Three Approaches to MySQL Applications on Dell PowerEdge Servers

Enterprise Product Group (EPG)

By Dave Jaffe and Todd Muirhead

August 2005

Contents

Section 1

Executive Summary

To demonstrate best practices for developing web applications on Dell™ PowerEdge™ servers, three popular software platforms were used to implement the front end of the same online store application. The online DVD store was coded in PHP, ASP.NET and JSP, using the same MySQL database back end. All code can run either on Windows Server 2003 or on Linux (using the Mono .NET platform for ASP.NET on Linux). This paper describes in detail the three implementations, gives installation instructions, and provides sample code fragments showing how each language connects to MySQL, calls stored procedures or inserts, and handles transactions.

Section 2

Introduction

Dell Inc.'s line of industry standards-based PowerEdge servers are a natural platform for open source software such as the MySQL Database from MySQL AB. In a recently announced partnership (see www.dell.com/mysql) the MySQL Network is now available direct from Dell. Dell does not force its customers into a single operating system. Windows Server™ 2003 and both Red Hat® Linux® and Novell's SUSE® Linux are available, factory installed, on Dell servers. MySQL runs well and is fully supported on all of these operating systems.

In this paper, which is based on a presentation given by the authors at the MySQL Users Conference in Santa Clara, California in April 2005 (http://conferences.oreillynet.com/cs/mysqluc2005/view/e_sess/6226), three different methods of building web applications with a MySQL backend are described. The PHP (PHP Hypertext Processor, from Zend, http://www.php.net) language is a very popular language for building web applications, with a highly developed interface to MySQL. Microsoft® Active Server pages for .NET (ASP.NET) enable web pages written in the C# language to communicate with MySQL through the MySQL Connector/NET. And Java Server Pages (JSP) from Sun Microsystems, Inc. allow web pages written in the Java language to talk to MySQL through the MySQL Connector/J.

All of these application platforms run under both Windows® and Linux. The ASP.NET pages require the use of the Mono platform (http://www.mono-project.com/Main_Page) from Novell to run on Linux.

The MySQL database used in this test is similar to that used in a recent paper, "MySQL Network and the Dell PowerEdge 2800: Capacity Sizing and Performance Tuning Guide for Transactional Applications", available at http://www.dell.com/downloads/global/solutions/mysql_network_2800.pdf. For this paper the database was moved to MySQL 5.03 beta to demonstrate the use of advanced database technology such as stored procedures. All of the code described here and the DVD store database code has been made available as open source code by Dell under the GNU Public License and may be obtained from linux.dell.com/dvdstore.

The overall application architecture, web page design and MySQL database are discussed in Sections 3 – 5. Following that, the PHP, ASP.NET and JSP implementations of the application are shown in Sections 6 – 8.

Section 3

DVD Store Application Architecture

The software system used to demonstrate the different ways to build applications with MySQL is a three tiered e-commerce program, representing an online DVD Store. As shown in Figure 1, the three tiers are Users, an Application Layer and a Database Layer.

The Users layer represents customers using web browsers to search for and purchase DVDs on the online DVD store.

The Application Layer consists of a web server (such as Microsoft® Internet Information Server (IIS) on Microsoft® Windows® or Apache from the Apache Software Foundation on Linux) which hosts the web pages that make up the application. The web pages, written in the PHP, ASP.NET or JSP language, contain code that reads the requests submitted by the user, accesses the backend MySQL database and writes the appropriate Hypertext Markup Language (HTML) code back to the browser. Often, several PowerEdge servers host the web servers that constitute the Application Layer. The PowerEdge 1855 blade server, which offers 10 2-processor blade servers in a 7-rack unit (12.25") chassis, makes a great platform for the Application Layer.

The Database Layer consists of the MySQL® Database Server from MySQL AB, running under one or more clustered PowerEdge servers. The MySQL database used in the test is described in detail in Section 5.

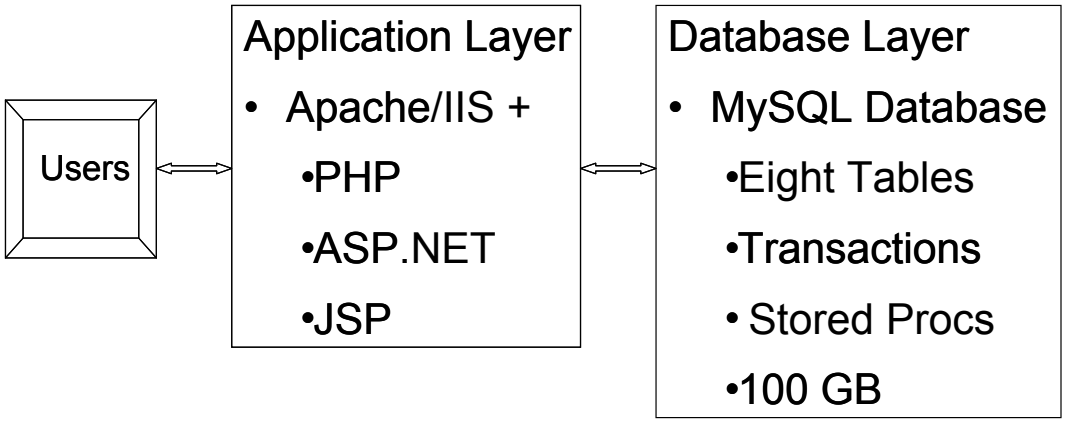

Figure 1: DVD Store Application Architecture

Section 4

The Web Pages

The DVD Store application consists of four web pages: Login, NewCustomer, Browse and Purchase. For an example see the Purchase page in Figure 2.

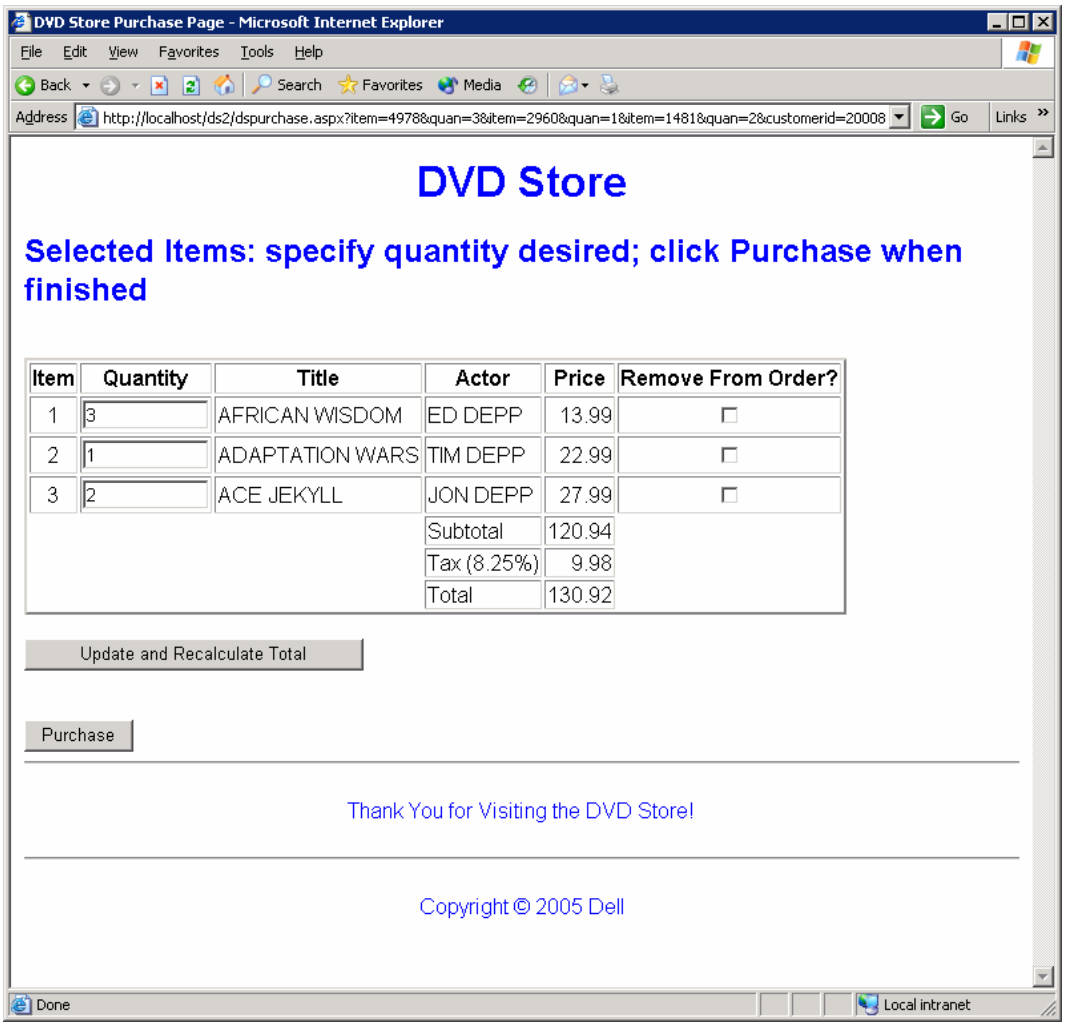

Figure 2: DVD Store Purchase Page

Customers who have already created an account use the Login page to start a new order. The code in the Login page checks the user name and password entered by the customer, and looks up the customer's account number. Additionally the page returns the previous ten titles ordered by the customer and, for each title, a title recommended by others who also enjoyed that title.

New customers use the NewCustomer page to create a new account by entering a username, personal data and credit card information. The NewCustomer code first checks that the new username is not already in use, and then inserts a new row in the CUSTOMERS table with all the information entered on the page.

After successful login or new account creation, the customer sees the Browse page, which enables the customer to search for DVDs by title, lead actor or category. Titles returned by the searches may be added to the customer's shopping cart.

Finally, the Purchase page allows the user to specify quantities, optionally delete titles from the shopping cart, and finally complete the purchase. The code in the Purchase page first checks that there is sufficient quantity in stock for every title in the order, then updates the appropriate database tables. For simplicity there is no partial order handling in this version.

Section 5

The MySQL Database

The MySQL database backend used with these applications is a large database (100 GB total size), representing an online DVD store with 1 million DVD titles, 200 million customers and 120 million orders. Advanced database features such as transactions and referential integrity constraints have been utilized.

The Database Schema

The MySQL DVD store database consists of seven main tables and one other small table (see Table 1).

Table	Columns	Number of Rows
CUSTOMERS	CUSTOMERID, FIRSTNAME, LASTNAME, ADDRESS1, ADDRESS2, CITY, STATE, ZIP, COUNTRY, REGION, EMAIL, PHONE, CREDITCARDTYPE, CREDITCARD, CREDITCARDEXPIRATION, USERNAME, PASSWORD, AGE, INCOME, GENDER, PROD_ID_IDX, PROD_ID1, PROD_ID2 … PROD_ID10	200 million
ORDERS	ORDERID, ORDERDATE, CUSTOMERID, NETAMOUNT, TAX, TOTALAMOUNT	120 million
ORDERLINES	ORDERLINEID, ORDERID, PROD_ID, QUANTITY, ORDERDATE	600 million
CUST_HIST	CUSTOMERID, ORDERID, PROD_ID	600 million
PRODUCTS	PROD_ID, CATEGORY, TITLE, ACTOR, PRICE, SPECIAL, COMMON_PROD_ID1, COMMON_RATING1, COMMON_PROD_ID2, COMMON_RATING2, COMMON_PROD_ID3, COMMON_RATING3	1 million
INVENTORY	PROD_ID, QUAN_IN_STOCK, SALES	1 million
REORDER	PROD_ID, DATE_LOW, QUAN_LOW, DATE_REORDERED, QUAN_REORDERED, DATE_EXPECTED	Variable
CATEGORIES	CATEGORY, CATEGORYNAME	16

Table 1: DVD Store Database Schema

The CUSTOMERS table was pre-populated with two hundred million customers, one hundred million US customers and one hundred million customers from the rest of the world. The ORDERS table was pre-populated with ten million orders per month for a full year. The ORDERLINES and CUST_HIST tables were pre-populated with an average of 5 items per order. The PRODUCTS table contains one million DVD titles, each with a principal actor listed for search purposes.

For realism, titles and actor names were generated by taking combinations of real movie titles and actor names, as seen in Figure 2. Additionally, the CATEGORIES table contains the 16 DVD categories.

The schema is fully documented in the database build script in Appendix A.

InnoDB Tables, MyISAM Tables and Indexing

All of the tables used in the DVD Store database were created as InnoDB tables except for the PRODUCTS table which was created as a MyISAM table. InnoDB was used specifically because of its support for transactions and foreign keys. MyISAM was used for PRODUCTS to take advantage of the full text search capability. InnoDB tables do not have the option of creating a full text index type and so can only be searched with a select statement using wild cards around the string. With a MyISAM table it was possible to create a full text index type on the ACTOR and TITLE columns and then use a SELECT statement like the one below to do a full text search for a given string:

select * from PRODUCTS where MATCH (TITLE) AGAINST ('WIND');

The select using wildcards for a title or actor string against the million row PRODUCTS table setup as an InnoDB table took approximately 3 seconds. A select for a title or actor string using the full text index and the MATCH syntax took less than a half a second against the same PRODUCTS table setup as a MyISAM table.

Foreign key constraints were added to maintain database consistency. For example, ORDERLINES rows without a corresponding ORDER are not allowed. Similarly, if an ORDER row is deleted, all the ORDERLINES that were part of that order are deleted.

The database indexing scheme is documented in Appendix B.

Section 6

The PHP Application

PHP5 Setup

PHP (PHP Hypertext Processor, from Zend, http://www.php.net) is a widely-used general-purpose scripting language that is especially suited for Web development. To take advantage of the latest features in the Improved MySQL Extension (mysqli) package for connecting to MySQL, PHP 5 was used. The PHP5 binaries for Windows are currently shipping. However to use PHP5 with Linux it is necessary to obtain the source code from www.php.net (we used php-5.0.4.tar.gz) and compile it as follows. In the directory into which php.5.0.4.targ.gz is uncompressed and expanded:

```
configure --with-apxs2=/usr/sbin/apxs --with-mysqli=/usr/bin/mysql_config
make
make install
```

(This requires that the Apache development kit package, httpd-devel-2.0.46-44.ent.i386.rpm, is already installed.)

Using PHP with MySQL

The web interface to the MySQL DVD Store database was implemented through four PHP pages, *dslogin.php*, *dsnewcustomer.php*, *dsbrowse.php* and *dspurchase.php*. Mysqli code imbedded in the web pages handles the interface to MySQL.

MySQL Connection

The connection to the database is created via the command

```
if (!($link_id = mysqli_connect())) die(mysqli_connect_error());
```

This same command appears in all four web pages. The connection is maintained by PHP throughout the user's session. In this command, if PHP detects that there is no existing connection (such as in the Login page), it will create a new connection and place the connection's identifying information in *$link_id*. If the connection already exists for that session, it will leave the identification of the connection in *$link_id*. If there is a failure in calling the connection, the program will exit (as called by the *die* function) while calling the *mysqli_connect_error()* function to get the cause of the error and display it on the browser.

MySQL Stored Procedure Call

Two steps are required to get a return value from a MySQL stored procedure. This is illustrated in the NewCustomer page, where the NEW_CUSTOMER stored procedure (Appendix C) takes the new customer's information as input

and returns the account number (customer id) assigned to that customer. It is necessary to call the procedure, with the customer id stored in session variable *@customerid_out*, and then issue another query to the database to get the value assigned to *@customerid_out*.

```
$new_customer_proc_call = "call DS2.NEW_CUSTOMER(" .
"'$firstname','$lastname','$address1','$address2','$city', " .
"'$state','$zip','$country','$region','$email','$phone', " .
"'$creditcardtype','$creditcard','$creditcardexpiration'," .
"'$username','$password','$age','$income','$gender',"
"@customerid_out);";

mysqli_query($link_id, $new_customer_proc_call);

$query = "select @customerid_out;";
mysqli_real_query($link_id, $query);
$result = mysqli_store_result($link_id);
$row = mysqli_fetch_row($result);
$customerid = $row[0];
```

MySQL Transaction

The *dspurchase.php* page checks the QUAN_IN_STOCK field from the INVENTORY table for each title in the order before committing the order. This is done using a database transaction, so that if there is insufficient quantity to fill the order neither the QUAN_IN_STOCK data is updated nor is a new record written to the ORDERS table. To implement that from the PHP application layer the PHP command, *mysql_query('START TRANSACTION;')*, is issued before the inventory check and inserts into the ORDERS and ORDERLINES tables are performed. Depending on the success or failure of the entire sequence (represented in the code by whether the $success variable is TRUE or FALSE) the INVENTORY, ORDERS and ORDERLINE updates are either all committed or all rolled back. The PHP code to do this is

```
if ($success) mysql_query('COMMIT;');
   else  mysql_query('ROLLBACK;');
```

Section 7

The ASP.NET Application

ASP.NET Setup

Microsoft Active Server Pages for .NET (ASP.NET) enable web pages written in any .NET-compliant language (Visual Basic.NET and C# are the two most common) to take advantage of .NET Common Language Runtime facilities such as ActiveX Data Objects (ADO.NET) database connectivity. ASP.NET is part of Microsoft's Internet Information Server (IIS) which comes with Microsoft Server 2003, and is well integrated into the Visual Studio.NET code development platform. The output of this compiler is a dynamic link library (DLL) that provides the compiled instructions that run the application.

A second approach is to run the same ASP.NET pages under the Apache web server on Linux, using Mono, an open source Common Language Runtime environment from Novell (http://www.mono-project.com). The ASP.NET DLL compiled under Visual Studio.NET may be used directly or the ASP.NET pages may be recompiled with mcs, the Mono C# compiler. Complete instructions for using ASP.NET under Mono are included in the distribution at dell.linux.com/dvdstore.

Using ASP.NET with MySQL

The four DVD Store web pages in the ASP.NET implementation are named *dslogin.aspx*, *dsnewcustomer.aspx*, *dsbrowse.aspx*, and *dspurchase.aspx* but each of these pages are backed by "code behind" C# files (named *dslogin.aspx.cs*, etc.) that contain the actual code.

MySQL Connection

The connection from ASP.NET to MySQL is provided by the MySQL Connector/Net, available at http://www.mysql.com/products/connector/net. Version 1.04 was used in this study. The syntax follows very closely that of the ADO.NET connector for Microsoft SQL Server. A connection string defining the properties of the connection is used to create a new *MySqlConnection* object, whose *Open()* method is then invoked to open the connection. Then a *MySqlCommand* object is created using the connection and the database query. Finally the command is executed and the result (if any) is returned. For example, in the *dslogin.aspx.cs* code, after the customer enters their username and password, the program queries the database for the customer's id number. In this case, the command is performed using *ExecuteScalar* since only a single return value is expected. The code is:

```
conn_str = "Server=localhost;UserID=web;Password=web;”  +
  “Database=DS2;Pooling=false";
conn = new MySqlConnection(conn_str);
conn.Open();
db_query="select CUSTOMERID FROM DS2.CUSTOMERS where USERNAME='" +
  username + "' and PASSWORD='" + password + "';";
cmd = new MySqlCommand(db_query, conn);
customerid = (int) cmd.ExecuteScalar();
```

Here the variables username and password are read from the browser input by code earlier in the program.

MySQL Stored Procedure Call

In a similar fashion as the php implementation, the ASP.NET Stored Procedure Call is accomplished with two separate database queries, one to call the stored procedure, and a second one to retrieve the session variable containing the result. The call to the NEW_CUSTOMER stored procedure in *dsnewcustomer.aspx.cs*, for example, is:

```
db_query = "call DS2.NEW_CUSTOMER('" + firstname + "','" +
  lastname + "','" + address1 + "','" + address2 + "','" + city +
  "','" + state + "','" + zip +  "','" + country + "','" + region
  + "','" + email + "','" + phone + "','" + creditcardtype + "','"
  + creditcard + "','" + creditcardexpiration + "','" +
  username + "','" + password + "','" + age +"','" + income +
  "','" + gender + "',@customerid_out);";

cmd = new MySqlCommand(db_query, conn);
cmd.ExecuteNonQuery();
db_query = "select @customerid_out;";
cmd = new MySqlCommand(db_query, conn);
Convert.ToInt32(cmd.ExecuteScalar().ToString());
```

MySQL Transaction

As in the php application, the *dspurchase.aspx.cs* page handles the end of the purchase transactionally, that is, it will only decrement the QUAN_IN_STOCK for each title in the order in the INVENTORY table and insert new rows into the ORDERS and ORDERLINES tables upon successful completion of the entire purchase process (that is, that all the requested titles have sufficient stock on hand, and the inserts happen successfully). If any of these pieces fail the entire transaction will be rolled back.

The BeginTransaction() method of the connection object is used to start a transaction:

```
trans = conn.BeginTransaction(IsolationLevel.RepeatableRead);
```

Here the optional parameter is used to set the isolation level (this will be the default in the next release of the code).

An overloaded version of the MySqlCommand constructor is then invoked to specify that the next queries are part of the transaction:

```
cmd = new MySqlCommand(db_query, conn, trans);
```

Finally, after executing the commands that are part of the transaction the code either commits or rolls back the entire transaction:

```
if (success)trans.Commit();
  else trans.Rollback();
```

Section 8

The JSP Application

About JSPs, JVMs, JConnector and Tomcat

Java Server Pages allow for Java code to be put directly into an HTML web page. The Java code actually gets executed on the server and the results are returned as part of the web page. Simple tags are used to identify what is Java code on the otherwise HTML page.

There is an Apache subproject called Jakarta where the Tomcat JSP server is maintained and developed. To run the JSPs used in this paper Tomcat was installed and used standalone, however any Java 2 Enterprise Edition (J2EE) application server, such as JBoss, IBM's WebSpere, BEA's WebLogic, and Oracle's 10gAS, should be able to run these same JSP pages without change.

There are several components involved in getting JSPs to work with MySQL. A Java Virtual Machine or JVM is required. The J2SE5 for Linux version was used in this testing. The Tomcat server is Java based so it only requires that the JVM be installed. Tomcat 5.5.7 was used, which was the current version at the time of testing. The final component is the MySQL J Connector from MySQL (http://www.mysql.com/products/connector/j/). This is the JDBC (Java Database Connector) implementation for MySQL which provides all the necessary code to allow Java JDBC calls to be translated into the correct MySQL code.

Setup consisted of decompressing and extracting the JVM and Tomcat installation files to desired locations and including the JConnector Java Archive Repository (JAR) file in the Tomcat classpath. Additionally the session variables CATALINA_HOME and JRE_HOME were set to match where Tomcat and Java respectfully had been installed.

Tomcat is started with startup.sh which is located in the Tomcat /bin directory. By default Tomcat runs on port 8080, so to access it the following URL would be used in the browser:

http://<hostname>:8080

Using JSP with MySQL

The MySQL JConnector is the implementation for the JDBC API for MySQL. This makes using MySQL in a JSP or any Java program the same as using any other JDBC compliant database. The MySQL driver from JConnector is specified initially and then everything else is standard JDBC.

MySQL Connection

In the code sample below from the *login.jsp* page a typical set of JDBC calls is made. In this case it is to check the login *userid* and *password*. First the MySQL jdbc driver is selected to be used. Then a connection object is created. A statement object is then created and executed with the contents of the query that needs to be executed on the database. The results are contained in the result set object. Before attempting to read from the result set object it is necessary to call the *next()* method which here is done in the if conditional.

```
try { Class.forName("com.mysql.jdbc.Driver"); }
catch (Exception e) {System.out.println("Error");}
Connection conn =
  DriverManager.getConnection("jdbc:mysql:///DS2?user=web");
String query = "select CUSTOMERID FROM DS2.CUSTOMERS where USERNAME='" +
  user_name + "' and PASSWORD='" + password + "';";
Statement userqueryStatement = conn.createStatement();
ResultSet userqueryResult = userqueryStatement.executeQuery(query);
if (userqueryResult != null && userqueryResult.next())
```

MySQL Insert

When doing an insert into the database the *executeUpdate()* method instead of the *executeQuery()* method of the statement object is used. The code fragment below if from the *dsnewcustomer.jsp* page where a new user record is being inserted into the database. In this case we need to return the auto generated *customerid* to the application. This is done by enabling RETURN_GENERATED_KEYS in the *executeUpdate()* method and then getting the auto generated *customerid* with the *getGeneratedKeys()* method on the statement object. This *getGeneratedKeys()* method is equivalent to the MySQL function *LAST_INSERT_ID().*

```
try {   Class.forName("com.mysql.jdbc.Driver");   }
catch (Exception e) {System.out.println("Error opening connection");}
Connection newuserconn =
  DriverManager.getConnection("jdbc:mysql://localhost/DS2?user=web");
Statement userInsertStatement = newuserconn.createStatement();
userInsertStatement.executeUpdate(insert_newuser_query,
  Statement.RETURN_GENERATED_KEYS);
       // the RETURN_GENERATED_KEYS option on the executeUpdate is needed
       // for the auto increment customerid column to be returned.
ResultSet userInsertResult = userInsertStatement.getGeneratedKeys();
userInsertResult.next();
String customerid = userInsertResult.getString(1);   // get customerid
into string
```

MySQL Transaction

In order to do transactions in MySQL with a JSP the JDBC implementation of working with transactions is used. A connection object is created with its *AutoCommit* property set to false. By default *AutoCommit* is *true* meaning that every statement executed against the database is committed immediately. By setting it to false, a transaction is started and is not completed until a commit is issued at the end of the transaction. If any SQL error occurs, then a rollback is issued or if any business logic requires a rollback a rollback is issued. In the example below a purchase transaction is being completed on the *dspurchase.jsp* page which consists of entries into multiple tables.

```
try
  {
  orderconn =
    DriverManager.getConnection("jdbc:mysql:///DS2?user=web");
  orderconn.setAutoCommit(false); // tell connection to not commit until
    instructed
  Statement purchaseupdateStatement = orderconn.createStatement();
  purchaseupdateStatement.executeUpdate
   (purchase_insert_query,Statement.RETURN_GENERATED_KEYS);
  ResultSet orderIDResult = purchaseupdateStatement.getGeneratedKeys();
    // to get orderid
  orderIDResult.next();
  orderid = orderIDResult.getString(1);
  // loop through purchased items and make inserts into orderlines table
    …
  purchaseupdateStatement.executeUpdate(p_query);
    // Insert into orderlines
  purchaseupdateStatement.executeUpdate(c_update);
    // Update customers with recent purchases
  if ( success == true )  // if no errors were found, commit all
    {orderconn.commit();}
  else
    {orderconn.rollback();}        // otherwise, rollback
  orderconn.close();
  }  //end of try for order entry transaction
catch  (SQLException e)  // if any SQL exceptions were thrown, rollback
  {
  if (orderconn != null)
    {
    try
     { orderconn.rollback(); }
    catch (SQLException rbexception)
     {rbexception.getMessage()}
  }
```

Section 9

Conclusions

This paper illustrates how three very different web application languages can be used to build on line applications running on Dell PowerEdge servers, either under Windows or any Linux variant. The choice of the language is entirely up to the developer. PHP is probably the easiest to interface to MySQL and to code web pages. ASP.NET utilizes C# skills and enables use of the .NET Framework. And JSP utilizes the large population of Java developers.

The installation tips and code fragments shown in the paper, together with the full code available for free from linux.dell.com/dvdstore, should accelerate the learning curve of any developer interested in building applications with one of these three approaches.

Appendix A. mysqlds2 Build Script

```
-- mysqlds2_create_db.sql: DVD Store Database Version 2.1 Build Script - MySQL
version
-- Copyright (C) 2005 Dell, Inc. dave_jaffe@dell.com and todd_muirhead@dell.com
-- Last updated 5/13/05


-- Database

DROP DATABASE IF EXISTS DS2;
CREATE DATABASE DS2;
USE DS2;


-- Tables

CREATE TABLE CUSTOMERS
  (
  CUSTOMERID INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
  FIRSTNAME VARCHAR(50) NOT NULL,
  LASTNAME VARCHAR(50) NOT NULL,
  ADDRESS1 VARCHAR(50) NOT NULL,
  ADDRESS2 VARCHAR(50),
  CITY VARCHAR(50) NOT NULL,
  STATE VARCHAR(50),
  ZIP INT,
  COUNTRY VARCHAR(50) NOT NULL,
  REGION TINYINT NOT NULL,
  EMAIL VARCHAR(50),
  PHONE VARCHAR(50),
  CREDITCARDTYPE INT NOT NULL,
  CREDITCARD VARCHAR(50) NOT NULL,
  CREDITCARDEXPIRATION VARCHAR(50) NOT NULL,
  USERNAME VARCHAR(50) NOT NULL,
  PASSWORD VARCHAR(50) NOT NULL,
  AGE TINYINT,
  INCOME INT,
  GENDER VARCHAR(1)
  )
  TYPE=InnoDB;

CREATE TABLE CUST_HIST
  (
  CUSTOMERID INT NOT NULL,
  ORDERID INT NOT NULL,
  PROD_ID INT NOT NULL
  )
  TYPE=InnoDB;

CREATE TABLE ORDERS
  (
  ORDERID INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
  ORDERDATE DATE NOT NULL,
  CUSTOMERID INT,
  NETAMOUNT NUMERIC(12,2) NOT NULL,
  TAX NUMERIC(12,2) NOT NULL,
  TOTALAMOUNT NUMERIC(12,2) NOT NULL
  )
  TYPE=InnoDB;
```

```
CREATE TABLE ORDERLINES
  (
  ORDERLINEID SMALLINT NOT NULL,
  ORDERID INT NOT NULL,
  PROD_ID INT NOT NULL,
  QUANTITY SMALLINT NOT NULL,
  ORDERDATE DATE NOT NULL
  )
  TYPE=InnoDB;

CREATE TABLE PRODUCTS
  (
  PROD_ID INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
  CATEGORY TINYINT NOT NULL,
  TITLE VARCHAR(50) NOT NULL,
  ACTOR VARCHAR(50) NOT NULL,
  PRICE NUMERIC(12,2) NOT NULL,
  SPECIAL TINYINT,
  COMMON_PROD_ID INT NOT NULL
  )
  ;

CREATE TABLE INVENTORY
  (
  PROD_ID INT NOT NULL PRIMARY KEY,
  QUAN_IN_STOCK INT NOT NULL,
  SALES INT NOT NULL
  )
  TYPE=InnoDB;


CREATE TABLE CATEGORIES
  (
  CATEGORY TINYINT NOT NULL AUTO_INCREMENT PRIMARY KEY,
  CATEGORYNAME VARCHAR(50) NOT NULL
  )
  TYPE=InnoDB;


  INSERT INTO CATEGORIES (CATEGORY, CATEGORYNAME) VALUES (1,'Action');
  INSERT INTO CATEGORIES (CATEGORY, CATEGORYNAME) VALUES (2,'Animation');
  INSERT INTO CATEGORIES (CATEGORY, CATEGORYNAME) VALUES (3,'Children');
  INSERT INTO CATEGORIES (CATEGORY, CATEGORYNAME) VALUES (4,'Classics');
  INSERT INTO CATEGORIES (CATEGORY, CATEGORYNAME) VALUES (5,'Comedy');
  INSERT INTO CATEGORIES (CATEGORY, CATEGORYNAME) VALUES (6,'Documentary');
  INSERT INTO CATEGORIES (CATEGORY, CATEGORYNAME) VALUES (7,'Drama');
  INSERT INTO CATEGORIES (CATEGORY, CATEGORYNAME) VALUES (8,'Family');
  INSERT INTO CATEGORIES (CATEGORY, CATEGORYNAME) VALUES (9,'Foreign');
  INSERT INTO CATEGORIES (CATEGORY, CATEGORYNAME) VALUES (10,'Games');
  INSERT INTO CATEGORIES (CATEGORY, CATEGORYNAME) VALUES (11,'Horror');
  INSERT INTO CATEGORIES (CATEGORY, CATEGORYNAME) VALUES (12,'Music');
  INSERT INTO CATEGORIES (CATEGORY, CATEGORYNAME) VALUES (13,'New');
  INSERT INTO CATEGORIES (CATEGORY, CATEGORYNAME) VALUES (14,'Sci-Fi');
  INSERT INTO CATEGORIES (CATEGORY, CATEGORYNAME) VALUES (15,'Sports');
  INSERT INTO CATEGORIES (CATEGORY, CATEGORYNAME) VALUES (16,'Travel');


CREATE TABLE REORDER
  (
  PROD_ID INT NOT NULL,
  DATE_LOW DATE NOT NULL,
  QUAN_LOW INT NOT NULL,
  DATE_REORDERED DATE,
```

```
QUAN_REORDERED INT,
DATE_EXPECTED DATE
)
TYPE=InnoDB;
```

Appendix B. mysqlds2 Index Creation Script

```
-- mysqlds2_create_ind.sql: DVD Store Database Version 2.1 Create Indexes
Script - MySQL version
--Copyright (C) 2005 Dell, Inc. dave_jaffe@dell.com and todd_muirhead@dell.com

-- Last updated 5/13/05


USE DS2;

CREATE UNIQUE INDEX IX_CUST_USERNAME ON CUSTOMERS
  (
  USERNAME
  );

CREATE INDEX IX_CUST_HIST_CUSTOMERID ON CUST_HIST
  (
  CUSTOMERID
  );

ALTER TABLE CUST_HIST
  ADD CONSTRAINT FK_CUST_HIST_CUSTOMERID FOREIGN KEY (CUSTOMERID)
  REFERENCES CUSTOMERS (CUSTOMERID)
  ON DELETE CASCADE
  ;

CREATE INDEX IX_ORDER_CUSTID ON ORDERS
  (
  CUSTOMERID
  );

ALTER TABLE ORDERS
  ADD CONSTRAINT FK_CUSTOMERID FOREIGN KEY (CUSTOMERID)
  REFERENCES CUSTOMERS (CUSTOMERID)
  ON DELETE SET NULL
  ;

CREATE UNIQUE INDEX IX_ORDERLINES_ORDERID ON ORDERLINES
  (
  ORDERID, ORDERLINEID
  );

ALTER TABLE ORDERLINES
  ADD CONSTRAINT FK_ORDERID FOREIGN KEY (ORDERID)
  REFERENCES ORDERS (ORDERID)
  ON DELETE CASCADE
  ;

CREATE FULLTEXT INDEX IX_PROD_ACTOR ON PRODUCTS
  (
  ACTOR
  );

CREATE INDEX IX_PROD_CATEGORY ON PRODUCTS
  (
  CATEGORY
```

```
  );

CREATE FULLTEXT INDEX IX_PROD_TITLE ON PRODUCTS
  (
  TITLE
  );

CREATE INDEX IX_PROD_SPECIAL ON PRODUCTS
  (
  SPECIAL
  );
```

Appendix C. NEW_CUSTOMER Stored Procedure

```
-- mysqlds2_create_sp.sql: DVD Store Database Version 2.1 Create Stored
Procedures Script - MySQL version
-- Copyright (C) 2005 Dell, Inc. dave_jaffe@dell.com and Todd_muirhead@dell.com
-- Last updated 5/13/05

Delimiter $
DROP PROCEDURE IF EXISTS DS2.NEW_CUSTOMER $
CREATE PROCEDURE DS2.NEW_CUSTOMER ( IN firstname_in varchar(50), IN lastname_in
varchar(50), IN address1_in varchar(50), IN address2_in varchar(50), IN city_in
varchar(50), IN state_in varchar(50), IN zip_in int, IN country_in varchar(50),
IN region_in int, IN email_in varchar(50), IN phone_in varchar(50), IN
creditcardtype_in int, IN creditcard_in varchar(50), IN creditcardexpiration_in
varchar(50), IN username_in varchar(50), IN password_in varchar(50), IN age_in
int, IN income_in int, IN gender_in varchar(1), OUT customerid_out INT)
  BEGIN
  DECLARE rows_returned INT;
  SELECT COUNT(*) INTO rows_returned FROM CUSTOMERS WHERE USERNAME =
username_in;
  IF rows_returned = 0
  THEN
    INSERT INTO CUSTOMERS
      (
      FIRSTNAME,
      LASTNAME,
      EMAIL,
      PHONE,
      USERNAME,
      PASSWORD,
      ADDRESS1,
      ADDRESS2,
      CITY,
      STATE,
      ZIP,
      COUNTRY,
      REGION,
      CREDITCARDTYPE,
      CREDITCARD,
      CREDITCARDEXPIRATION,
      AGE,
      INCOME,
      GENDER
      )
    VALUES
      (
      firstname_in,
      lastname_in,
      email_in,
      phone_in,
      username_in,
      password_in,
      address1_in,
      address2_in,
      city_in,
      state_in,
      zip_in,
      country_in,
      region_in,
```

```
    creditcardtype_in,
    creditcard_in,
    creditcardexpiration_in,
    age_in,
    income_in,
    gender_in
    )
    ;
  select last_insert_id() into customerid_out;
ELSE SET customerid_out = 0;
END IF;
END; $
```