

3

Aggregate Calculations and Subqueries

Chapter Overview

So far we have examined all the basic ways to query information from a single table, but there are many more powerful query tools in SQL. In this chapter we will examine two more. One uses aggregate functions to assemble rows of data into totals, counts, and other calculations. The other sets a query inside a query. This is called a subquery, and it provides tremendous extensions to the power of SQL.

Chapter Objectives

In this chapter, we will:

- Learn what aggregate functions are
- Write SQL queries to summarize data into aggregate calculations
- Learn what a subquery is and where it can be used in SQL
- Learn how to use subqueries in the WHERE clause
- Use the ANY and ALL keywords with subqueries

Aggregate Functions

We have already seen how to create calculated columns in a query. *Aggregates* are also calculations, but in a very different way. A calculated column calculates based on the values of a single

row at a time. An aggregate calculation summarizes values from entire groups of rows. The word *aggregate* is one we don't often use in everyday speech, but it simply means a summary calculation, such as a total or average. The standard aggregate functions are:

Standard Aggregate Functions	
Sum	To calculate totals
Avg	To calculate averages
Count	To count the number of records
Min	To report the minimum value
Max	To report the maximum value

Some database systems add other aggregates. For instance, Access adds standard deviation, variance, first, and last. But these are rarely used. A recent newsgroup search for Access's StDev function yielded just 288 messages compared to 132,000 messages for Sum, for example. We will confine our discussion to the standard aggregate functions above. Let's see how to use them.

Basic Aggregate Functions	
Access, SQL Server, Oracle, MySQL	
Syntax	SELECT Aggregate (Field Expression) AS ColumnName FROM Table
Examples (All)	1. Report the total time in seconds of all tracks. <pre>Select Sum(lengthseconds) From Tracks</pre>
	2. Report the number of members in the members table. <pre>Select Count(*) As NumMembers From Members</pre>
	3. Report the average length in minutes of the tracks for TitleID 1. <pre>Select Avg(lengthseconds)/60 From Tracks Where TitleID = 1</pre>
	4. Report the shortest and longest track lengths in seconds. <pre>Select Min(lengthseconds) As Shortest, Max(lengthseconds) As Longest From Tracks</pre>

As you see from the above examples, the function is typed with the field or expression in parentheses. You can follow the function with the keyword AS and an alias column name after the function. In many database systems an alias is optional, but it is always good practice.

We also see that the aggregate can be used as just part of an expression (Example 2), that a WHERE clause can limit the number of rows being aggregated (Example 2), and that multiple aggregate functions can be included together (Example 3).

Sum

The *SUM* aggregate function is pretty obvious. It adds up a column. Normally you will want to sum up a numeric column as shown above in example 1. Any null values in the column of numbers are ignored, which essentially treats them as zeros.

Can it sum up anything else? It depends on the database. If you try to sum up a text column, Access, SQL Server, and Oracle will give you an error while MySQL will return a zero. If you try to sum up a column of dates, Access and MySQL will return the sum of the dates' numeric equivalents, though that probably has no relevance in the real world. Summing a date column will return an error in SQL Server and Oracle. You can sum a column of Boolean (True/False) values in Access, MySQL, and Oracle. Since true values are stored as 1 (or -1 in Access) while false values are stored as 0, summing these values will, in effect, count of the number of true values. Summing a Boolean bit value in SQL Server will yield an error.

Avg

The *AVG* aggregate function will return an average of the values. This also is generally done on numbers. The same rules apply to averaging other datatypes as with summing the datatypes discussed above. It is worth noting that AVG will not treat null values as zero. It will completely ignore nulls. For instance, consider the data below from the Salesperson table:

```
Select Base
From Salesperson
```

```
Base
-----
100.0000
300.0000
100.0000
NULL
```

```
(4 row(s) affected)
```

Now consider an average of this data.

```
Select Avg(Base)
From Salespeople
```

```
-----
166.6666
```

```
(1 row(s) affected)
```

If you do the math, you will see that it is the sum of the numbers (500) divided by 3, not divided by 4. The null has been left out of the average entirely. You will get this same result with each of our four target databases. What if you want to count the null value as a zero? You can do that by applying a CASE statement as shown below. This works in SQL Server, Oracle, and MySQL. Access does not support the CASE statement. However, if you need to do this in Access, you could use the Access-specific IIF function documented in Chapter 5.

```
Select
Avg(Case
  When Base is Null Then 0
  Else Base
End)
From Salespeople
```

```
-----
125.0000
```

```
(1 row(s) affected)
```

Note: SQL Server

If you run Example 3 from the ‘Basic Aggregate Functions’ table above (Select Avg(lengthseconds)/60 from Tracks Where TitleID=1) against SQL Server, it returns a whole number. This is because LengthSeconds is stored in a field with a small integer datatype, and all calculations from that field maintain that datatype. If you want to see the fractional minutes, you need to use an SQL Server function to cast or convert the calculated column to a different datatype. These will be explained in Chapter 5. Access, MySQL, and Oracle will return numbers to the right of the decimal place, even when averaging an integer value.

Count

The *COUNT* aggregate function simply counts the resulting values or rows. Without a *WHERE* clause, *COUNT* counts all the rows in the table. If you add a *WHERE* clause it will count the rows that are returned. You can use *COUNT* on any datatype.

The *COUNT* aggregate function can take as an argument either a field name or an asterisk (*). Using an asterisk will simply count the rows. Counting a field will count the number of non-null values in that field. The following two SQL statements illustrate this:

```
Select Count(*) As NumArtists
From Artists
```

```
NumArtists
-----
11
```

```
Select Count(WebAddress) As NumArtistsWithWebPage
From Artists
```

```
NumArtistsWithWebPage
-----
6
```

Tip

If you want to count all the rows in a query and not just those with non-null values, either use *COUNT*(*) or count the primary key column. The primary key cannot be null.

Min and Max

The *MIN* and *MAX* aggregate functions report the minimum and maximum values. In addition to being used with numeric datatypes, they can be also used with dates to report the earliest and latest dates and with text to report the lowest and highest alphabetically.

```
Select Min(Lastname) As Lowest, Max(Lastname) as Highest
From Members
```

```
Lowest                                Highest
-----                                -----
Alvarez                                Wong
```

Notice in the SQL results above that unlike the other aggregates that return summary statistics, MIN and MAX return raw field values. Alvarez and Wong come from two separate rows in the table, but they are reported together in one row because one is the minimum and one is the maximum.

```
Select Min(Birthday) As Oldest, Max(Birthday) as Youngest
From Members
```

Oldest	Youngest
-----	-----
1955-11-01	1983-09-02

From these results we can identify the birth dates of the youngest and oldest members. We cannot identify who those people are. To do that we will need to use a subquery, as we will see later in this chapter.

Group By

In all the examples above, the columns in a query are aggregate functions. When that is the case the query will report just one row showing the sum, average, count, minimum, or maximum for the entire set of records selected for the query. If additional non-aggregate fields are included in the SELECT clause, the query will report one row for each combination of the non-aggregate fields with the sum, average, etc. for each of those combinations. In other words, the query will group the rows by the non-aggregate fields and calculate the aggregate functions for each group.

When you include non-aggregate fields with aggregates, you must include a GROUP BY clause listing the non-aggregate fields. The more fields in the GROUP BY clause, the more rows will be reported. One row will be reported for each combination of the non-aggregate fields. The GROUP BY clause must come after the WHERE clause. The proper order for all the clauses we have learned so far is:

```
SELECT column, Aggregate(column | expression) As column_name
FROM tablename
WHERE condition
GROUP BY column
ORDER BY column
```

Group By	
Access, SQL Server, Oracle, MySQL	
Syntax	<pre>SELECT Field, Aggregate(Expression) AS Column_Name FROM Table Where Field Expression comparison Value Field Group By Field</pre>
Examples	<p>1. Report the total time in seconds for each title.</p> <pre>Select TitleID, Sum(lengthseconds) From Tracks Group By TitleID</pre>
	<p>2. Report the number of members in each state.</p> <pre>Select Region, Count(*) As NumMembers From Members Group by Region</pre>
	<p>3. Report the number of members by state and gender.</p> <pre>Select Region, Gender, Count(*) AS NumMembers From Members Group by Region, Gender</pre>
	<p>4. Report the shortest and longest track lengths in seconds for each title.</p> <pre>Select TitleID, Min(lengthseconds) As Shortest, Max(lengthseconds) As Longest From Tracks Group By TitleID</pre>

Note

In each of the above examples that the fields listed in the GROUP BY clause are all the non-aggregate fields in the SELECT clause. It must be done this way or the SQL statement will not run. They do not have to be listed in the same order, but they all must be there.

Having and Where with Aggregates

We have already seen that aggregate functions can use WHERE clauses. *HAVING* is similar to WHERE. But while WHERE restricts the results based on individual row values, HAVING

restricts the results based on aggregated values. Another way of saying this is that WHERE can eliminate records from the results before the aggregates are calculated while HAVING eliminates entire groups of records from the results based on the aggregated calculations. Because HAVING works on aggregated rows, it always uses an aggregate function as its test.

The HAVING clause must come after the GROUP BY clause and before the ORDER BY clause. The proper order for SQL clauses is:

```
SELECT column, Aggregate(column | expression) As column_name
FROM tablename
WHERE condition
GROUP BY column
HAVING condition
ORDER BY column
```

Having	
Access, SQL Server, Oracle, MySQL	
Syntax	<pre>SELECT Field, Aggregate(Expression) AS Column_Name FROM Table Where Field Expression comparison Value Field Group By Field Having Aggregate(Expression) comparison Value</pre>
Examples	<p>1. Report the total time in minutes for any title whose total length in more than 40 minutes.</p> <pre>Select TitleID, Sum(lengthseconds)/60 As TotMin From Tracks Group By TitleID Having Sum(lengthseconds)/60>40</pre>

Let's explore the differences in how WHERE and HAVING work.

```
Select TitleID, Avg(lengthseconds) As AvgLength
From Tracks
Group by TitleID
```

TitleID	AvgLength
1	279
3	212

4	221
5	231
6	532
7	309

This first SQL statement reports the average length in seconds of the titles on each track.

```
Select TitleID, Avg(lengthseconds) As AvgLength
From Tracks
Where lengthseconds>240
Group by TitleID
```

TitleID	AvgLength
1	327
3	294
4	352
5	282
6	532
7	325

Now we add a WHERE clause. This eliminates all records with a length of 240 seconds (4 minutes) or less, so these shorter tracks are not even included in the calculated average. Thus the reported averages are higher. Since the numbers changed for all titles, we can assume that all of them had tracks of 240 seconds or less.

```
Select TitleID, Avg(lengthseconds) As AvgLength
From Tracks
Group by TitleID
Having Avg(lengthseconds)>240
```

TitleID	AvgLength
1	279
6	532
7	309

In the last SQL statement we change the WHERE clause to a HAVING clause. We also change the test from testing for rows that are greater than 240 seconds to testing for aggregated averages greater than 240 seconds. Notice that the reported average lengths return to what they

were in our first SQL statement. That is because we are no longer eliminating any records from the calculation. However, the HAVING clause eliminates some of the aggregated rows from the final results.

Note

You might be tempted to write the above SQL using the column alias in the HAVING clause as shown below:

```
Select TitleID, Avg(lengthseconds) As AvgLength
From Tracks
Group by TitleID
Having AvgLength>240
```

That will work in MySQL. But in Access, Oracle, and SQL Server the column alias from the SELECT clause has no meaning in the HAVING clause. So except in MySQL, you must repeat the aggregate function in the HAVING clause.

Subqueries in the Where Clause

Queries can be used within queries. These are called *subqueries*, and they can be used in the WHERE clause returning a value or list of values to test against, in the SELECT clause returning a single value to use as a column, or in the FROM clause as if it were a table. We will save FROM clause subqueries for Chapter 4 and SELECT clause subqueries for Chapter 6.

A subquery is actually a query that could be run independently. It is placed inside parentheses when used inside another query. Subqueries are supported to varying degrees in different database systems. MySQL versions 4.1 and higher support sub-queries in the WHERE and FROM clauses. Access 2000 and higher supports WHERE clause sub-queries and FROM clause sub-queries.

Where Clause Subqueries	
Access, SQL Server, Oracle, MySQL	
Syntax	<pre>SELECT Field Field, Field, Field * FROM Table WHERE Field Expression comparison (Select Field FROM Table)</pre>
Examples	<p>1. List the name of the oldest member.</p> <pre>Select Lastname, Firstname From Members Where Birthday = (Select Min(Birthday) From Members)</pre>
	<p>2. List all track titles and lengths of all tracks whose length is longer than the average of all track lengths.</p> <pre>Select Tracktitle, Lengthseconds From Tracks Where Lengthseconds > (Select Avg(Lengthseconds) From Tracks)</pre>
	<p>3. List the names of all artists who have recorded a title.</p> <pre>Select Artistname From Artists Where ArtistID IN (Select ArtistID From Titles)</pre>

Let's play with each of the above examples. Example 1 uses a subquery similar to a query we saw earlier in the chapter. It reports the minimum (or earliest) birthday from the Members table. If we ran the subquery separately we would get:

```
Select Min(Birthday)
From Members
```

```
-----
1955-11-01
```

This tells us the earliest birthday, but it does not tell us whose birthday it is. Further, there is no way to get that additional information from a simple query using an aggregate function. If we added first name and last name to the above query, we would also have to GROUP BY those non-aggregate fields. That would then return one row for each person rather than just a single row reporting the earliest birthday. To find the name of the oldest member we have to use a

subquery. The outer query in Example 1 lists the first and last name but adds a WHERE clause to report only the member whose birthday is the date returned by the subquery. In other words, the query matches the birthdays of all members to the birthday of the oldest member and reports only the match. What if two people were born on November 1, 1955? In that case, both would be reported. But that would be okay because both would equally be oldest.

```
Select Lastname, Firstname
From Members
Where Birthday = (Select Min(Birthday)
                  From Members)
```

Lastname	Firstname
-----	-----
Wong	Tony

Example 2 is very similar except that it uses greater than instead of equal. The subquery calculates the average length of all tracks. The outer query reports all tracks whose length exceeds that average.

```
Select Tracktitle, Lengthseconds
From Tracks
Where Lengthseconds >(Select Avg(Lengthseconds)
                       From Tracks)
```

Tracktitle	Lengthseconds
Third's Folly	352
Fat Cheeks	352
Goodtime March	293
TV Day	305
Call Me an Idiot	315
25	402
Palm	322
Rocky and Natasha	283
Violin Sonata No. 1 in D Major	511
Violin Sonata No. 2 in A Major	438
Violin Sonata No. 4 in E Minor	821
Piano Sonata No. 1	493
Clarinet Sonata in E Flat	399
Song 1	285
Song 3	299
Song 7	303

What's the Day	332
Sirius	287
Hamburger Blues	292
Road Trip	314
Meeting You	321
Improv 34	441
Hey	288
Wooden Man	314

Example three uses the IN keyword. In Chapter 2 we saw that IN could evaluate each row against a list of possible values. In this example the subquery provides the possible values. The subquery (`Select ArtistID from Titles`) lists all the ArtistIDs in the Titles table. The outer query then reports the names of all artists in that list.

```
Select Artistname
From Artists
Where ArtistID IN(Select ArtistID
                  From Titles)
```

Artistname

```
-----
The Neurotics
Louis Holiday
Sonata
The Bullets
Confused
```

We can easily turn this last example around to report Artists who haven't recorded a title. This is a very useful way to find unmatched rows between two tables. However, we will see another way to do this using JOIN in Chapter 4, and that way is often faster.

```
Select Artistname
From Artists
Where ArtistID NOT IN(Select ArtistID
                      From Titles)
```

Artistname

```
-----
Word
Jose MacArthur
The Kicks
Today
```

21 West Elm
Highlander

All & Any

ALL and *ANY* are the WHERE clause keywords used most often with subqueries. They could be used with a simple list of values, but you would be hard pressed to think of an example when you would want to. *ALL* and *ANY* are used like *IN* to compare a value to a list of values in a subquery. But while *IN* is essentially an equal to comparison, *ALL* and *ANY* can be used with less than or greater than comparisons. If the subquery is preceded by *ANY*, the comparison will be true if it satisfies any value produced by the subquery. If the subquery is preceded by *ALL*, the comparison will be true if it satisfies all values produced by the subquery.

Note

SQL also has a keyword *SOME*. It is functionally equivalent to *ANY*. Feel free to test the *ANY* example below using *SOME*.

Any & All	
Access, SQL Server, Oracle, MySQL	
Syntax	<pre>SELECT Field Field, Field, Field * FROM Table Where Field Expression comparison Any All (Select Field FROM Table)</pre>
Examples	<p>1. List the name, region, and birthday of every member who is older than all of the members in Georgia (GA).</p> <pre>Select Lastname, Firstname, Region, Birthday From Members Where Birthday < ALL(Select Birthday From Members Where Region = 'GA')</pre>
	<p>2. List the name, region, and birthday of every member who is older than any of the members in Georgia (GA).</p> <pre>Select Lastname, Firstname, Region, Birthday From Members Where Birthday < ANY(Select Birthday From Members Where Region = 'GA')</pre>

Let's begin by looking at the subquery by itself:

```
Select Birthday
From Members
Where Region = 'GA'
```

```
Birthday
-----
1963-08-04
1959-06-22
1964-03-15
```

This gives us a list of three birthdays. Now in Example 1 we want to look at all members (not just those from Georgia) and report anyone who is older than all of these three. In effect, that means anyone who is older than the oldest of these three, the person born in 1959.

```
Select Lastname, Firstname, Region, Birthday
From Members
Where Birthday < ALL(Select Birthday
                    From Members
                    Where Region='GA')
```

Lastname	Firstname	Region	Birthday
-----	-----	-----	-----
Ranier	Brian	ONT	1957-10-19
Kale	Caroline	VA	1956-05-30
Wong	Tony	ONT	1955-11-01
Cleaver	Vic	VT	1957-02-10

By definition, no member from Georgia is older than the members from Georgia, so we don't see any Georgia members on the list. We only see people born prior to June 22, 1959. Now if we change the ALL to ANY we are looking for people older than anyone on our subquery list. In effect, that means anyone who is older than the youngest of these three, the person born in 1964.

```
Select Lastname, Firstname, Region, Birthday
From Members
Where Birthday < ANY(Select Birthday
                    From Members
                    Where Region='GA')
```

Lastname	Firstname	Region	Birthday
Finney	Doug	GA	1963-08-04
Irving	Terry	GA	1959-06-22
Payne	Frank	NY	1960-01-17
Ranier	Brian	ONT	1957-10-19
Lambert	Marcellin	VA	1959-11-14
Kale	Caroline	VA	1956-05-30
Fernandez	Kerry	VA	1962-01-16
Wong	Tony	ONT	1955-11-01
Taft	Bonnie	VT	1960-09-21
Cleaver	Vic	VT	1957-02-10

Chapter Summary

In addition to selecting data from a table, SQL can perform aggregate functions that calculate totals, averages, and other summary information. The standard aggregate functions are SUM (totals), AVG (averages), COUNT (count), MIN (minimum), and MAX (maximum). If the SELECT clause is made up entirely of aggregate functions, only one row will be returned reporting summary statistics for the entire table. If non-aggregate columns are included in the SELECT clause, and if a GROUP BY clause is included listing each non-aggregate column, then the query will report summary statistics for each combination of the non-aggregate column values. Queries with aggregates can use a WHERE clause to select the rows that will be included in the aggregated numbers. You can also use a HAVING clause, which will select aggregated values to be included in the final results.

Another powerful feature of SQL is its ability to place a query inside a query. These subqueries can be used in the WHERE clause, the SELECT clause, and the FROM clause. This chapter examined the use of subqueries in the WHERE clause. The chapter also discussed the ALL and ANY keywords that are often used with subqueries.

Key Terms

aggregate	COUNT	SOME
ALL	HAVING	subquery
ANY	MAX	SUM
AVG	MIN	

Review Questions

1. What is an aggregate function used for in SQL?
2. What is the purpose of the HAVING SQL keyword?
3. What is the difference between COUNT(*) and COUNT(field)?
4. How does the AVG aggregate treat null values?
5. What is the proper order of the SQL keywords FROM, GROUP BY, HAVING, ORDER BY, SELECT, and WHERE?
6. What is a subquery?
7. Name the three SQL clauses in which a subquery can appear.
8. To what extent are subqueries supported in Access, SQL Server, Oracle, and MySQL?
9. If two queries with subqueries were identical except that one used > ANY and the other used > ALL, which one should return more rows? Why?
10. What is the difference between ANY and SOME?

Exercises

Using any SQL tool, write SQL commands to do the following. Use an alias for every aggregated and calculated column. Note that MySQL cannot be used for subqueries.

1. Report the number of tracks for each TitleID.
2. Report the TitleID and number of tracks for any TitleID with fewer than nine tracks.
3. For each kind of LeadSource, report the number of artists who came into the system through that lead source, the earliest EntryDate, and the most recent EntryDate.
4. Report the last name of the member who would be reported first in alphabetical order.
5. List the number of track titles that begin with the letter S and the average length of these tracks in seconds.
6. List the track titles of all titles in the 'alternative' genre.
7. List the length of the longest RealAud track in the "metal" genre.
8. For any region that has more than one member with an e-mail address, list the region and the number of members with an e-mail address.
9. List all genres from the Genre table that are not represented in the Titles tables.
10. List track titles and lengths of tracks with a length longer than all tracks of the "metal" genre.
Hint: This requires a subquery within a subquery.

Additional References

W3Schools.com –
SQL Functions

http://www.w3schools.com/sql/sql_functions.asp

MySQL Reference Manual

<http://www.mysql.com/documentation/mysql/bychapter/>