MGMT 329: Database Management

PROJECT TASK 7: DML AND REPORTING

The class project will require students to apply the knowledge they learn in class in a practical manner. Each student will develop an E-Commerce database used to maintain customers, products and sales information. You are required to 1) gather and analyze requirements 2) design logical structure of the database 3) create stored procedures to develop the tables and insert the data 4) write SQL statements for data extraction and reporting. All work for this project must be on your own. This is an individual assignment, not group assignment. Anybody caught copying the work of others will receive a zero for the project.

Project Task 7: Background

Throughout the course of this semester you have analyzed the requirements for an eCommerce database, designed and developed your database. As a class we have gone through the process of writing the DDL scripts to create our database and the DML scripts to load the data. Now that we have a functional database loaded with data, we can start working with our database and performing business functions.

Maintaining the data and reporting are two tasks that are performed daily on all databases. Data is constantly being inserted, updated and deleted. Managers need reports and users execute queries to look up information. We will first take a look at some queries to get an understanding of how our database can be used. Then, you will write your own queries for reporting.

Query 1: upSell

Using data mining techniques and fuzzy logic, we are able to identify products that customers are most likely to purchase based on their purchase history or viewing habits. The *upSells* table stores the customer ID along with several products and a percentage ranking of how likely they are to purchase those products. The *upSells* information will be displayed on the Home page and Shopping Cart page to try and increase sales. We use the following query to select the products to be displayed.

```
SELECT Top 3 u.prodID, u.percentage, p.prodName, p.rPrice, p.sPrice
FROM upSells u, product p
WHERE u.prodID = p.prodID and u.custID = 215
ORDER BY u.percentage desc;
```

We want to select the product information and percentage value from our *upSells* and *product* tables. An *inner join* is used to connect these two tables. We filter the results based on the *custID* so that the customer only sees their *upSells* recommendations. Finally, the results are ordered by percentage in descending order so that the highest percentage results are displayed first. When applying this query to our actual eCommerce site, we would use a variable in the place of our *custID* value (i.e. *u.custID = @custID*). The value for this variable would be passed to our query from the programming code calling our query. Using (and introducing) the “Top #” function in

our SELECT statement, we will only display the top three results from the query. For customer 215, our query would produce the following results.

prodID	Perc	prodName	rPrice	sPrice
313234	0.82	Toshiba Canvio Connect 1TB Portable External Hard Drive	63.96	63.96
424462	0.81	Jensen JENSEN CD-750 Portable AM FM Stereo CD Play..	87.06	87.06
508339	0.76	Backyard Grill Kettle Charcoal Grill	44.00	44.00

We could also run a similar query for *crossSells*. The *crossSells* table would recommend products on the product page based on all customer purchasing habits of that product. The *prodID* would be used to filter the results based on which product was viewed.

Query 2: Billing and Shipping Address

When the user adds items to their shopping cart and decides to checkout, we need to display their billing and payment options that we have stored in the database. To do this, we should select information from the *customer*, *shippingAddress* and *paymentInfo* tables. If we join these three tables together then we will end up with both the shippingAddress and paymentInfo information stored in a single record and also have duplicate output if multiple shipping addresses or payment methods exist. To solve this problem we will introduce the UNION command. A UNION will combine the results of two or more queries into a single output. For a union to work properly, there must be an even number of attributes in both queries that we are trying to combine. Below is the query to output shipping and billing information following by a detailed description of the query.

```
SELECT 'Billing Address:', p.billAddress, Null, p.city, p.state, p.zip, ccType, ccNum, ccExpire
FROM customer c, paymentInfo p
WHERE c.custID = p.custID and c.custID = 243

UNION

SELECT 'Shipping Address', s.address1, s.address2, s.city, s.state, s.zip, Null, Null, Null
FROM customer c, shippingAddress s
WHERE c.custID = s.custID and c.custID = 243;
```

The first query will output the Billing Address records stored in the database. We use the ‘Billing Address:’ string at the beginning to output this text on each record to identify it as billing information. The second query will output shipping information and use the string ‘Shipping Address:’ string at the beginning to identify these records as shipping address records in the output. Notice the use of Null in the attributes list of the SELECT statement. The *shippingAddress* table has an *address2* attribute but the *paymentInfo* table does not. Therefore,

the word Null is entered into the SELECT statement to output Null values in order to make the attribute count equal in both queries. The *shippingAddress* table does not contain credit card information, so Null values where entered into the second query to make the number of attributes match the number of attributes from the first query. Both queries should filter the data based on the *custID = @custID* (i.e. 243 in our example). Once the two queries have been written with the desired output, we use the word UNION between the two queries. The UNION command will combine the results of both queries into one output. Notice that both queries return the same number of attributes with matching data types. Try copying this UNION in your vLab to see it in action.

Query 3: Fraud Detection

Online companies are always concerned about fraudulent transactions. Most online fraud can be detected by unusually large transactions. While there are other measures of detecting fraud, we will keep it simple and look for any orders that are greater than $1,000. Using our Sum( ) function and filtering based on the aggregate functions, we can use the following query to detect transactions that should be flagged as possible fraud.

```
SELECT o.OID, o.custID, o.orderDt, sum(oi.qty * oi.price)
FROM orders o, orderItems oi
WHERE o.OID = oi.OID
GROUP BY o.OID, o.custID, o.orderDt
HAVING sum(oi.qty * oi.price) > 1000;
```

This query identifies the customer ID and the date that the order took place. Using the Sum( ) function, we can summate the total of each order (*qty * price*) and filter using the HAVING clause to only output orders greater than 1000. These orders should require further attention to ensure that fraud is not taking place and protect our customers and our company. Try running the above query in vLab and view the results.

Project Task 7: Requirements

The most advance queries have been provided for you. Now it's your turn to write SQL queries on your own. Your task is to write the SQL statements to produce the reports needed.

Create a Microsoft Word document containing your SQL statements for each of the following reports and upload your document to UNM Learn.

Query 1: Categories and Products

We want a report that will list all the categories in our database and the total number of products we have for each category. Generate a report that will display the **category ID**, **category name** and the **total number of products for each category**. *Note: There are an equal number of products under each category in our database.*

MGMT 329: Database Management

Query 2: Customer Loyalty

We want a report that will let us know who our repeat customers are. We want to be able to track customer loyalty and provide benefits to these customers who have made our company a success. Generate a report that will display the **customer ID**, their **first** and **last name** and the **number of orders that each customer has placed** on our website. The report should only show customers that have ordered from us more than once.

Query 3: Weekly Sales Report

We want a report that will show the total sales for the week of October 12th, 2014. This report should include the total sales amount for each day from October 12th to October 18th. Generate a report that will display the **date** and **total sales for each day** for the **week of October 12th, 2014**.

Queries 4 and 5: User Generated Reports

You have written three queries to meet the reporting needs of management. Now it's your turn to create two reports of your own. You may write any SQL statement you wish based on what you have learned in the class. You should use the eCommerce tables developed in the group project to produce your reports. Write a brief explanation of each query, what it is supposed to do and how it should be used by the users. Below are the minimum requirements for each report.

Report 1: This report must **join** two or more tables together. You may use an inner join our outer join. The number of attributes you select and the filtering criteria are up to you. The minimum requirement for full credit is using a join statement and explaining the purpose of this query. You must explain the results of the query and how these results can be used by someone within the company.

Report 2: This report must contain an **advanced SQL function** (aggregate or scalar). You may select results from a single table or join multiple tables together. The number of attributes you select and the filtering criteria are up to you. The minimum requirement for full credit is using an advanced function and explaining the purpose of this query. You must explain the results of the query and how these results can be used by someone with the company.