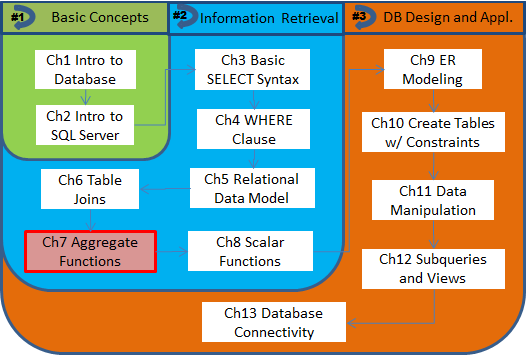
**6Chapter 7 Aggregate Functions**

**Where Is This Chapter Covered In The Course?**

****

**Chapter Outlines**

The Needs for Generating Summarized Information

Aggregate Functions to Be Discussed

Summarizing Data Using Aggregate Functions

* SUM()
* COUNT()
* MIN() and MAX()
* AVG()

Summarize Data by Groups

* The GROUP BY clause
* The HAVING clause

Putting It All Together

* Row-level vs. group-level conditions
* A query with table join

**The Needs for Generating Summarized Information**

We have covered how to use the SELECT statement to retrieve information based on data stored in one or more tables in the database. We can use various ways to specify conditions so that only information relevant to the questions at hand will be returned. However, we have not explore all power embedded in the SELECT statement. Two of the six clauses the can be used are still not discussed. Literally, these two clauses are GROUP BY and HAVING. We still do not know how to summarize the data to answer questions like “What is the total sales amount in July 2010?”

To generate summarized information, we need to use ***aggregate functions***, which tend to return one summarized value for a group, such as the whole table. These kind of summarized terms (total, average, min/max, etc.) are essential for a good understanding of business performances, and hence are frequently needed by managers and other users.

**Aggregate Functions to Be Discussed**

In SQL, a number of aggregate functions are standard to come up with summarized information. In this course we are going to discuss the following aggregate functions as supported in SQL Server[[1]](#footnote-1).

Table 7.1 - Select Aggregate Functions Discussed

|  |  |  |
| --- | --- | --- |
| **Function** | **Returns** | **Applicable Data Types** |
| AVG() | The average of the values in a dataset. Null values are ignored. | Numeric |
| COUNT() | The number of items in a group. | Numeric, character strings, data/time |
| MIN()/MAX() | The minimum/maximum value in the expression included in the parenthesis. | Numeric, character strings, data/time |
| SUM() | The sum of all the values in the expression included in the parenthesis. | Numeric |

Some general comments on the aggregate functions include:

* All the functions take only one argument, in the types as specified in the table.
* The functions will ignore NULL values when a specific expression (such as a column or a formula) is used as the argument.
* The DISTINCT keyword can be used to make the calculation applicable to distinct values only.

**Summarizing Data Using Aggregate Functions**

We will show some examples of how to use aggregate functions. When used, they can only appear in the SELECT clause (and the HAVING clause that we will introduce later in this chapter).

**SUM().** We will start with the question that we asked a bit earlier, “*What is the total sales amount in July 2010?*” To limit the invoice date to the particular month, we can use a BETWEEN … AND … expression. An alias "July Sales Total" can be used to display the result under a meaningful column label.

Our first summarized query (Q7.1) using the SUM() function is shown as below.

/\* Q7.1 \*/

SELECT **SUM(InvoiceTotal)** "July Sales Total"

FROM Invoices

WHERE InvoiceDate BETWEEN '07-01-2010' AND '07-31-2010'

Recall that, the double quotes are needed to enclose the alias as one token.

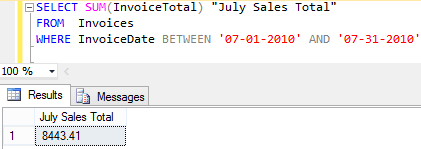


Figure 7.1 - SUM of InvoiceTotal Amount (Q7.1)

Q7.1 shows a very simple query, but it reveals a significant aspect of result from a query using an aggregate function. It summarizes the data and returns one single value for the whole dataset according to the applicable criteria.

By default, all InvoiceTotal values for the particular month are used. To clarify that, we can use the ALL keyword as in

SUM(**ALL** InvoiceTotal)

Another option other than ALL is DISTINCT, which is needed when you want to remove duplicate values. It is not applicable to this case. We will show how to use the DISTINCT keyword with the COUNT() function.

**COUNT().** In Chapter 6, we could use some help from this function when trying to get the number of vendors in our database. Here we will learn how to use the COUNT() function correctly.

First thing to notice from Table 7.1 is that the COUNT() function can take all data types that we discuss in this course, whereas SUM()or AVG() are applicable to numeric values only.

The most important thing to remember is that there are three ways to use COUNT() function.

Table 2 - Three Ways to Use the COUNT Function

|  |  |
| --- | --- |
| **Syntax** | **Returns** |
| COUNT(\*) | The number of items in a dataset. |
| COUNT(expr) | The number of items with a non-NULL value in the expression. |
| COUNT(DISTINCT expr) | The number of items with a distinct (non-NULL) value in the expression. |

***Count anything (or \*)*:** Let us see the differences with the next three queries. The first one (Q7.2) demonstrates how to get “*how many vendors are there in the Vendors table?*”

/\* Q7.2 \*/

SELECT **COUNT(\*)** "Total Vendor Count"

FROM Vendors

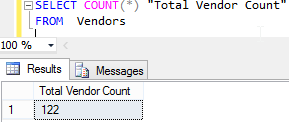


Figure 7.2 - Simply Counting Number of Entries in the Table (Q7.2)

Checking with results from Chapter 6, we know that when the wildcard character '\*' is used, counting is carried out based on number of entries (or rows) in the dataset (here the whole table), since no specific column is involved.

***Count non-null values*:** When a specific expression is used as the argument, the COUNT() function will ignore NULL values. To show this, we will display number of invoices in the Invoices table using COUNT(\*) and number of paid invoices (of which the PaymentDate contains a non-NULL value) using COUNT(PaymentDate).

/\* Q7.3 \*/

SELECT **COUNT(\*)** "Total Invoice Count",

**COUNT(PaymentDate)** "Paid Invoice Count"

FROM Invoices

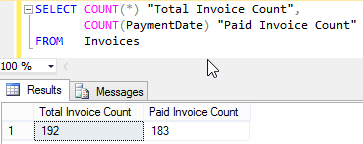


Figure 7. - Count Non-NULL Values in an Expression (Q7.3)

Query Q7.3 returns the total and paid invoice counts side-by-side: 183 of the 192 invoices in the table are paid. The count of unpaid invoices can be easily verified by using query (Q7.3a), which returns number of entries in the dataset filtered by the given criterion.

/\* Q7.3a \*/

SELECT **COUNT(\*)** "Unpaid Invoice Count"

FROM Invoices

WHERE **PaymentDate IS NULL**

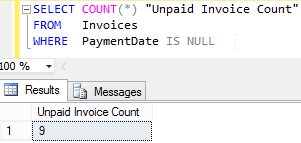


Figure 7. - COUNT(\*) with a WHERE Clause (Q7.3a)

***Count distinct values*:** In some cases, we need to know the count of distinct values, as in “*how many different vendors have any invoices associated with them*”. To answer such questions, we need to use the DISTINCT keyword in the COUNT() function, before the specific expression that we are checking on. Query Q7.4 shows the SQL that provides the vendor count.

/\* Q7.4 \*/

SELECT **COUNT(DISTINCT VendorID)** "Distinct Vendor Count",

COUNT(VendorID) "Non-NULL Vendor Count",

COUNT(VendorID) "Total Invoice Count"

FROM Invoices

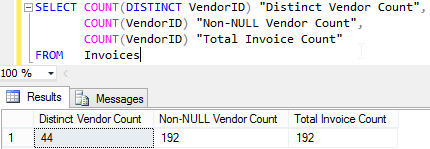


Figure 7.5 - Count Distinct Vendors (Q7.4)

**MIN() and MAX().** Just like the COUNT() function, the MIN() and MAX() functions can be applied to all data types that we discuss in this course. We will show sample queries for numeric and date values.

***Numeric values*:** For numeric values, these two functions return what their names suggest. For instance, we can retrieve the highest number of items included in any invoice suing the MAX() function as in query Q7.5.

/\* Q7.5 \*/

SELECT **MAX(InvoiceSequence)** "Highest Number of Items"

FROM InvoiceLineItems

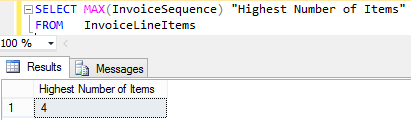


Figure 7. - Applying MAX() Function on Numeric Values (Q7.5)

***Date/time values*:** When taking a date/time expression as argument, these two functions return earliest (MIN()) or latest (MAX()) date/time values accordingly. In query Q7.6, we will retrieve earliest and latest payment dates.

/\* Q7.6 \*/

SELECT **MIN(PaymentDate)** "Earliest Payment Date",

**MAX(PaymentDate)** "Latest Payment Date"

FROM Invoices

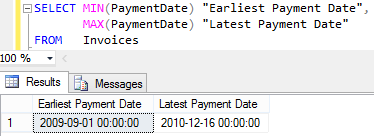


Figure 7. - Using MIN/MAX Functions Retrieving Earliest/Latest Dates (Q7.6)

From Q7.3a we know that there are (9) NULL values in the PaymentDate column. However, neither earliest nor latest date is null. This shows that NULL values are simply ignored in carrying out these two functions. The same is true for other aggregate functions that we cover, when a specific column or formula is use as argument.

**AVG().** This function applies to numeric values only, just like the SUM() function. It also ignores NULL values. This can be seen in query Q7.7, which retrieves the average payment total amount using

AVG(PaymentTotal)

To understand how the AVERAGE() function dealing with NULL values, Let us examine query Q7.7, where the average invoice total for all records is compared with two expressions with division, SUM(PaymentTotal)/COUNT(PaymentDate) and SUM(PaymentTotal)/COUNT(\*):

/\* Q7.7 \*/

SELECT COUNT(PaymentTotal) "Invoice Count",

COUNT(PaymentDate) "Payment Count",

**AVG(PaymentTotal)** "Average Payment Amount",

**SUM(PaymentTotal)/COUNT(PaymentDate)** AS "Q1",

**SUM(PaymentTotal)/COUNT(\*)** AS "Q2"

FROM Invoices;

An additional query reveals that for the 9 unpaid invoices, 0 is stored as default value for payment total.

SELECT PaymentTotal, PaymentDate

FROM Invoices

WHERE PaymentDate IS NULL;

First off, Figure 7.8 shows that it is possible to execute more than one query at a time. Result sets returned from different queries are displayed in separate Results tabs.

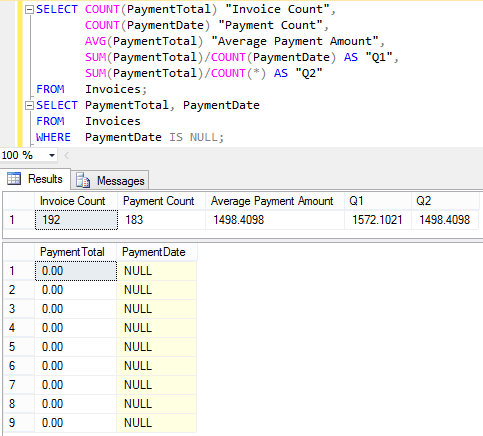


Figure 7. - A Query with Multiple Aggregate Terms (Q7.7)

In this query, the average value of PaymentTotal has been calculated in three ways, one from direct application of the AVG() function, two expressions with division are labeled as Q1 and Q2, as detailed below:

* Q1: the quotient of sum of PaymentTotal and the number of invoices that have been paid (indicated by a non-null PaymentDate);
* Q2: the quotient of sum of PaymentTotal and the number of all invoices in the table, including the 8 unpaid one.

The average value returned by using the AVG() function coincides with Q2, since the unpaid invoices have 0.00 (instead of NULL) in the PaymentTotal field. If you need to report the average of all the paid invoices, you need to use the expression for Q1.

**Summarize Data by Groups**

Now we know how to use aggregate functions to summarize data to answer various business questions. But we are limited to generate the total for one “group” at a time, with the group being all records in a table, or records that meet certain condition, such as “vendors in a certain state”. We do not want to run a query once for each of the states. We will learn how to partition a dataset (like a table) into groups and summarize data by group using the GROUP BY clause.

**The GROUP BY clause.** We are introduced to this clause in Chapter 3, which is listed as one of the 6 clauses that can appear in the SELECT statement. To show how to use it, let us consider this scenario: “*We need to report number of vendors in each state, and list the vendor count next to the state name*”.

It is attempting to modify Q7.2 into

SELECT **State,** COUNT(\*) "Vendor Count"

FROM Vendors

so as to generate results in the format of

|  |  |
| --- | --- |
| **State** | **Vendor Count** |
| AZ | 3 |
| CA | 75 |
| … | … |

Unfortunately, this query will not work. When tried in the Query Editor, an error message will be returned, such as “Column 'Vendors.State' is invalid in the select list because it is not contained in either an aggregate function or the GROUP BY clause.”. The reason for that is the mismatch between results returned by using State (a normal term) and COUNT(\*) (an aggregate term):

* As we learned in Chapters 3, 4, and 6, normal terms will return a value for each record in the group; whereas
* In this chapter, we have seen aggregate terms consolidate all records in the group and return only one summarized value for the whole group.

To illustrate this discrepancy, if the query would return any results, values returned in the two columns would look like

|  |  |
| --- | --- |
| **State** | **Vendor Count** |
| AZ | 3 |
| AZ |
| AZ |
| … | … |

The GROUP BY clause is designed to be used in this case, to label the normal term such as State is a group label. Therefore, only one value needs to be returned per group, so as to match up with the aggregate term, which also returns a value for each group. As in query (Q7.8)

/\* Q7.8 \*/

SELECT **State,** COUNT(\*) "Vendor Count"

FROM Vendors

**GROUP BY State**

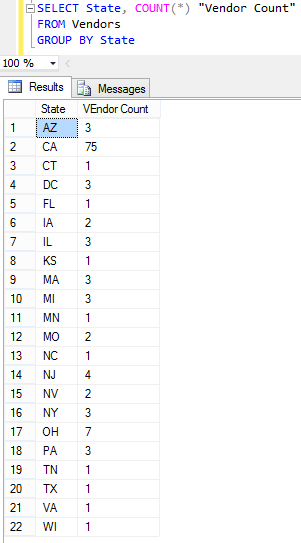


Figure 7. - The GROUP BY Clause Specifies a Group Label (Q7.8)

State (the normal term) that appears together with an aggregate term needs to be included in a GROUP BY clause. This query executes successfully to generate a State-Vendor Count list, as shown in Figure 7.9.

Examining the list closely, you can find out that the result set is automatically sorted on the group label (State) in ascending order. To sort the results in a different way (such as listing results by vendor count from highest to lowest), an ORDER BY clause should be used at the end of the SELECT statement.

**The HAVING clause.** This is the last clause to cover in the SELECT statement. We are almost there.

The reader of the last report (State-Vendor Count list) may not be interest in seeing all 22 states. In reporting to your client, providing more data is just as unacceptable as not providing enough. The client might just need to know the state and associated vendor count if a state has more 3 vendors. We need to add an appropriate criterion to filter out the insignificant pairs. But how?

Your first impression may be “I need to add a WHERE clause”. That is not the right answer:

* No aggregate term can be used in the WHERE clause; also
* It is not logically sound because the WHERE clause needs to appear before the GROUP BY clause: counting cannot be carried out before “how to count it” is specified (in the GROUP BY clause).

Your second guess is: “it should be the clause that yet to be covered.” You are right this time: It is the HAVING clause, which is designed to specify ***group-level conditions***. When used, the HAVINGclause needs to appear right after the GROUP BY clause, using only aggregate terms or constants.

/\* Q7.9 \*/

SELECT State, COUNT(\*) "Vendor Count"

FROM Vendors

GROUP BY State

**HAVING COUNT(\*) > 3**

ORDER BY 2 DESC

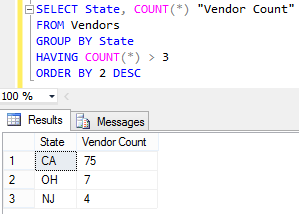


Figure 7. - Specifying Group-Level Conditions in the HAVING Clause (Q7.9)

As shown in Figure 7.10, only states with more than three (3) vendors are included in the result set. Also, results are sorted by vendor count in descending order as specified with the ORDER BY clause.

It is worth mentioning that non-aggregate terms can be used in the HAVING clause, since group-wide condition can only be specified with aggregate terms.

**Putting It All Together**

We have finally covered all six clauses that can be used in a SELECT statement. We will show a couple examples of using all six in one query. We will also discuss how a SELECT query will be executed with all six clauses in place. This insight can help us remember the order of the clauses and plan for a complex query in this course and in solving real world tasks.

**Row-level vs. group-level conditions**. It is essential to apply appropriate conditions to retrieve just the relevant information for a question from the client. Now we have two clauses to specify conditions:

* The WHERE clause: for conditions applicable for every row that needs to include; and
* The HAVING clause: for conditions applicable for each group as a whole.

Let us modify on our request for the State-Vendor Count list. Assume that the client is only concerned about vendors that have its default (payment) term as 30 days when asking about vendor counts. A close look at the Terms table shows that there are 6 different terms, varying from 10 days to 90 days; and 30 days corresponds to a TermsID value of 3.

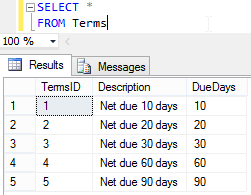


Figure 7. - Contents of the Terms Table

Adding a WHERE clause into Q7.9 and modify the group-wise condition slightly (to COUNT(\*) >= 3), we finally come up with our first SELECT statement (Q7.10) with all six (6) clauses.

/\* Q7.10 \*/

SELECT State, COUNT(\*) "Vendor Count"

FROM Vendors

WHERE **DefaultTermsID = 3**

GROUP BY State

HAVING **COUNT(\*) >= 3**

ORDER BY "Vendor Count" DESC

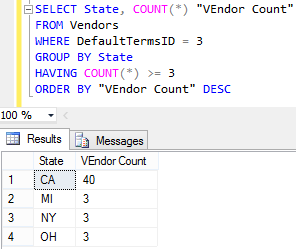


Figure 7. - Row-level Vs. Group-Level Conditions (Q7.10)

As shown in Figure 7.12, four states with 3 or more vendors are listed. Comparing against Figure 7.10 we know that vendor counts for CA and OH are down and NJ is now absent, both indicating the row-level condition on terms is taken into account. MI and NY are listed with 3 vendors, which coincide with numbers shown in Figure 7.9. All vendors in these two states must enjoy the 30-day payment term (which can be verified with a simple query).

How exactly will such a 6-clause query be carried out? Let us walk it through in our mind:

* First off, the two mandatory clauses (SELECT and FROM) will be processed to get a sense of what table (or tables) and columns in the table(s) will be needed.
  + Expressions in the other 4 clauses may add to the column list.
  + Results cannot be determined before the next three (3) clauses are processed nor sorted until the ORDER BY clause is finished.
* Row-level conditions will be applied to the source dataset (with possible table joins).
* The results will then be partitioned according to the GROUP BY scheme, and
* Group-level conditions will be applied.
* Before returning to the Results tab, results will be sorted. Sorting schemes can be specified by
  + Column names or expressions with operator/functions;
  + Column aliases; or
  + Column positions as appeared in the SELECT clause.

**A query with table join**. To wrap up the chapter, we will use the next query trying to incorporate all features that we have covered so far in this course, including

* Joining tables;
* Using multiple grouping attributes;
* Specifying a group-level condition with column not used in the SELECT clause; and
* Sorting results on multiple expressions with both ASC and DESC orders.

The business question to be answered is “*total invoice amount for each of the vendors not from New York or Pennsylvania and with at least invoices*”. The results need to be grouped are displayed first by state alphabetically and then by vendor name from highest to lowest.

To plan for this query, one needs to:

* Find out what tables to include.
  + Do not forget to include any intermediate table that does not provide any (descriptive) attribute but is necessary in connecting the path.
* Format the table joins (inner or outer) with appropriate joining conditions.
* Identify row-level conditions as applicable.
* Use a GROUP BY clause and include all non-aggregate terms in the SELECT clause if any aggregate term also appears.
  + A group by attribute does not need to appear in the SELECT clause, though it makes the results less readable.
  + Consider the order for listing the group by terms.
* Use a HAVING clause to specify group-level conditions as necessary. Only aggregate terms may be used.
  + Aggregate terms used here do not need to appear in the SELECT list.
* Sort the results with an ORDER BY clause.

The complete query (Q7.11) is listed below, with the results showing in Figure 7.13.

/\* Q7.11 \*/

SELECT State, Name, SUM(InvoiceTotal)

FROM Vendors JOIN Invoices

ON (Vendors.VendorID = Invoices.VendorID)

WHERE State NOT IN ('NY', 'PA')

GROUP BY State, Name

HAVING COUNT(InvoiceTotal) > 3

ORDER BY 1, 3 DESC

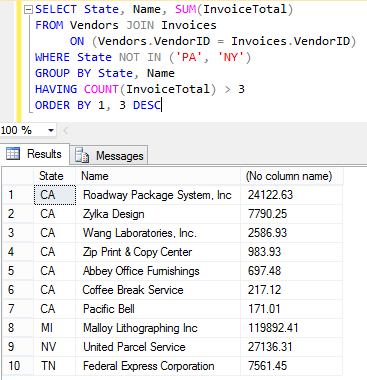


Figure 7.13 - Putting It All Together (Q7.11)

**Review Checklist**

The Needs for Generating Summarized Information

Aggregate Functions to Be Discussed

Summarizing Data Using Aggregate Functions

* SUM()
* COUNT()
  + Count all entries in a table/group
  + Count (all) non-null values in a column/formula
  + Count distinctive non-null values in a column/formula
* MIN() and MAX()
* AVG()

Summarize Data by Groups

* The GROUP BY clause
  + Normal terms listed together with aggregate terms need to be included in the GROUP BY clause
* The HAVING clause
  + May use aggregate terms and constants only

Putting It All Together

* Row-level vs. group-level conditions
* Using table joins
* Sorting results

1. Aggregate Functions <https://docs.microsoft.com/en-us/sql/t-sql/functions/aggregate-functions-transact-sql>. [↑](#footnote-ref-1)