# chaper 2

## Introduction to Structured Query Language

# 6、Performing Calculations in SQL Queries

It is possible to perform certain types of calculations in SQL query statements. One group of calculations involves the use of SQL built-in functions. Another group involves simple arithmetic operations on the columns in the SELECT statement. We will consider each, in turn.

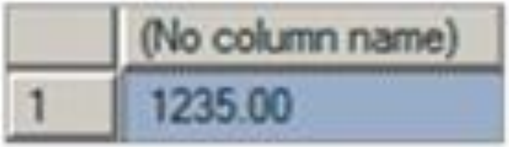
➢**Using SQL Built-in Functions**

There are five SQL built-in functions for performing arithmetic on table columns: SUM, AVG, MIN, MAX, and COUNT. Some DBMS products extend these standard built-in functions by providing additional functions. Here, we will focus only on the five standard SQL built-in functions.

Suppose we want to know the sum of OrderTotal for all of the orders in RETAIL_ORDER. We can obtain that sum by using the SQL built-in SUM function:

```
/*  ***  SQL-Query-CH02-26  ***            */
SELECT  SUM(OrderTotal)
FROM     RETAIL_ORDER;
```

The result will be:



|   | (No column name) |
|---|---|
| 1 | 1235.00 |

# 6、Performing Calculations in SQL Queries

➢**Using SQL Built-in Functions**

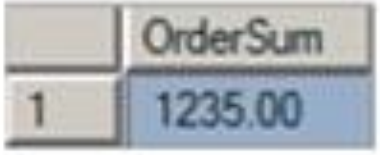
Recall that the result of an SQL statement is always a table. In this case, the table has one cell (the intersection of one row and one column that contains the sum of OrderTotal). But because the OrderTotal sum is not a column in a table, the DBMS has no column name to provide.

The preceding result was produced by Microsoft SQL Server 2008 R2, and it names the column ‘(No column name)’. Other DBMS products take other, equivalent actions.

This result is ugly. We would prefer to have a meaningful column name, and SQL allows us to assign one using the SQL AS keyword. If we use the AS keyword in the query as follow:

```
/*  ***  SQL-Query-CH02-27  ***      */
SELECTSUM(OrderTotal)  AS  OrderSum
FROM    RETAIL_ORDER;
```

The result of this modified query will be:



|   | OrderSum |
|---|---|
| 1 | 1235.00 |

# 6、Performing Calculations in SQL Queries

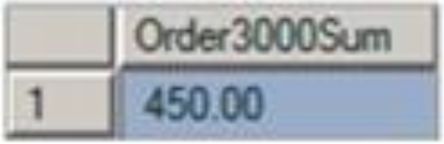
➢**Using SQL Built-in Functions**

This result has a much more meaningful column label. The name OrderSum is arbitrary— we are free to pick any name that we think would be meaningful to the user of the result. We could pick OrderTotal_Total, OrderTotalSum, or any other label that we think would be useful.

The utility of the built-in functions increases when you use them with an SQL WHERE clause. For example, we can write the SQL query:

```
/*  ***  SQL-Query-CH02-28  ***      */
SELECTSUM(ExtendedPrice)  AS  Order3000Sum
FROM    ORDER_ITEM
WHERE OrderNumber=3000;
```

The result of this query is:



|   | Order3000Sum |
|---|---|
| 1 | 450.00 |

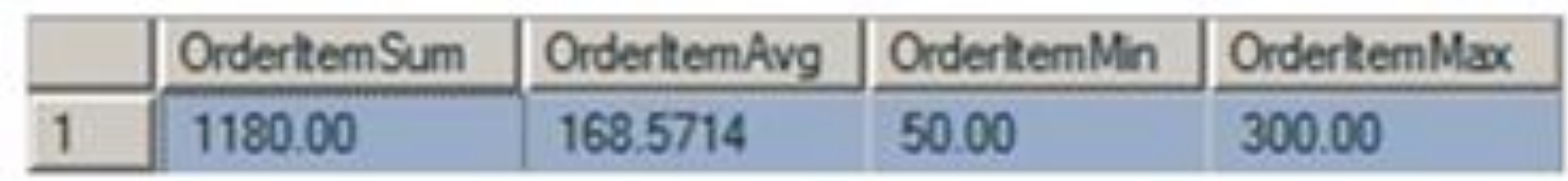
# 6、Performing Calculations in SQL Queries

➢**Using SQL Built-in Functions**

The SQL built-in functions can be mixed and matched in a single statement. For example, we can create the following SQL statement:

```
/*  ***  SQL-Query-CH02-29  ***      */
SELECTSUM(ExtendedPrice)  AS  OrderItemSum,
AVG(ExtendedPrice)  AS  OrderItemAvg,
MIN(ExtendedPrice)  AS  OrderItemMin,
MAX(ExtendedPrice)  AS  OrderItemMax
FROM    ORDER_ITEM;
```

The result of this query is:



| | OrderItemSum | OrderItemAvg | OrderItemMin | OrderItemMax |
|---|---|---|---|---|
| 1 | 1180.00 | 168.5714 | 50.00 | 300.00 |

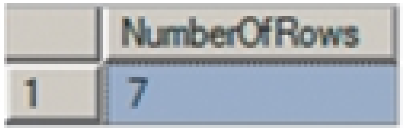
# 6、Performing Calculations in SQL Queries

- **Using SQL Built-in Functions**

The SQL built-in COUNT function sounds similar to the SUM function, but it produces very different results. The COUNT function counts the number of rows, whereas the SUM function adds the values in a column. For example, we can use the SQL built-in COUNT function to determine how many rows are in the ORDER_ITEM table:

```
/*  ***  SQL-Query-CH02-30  ***      */
SELECTCOUNT(*)  AS  NumberOfRows
FROM    ORDER_ITEM;
```

**The result of this query is:**



|   | NumberOfRows |
|---|---|
| 1 | 7 |

# 6、Performing Calculations in SQL Queries

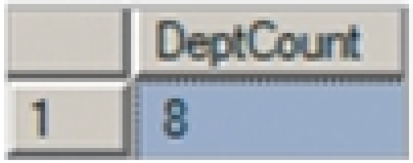
➢**Using SQL Built-in Functions**

This result indicates that there are seven rows in the ORDER_ITEM table. Notice that we need to provide an asterisk (*) after the COUNT function when we want to count rows. COUNT is the only built-in function that requires an asterisk. The COUNT function is also unique because it can be used on any type of data, but the SUM, AVG, MIN, and MAX functions can only be used with numeric data.

The COUNT function can produce some surprising results. For example, suppose you want to count the number of departments in the SKU_DATA table. If we use the following query:

```
/*  ***  SQL-Query-CH02-31  ***      */
SELECTCOUNT(Department)  AS  DeptCount
FROM    SKU_DATA;
```

**The result is:**



|   | DeptCount |
|---|---|
| 1 | 8 |

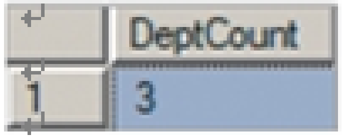
# 6、Performing Calculations in SQL Queries

➢**Using SQL Built-in Functions**

which is the number of rows in the SKU_DATA table, not the number of unique values of Department, as shown in Figure 2-4. If we want to count the unique values of Department, we need to use the SQL DISTINCT keyword, as follows:

```
/*  ***  SQL-Query-CH02-32  ***     */
SELECTCOUNT(DISTINCT  Department)  AS  DeptCount
FROM   SKU_DATA;
```

**The result of this query is:**



|   | DeptCount |
|---|---|
| 1 | 3 |

# 6、Performing Calculations in SQL Queries

Does Not Work
With MS Access
ANSI-89 SQL

Microsoft Access does not support the DISTINCT keyword as part of the COUNT expression, so although the SQL command with COUNT(Department) will work, the SQL command with COUNT(DISTINCT Department) will fail.

Solution: Use an SQL subquery structure (discussed later in this chapter) with the DISTINCT keyword in the subquery itself. This SQL query works:

```
/*  ***  SQL-Query-CH02-32-Access  *** */
SELECT  COUNT(*)  AS  DeptCount
FROM    (SELECT  DISTINCT  Department
FROM  SKU_DATA)  AS  DEPT;
```

Note that this query is a bit different from the other SQL queries using subqueries we show in this text because this subquery is in the FROM clause instead of (as you'll see) the WHERE clause. Basically, this subquery builds a new temporary table named DEPT containing only distinct Department values, and the query counts the number of those values.

# 6、Performing Calculations in SQL Queries

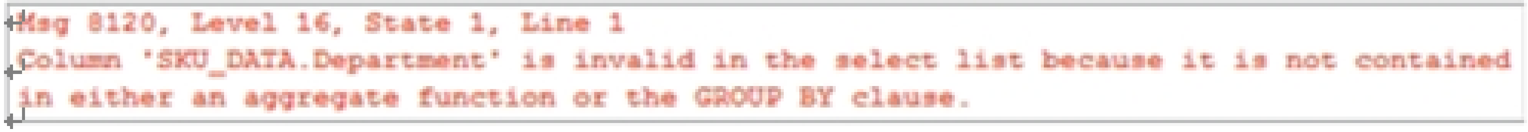
You should be aware of two limitations to SQL built-in functions. First, except for grouping (defined later), you cannot combine a table column name with an SQL built-in function. For example, what happens if we run the following SQL query?

**/* *** SQL-Qu H0 *** */**

**SELECT Departm UNT(*)**

**FROM SKU_DAT**

**The result in SQ 008 R2 is:**



```
Msg 8120, Level 16, State 1, Line 1
Column 'SKU_DATA.Department' is invalid in the select list because it is not contained
in either an aggregate function or the GROUP BY clause.
```

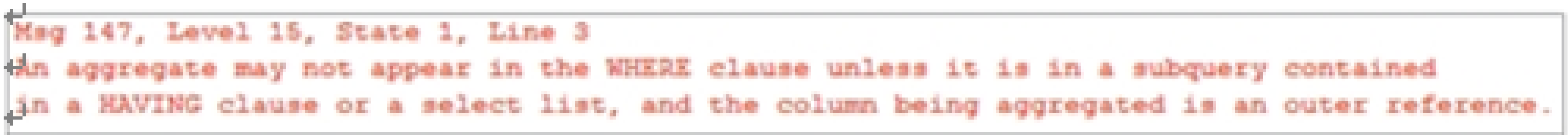
# 6、Performing Calculations in SQL Queries

This is the specific SQL Server 2008 R2 error message. However, you will receive an equivalent message from Microsoft Access, Oracle Database, DB2, or MySQL.

The second problem with the SQL built-in functions that you should understand is that you cannot use them in an SQL WHERE clause. Thus, you cannot use the following SQL statement:

```
/*  ***  SQL-Query-CH02-04  ***      */
SELECT*
FROM    RETAIL_ORDER
WHERE OrderTotal >  AVG(OrderTotal);
```

An attempt to use such a statement will also result in an error statement from the DBMS:



```
Msg 147, Level 15, State 1, Line 3
An aggregate may not appear in the WHERE clause unless it is in a subquery contained
in a HAVING clause or a select list, and the column being aggregated is an outer reference.
```

Again, this is the specific SQL Server 2008 error message, but other DBMS products will give you an equivalent error message. In Chapter 7, you will learn how to obtain the desired result of the above query using a sequence of SQL views.

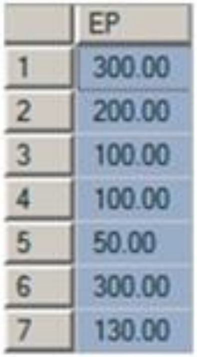
# 6、Performing Calculations in SQL Queries

## ➢SQL Expressions in SQL SELECT Statements

It is possible to do basic arithmetic in SQL statements. For example, suppose we want to compute the values of extended price, perhaps because we want to verify the accuracy of the data in the ORDER_ITEM table. To compute the extended price, we can use the SQL expression Quantity * Price in the SQL query:

```
/*  ***  SQL-Query-CH02-35  ***      */
SELECTQuantity  *  Price  AS  EP
FROM    ORDER_ITEM;
```

The result is:



| | EP |
|---|---|
| 1 | 300.00 |
| 2 | 200.00 |
| 3 | 100.00 |
| 4 | 100.00 |
| 5 | 50.00 |
| 6 | 300.00 |
| 7 | 130.00 |

# 6、Performing Calculations in SQL Queries

➢**SQL Expressions in SQL SELECT Statements**

An SQL expression is basically a formula or set of values that determines the exact results of an SQL query. We can think of an SQL expression as anything that follows an actual or implied equal to (=) character (or any other relational operator, such as greater than (>), less than (<), and so on) or that follows certain SQL keywords, such as LIKE and BETWEEN. Thus, the SELECT clause in the preceding query includes the implied equal to (=) sign as EP = Quantity * Price. For another example, in the WHERE clause:

```
WHERE          Buyer IN ('Nancy Meyers', 'Cindy Lo', 'Jerry Martin');
```

the SQL expression consists of the three text values following the IN keyword.

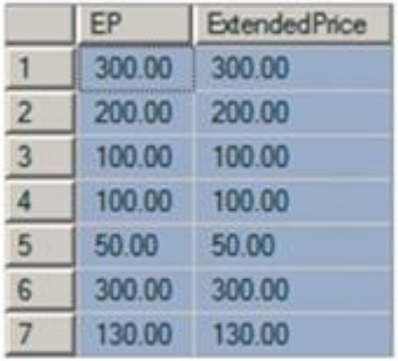
# 6、Performing Calculations in SQL Queries

## ➢SQL Expressions in SQL SELECT Statements

Now that we know how to calculate the value of extended price, we can compare this computed value to the stored value of ExtendedPrice by using the SQL query:

```
/*  ***  SQL-Query-CH02-36  ***      */
SELECTQuantity  *  Price  AS  EP,  ExtendedPrice
FROM    ORDER_ITEM;
```

The result of this statement now allows us to visually compare the two values to ensure that the stored data are correct:



|  | EP | ExtendedPrice |
|---|---|---|
| 1 | 300.00 | 300.00 |
| 2 | 200.00 | 200.00 |
| 3 | 100.00 | 100.00 |
| 4 | 100.00 | 100.00 |
| 5 | 50.00 | 50.00 |
| 6 | 300.00 | 300.00 |
| 7 | 130.00 | 130.00 |

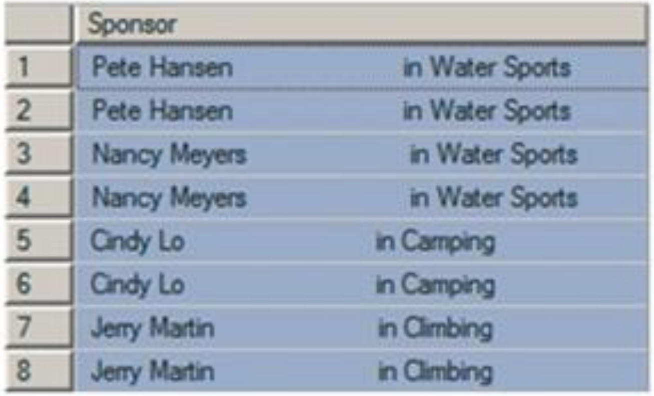
# 6、Performing Calculations in SQL Queries

➢ **SQL Expressions in SQL SELECT Statements**

Another use for SQL expressions in SQL statements is to perform string manipulation. Suppose we want to combine (using the concatenation operator, which is the plus sign [+] in SQL Server 2008 R2) the Buyer and Department columns into a single column named Sponsor. To do this, we can use the SQL statement:

```
/*  ***  SQL-Query-CH02-37  ***      */
SELECTBuyer+'  in  '+Department  AS  Sponsor
FROM   SKU_DATA;
```

The result will include a column named Sponsor that contains the combined text values:



|   | Sponsor |
|---|---|
| 1 | Pete Hansen            in Water Sports |
| 2 | Pete Hansen            in Water Sports |
| 3 | Nancy Meyers           in Water Sports |
| 4 | Nancy Meyers           in Water Sports |
| 5 | Cindy Lo               in Camping |
| 6 | Cindy Lo               in Camping |
| 7 | Jerry Martin           in Climbing |
| 8 | Jerry Martin           in Climbing |

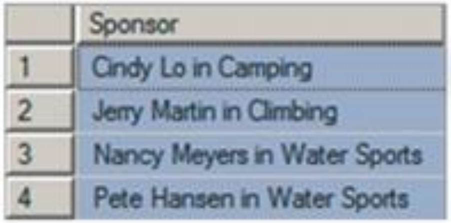
# 6、Performing Calculations in SQL Queries

## ➢ SQL Expressions in SQL SELECT Statements

The result of SQL-Query-CH02-37 is ugly because of the extra spaces in each row. We can eliminate these extra spaces by using more advanced functions. The syntax and use of such functions vary from one DBMS to another, however, and a discussion of the features of each product will take us away from the point of this discussion. To learn more, search on string functions in the documentation for your specific DBMS product. Just to illustrate the possibilities, however, here is an SQL Server 2008 R2 statement using the RTRIM function that strips the tailing blanks off the right-hand side of Buyer and Department:

```
/*  ***  SQL-Query-CH02-38  ***      */
SELECTDISTINCT  RTRIM(Buyer)+'  in  '+RTRIM(Department)  AS
Sponsor
FROM    SKU_DATA;
```

The result of this query is much more visually pleasing:



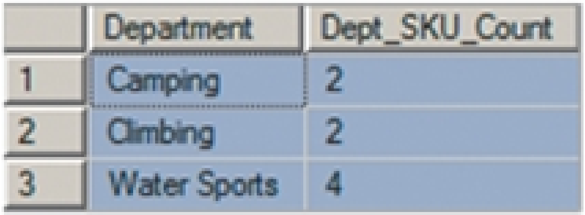
|   | Sponsor |
|---|---|
| 1 | Cindy Lo in Camping |
| 2 | Jerry Martin in Climbing |
| 3 | Nancy Meyers in Water Sports |
| 4 | Pete Hansen in Water Sports |

# 7、Grouping in SQL SELECT Statements

In SQL queries, rows can be grouped according to common values using the SQL GROUP BY clause. For example, if you specify GROUP BY Department in a SELECT statement on the SKU_DATA table, the DBMS will first sort all rows by Department and then combine all of the rows having the same value into a group for that department. A grouping will be formed for each unique value of Department. For example, we can use the GROUP BY clause in the SQL query:

```
/*  ***  SQL-Query-CH02-39  ***      */
SELECTDepartment,  COUNT(*)  AS  Dept_SKU_Count
FROM    SKU_DATA
GROUP  BY     Department;
```

We get the result:



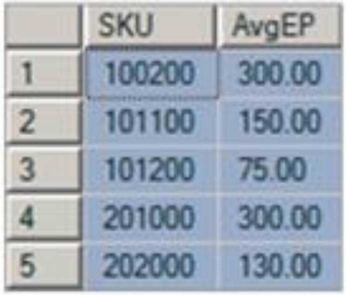
| | Department | Dept_SKU_Count |
|---|---|---|
| 1 | Camping | 2 |
| 2 | Climbing | 2 |
| 3 | Water Sports | 4 |

# 7、Grouping in SQL SELECT Statements

To obtain this result, the DBMS first sorts the rows according to Department and then counts the number of rows having the same value of Department. Here is another example of an SQL query using GROUP BY:

```
/*  ***  SQL-Query-CH02-40  ***      */
SELECTSKU,  AVG(ExtendedPrice)  AS  AvgEP
FROM    ORDER_ITEM
GROUP  BY    SKU;
```

The result for this query is:



|   | SKU | AvgEP |
|---|---|---|
| 1 | 100200 | 300.00 |
| 2 | 101100 | 150.00 |
| 3 | 101200 | 75.00 |
| 4 | 201000 | 300.00 |
| 5 | 202000 | 130.00 |

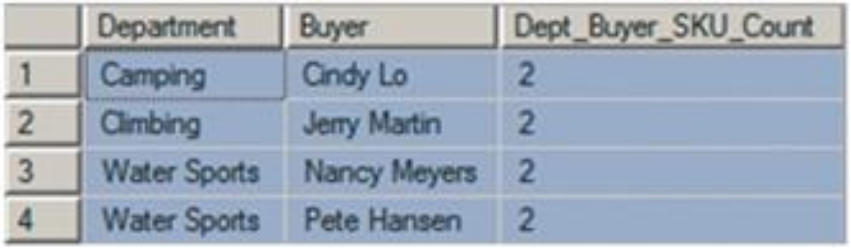
Here the rows have been sorted and grouped by SKU and the average ExtendedPrice for each group of SKU items has been calculated.

# 7、Grouping in SQL SELECT Statements

We can include more than one column in a GROUP BY expression. For example, the SQL statement:

```
/*  ***  SQL-Query-CH02-41  ***     */
SELECTDepartment,  Buyer,  COUNT(*)  AS  Dept_Buyer_SKU_Count
FROM    SKU_DATA
GROUP  BY     Department,  Buyer;
```

groups rows according to the value of Department first, then according to Buyer, and then counts the number of rows for each combination of Department and Buyer. The result is:



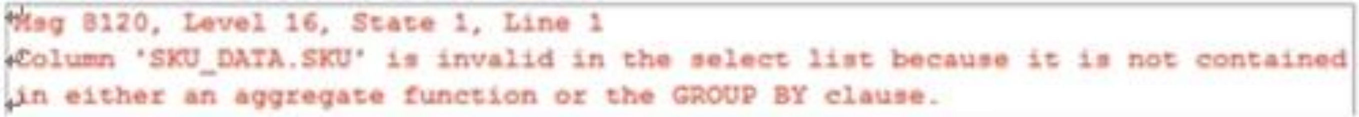
| | Department | Buyer | Dept_Buyer_SKU_Count |
|---|---|---|---|
| 1 | Camping | Cindy Lo | 2 |
| 2 | Climbing | Jerry Martin | 2 |
| 3 | Water Sports | Nancy Meyers | 2 |
| 4 | Water Sports | Pete Hansen | 2 |

# 7、Grouping in SQL SELECT Statements

When using the GROUP BY clause, only the column or columns in the GROUP BY expression and the SQL built-in functions can be used in the expressions in the SELECT clause. The following expressions will result in an error:

```
/*  ***  SQL-Query-CH02      *             */
SELECT  SKU,  Departm        COUNT(*)  AS  Dept_SKU_Count
FROM     SKU_DATA
GROUP  BY     Departm     ;
```

The resulting error message is:



```
Msg 8120, Level 16, State 1, Line 1
Column 'SKU_DATA.SKU' is invalid in the select list because it is not contained
in either an aggregate function or the GROUP BY clause.
```

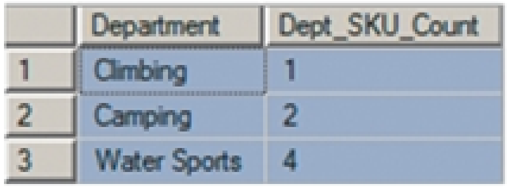
This is the specific SQL Server 2008 R2 error message, but other DBMS products will give you an equivalent error message. Statements like this one are invalid because there are many values of SKU for each Department group. The DBMS has no place to put those multiple values in the result. If you do not understand the problem, try to process this statement by hand. It cannot be done.

# 7、Grouping in SQL SELECT Statements

Of course, the SQL WHERE and ORDER BY clauses can also be used with SELECT statements, as shown in the following query:

```
 /*  ***  SQL-Query-CH02-43  ***            */
SELECT  Department,  COUNT(*)  AS  Dept_SKU_Count
FROM      SKU_DATA
WHERE   SKU  <>  302000
GROUP  BY     Department
ORDER  BY     Dept_SKU_Count;
```

The result is:



|   | Department | Dept_SKU_Count |
|---|---|---|
| 1 | Climbing | 1 |
| 2 | Camping | 2 |
| 3 | Water Sports | 4 |

Notice that one of the rows of the Climbing department has been removed from the count because it did not meet the WHERE clause condition. Without the ORDER BY clause, the rows would be presented in arbitrary order of Department. With it, the order is as shown. In general, to be safe, always place the WHERE clause before the GROUP BY clause. Some DBMS products do not require that placement, but others do.

# 7、Grouping in SQL SELECT Statements

Does Not Work With MS Access ANSI-89 SQL

Microsoft Access does not properly recognize the alias Dept_SKU_Count in the ORDER BY clause and creates a parameter query that requests an input value of as yet nonexistent Dept_SKU_Count! However, it doesn’t matter whether you enter parameter values or not—click the OK button and the query will run. The results will be basically correct, but they will not be sorted correctly.

Solution: Use the Microsoft Access QBE GUI to modify the query structure. The correct QBE structure is shown in Figure 2-21. The resulting Microsoft Access ANSI-89 SQL is:

```
/*  ***  SQL-Query-CH02-43-Access-A  ***              */
SELECT  SKU_DATA.Department,  Count(*)  AS  Dept_SKU_Count
FROM     SKU_DATA
WHERE   (((SKU_DATA.SKU)<>302000))
GROUP  BY    SKU_DATA.Department
ORDER  BY    Count(*);
```

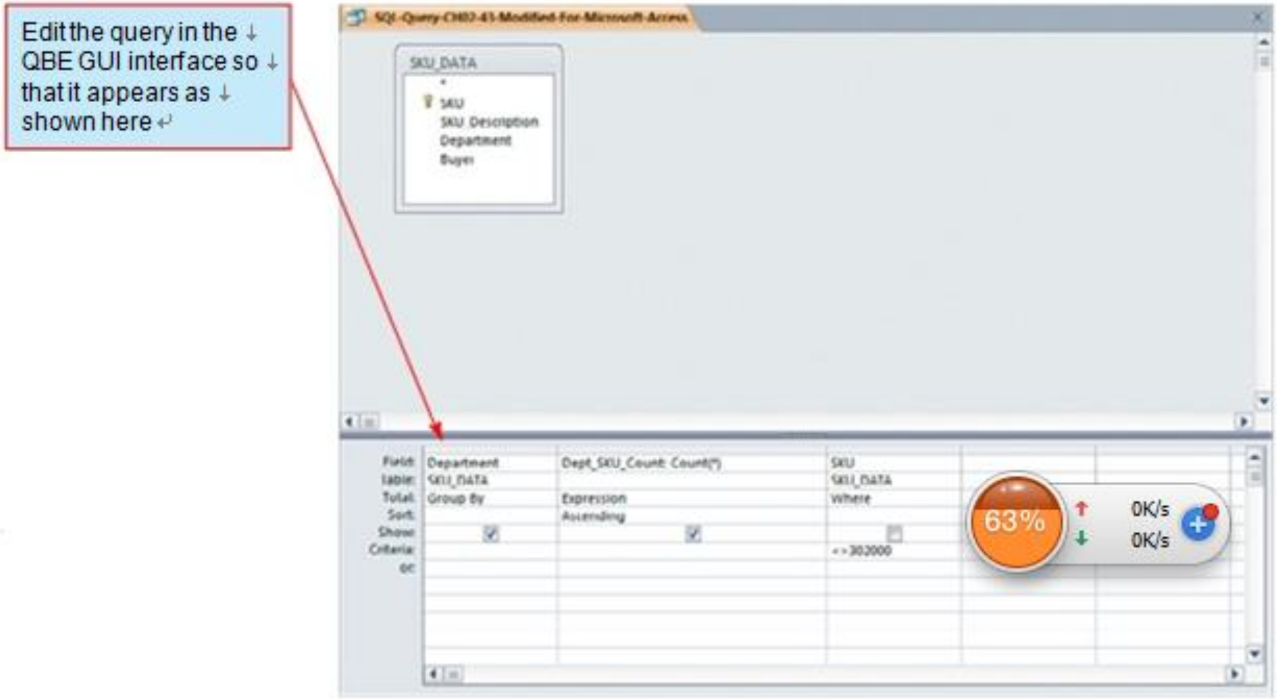
which can be edited down to:

```
/*  ***  SQL-Query-CH02-43-Access-B  ***              */
SELECT  Department,  Count(*)  AS  Dept_SKU_Count
FROM     SKU_DATA
WHERE   SKU<>302000
GROUP  BY    Department
ORDER  BY    Count(*);
```

# 7、Grouping in SQL SELECT Statements

Figure 2-21

Editing the SQL Query in the Access 2010 QBE GUI Interface



Edit the query in the QBE GUI interface so that it appears as shown here

SQL-Query-CH02-43-Modified-For-Microsoft-Access

SKU_DATA
- *
- SKU
- SKU_Description
- Department
- Buyer

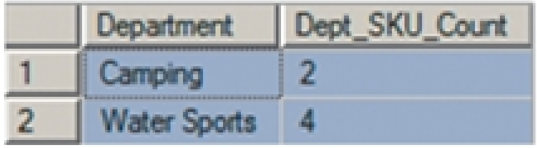
|  |  |  |  |
|---|---|---|---|
| Field: | Department | Dept_SKU_Count: Count(*) | SKU |
| Table: | SKU_DATA |  | SKU_DATA |
| Total: | Group By | Expression | Where |
| Sort: |  | Ascending |  |
| Show: | ☑ | ☑ | □ |
| Criteria: |  |  | <>302000 |
| or: |  |  |  |

# 7、Grouping in SQL SELECT Statements

SQL provides one more GROUP BY clause feature that extends its functionality even further. The SQL HAVING clause restricts the groups that are presented in the result. We can restrict the previous query to display only groups having more than one row by using the SQL query:

```
/*  ***  SQL-Query-CH02-44  ***      */
SELECTDepartment,  COUNT(*)  AS  Dept_SKU_Count
FROM    SKU_DATA
WHERE SKU  <>  302000
GROUP  BY     Department
HAVING COUNT  (*)  >  1
ORDER  BY     Dept_SKU_Count;
```

The result of this modified query is:



|   | Department | Dept_SKU_Count |
|---|---|---|
| 1 | Camping | 2 |
| 2 | Water Sports | 4 |

Comparing this result with the previous one, the row for Climbing (which has a count of 1) has been eliminated.

# 7、Grouping in SQL SELECT Statements

Does Not Work
With MS Access
ANSI-89 SQL

This query fails in Microsoft Access ANSI-89 SQL for the same reason as the previous query.

**Solution:** See the solution described in the previous "Does Not Work with Microsoft Access ANSI-89 SQL" box. The correct Microsoft Access ANSI-89 SQL for this query is:

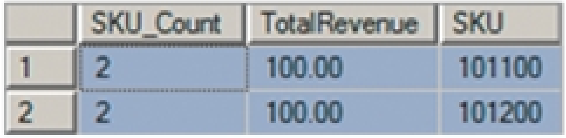
```
/* *** SQL-Query-CH02-44-Access *** */
SELECT      Department, Count(*) AS
Dept_SKU_Count
FROM  SKU_DATA
WHERE       SKU<>302000
GROUP  BY    Department
HAVING      Count(*)>1
ORDER  BY    Count(*);
```

# 7、Grouping in SQL SELECT Statements

SQL built-in functions can be used in the HAVING clause. For example, the following is a valid SQL query:

```
/*  ***  SQL-Query-CH02-45  ***      */
SELECTCOUNT(*)  AS  SKU_Count,  SUM(Price)  AS  TotalRevenue,  SKU
FROM    ORDER_ITEM
GROUP  BY     SKU
HAVING SUM(Price)=100;
```

**The results for this query are:**



|   | SKU_Count | TotalRevenue | SKU |
|---|---|---|---|
| 1 | 2 | 100.00 | 101100 |
| 2 | 2 | 100.00 | 101200 |

Be aware that there is an ambiguity in statements that include both WHERE and HAVING clauses. The results vary depending on whether the WHERE condition is applied before or after the HAVING. To eliminate this ambiguity, the WHERE clause is always applied before the HAVING clause.

LOGO

# Thank You!