2.4 Aggregating

2.4.1 Why Aggregate?

Decision makers are often overwhelmed with details. Consider a retail example, showing all sales:

<table>
<thead>
<tr>
<th>Date</th>
<th>Time</th>
<th>OrderID</th>
<th>CustomerID</th>
<th>EmployeeID</th>
<th>ItemNumber</th>
<th>Quantity</th>
<th>UnitPrice</th>
</tr>
</thead>
<tbody>
<tr>
<td>2/10/2008</td>
<td>11:52AM</td>
<td>22345</td>
<td>8934</td>
<td>112</td>
<td>XRGW1232</td>
<td>2</td>
<td>4.95</td>
</tr>
<tr>
<td>2/10/2008</td>
<td>11:52AM</td>
<td>22345</td>
<td>8934</td>
<td>112</td>
<td>AJKD1115</td>
<td>1</td>
<td>9.95</td>
</tr>
<tr>
<td>2/10/2008</td>
<td>11:52AM</td>
<td>22345</td>
<td>8934</td>
<td>112</td>
<td>JKL8234</td>
<td>10</td>
<td>1.95</td>
</tr>
<tr>
<td>2/10/2008</td>
<td>11:53AM</td>
<td>22346</td>
<td>1389</td>
<td>098</td>
<td>RRD3982</td>
<td>1</td>
<td>15.50</td>
</tr>
<tr>
<td>2/10/2008</td>
<td>11:54AM</td>
<td>22347</td>
<td>4533</td>
<td>112</td>
<td>JDK2289</td>
<td>3</td>
<td>8.95</td>
</tr>
</tbody>
</table>

This is classic Online Transaction Processing (OLTP) data. Every detail of every event in the retail sales process is captured. However, presenting all this data is can be overwhelming. Consider these questions a manager might ask:

- What was the total of all sales today?
- What were the top-selling items today?
- Which customers spent the most today?
- Which employees took the most orders today?
- Which employees sold the most today, in dollars?

The questions above can all be answered by aggregating the detail data. For example, the first question could be answered with a report like this:

<table>
<thead>
<tr>
<th>Date</th>
<th>Total Sales</th>
</tr>
</thead>
<tbody>
<tr>
<td>2/10/2008</td>
<td>113,883.52</td>
</tr>
</tbody>
</table>

This report is drawn from the same data shown in the previous table. The difference is that a single record represents the whole day’s sales, rather than each individual item sold.

Decision makers often need information in an aggregated form. In the example above, summing is the form of aggregation. Another form of aggregation is counting, so the retail report might also show a count of the number of individual sales for the day. By dividing the sum by the count, we can get another form of aggregating: averaging. Even more helpful, might be a report that breaks down the day’s sales by category, store, or employee.

Aggregation performs two main functions, data reduction and the creation of new information. Data reduction reduces the amount of information that a decision maker must deal with. Many data records are replaced by few aggregated records. Data reduction can also be accomplished by filtering data. In addition, aggregation creates new information by performing a calculation over a group of records.
In SQL, a single statement can create can accomplish aggregation. This capability of SQL was a great relief to programmers, who were burdened with writing record-at-a-time software code. Before these aggregate functions were available, programmers had to get all the records, inspect each one, and write their own aggregation code. One of the important design goals of SQL, was to provide programmers with set-at-a-time operations. With one line of code, programmers were able to aggregate a set of records, which might have taken hundreds of lines without SQL.

2.4.2 Basics of Aggregation

Here are the built-in aggregating function in SQL.

- **COUNT** – count of non-null field values in a set of records
- **SUM** – sum of non-null (numeric) field values in a set of records
- **MIN** – the minimum of non-null field values in a set of records
- **MAX** – the maximum of non-null field values in a set of records
- **AVG** – the average of non-null (numeric) field values in a set of records
- **STD** – the standard deviation of non-null (numeric) field values in a set of records
- **VAR** – the variance of non-null field (numeric) values in a set of records

Each of these functions needs a specific field to operate on, as in this example:

```sql
SELECT COUNT(ID) AS [number of people]
FROM Person
```

In simple terms, the SQL statement above returns the number of people in the Person table. In more specific terms, it returns the number of records in the Person table that have non-null values in the ID field. Because the ID field is the primary key of the Person table, it must always be non-null; the database engine will not allow anyone to enter a NULL for that field value. So, counting on the ID field will accurately give the number of records in the Person table. Consider this SQL statement:

```sql
SELECT COUNT(weight) AS [number of people with non-null weight]
FROM Person
```

In this example, weight is not a primary key field, and can contain NULL values. In this case, only records where weight is non-null will be counted. In short, it is recommended that you, in general, use COUNT() on primary key fields. Of course, there are times that you will want to get a COUNT of people who have a non-null weight, and you will have to COUNT() on the weight field. This is the exception, rather than the rule, and it is a good habit to count on a primary key field.

You may see examples of aggregation using the * wildcard:

```sql
SELECT COUNT(*)
FROM Person
```

As you recall, * in the SELECT means “all columns”. In this situation it means: to determine which records to count, check all fields for non-null values. Assuming the table has a primary key, the COUNT is guaranteed to find a non-null field for every record. However, there are database tables that do not
have a primary key. (Most database experts would avoid this in an OLTP setting, but it happens in older systems, and in maintenance situations.) In this case, the * operator is useful, and is the most accurate way to get a record count. However, in a table without a primary key, there is no guarantee that there isn’t a record with entirely null values.

In general, the * operator is to be avoided. In situations where there are multiple tables from which data is drawn, the * operator is ambiguous. By using, for example, the ID field from the Person table, you are documenting that you are counting Person records.

Here are several SQL aggregation examples and what each means in English. Note that each of the following SQL statements results in a one-row, one-column table. When there are no non-null values (such as an empty table), the result will be a zero-row, one-column table.

```
SELECT AVG(weight) AS [average weight]  
FROM   Person  

The average of all non-null weights in the Person table, e.g., the average weight people.
```

```
SELECT MIN(weight) AS [minimum weight]  
FROM   Person  

The minimum of all non-null weights in the Person table, e.g., the lowest weight.
```

```
SELECT MIN(lastName) AS [first lastName]  
FROM   Person  

Alphabetically, the first non-null lastName in the Person table.
```

```
SELECT MIN(weight) AS [minimum weight]  
FROM   Person  
WHERE  gender='M'  

The minimum of all non-null weights for records that have ‘M’ in the gender field, e.g., the lowest weight of a male in the table.
```

```
SELECT SUM(orderTotal) AS [total revenue]  
FROM   Sale  
WHERE  MONTH(orderDate)=1  
AND     YEAR(orderDate)=2009  

The total of all non-null orderTotals from the Sale table for January 2009.
```
2.4.3 DISTINCT

Suppose that you wish to see all the values that appear in the gender column of the Person table. You could write this:

```
SELECT gender
FROM Person
```

This would produce a 1-column table with one row for every record in the Person table. Unfortunately, there would be repeated values, which might look something like this:

<table>
<thead>
<tr>
<th>gender</th>
</tr>
</thead>
<tbody>
<tr>
<td>M</td>
</tr>
<tr>
<td>M</td>
</tr>
<tr>
<td>F</td>
</tr>
<tr>
<td>F</td>
</tr>
<tr>
<td>F</td>
</tr>
<tr>
<td>M</td>
</tr>
<tr>
<td>NULL</td>
</tr>
<tr>
<td>M</td>
</tr>
<tr>
<td>NULL</td>
</tr>
</tbody>
</table>

With just a few records (as in the results above), you can see that there are only three different values in the gender column. However, if there were millions of rows, it would be very difficult to verify that only three values appeared: M, F, and NULL. To see only the different ones, we can specify in the SELECT clause to only show the DISTINCT ones:

```
SELECT DISTINCT gender
FROM Person
```

Which would produce:

<table>
<thead>
<tr>
<th>gender</th>
</tr>
</thead>
<tbody>
<tr>
<td>M</td>
</tr>
<tr>
<td>F</td>
</tr>
<tr>
<td>NULL</td>
</tr>
</tbody>
</table>

In other words, the duplicate values in the results are removed. The duplicates are determined based on the fields displayed, not on the underlying records. For example, suppose our person table has these records:

<table>
<thead>
<tr>
<th>ID</th>
<th>firstName</th>
<th>lastName</th>
<th>gender</th>
</tr>
</thead>
</table>

© 2010, Douglas M. Kline  Do Not Copy, Personal Use Only 37
### 2.4.4 GROUP BY

So far, the aggregation capability has been limited by what was possible with the WHERE clause. For example, to determine the average weight of males and females, two separate SELECT statements with different WHERE clauses are needed:

```sql
SELECT AVG(weight) AS [average male weight]
FROM Person
WHERE gender='M'

SELECT AVG(weight) AS [average female weight]
FROM Person
WHERE gender='F'
```

Using the GROUP BY clause, this can be accomplished with a single SQL statement:

```sql
SELECT gender,
       AVG(weight) AS [average weight]
FROM Person
GROUP BY gender
```

Which will give results like:

<table>
<thead>
<tr>
<th>gender</th>
<th>average weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>NULL</td>
<td>185.2</td>
</tr>
<tr>
<td>M</td>
<td>185.9</td>
</tr>
<tr>
<td>F</td>
<td>121.1</td>
</tr>
</tbody>
</table>

The GROUP BY clause in this statement tells the database engine to group all the records in the Person table by gender, and calculate the average of the weight column within each of the groups. In this case, there happened to be records that contained NULLs in the gender field, so this was treated as a separate group. A better version of this example that produces cleaner results might be:

```sql
SELECT gender,
       AVG(weight) AS [average weight]
FROM Person
WHERE weight IS NOT NULL
```
GROUP BY gender
ORDER BY gender

Which will give results like:

<table>
<thead>
<tr>
<th>gender</th>
<th>average weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>121.1</td>
</tr>
<tr>
<td>M</td>
<td>185.9</td>
</tr>
</tbody>
</table>

A common syntax error when using aggregate functions is to GROUP BY a field in the SELECT clause. For example:

```sql
SELECT gender,
       AVG(weight) AS [average weight]
FROM Person
-- produces a syntax error!
```

This will produce a syntax error, generally requesting that you include gender in a GROUP BY clause. To see why, think about the number of rows that should be returned by the SELECT statement. If the above SELECT were separated into two, by column, we would get:

```sql
SELECT gender
FROM Person
-- returns one row for each record in the Person table

SELECT AVG(weight)
FROM Person
-- returns one row
```

As you can see, the database engine wants to create many rows for the first column, but only one for the second column. In other words, the database engine wants for a clear specification of what the resulting table should look like. In particular, specify the grouping using the GROUP BY clause.

Another way to think of this is that when using aggregation, you are asking the result of the SELECT statement to have one record per group, rather than one record per record in the base table. If you are still learning the GROUP BY statement, it is helpful to think carefully about how many rows you’d like the SELECT statement to produce. One for each record? One for each gender? One for each lastName?

The reasoning behind this syntax error can be hard to remember, so here is the general rule:

When aggregating, every column in the SELECT clause must be either (1) in the GROUP BY or (2) aggregated.

Here is a more complicated example that satisfies the rule above:

```sql
SELECT gender,
       firstName,
       AVG(weight) AS [average weight],
       COUNT(ID) AS [count]
```
FROM Person
WHERE gender IS NOT NULL
AND firstName IS NOT NULL
GROUP BY gender, firstName
ORDER BY gender, firstName

Notice that the gender and firstName fields, which are not aggregated, appear in the GROUP BY clause. However, the weight and ID fields, which are averaged and counted, do not appear in the GROUP BY clause. Here are possible results for the above SQL statement:

<table>
<thead>
<tr>
<th>gender</th>
<th>firstName</th>
<th>average weight</th>
<th>count</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>Abbey</td>
<td>123.4</td>
<td>2</td>
</tr>
<tr>
<td>F</td>
<td>Alice</td>
<td>122.8</td>
<td>4</td>
</tr>
<tr>
<td>F</td>
<td>Allison</td>
<td>122.9</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>Barbara</td>
<td>124.0</td>
<td>5</td>
</tr>
<tr>
<td>F</td>
<td>Bella</td>
<td>123.3</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>M</td>
<td>Gary</td>
<td>182.2</td>
<td>4</td>
</tr>
<tr>
<td>M</td>
<td>Gerald</td>
<td>184.8</td>
<td>2</td>
</tr>
</tbody>
</table>

The general syntax of the GROUP BY clause is similar to the ORDER BY clause:

GROUP BY expression, expression, ...

Although it is common to GROUP BY basic fields from a table, you can use any expression GROUP BY clause. For instance, a revenue report SELECT might look like:

```
SELECT MONTH(orderDate) [month],
      SUM(orderAmount) [total sales]
FROM Sale
WHERE YEAR(orderDate) = 2009
GROUP BY MONTH(orderDate)
ORDER BY MONTH(orderDate)
```

Here, the MONTH() function is used to get the number of the month of the orderDate, which is the basis for grouping. Also notice that the ORDER BY clause looks very similar to the GROUP BY clause. It is common (but not necessary) to want sort the groups in the same way that the aggregation is performed.

### 2.4.5 HAVING

As mentioned above, the WHERE clause specifies which records will be included in the SELECT results. However, when you use GROUP BY, each resulting record represents a group, e.g., a record for each
month’s total sales, or a record for each gender’s average weight. There must be a way to control which group records are returned. There is – it is the HAVING clause. Here is the retail revenue example for only months where sales exceed $1 million:

```
SELECT MONTH(orderDate) [month],
       SUM(orderAmount) [total sales]
FROM Sale
WHERE YEAR(orderDate) = 2009
GROUP BY MONTH(orderDate)
HAVING SUM(orderAmount) > 1000000
ORDER BY MONTH(orderDate)
```

Notice that the HAVING clause follows the form of the WHERE clause, except that the logical expressions in HAVING must include an aggregate function, and the logical expressions in WHERE must not include an aggregate function. And similar to the WHERE clause, logical expressions in the HAVING clause can be combined with the logical operators NOT, AND, and OR.

The SELECT statement below shows only months where the total sales are less than $1 million, and the average orderAmount is less than $100:

```
SELECT MONTH(orderDate) [month],
       SUM(orderAmount) [total sales]
FROM Sale
WHERE YEAR(orderDate) = 2009
GROUP BY MONTH(orderDate)
HAVING NOT SUM(orderAmount) > 1000000
   AND AVG(orderAmount) < 100
ORDER BY MONTH(orderDate)
```

In general, if you use a regular (not a aggregated) field in the HAVING clause, the database engine will give you a syntax error. However, be careful of this common mistake, which generally does not give an error:

```
SELECT firstName, lastName, weight
FROM Person
WHERE weight > AVG(weight)
-- this is a common error!
```

Novices may write this SELECT statement to try to obtain all above-average-weight people. However, because this is in a WHERE clause, which is evaluated on a record-by-record basis, this is actually asking for all people whose weight is above the average of their own weight. For instance, if David Jones weighs 185.0 pounds, is [185.0 > AVG(185.0)]? In all cases, this logical expression will evaluate to zero, and there will be zero records in the result set. But the database engine won’t give an error message – it assumes that you know what you are asking for, and speedily provides the result. (To get all above-average-weight people requires a sub-query, which will be covered shortly.)

Aggregating is an important way to produce valuable information from raw data. First, it addresses information overload by summarizing the attributes of each group (SUM, COUNT, MIN, etc.) rather than the details of every item in the group. Aggregated values are commonly used as the upper levels of drill-down reports. Decision makers at higher levels of management in an organization are likely to desire
information aggregated specifically to their level of management. For instance, a sales manager in charge of sales over the Southeastern United States will likely want sales reports grouped by the states she is in charge of, rather than reports of every individual sale, or reports grouped by continent.