Data Collection and Modeling

Course 11 - SQL DML Part 2

SQL - GROUP BY (From previous course)

A clause that is used to generate summary rows based on grouping rows using identical values one or several columns; the result will contain one row per group (set of distinct values). The restriction is that any column in the SELECT list that is not part of an aggregate expression must be included in the GROUP BY list.

Usual aggregate functions used:

- COUNT()
- MIN(), MAX()
- SUM(), AVG()

SQL Aggregate Extensions

`GROUP BY` clause can be used to obtain aggregation based on a set of columns; if we need to compose a result set with similar columns from the same data source based on different lists, the easiest way is to use `UNION` to reunite all distinct result sets:

```
Students[id, name, group, country_origin, age]

-- return number of students from each group
SELECT group, count(id) FROM Students GROUP BY group;

-- return number of students from each country of origin
SELECT country_origin, count(id) FROM Students GROUP BY 1;

-- return number of students group and country of origin
SELECT group, country_origin, count(id) FROM Students GROUP BY
1,2;
```

SQL Aggregate Extensions

The previous result sets are not compatible as they have different number of columns and may contain columns with different data types; in order to make them compatible:

```
-- return number of students from each group
SELECT group, NULL, count(id) FROM Students GROUP BY 1
UNION ALL
-- return number of students from each country of origin
SELECT NULL, country_origin, count(id) FROM Students GROUP BY 2
UNION ALL
-- return number of students group and country of origin
SELECT group, country_origin, count(id) FROM Students GROUP BY
1,2;
```

SQL Aggregate Extensions

SQL `GROUPING SETS` is an extension of the `GROUP BY` clause that allows to specify multiple grouping sets within a single query. This feature enables performing aggregated calculations at different levels of granularity within the same result set. It is particularly useful when generating subtotals and totals for different combinations of columns.
The previous union of result sets can be solved in a more performant way with much less code using the `GROUPING SETS` clause:

```
SELECT group, country_origin, count(id) FROM Students GROUP BY GROUPING
SETS(
        (group, country_id),
        (group),
        (country_origin)
-- ,() if we want also a grand total
);
```

SQL Aggregate Extensions

In order to see if a row is produced as a result of a specific grouping set, there is a function, `GROUPING`, that returns 0 if the argument is member of the current grouping set and 1 otherwise:

```
SELECT
GROUPING(group) uses_group,
group, country_origin, count(id) FROM Students GROUP BY
GROUPING SETS(
       (group, country_id),
       (group),
       (country_origin)
);
```

SQL Aggregate Extensions

The `GROUPING` function can be also used in the `HAVING` clause to filter the result set:

```
SELECT group, country_origin, count(id) FROM Students GROUP
BY GROUPING SETS(
       (group, country_id),
       (group),
       (country_origin)
HAVING GROUPING(group)=0
);
```

SQL Aggregate Extensions

The `GROUP BY` clause has another additional clause that can define several grouping sets as a hierarchy (that makes sense) between lists of columns.
`GROUP BY ROLLUP` is a clause used to generate subtotals and grand totals for a set of columns in a result set. It is an extension of the `GROUP BY` clause useful for creating summary reports with hierarchical aggregates. The `ROLLUP` operation generates a result set that includes not only the usual grouping, but also additional rows that represent various levels of subtotal and grand total.

```
SELECT group, country_origin, count(id) FROM Students GROUP BY
ROLLUP(group, country_id);
```

It will generate groups for the next sets:

```
(group, country_origin)
(group)
()
```

SQL Aggregate Extensions

`CUBE` is a subclause of `GROUP BY` clause that allows the generation of multiple grouping sets based on all possible grouping sets specified by columns list:

```
SELECT group, country_origin, count(id) FROM Students GROUP BY CUBE(group, country_id);
```

It will generate groups for the next sets:

```
(group, country_origin)
(group)
(country_origin)
()
```

SQL Window Functions

“A window function performs a calculation across a set of table rows that are somehow related to the current row. This is comparable to the type of calculation that can be done with an aggregate function. But unlike regular aggregate functions, use of a window function does not cause rows to become grouped into a single output row — the rows retain their separate identities. Behind the scenes, the window function is able to access more than just the current row of the query result.” (PostgreSQL documentation)

Unlike aggregate functions (e.g., `SUM`, `AVG`, `COUNT`), which operate on entire groups of rows, window functions operate on a "window" of rows defined by an `OVER` clause.

SQL Window Functions

Aggregate functions aggregate data from each group of distinct values into a single row using `GROUP BY`.

Window functions operate in a similar fashion on each group of distinct values but preserves the initial number of rows from the group.

```
window_function(arg1, arg2,..) OVER (
   [PARTITION BY partition_expression]
   [ORDER BY sort_expression [ASC|DESC] [NULLS {FIRST|LAST}])
```

SQL Window Functions

Components:

`window_function`: The window function itself (e.g., `SUM`, `AVG`, `ROW_NUMBER`, etc.).

`PARTITION BY`: Divides the result set into partitions to which the window function is applied separately. It's an optional clause, and if omitted, the window function is applied to the entire result set.

`ORDER BY`: Specifies the order of rows within each partition.

`ROWS BETWEEN n PRECEDING AND m FOLLOWING`: Defines the window frame, indicating the range of rows to which the window function is applied relative to the current row.

SQL Window Functions

There are aggregate functions that can be used as window functions:

```
AVG(), COUNT(), MIN(), MAX(), SUM()
```

But there are also analytic functions specific to this context:

```
ROW_NUMBER()

RANK(), DENSE_RANK()

NTILE()

LEAD(), LAG()

FIRST_VALUE(), LAST_VALUE()
```

SQL Window Functions

```
Students[id, name, group, country_origin, finale_grade]

-- return the students with their position in their group
-- based on final grade
SELECT name, group, RANK() OVER (PARTITION BY group ORDER BY
finale_grade desc) as position
FROM Students;

SELECT name, group, DENSE_RANK() OVER (PARTITION BY group
ORDER BY finale_grade desc) as position
FROM Students;
```

SQL Window Functions

```
Students[id, name, group, country_origin, finale_grade]

-- return the students with their position in their group
-- based on final grade and the difference from 1st
SELECT name, group, RANK() OVER (PARTITION BY group ORDER BY
finale_grade desc) as position, FIRST_VALUE(final_grade) OVER
(PARTITION BY group ORDER BY finale_grade desc) -
finale_grade as difference
FROM Students;
```