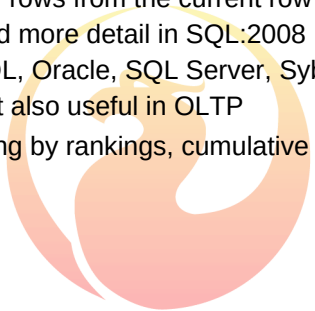
Firebird 3 Windowing Functions

Author: Philippe Makowski IBPhoenix
Email: pmakowski@ibphoenix

Date: 2011-11-22

What are Windowing Functions?

- Similar to classical aggregates but does more!
- Provides access to set of rows from the current row
- Introduced SQL:2003 and more detail in SQL:2008
- Supported by PostgreSQL, Oracle, SQL Server, Sybase and DB2
- Used in OLAP mainly but also useful in OLTP
    - Analysis and reporting by rankings, cumulative aggregates



Windowed Table Functions

- Windowed table function
  - operates on a window of a table
  - returns a value for every row in that window
  - the value is calculated by taking into consideration values from the set of rows in that window
- 8 new windowed table functions
- In addition, old aggregate functions can also be used as windowed table functions
- Allows calculation of moving and cumulative aggregate values.

A Window

- Represents set of rows that is used to compute additionnal attributes
- Based on three main concepts
  - **partition**
    - specified by PARTITION BY clause in OVER()
    - Allows to subdivide the table, much like GROUP BY clause
    - Without a PARTITION BY clause, the whole table is in a single partition
  - **order**
    - defines an order with a partition
    - may contain multiple order items
      - Each item includes a value-expression
      - NULLS FIRST/LAST defines ordering semantics for NULL
    - this clause is independant of the query's ORDER BY clause
  - **frame** (Firebird don't implement frame yet)

Window function syntax in Firebird

```
<window function> ::=
  <window function type> OVER <window specification>

<window function type> ::=
   <rank function type> <left paren> <right paren>
 | ROW_NUMBER <left paren> <right paren>
 | <aggregate function>
 | <lead or lag function>
 | <first or last value function>
 | <nth value function>

<rank function type> ::=
   RANK
 | DENSE_RANK
```

Window function syntax in Firebird

```
<lead or lag function> ::=
 <lead or lag> <left paren> <lead or lag extent>
    [ <comma> <offset> [ <comma> <default expression> ] ] <right paren>

<lead or lag> ::=
 LEAD | LAG

<lead or lag extent> ::=
 <value expression>

<offset> ::=
 <exact numeric literal>

<default expression> ::=
 <value expression>
```

Window function syntax in Firebird

```
<first or last value function> ::=
 <first or last value> <left paren> <value expression> <right paren>

<first or last value> ::=
 FIRST_VALUE | LAST_VALUE

<nth value function> ::=
 NTH_VALUE <left paren> <value expression> <comma> <nth row>
 <right paren>

<nth row> ::=
  <simple value specification>
| <dynamic parameter specification>
(must be an integer >= 1)
```

Window specification syntax in Firebird

```
<window specification> ::=
 <left paren> <window specification details> <right paren>

<window specification details> ::=
  [ <window partition clause> ]
  [ <window order clause> ]

<window partition clause> ::=
  PARTITION BY <window partition column reference or expression list>

<window partition column reference or expression list> ::=
 <window partition column reference or expression>
    [ { <comma> <window partition column reference or expression> }... ]

<window partition column reference or expression> ::=
 <column reference or expression> [ <collate clause> ]

<window order clause> ::=
  ORDER BY <sort specification list>
```

About order

The window order clause is to calculate its values, and has nothing to do with the main order.

```
select emp_no, salary,
   sum(salary) over (order by salary) cum_salary,
   sum(salary) over (order by salary desc) cum_salary_desc
 from employee order by emp_no;
EMP_NO                SALARY            CUM_SALARY       CUM_SALARY_DESC
======= ==================== ==================== ====================
      2             105900.00            1990493.02          113637875.00
      4              97500.00            1680929.02          113939039.00
     28              22935.00              22935.00          115522468.02
    121           99000000.00          115522468.02           99000000.00
    145              32000.00             113210.00          115441258.02
```

About order

If the main query doesn't have a order by but a window have, the query will be ordered accordingly to the last window order.

```
select emp_no, salary,
   sum(salary) over (order by salary) cum_salary,
   sum(salary) over (order by salary desc) cum_salary_desc
 from employee;
EMP_NO                SALARY            CUM_SALARY       CUM_SALARY_DESC
======= ==================== ==================== ====================
    121           99000000.00          115522468.02           99000000.00
    118            7480000.00           16522468.02          106480000.00
    110            6000000.00            9042468.02          112480000.00
    134             390500.00            3042468.02          112870500.00
```

About order

```
select emp_no, salary,
   sum(salary) over (order by salary desc) cum_salary_desc,
   sum(salary) over (order by salary) cum_salary
 from employee;
EMP_NO                SALARY       CUM_SALARY_DESC            CUM_SALARY
======= ==================== ==================== ====================
     28             22935.00          115522468.02             22935.00
    109             27000.00          115499533.02             49935.00
     65             31275.00          115472533.02             81210.00
    145             32000.00          115441258.02            113210.00
```

Set Functions as Window Functions

The OVER clause turns a set function into a window function

- Aggregated value is computed per current row window

```
select emp_no, dept_no, salary,
 avg(salary) over (partition by dept_no) as dept_avg
from employee;
EMP_NO DEPT_NO                 SALARY              DEPT_AVG
======= ======= ==================== ====================
     12 000                  53793.00             133321.50
    105 000                 212850.00             133321.50
    127 100                  44000.00              77631.25
     85 100                 111262.50              77631.25
     34 110                  61637.81              65221.40
     61 110                  68805.00              65221.40
    110 115                6000000.00            6740000.00
    118 115                7480000.00            6740000.00
```

Set Functions as Window Functions

Who are the highest paid relatively compared with the department average?

```
select emp_no, dept_no, salary,
  avg(salary) over (partition by dept_no) as dept_avg,
  salary - avg(salary) over (partition by dept_no) as diff
from employee
order by diff desc;
```

EMP_NO	DEPT_NO	SALARY	DEPT_AVG	DIFF
118	115	7480000.00	6740000.00	740000.00
105	000	212850.00	133321.50	79528.50
107	670	111262.50	71268.75	39993.75
2	600	105900.00	66450.00	39450.00
85	100	111262.50	77631.25	33631.25
4	621	97500.00	69184.87	28315.13
46	900	116100.00	92791.31	23308.69
9	622	75060.00	53409.16	21650.84

Built-in Windowing Functions

- RANK () OVER ...
- DENSE_RANK () OVER ...
- LAG () OVER ...
- LEAD () OVER ...
- ROW_NUMBER () OVER ...
- FIRST_VALUE () OVER ...
- LAST_VALUES () OVER ...
- NTH_VALUE () OVER ...

Built-in Windowing Functions : row_number

Returns number of the current row

```
select emp_no, dept_no, salary,
row_number() over (order by salary desc nulls last) as row_num
from employee;
 EMP_NO DEPT_NO                SALARY               ROW_NUM
======= ======= ==================== ====================
    121 125               99000000.00                    1
    118 115                7480000.00                    2
    110 115                6000000.00                    3
    134 123                 390500.00                    4
    105 000                 212850.00                    5
     46 900                 116100.00                    6
     85 100                 111262.50                    7
    107 670                 111262.50                    8
    141 121                 110000.00                    9
```

row_number() always incremented values independent of frame

Built-in Windowing Functions : rank

Returns rank of the current row with gap

```
select emp_no, dept_no, salary,
rank() over (order by salary desc nulls last) as rank
from employee;
 EMP_NO DEPT_NO                SALARY                    RANK
======= ======= ===================== =====================
    121 125               99000000.00                     1
    118 115                7480000.00                     2
    110 115                6000000.00                     3
    134 123                 390500.00                     4
    105 000                 212850.00                     5
     46 900                 116100.00                     6
     85 100                 111262.50                     7
    107 670                 111262.50                     7
    141 121                 110000.00                     9
```

rank() OVER(*empty*) returns 1 for all rows, since all rows are peers to each other

Built-in Windowing Functions : dense_rank

Returns rank of the current row without gap

```
select emp_no, dept_no, salary,
dense_rank() over (order by salary desc nulls last)
from employee;
 EMP_NO DEPT_NO                SALARY            DENSE_RANK
======= ======= ==================== ====================
    121 125               99000000.00                     1
    118 115                7480000.00                     2
    110 115                6000000.00                     3
    134 123                 390500.00                     4
    105 000                 212850.00                     5
     46 900                 116100.00                     6
     85 100                 111262.50                     7
    107 670                 111262.50                     7
    141 121                 110000.00                     8
```

dense_rank() OVER(*empty*) returns 1 for all rows, since all rows are peers to each other

Built-in Windowing Functions : rank, dense_rank, row_number

```
select emp_no, dept_no, salary,
 rank() over (order by salary desc nulls last),
 dense_rank() over (order by salary desc nulls last),
 row_number() over (order by salary desc nulls last)
from employee;
 EMP_NO DEPT_NO        SALARY    RANK    DENSE_RANK    ROW_NUMBER
======= ======= ============= ======= ============= =============
    121 125       99000000.00       1             1             1
    118 115        7480000.00       2             2             2
    110 115        6000000.00       3             3             3
    134 123         390500.00       4             4             4
    105 000         212850.00       5             5             5
     46 900         116100.00       6             6             6
     85 100         111262.50       7             7             7
    107 670         111262.50       7             7             8
    141 121         110000.00       9             8             9
```

Missing Built-in Windowing Functions : percent_rank

Returns relative rank of the current row: (rank - 1) / (total rows - 1)
can be emulated with rank() and count()

```
select emp_no, dept_no, salary,
 cast((rank() over (order by salary asc))-1 as double precision)
      / ( count(*) over () -1) as percent_rank
from employee
 EMP_NO DEPT_NO                 SALARY             PERCENT_RANK
======= ======= ==================== ======================
     28 120                   22935.00       0.000000000000000
    109 600                   27000.00     0.02439024390243903
...
    144 672                   35000.00      0.1219512195121951
    114 623                   35000.00      0.1219512195121951
    138 621                   36000.00      0.1707317073170732
...
    118 115                 7480000.00      0.9756097560975610
    121 125                99000000.00       1.000000000000000
```

Missing Built-in Windowing Functions : cum_dist

Returns relative rank; (# of preced. or peers) / (total row)
can be emulated with count()

```
select emp_no, dept_no, salary,
  cast((count(*) over (order by salary asc)) as double precision)
       / ( COUNT(*) over () ) as cum_dist
from employee
 EMP_NO DEPT_NO                 SALARY                 CUM_DIST
======= ======= ==================== ======================
     28 120                  22935.00     0.02380952380952381
    109 600                  27000.00     0.04761904761904762
...
    144 672                  35000.00      0.1666666666666667
    114 623                  35000.00      0.1666666666666667
    138 621                  36000.00      0.1904761904761905
...
    118 115                7480000.00      0.9761904761904762
    121 125               99000000.00       1.000000000000000
```

Built-in Windowing Functions : first_value,nth_value,last_value

Warning

Note that first_value, last_value, and nth_value consider only the rows within the "window frame", which by default contains the rows from the start of the partition through the last peer of the current row.

And Firebird today don't allow you to change the frame definition.

This is likely to give unhelpful results for nth_value and particularly last_value.

Built-in Windowing Functions : first_value

Returns value of the first row in a window

```
select emp_no, dept_no, salary,
  first_value(salary) over (order by salary desc nulls last) overall_max,
  first_value(salary) over (partition by dept_no order by salary desc nulls last) dept_max,
  first_value(salary) over (order by salary asc nulls last) overall_min,
  first_value(salary) over (partition by dept_no order by salary asc nulls last) dept_min
from employee;
```

EMP_NO	DEPT_NO	SALARY	OVERALL_MAX	DEPT_MAX	OVERALL_MIN	DEPT_MIN
12	000	53793.00	99000000.00	212850.00	22935.00	53793.00
105	000	212850.00	99000000.00	212850.00	22935.00	53793.00
127	100	44000.00	99000000.00	111262.50	22935.00	44000.00
85	100	111262.50	99000000.00	111262.50	22935.00	44000.00
34	110	61637.81	99000000.00	68805.00	22935.00	61637.81
61	110	68805.00	99000000.00	68805.00	22935.00	61637.81
110	115	6000000.00	99000000.00	7480000.00	22935.00	6000000.00
118	115	7480000.00	99000000.00	7480000.00	22935.00	6000000.00
28	120	22935.00	99000000.00	39224.06	22935.00	22935.00
37	120	39224.06	99000000.00	39224.06	22935.00	22935.00
121	125	99000000.00	99000000.00	99000000.00	22935.00	99000000.00

Built-in Windowing Functions : nth_value

Returns value of the *nth* row in a window

```
select emp_no, dept_no, salary,
  nth_value(salary,2) over (partition by dept_no order by salary desc)
from employee;
 EMP_NO DEPT_NO                SALARY             NTH_VALUE
======= ======= ==================== ====================
    105 000                212850.00                <null>
     12 000                 53793.00              53793.00
     85 100                111262.50                <null>
    127 100                 44000.00              44000.00
     61 110                 68805.00                <null>
     34 110                 61637.81              61637.81
    118 115               7480000.00                <null>
    110 115               6000000.00             6000000.00
     37 120                 39224.06                <null>
     36 120                 33620.63              33620.63
     28 120                 22935.00              33620.63
```

Built-in Windowing Functions : last_value

Returns value of the last row in a window

```
select emp_no, dept_no, salary,
   last_value(salary) over (partition by dept_no order by salary asc)
    from employee;
 EMP_NO DEPT_NO                SALARY            LAST_VALUE
======= ======= ==================== ====================
     12 000                 53793.00              53793.00
    105 000                212850.00             212850.00
    127 100                 44000.00              44000.00
     85 100                111262.50             111262.50
     34 110                 61637.81              61637.81
     61 110                 68805.00              68805.00
```

Built-in Windowing Functions : lag and lead

Both LAG and LEAD functions have the same usage.

```
LAG  (value [,offset[, default]]) OVER ([query_partition_clause] order_by_clause)
LEAD (value [,offset[, default]]) OVER ([query_partition_clause] order_by_clause)
```

- value - Can be a column or a built-in function, except for other analytic functions.
- offset - The number of rows preceeding/following the current row, from which the data is to be retrieved. The default value is 1.
- default - The value returned if the offset is outside the scope of the window. The default value is NULL.

Built-in Windowing Functions : lag

Returns value of row above

```
select emp_no, dept_no, salary,
 lag(salary) over (order by salary desc nulls last)
from employee;
 EMP_NO DEPT_NO                SALARY                    LAG
======= ======= ==================== ====================
    121 125               99000000.00                <null>
    118 115                7480000.00           99000000.00
    110 115                6000000.00            7480000.00
    134 123                 390500.00            6000000.00
    105 000                 212850.00             390500.00
     46 900                 116100.00             212850.00
     85 100                 111262.50             116100.00
    107 670                 111262.50             111262.50
    141 121                 110000.00             111262.50
```

lag() only acts on a partition.

Built-in Windowing Functions : lead

Returns value of row below

```
select emp_no, dept_no, salary,
 lead(salary) over (order by salary desc nulls last)
from employee;
 EMP_NO DEPT_NO                SALARY                   LEAD
======= ======= ==================== ====================
    121 125               99000000.00            7480000.00
    118 115                7480000.00            6000000.00
    110 115                6000000.00             390500.00
    134 123                 390500.00             212850.00
    105 000                 212850.00             116100.00
     46 900                 116100.00             111262.50
     85 100                 111262.50             111262.50
    107 670                 111262.50             110000.00
    141 121                 110000.00             105900.00
     28 120                  22935.00                <null>
```

lead() only acts on a partition.

Cumulative aggregates

The current row window is now restricted to all rows equal to or preceding the current row within the current partition

- *Find the total sales per quarter, and cumulative sales in quarter order*

```
select order_year, order_quarter, sum(total_value) as q_sales,
 sum(sum(total_value)) over (partition by order_year order by order_quarter)
     as cum_sales_year
from  v_sales_q
group by order_year, order_quarter;
ORDER_YEAR ORDER_QUARTER               Q_SALES        CUM_SALES_YEAR
========== ============= ==================== ====================
     1992             3              2985.00               2985.00
     1992             4             72000.00              74985.00
     1993             1               648.00                648.00
     1993             2             20000.00              20648.00
     1993             3           1009643.04            1030291.04
     1993             4            699724.16            1730015.20
     1994             1            440590.83             440590.83
```

Performance

List orders, quantity ordered and cumulative quantity ordered by day

```
 ORDER_DATE PO_NUMBER  QTY_ORDERED         QTY_CUMUL_DAY
========== ========= ============ =====================
1991-03-04  V91E0210            10                    10
1992-07-26  V92J1003            15                    15
1992-10-15  V92E0340             7                     7
1992-10-15  V92F3004             3                    10
1993-02-03  V9333005             2                     2
1993-03-22  V93C0120             1                     1
1993-04-27  V9333006             5                     5
1993-08-01  V93H3009             3                     3
1993-08-09  V9324200          1000                  1000
1993-08-09  V93C0990            40                  1040
```

Performance

Without window function

```
SELECT ORDER_DATE, CUST_NO,QTY_ORDERED,
      (SELECT SUM(QTY_ORDERED)
       FROM   SALES AS Si
       WHERE  Si.ORDER_DATE = S.ORDER_DATE
       AND Si.CUST_NO <= S.CUST_NO)
      AS QTY_CUMUL_DAY
FROM   SALES AS S
ORDER BY S.ORDER_DATE, S.CUST_NO
```

```
PLAN (SI INDEX (RDB$FOREIGN25))
PLAN SORT (S NATURAL)
SALES 591 indexed reads
SALES  33 non indexed reads
```

Performance

With window function

```
SELECT ORDER_DATE, PO_NUMBER,QTY_ORDERED,
      SUM(QTY_ORDERED)
      OVER (PARTITION BY ORDER_DATE
       ORDER BY PO_NUMBER)
      AS QTY_CUMUL_DAY
FROM   SALES
ORDER BY ORDER_DATE, PO_NUMBER
```

```
PLAN SORT (SALES NATURAL)
SALES 33 non indexed reads
```

Performance

And you can extend it nearly without cost

```
SELECT ORDER_DATE, PO_NUMBER,QTY_ORDERED,
      SUM(QTY_ORDERED)
      OVER (PARTITION BY ORDER_DATE
       ORDER BY PO_NUMBER)
      AS QTY_CUMUL_DAY,
      SUM(QTY_ORDERED)
      OVER (PARTITION BY  EXTRACT(YEAR FROM ORDER_DATE),EXTRACT(MONTH FROM ORDER_DATE)
       ORDER BY  ORDER_DATE, PO_NUMBER)
      AS QTY_CUMUL_MONTH,
      SUM(QTY_ORDERED)
      OVER (PARTITION BY  EXTRACT(YEAR FROM ORDER_DATE)
       ORDER BY  ORDER_DATE, PO_NUMBER)
      AS QTY_CUMUL_YEAR
FROM   SALES
ORDER BY ORDER_DATE, PO_NUMBER
```

```
PLAN SORT (SALES NATURAL)
SALES 33 non indexed reads
```

ORDER_DATE	PO_NUMBER	QTY_ORDERED	QTY_CUMUL_DAY	QTY_CUMUL_MONTH	QTY_CUMUL_YEAR
1991-03-04	V91E0210	10	10	10	10
1992-07-26	V92J1003	15	15	15	15
1992-10-15	V92E0340	7	7	7	22
1992-10-15	V92F3004	3	10	10	25
1993-02-03	V9333005	2	2	2	2
1993-03-22	V93C0120	1	1	1	3
1993-04-27	V9333006	5	5	5	8
1993-08-01	V93H3009	3	3	3	11
1993-08-09	V9324200	1000	1000	1003	1011
1993-08-09	V93C0990	40	1040	1043	1051
1993-08-16	V9324320	1	1	1044	1052
1993-08-20	V93J3100	16	16	1060	1068
1993-08-27	V93F3088	10	10	1070	1078

Thank you !