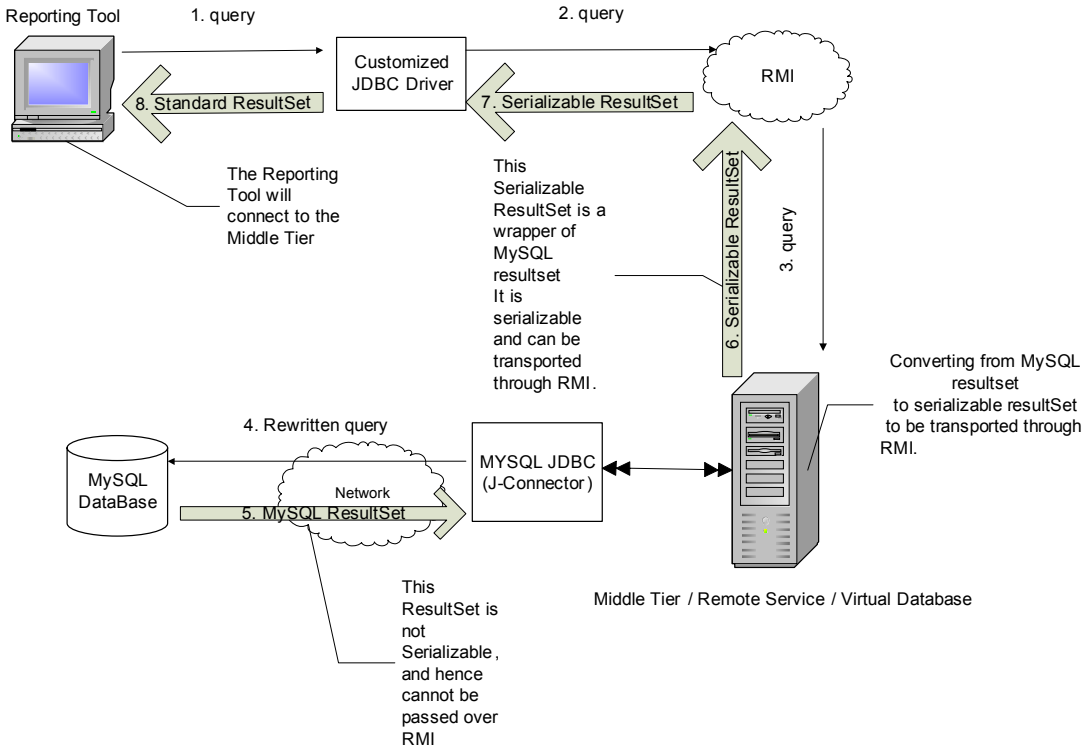


Figure 5.8 Getting ResultSet through RMI

It is clear that as long as we can convert the *ResultSet* obtained from Step.5 into the serializable *ResultSet* shown in Step 6 and 7, there will be no problem with this approach.

So far, we have found three ways of converting the JDBC *ResultSet* object into a serializable object. Each of those implementations requires modifications to the *DataFecther* class, since *DataFetcher* is responsible for converting the *ResultSet* it obtained into a serializable form. We are going to look at each of them in detail. All of them process the JDBC *ResultSet* object at the remote side once, and convert it to a serializable object to be transportable through RMI.

- Packing the *ResultSet* into a *Map* object:

  "The *Map Interface* is a member of the Java Collection Framework. It provides three *collection views*, which allow a map's contents to be viewed as a set of keys, collection of values, or set of key-value mappings. The *order* of a map is defined as the order in which the iterators on the map's collection views return their elements." [JAVA 5.0 API]

  In our implementation, a *LinkedHashMap* is typically used. The *Map* object contains the <key, value> pair as <columnName, listOfColumnValues>, where *listOfColumnValues* is an *ArrayList* object contains all the values within one column.

A graphical view of the Map object is shown in Figure 5.9:

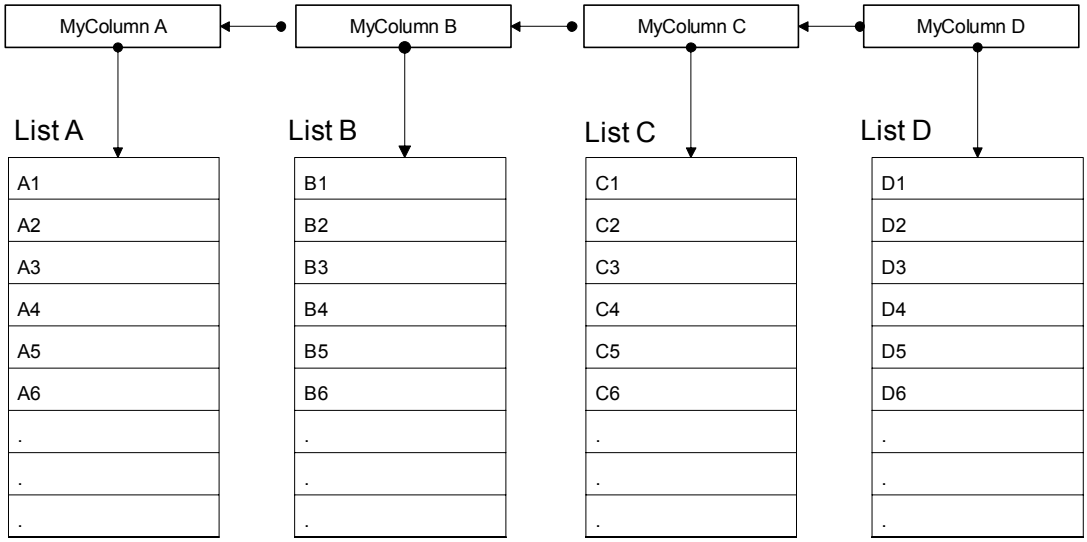


Figure 5.9 Graphical representation of *Map* Object for encapsulating the ResultSet

The following code snippet shows how the Map object is constructed programmatically (inside *DataFetcher* class):

```
/** Helper method that maps a ResultSet into a map of column lists
  * @param ResultSet
  * @return map of lists, one per column, with column name as the key */
private Map toMap(ResultSet rs) throws SQLException{
    ResultSetMetaData meta = rs.getMetaData();
    Map columns = new LinkedHashMap();    // Set up the map of columns
    int numWantedColumns    = meta.getColumnCount();
    for (int i = 1; i <= numWantedColumns; i++){
        List columnValues     = new ArrayList();
        columns.put(meta.getColumnName(i), columnValues);
    }
    while (rs.next()){
        for (int i = 1; i <= numWantedColumns; i++){
            String columnName     = meta.getColumnName(i);
            Object value          = rs.getObject(columnName);
            List columnValues     = (List)columns.get(columnName);
            columnValues.add(value);
            columns.put(columnName, columnValues);
        }
    }
    return columns;
}
```

Figure 5.10 Code snippet of the *toMap()* method.

- Converting the *ResultSet* object into *CachedRowSet* object
  "A CachedRowSet object is a container for rows of data that caches its rows in memory, which makes it possible to operate without always being connected to its data source. Further, it is a JavaBeans™ component and is scrollable, updatable, and serializable. A CachedRowSet object typically contains rows from a result set.

  A *CachedRowSet* object is a *disconnected* rowset, which means that it makes use of a connection to its data source only briefly. It connects to its data source while it is reading data to populate itself with rows and again while it is propagating changes back to its underlying data source. The rest of the time, a CachedRowSet object is disconnected, including while its data is being modified. Being disconnected makes a RowSet object much leaner and therefore much easier to pass to another component." [JAVA 5.0 API]

  The above two paragraphs are obtained from Java API (5.0). It describes the *CachedRowSet* object from package *javax.sql.rowset*. We used the standard implementation provided by Sun Microsystems in our implementation, therefore, the detailed implementation is unknown. However, it is worth mentioning that *CachedRowSet* does not maintain a continuous connection to the data source. This suggests that this implementation uses somehow similar approach to the *Map* approach described earlier, i.e. the *ResultSet* is processed once at the remote side before it can be transported to the reporting's side.

  The following code fragment shows how the *CachedRowSet* object is obtained (inside *DataFetcher* class):

```
/**
 * This method executes the rewrittern query
 * and return the ResultSet as a CachedRowSet object which can be
 * serialized into the RT site
 * @ param rewrittern query
 */
public CachedRowSet getRowSet(String sql){
    CachedRowSetImpl crs = null;
    try{
        crs = new CachedRowSetImpl();
        Statement stmt = conn.createStatement();
        ResultSet rs =  stmt.executeQuery(sql);
       crs = new CachedRowSetImpl(); // The implementation is
                                     // provided by Sun Microsystem.
        crs.populate(rs);
```

```
        catch(Exception e){
            e.printStackTrace();
        }
        return crs;
    }
```

Figure 5.11 Code snippet of constructing *CachedRowSet* object from DataFetcher class.

- Converting the *ResultSet* object into XML document

Since all the data the reporting tool wants is already contained in the *ResultSet* object, we can therefore process the *ResultSet* at the server side and convert the data into an XML document as long as we have a proper XML schema (or DTD) defined.

The interface *WebRowSet* inside *javax.sql.rowset* package offers a way of converting JDBC *ResultSet* into an XML document. Such an XML document contains three main elements: <*properties*>, <*metadata*> and <*data*>. The XML document conforms to the XML schema defined at the following address:

http://java.sun.com/xml/ns/jdbc/webrowset.xsd

In our implementation, we also used the standard implementation provided by Sun Microsystems. The following code fragment shows how the *WebRowSet* object is constructed using the JDBC *ResultSet* (inside *DataFetcher* class):

```
/**
 * This method executes the rewrittern query
 * and return the ResultSet as a CachedRowSet object
 * which can be serialized into the RT site
 * @param rewrittern sql
 */
 public WebRowSet getRowSet(String sql){
   WebRowSetImpl wrs = null;
   try{
        Statement stmt = conn.createStatement();
        ResultSet rs =   stmt.executeQuery(sql);
        wrs = new WebRowSetImpl(); // The implementation is provided
                                   // by Sun. Microsystems
        wrs.populate(rs);
   }
   catch(Exception e){……}
   return wrs;
 }
```

Figure 5.11 Code snippet of constructing *WebRowSet* object from *DataFetcher* class.

To summarize the above three scenarios, in order to obtain the data from the remote side to the reporting tool's side, we need to extract all the information from the *ResultSet* and convert it into a serializable object. This kind of approach is however costly as far as efficiency is concerned (it is discussed in the next Chapter).

5.3.2 Approach Two – Execute the rewritten query from reporting tool

It would be much better if the *ResultSet* object can be obtained by the reporting tool without being transported through RMI. Hence, we introduce the second approach. In this approach, the middle tier is not responsible for fetching the data, and hence no *DataFetcher* is needed. What the middle tier does is to receive the SQL query and modify it so that it can be executed against the current data model. After the query is rewritten by the middle tier, the query itself, rather than the *ResultSet* produced by executing the query, is transported through RMI to the reporting tool (more accurately, to the customized JDBC driver). The customized JDBC driver which is sitting at the reporting tool's side (client), upon receiving the rewritten query, executes the new query against the current database and fetches the result. Note that the *ResultSet* now, is a standard JDBC ResultSet, hence no serialization is needed. The following diagram illustrates this:

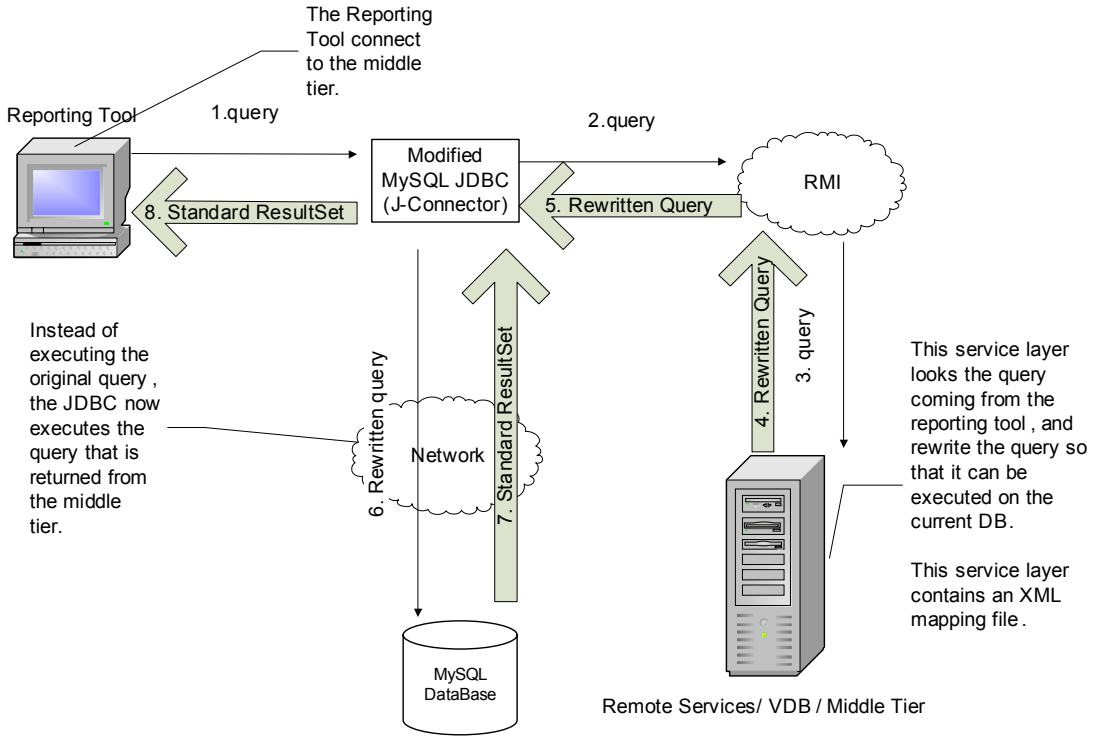


Figure 5.12 Getting ResultSet directly by executing the rewritten query

This approach, is very similar to the previous approach, the only difference is that after the middle tier has got the rewritten query, it does not execute the query against

the current database, instead, it passes this rewritten query to the JDBC driver used by the reporting tool though RMI. (Refer to the sequence diagrams in Figure 3.3 and Figure 3.5 for comparison)

We can also see that in this approach, there is only one JDBC driver needed due to the fact that the middle tier does not keep a connection to the actual database. The only JDBC driver, is very likely to be a vendor supported JDBC Driver (e.g. in the diagram, it is a MySQL Connector/J since the database is a MySQL database). However, this JDBC driver needs to be modified from the original version, typically, the following modifications need to be considered:

- Has an instance of *VDB* class as the remote stub.
- Calls the *getQuery()* remote method on *VDB* to obtain the rewritten query before executing it.
- All the meta-data information cannot be obtained from the database directly. This information must be obtained from the XML mapping file resides on the middle tier since the data model in the database may be different from what the reporting tool expects.

The following code snippet shows how the rewritten query is obtained inside *executeQuery()* method of *Statement* class (This is a modified version of the MySQL JDBC implementation):

```
public java.sql.ResultSet executeQuery(String sql)
    throws SQLException {
        checkClosed();
        try{
            if(vdb != null){
                // this method returns the rewritten query
                sql = vdb.getQuery(sql);    //vdb is the remote stub
            }
        }
        catch(Exception e){
            e.printStackTrace();
        }
        // the query executed will be the rewritten query
        ……
}
```

Figure 5.13 Code snippet of obtaining the rewritten query from JDBC.

Up until now, there are two approaches of getting the *ResultSet,* the first approach obtains the *ResultSet* through the middle tier (Figure 5.8). The second one obtains the *ResultSet* directly from the database by executing the rewritten query (Figure 5.12). The following table is a comparison between the two approaches:

Approach / *Issue*	**Approach One**	**Approach Two**
Middle tier maintains connection to database	**Yes**	No
ResultSet Serialization	**Yes**	No
No. JDBC Drivers	**2**	1
Modifications to vendor provided JDBC Driver	**No**	Yes
Query rewriting	**Yes**	Yes
Passing ResultSet through RMI	Yes	No

Table 5.1 Comparison between two approaches

6 Performance Evaluations

So far, we have looked at all the implementations of the middle tier. But we are still not sure which implementation gives us the best performance quality. If we were to use this middle tier into practice, we also want an implementation which not only does the "job" but also retrieve the information efficiently. As mentioned previously, the biggest problem that influences the performance of the underlying system is how the *ResultSet* object is handled. In this chapter, we are going to compare different implementations in terms of their performances. In Section 6.1, we specify the testing environment. From Section 6.2 to 6.4, we are going to see the performances of the first approach discussed above, where the *ResultSet* is obtained through the RMI. In Section 6.5, we are also going to look at the performance of the second approach, where the *ResultSet* is passed directly from the database. Finally, in Section 6.6, we are going to compare both of the approaches and conclude on which implementation gives the best performance quality as far as speed is concerned.

Before we look at the testing results of different implementations, let us first of all, look at how the timing is done in order to give a fair comparison among all the implementations.

The timing is conducted in the reporting tool's side. The timer starts right before the command "*statement.executeQuery(someQuery)*" where the reporting tool requests its JDBC driver to fetch the *ResultSet* for further processing and ends right after the *ResultSet* has been successfully fetched by the JDBC driver and accessed each object in the *ResultSet* **once** by a dummy command**:**

Object obj = resultSet.getObject(columnNumber)

The reason that we need such a dummy step of accessing each object inside the *ResultSet* once is because: for the implantations which packs *ResultSet* into *Map*, *CachedRowSet*, and XML document. The resulting rowset does not keep a connection to the data source (i.e. all the element is already cached into the rowset before it is being transported), while the standard JDBC *ResultSet* (the approach executing rewritten query directly from reporting tool) keeps persistent connection to the data source, i.e. for standard JDBC *ResultSet*, at the time the *ResultSet* is obtained, we do not have all the data cached inside the *ResultSet*, in fact, only every time the *next()* method is called, the JDBC driver will then fetch a new row from the current data source. Therefore, it is not a fair comparison to only time the period of getting the *ResultSet*, but not process each element inside the *ResultSet*. Therefore, in the timing routine, we use this dummy step to run a simulation of processing all the elements inside the *ResultSet* object for both of the approaches.

In order to get an average value, the above timing routine is to be executed for 100 times to get an average reading with high precision. The following segment of code

shows how the timing is conducted inside the reporting tool:

```
……
int limit = 100; // No. of time to run the loop, to give an average value
while(c < limit){
    long start = System.nanoTime(); // start timing
    resultSet = statement.executeQuery(query);
    while(resultSet.next()){
        for (int i = 1; i <= numberOfColumns; i++) {
            Object obj = resultSet.getObject(i); // dummy step
        }
    }
    long end = System.nanoTime(); // end timing
    long diff = end - start;
    sum += diff;
    c++;
}
resultSet.close();
System.out.println("average time is: "+(sum/limit)/100000.0 + " mill seconds");
……
```

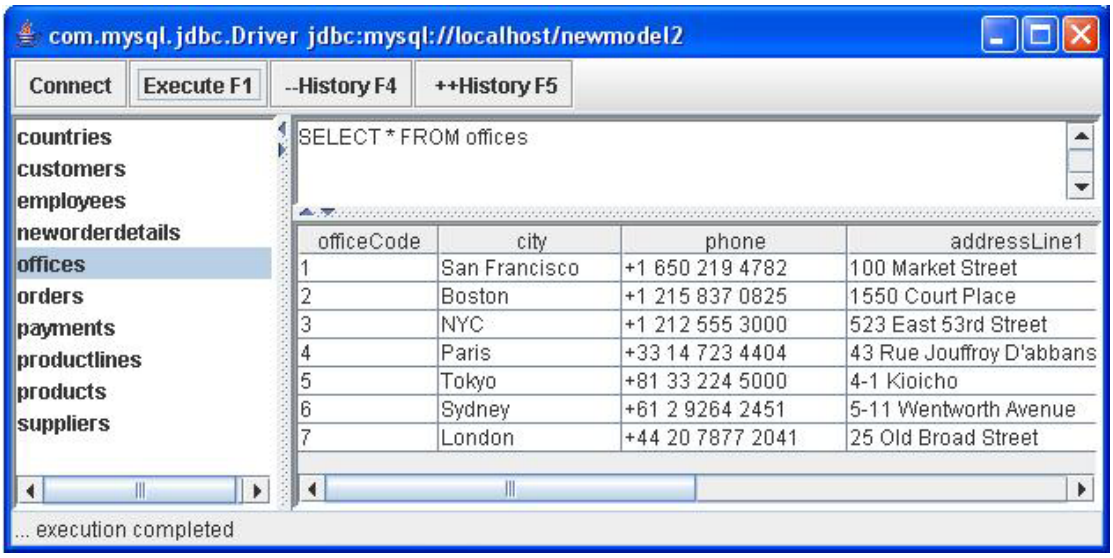
Figure 6.1 Code snippet of timing mechanism.

From Section 6.2 to Section 6.6, each of the implementations (refer to Section 5.3) are tested using the above timing mechanism. In order to compare our implementation with the traditional reporting tool implementation, the traditional implementation of the reporting tool (i.e. not through middle tier, every time the data model change, the query issued by the reporting tool changes accordingly) is also tested under the same timing mechanism in order to identify any possible overhead in our implementations.

6.1 Testing Environment

The test is conducted on a database with data model changes by comparing the performance of our implementations and the traditional reporting tool. In our implementation, the query issued by the reporting tool does not reveal any data model changes. In the traditional implementation, the query issued by the reporting tool is manually changed to conform to the data model changes, but it would otherwise produce the same result.

For the sake of simplicity, a simplified version of reporting tool is used, where an SQL statement can be entered into the program to simulate the setting that the reporting tool is configured to. The following is a screen-shot of the testing program:



Screen-shot of testing program

For our implementations, the tests are conducted in such a way that the testing program, middle tier and the database are residing on different machines (same configurations, see table below).

For the traditional reporting tool implementation, the testing program and the database are residing on different machines (same configurations, see table below).

The testing environment is summarized in the following tables:

System Environment:

Operating System	Windows XP Professional (5.1, Build 2600) Service Pack 2
Processor	Intel(R) Pentium(R) 4 CPU 3.40GHz (2 CPUs)
Memory	2048MB RAM
Java Version	Java (TM) 2 Runtime Environment, Standard Edition (build 1.5.0_06-b05)

Database Environment:

Name	MySQL 5.0.20
Host URL	http://studwww.cs.auckland.ac.nz/
Database Driver	MySQL Connector/J 5.0.2
Driver Details	JDBC-3.0, "Type 4" driver

Network Environment:

Protocol	TCP/IP
Type	Full Duplex
Speed (PC to router)	100Mbps
Speed (between router)	1Gbps

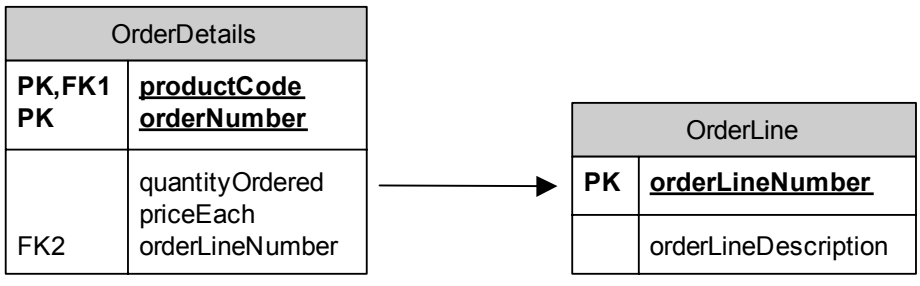
When the tests are conducted, the related table defined in the real database is:

Related table in the *real database*

OrderDetails	
PK,FK1 **PK**	**productCode** **orderNumber**
	quantityOrdered priceEach orderLineNumber orderLineDescription orderDate

The mapping view of the table that the reporting tool views is (i.e. ***Virtual Database***):

Related table in the *Virtual Database*



The query used to simulate the reporting tool's setting is (i.e. the query issued by the reporting tool):

```
SELECT orderNumber, productCode,
        orderdetails. orderLineNumber, orderLineDescription
FROM    orderdetails, orderlines
WHERE orderdetails.orderLineNumber = orderlines.orderLineNumber
```

The tests are conducted by retrieving a *ResultSet* with different sizes ranging from 40Kb to 400Kb approximately (raw data size, no meta-data size included). The different sizes of the *ResultSet* is obtained by changing the number of rows inside the table "OrderDetails" in the real database, from 1000 rows to 10,000 rows.

6.2 Serialize using Map Object

The following table illustrates the comparison between the middle tier implementation which encapsulates the *ResultSet* into a *Map* object and the traditional reporting tool implementation:

ResultSet Size (Kb) / Method	**39.8**	**79.7**	**119.5**	**159.4**	**199.2**	**239.1**	**278.9**	**318.8**	**358.6**	**398.5**
Middle Tier Implementation (*Map*) /ms	38.8	68.7	101.6	122.3	157.0	197.8	205.9	230.9	266.6	284.8
Traditional Reporting tool /ms	14.9	32.5	44.4	55.1	68.3	80.9	82.5	89.3	101.8	115.7

Table 6.1 Testing data (*Map* v.s. traditional reporting tool).

The following graph is plotted according to the above data:

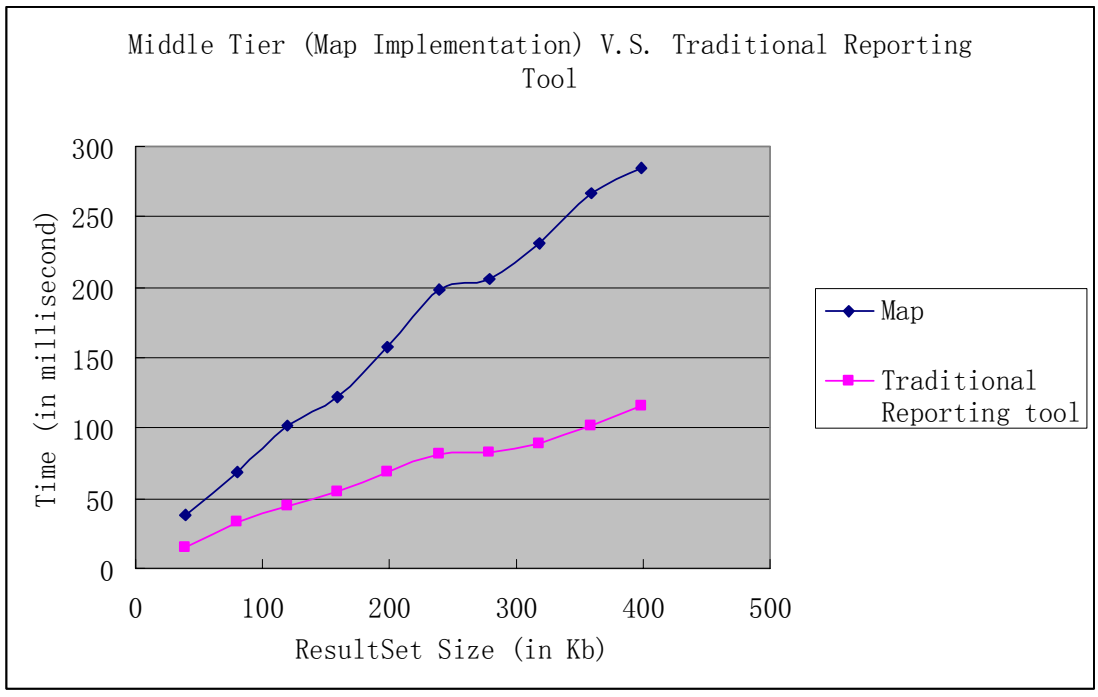


Figure 6.2 Plot: Map Implementation against Traditional Reporting Tool

From Figure 6.2, we can observe both the traditional implementation and our middle tier implementation produce result nearly linear, i.e. as the *ResultSet* increases its size, the time it take increases as well. It seems that the Map implementation tends to increase at a faster rate than that of the traditional implementation. There is also significant overhead introduced by the Map implementation as expected, this overhead is very likely produced due to the time it takes to process the *ResultSet* object at the remote side (server side) before it can be transported through RMI.

6.3 Serialize using CachedRowSet Object

The following table illustrates the comparison between the middle tier implementation which encapsulates the *ResultSet* into a *CachedRowSet* object and the traditional reporting tool implementation:

ResultSet Size (Kb) / Method	**39.8**	**79.7**	**119.5**	**159.4**	**199.2**	**239.1**	**278.9**	**318.8**	**358.6**	**398.5**
Middle Tier Implementation (*CachedRowSet*) /ms	67.5	123.7	157.4	202.5	276.6	290.3	342.7	413.6	450.4	510.6
Traditional Reporting tool /ms	14.9	32.5	44.4	55.1	68.3	80.9	82.5	89.3	101.8	115.7

Table 6.2 Testing data (CachedRowSet v.s. traditional reporting tool).

The following graph is plotted according to the above data:

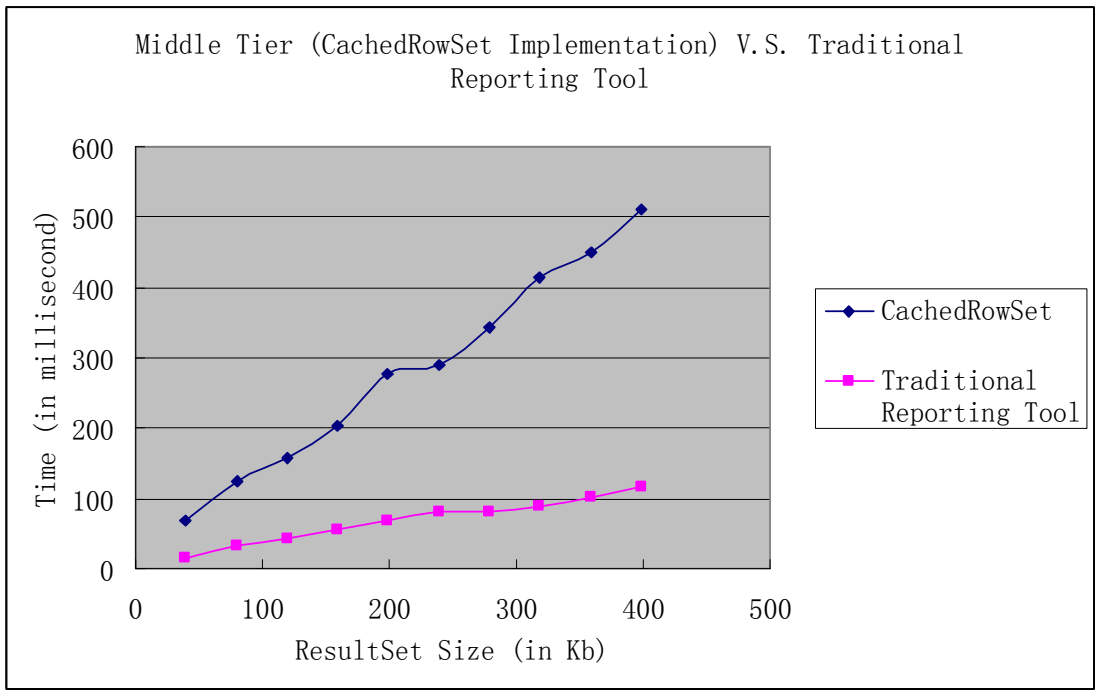


Figure 6.3 Plot: CachedRowSet Implementation against Traditional Reporting Tool

Again, as expected, the *CachedRowSet* implementation also seems to take a significant amount of time serializing the *ResultSet*. As the size of the ResultSet increases, the time it takes to serialize increases at a very fast rate.

6.4 Serialize by converting ResultSet into XML document (WebRowSet)

The following table illustrates the comparison between the middle tier implementation which encapsulates the *ResultSet* into an *XML* document and the traditional reporting tool implementation:

ResultSet Size (Kb) / Method	**39.8**	**79.7**	**119.5**	**159.4**	**199.2**	**239.1**	**278.9**	**318.8**	**358.6**	**398.5**
Middle Tier Implementation (XML) /ms	71.6	122.3	153.5	208.6	261.2	316.6	351.9	436.5	479.7	540.7
Traditional Reporting tool /ms	14.9	32.5	44.4	55.1	68.3	80.9	82.5	89.3	101.8	115.7

Table 6.3 Testing data (XML v.s. traditional reporting tool).

The following graph is plotted according to the above data:

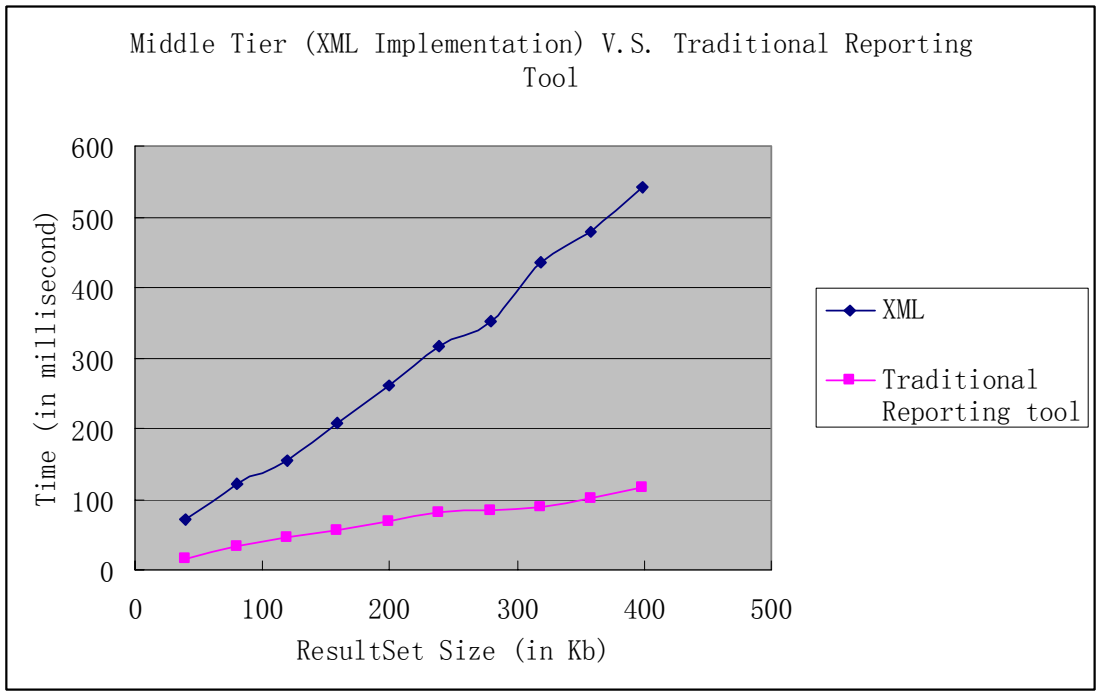


Figure 6.4 Plot: XML document (WebRowSet) Implementation against Traditional Reporting Tool

Although XML seems to be widely used for serialization of dataset, this implementation does not seems to be favorable either, the overhead introduced is quite large and increase in a very fast rate as the *ResultSet* size increases.

6.5 Executing rewritten query directly

Up to now, we have looked at the performance of all the implementations which convert the *ResultSet* into a serializable object before transporting them trough RMI. Although the time ranges are all within millisecond, compared to the traditional reporting tool implementation, all of them seem to introduce rather significant overheads. In this section, we are going test the performance of the second approach, where the *ResultSet* does not go through RMI, only the rewritten query get passed through RMI (Section 5.3.2). If we can have a result with relatively low overhead compared to the previous testing, not only we have found an improvement to our solution, we also can prove that the overhead is indeed produced by *ResultSet* serialization.

The following table illustrates the comparison between the middle tier implementation which executes the rewritten query directly from the reporting tool and the traditional reporting tool implementation:

ResultSet Size (Kb) / Method	**39.8**	**79.7**	**119.5**	**159.4**	**199.2**	**239.1**	**278.9**	**318.8**	**358.6**	**398.5**
Middle Tier Impl (rewritten query from RT) /ms	20.2	42.8	57.6	62.1	71.3	80.3	94.5	102.3	106.1	113.7
Traditional Reporting tool /ms	14.9	32.5	44.4	55.1	68.3	80.9	82.5	89.3	101.8	115.7

Table 6.4 Testing data (execute rewritten query from RT v.s. traditional reporting tool).

The following graph is plotted according to the above data:

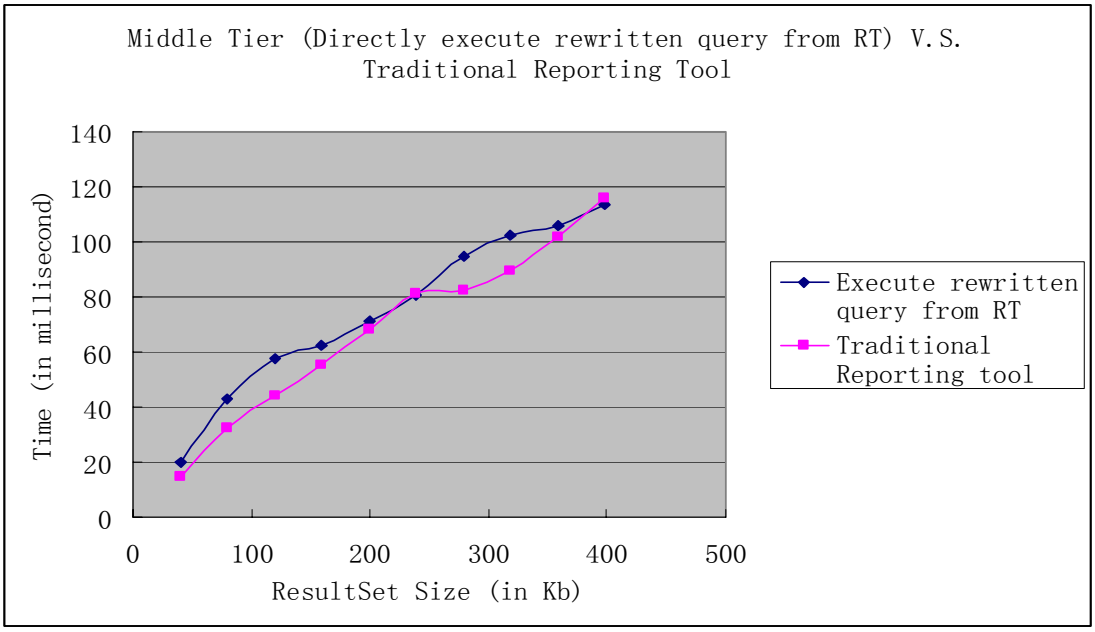


Figure 6.5 Plot: Execute written query from RT against Traditional Reporting Tool

Clearly from Figure 6.5, we can see that the second approach, by passing the rewritten query through RMI and execute the query from the reporting tool directly has decreased the time dramatically. In fact, the time difference with the traditional reporting tool implementation is nearly negligible. Meanwhile, we also proved that the overhead in our first approach is indeed produced by the *ResultSet* serialization.

6.6 All together

In this section, we are going to compare all the implementations together. In particular, we are going to see by how much, the second approach in favor to approach one:

Figure 6.6 is plotted using the same set of data obtained from Section 6.2 to 6.5, but combine them into one graph to get a better comparison schema.

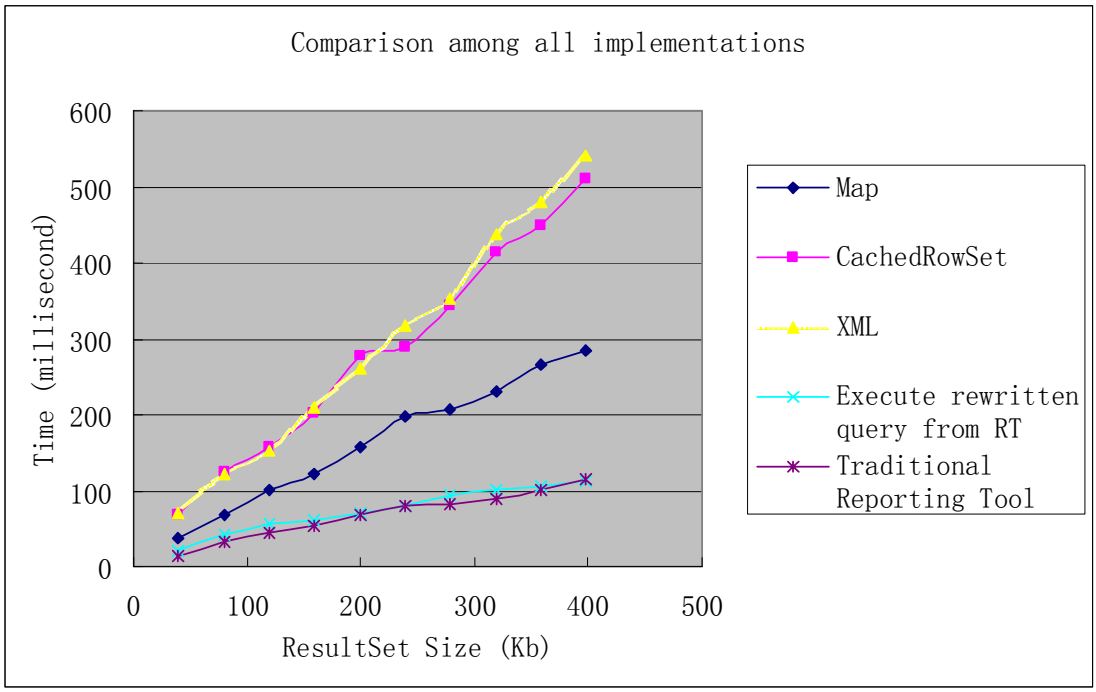


Figure 6.6 Plot: Comparison among all implementations

According to Figure 6.6, we can roughly separate all the implementations into three distinct categories:

1. **High**: This category includes *CachedRowSet* and XML implementations from the first approach. They seem to have a very high overhead due to *ResultSet* Serialization.

2. **Medium**: *Map* implementation from the first approach. Compared to *CachedRowSet* and XML, it seems that it takes relatively less time in serializing the *ResultSet*. However, it still produces some amount of overhead due to the fact that when compared to the traditional direct database access, it still shows a fairly big overhead in time.

3. **Low**: Execute the rewritten query from the reporting tool from the second approach. Clearly, it hardly produces any overhead in time. Hence, this approach is the best among all implementations. If this system is to be used to practice, and performance is of concern, this approach should certainly be adopted.

7 Conclusion

Let us summary the report as the following:

We start this project by looking at the behaviors of the traditional reporting tools. Indeed, we found that most of the reporting connects to a data source with no middle tier in between. Therefore, the reporting tool can only cope with any data model changes by changing its configurations.

We found a possible solution to solve this problem by adding a middle tier in between the reporting tool and the data source. Instead of connecting to the data source directly, the reporting tool will now connect to the middle tier which acts as a **Virtual Database** to the reporting tool. Each request issued by the reporting tool is intercepted by the middle tier before it gets processed.

We have also defined an XML mapping mechanism to map the current data model to the data model that reporting tool views. All the requests from the reporting tool, when it comes to the middle tier, are now mapped to a new request which is compatible to the data model before any changes are made.

Thus, we will have a new request generated by the middle tier which can be executed against the most-recent data model, but produce the results that the reporting tool wants. We came cross different implementations of getting the result to the reporting tool **either** by executing the request from the middle tier and pass the result to the reporting tool through RMI **or** sending the new request back to reporting tool and let the reporting tool itself gets the *ResultSet*.

We further investigate the different implementations and found, by performance evaluation, that the approach where the rewritten query gets transported through RMI achieves our goal very efficiently, it hardly create any overhead.

Overall, the initial problem can be solved and it can also be achieved relatively efficiently.

8 Future works

As mentioned in the previous chapters, the XML mapping file in this system needs to be manually defined. Any changes made to the data model must be visible to the person who defines the XML mapping file. This of course is not desired in many situations, typically when the data model gets very complicated. Therefore, automation of this process is definitely worth investigating.

A wide suggestion can be: instead of defining the data model changes directly in the XML mapping file, we can have two XML files which define both the current data model in the database and the old data model that reporting tool views. Then we can build some mechanism which examines the two data model definition and figures out the changes made automatically. This, however, will require some restrictions on how the data model can be modified. E.g. how can we know that an old column name "customer" is now called "client" in the new data model? A possible restriction to make the mapping automatic can be, say, to specify an ID attribute to each column/table. So before and after the data model changes, "customer" and "client" will end up with the same ID.

9 Acknowledgment

I wish to extend my great thanks to my supervisors: Dr. Santokh Singh & Dr. Xinfeng Ye for all the suggestions, helps and encouragements. Without their guidance, I doubt I could finish this work.

Special thanks to Vijay Savanth, who is also working on this project, for all the suggestions either to the development of system or to this report. I wish him all the best.

Lots of thanks to all my friends, for the confidence they gave me when I run into fear and difficulties.

10 References

1. Elizabeth Montalbano, "***Eclipse Developers To Get Open-Source Reporting Tool. (Business Intelligence and Reporting Tool)"***, Computer Reseller News Sept 13, 2004 p37.

2. Harvey, Troy Alan*, M.Eng.*, "***An XML-based architecture for translating SAS datasets into Web reports***", University of Louisville, 2005.

3. Prakash, N.; Garg, K.; Chopra, Y.C, "***SQL translator using artificial neural networks",*** Intelligent Information Systems, Australian and New Zealand Conference, 1996

4. Mary Stearns Sgarioto*, "**Object databases move to the middle***" InformationWeek**.** Manhasset: Nov 29, 1999. p. 115.

5. *Liang, Wei Hua*, M.Comp.Sc., *"**WISH XML Query Composer***", Concordia University (Canada), 2003.

6. Sihem Amer-Yahia, Fang Du, Juliana Freire, "***A comprehensive solution to the XML-to-relational mapping problem"***, Sixth ACM CIKM International Workshop on Web Information and Data Management, November 2004.

7. Iraklis Varlamis, Michalis Vazirgiannis, "***Bridging XML-schema and relational databases: a system for generating and manipulating relational databases using valid XML documents"***, November 2001.

8. Mary Fernandez, Atsuyuki Morishima, Dan Suciu, "***Efficient evaluation of XML middle-ware queries***", SIGMOD Conference, May 2001.

9. Cong Yu, Lucian Popa, "***Constraint-based XML query rewriting for data integration***", International Conference on Management of Data**,** June 2004.

10. JasperReport Homepage, "***JasperReports Documentation"***, Available from: http://jasperforge.org/sf/wiki/do/viewPage/projects.jasperreports/wiki/HomePage. Accessed Oct 2006.

11. Nitin Nanda and Sunil Kumar**, "*Create your own type 3 JDBC driver"***, Available from: http://www.javaworld.com/javaworld/jw-05-2002/jw-0517-jdbcdriver.html. Accessed Oct 2006.

12. MySQL, "**MySQL®Connector/J**", Available from: http://www.mysql.com/products/connector/j/ Accessed Aug 2006.

13. BIRT, "**BIRT Tutorial**", Available from: http://www.eclipse.org/birt/phoenix/tutorial/#tutorial Accessed Jul 2006.