DEVELOPING MULTITHREADED DATABASE APPLICATION USING JAVA TOOLS AND ORACLE DATABASE MANAGEMENT SYSTEM IN INTRANET ENVIRONMENT

Raied Salman

Computer Information Science,
American College of Commerce and Technology – Falls Church, VA, U.S.A.
raied.salman@acct.edu

ABSTRACT

In many business organizations, database applications are designed and implemented using various DBMS and Programming Languages. These applications are used to maintain databases for the organizations. The organization departments can be located at different locations and can be connected by intranet environment. In such environment maintenance of database records become an assignment of complexity which needs to be resolved. In this paper an intranet application is designed and implemented using Object-Oriented Programming Language Java and Object-Relational Database Management System Oracle in multithreaded Operating System environment.

KEYWORDS

Intranet, Multithreads, OOP, ORDBMS, JDBC, Applets, Oracle, Java Programming Language.

1. INTRODUCTION

The Intranet technology has opened new areas of research for business application designers and implementers. The application is designed using System Development Life Cycle (SDLC) methodology [1,2,4]. The database can be stored on a database server using Oracle Database Management System and can be processed using Java Programming language [5,6,7,8]. Java is an object-oriented programming language. The peoples who have used structured programming languages C, PASCAL etc. has to refuel their programming power to accept object-based programming such as Java and C++ [10] etc. It is very difficult to decide which programming language will lead the application for development in the future. In this paper the basic concepts and tools are discussed which can be used to implement business applications in an intranet environment.

Jan Zizka et al. (Eds) : ICAITA, SAI, CDKP, Signal, NCO - 2015
pp. 67–79, 2015. © CS & IT-CSCP 2015 DOI : 10.5121/csit.2015.51507

2. APPLICATION INFRASTRUCTURE FOR INTRANET ENVIRONMENT

It is understood that each database application has to apply four basic functions, *INSERT*, *UPDATE*, *RETRIEVE* and *DELETE* on database records [1,2,3,4]. A database schema is developed using analysis and design techniques. After further refinement this schema is implemented using a specified RDBMS such as ORACLE [12,13,14] on the ORACLE Server. This schema always reflects the data requirements of the organization in which it will be implemented. These basic functions can be implemented using RDBMS selected. The main problem is with the processing of the business applications where many more functions are involved in addition to these four basic functions such as new calculations. The languages provided by the DBMS are not process-oriented so the implementer has to look for a language, which can facilitate the process implementations for business applications. Different database development modelling strategies are discussed in this paper.

2.1 Single Tier Database Design Strategy

The earlier business applications were developed using RDBMS based on an integrated model which consists of *user interface code*, application *code*, and *database libraries*. These applications ran only on a mainframe machine connected to terminals, used to make different queries on the databases. Figure 1 illustrates single-tier application infrastructure.

These business applications were simple but inefficient and did not work over Local Area Networks (LANs). This model did not scale, and the *application code* and the *user interface code* were tightly coupled to the database through database libraries. This approach did not allow multiple instances [1,11,12,13,14] of the application to communicate with each other, so there was often problem of contention between instances of the same business application [12,13]. In order to eliminate some of the contentions occurring, the two-tier database design strategy was suggested [1,2,3,4].

2.2 Two-Tier Database Design Strategy

The server technology gave birth to two-tier RDBMS models. Communication-protocol development and extensive use of LANs and WANs [12,13,14,15] allowed the database developers to create an application front-end that typically accessed data through a connection to the back-end server [14,15]. Figure 2. illustrates a two-tier database design, where the client is connected to the server through a socket [5,6,7,8,9,15] connection. The program design method is very carefully used to accommodate all types of changes taking place in database design strategies [10].

Business applications / client programs through user interface send **SQL** requests to the database server. The server responds with the requested data to the business application / client machine with the specified format, after the verification of these requests. The communication between either of them and the server is managed by the library functions provided by the venders / third party software developers [7,9,10,11]. The limitations to this application design are mentioned below.

2.2.1 Limitations of the Two-Tier Database Model

i. Two-tier models are limited by the vendor-provided library [8, 9]. Switching from one database vendor to another requires a lot of modification to the business application code running on the client machine of the two-tier model.

ii. Version control is another issue. Updating the client-side libraries provided by the vendors causes the database applications to be recompiled and redistributed in the organization [9].

iii. Vendors libraries deal with low-level data manipulation. Many basic libraries deal with fetches and updates on a single row or a column. The stored procedures can be used on the server to enhance these operations increasing the complexity of the application [12,13,14].

iv. All the logic required to use and manipulate the data is implemented in the business application on the client machine, creating large client-side runtimes. This creates a fat client [1,2,3,4].

These limitations can be fully / partially removed from the two-tier model by using a three-tier model.

2.3 Three-Tier Database Design Strategy

In this model the client application communicates with an intermediate server that provides a layer of abstraction from the RDBMS. Figure 3. illustrates this model.

The intermediate layer is designed to handle multiple client requests and manage the connection to one or more database servers. The detail for this design model can be found in [1,2,3,4,14].

3. IMPLEMENTATION OF BUSINESS APPLICATION IN INTRANET ENVIRONMENT

Business applications are mission critical applications. These have to be implemented with great care and sense of responsibility. After the analysis of the user requirements for applications, the implementers have to decide in addition to RDBMS, about the programming languages, which provide the functionality of the application with minimal changes and development time if required to install on different platforms. In the present case, Java programming language is selected to implement such application, because it is platform free language [5,6,7,8]. A segment of the Payroll System is implemented using Java programming language and ORACLE database management System.

The relations / tables, which were used to explain the implementation step are given in Appendix A.

In the** Department **table,** Dept_No **is a primary key, which has unique values for individual records.

The second table / relation used to implement **one-to-many** relationship is an ***Employee*** table. In the **Employee** table, ***Emp_No*** is a primary key, and **Emp_Dept_No** is a foreign key to create a one-to- many relationship between them.

The relationship between these two tables is represented in Fig. 4

Since a segment of a **Payroll System** is to be implemented in Java programming language, the multithreading programming technique is used [9] to reduce the development time and other resources.

4. DESCRIPTION OF MULTITHREADING TECHNIQUE IN JAVA IMPLEMENTATION

The concurrency or parallelism that computers can perform is implemented through **Operating Systems** primitives available to highly experienced system programmers [5,6,7]. Using Java programming language these primitives are made available to the application programmers too. Each application can contain threads of execution such that each thread being designated a portion of the application that may execute with other threads concurrently. Multiple threading is a powerful capability of Java language not available in C and C++ [5,6,7,8,9]. Java programming includes multithreading primitives as part of the language in the form of classes such as ***Thread***, ***ThreadGroup***, ***ThreadLocal*** and ***ThreadDeath*** of *the* ***java.lang*** package [5,6,7,8]. There are many constructor methods related to the ***Thread*** class which play an important role in the ***Thread*** class operations [5,6,7,8,9]. The thread life cycle is given in [5,6].

4.1 Connecting to the ORACLE Database System

It is difficult to join two different technologies such as **Java** based on object–orientation and **ORACLE** based on Relations (tables). Tools which are used to establish the connection between these two different technologies for Multithreaded Intranet Windows applications [5, 6] development are given below.

4.2 Java Database Connectivity (JDBC): Application Programming Interface (API)

Java programming language offers several benefits to the developer creating front-end and middle-ware applications for a database server. The platform-independent nature [5,6,7,8,9] and adaptability of Java [6,7,8,9] allows a wide variety of business applications on the client machines to connect to the database systems installed on the servers [6,7]. Enterprise JavaBeans (EJB) provides a very scalable and robust database access and persistent layer [8, 9].

Servlets and **JSP** (Java Server Pages) [8,9] provide an ideal way for *thin web browser clients* or any variety of other **HTTP**-based clients to access database resources [8, 9]. The **JDBC API** is designed to allow the application developers to create Java code that can be used to access almost any relational database without needing to continually rewrite their application code. Java **servlets**, **JSP** pages, Enterprise JavaBeans (**EJB**) and **Java** classes or any other *Java code* can use **JDBC** to connect to the database server [8,9].

4.3 The JDBC API Characteristics

Recently developed Java Development Kit version 1.4 (JDK 1.4) contains JDBC 3.0 API. It is composed of the java.sql and javax.sql packages.

i. The JDBC interface provides application developer with a single API that is uniform and database independent [9]. Its database independence is due to the availability of a set of Java interfaces that are implemented by a driver [9]. The driver is used to translate the standard JDBC calls into specific calls required by the RDBMS it supports [6,7,8,9].
ii. The business application is developed only once, and then moved to the various drivers, it means that application remains the same and only drivers are changed according to the RDBMS [7,8,9] provided by the vendors.
iii. JDBC also provides a means of allowing developers to retain the specific functionality that their database vendor offers.
iv. JDBC allows the application developers to pass query strings directly to the connected driver. These query string may or may not be ANSI SQL compatible. The query depends on the driver.
v. Every Java application (Client or J2EE) that uses JDBC must have at least one JDBC driver, and each driver is specific to the type of RDBMS under consideration [6,7,8].
vi. JDBC is not derived from Microsoft ODBC [7,8,9]
vii. JavaSoft provides a JDBC-ODBC bridge that translates JDBC calls to ODBC calls [6,7,8,9].

In order to connect business applications / client machines to various RDBMS on the servers, through JDBC are discussed below, for various database design strategies.

4.4 Single-tier JDBC Database Design Strategy

In this configuration, a business application can be connected to different database servers through JDBC interface using different drivers provided by their venders. It is illustrated in Appendix A, Figure 5.

4.5 Multi-tier JDBC Database Design Strategy

In this configuration, a middle tier is used to handle protocols and DBMS libraries implemented for the client sides. Through these protocols, business application can be implemented to access the database servers by different venders in parallel or concurrently. The drivers are dependent on the venders whereas the JDBC is independent of the drivers offered by various venders. The details of this configuration is given in [6,7,8,9] and is illustrated in Appendix A, Fig. 6

5. THE JAVA DATABASE CONNECTIVITY (JDBC) INTERFACE LEVELS

The **JDBC** has two levels of **API** interface: Driver Layer and Application Layer, which are discussed below:

i. **Driver Layer:** It handles all types of communications with a specific driver during implementation to the application layer.

ii. **Application Layer:** This layer is used by the business application developer to make calls to the database via SQL queries and retrieve the results to these queries.

The application developer is not concerned with the details of the implementation of these layers. It is necessary to understand the ***Driver layer***, and how some of the objects that are used in the ***Application layer*** are created by the driver in use [1,3,6,8]. Every driver must implement four main interfaces and one class that create connection between the Driver and Application layers.

5.1 The Driver layer and Driver Interface

Each vendor supplies a driver class called ***DriverManager*** class which controls the Application layer through the driver as an interface. ***Driver Manager*** class also performs: loading and unloading of drivers and making connections using drivers. It also performs some functions on database for login and login times out [6].

a- Driver Interface

It is important to note that every **JDBC** application must have at least one **JDBC** driver. This interface permits the ***DriverManager*** and **JDBC** Application **layer** to exist independently of the database being used. This interface implements **JDBC** driver [6,7,8,9]. Drivers use a string referred to as a *URL* with a purpose to separate the application developer from the driver developer. The syntax for such *URL* for **JDBC** driver is given as

String url = jdbc: <subprotocol>:<subname>

Where <*subprotocol*> is the type of the *driver*, and <*subname*> provides the *network-encoded database name on the server*, as in

String url = "jdbc:oracle:Depts"

In this example, the driver type is **oracle** driver, and the subname is a local database host called **Depts**.

The application developer can also include the location of the database host or instance of the database, the specific port, and user information (user-name, user-password) as in the following example:

String url = "jdbc: oracle: thin: @dbserver:1521: infs" ;

In this statement, the name of the driver is ***oracle*** driver, the name of the database server is ***dbserver*****,** the port is **1521** and database instance is ***infs*****.**

The following two statements describe the user name and password of the user.

String User = "user_name";

String Password = "user_password";

The driver interface has two important methods from practical point of view [6,7,8,9]:

i- ***public Connection connect (String url, String User, String Password) throws SQLException*****.**

In order to return the object of ***Connection*** type, the ***String url*** must match the ***url*** of the **JDBC** driver otherwise no connection will be established. The strings ***User*** and ***Password*** are also matched with those stored on the database server, ***dbserver,*** with instance ***infs*** on the thin client with port **1521**. Since it is public method, the object returned can be used by other classes. If these matches are invalid, it will throw an ***SQLException*** indicating that no connection object is returned**.**

ii- ***public boolean acceptsURL(String url) throws SQLException.***
This method is simply used to check whether the ***url*** is valid or not. If it is not, it will throw an ***SQLException***. It will not establish the connection.
The ***DriverManager*** class calls the ***Driver connect()*** method to obtain the ***Connection*** object which is the starting point for the ***Application Layer***. The ***Connection*** object is used to create ***Statement*** objects that perform queries.

The DriverManager **Class**: As the name indicates this class is used to manage ***JDBC*** drivers. Public Methods available in this class are:
i- ***public static synchronized Connection getConnection (String url, String User, String Password) throws SQLException***.
This method is used to obtain ***Connection*** object by sweeping through a vector of stored ***Driver*** classes using ***url*** and other parameter values regarding the user of the database and his password. This method is used to find a ***driver*** which returns a **Connection** object. That ***Driver*** class is used for which the driver is found. This method can be used as an overloaded method with different number of arguments.
ii- ***public static synchronized void registerDriver (java.sql.Driver driver) throws SQLException*****.**
This method stores the information of the driver interface implementation into a vector of drivers. It also stores information about security Context [7,8], that identifies where the driver came from.
iii- ***Public static void setLogWriter(java.io.PrintWriter out).***
Sets a private static **java.io.PrintWriter** reference to the **PrintWriter** object passed to the method.

b- Registration of Drivers

When ***DriverManager*** class is loaded, a static code of this class is executed to load ***jdbc.drivers***.
jdbc.drivers property can be used to define a list of colon-separated driver class names such as:
jdbc.drivers = oracle.jdbc.driver.OracleDriver**;**
Each driver name is also a class name [6,7,8], this means that class name and driver name are the same, for example, ***oracle.jdbc.driver*** is both a driver name and a class name. The ***DriverManager*** tries to load the driver through the current ***CLASSPATH*** *given in the System Environment of the computing machine***.** The ***DriverManager*** class uses the following piece of Java program to locate, load and link the named class.
Class.forName(driver).newInstance().
In case of ***oracle.jdbc.driver.OracleDriver*****,** the driver class name can be located by
Class.forName(oracle.jdbc.driver.OracleDriver);
Now use the ***DriverManager*** class method ***registerDriver()*** to register the ***oracle.jdbc.driver.OracleDriver*** driver's class instance as:
DriverManager.registerDriver (new oracle.jdbc.driver.OracleDriver());
When above statement is executed, a new instance of the driver class is registered. It will not verify whether the connection is established or not. In order to establish the connection to the database, the following method of the ***DriverManager*** class is used

// define the Connection instance
Connection sqlconn = null**;** // initially it is null
sqlconn = DriverManager.getConnection (url, User, Password);
Where url, User, and Password are declared as:

String url = "jdbc:oracle:thin:@dbserver:1521:infs";
String User = "user-name";
String Password = "user-password";
In the ***url*** string: ***dbserver*** is the name of the Oracle Server, ***1521*** is port of the machine on which this server is running and ***infs*** the instance of the **Oracle Database**. Other string variables are self -explanatory. When the connection is established and validated, the ***Application layer*** can be approached. A list of driver class names for different database management systems is given in Table 3.

In the above table, the name of the driver is also a driver class name, for example, for Oracle database system, the driver name ***Oracle.jdbc.driver.OracleDriver*** is also the driver class name, any instance of this class can be defined as **new** ***Oracle.jdbc.driver.OracleDriver()*** using constructor of this class [6,7,8], for example
Driver Driver_Name = new Oracle.jdbc.driver.OracleDriver();
The above statement creates a new instance of class ***Driver*** which can be registered with ***DriverManager*** class using ***registerDriver()*** method as
DriverManager.registerDriver(Driver_Name);

5.2 Application Layer

Application Interface: In Java programming language [8, 9], the application interface provides a means of using a general type to indicate a specific class. Three main application layer interfaces are ***Connection***, ***Statement*** and ***ResultSet*** classes. Each one of them is described below:

5.2.1 The Connection Interface:

A Connection ***object is obtained by using the*** DriverManager.getConnection() ***method call as***
Connection sqlconn = DriverManager.getConnection (url, User, Password);
where ***sqlconn*** is the ***Connection*** object returned by the called method ***DriverManager.get Connection (url, User, Password);***
where ***getConnection (url, User, Password)*** method uses three arguments ***url***, ***User*** and ***Password*** as described above.
Typical database connection include the ability to control changes made to the actual database stored through transactions [6,7,9]. When connection is created, it is in an ***auto-commit*** mode, that is, there is no **rollback** possible. After the connection from the driver is established, the application developer can set auto-commit to **false** by using ***setAutoCommit (boolean b*)** method. After setting this method call, the ***Connection*** will support both ***Connection.Commit()*** and ***Connection.rollback()*** method calls.
a-The ***Connection*** Class interface
The ***Connection*** class interface has the following methods:

i- Statement createStatement() throws SQLException : The ***Connection*** object will return an object of a ***Statement*** implementation such as
Statement sqlStatement = sqlconn.createStatement(); // use *sqlconn* Connection instance to create a statement
The ***Statement*** class object ***sqlStatement*** is implemented to execute a query if required and get a single ***ResulSet*** object

ii- PreparedStatement preparedStatement (String sql) throws SQLException: The ***Connection*** object implementation will return an instance of ***PreparedStatement*** object which is configured with **sql** string passed [8, 9]. The driver may then send the statement to the database if the driver handles the precompiled statements; otherwise the driver may wait until the ***PreparedStatement*** is executed by an ***execute()*** method

iii. void setAutoCommit (Boolean b) throws SQLException: This method sets a flag in the driver implementation that enables commit/rollback (false) or make all transactions commit immediately (true) as
sqlconn.setAutoCommit(false); // rollback all transactions

iv-void commit() throws SQLException: Makes all changes made since the beginning of the current transaction.

v- CallableStatement preparedCall(String sql) throws SQLException: The **Connection** object implementation will return an instance of a **CallableStatement. CallableStatements** are optimized for handling stored procedures. The driver may then send the **sql** string immediately when **prepareCall()** method is complete or may wait until an **execute** method executes.

vi- void rollback() throws SQLException: Drop all changes made since the beginning of the current transaction.
Mainly the ***Connection*** object interface is used to create a ***Statement*** object as
Connection sqlconn ; // declare Connection object

Statement sqlStatement ; // declare Statement object
sqlconn = DriverManager.getConnection (url, User, Password); **//** establish connection to the //database
sqlStatement = sqlconn.createStatement(); **//** create a statement object, to be used for executing a query sent to the database server

5.2.2 The Statement Interface: *Statement* Class methods

This interface is used to send **SQL** statements (**insert**, **delete**, **update** , **selec**t) to the database on the server and constructing corresponding result sets. This can also be used to create or drop tables from the database. ***SQLException*** is thrown if there is a problem with the connection of the database. The following methods are available with this interface [8,9].

i- ResultSet executeQuery(String sql) throws SQLException: Executes a single **SQL** query and returns the results in an object of type ***ResultSet***. This method can be used as
ResultSet rset ; // declare an object of a ResultSet
String sqlQuery = " select * from Dept "; **//** Dept is the name of the table on the database server
rset = sqlStatement . executeQuery(sqlQuery); **//** a result set is created

ii- int executeUpdate(String sql) throws SQLException : This method executes a single **SQL** query to return the number of rows affected rather than a set of results.

iii- boolean execute(String sql) throws SQLException: This method can be used in the following way:

a. To execute **SQL** statements that returns multiple result sets.
b. To execute for updating counts.
c. To execute stored procedures that return ***out*** and ***inout*** parameters.

This method is less commonly used in database processing than ***executeQuery()*** and ***executeUpdate()*** methods. The methods ***getResultSet(), getUpdate()*** and ***getMoreResultSet()*** are used to retrieve the returned data [7,8,9].

5.2.3. ResultSet Class Interface

The ***ResultSet*** interface defines the methods for accessing tables of data generated as a result of executing a ***Statement*** [5,6,7,8,9]. ***ResultSet*** column values may be accesses in any order, that is, they are indexed and may be selected by either the name or the number of the column. ***ResultSet*** maintains the current position of the row, starting first row of the data returned. The next() method moves to the next row of the data. The following program segment explains ***next()*** method.

```
ResultSet rset ;  // declare an object of a ResultSet
String sqlQuery = " select * from Dept ";          // Dept is the name of the table on the database
                                                   //server
rset =  sqlStatement . executeQuery( sqlQuery);    // a result set is created

// processing of the resultset

if (rset.next()) {

    // processing statements goes here
}
```

The details of the ResultSet interface are discussed in [5,6,7,8,9].

6. APPLICATION INTERFACE-STRUCTURE CHART

Application interface is defined in Appendix A, Fig. 7.
It depends on the programmer which threads he /she wants to run to create a concurrency, for example, Thread-1 and Thread-2 can be executed in concurrent states to **insert** and **display** data at the same time. Concurrency programming is a tricky job. Similarly, Thread-1 and Thread-3 can be used concurrently to **update** and **display** data in the database. Different combinations of these threads can be used to compromise between the execution and the complexity of the code developed for the application. In a single thread execution, activities take place in sequential order [5,6,7,8,9]. The complete program is given in the following section. This program is used to retrieve and display data under thread-1. The development tool used is **NetBeans** IDE 3.5.1. This IDE has partially built-in Java programming document, which can be used to code the program, thereby, minimizing the development time for business applications.

7. DESCRIPTION OF THE PROGRAM USING A SINGLE THREAD

This program is defined as a single class ***jdbcDbRetrieval*** which extends to a class ***JApplet*** and is running under a ***thread*** control to create a concurrency or parallelism. This program is running under Windows Operating System to check the effect of multithreading techniques built into Java Programming Language [6,7,8,9]. This is a unique program in itself. The other database functions such as Insert, Update and Delete are also programmed but are not given in this paper. Each one of them is an applet running under a single thread and coordinating the other threads when required.

8. CONCLUSION

Java Programming Language can be used to development Distributed or Concurrent business applications in order to decrease the development time and other resources. Java API is an important part of the application development stage where a large number of built-in class and their methods are available to take full advantage of Java Development Kit. It also provides a guideline to those who are interested in developing business applications which can be run in parallel. Using Java applications are implemented and installed on different platforms with little or no change in the coding of the applications. To incorporate all these concepts and tools a complete program to implement retrieval operation of the database is given in Appendix A.

REFERENCES

[1] R. Greg (2001): Principles of Database Systems with Internet and Java Applications, Addison Wesley New York.
[2] Jeffrey A. Hoffer, Mary B. Prescott, Fred R. McFadden (2005): Modern Database Management, Seventh Edition, Pearson-Prentice Hall, U.K.
[3] R. Peter; C. Carlos (2002): Database Systems, Fifth Edition, Course Technology, Thomson Learning, U. K.
[4] V. Michael (2004): Database Design, Application Development and Administration, Second Edition, McGrawHill, Toronto, Canada.
[5] H. M. Deitel; P. J. Deitel (2002): Java: How to Program, Fourth Edition, Prentice Hall, New Jersey, U. S. A.
[6] H. M. Deitel; P. J. Deitel (2003): Java: How to Program, Fifth Edition, Prentice Hall, New Jersey, U. S. A.
[7] H. M. Deitel; P. J. Deitel (2005): Java: How to Program, Sixth Edition, Prentice Hall, New Jersey, U. S. A.
[8] B. Kurniawan (2002): Java for the Servlets, JSP, and EJB, Techmedia, Delhi, India.
[9] H. M. Deitel; P. J. Deitel; S. E. Santry (2002): Advanced Java 2 Platform: How to Program, Prentice Hall, New Jersey, U. S. A.
[10] D. Cohoon (2004): Java 1.5: Program design, McGrawHill, U. K.
[11] C. Thomas Wu (2004): An Introduction to Object-Oriented Programming with Java, Third Edition, McGraw-Hill, U. K.
[12] J. Adolph Palinski (2003): Oracle 9i Developer: Developing Web Applications with Forms Builder, Thomson, U.K.
[13] J. Morrison; M. Morrison (2003): Guide to Oracle 9i, Thomson, U. K.
[14] M. A. Ajiz (2002): E-Commerce Systems development: Case Study, Pakistan Journal of Applied Sciences 2 (2): pp.245-259, Lahore, Pakistan.
[15] R. Greenlaw; E. Hepp (1999): Fundamentals of the Internet and World Wide Web, McGraw-Hill, Toronto, Canada

Appendix A

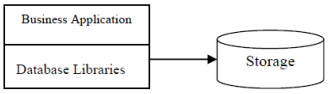


Figure 1. Single-tier database design

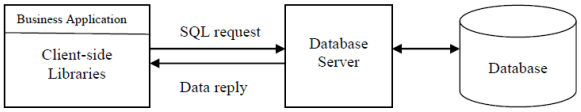


Figure 2. Two-tier database design

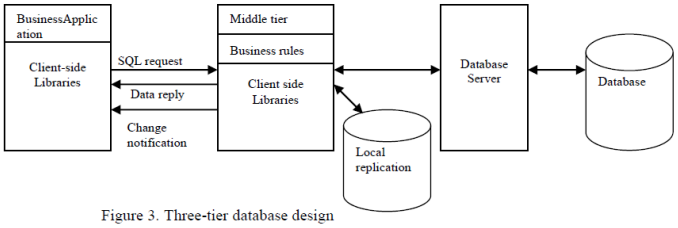


Figure 3. Three-tier database design

Figure 6: Multi-tier JDBC database design

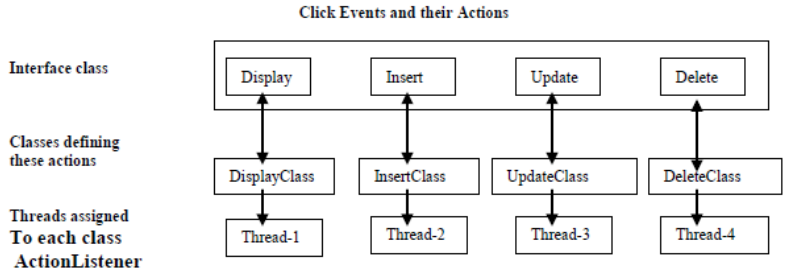


Figure 7: Application Interface

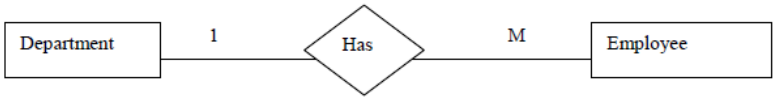


Fig. 4: One-to-many relationship

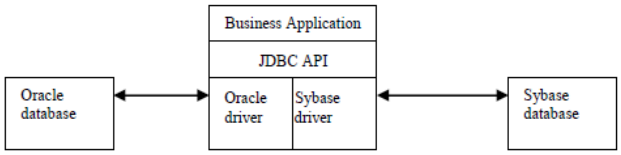
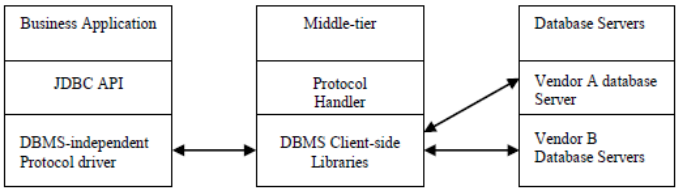


Figure 5: Single-tier JDBC database design strategy



Department Table

Column / Field Name	*Data Type*
	Number (3)
Dept_Name	Varchar2 (20)
Dept_Loc	Varchar2 (20)

Table 1: Department

Employee Table

Column / Field Name	*Data Type*
Emp_No	***Number (4)***
Emp_Name	***Varchar2 (20)***
Emp_Job	***Varchar2 (20)***
Emp_HDate	***Date***
Emp_Sal	***Number (7, 2)***
Emp_Dept_No	***Varchar2 (3)***

Table 2: Employee

RDBMS	**JDBC driver name**	**Database URL format**
MySQL	Com.mysql.jdbc.Driver	Jdbc:mysql://hostname/database-Name
ORACLE	Oracle.jdbc.driver.OracleDriver	jdbc:oracle:thin:@hostname:portNumber:
DB2	COM.ibm.db2.jdbc.net.DB2Driver	jdbc:db2:hostname:portnumber/databaseName
Sybase	com.sybase.jdbc.SybDriver	jdbc:sybase:Tds:hostname:portnumber/databaseName

Table 3 : List of JDBC Drivers and their Classes

AUTHORS

Dr. Raied Salman received his second Ph.D. in computer science from the Department of Computer Science at Virginia Commonwealth University (Richmond / USA). He also received his first Ph.D. from Brunel University (England / UK) in Electrical Engineering and both Bachelor degree and Master degree of Electrical Engineering from The University of Technology (Baghdad / Iraq). His research interests include machine learning and data mining.