Data Mining – Commonly Used SQL Statements

APPLICATIONS NOTE

December 2007

Description: Guide to some commonly used SQL Statements
OS Requirement: Win 2000 Pro/Server, XP Pro, Server 2003
General Requirement: You will need a Relational Database (SQL server, MSDE, Access, Oracle, etc), Installation of GENESIS32 or BizViz.

Introduction

You can use Structured Query Language (SQL) to query, update, and manage relational databases. You can also use it to retrieve, sort, and filter specific data from any database.

A database contains one or more table(s), where each table contains records with relevant information. In GENESIS32 for example, you can configured the Alarm Logger to create a table that will hold historical alarm records. You will be able to query the alarm table to list all alarms of a certain priority using SQL statements. You may also want to query the number of times an alarm occurred, calculate the total times an alarm was active, or query anything else to get more value out of your data. Once the queries are put in place, you can use ICONICS products to manufacture intelligence to the data.

This document will list some helpful SQL commands for you to query your data. We will assume that you are using an SQL Database Server.

Creating a Database

Historical alarm management requires an alarm database. You can create a new database using the SQL Server Enterprise Manager or using the CREATE DATABASE statement. In this example, we will use the Teratrex Database Manager

1. Launch the Teratrex Database Manager by going to Start → Programs → ICONICS Tools → MSDE Manager.
2. If this is the first time you are using this, accept the license agreement and click on the "Connect" button connect to your SQL Server.

   NOTE: Your SQL server must be running in order for the connection to be successful. Check to make sure that you have in your system tray that signifies your SQL Server is running.

3. Once you have connected to the database server, right-click on Databases and select New.

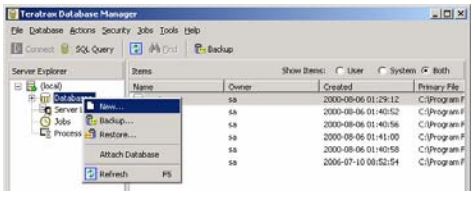

Figure 1 - MSDE Database Manager: Create a New Database

4. An SQL Query window will appear. We will execute the CREATE DATABASE code here. Your code should look similar to the code shown in Figure 2.

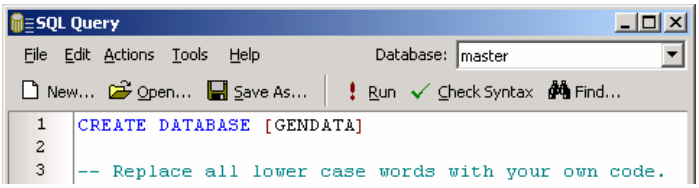

Figure 2: CREATE DATABASE Command

5. Run the code by click on the "Run" button. Your database should be available now. You can close the SQL Query widow; refresh the database server to see your newly created database.

Memory Management of a Database

By default, an MSDE database uses the maximum available computer memory. It may be better to specify the minimum/maximum amount of memory (i.e. 25%-50% of physical available memory). This would allow sufficient free memory for other applications. The following example could be used for a computer with one Gigabyte of RAM.

1. In the Teratrax Database Manager, select File → SQL Query. The SQL Query Window appears again.
2. In the Database dropdown menu, select GENDATA.
3. Enter the code as you see in Figure 3. This sets the minimum amount of RAM to 128 MB and the maximum at 512 MB.

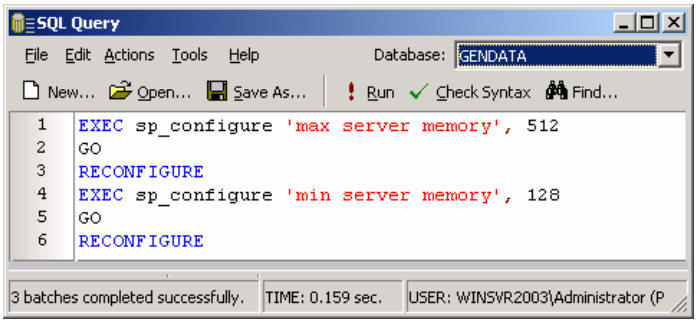

Figure 3 - Setting Max and Min Amount of Hard Disk Space

NOTE: If the SQL command successfully runs, you will get a "completed successfully" message on the bottom of the window as shown in the bottom left corner of Figure 3.

Hard Disk Management of a Database

The database needs some initial hard disk space and when it is expected to grow, we would like to specify the amount of hard disk space to allocate additionally for the required growth.

1. Open the SQL Query window and set it to query on GENDATA by following Step 1-2 in the Memory Management of a Database section.
2. Enter the code as you shown in Figure 4. This sets the initial database size to 32 Megabytes and the growth rate to 5 Megabytes.

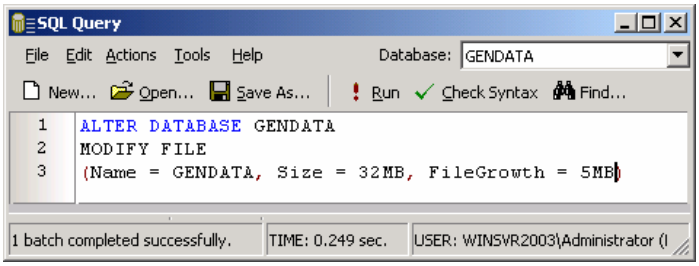

Figure 4 - Setting Initial DB Size and Growth Size

Importing Data

Imagine that we need to take the content of a Microsoft Excel spreadsheet or data in an Access database into a Microsoft SQL database. For small amounts of data, we can do it manually; however, with files containing hundreds of records, it would be much easier to import it. Following are examples of how to import different types of files into a MS SQL Database.

Microsoft Excel Files

In this example we show you how to accomplish the import with SQL Statements. We will use the RECIPE.XLS located in C:\Program Files\ICONICS\GENESIS32\Examples\GEN32DEMO. To import an Excel file, enter the code you see in Figure 5.

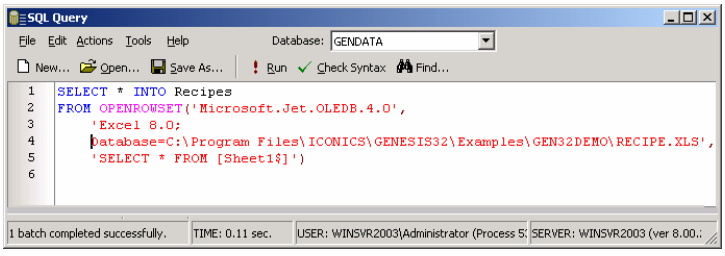

Figure 5 - Importing RECIPE.XLS into Recipes

The table named Recipes will be automatically created and the column names are automatically inherited from the first column in the Excel sheet.

CSV Files

You can also import data into a table from a comma separated variable file. You must first create the table before you can import the data. You can use the following steps to import a csv version of the RECIPE.xls.

1. Open the RECIPE.xls file and save it as a csv file.
2. Change the .csv extension to .txt. You should now have a text file RECIPE.txt.
3. Open the file and delete the first row and any dashes ("-") you see in the file. Save the file and close it.
4. Bring up the SQL Query window and let us create a new table named RecipesCSV. You can use the code shown in Figure 6 and run it.

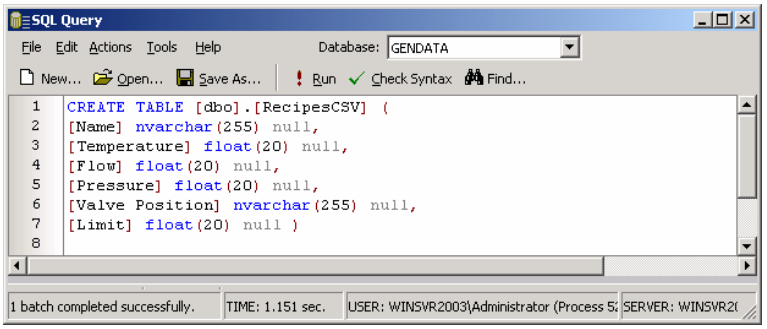

Figure 6 - Create Table RecipesCSV

5. Make sure that the table is created successfully.
6. In the SQL Query window, enter the code shown in Figure 7 and run it.

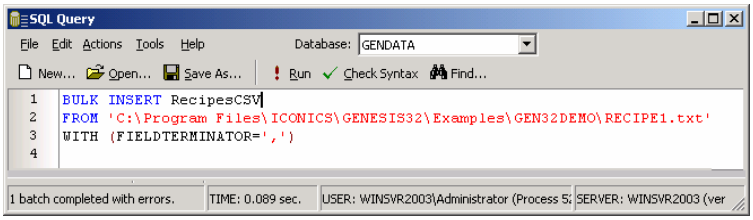

Figure 7 - BULK INSERT

7. Query the table to return all rows to make sure that you have successfully imported the data.

Deleting Data

As you get new data, it is inevitable that the old data needs to be replaced. Sometimes you can simple remove data with a specified parameter and sometimes you can just delete an entire table. This section will show you how to delete data using SQL commands.

To delete records with a specific parameter, you can use

DELETE FROM table
WHERE condition

For example, we want to delete the recipe named Cheddar in the RecipesCSV table that we have created earlier. We would use the command as shown in Figure 8 to remove the record.

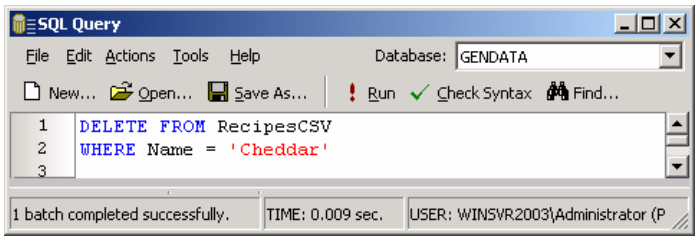

Figure 8 - Delete Record "Cheddar"

NOTE: Depending on the data type of your condition, your condition could use no quotes (numeric values), single quotes (nvarchar), or double quotes (char).

You can also delete the entire table if you wish. To do that, you can use the code shown in Figure 9. This will delete the Recipes Table that we have created in the previous section.

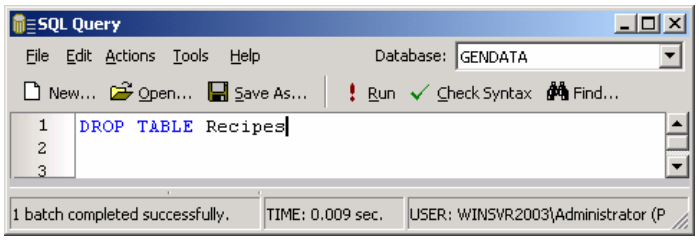

Figure 9 - Deleting Recipes Table

SQL Select Statements

When databases are available, you can use SQL to query them. Perhaps the most common SQL statement is the **SELECT** statement. The result of a select statement is usually a table of queried data.

The general form of a SELECT statement takes the form of

SELECT Comma_Separated_Select_List
FROM source
[WHERE condition(s)]
[GROUP BY expression]
[HAVING condition]
[ORDER BY expression]

In this example, we will use the Customers table in the Northwind database that comes with your SQL server installation to execute a very simple SELECT statement.

In the SELECT statement that you see in Figure 10 returns all rows and columns in the Customer's table. The asterisk (*) you see after SELECT acts as wild card character that allows you to select everything.

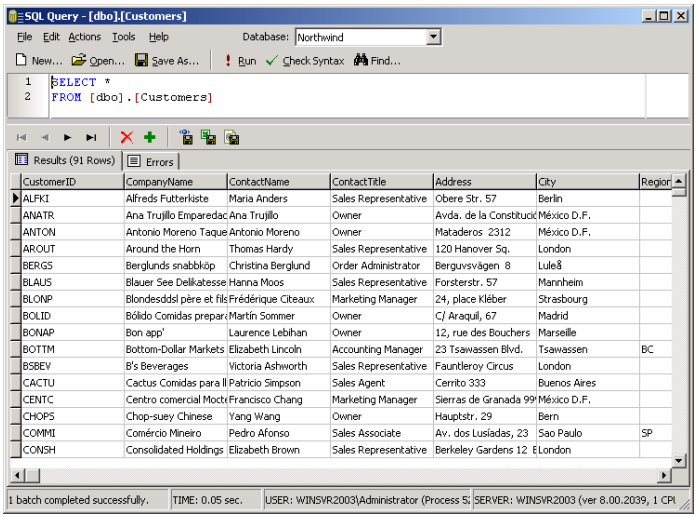

Figure 10 - Customers Table

If you had only wanted to look at the Company Name and the address of a particular customer using the Contact Name, you could run the query shown in Figure 11.

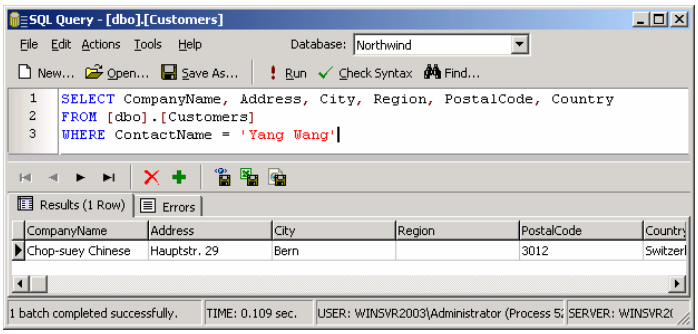

Figure 11 - Query with Where Condition

Assume that you want to count the number of orders from this customer, you could simply do a select statement to return all of the orders from the customer and count it manually, or you can use the COUNT method.

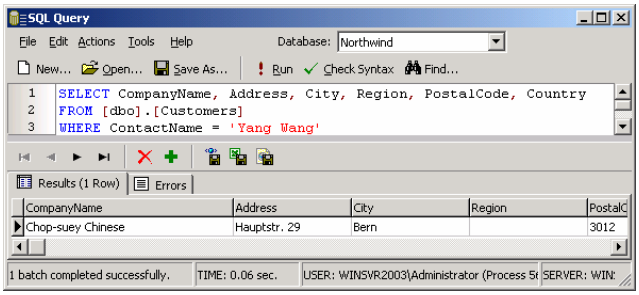

Figure 12 - COUNT Method in SQL

Imagine that now you need a report showing all of the orders from a particular company. This should be as easy as running a query in the Orders table. However, when you look into the table, you realized that the name of the Company is not in the Order table and the information is only in the Customers table. In this case, you will need to query both tables. Figure 13 shows the query and the result.

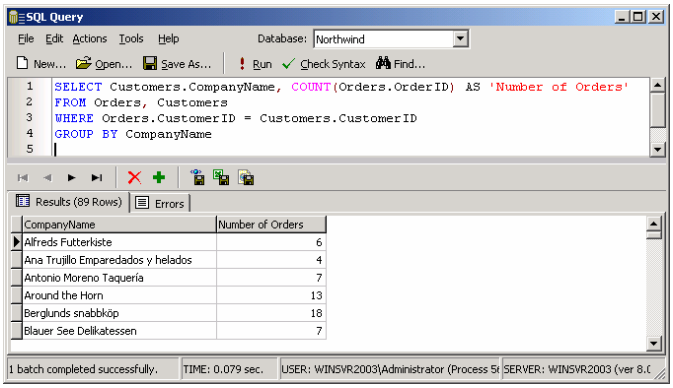

Figure 13 - Count Orders by Company

Your company is now doing a special and is going to give customers who have made more than 15 orders with a total price of over $500 a 10% discount. You can run a query to show the total number of orders with the sum of the orders they have ever made.

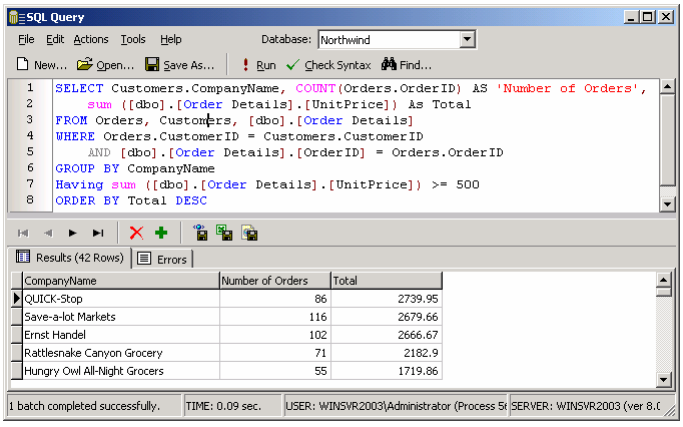

Figure 14 - Sum Method in SQL

You can take this equerry a step further and have it return the top 10 customers who paid the most for their orders. Figure 15 shows an example of how to do this.

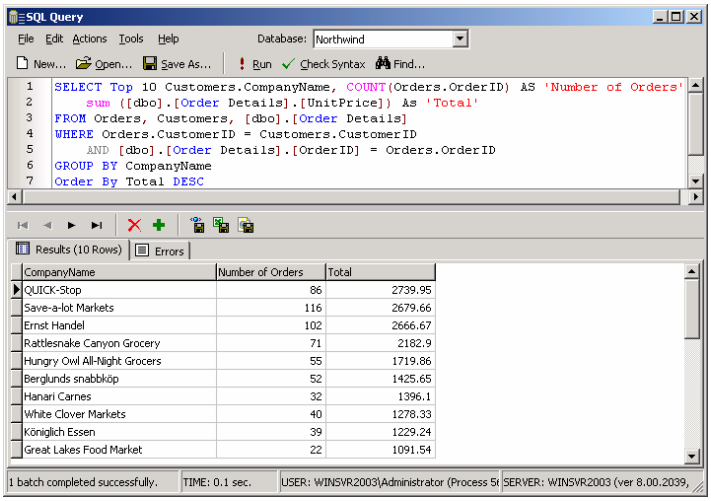

Figure 15 - TOP 10 Query

AlarmWorX32 SQL Commands

You can use the Alarm Logger to log alarm data to an SQL database. In this section, we will show you some useful SQL commands to aid you in turning your raw data into meaningful reports by using SQL queries. If you like to have advanced alarm analysis capability, consider using Alarm Analytics in ReportWorX.

Alarm Condition Frequency within a Specified Period
You may want to only look at alarms that occurred between a certain days. Of course, you can sort the table by date and then look at data between those days, but running a query will ensure that you will only look at data between the specified days.

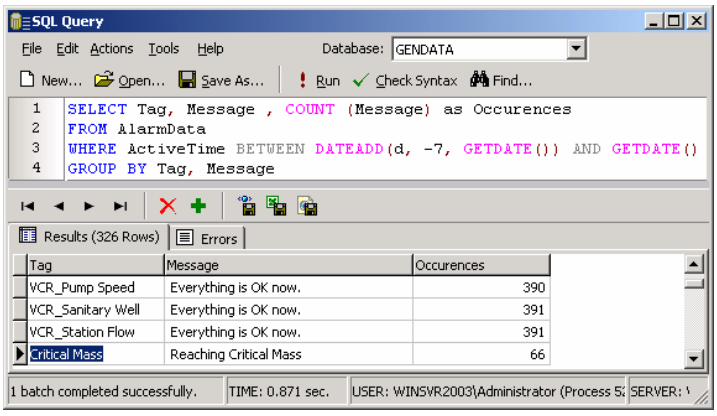

Figure 16 - GETDATE() and DATEADD() Commands

The code shown in Figure 16 counts the occurrences of each alarm in the AlarmData table between today and seven days ago. To get today's date, we used the GETDATE() function and we used the DATEADD() function to subtract seven days from today's date to return a weeks worth of data.

Calculating Downtime
Alarm records contain the time/date fields ActiveTime and EventTime. When an alarm occurs, both these fields are set to the time the alarm happened. When the alarm returns to normal, the EventTime gets updated, but the ActiveTime stays the same. We can use the DATEDIFF() function to calculate the differences between the two times for the total number of seconds the tag was in alarm state.

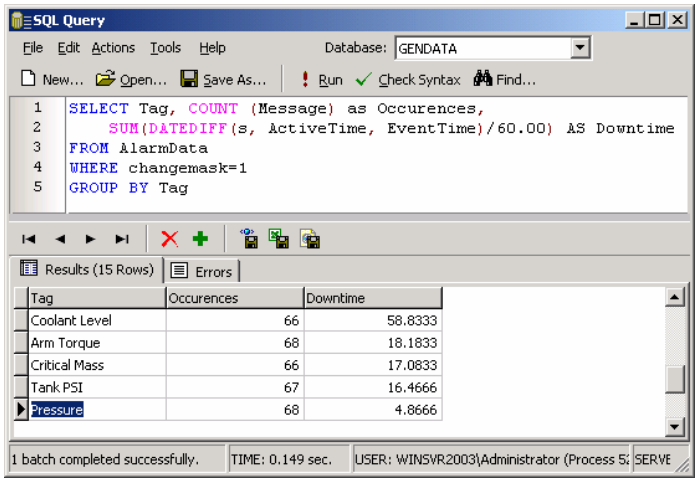

Figure 17 - DATEDIFF() and SUM()

The changemask field indicates the Normal State of the alarm tag when the value equals 1.

We can also add a HAVING clause to the above command to sort show alarms with only a downtime exceeding a certain number. The Syntax would be similar to Figure 14.

Advanced SQL Commands

You can create Views and Stored Procedures to save time and run queries that you need often. In this section, we will cover some basic information on views and stored procedures.

Creating Database Views
A view is a virtual table in the database which masks database complexity and simplifies the management of user permissions. The Syntax to create a view is in general

```
CREATE VIEW name_of_view AS
SELECT STATEMENT
```

Once you have a view, you can run the view by using

```
SELECT * FROM name_of_view
```

This eliminates the fact that you have to type in the same query, which could be complicated, every time you want to run it. This should be very helpful for queries that you need to run often.

When you no longer need a view, you can use the DROP VIEW name_of_view command to delete it.

Creating Stored Procedures
A stored procedure is a named collection of SQL statements that is stored on the database server. Stored procedures can accept input parameters that are used by the procedure internally. An advantage of stored procedures is that it reduces network traffic because it allows users to perform complex operations by sending only a single statement. A stored procedure is also useful if you want to grant a user access to procedure even if the user should not have direct access to certain tables.

The general syntax for Stored Procedures is similar to Views:

```
CREATE PROC my_stored_procedure_name
    @param1 paramType = ParamValue, @param2 paramType AS
SELECT STATEMEN
```

And when you want to run the procedure, you can use:

```
EXEC my_stored_procedure_name
    @param1 = paramValue, @param2 = paramValue
```