Window (Analytical) Functions

Adriano dos Santos Fernandes

According to the SQL specification, window functions (also know as analytical functions) are a kind of aggregation, but one that does not “filter” the result set of a query. The rows of aggregated data are mixed with the query result set.

The window functions are used with the OVER clause. They may appear only in the SELECT list or the ORDER BY clause of a query.

Besides the OVER clause, Firebird window functions may be partitioned and ordered.

Syntax Pattern

```
<window function> ::= <window function name>([<expr> [, <expr> ...]]) OVER (
  [PARTITION BY <expr> [, <expr> ...]]
  [ORDER BY <expr>
    [<direction>]
    [<nulls placement>]
    [, <expr> [<direction>] [<nulls placement>] ...]
)

<direction> ::= {ASC | DESC}

<nulls placement> ::= NULLS {FIRST | LAST}
```

Aggregate functions used as window functions

All aggregate functions may be used as window functions, adding the OVER clause.

Imagine a table EMPLOYEE with columns ID, NAME and SALARY, and the need to show each employee with his respective salary and the percentage of his salary over the payroll.

A normal query could achieve this, as follows:

```
select
    id,
    department,
    salary,
    salary / (select sum(salary) from employee) percentage
  from employee
  order by id;
```

Results

```
id department salary percentage
-- ---------- ------ ----------
1  R & D       10.00     0.2040
2  SALES       12.00     0.2448
3  SALES        8.00     0.1632
4  R & D        9.00     0.1836
5  R & D       10.00     0.2040
```

The query is repetitive and lengthy to run, especially if EMPLOYEE happened to be a complex view.

The same query could be specified in a much faster and more elegant way using a window function:

```
select
    id,
    department,
    salary,
    salary / sum(salary) OVER () percentage
  from employee
  order by id;
```

Here, sum(salary) over () is computed with the sum of all SALARY from the query (the employee table).

back to top of page

Partitioning

Like aggregate functions, that may operate alone or in relation to a group, window functions may also operate on a group, which is called a “partition”.

Syntax Pattern

```
<window function>(...) OVER (PARTITION BY <expr> [, <expr> ...])
```

Aggregation over a group could produce more than one row, so the result set generated by a partition is joined with the main query using the same expression list as the partition.

Continuing the employee example, instead of getting the percentage of each employee's salary over the allemployees total, we would like to get the percentage based on just the employees in the same department:

```
select
    id,
    department,
    salary,
    salary / sum(salary) OVER (PARTITION BY department) percentage
  from employee
```

```
  order by id;
```

Results

```
id department salary percentage
-- ---------- ------ ----------
1  R & D       10.00     0.3448
2  SALES       12.00     0.6000
3  SALES        8.00     0.4000
4  R & D        9.00     0.3103
5  R & D       10.00     0.3448
```

back to top of page

Ordering

The ORDER BY sub-clause can be used with or without partitions and, with the standard aggregate functions, make them return the partial aggregations as the records are being processed.

Example

```
select
    id,
    salary,
    sum(salary) over (order by salary) cumul_salary
  from employee
  order by salary;
```

The result set produced:

```
id salary cumul_salary
-- ------ ------------
3   8.00         8.00
4   9.00        17.00
1  10.00        37.00
5  10.00        37.00
2  12.00        49.00
```

Then cumul_salary returns the partial/accumulated (or running) aggregation (of the SUM function). It may appear strange that 37.00 is repeated for the ids 1 and 5, but that is how it should work. The ORDER BY keys are grouped together and the aggregation is computed once (but summing the two 10.00). To avoid this, you can add the ID field to the end of the ORDER BY clause.

It's possible to use multiple windows with different orders, and ORDER BY parts like ASC/DESC and NULLS FIRST/LAST.

With a partition, ORDER BY works the same way, but at each partition boundary the aggregation is reset.

All aggregation functions, other than LIST(), are usable with ORDER BY.

back to top of page

Exclusive window functions

Beyond aggregate functions are the exclusive window functions, currently divided into ranking and navigational categories. Both sets can be used with or without partition and ordering, although the usage does not make much sense without ordering.

Ranking functions

The rank functions compute the ordinal rank of a row within the window partition. In this category are the functions DENSE_RANK, RANK and ROW_NUMBER.

Syntax

```
<ranking window function> ::=
    DENSE_RANK() |
    RANK() |
    ROW_NUMBER()
```

The ranking functions can be used to create different type of incremental counters. Consider SUM(1) OVER (ORDER BY SALARY) as an example of what they can do, each of them in a different way. The following is an example query, also comparing with the SUM behavior.

```
select
    id,
    salary,
    dense_rank() over (order by salary),
    rank() over (order by salary),
    row_number() over (order by salary),
    sum(1) over (order by salary)
  from employee
  order by salary;
```

The result set:

```
id salary dense_rank rank row_number sum
-- ------ ---------- ---- ---------- ---
3    8.00          1    1          1   1
4    9.00          2    2          2   2
1   10.00          3    3          3   4
5   10.00          3    3          4   4
2   12.00          4    5          5   5
```

The difference between DENSE_RANK and RANK is that there is a gap related to duplicate rows (relative to the window ordering) only in RANK. DENSE_RANK continues assigning sequential numbers after the duplicate salary. On the other hand, ROW_NUMBER always assigns sequential numbers, even when there are duplicate values.

back to top of page

Navigational functions

The navigational functions get the simple (non-aggregated) value of an expression from another row of the query, within the same partition.

Syntax

```
<navigational window function> ::=
    FIRST_VALUE(<expr>) |
    LAST_VALUE(<expr>) |
    NTH_VALUE(<expr>, <offset>) [FROM FIRST | FROM LAST] |
    LAG(<expr> [ [, <offset> [, <default> ] ] ) |
    LEAD(<expr> [ [, <offset> [, <default> ] ] )
```

Important to note:

FIRST_VALUE, LAST_VALUE and NTH_VALUE also operate on a window frame. Currently, Firebird always frames from the first to the current row of the partition, not to the last. This is likely to produce strange results for NTH_VALUE and especially LAST_VALUE.

Example

```
select
    id,
    salary,
    first_value(salary) over (order by salary),
    last_value(salary) over (order by salary),
    nth_value(salary, 2) over (order by salary),
    lag(salary) over (order by salary),
    lead(salary) over (order by salary)
  from employee
  order by salary;
```

The result set:

```
id salary first_value last_value nth_value    lag   lead
-- ------ ----------- ---------- --------- ------ ------
3    8.00        8.00       8.00    <null> <null>   9.00
4    9.00        8.00       9.00      9.00   8.00  10.00
1   10.00        8.00      10.00      9.00   9.00  10.00
5   10.00        8.00      10.00      9.00  10.00  12.00
```

```
2    12.00         8.00       12.00       9.00   10.00 <null>
```

FIRST_VALUE and LAST_VALUE get, respectively, the first and last value of the ordered partition.

NTH_VALUE gets the n-th value, starting from the first (default) or the last record, from the ordered partition. An offset of 1 from first would be equivalent to FIRST_VALUE; an offset of 1 from last is equivalent to LAST_VALUE.

LAG looks for a preceding row, and LEAD for a following row. LAG and LEAD get their values within a distance respective to the current row and the offset (which defaults to 1) passed.

In a case where the offset points outside the partition, the default parameter (which defaults to NULL) is returned.