CHAPTER 11

Aggregating Data

In This Chapter

Calculating sums and averages

Statistical analysis

Grouping data within a query

Solving aggravating aggregation problems

Building dynamic crosstab queries

The Information Architecture Principle in Chapter 1 implies that the asset is information, not just data. Turning raw lists of data and keys into useful information often involves summarizing data and grouping it in meaningful ways. While a certain amount of summarization and analysis can be performed with other tools, such as Reporting Services or Analysis Services, SQL is a set-based language, and a fair amount of summarizing and grouping can be performed very well within the SQL `select` statement.

SQL excels at calculating sums, max values, and averages for the entire data set or for segments of data. In addition, SQL queries can create cross-tabulations, commonly known as *pivot tables*.

While ANSI-92 SQL includes plenty of standard aggregation features, SQL Server 2005 includes the capability to roll your own aggregate functions using the common language runtime. I have no doubt that third-party libraries of custom aggregate functions will appear.

Simple Aggregations

The basic gist of an aggregate query is that instead of returning all the selected rows, SQL Server returns a single row of computed values that summarizes the original data set, as illustrated in Figure 11-1. The types of calculations range from totaling the data to performing basic statistical operations.

It's important to note that in the logical order of the SQL query, the aggregate functions occur following the `from` clause and the `where` filters. This means that the data can be assembled and filtered prior to being summarized without having to use a subquery.

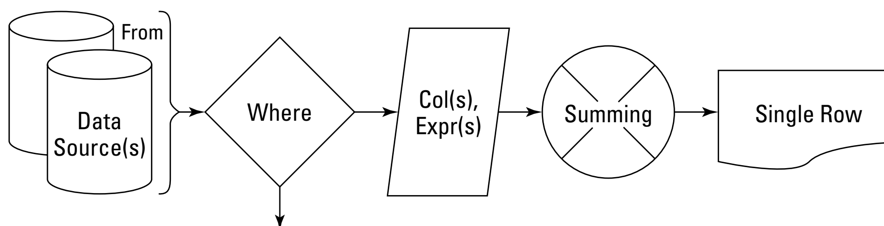


Figure 11-1: The aggregate function produces a single row result from a data set.

Basic Aggregations

SQL includes a set of *aggregate functions*, listed in Table 11-1, which can be used as expressions in the `select` statement to return summary data.

Table 11-1: Basic Aggregate Functions

Aggregate Function	*Data Type Supported*	*Description*
`sum()`	Numeric	Totals all the non-null values in the column.
`avg()`	Numeric	Averages all the non-null values in the column. Input data type will be returned by avg(), so the input is often converted to a higher precision, such as avg(cast col as a float).
`min()`	numeric, string, datetime	Returns the smallest number or the first datetime or the first string according to the current collation from the column.
`max()`	numeric, string, datetime	Returns the largest number or the last datetime or the last string according to the current collation from the column.
`count([distinct] *)`	Any data type (row-based)	Performs a simple count of all the rows in the result set up to 2,147,483,647. Will not count unique identifiers or blobs.
`count_big([distinct] *)`	Any data type (row-based)	Similar to the count() function, but the bigint data type can handle up to 2^63-1 rows.

On The Web

The code examples for this chapter use a small table called `RawData`. The code to create and populate this data set is at the beginning of the chapter's script. You can download the script from `www.SQLServerBible.com`.

```
CREATE TABLE RawData (
  Region VARCHAR(10),
  Category CHAR(1),
  Amount INT,
  SalesDate DateTime
  )
```

This simple aggregate query counts the number of rows in the table and totals the `Amount` column. In lieu of returning the actual rows from the `RawData` table, the query returns the summary row with the count and total:

```
SELECT
    Count(*) as Count,
    Sum(Amount) as [Sum]
  FROM RawData
```

Result:

```
Count       Sum
----------- -----------
20          946
Warning: Null value is eliminated by an aggregate or
other SET operation.
```

If you're using Management Studio's Query Designer, the query can be converted into an aggregate query using the Group By toolbar button, as illustrated in Figure 11-2. The Group By column is used to select query columns for grouping or aggregate functions. To find the Query Designer, open a table using the Object Explorer.

Figure 11-2: Performing an aggregate query within Management Studio's Query Designer

Using the aggregate functions within a `select` statement is pretty straightforward. Here are a few rules to keep in mind while using aggregate functions:

✦ Because SQL is now returning information from a set, rather than building a record set of rows, as soon as a query includes an aggregate function, every column (in the column list, in the expression, or in the `order by`) must participate in an aggregate function. This is logical because if a query returned the total number of order sales, then it could not return a single order number on the same row.

✦ The aggregate (`distinct`) option serves the same purpose as `select distinct` except that it eliminates duplicate values instead of duplicate rows. Therefore, it's of questionable usefulness when used with `sum()` and `avg()`. `Count(distinct *)` is invalid; a column must be specified.

✦ `Count(*)` counts all the rows, but `count(column)` counts all the rows with a value in that column.

✦ Because aggregate functions are expressions, the result will have a null column name. Therefore, use an alias to name the column.

✦ The precision of the aggregate function is determined by the data type precision of the source column. The `Amount` column in the table `RawData` is only an integer data type, so the `avg()` function is calculated as an integer. Converting the data to `numeric(9,5)` can increase the precision of the result:

```
SELECT Avg(amount) as [Integer Avg],
  Avg(Cast((Amount)as Numeric(9,5))) as [Numeric Avg],
  Sum(amount) / Count(*) as [Manual Avg]
  FROM RawData
```

Result:

```
Integer Avg   Numeric Avg   Manual Avg
------------- ------------- ----------------
47            47.300000     39
```

✦ Aggregate queries ignore any null values, so a `sum()` or `avg()` aggregate function will not error out on a null, but simply skip the row with a null. For this reason, a `sum()/count(*)` calculation may provide a different result than an `avg()` function.

Beginning Statistics

Statistics is a large and complex field of study, and while SQL Server does not pretend to replace a full statistical analysis software package, it does calculate standard deviation and variance that is important for understanding the bell-curve spread of numbers.

An average alone is not sufficient to summarize a set of values (in the lexicon of statistics, a "set" is referred to as a *population*). The value in the exact middle of a population is the *mean* (which is different from the average). The difference, or how widely dispersed the values are from the mean, is called the population's *variance*. For example, the populations (1, 2, 3, 4, 5, 6, 7, 8, 9, 10) and (4, 4, 5, 5, 5, 5, 6, 6) both average to 5, but the values in the first set vary widely from the mean, while the second set's values are all close to the mean. The standard deviation is the square root of the variance, and describes the shape of the bell curve formed by the population.

The following query uses the StDevP() and VarP() functions to returns the statistical variance and the standard deviation of the entire population of the RawData table:

```
SELECT
    StDevP(Amount) as [StDev],
    VarP(Amount) as [Var]
  FROM RawData
```

Result:

```
StDevP               VarP
-------------------- --------
24.2715883287435     589.11
```

Note: If you need to perform extensive statistical data analysis, I recommend exporting the query result set to Excel and tapping Excel's broad range of statistical functions.

The statistical formulas differ slightly when calculating variance and standard deviation from the entire population versus a sampling of the population. If the aggregate query includes the entire population, use the StDevP() and VarP() aggregate functions, which use the *bias* or *n* method of calculating the deviation.

However, if the query is using a sampling or subset of the population, then use the StDev() and VarP() aggregate functions so SQL Server will use the unbiased or n-1 statistical method. Because group by queries slice the population into subsets, these queries should use StDev() and Var() functions.

Cross-Reference: For ranking functions including calculating percentiles, see Chapter 7, "Understanding Basic Query Flow."

Grouping within a Result Set

Aggregate functions are all well and good, but how often do you need a total for an entire table? Most aggregate requirements will include a date range, department, type of sale, region, or the like. That presents a problem. If the only tool to restrict the aggregate function were the where clause, database developers would waste hours replicating the same query, or writing a lot of dynamic SQL queries and the code to execute the aggregate queries in sequence.

Fortunately, aggregate functions are complemented by the group by function, which automatically partitions the data set into subsets based upon the values in certain columns. Once the data set is divided into subgroups, the aggregate functions are performed on each subgroup. The final result is one summation row for each group, as shown in Figure 11-3.

A common example is grouping the sales result by salesperson. A sum() function without the grouping would produce the sum() of all sales. Writing a query for each salesperson would provide a sum() for each person, but maintaining that over time would be a pain. The grouping function automatically creates a subset of data grouped for each unique salesperson, and then the sum() function is calculated for each salesperson's sales. Voilà.

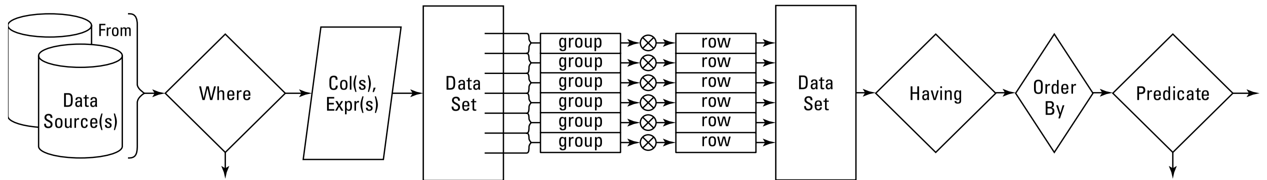


Figure 11-3: The group by clause slices the data set into multiple subgroups.

Simple Groupings

Some groupings use descriptive information for the grouping, so the data used by the `group by` is the same data you need to see to understand the groupings. These are straightforward, but in a large relational database, they can be rare. For example, the next query groups by the category:

```
SELECT Category,
    Count(*) as Count,
    Sum(Amount) as [Sum],
    Avg(Amount) as [Avg],
    Min(Amount) as [Min],
    Max(Amount) as [Max]
  FROM RawData
  GROUP BY Category
```

Result:

```
Category  Count    Sum      Avg      Min      Max
--------- -------- -------- -------- -------- --------
X         5        225      45       11       86
Y         11       506      46       12       91
Z         4        215      53       33       83
```

The first column of this query returns the `Category` column. While this column does not have an aggregate function, it still participates within the aggregate because that's the column by which the query is being grouped, and it may therefore be included in the result set. Each row in the result set summarizes one category, and the aggregate functions now calculate the row count, sum average, minimum value, and maximum value for each category.

SQL is not limited to grouping by one column. The preceding query is enhanced with the addition of a grouping by `ProductCategoryName`, as follows:

```
SELECT Year(SalesDate) as [Year], DatePart(q,SalesDate) as [Quarter],
    Count(*) as Count,
    Sum(Amount) as [Sum],
    Avg(Amount) as [Avg],
    Min(Amount) as [Min],
    Max(Amount) as [Max]
  FROM RawData
  GROUP BY Year(SalesDate), DatePart(q,SalesDate)
```

Result:

```
Year   Quarter   Count   Sum   Avg   Min   Max
------ --------- ------- ----- ----- ----- -----
2006   1         6       218   36    11    62
2006   2         6       369   61    33    86
2006   3         8       280   70    54    91
2005   4         4       79    19    12    28
```

For the purposes of a `group by`, null values are considered equal to other nulls and will be grouped together into a single result row

Aggravating Queries

There are a few aspects of group by queries that can be aggravating when developing applications. Some developers simply avoid aggregate queries and make the reporting tool do the work, but the Database Engine will be more efficient than any client tool. Here are five typical aggravating problems and my recommended solutions.

Including group by Descriptions

The previous aggregation queries all executed without error because every column participated in the aggregate purpose of the query. To test the rule, the following script adds a category table and then attempts to return a column that isn't included as an aggregation function or group by column:

```
CREATE TABLE RawCategory (
  RawCategoryID  CHAR(1),
  CategoryName   VARCHAR(25)
  )

INSERT RawCategory (RawCategoryID, CategoryName)
  VALUES ('X', 'Sci-Fi')
INSERT RawCategory (RawCategoryID, CategoryName)
  VALUES ('Y', 'Philosophy')
INSERT RawCategory (RawCategoryID, CategoryName)
  VALUES ('Z', 'Zoology')

-- including data outside the aggregate function or group by
SELECT Category, CategoryName,
    Sum(Amount) as [Sum],
    Avg(Amount) as [Avg],
    Min(Amount) as [Min],
    Max(Amount) as [Max]
  FROM RawData R
    JOIN RawCategory C
      ON R.Category = C.RawCategoryID
  GROUP BY Category
```

As expected, including `region` in the column list causes the query to return an error message:

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

There are two solutions for including non-aggregate descriptive columns.

If the query is an ad hoc, run once query, then it's OK to just include the additional columns in the `group by` clause:

```
SELECT Category, CategoryName,
    Sum(Amount) as [Sum],
    Avg(Amount) as [Avg],
    Min(Amount) as [Min],
    Max(Amount) as [Max]
  FROM RawData R
```

```
    JOIN RawCategory C
      ON R.Category = C.RawCategoryID
  GROUP BY Category, CategoryName
  ORDER BY Category, CategoryName
```

Result:

```
Category CategoryName   Sum   Avg   Min   Max
-------- -------------- ----- ----- ----- -----
X        Sci-Fi         225   45    11    86
Y        Philosophy     506   46    12    91
Z        Zoology        215   53    33    83
```

The problem with this approach is that it forces SQL Server to actually perform a grouping operation on every column regardless of whether that column is required or not to group the data, which can be an unnecessary performance hit.

For a query used in production, a better solution is to take it to the next level and perform the aggregate function in a subquery and then include the additional columns in the outer query:

```
SELECT sq.Category, CategoryName,
    sq.[Sum], sq.[Avg], sq.[Min], sq.[Max]
  FROM (SELECT Category,
            Sum(Amount) as [Sum],
            Avg(Amount) as [Avg],
            Min(Amount) as [Min],
            Max(Amount) as [Max]
          FROM RawData
          GROUP BY Category ) sq
    JOIN RawCategory C
      ON sq.Category = C.RawCategoryID
  ORDER BY Category, CategoryName
```

The subquery only has to do the work of the aggregate query and group by the category column. To fetch the category name, the result of the subquery is passed to the outer query, which uses a join to include the `RawCategory` table and access the `CategoryName` column.

Including All Group By Values

The group by functions occur following the `where` clause in the logical order of the query. This can present a problem if the query needs to report all the group by column values even though the data needs to be filtered. Although this is a rare request, there is an aggregate query solution that doesn't require outer joins and subqueries. The `Group By All` option includes all group by values regardless of the `where` clause. The next query uses this feature to return a 2005 group by row, even though 2005 data is not included in the aggregate calculation:

```
SELECT Year(SalesDate) as Year,
    Count(*) as Count,
    Sum(Amount) as [Sum],
    Avg(Amount) as [Avg],
    Min(Amount) as [Min],
    Max(Amount) as [Max]
  FROM RawData
  WHERE Year(SalesDate) = 2006
  GROUP BY ALL Year(SalesDate)
```

Result:

```
Year   Count  Sum   Avg    Min    Max
------ ------ ----- ------ ------ -----
2005   0      NULL  NULL   NULL   NULL
2006   20     867   54     11     91
```

Nesting Aggregations

Aggregated data is often useful, and it can be even more useful to perform secondary aggregations on aggregated data. For example, an aggregate query can easily `sum()` each category and year/quarter within a subquery, but which category has the max value for each year /quarter? An obvious `max(sum())` doesn't work because there's not enough information to tell SQL Server how to nest the aggregation groupings. Solving this problem requires a subquery to create a record set from the first aggregation, and an outer query to perform the second level of aggregation. For example, the following query sums by quarter and category, and then the outer query uses a `max()` to determine which sum is the greatest for each quarter:

```
Select Y,Q, Max(Sum) as MaxSum
  FROM ( -- Calculate Sums
         SELECT Category, Year(SalesDate) as Y,
            DatePart(q,SalesDate) as Q, Sum(Amount) as Sum
          FROM RawData
          GROUP BY Category, Year(SalesDate),
            DatePart(q,SalesDate)
        ) sq
  GROUP BY Y,Q
  ORDER BY Y,Q
```

Result:

```
Y           Q           MaxSum
----------- ----------- -----------
2005        4           79
2006        1           147
2006        2           215
2006        3           280
```

Including Detail Descriptions

While it's nice to report the `max(sum())` of 147 for the first quarter of 2006, who wants to manually look up which category matches that sum? The next logical step is including descriptive information about the aggregate data. To add descriptive information for the detail columns, join with a subquery on the detail values:

```
SELECT MaxQuery.Y, MaxQuery.Q, AllQuery.Category, MaxQuery.MaxSum as MaxSum
  FROM (-- Find Max Sum Per Year/Quarter
       Select Y,Q, Max(Sum) as MaxSum
         From ( -- Calculate Sums
                 select Category, Year(SalesDate) as Y,
                    DatePart(q,SalesDate) as Q, Sum(Amount) as Sum
                   from RawData
                   group by Category, Year(SalesDate), DatePart(q,SalesDate)
                   ) sq
```

```
        Group By Y,Q
        ) MaxQuery
  JOIN (-- All Data Query
       Select Category, Year(SalesDate) as Y, DatePart(q,SalesDate) as Q,
          Sum(Amount) as Sum
        From RawData
        Group By Category, Year(SalesDate), DatePart(q,SalesDate)
        ) AllQuery
    ON MaxQuery.Y = AllQuery.Y
      AND MaxQuery.Q = AllQuery.Q
      AND MaxQuery.MaxSum = AllQuery.Sum
  ORDER BY MaxQuery.Y, MaxQuery.Q
```

Result:

```
Y           Q           Category MaxSum
----------- ----------- -------- -----------
2005        4           Y        79
2006        1           Y        147
2006        2           Z        215
2006        3           Y        280
```

While the query appears complex at first glance, it's actually just an extension of the preceding query (in bold with the table alias of `MaxQuery`.)

The last subquery (with the alias of `AllQuery`) finds every the sum of category and year/quarter. Joining `MaxQuery` with `AllQuery` on the sum and year/quarter is used to locate the category and return the descriptive value along with the detail data.

Filtering Grouped Results

Filtering, when combined with grouping, can be a problem. Are the row restrictions applied before the `group by` or after the `group by`? Some databases use nested queries to properly filter before or after the `group by`. SQL, however, uses the `having` clause to filter the groups. At the beginning of this chapter you saw the simplified order of the SQL `select` statement's execution. A more complete order is as follows:

1. The `from` clause assembles the data from the data sources.
2. The `where` clause restricts the rows based on the conditions.
3. The `group by` clause assembles subsets of data.
4. Aggregate functions are calculated.
5. The `having` clause filters the subsets of data.
6. Any expressions are calculated.
7. The `order by` sorts the results.

Continuing with the `RawData` sample table, the following query removes from the analysis any grouping "having" an average of less than 25:

```
SELECT Year(SalesDate) as [Year],
    DatePart(q,SalesDate) as [Quarter],
    Count(*) as Count,
```

```
    Sum(Amount) as [Sum],
    Avg(Amount) as [Avg]
  FROM RawData
  GROUP BY Year(SalesDate), DatePart(q,SalesDate)
  HAVING Avg(Amount) > 25
  ORDER BY [Year], [Quarter]
```

Result:

```
Year   Quarter   Count   Sum   Avg
------ --------- ------- ----- -----
2006   1         6       218   36
2006   2         6       369   61
2006   3         8       280   70
```

Without the `having` clause, the fourth quarter of 2005, with an average of 19, would have been included in the result set. But the `having` clause executed after `group by` and calculation of the aggregate function, serving as a post-aggregate filter.

Generating Totals

While Reporting Services can easily add subtotals and totals without any extra work by the query, it might prove useful to supply the total for a .NET application for display at the bottom of a form or web page. If that's your challenge, then these next three aggregate commands are perfect solutions.

Rollup Subtotals

The `rollup` and `cube` aggregate functions generate subtotals and grand totals as separate rows, and supply a null in the `group by` column to indicate the grand total. `Rollup` generates subtotal and total rows for the `group by` columns. `Cube` extends the capabilities by generating subtotal rows for every `group by` column. A special function called `grouping()` is true when the row is a subtotal or total row. Here I'll demonstrate the `rollup` function.

The `rollup` option, placed after the `group by` clause, instructs SQL Server to generate an additional total row. In this example, the `grouping()` function is used by a `case` expression to convert the total row to something understandable:

```
SELECT
    CASE Grouping(Category)
      WHEN 0 THEN Category
      WHEN 1 THEN 'All Categories'
    END AS Category,
    Count(*) as Count
  FROM RawData
  GROUP BY Category
    WITH ROLLUP
```

Result:

```
Category       Count
-------------- -----------
```

```
X             5
Y             15
Z             4
All Categories 24
```

Adding a second column, `Year(SalesDate)`, to the `group by with rollup` query will cause SQL Server to calculate subtotals for the second column:

```
SELECT
    CASE Grouping(Category)
      WHEN 0 THEN Category
      WHEN 1 THEN 'All Categories'
    END AS Category,
    CASE Grouping(Year(SalesDate))
      WHEN 0 THEN Cast(Year(SalesDate) as CHAR(8))
      WHEN 1 THEN 'All Years'
    END AS Year,
    Count(*) as Count
  FROM RawData
  GROUP BY Category, Year(SalesDate)
    WITH ROLLUP
```

Result:

```
Category       Year       Count
-------------- ---------- -----------
X              2006       5
X              All Years  5
Y              2005       4
Y              2006       11
Y              All Years  15
Z              2006       4
Z              All Years  4
All Categories All Years  24
```

Cube Queries

A *cube query* is the next logical progression beyond a rollup query: It adds subtotals for every grouping in a multidemensional manner. Using the same example, the rollup query had subtotals for each category; the cube query adds subtotals for each year:

```
SELECT
    CASE Grouping(Category)
      WHEN 0 THEN Category
      WHEN 1 THEN 'All Categories'
    END AS Category,
    CASE Grouping(Year(SalesDate))
      WHEN 0 THEN Cast(Year(SalesDate) as CHAR(8))
      WHEN 1 THEN 'All Years'
    END AS Year,    Count(*) as Count
  FROM RawData
  GROUP BY Category, Year(SalesDate)
```

```
    WITH CUBE
  ORDER BY IsNull(Category, 'zzz')
```

Result:

```
Category       Year      Count
-------------- --------- -----------
X              2006      5
X              All Years 5
Y              2005      4
Y              2006      11
Y              All Years 15
Z              2006      4
Z              All Years 4
All Categories All Years 24
All Categories 2005      4
All Categories 2006      20
```

Computing Aggregates

The compute option is a completely different animal from any other aggregate query. Rather than build an aggregate query, think of the compute clause as an aggregate query tacked on to the end of a normal query. This query will return a normal result set with the detail rows and then add a few rows at the end to report some aggregate information about the same result set. In the following example the entire RawData table is returned, followed by what looks like a second result set with the average and sum:

```
SELECT Category, SalesDate, Amount
  FROM RawData
  WHERE Year(SalesDate) = '2006'
  COMPUTE  Avg(Amount), sum(Amount)
```

Result:

```
Category SalesDate               Amount
-------- ----------------------- -----------
X        2006-01-01 00:00:00.000 11
X        2006-01-01 00:00:00.000 24
...
Y        2006-08-01 00:00:00.000 NULL

avg         sum
----------- -----------
54          867
```

Compute clauses can even contain their own miniature group by clauses. In this case, all the detail rows are returned divided by the group by, and with subtotals, just like a full-featured report:

```
SELECT Category, SalesDate, Amount
  FROM RawData
  WHERE Year(SalesDate) = '2006'
  ORDER BY Category
```

```
  COMPUTE  Avg(Amount), sum(Amount)
    BY Category
```

Result:

```
Category SalesDate               Amount
-------- ----------------------- -----------
X        2006-01-01 00:00:00.000 11
...
X        2006-06-01 00:00:00.000 86

avg         sum
----------- -----------
45          225

Category SalesDate               Amount
-------- ----------------------- -----------
Y        2006-07-01 00:00:00.000 54
Y        2006-07-01 00:00:00.000 63
...
Y        2006-03-01 00:00:00.000 62

avg         sum
----------- -----------
61          427

Category SalesDate               Amount
-------- ----------------------- -----------
Z        2006-04-01 00:00:00.000 33
...
Z        2006-05-01 00:00:00.000 55

avg         sum
----------- -----------
53          215
```

The `compute by` did a fair job of including subtotals but didn't calculate a grand total. As strange as it seems, to add a grand total, combine both a `compute` and a `compute by`:

```
SELECT Category, SalesDate, Amount
  FROM RawData
  WHERE Year(SalesDate) = '2006'
  ORDER BY Category
  COMPUTE avg(Amount), sum(amount)
  COMPUTE sum(Amount)
    BY Category
```

Result:

```
Category SalesDate               Amount
-------- ----------------------- -----------
X        2006-01-01 00:00:00.000 11
...
```

```
X        2006-06-01 00:00:00.000 86

sum
-----------
225

Category SalesDate               Amount
-------- ----------------------- -----------
Y        2006-07-01 00:00:00.000 54
...
Y        2006-03-01 00:00:00.000 62

sum
-----------
427

Category SalesDate               Amount
-------- ----------------------- -----------
Z        2006-04-01 00:00:00.000 33
...
Z        2006-05-01 00:00:00.000 55

sum
-----------
215

avg         sum
----------- -----------
54          867
```

Notice that whereas the other aggregate functions did not supply a column header name and thus needed an alias to identify the column in the result set, the `compute` function provides a column name.

Building Crosstab Queries

While an aggregate query can `group by` multiple columns, the result is still columnar and less than perfect for scanning numbers quickly. The cross-tabulation, or crosstab, query pivots one `group by` column (or dimension) counterclockwise 90 degrees and turns it into the result set's columns, as shown in Figure 11-4. The limitation, of course, is that while a columnar `group by` query can have multiple aggregate functions, a crosstab query can display but a single measure.

The term *crosstab query* describes the result set, not the method of creating the crosstab, because there are multiple programmatic methods of generating a crosstab query — some better than others. The following sections describe several methods for creating the same result.

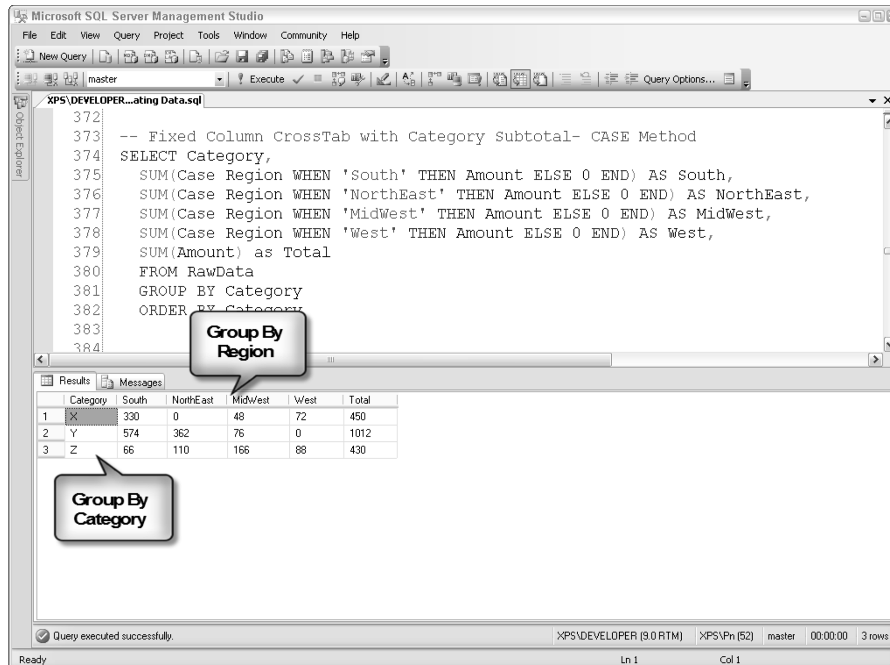


Figure 11-4: Pivoting a group by column creates a crosstab query.

Fixed-Column Crosstab Queries

There are three methods of generating a crosstab query with known, fixed columns using SQL Server 2005. While the columns for every crosstab query may not be known at development time, if the columns can be determined, then there's value in knowing the columns — coding the application forms and reports will be significantly easier with known columns.

Correlated Subquery Method

The first T-SQL method is a common solution, but the performance it offers is poor compared to the other methods and I don't recommend it. However, it is still useful to understand the requirements of creating a crosstab query result set.

The basic idea is that a subquery is executed for every instance of every measure for every pivoted `group by` column. To walk though this query, visualize each row from the `select` statement. The first column is the `category` from the `RawData` table. The `South`, `NorthEast`, `Midwest`, and `West` columns are all correlated subqueries that `sum()` the `Amount` column filtered by the region and the row's category. If the crosstab query had 1,000 rows and a crosstab column for every week of the year, SQL Server would execute 52,000 subqueries with this method.

The last column adds a subtotal for each category:

```
SELECT R.Category,
    (SELECT SUM(Amount)
      FROM RawData
      WHERE Region = 'South' AND Category = R.Category) AS 'South',
    (SELECT SUM(Amount)
      FROM RawData
      WHERE Region = 'NorthEast' AND Category = R.Category) AS 'NorthEast',
    (SELECT SUM(Amount)
      FROM RawData
      WHERE Region = 'MidWest' AND Category = R.Category) AS 'MidWest',
    (SELECT SUM(Amount)
      FROM RawData
      WHERE Region = 'West' AND Category = R.Category) AS 'West',
    SUM(Amount) as Total
  FROM RawData R
  GROUP BY Category
```

Result:

```
Category South   NorthEast   MidWest   West    Total
-------- ------- ----------- --------- ------- --------
X        165     NULL        24        36      225
Y        287     181         38        NULL    506
Z        33      55          83        44      215
```

Case Method

Rather than filter within a correlated subquery for each instance of a measure, this method uses a `case` expression to filter the data summed by the aggregate so the query engine can process the entire crosstab query as a single set-based operation. That's why this method performs so well and is by far the most popular method of calculating a crosstab query.

To walk through this query, data `from` the `RawData` table is not restricted by any `where` clause. The `group by` clause partitions the data set by categories. The aggregate functions are then calculated to create a single result row for each category. And here's the trick for this query: The `sum()` includes a `case` expression so each column sees only the amount for each region:

```
SELECT Category,
  SUM(Case Region WHEN 'South' THEN Amount ELSE 0 END) AS South,
  SUM(Case Region WHEN 'NorthEast' THEN Amount ELSE 0 END) AS NorthEast,
  SUM(Case Region WHEN 'MidWest' THEN Amount ELSE 0 END) AS MidWest,
  SUM(Case Region WHEN 'West' THEN Amount ELSE 0 END) AS West,
  SUM(Amount) as Total
  FROM RawData
  GROUP BY Category
  ORDER BY Category
```

The result is the same as the correlated subquery method.

Pivot Method

The pivot method deviates from the normal query flow by performing the aggregate function, creating the crosstab results as a data source within the `from` clause.

SQL Server 2005 includes the new Pivot command, designed to ease the creation of crosstab queries. Together with its corollary command, UnPivot, these two commands also ease normalizing and denormalizing data.

If you think of pivot as a table-valued function that's used as a data source, it accepts two parameters. The first parameter is the aggregate function for the crosstab's values. The second measure lists the pivoted columns — in this example, the aggregate function sums the Amount column, and the pivoted columns are the regions. Because pivot is part of the from clause, the data set needs a named range or alias:

```
SELECT Category, SalesDate, South, NorthEast, MidWest, West
  FROM RawData
    PIVOT
      (Sum (Amount)
      FOR Region IN (South, NorthEast, MidWest, West)
      ) AS pt
```

Result:

```
Category SalesDate               South       NorthEast   MidWest     West
-------- ----------------------- ----------- ----------- ----------- -----------
Y        2005-11-01 00:00:00.000 36          NULL        NULL        NULL
Y        2005-12-01 00:00:00.000 15          28          NULL        NULL
X        2006-01-01 00:00:00.000 11          NULL        24          NULL
X        2006-02-01 00:00:00.000 NULL        NULL        NULL        36
Y        2006-02-01 00:00:00.000 47          NULL        NULL        NULL
Y        2006-03-01 00:00:00.000 NULL        62          38          NULL
Z        2006-04-01 00:00:00.000 33          NULL        83          NULL
Z        2006-05-01 00:00:00.000 NULL        55          NULL        44
X        2006-06-01 00:00:00.000 154         NULL        NULL        NULL
Y        2006-07-01 00:00:00.000 117         NULL        NULL        NULL
Y        2006-08-01 00:00:00.000 72          91          NULL        NULL
```

The result is not what was expected. The pivot command used every column. Because the Amount and Region are specified, it assumed that every remaining column should be used for the group by, and grouped by Category and SalesDate.

The solution is to use a subquery to select only the columns that should be submitted to the pivot command

```
SELECT Category, South, NorthEast, MidWest, West
  FROM (Select Category, Region, Amount from RawData) sq
    PIVOT
      (Sum (Amount)
      FOR Region IN (South, NorthEast, MidWest, West)
      ) AS pt
```

The result is the same as the previous crosstab queries without the unneeded SalesDate column.

The final fixed-column crosstab pivot query method is the polished example. Because the pivot is logically part of the from clause, it occurs prior to the where clause. Therefore, another reason to use a subquery to prepare the data for the pivot command is to filter the data or join with other data sources.

The next example filters the data where `category = 'z'` and adds the category total column so the `pivot` command creates the same result as the previous `case` method:

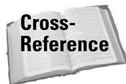
```
SELECT Category, South, NorthEast, MidWest, West,
  IsNull(South,0) + IsNull(NorthEast,0) + IsNull(MidWest,0) +
IsNull(West,0) as Total
  FROM (Select Region, Category, Amount
          From RawData
          Where Category = 'Z') sq
    PIVOT
      (Sum (Amount)
      FOR Region IN (South, NorthEast, MidWest, West)
      ) AS pt
```

The result is the same as the previous correlated subquery and `case` crosstab queries.

Dynamic Crosstab Queries

The rows of a crosstab query are automatically dynamically generated by the aggregation; however, in every one of the previous crosstab methods, the crosstab columns (`region` in this example) must be hard-coded in the SQL statement. The only way to create a crosstab query with dynamic columns is to use a SQL batch (possible saved as a stored procedure or user-defined function) to determine the columns at execution time and assemble a dynamic SQL command to execute the crosstab query.

Traditionally, cursors have been used to brute-force through the data, or to assemble the columns so that dynamic SQL can execute the dynamic crosstab query. Using the `pivot` command with the multiple-assignment variable `select` (described later in the chapter) makes quick work of a dynamic crosstab query.



An Analysis Services cube is basically a dynamic crosstab query on steroids. For more about designing these high-performance interactive cubes, turn to Chapter 43, "Business Intelligence with Analysis Services."

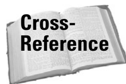
Cursor and Pivot Method

The goal of the cursor is to iterate through the `distinct` regions and assemble a comma-delimited string in the `@RegionColumn` variable. Once the trailing comma is removed from `@RegionColumn`, it's then used as part of a `pivot` command in a dynamic SQL statement, which is executed using `sp_executesql`:

```
DECLARE
  @SQLStr NVARCHAR(1024),
  @RegionColumn VARCHAR(50),
  @SemiColon BIT
SET @Semicolon = 0
SET @SQLStr = ''
DECLARE ColNames CURSOR FAST_FORWARD
  FOR
  SELECT DISTINCT Region as [Column]
    FROM RawData
    ORDER BY Region
```

```
  OPEN ColNames
  FETCH ColNames INTO @RegionColumn
  WHILE @@Fetch_Status = 0
    BEGIN
        SET @SQLStr = @SQLStr + @RegionColumn + ', '
        FETCH ColNames INTO @RegionColumn  -- fetch next
    END
  CLOSE ColNames
DEALLOCATE ColNames
SET @SQLStr = Left(@SQLStr, Len(@SQLStr) - 1)
SET @SQLStr = 'SELECT Category, '
    + @SQLStr
    + ' FROM RawData PIVOT (Sum (Amount) FOR Region IN ('
    + @SQLStr
    + ')) AS pt'
PRINT @SQLStr
EXEC sp_executesql  @SQLStr
```

The result is the same as the crosstab results in previous examples, but if there were additional regions, they would appear as new columns in the crosstab query.



Because cursors tend to scale poorly, using a cursor is rarely a good practice, and using a cursor to create a crosstab query is no exception. In my tests, the cursor method performed about ten times slower than the following set-based multiple-assignment-variable method. Nevertheless, for more information about how to code a cursor, see Chapter 20, "Kill the Cursor!"

Multiple Assignment Variable and Pivot Method

This method uses the exact same dynamic SQL string to execute the dynamic crosstab query, but employs a set-based `select` statement to create the list of regions, instead of a cursor. The `select` statement appends every region to a comma-delimited list, which is then executed in the same manner as the preceding example:

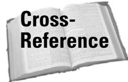
```
DECLARE @XColumns NVARCHAR(1024)
SET @XColumns = ''
SELECT @XColumns = @XColumns + [a].[Column] + ', '
  FROM
    (SELECT DISTINCT Region as [Column]
      FROM RawData) as a

SET @XColumns = Left(@XColumns, Len(@XColumns) - 1)

SET @XColumns = 'SELECT Category, '
    + @XColumns
    + ' FROM RawData PIVOT (Sum (Amount) FOR Region IN ('
    + @XColumns
    + ')) AS pt'
PRINT @XColumns

EXEC sp_executesql @XColumns
```

The job of the dynamic query code is to assemble the fixed-code crosstab query without specifying the `X`, or column, values. The subquery returns a list of `X` values. The recursive `select` variable appends the values, along with the other text required to build the dynamic crosstab query, to the `@XColumns` variable. The final `set` statement builds the completed dynamic SQL string:



The inverse of a crosstab query is the `unpivot` command, which is extremely useful for normalizing denormalized data. For an explanation of the `unpivot` command and examples, turn to Chapter 24, "Exploring Advanced T-SQL Techniques."

Summary

SQL Server excels in aggregate functions, with the proverbial rich suite of features; and it is very capable of calculating sums and aggregates to suit nearly any need. From the simple `count()` aggregate function to the complex dynamic crosstab query and the new `pivot` command, these query methods enable you to create powerful data analysis queries for impressive reports.

The next chapter examines hierarchical data — to continue the theme of working with the `select` statement to accomplish increasingly complex tasks.