Automatic Generation of JavaQuery API Capable of Simulating Active Database

Toshiaki Majima

Language Design Laboratory

Nara Institute of Science and Technology

TABLE OF CONTENTS

1. ABSTRACT.......... 5

2. INTRODUCTION.......... 6

3. TERMINOLOGY LIST.......... 9

3-1. STRUCTURED QUERY LANGUAGE (SQL).......... 9
3-2. SQL STATEMENTS.......... 9
3-3. SQL DIALECT PROBLEM.......... 10
3-4. JAVA 2 PLATFORM.......... 10
3-5. JAVA DATABASE CONNECTIVITY (JDBC).......... 10
3-6. JAVA INTERNATIONALIZATION.......... 11

4. MOTIVATING ISSUES.......... 12

4-1. SQL LEARNING COST.......... 12
4-2. SOURCE CODE INCONSISTENCY.......... 13
4-3. LOW REUSABILITY.......... 13
4-4. PROPOSED SOLUTION.......... 14

5. SOLUTION.......... 15

5-1 OVERVIEW.......... 15
5-1.1 Java Query System Development Kit (Java Query SDK).......... 15
5-1.2 System Generation Script (SG Script).......... 15
5-1.3 Just-in-time Implementer.......... 16
5-1.4 Java Query System Environment (Java Query API).......... 17
5-2 TUTORIAL.......... 18
Step 1. Write SG Script.......... 18
Step 2. Auto-generate Java Query API with Just-in-time Implementer.......... 19
Step 3. Developing DB Applications in Java Query API.......... 20
5-3 FEATURES.......... 23
5-3.1. Synchronizing Accesses with Table-level Lock Granularity.......... 23
5-3.2. Dynamic SQL Dialect Composer.......... 24
5-3.3. Pooling and Garbage Collection for Connection.......... 27
5-3.4. Simple API for Data Loading.......... 29
5-3.5. Active Database Features.......... 30

6 MERITS AND ORIGINALITIES 32

6-1 A BASIC KNOWLEDGE OF JAVA IS ALL THAT IS NEEDED 32

6-2 STAND-ALONE ON J2SE PLATFORM 32

6-3 AUTOMATIC CODE GENERATION 32

6-4 SIMPLE SG SCRIPT RESULTS IN A DEPLOYABLE SYSTEM 33

6-5 CODE SIMPLICITY 33

7. QUERY PERFORMANCE TEST RESULT 39

7-1. PERFORMANCE COMPARISON (JAVA QUERY API VS. JDBC) 39

7-2. THOUGHTS ON JAVA QUERY API PERFORMANCE OVERHEAD 40

8. REFERENCES 40

List of Figures

Figure 1 Development Flow with Java Query SDK ______ 7
Figure 2 JDBC Code Snippets ______ 8
Figure 3 Activity diagram of the JDBC programmers ______ 12
Figure 4 System Generation Script (sample) ______ 15
Figure 5 Activity Diagram of Java Query SDK Developer ______ 17
Figure 6 Synchronization with Lock on Table Accessor Class ______ 24
Figure 7 Java Query API Dynamically Composing MySQL SQL Statements ______ 26
Figure 8 Java Query API Dynamically Composing Oracle Statements ______ 26
Figure 9 Dynamic Connection Router ______ 28
Figure 10 Data Loading Capability of Java Query API ______ 29
Figure 11 Application Level Trigger ______ 31
Figure 12 Query Performance Result ______ 39

1. Abstract

Various programming languages provide application programmer's interfaces (API) for accessing databases. Traditional APIs, however, require the developers to explicitly include Structured Query Language (SQL) statements as string literals within the source code of the host programming language. For example, Java Database Connectivity, a database access methodology of Java, requires Java programmers to embed SQL statements as string literals in the Java code. While accomplishing the integration of database access capability, the mixture of two different languages causes several difficulties, such as SQL learning cost and inconsistency. To solve these issues, we propose a programming paradigm that will let Java database application programmers concentrate solely on Java throughout the development phases, without having to write a single SQL command.

2. Introduction

The database research and development field can be subdivided into the following three major fields (Urman, 2000):

- The User Interface
- The Application Logic
- The Database

Our concentration is on database application development, focusing on designing and establishing a new programming paradigm.

The purpose of this research project is to extend the relational database technology using Java, currently one of the most popular object-oriented programming languages (see section titled "Terminology" for more details of Java). Specifically, in this paper I introduce development methodology that accepts a Java-oriented simple script as input, and outputs a development environment where the client can manipulate database tables with auto-generated pure Java application programmer's interface (API). The development tool established for this research project is what we call the Java Query System Development Kit (Java Query SDK). The key development components of Java Query SDK include Java Query System Generation Script (SG Script), an automatic API generator called Just-in-time Implementer that accepts the SG Script, and the resulting output from the Just-in-time Implementer which we call the Java Query System Environment (Java Query API). Figure 1 below shows an schematic overview of how Java Query SDK development proceeds.

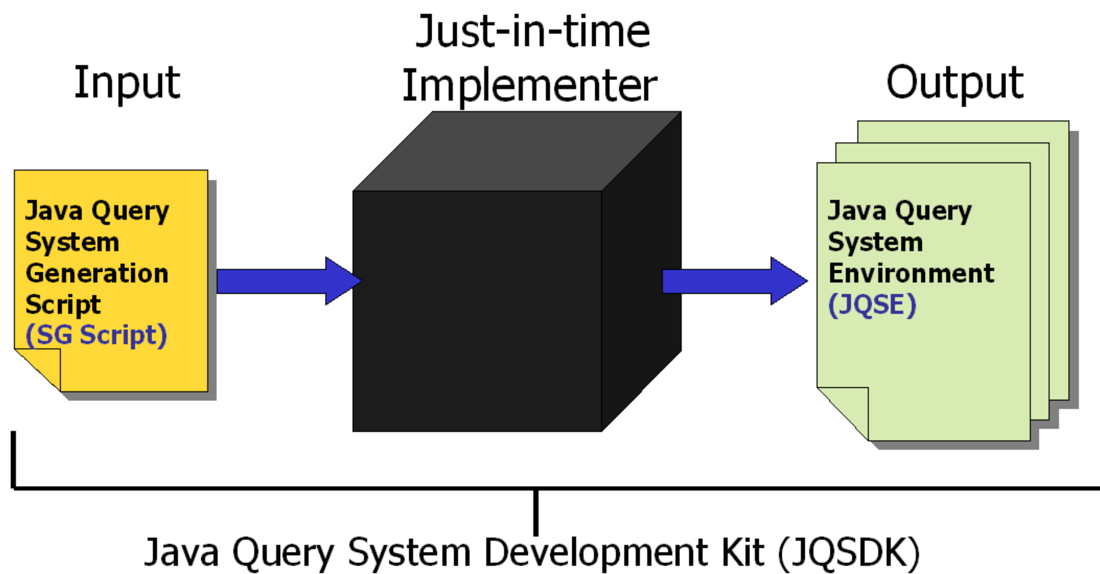

Figure 1 Development Flow with Java Query SDK

Java Query API is a development environment that allows developers to build database access applications which are based purely on Java, without having any other language embedded. By "pure Java," we mean that the Java Query API API, unlike the JDBC API, does not require the client to explicitly insert SQL statements to access relational databases.

The Java Query API is a stand-alone system based on J2SE technology, being composed of Java Bean, JDBC, and relational database table schema.

A relational database is composed of data stored in tables as records, which are traditionally accessed and manipulated using Structure Query Language (SQL). SQL commands are often executed interactively through console applications such as Oracle SQL*Plus.

To process SQL statements from within a programming logic, many current programming languages provide an application programmer's interface to databases. Java Database Connectivity (JDBC), which enables Java programs to access and manipulate data stored in databases, is an example of such an interface. The traditional application programmer's interfaces to

databases, however, require developers to have sufficient knowledge of SQL and to express statements explicitly within the source code. JDBC is no exception: the source code of JDBC consists of a mixture of Java language and SQL statements (see Figure 2, below, for a JDBC code snippet).

As shown in Figure 2, Java programmers embed a string literal representing an SQL statement to be executed. That is, a JDBC programmer must write SQL statements in Java. A crucial point for this discussion is that Java and SQL, two clearly different languages, are commingled in one source code. Java is a third generation, object-oriented language, while SQL is a fourth generation, record-oriented language.

Java = *3GL* Language
Object-oriented Language

```
Connection conn =
  Driver.getConnection(...dburl, etc...);
String updateString = "UPDATE
COFFEES " + "SET TOTAL = TOTAL
+ 75 " + "WHERE COF_NAME LIKE
'Colombian'";
Statement stmt =
                    conn.createStatement();
stmt.executeUpdate(updateString);
```

SQL = *4GL* Language
Record-oriented Language

Figure 2 JDBC Code Snippets

3. Terminology List

3-1. Structured Query Language (SQL)

SQL is an ANSI (American National Standards Institute) standard computer language for accessing and manipulating database systems. SQL statements are used to retrieve and update data in a database…Unfortunately, there are many different versions of the SQL language, but to be in compliance with the ANSI standard, they must support the same major keywords in a similar manner (such as SELECT, UPDATE, DELETE, INSERT, WHERE, and others) [1].

3-2. SQL Statements

The following is a list of SQL statements that are most relevant to the discussion of this thesis. The description of the functionality of each statement indicates a generic usage of the statement; you may find more usages (please consult the manual of the database that you will be using for the complete reference of usages).

- SELECT retrieves a record or multiple records from a single table or combination (join) of tables based on the selective condition, if any, specified in the statement's WHERE clause.
- DELETE removes a record or multiple records from a table.
- INSERT inserts a record into a table.
- UPDATE updates a record or multiple records already in a table.
- ROLLBACK is a command to "undo" a previous transaction.
- COMMIT is a command to "make permanent" a previous transaction. That is, ROLLBACK will not be able to undo a committed transaction.
- JOIN is a special usage of SELECT statement to retrieve records from a combination of multiple tables.

3-3. SQL Dialect Problem

But note that despite a long history of standardization, setting a common language for RDBMS has not necessarily been outstandingly successful…the bottom line is though SQL has the very same intent, functionality, and general purpose across database vendors - users have had to get used to working in dialects. SQL interoperability is problematical [2].

3-4. Java 2 Platform

The "Write Once, Run Anywhere" Java 2 Platform is a safe, flexible, and complete cross-platform solution for developing robust Java applications for the Internet and corporate intranets. The open and extensible Java Platform APIs are a set of essential interfaces that enable developers to build their Java applications and applets. The Java 2 Platform provides uniform, industry-standard, seamless connectivity and interoperability with enterprise information assets [3].

3-5. Java Database Connectivity (JDBC)

JDBC technology is an API that lets you access virtually any tabular data source from the Java programming language. It provides cross-DBMS connectivity to a wide range of SQL databases [4].

JDBC programmers write SQL statements within Java programs in order to communicate with a database.

3-6. Java Internationalization

Internationalization is the process of designing an application so that it can be adapted to various languages and regions without engineering changes [5].

For example, suppose there is an internationalized Java application for Japanese and English that outputs a greeting message. When the application is used in a Japanese environment, the internationalization technology will incorporates a property file containing a key-value pair (e.g. evening="こんばんは") to let the program output in a proper language. When used in an English environment, the property file will be switched to that for English, containing a value of "Good Evening" for the same "evening" key to make a proper output.

4. Motivating Issues

The following three issues, in particular, motivated us into working on this research project.

4-1. SQL Learning Cost

The first issue to introduce is the cost for JDBC programmers of learning SQL so that they can explicitly write SQL statements (see the Activity Diagram of JDBC programmers in Figure 3, below).

It is not reasonable to assume that all Java programmers have expertise in manipulating SQL-based databases. Therefore, those Java programmers who are not used to writing SQL must spend time and labor on acquiring sufficient proficiency in SQL. In addition, as mentioned earlier, SQL syntax is quite different from that of Java and other object-oriented programming languages, so the cost of learning SQL may be considerable. In short, the mixture of Java and SQL can put an extra burden on JDBC application developers.

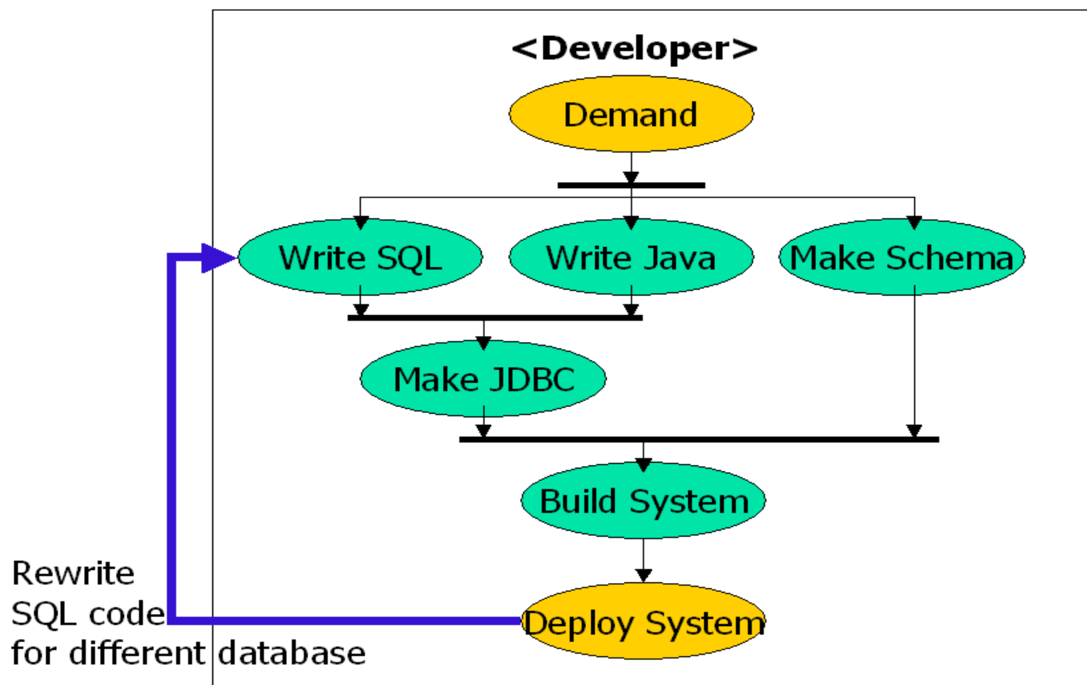

Figure 3 Activity diagram of the JDBC programmers

4-2. Source Code Inconsistency

The second issue is the inconsistency of the application source code, also caused by the mixture of Java and SQL. Two different concepts, the object-oriented concept and the record-oriented, represented respectively by Java and SQL, are mixed in the process of achieving the goal of building database access applications. Therefore, the mixture of the two languages not only imposes a syntactical inconsistency, as discussed in the previous paragraph, but also a conceptual inconsistency.

4-3. Low Reusability

The final issue regarding features of the JDBC technology arises because SQL is not completely standardized (see the section entitled “SQL Dialect Problem”). SQL syntax differs slightly between one vendor’s database and another. The implication of the SQL difference is that one JDBC application developed specifically for one database may not be reusable for another database. As acknowledged by the Zope Community: We were hoping to have one code base support multiple database vendors' products but as it turned out the differences in SQL implementations between vendors is large enough to make life extremely difficult [7].

Java has become very popular thanks to its cross-platform neutrality, “write once, run anywhere,” feature. When we incorporate SQL technology explicitly into Java language, however, Java is forced to be dependent on databases. It is true that Java integrates itself with the power of SQL by literally including SQL statements, but we should note that this methodology comes with the significant cost of losing Java code neutrality and reusability.

4-4. Proposed Solution

As a solution to the issues discussed above, we have explored the possibility of generating a methodology which will enable programmers to concentrate on a single language and hence a single concept during the development of database-related applications. As the product of our exploration, we developed software which we name the Java Query System Development Kit (Java Query SDK). The Java Query SDK development mechanisms are described in detail in the next section.

5. Solution

5-1 Overview

5-1.1 Java Query System Development Kit (Java Query SDK)

The Java Query System development process with the Java Query SDK consists of three key components: the input is Java Query System Generation Script (SG Script), the processor is the Just-in-time Implementer, and the output is an immediately deployable Java Query System Environment (Java Query API). These three components are individually explained in detail in the following sections.

5-1.2 System Generation Script (SG Script)

When a table is required, a user will specify the table requirements in SG Script, whose syntax is based purely on Java. This step requires only a basic knowledge of Java (by "basic," we mean the scope of Java 2 Standard Edition) and of the structure of database tables.

```
Java Query System Generation Script
class: Employee
//<Type> <Name>:<Permission>:[*TRIGGER_SPECIFICATION]
String id : r : (id.length()==9)
String name : r
double salary : rw : if(salary>500000)Mailer.warning()
char rank='c' : rw
int age : rw : if(age>17&&age<55)ErrorFile.log()
# PRIMARY KEY (id)
```

Figure 4 System Generation Script (sample)

As can be seen in Figure 4, Java Query SDK allows users to specify the system requirements in SG Script format (see the section entitled "Specification" for detailed description of the SG Script syntax). The Java Query SDK user starts development by writing SG Script, whose only requirements are the class name and property list, with other available syntax options. The user is presented with syntax options for defining default values, setting read/write/both permission levels, and Event-Condition-Action rules for triggers (see the section entitled "Features" for a description of Active Databases).

5-1.3 Just-in-time Implementer

The Just-in-time Implementer analyses an SG Script and generates an environment where the user can express commands to query and manipulate the data via statements based purely on Java. This software implements the cross-database, Java query system 'just in time' to meet the needs of the user, as described in the SG Script.

The activity diagram for Java Query SDK developers in Figure 5, shown below, indicates the responsibility of the Just-in-time Implementer. As can be seen, the Just-in-time Implementer takes much of the programming burden off the developer.

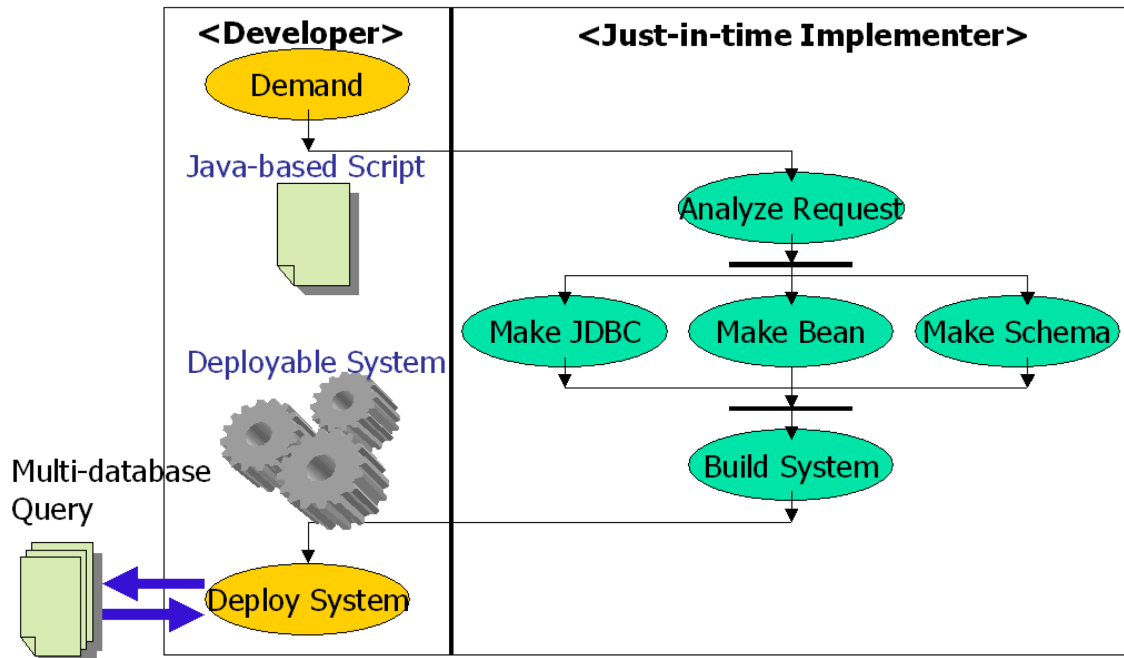

Figure 5 Activity Diagram of Java Query SDK Developer

5-1.4 Java Query System Environment (Java Query API)

Java Query API, the output of the Just-in-time Implementer, is composed of three components: Java Bean, JDBC code, and relational stable schema definition, all of which are automatically generated by the Just-in-time Implementer. The Java Bean serves as the interface for database application developers; that is, programmers will manipulate the data in the target database via the methods provided by the Java Bean. The complexity of SQL handling is processed by JDBC code generated behind the scenes. In short, the Java Query System encapsulates SQL-based access to the relational database and provides developers with a purely Java-based, object-oriented programming interface. For a more detailed description of Java Query API usages, please refer to the section entitled "Tutorial" below.

5-2 Tutorial

Suppose there is a need to develop a database application to manage corporate data. Suppose, for a simple tutorial example, that two database tables are required to store Employee data and Departmental data, respectively.

Step 1. Write SG Script

Suppose that the Employee data are specified in the following SG Script.

```
class: Employee
int id rw
String firstName rw
String lastName rw
String rank rw
int age rw if(age <=18 || age >= 60) throw new
        IllegalArgumentException("The age entry is out of
        bounds.");
double salary rw if(salary > 1000000) throw new
        IllegalArgumentException("The salary entry is over the
        permitted upper limit.");
#PRIMARY KEY (id)
```

The above SG Script sample indicates that the Employee table has several fields (id, first name, last name, rank, age, and salary), with corresponding Java types specified for all those fields. The letters "rw" indicate that the fields are accessible (r) and modifiable (w); that is, the to-be-generated Java Bean code will be equipped with both setters and getters for these fields.

Note also that triggers are specified for two fields, age and salary. Here, assume that the company has a business policy that employment age ranges from 18 to 60, and that the salary must not exceed $1,000,000. Although

any legal Java statement or function may be specified as a trigger, however long it may be, the specification of this tutorial simply throws an exception.

The last line of the SG Script indicates that we want to make the id field as the Primary Key for the to-be-generated table schema.

The following is an SG Script for Departmental data.

Class: Department
int managerID **rw**
int numWorkers **rw**
String deptName **rw**
int deptID **rw**
#PRIMARY KEY(deptID**), FOREIGN KEY (**managerID**) REFERENCES Employee(**id**)**

We keep the schema simple for the tutorial. An important point to note about the SG Script for Departmental data is that the constraint specified in the last line includes that for a Foreign Key.

Step 2. Auto-generate Java Query API with Justin-time Implementer

The step of generating Java Query API is automated. At a command prompt of a console window, type as follows:

```
%>   javaq Employee
```

Messages and the Java source files and API documentation of Employee and EmployeeQuery class are auto-generated by the Just-in-time Implementer.

Next, repeat for Departmental data.

%> javaq Department

The above will generate additional Java source files: Department.java and DepartmentQuery.java.
Java Query API is now generated and ready for use. We will now move on to the next step.

Step 3. Developing DB Applications inJava Query API

Assume, for this tutorial, that the target database system is a MySQL database. First of all, connection to the target database must be established. To make the connection, write a code as follows (using the correct username and password as parameters to the constructor):

```
31        //create management object
32        Employee empTableManager = new Employee("mysql", "username", "password");
33        Department deptTableManager = new Department("mysql", "username",
34            "password");
```

In the case of developing Java Query API applications by creating a new table, start from Step 3-1. If working with an existing database table, skip Step 3-1 and start from Step 3-2. Users may secure a password however they wish, instead of explicitly writing it in the code. The above code is only an example, for tutorial purposes, and security issues should be evaluated by individual Java programmers.

Step 3-1. Creating database tables (if not already in existene)

Write the following code to create table in the target database:

```
36        //create table in mysql database
37        empTableManager.createTable();
38        deptTableManager.createTable();
```

It is recommended, but not required, that Java Query API developers separate instances of Java Bean class (i.e. Employee class in this tutorial): one for the table management object that is used for table-level transaction controls, such as table creation, and others for manipulating table record objects representing records stored in the database. This coding style may enhance the readability of the Java Query API program.

Step 3-2. Processing SQL-equivalent actions

- Inserting

The following code snippet inserts the records of newly hired employees, here Tom Smith and John Dow, into the database table. The attributes of the two employees are to be specified in the arguments to the constructor.

```
9       //create "record" objects
10      Employee tom_smith = new Employee(1000, "Tom", "Smith", "A", 45, 70000);
11      Employee john_dow = new Employee(1001, "John", "Dow", "A", 50, 75000);
12
13      //make the records objects persistent
14      tom_smith.store();
15      john_dow.store();
```

- Deleting

The following code snippet shows how to delete an employee's record (here, the employee whose id is equal to 1001). The condition passed to the fetch() method is used to specify the records to be extracted. The records are extracted in the form of a bundle stored in ArrayList class. Thus, invoke get() method of ArrayList to obtain the record object to be deleted, then cast its type to Employee type, and finally call die() method, which will delete the record corresponding to the invoking object. The parameter to the get() method here is 0, which is the index of the list, containing a single employee record object whose id is 1001, returned by the fetch() method.

```
61      //delete a record with id = 1001
62      ((Employee)(Employee.fetch("id = 1001").get(0))).die();
63
```

- Update

The following code updates the salary to $78,000 for the employee whose id is equal to 1000. All the setter methods of Java Bean are used to update the attribute values of the record stored in the database which the invoking object represents.

```
42
43      ((Employee)Employee.fetch("id = 1000").get(0)).setSalary(78000);
44
```

5-3 Features

5-3.1.Synchronizing Accesses with Tablelevel Lock Granularity

In the Java Query API, an auto-generated "executed-behind-the-scenes" JDBC class is the only access gateway to the corresponding table (e.g. for Employee data, the auto-generated JDBC class which gives access to the Employee table is named EmployeeQuery). The Java Query System makes use of this feature and provides the client with a exclusive access capability by locking the JDBC class, which we call the Table Accessor in this context. Note that each Java instance, including that of java.lang.Class class loaded by Java Virtual Machine (JVM) for each Java class, comes with a single lock. Since there is only a single instance of java.lang.Class for each class loaded by JVM for the EmployeeQuery class, the client can specify the guarded region by locking on that java.lang.Class instance for the EmployeeQuery class.

Table-level manipulations, including SELECT, are done via the JDBC class. Methods for table-level operations are described in detail later. Each record of the database table, on the other hand, is accessible and modifiable by way of instances of the corresponding Java class.

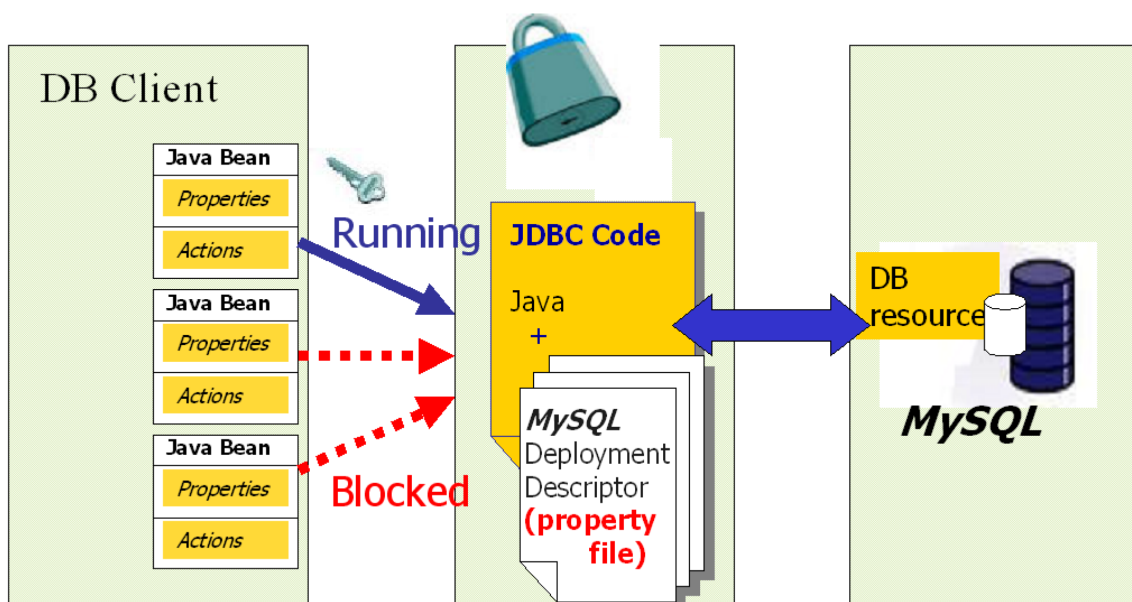

Figure 6 Synchronization with Lock on Table Accessor Class

5-3.2. Dynamic SQL Dialect Composer

Another important feature of the Java Query API, in addition to its purely Java-based syntax, is that it enables Java Query System clients to switch databases dynamically without having to modify the SQL statements embedded in the auto-generated code. The Java Query API makes use of Java Internationalization technology (see the section entitled "Terminology List" for Java Internationalization) to resolve the problem of SQL dialects.

The Java Query System incorporates multiple property files which define the core SQL syntax and data types that are employed by the group of target databases. The system switches the property files and dynamically generates a proper SQL statement according to the database that the program is currently working on.

For example, suppose a Java Query API Application first accesses the MySQL database (see line 4 of the sample code snippet shown below), and then the program switches its connection and redirects its access to Oracle database (see line 24 of the sample code snippet shown below). As discussed earlier, without any modification, the SQL difference (such as a difference in supported data types) could cause the program to halt. Java Query API automatically and dynamically modifies the to-be-executed SQL statements, without the Java Query API Application programmers' knowing. The mechanism is illustrated below.

First, the Java Query API allows the Java Query API Application to talk with MySQL, by dynamically composing proper SQL statements with MySQL data types (see Figure 7).

```
1
2       //create "management" object
3       Employee empTableManager = new Employee("mysql");
4       empTableManager.connect("username", "password");
5
6       //create table in mysql database
7       empTableManager.createTable();
8
9       //create "record" objects
10      Employee tom_smith = new Employee(1000, "Tom", "Smith", "A", 45, 70000);
11      Employee john_dow = new Employee(1001, "John", "Dow", "A", 50, 75000);
12
13      //make the records objects persistent
14      tom_smith.store();
15      john_dow.store();
16
17      //commit
18      empTableManager.commit();
19
20      //show all the records from the table
21      //empTableManager.showAll();
22
23      //Redirect the connection to Oracle
24      empTableManager.connect("username", "password", "oracle");
25
26      //create table in Oracle database
27      empTableManager.createTable();
28
29      //reset the isStored flag (not stored in oracle yet)
30      tom_smith.isStored(false);
31      //store into the currently connected database (i.e. Oracle)
32      tom.store();
33
34      //commit (now on Oracle)
35      empTableManager.commit();
36
37
```

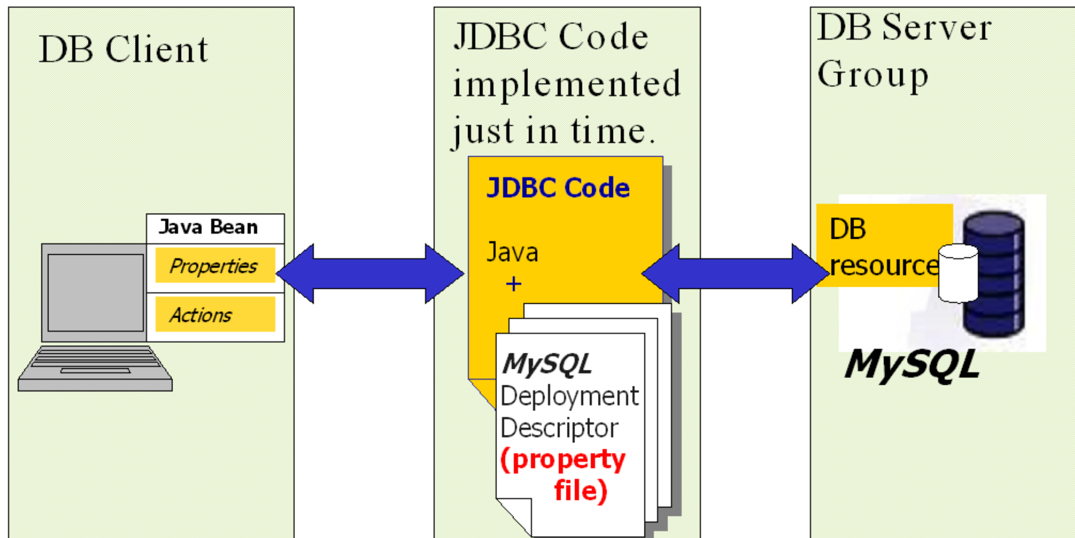

Figure 7 Java Query API Dynamically Composing MySQL SQL Statements

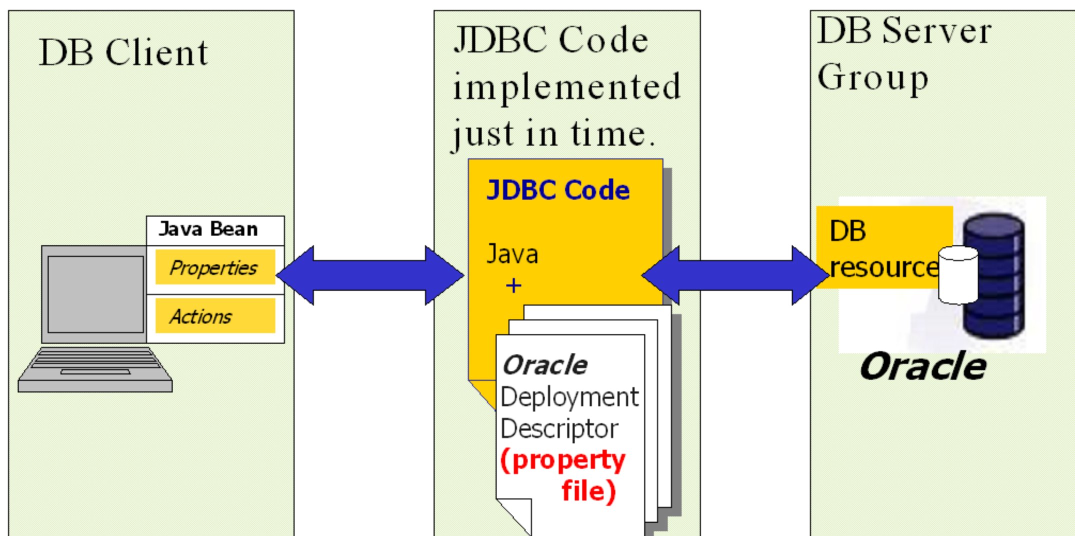

Figure 8 Java Query API Dynamically Composing Oracle Statements

Next, when the program redirects its connection to the Oracle database, Java Query API applies Java Internationalization technology to solve the SQL dialect issue. Note that in Figure 8, as shown above, the property file has been switched to that for Oracle.

This Java Query API mechanism utilizes Java Internationalization technology and ensures that the client Java Query API Application will be free from any unexpected application failure due to syntax errors of SQL statements when the application is used against multiple, heterogeneous databases.

5-3.3. Pooling of Connection

Establishing a connection to databases is a costly operation, and we should try to keep the connection state alive, and hence reusable, throughout the period in which the client application is still considered in the session. The term 'session' used here is very similar to that used in the context of web applications, in which the server maintains the state with individual users during a specified period.

Our definition of the term session is as follows: the Java Query System Session starts for a client when the client connects for accessing one table via the corresponding instance of the Java Bean. The connection to the database for this client is kept alive until the Session ends for that client, i.e. when the client disconnects from all the instances for which the client built connections.

The benefit of keeping the database connection for the duration of the session is that the client applications avoid unnecessary reconnections that can be a considerable performance overhead. The Java Query System manages the session automatically, leaving the developers free of session management responsibilities. As soon as the user disconnects from the last connected instance, the Java Query System automatically terminates the connection and ends the session for the user.

As can be seen in Figure 9, a connection router routes client requests to a proper database.

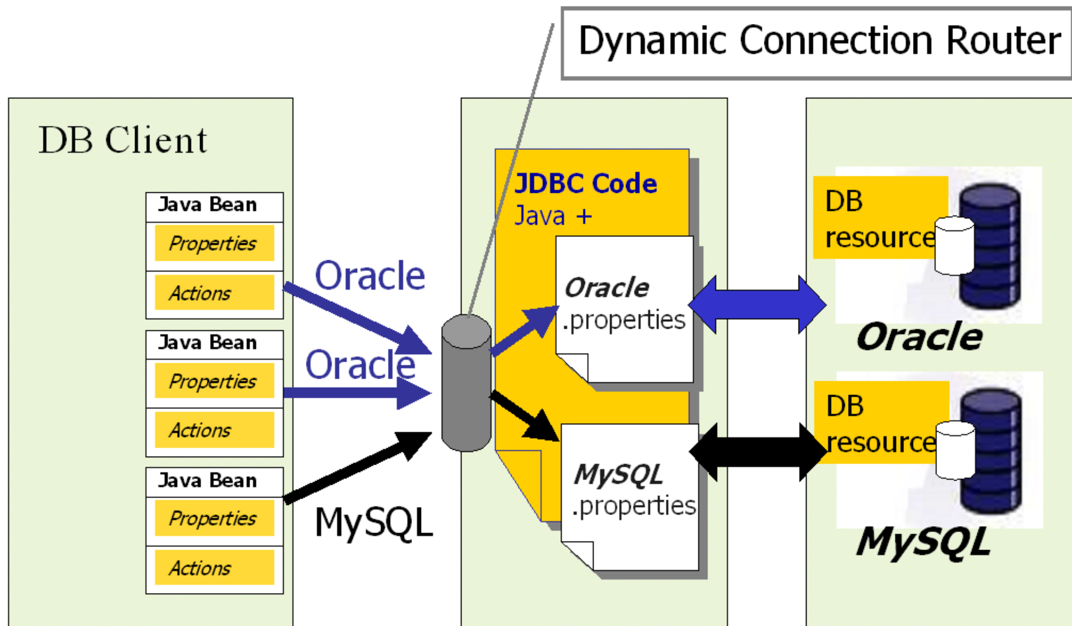

Figure 9　Dynamic Connection Router

5-3.4. Simple API for Data Loading

Commercial products, such as Oracle SQL Loader, are available for transferring all the data in one table into another table. The Java Query System provides the client with this transport capability through its API. While some commercial products specifically developed for this purpose require users to learn to write a detailed script for transportation, the Java Query System requires only a call to a single method, "transfer," defined in its auto-generated API. The functionality of the transfer method is shown in Figure 10.

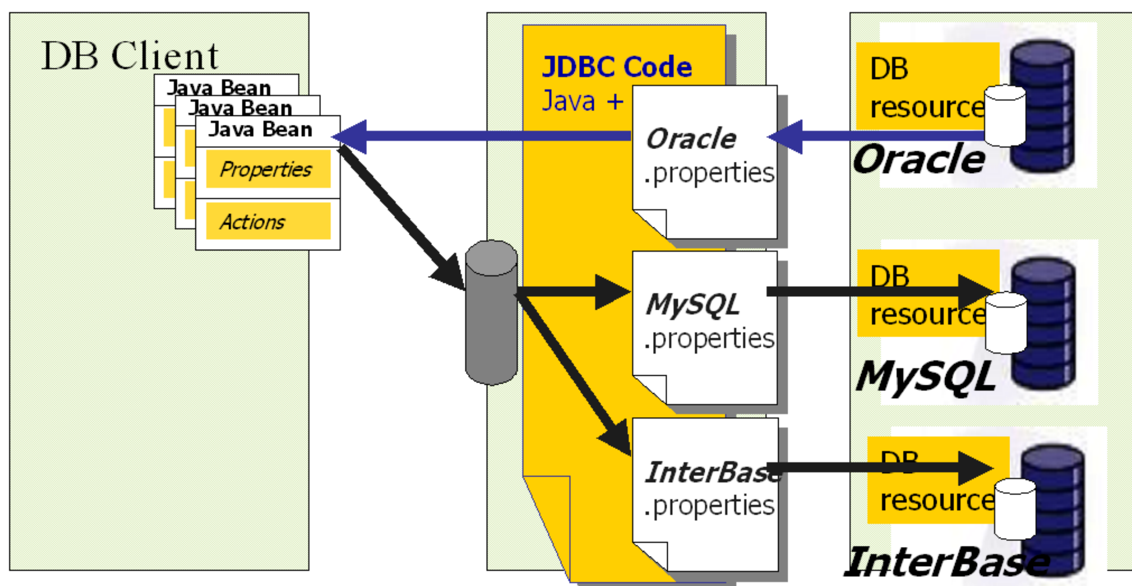

Figure 10.

Figure 10 Data Loading Capability of Java Query API

5-3.5. Active Database Features

In the syntax for system generation scripts, Java Query SDK users can also specify triggers for the to-be-generated system. The Java Query System imitates and simulates the trigger functionality offered by certain relational databases, specifically those called Active Databases. The trigger functionality can be any Java statement or expression, however long and complex it may be. The user can, of course, integrate the Java technologies of Graphical User Interface, Mailing, or any other complex processing. Note also that the triggers are furnished within the application logic, being embedded in the implementation of the Java Query API.

In traditional Active Databases, the trigger can be fired only when a triggering event is detected by the database itself. In contrast, the Java Query System detects triggering events within the to-be-executed SQL statements before the statements reach the database (see Figure 11). This mechanism of application logic triggering is provided as part of the efforts to avoid bottlenecks on the database server.

In addition, any complex Java Boolean expression can be used as the trigger condition and any Java statement or expression can be specified as the trigger action. Thus, the triggers can be more flexible and powerful than those offered by Active Databases.

It is notable that some relational databases are not "active," in the sense that they do not support trigger functionality. The application logic triggers offered by the Java Query System, therefore, can encapsulate and "make active" even non-Active Databases (i.e. equip them with trigger capability), providing an illusion that a non-active relational database is active.

Furthermore, even with originally Active Databases, it should not be assumed that the database server administrator will integrate the trigger specification for all the requests of each client. Individual clients can have many business policies that they wish to be enforced by a database using trigger specifications. The database administrator must, however, keep the database tuned and may choose not to implement any requested trigger that

might cause a performance decline. Here again, the application logic trigger, enabling each client to specify triggers at the cost of the client machine's performance, should be a solution.

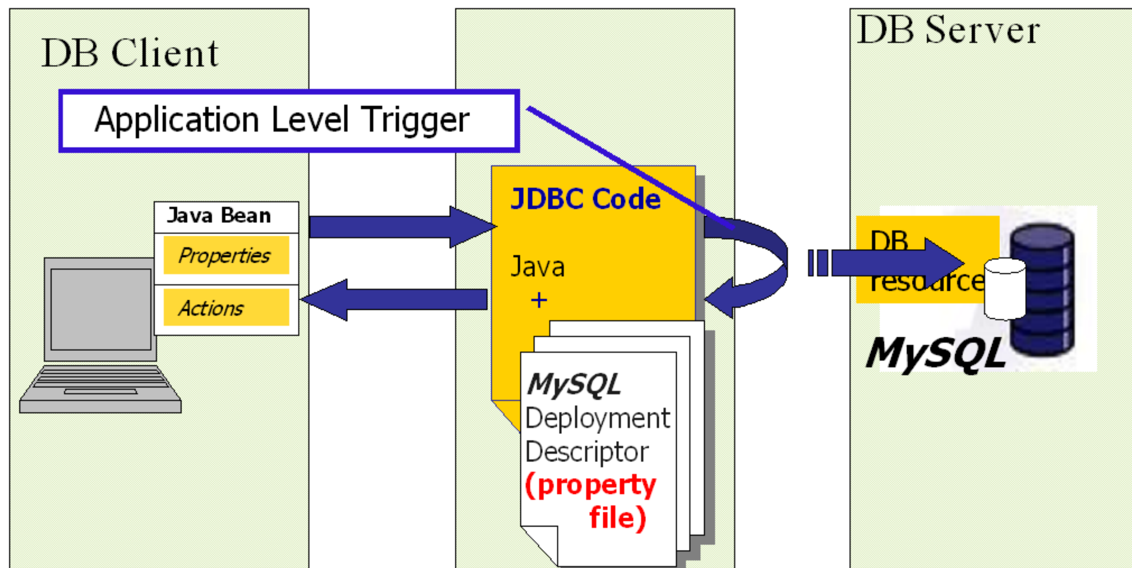

Figure 11 Application Level Trigger

6 Merits and Originalities

The following is a list of merits and originalities of the proposed solution.

6-1 A Basic Knowledge of Java Is All That Is Needed

First, and most importantly, the merit of the proposed solution is that a Java Query API client has only to know basic Java for building database access applications. Therefore, unlike the JDBC technology, which requires the developers to explicitly write SQL statements mixed with Java language, the Java Query API application code is consistent both in its syntax and in its concept.

6-2 Stand-alone on J2SE Platform

The originality of Java Query API is that it is a stand-alone J2SE software component during the deployment phase, unlike Enterprise Java Beans (EJB), which can run only on a Java 2 Enterprise Edition-compliant EJB Container. Therefore, although the goals of using Java Query API and EJB may be similar, the Java Query API holds its originality in the way it is to be deployed.

6-3 Automatic Code Generation

Furthermore, both as a merit and as originality, the Java Query SDK auto-generates the source codes for the three components of Java Query API, namely, the codes of Java Bean, JDBC, and Table Schema. The resulting three components of Java Query API are already linked together, such that

the Java Bean methods are designed to invoke the corresponding JDBC methods, which in turn execute the corresponding SQL statements against the target database.

6-4 Simple SG Script Results in a Deployable System

Users of Java Query SDK, as explained earlier, will start by writing an SG Script. The Just-in-time Implementer, a core part of Java Query SDK, will then auto-generate all the necessary components for the cross-database, Java-based query system. Indeed, only a handful of lines of code in the SG Script are sufficient for Java Query SDK users to build an entire API for Java Query.

6-5 Code Simplicity

Although performance may be slightly degraded compared to the equivalent JDBC programs (see the following comparisons between the code snippets of JDBC and those of Java Query API), the Java Query API applications can manipulate databases using codes which are simpler than those of JDBC applications. .

Suppose there are two tasks, one for retrieving a record with id equal to 1001 and another for retrieving records with id greater than 1242.

JDBC SELECT

With JDBC technology, codes like the snippet shown below must be written. The JDBC code, as discussed, contains both Java and SQL.

```
29    //set up connection to the database
30    conn = DriverManager.getConnection("jdbc:mysql:///mydb");
31    conn.setAutoCommit(false); //auto-commit false by default
32
33    //create an instance of java.sql.Statement class for processing SQL
34    Statement stmt = conn.createStatement();
35
36    //prepare String representing the first query of retrieving
37    //Employee records with id = 1002
38    String sqlString = "SELECT * FROM Employee WHERE id = 1001";
39
40    //execute the SQL statement
41    ResultSet rset = stmt.executeQuery(sqlString);
42
43    Employee obj = null;
44
45    if(rset.next()) {
46      obj = new Employee(); //constructing a default object of Employee
47      obj.id = (int) rset.getDouble("id"); ;
48      obj.firstName = (String) rset.getString("firstName"); ;
49      obj.lastName = (String) rset.getString("lastName"); ;
50      obj.rank = (String) rset.getString("rank"); ;
51      obj.age = (int) rset.getDouble("age"); ;
52      obj.salary = (double) rset.getDouble("salary"); ;
53    }
54
55    //prepare String representing the second query of retrieving
56    //Employee records with id > 1242
57    sqlString = "SELECT * FROM Employee WHERE id > 1242";
58
59    //execute the SQL statement
60    rset = stmt.executeQuery(sqlString);
61
62    ArrayList arrayList = new ArrayList();
63    while (rset.next()) {
64      Employee obj = new Employee(); //constructing a default object of Employee
65      obj.id = (int) rset.getDouble("id"); ;
66      obj.firstName = (String) rset.getString("firstName"); ;
67      obj.lastName = (String) rset.getString("lastName"); ;
68      obj.rank = (String) rset.getString("rank"); ;
69      obj.age = (int) rset.getDouble("age"); ;
70      obj.salary = (double) rset.getDouble("salary"); ;
71
72      arrayList.add(obj);
73    }
```

Compare the above JDBC code with the equivalent Java Query API code, listed below.

Java Query API SELECT

All that needs to be written is a single line of code for each task, i.e. line 54 for the first task and line 58 for the second, as shown in the following code snippet.

```
53      //select one record
54      Employee john_smith = (Employee)(Employee.fetch("id = 1001").get(0));
55
56
57      //select multiple records
58      ArrayList empRecords = Employee.fetch("id > 1242");
59
```

Now suppose that there is another task of inserting records, say, of two newly hired company employees.

JDBC INSERT

A proper SQL insert statement must be embedded within a Java code, as shown in the following code fragment.

```
65        //create connection object
66        conn =
67            DriverManager.getConnection("jdbc:oracle:oci8:@" + database,
68                                        user, password);
69        //prepare String literals representing SQL statements
70        String sqlString_1 =
71            "INSERT INTO Employee (id,firstName,lastName,rank,age,salary) " +
72            "VALUES (1000, \"Tom\", \"Smith\", \"A\", 45, 70000)";
73        String sqlString_2 =
74            "INSERT INTO Employee (id,firstName,lastName,rank,age,salary) " +
75            "VALUES (1001, \"John\", \"Dow\", \"A\", 50, 75000))";
76
77        //prepare java.sql.Statement object for processing the above SQL strings
78        Statement stmt = conn.createStatement();
79
80        //execute the SQL statements
81       stmt.executeUpdate(sqlString_1);
82       stmt.executeUpdate(sqlString_2);
```

Compare the JDBC code above with that of Java Query API, listed below.

Java Query API INSERT

```
 9        //create "record" objects
10        Employee tom_smith = new Employee(1000, "Tom", "Smith", "A", 45, 70000);
11        Employee john_dow = new Employee(1001, "John", "Dow", "A", 50, 75000);
12
13        //make the records objects persistent
14        tom_smith.store();
15        john_dow.store();
```

The last comparison between JDBC and Java Query API is for the task of deleting records. The codes required by the two technologies are listed below.

JDBC DELETE

```
29    //set up connection to the database
30    conn = DriverManager.getConnection("jdbc:mysql:///mydb");
31    conn.setAutoCommit(false); //auto-commit false by default
32
33    //create an instance of java.sql.Statement class for processing SQL
34    Statement stmt = conn.createStatement();
35
36    //prepare String representing the delete command
37    String sqlString = "DELETE FROM WHERE id = 1001";
38
39    //execute the SQL statement
40    stmt.executeUpdate(sqlString);
```

Java Query API DELETE

```
61    //delete a record with id = 1001
62    ((Employee)(Employee.fetch("id = 1001").get(0))).die();
63
```

Now assume that there is a need to update the salary of the employee whose id is equal to 1000 to $78,000.

JDBC UPDATE

```
29    //set up connection to the database
30    conn = DriverManager.getConnection("jdbc:mysql:///mydb");
31    conn.setAutoCommit(false); //auto-commit false by default
32
33    //create an instance of java.sql.Statement class for processing SQL
34    Statement stmt = conn.createStatement();
35
36    //prepare String representing the delete command
37    String sqlString = "UPDATE Employee SET salary = 78000 " +
38                       "WHERE id = 1000";
39
40    //execute the SQL statement
41    stmt.executeUpdate(sqlString);
```

Java Query API UPDATE

```
42
43      ((Employee)Employee.fetch("id = 1000").get(0)).setSalary(78000);
44
```

As can be seen in the above comparisons, the Java Query API application can achieve the same SQL effects using the same or a smaller number of lines.

7.Query Performance Test Result

7-1. Performance Comparison (Java Query APIvs. JDBC)

The performance of the queries against a sample Employee table was tested using JDBC and Java Query. The response time taken by each of the two technologies to extract records is shown in Figure 12.

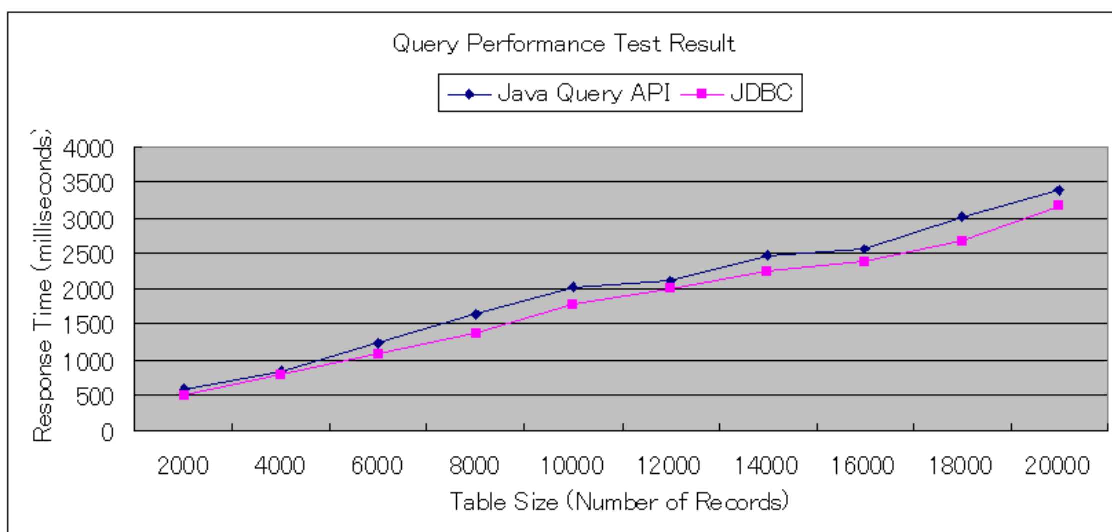

Figure 12 Query Performance Result

Each point represents the average value from five measurements.

As can be seen in the above figure, Java Query API shows a performance overhead. Response times for both Java Query API and JDBC increase in linear fashion. Since we have no control over when JVM runs background processes such as the Garbage Collector, minor variations in the response time were expected.

7-2. Thoughts on Java Query API Performance Overhead

The response time difference between Java Query API and JDBC does not exceed 300 milliseconds for any sample size, from 2000 to 20000 records, that was extracted.

We believe the performance difference is subtle and hence negligible enough for Java Query API to be used in applications for various purposes. For time-critical applications that require millisecond accuracy, however, users may consider the average number of records that are to be queried, and decide whether to use Java Query API or JDBC.

8. References

[1] Introduction to SQL http://www.w3schools.com/sql/sql_intro.asp

[2] SQL Overview http://www.w3schools.com/sql/sql_intro.asp

[3] The JDBC API http://java.sun.com/products/jdbc/overview.html

[4] JDBC Data Access API http://java.sun.com/products/jdbc/

[5] The Java Tutorial
http://java.sun.com/docs/books/tutorial/i18n/intro/index.html

[6] Migrating to Oracle from MySQL
http://www.zope.org/Members/peterb/mysql_to_oracle