CHAPTER 14

Obtaining Aggregate Data

About this chapter

This chapter describes how to construct queries that tell you aggregate information. Examples of aggregate information are as follows:

- ♦ The total of all values in a column
- ♦ The number of distinct entries in a column
- ♦ The average value of entries in a column

Contents

A first look at aggregate functions

Suppose you want to know how many employees there are. The following statement retrieves the number of rows in the **employee** table:

```
SELECT count( * )
FROM employee
```

count(*)
75

The result returned from this query is a table with only one column (with title **count(*)**) and one row, which contains the number of employees.

The following command is a slightly more complicated aggregate query:

```
SELECT   count( * ),
   min( birth_date ),
   max( birth_date )
FROM employee
```

count(*)	min(birth_date)	max(birth_date)
75	1936-01-02	1973-01-18

The result set from this query has three columns and only one row. The three columns contain the number of employees, the birthdate of the oldest employee, and the birthdate of the youngest employee.

COUNT, MIN and MAX are called **aggregate functions**. Each of these functions summarizes information for an entire table. In total, there are six aggregate functions: MIN, MAX, COUNT, AVG, SUM, and LIST. All but COUNT have the name of a column as a parameter. As you have seen, COUNT has an asterisk as its parameter.

Using aggregate functions to obtain grouped data

In addition to providing information about an entire table, aggregate functions can be used on groups of rows.

❖ **To list the number of orders each sales representative is responsible for:**

♦ Type the following:

```
SELECT sales_rep, count( * )
FROM sales_order
GROUP BY sales_rep
```

sales_rep	count(*)
129	57
195	50
299	114
467	56
667	54

The results of this query consist of one row for each sales rep ID number, containing the sales rep ID, and the number of rows in the **sales_order** table with that number.

Whenever GROUP BY is used, the resulting table has one row for each different value found in the GROUP BY column or columns.

A common error with GROUP BY

A common error with GROUP BY is to try to get information which cannot properly be put in a group. For example,

```
SELECT sales_rep, emp_lname, count( * )
FROM sales_order
   KEY JOIN employee
GROUP BY sales_rep
```

gives the following error:

Function or column reference to 'emp_lname' in the select list must also appear in a GROUP BY

SQL does not realize that each of the result rows for an employee with a given ID have the same value of **emp_lname**. An error is reported since SQL does not know which of the names to display.

However, the following is valid:

```
SELECT sales_rep, max( emp_lname ), count( * )
FROM sales_order
   KEY JOIN employee
GROUP BY sales_rep
```

The **max** function chooses the maximum (last alphabetically) surname from the detail rows for each group. The surname is the same on every detail row within a group so the **max** is just a trick to bypass a limitation of SQL.

Restricting groups

You have already seen how to restrict rows in a query using the WHERE clause. You can restrict GROUP BY clauses by using the HAVING keyword.

❖ **To list all sales reps with more than 55 orders:**

♦ Type the following:

```
SELECT sales_rep, count( * )
FROM  sales_order
KEY JOIN employee
GROUP BY sales_rep
HAVING count( * ) > 55
```

sales_rep	count(*)
129	57
299	114
467	56
1142	57

Order of clauses
GROUP BY must always appear before HAVING. In the same manner, WHERE must appear before GROUP BY.

HAVING clauses and WHERE clauses can be combined. When combining these clauses, the efficiency of the query can depend on whether criteria are placed in the HAVING clause or in the WHERE clause. Criteria in the HAVING clause restrict the rows of the result only after the groups have been constructed. Criteria in the WHERE clause are evaluated before the groups are constructed, and save time.

❖ **To list all sales reps with more than 55 orders and an ID of more than 1000:**

♦ Type the following:

```
SELECT sales_rep, count( * )
FROM sales_order
KEY JOIN employee
WHERE sales_rep > 1000
GROUP BY sales_rep
HAVING count( * ) > 55
```

The following statement produces the same results.

❖ **To list all sales reps with more than 55 orders and an ID of more than 1000:**

♦ Type the following:

```
SELECT sales_rep, count( * )
FROM sales_order
KEY JOIN employee
GROUP BY sales_rep
HAVING count( * ) > 55
AND sales_rep > 1000
```

The first statement is faster because it can eliminate making up groups for some of the employees. The second statement builds a group for each sales rep and then eliminates the groups with wrong employee numbers. For example, in the first statement, the database engine would not have to make up a group for the sales rep with employee ID 129. In the second command, the database engine would make up a group for employee 129 and eliminate that group with the HAVING clause.

Fortunately, Adaptive Server Anywhere detects this particular problem and changes the second query to be the same as the first. Adaptive Server Anywhere performs this optimization with simple conditions (nothing involving OR or IN). For this reason, when constructing queries with both a WHERE clause and a HAVING clause, you should be careful to put as many of the conditions as possible in the WHERE clause.