5.5 Advanced Aggregation Features

The aggregation support in SQL is quite powerful and handles most common tasks with ease. However, there are some tasks that are hard to implement efficiently with the basic aggregation features. In this section, we study features in SQL to handle some such tasks.

5.5.1 Ranking

Finding the position of a value within a set is a common operation. For instance, we may wish to assign students a rank in class based on their grade-point average (GPA), with the rank 1 going to the student with the highest GPA, the rank 2 to the student with the next highest GPA, and so on. A related type of query is to find the percentile in which a value in a (multi)set belongs, for example, the bottom third, middle third, or top third. While such queries can be expressed using the SQL constructs we have seen so far, they are difficult to express and inefficient to evaluate. Programmers may resort to writing the query partly in SQL and partly in a programming language. We study SQL support for direct expression of these types of queries here.

In our university example, the takes relation shows the grade each student earned in each course taken. To illustrate ranking, let us assume we have a view student_grades (ID, GPA) giving the grade-point average of each student.¹⁰

Ranking is done with an order by specification. The following query gives the rank of each student:

```
select ID, rank() over (order by (GPA) desc) as s_rank
from student_grades;
```

Note that the order of tuples in the output is not defined, so they may not be sorted by rank. An extra order by clause is needed to get them in sorted order, as follows:

```
select ID, rank() over (order by (GPA) desc) as s_rank
from student_grades
order by s_rank;
```

A basic issue with ranking is how to deal with the case of multiple tuples that are the same on the ordering attribute(s). In our example, this means deciding what to do if there are two students with the same GPA. The rank function gives the same rank to all tuples that are equal on the order by attributes. For instance, if the highest GPA is shared by two students, both would get rank 1. The next rank given would be 3, not 2, so if three students get the next highest GPA, they would all get rank 3, and the next

¹⁰The SQL statement to create the view student_grades is somewhat complex since we must convert the letter grades in the takes relation to numbers and weight the grades for each course by the number of credits for that course. The definition of this view is the goal of Exercise 4.6.
student(s) would get rank 6, and so on. There is also a dense_rank function that does not create gaps in the ordering. In the preceding example, the tuples with the second highest value all get rank 2, and tuples with the third highest value get rank 3, and so on.

If there are null values among the values being ranked, they are treated as the highest values. That makes sense in some situations, although for our example, it would result in students with no courses being shown as having the highest GPAs. Thus, we see that care needs to be taken in writing ranking queries in cases where null values may appear. SQL permits the user to specify where they should occur by using nulls first or nulls last, for instance:

```sql
select ID, rank () over (order by GPA desc nulls last) as s_rank
from student_grades;
```

It is possible to express the preceding query with the basic SQL aggregation functions, using the following query:

```sql
select ID, (1 + (select count(*)
from student_grades B
where B.GPA > A.GPA)) as s_rank
from student_grades A
order by s_rank;
```

It should be clear that the rank of a student is merely 1 plus the number of students with a higher GPA, which is exactly what the query specifies. However, this computation of each student's rank takes time linear in the size of the relation, leading to an overall time quadratic in the size of the relation. On large relations, the above query could take a very long time to execute. In contrast, the system's implementation of the rank clause can sort the relation and compute the rank in much less time.

Ranking can be done within partitions of the data. For instance, suppose we wish to rank students by department rather than across the entire university. Assume that a view is defined like student_grades but including the department name: dept_grades(ID, dept_name, GPA). The following query then gives the rank of students within each section:

```sql
select ID, dept_name,
rank () over (partition by dept_name order by GPA desc) as dept_rank
from dept_grades
order by dept_name, dept_rank;
```

---

11 There is a slight technical difference if a student has not taken any courses and therefore has a null GPA. Due to how comparisons of null values work in SQL, a student with a null GPA does not contribute to other students' count values.
The outer order by clause orders the result tuples by department name, and within each department by the rank.

Multiple rank expressions can be used within a single select statement; thus, we can obtain the overall rank and the rank within the department by using two rank expressions in the same select clause. When ranking (possibly with partitioning) occurs along with a group by clause, the group by clause is applied first, and partitioning and ranking are done on the results of the group by. Thus, aggregate values can then be used for ranking.

It is often the case, especially for large results, that we may be interested only in the top-ranking tuples of the result rather than the entire list. For rank queries, this can be done by nesting the ranking query within a containing query whose where clause chooses only those tuples whose rank is lower than some specified value. For example, to find the top 5 ranking students based on GPA we could extend our earlier example by writing:

```
select *
from (select ID, rank() over (order by (GPA) desc) as s_rank
       from student_grades)
where s_rank <= 5;
```

This query does not necessarily give 5 students, since there could be ties. For example, if 2 students tie for fifth, the result would contain a total of 6 tuples. Note that the bottom n is simply the same as the top n with a reverse sorting order.

Several database systems provide nonstandard SQL syntax to specify directly that only the top n results are required. In our example, this would allow us to find the top 5 students without the need to use the rank function. However, those constructs result in exactly the number of tuples specified (5 in our example), and so ties for the final position are broken arbitrarily. The exact syntax for these “top n” queries varies widely among systems; see Note 5.4 on page 222. Note that the top n constructs do not support partitioning; so we cannot get the top n within each partition without performing ranking.

Several other functions can be used in place of rank. For instance, percent_rank of a tuple gives the rank of the tuple as a fraction. If there are n tuples in the partition and the rank of the tuple is r, then its percent rank is defined as \((r - 1)/(n - 1)\) (and as null if there is only one tuple in the partition). The function cume_dist, short for cumulative distribution, for a tuple is defined as \(p/n\) where p is the number of tuples in the partition with ordering values preceding or equal to the ordering value of the tuple and n is the number of tuples in the partition. The function row_number sorts the rows and gives each row a unique number corresponding to its position in the sort order; different rows with the same ordering value would get different row numbers, in a nondeterministic fashion.

---

12The entire set is treated as a single partition if no explicit partition is used.
Often, only the first few tuples of a query result are required. This may occur in a ranking query where only top-ranked results are of interest. Another case where this may occur is in a query with an order by from which only the top values are of interest. Restricting results to the top-ranked results can be done using the rank function as we saw earlier, but that syntax is rather cumbersome. Many databases support a simpler syntax for such restriction, but the syntax varies widely among the leading database systems. We provide a few examples here.

Some systems (including MySQL and PostgreSQL) allow a clause limit $n$ to be added at the end of an SQL query to specify that only the first $n$ tuples should be output. This clause can be used in conjunction with an order by clause to fetch the top $n$ tuples, as illustrated by the following query, which retrieves the ID and GPA of the top 10 students in order of GPA:

```sql
select ID, GPA
from student_grades
order by GPA desc
limit 10;
```

In IBM DB2 and the most recent versions of Oracle, the equivalent of the limit clause is fetch first 10 rows only. Microsoft SQL Server places its version of this feature in the select clause rather than adding a separate limit clause. The select clause is written as: select top 10 ID, GPA.

Oracle (both current and older versions) offers the concept of a row number to provide this feature. A special, hidden attribute rownum numbers tuples of a result relation in order of retrieval. This attribute can then be used in a where clause within a containing query. However, the use of this feature is a bit tricky, since the rownum is decided before rows are sorted by an order by clause. To use it properly, a nested query should be used as follows:

```sql
select *
from (select ID, GPA
      from student_grades
      order by GPA desc)
where rownum <= 10;
```

The nested query ensures that the predicate on rownum is applied only after the order by is applied.

Some database systems have features allowing tuple limits to be exceeded in case of ties. See your system’s documentation for details.
Finally, for a given constant $n$, the ranking function $\text{ntile}(n)$ takes the tuples in each partition in the specified order and divides them into $n$ buckets with equal numbers of tuples.\footnote{If the total number of tuples in a partition is not divisible by $n$, then the number of tuples in each bucket can differ by at most 1. Tuples with the same value for the ordering attribute may be assigned to different buckets, nondeterministically, in order to make the number of tuples in each bucket equal.} For each tuple, $\text{ntile}(n)$ then gives the number of the bucket in which it is placed, with bucket numbers starting with 1. This function is particularly useful for constructing histograms based on percentiles. We can show the quartile into which each student falls based on GPA by the following query:

\[
\text{select } ID, \text{ntile}(4) \text{ over (order by } (\text{GPA desc}) \text{) as quartile}
\from student\_grades;
\]

### 5.5.2 Windowing

Window queries compute an aggregate function over ranges of tuples. This is useful, for example, to compute an aggregate of a fixed range of time; the time range is called a window. Windows may overlap, in which case a tuple may contribute to more than one window. This is unlike the partitions we saw earlier, where a tuple could contribute to only one partition.

An example of the use of windowing is trend analysis. Consider our earlier sales example. Sales may fluctuate widely from day to day based on factors like weather (e.g., a snowstorm, flood, hurricane, or earthquake might reduce sales for a period of time). However, over a sufficiently long period of time, fluctuations might be less (continuing the example, sales may “make up” for weather-related downturns). Stock-market trend analysis is another example of the use of the windowing concept. Various “moving averages” are found on business and investment web sites.

It is relatively easy to write an SQL query using those features we have already studied to compute an aggregate over one window, for example, sales over a fixed 3-day period. However, if we want to do this for every 3-day period, the query becomes cumbersome.

SQL provides a windowing feature to support such queries. Suppose we are given a view $\text{tot\_credits}(\text{year, num\_credits})$ giving the total number of credits taken by students in each year.\footnote{We leave the definition of this view in terms of our university example as an exercise.} Note that this relation can contain at most one tuple for each year. Consider the following query:

\[
\text{select } year, \text{avg(num\_credits)}
\over (\text{order by } year \text{ rows } 3 \text{ preceding})
\as \text{avg\_total\_credits}
\from tot\_credits;
\]
This query computes averages over the three preceding tuples in the specified sort order. Thus, for 2019, if tuples for years 2018 and 2017 are present in the relation \textit{tot\_credits}, since each year is represented by only one tuple, the result of the window definition is the average of the values for years 2017, 2018, and 2019. The averages each year would be computed in a similar manner. For the earliest year in the relation \textit{tot\_credits}, the average would be over only that year itself, while for the next year, the average would be over 2 years. Note that this example makes sense only because each year appears only once in \textit{tot\_weight}. Were this not the case, then there would be several possible orderings of tuples since tuples for the same year could be in any order. We shall see shortly a windowing query that uses a range of values instead of a specific number of tuples.

Suppose that instead of going back a fixed number of tuples, we want the window to consist of all prior years. That means the number of prior years considered is not fixed. To get the average total credits over all prior years, we write:

\[
\text{select year, avg(num\_credits) over (order by year rows unbounded preceding) as avg\_total\_credits from tot\_credits;}
\]

It is possible to use the keyword \texttt{following} in place of \texttt{preceding}. If we did this in our example, the \texttt{year} value specifies the beginning of the window instead of the end. Similarly, we can specify a window beginning before the current tuple and ending after it:

\[
\text{select year, avg(num\_credits) over (order by year rows between 3 preceding and 2 following) as avg\_total\_credits from tot\_credits;}
\]

In our example, all tuples pertain to the entire university. Suppose instead we have credit data for each department in a view \textit{tot\_credits\_dept} \((dept\_name, year, num\_credits)\) giving the total number of credits students took with the particular department in the specified year. (Again, we leave writing this view definition as an exercise.) We can write windowing queries that treat each department separately by partitioning by \texttt{dept\_name}:

\[
\text{select dept\_name, year, avg(num\_credits) over (partition by dept\_name order by year rows between 3 preceding and current row) as avg\_total\_credits from tot\_credits\_dept;}
\]
<table>
<thead>
<tr>
<th>item_name</th>
<th>color</th>
<th>clothes_size</th>
<th>quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>dress</td>
<td>dark</td>
<td>small</td>
<td>2</td>
</tr>
<tr>
<td>dress</td>
<td>dark</td>
<td>medium</td>
<td>6</td>
</tr>
<tr>
<td>dress</td>
<td>dark</td>
<td>large</td>
<td>12</td>
</tr>
<tr>
<td>dress</td>
<td>pastel</td>
<td>small</td>
<td>4</td>
</tr>
<tr>
<td>dress</td>
<td>pastel</td>
<td>medium</td>
<td>3</td>
</tr>
<tr>
<td>dress</td>
<td>pastel</td>
<td>large</td>
<td>3</td>
</tr>
<tr>
<td>dress</td>
<td>white</td>
<td>small</td>
<td>2</td>
</tr>
<tr>
<td>dress</td>
<td>white</td>
<td>medium</td>
<td>3</td>
</tr>
<tr>
<td>dress</td>
<td>white</td>
<td>large</td>
<td>0</td>
</tr>
<tr>
<td>pants</td>
<td>dark</td>
<td>small</td>
<td>14</td>
</tr>
<tr>
<td>pants</td>
<td>dark</td>
<td>medium</td>
<td>6</td>
</tr>
<tr>
<td>pants</td>
<td>dark</td>
<td>large</td>
<td>0</td>
</tr>
<tr>
<td>pants</td>
<td>pastel</td>
<td>small</td>
<td>1</td>
</tr>
<tr>
<td>pants</td>
<td>pastel</td>
<td>medium</td>
<td>0</td>
</tr>
<tr>
<td>pants</td>
<td>pastel</td>
<td>large</td>
<td>1</td>
</tr>
<tr>
<td>pants</td>
<td>white</td>
<td>small</td>
<td>3</td>
</tr>
<tr>
<td>pants</td>
<td>white</td>
<td>medium</td>
<td>0</td>
</tr>
<tr>
<td>pants</td>
<td>white</td>
<td>large</td>
<td>2</td>
</tr>
<tr>
<td>shirt</td>
<td>dark</td>
<td>small</td>
<td>2</td>
</tr>
<tr>
<td>shirt</td>
<td>dark</td>
<td>medium</td>
<td>6</td>
</tr>
<tr>
<td>shirt</td>
<td>dark</td>
<td>large</td>
<td>6</td>
</tr>
<tr>
<td>shirt</td>
<td>pastel</td>
<td>small</td>
<td>4</td>
</tr>
<tr>
<td>shirt</td>
<td>pastel</td>
<td>medium</td>
<td>1</td>
</tr>
<tr>
<td>shirt</td>
<td>pastel</td>
<td>large</td>
<td>2</td>
</tr>
<tr>
<td>shirt</td>
<td>white</td>
<td>small</td>
<td>17</td>
</tr>
<tr>
<td>shirt</td>
<td>white</td>
<td>medium</td>
<td>1</td>
</tr>
<tr>
<td>shirt</td>
<td>white</td>
<td>large</td>
<td>10</td>
</tr>
<tr>
<td>skirt</td>
<td>dark</td>
<td>small</td>
<td>2</td>
</tr>
<tr>
<td>skirt</td>
<td>dark</td>
<td>medium</td>
<td>5</td>
</tr>
<tr>
<td>skirt</td>
<td>dark</td>
<td>large</td>
<td>1</td>
</tr>
<tr>
<td>skirt</td>
<td>pastel</td>
<td>small</td>
<td>11</td>
</tr>
<tr>
<td>skirt</td>
<td>pastel</td>
<td>medium</td>
<td>9</td>
</tr>
<tr>
<td>skirt</td>
<td>pastel</td>
<td>large</td>
<td>15</td>
</tr>
<tr>
<td>skirt</td>
<td>white</td>
<td>small</td>
<td>2</td>
</tr>
<tr>
<td>skirt</td>
<td>white</td>
<td>medium</td>
<td>5</td>
</tr>
<tr>
<td>skirt</td>
<td>white</td>
<td>large</td>
<td>3</td>
</tr>
</tbody>
</table>

Figure 5.17 An example of sales relation.
The use of the keyword range in place of row allows the windowing query to cover all tuples with a particular value rather than covering a specific number of tuples. Thus for example, rows current row refers to exactly one tuple, while range current row refers to all tuples whose value for the sort attribute is the same as that of the current tuple. The range keyword is not implemented fully in every system.\footnote{Some systems, such as PostgreSQL, allow range only with unbounded.}

5.5.3 Pivoting

Consider an application where a shop wants to find out what kinds of clothes are popular. Let us suppose that clothes are characterized by their item\_name, color, and size, and that we have a relation sales with the schema.

\[ sales (item\_name, color, clothes\_size, quantity) \]

Suppose that item\_name can take on the values (skirt, dress, shirt, pants), color can take on the values (dark, pastel, white), clothes\_size can take on values (small, medium, large), and quantity is an integer value representing the total number of items sold of a given (item\_name, color, clothes\_size) combination. An instance of the sales relation is shown in Figure 5.17.

Figure 5.18 shows an alternative way to view the data that is present in Figure 5.17; the values “dark”, “pastel”, and “white” of attribute color have become attribute names in Figure 5.18. The table in Figure 5.18 is an example of a cross-tabulation (or cross-tab, for short), also referred to as a pivot-table.

The values of the new attributes dark, pastel and white in our example are defined as follows. For a particular combination of item\_name, clothes\_size (e.g., (“dress”, “dark”))
if there is a single tuple with color value “dark”, the quantity value of that attribute appears as the value for the attribute dark. If there are multiple such tuples, the values are aggregated using the sum aggregate in our example; in general other aggregate functions could be used instead. Values for the other two attributes, pastel and white, are similarly defined.

In general, a cross-tab is a table derived from a relation (say, R), where values for some attribute of relation R (say, A) become attribute names in the result; the attribute A is the pivot attribute. Cross-tabs are widely used for data analysis, and are discussed in more detail in Section 11.3.

Several SQL implementations, such as Microsoft SQL Server, and Oracle, support a pivot clause that allows creation of cross-tabs. Given the sales relation from Figure 5.17, the query:

```sql
select *
from sales
pivot (sum(quantity)
for color in ('dark', 'pastel', 'white'))
```

returns the result shown in Figure 5.18.

Note that the for clause within the pivot clause specifies (i) a pivot attribute (color, in the above query), (ii) the values of that attribute that should appear as attribute names in the pivot result (dark, pastel and white, in the above query), and (iii) the aggregate function that should be used to compute the value of the new attributes (aggregate function sum, on the attribute quantity, in the above query).

The attribute color and quantity do not appear in the result, but all other attributes are retained. In case more than one tuple contributes values to a given cell, the aggregate operation within the pivot clause specifies how the values should be combined. In the above example, the quantity values are aggregated using the sum function.

A query using pivot can be written using basic SQL constructs, without using the pivot construct, but the construct simplifies the task of writing such queries.

### 5.5.4 Rollup and Cube

SQL supports generalizations of the group by construct using the rollup and cube operations, which allow multiple group by queries to be run in a single query, with the result returned as a single relation.

Consider again our retail shop example and the relation:

```
sales (item_name, color, clothes_size, quantity)
```

We can find the number of items sold in each item name by writing a simple group by query:
```sql
select item_name, sum(quantity) as quantity
from sales
group by item_name;
```

Similarly, we can find the number of items sold in each color, and each size. We can further find a breakdown of sales by item-name and color by writing:

```sql
select item_name, color, sum(quantity) as quantity
from sales
group by item_name, color;
```

Similarly, a query with `group by item_name, color, clothes_size` would allow us to see the sales breakdown by `(item_name, color, clothes_size)` combinations.

Data analysts often need to view data aggregated in multiple ways as illustrated above. The SQL `rollup` and `cube` constructs provide a concise way to get multiple such aggregates using a single query, instead of writing multiple queries.

The `rollup` construct is illustrated using the following query:

```sql
select item_name, color, sum(quantity)
from sales
group by rollup(item_name, color);
```

The result of the query is shown in Figure 5.19. The above query is equivalent to the following query using the `union` operation.

```sql
(select item_name, color, sum(quantity) as quantity
from sales
group by item_name, color)
union
(select item_name, null as color, sum(quantity) as quantity
from sales
group by item_name)
union
(select null as item_name, null as color, sum(quantity) as quantity
from sales)
```

The construct `group by rollup(item_name, color)` generates 3 groupings:

```sql
{ (item_name, color), (item_name), () }
```

where () denotes an empty `group by` list. Observe that a grouping is present for each prefix of the attributes listed in the `rollup` clause, including the empty prefix. The query result contains the union of the results by these groupings. The different groupings generate different schemas; to bring the results of the different groupings to a common
Figure 5.19 Query result: group by rollup (item_name, color).

schema, tuples in the result contain null as the value of those attributes not present in a particular grouping.\textsuperscript{16}

The cube construct generates an even larger number of groupings, consisting of all subsets of the attributes listed in the cube construct. For example, the query:

\begin{verbatim}
select item_name, color, clothes_size, sum(quantity)
from sales
group by cube(item_name, color, clothes_size);
\end{verbatim}

generates the following groupings:

\[
\{(item_name, color, clothes_size), (item_name, color), (item_name, clothes_size),
\color{white}{(color, clothes_size), (item_name), (color), (clothes_size), (}\}
\]

To bring the results of the different groupings to a common schema, as with rollup, tuples in the result contain null as the value of those attributes not present in a particular grouping.

\textsuperscript{16}The SQL outer union operation can be used to perform a union of relations that may not have a common schema. The resultant schema has the union of all the attributes across the inputs; each input tuple is mapped to an output tuple by adding all the attributes missing in that tuple, with the value set to null. Our union query can be written using outer union, and in that case we do not need to explicitly generate null-value attributes using null as attribute-name constructs, as we have done in the above query.
Multiple rollups and cubes can be used in a single group by clause. For instance, the following query:

```sql
select item_name, color, clothes_size, sum(quantity)
from sales
group by rollup(item_name), rollup(color, clothes_size);
```

generates the groupings:

\[
\{ (item_name, color, clothes_size), (item_name, color), (item_name),
   (color, clothes_size), (color), () \}
\]

To understand why, observe that \texttt{rollup(item_name)} generates a set of two groupings, \{(item_name), ()\}, while \texttt{rollup(color, clothes_size)} generates a set of three groupings, \{(color, clothes_size), (color), ()\}. The Cartesian product of the two sets gives us the six groupings shown.

Neither the \texttt{rollup} nor the \texttt{cube} clause gives complete control on the groupings that are generated. For instance, we cannot use them to specify that we want only groupings \{(color, clothes_size), (clothes_size, item_name)\}. Such restricted groupings can be generated by using the \texttt{grouping sets} construct, in which one can specify the specific list of groupings to be used. To obtain only groupings \{(color, clothes_size), (clothes_size, item_name)\}, we would write:

```sql
select item_name, color, clothes_size, sum(quantity)
from sales
group by grouping sets ((color, clothes_size), (clothes_size, item_name));
```

Analysts may want to distinguish those nulls generated by \texttt{rollup} and \texttt{cube} operations from “normal” nulls actually stored in the database or arising from an outer join. The \texttt{grouping()} function returns 1 if its argument is a null value generated by a \texttt{rollup} or \texttt{cube} and 0 otherwise (note that the \texttt{grouping} function is different from the \texttt{grouping sets} construct). If we wish to display the \texttt{rollup} query result shown in Figure 5.19, but using the value “all” in place of nulls generated by \texttt{rollup}, we can use the query:

```sql
select (case when grouping(item_name) = 1 then 'all'
             else item_name end) as item_name,
       (case when grouping(color) = 1 then 'all'
             else color end) as color,
       sum(quantity) as quantity
from sales
group by rollup(item_name, color);
```

One might consider using the following query using \texttt{coalesce}, but it would incorrectly convert null item names and colors to all:
select coalesce (item_name,'all') as item_name,
        coalesce (color,'all') as color,
        sum(quantity) as quantity
from sales
group by rollup(item_name, color);

5.6 Summary

- SQL queries can be invoked from host languages via embedded and dynamic SQL. The ODBC and JDBC standards define application program interfaces to access SQL databases from C and Java language programs.
- Functions and procedures can be defined using SQL procedural extensions that allow iteration and conditional (if-then-else) statements.
- Triggers define actions to be executed automatically when certain events occur and corresponding conditions are satisfied. Triggers have many uses, such as business rule implementation and audit logging. They may carry out actions outside the database system by means of external language routines.
- Some queries, such as transitive closure, can be expressed either by using iteration or by using recursive SQL queries. Recursion can be expressed using either recursive views or recursive with clause definitions.
- SQL supports several advanced aggregation features, including ranking and windowing queries, as well as pivot, and rollup/cube operations. These simplify the expression of some aggregates and allow more efficient evaluation.

Review Terms

- JDBC
- Prepared statements
- SQL injection
- Metadata
- Updatable result sets
- Open Database Connectivity (ODBC)
- Embedded SQL
- Embedded database
- Stored procedures and functions
- Table functions.
- Parameterized views
- Persistent Storage Module (PSM).
- Exception conditions
- Handlers
- External language routines
- Sandbox
- Trigger
- Transitive closure
- Hierarchies