

Comparative Window Functions

Authors: Srikanth Bellamkonda (Oracle), Fred Zemke (Oracle)

Date: 10/7/2009

ABSTRACT

This paper describes several new extensions to SQL window functions, making them more powerful and suitable for analytic calculations.

1. INTRODUCTION

Analytic window functions provide a powerful mechanism whereby ranking, cumulative, moving, and reporting aggregate calculations can be specified succinctly in SQL and evaluated very efficiently by the RDBMS. Using this specification, the user or an analytic application can define a window (fixed or variable in size) for each row and have an aggregate (native or user-defined aggregate) applied on all rows in the window. Window specification looks like this:

```
AGGREGATE(<arguments>) OVER (  
  [<PARTITION BY clause>]  
  [<ORDER BY clause>  
  [<window specification>]])
```

Window specification's partition-by (PBY) clause logically divides the dataset into disjoint sets (or partitions) on which window aggregate computation is performed. The order-by (OBY) clause specifies the order of rows within each partition. The window specification defines the window for each row. The result of applying the aggregate function on the rows in the window becomes the window function's value for that row.

For example, a query to find for each stock ticker and each day, the number of times a stock price has exceeded \$30 in the past 30 days can be specified with the current window specification. Assuming the table stocks (ticker, day¹, value), SQL for this query would be:

```
Q1  
SELECT ticker, day, price,  
       SUM(CASE WHEN price > 30 THEN 1  
           ELSE 0 END)  
       OVER (  
         PARTITION BY ticker  
         ORDER BY day  
         ROWS BETWEEN 30 PRECEDING  
           AND 1 PRECEDING) as freq  
FROM stocks;
```

2. MOTIVATION

Though the window specification is powerful and allows efficient evaluation of ranking, moving, cumulative, and reporting aggregates by the RDBMS, it lacks the ability to solve what I call "comparative window function" calculations. These are calculations wherein the window function value for the current

row² is the result of a comparative analysis involving one or more rows in the window with the current row.

For example, window functions as they exist today will not be useful in answering queries like "find me the number of times the stock price has exceeded a day's price", instead of a fixed \$30 value as in the above example. SQL to answer this query will have to resort to self-joins and becomes inefficient:

```
Q2  
SELECT s1.ticker, s1.day, s1.price,  
       SUM(CASE WHEN s2.price > s1.price  
           THEN 1  
           ELSE 0 END) freq  
FROM stocks s1, stocks s2  
WHERE s1.ticker = s2.ticker  
      AND s2.day <= s1.day-1  
      AND s2.day >= s1.day-30  
GROUP BY s1.ticker, s1.day, s1.price;
```

3. OUR PROPOSAL

Our proposal extends window functions making them more powerful and suitable for "comparative analysis". The extensions include:

- **Markers** that give access to certain interesting rows.
- **Offsets** that provide ability to access a row at any offset from the marker rows.
- **Cursor** that moves across rows in the window.

We explain each of these now.

3.1 Markers

Markers give access to several interesting rows for *comparative analysis*. Markers we intend to introduce are:

- **ANCHOR_ROW** – this is the row for which we form the window and apply the window aggregate on the rows in the window.
- **FIRST_ROW** – this is the first row of the window for ANCHOR_ROW
- **LAST_ROW** – this is the last row of the window for ANCHOR_ROW

We call the row a marker points to as "marker row". Markers can only be used inside the window aggregate function. For example consider the following data with schema (ticker, day, price) and the window specification "rows between 4 preceding and 1

¹ Primary key

² For any given row in the input, a window gets defined as per the SQL window specification and the window aggregate is applied on the rows in the window.

preceding". Assume that "(ORCL, 6, 11)" is the row for which we are currently computing the window function.

```

orcl 1 10
orcl 2 11 ← FIRST_ROW
orcl 3 12
orcl 4 12
orcl 5 12 ← LAST_ROW
orcl 6 11 ← ANCHOR_ROW
orcl 7 12
orcl 8 12

```

Accessing individual columns from the marker rows can be done with a functional specification like

```
INDEX(<column name>, <marker>)
```

For example "*INDEX(price, ANCHOR_ROW)*" gives the value of column "*price*" from the ANCHOR_ROW. Other keyword choices in lieu of INDEX are SUBSCRIPT or VALUE.

Using the ANCHOR_ROW marker, the self-join query Q2 can be written elegantly as:

Q3

```

SELECT ticker, day, price,
       SUM(CASE WHEN
            price > INDEX(price, ANCHOR_ROW)
            THEN 1
            ELSE 0 END)
OVER (PARTITION BY ticker
      ORDER BY day
      ROWS BETWEEN 30 PRECEDING
      AND 1 PRECEDING) as freq
FROM stocks s1;

```

The query Q3 can be efficiently evaluated by the RDBMS as it fits well in the window function framework. Markers FIRST_ROW and LAST_ROW can be useful as well. As mentioned earlier, the markers or INDEX function can only be used inside the aggregate function.

Though it may be a pure syntactic sugar, we can allow an expression to be the first argument of INDEX function. For example, instead of having to write multiple index functions as in:

```

INDEX(sal, ANCHOR_ROW) +
INDEX(commission, ANCHOR_ROW) -
INDEX(revenue, ANCHOR_ROW)

```

SQL can be written as:

```
INDEX(sal+commission-revenue, ANCHOR_ROW) .
```

This brings up the concept of nesting where in one can nest INDEX functions on different markers. For example, one can write:

```

INDEX(sal-0.2*
      INDEX(commission, FIRST_ROW),
      ANCHOR_ROW) .

```

3.2 Offsets

We propose positive and negative integer "*offsets*" to access rows at a specified offset from the *marker* rows. For example one can say "*INDEX(price, FIRST_ROW + 1)*" to get the value of column "*price*" from the second row in the window. Similarly, "*INDEX(price, LAST_ROW - 1)*" gives access to *price* value from row before the last row in the window.

When an offset takes to a row outside of the window i.e., before FIRST_ROW and after LAST_ROW, then INDEX would return NULL. *Offsets* can be specified from the ANCHOR_ROW as well. We define the access semantic rules as:

1. From any *marker row*, only rows within the window [FIRST_ROW, LAST_ROW] are visible/accessible.
2. When an offset goes beyond the window, the value returned would be NULL.
3. ANCHOR_ROW is special in that it's always accessible/available.

To illustrate the semantics, consider a window that is "*between 5 preceding and 2 preceding*" operating on the following data (ticker, day, price). Assume that the ANCHOR_ROW is at (orcl, 7, 12) and we are accessing *price* column using INDEX function.

```

orcl 1 10
orcl 2 11 ← FIRST_ROW
orcl 3 12
orcl 4 12
orcl 5 12 ← LAST_ROW
orcl 6 11
orcl 7 12 ← ANCHOR_ROW
orcl 8 12

```

- As per rule #1, offsets FIRST_ROW + 1, LAST_ROW - 3 would provide values from rows (orcl,3,12) and (orcl,2,11) respectively.
- As per rule #1, the row at offset ANCHOR_ROW - 2 is accessible as it is the LAST_ROW.
- As per rule #2, offset specifications FIRST_ROW - 1, FIRST_ROW + 4, LAST_ROW + 1, LAST_ROW - 4 would return NULL values as they go beyond the window [FIRST_ROW, LAST_ROW].
- Similarly, as per rule #2, offsets ANCHOR_ROW + 1 and ANCHOR_ROW - 1 will be NULL as they are outside of the window.
- As per rule #3, values from ANCHOR_ROW are available even though it's outside the window. That is, INDEX(price, ANCHOR_ROW) would return 12.
- As per rule #2, the offset specifications like LAST_ROW + 2 and FIRST_ROW + 5 would return NULL even though they point us to the ANCHOR_ROW. That is, INDEX(price, LAST_ROW + 2) would be NULL while INDEX(price, ANCHOR_ROW) would return 12.

To enable users to differentiate between NULLs from accessing non-visible rows and the actual NULLs in the data, we provide a function *ISPRESNT (<marker>)*. This function would return TRUE if the marker row is in the window, and FALSE otherwise. In addition, we can extend INDEX to take an optional default value argument. The default value would be returned when the offset takes to a row outside of the window [FIRST_ROW, LAST_ROW]. INDEX syntax would then be like:

```

INDEX(<expression>, <marker row>,
      <default>)

```

EXCLUSION Clause

ANSI window function syntax has an “*exclusion*” clause to exclude rows from being aggregated. This affects our ANCHOR_ROW semantic in a strange way.

```
EXCLUDE [CURRENT ROW | GROUP |
        TIES | NO OTHERS]
```

For example, EXCLUDE CURRENT ROW excludes the current row from being aggregated by the window aggregate. Note that CURRENT ROW is same as the ANCHOR_ROW³.

We defined the semantic of ANCHOR_ROW such that it is always available (rule #3). So even with *exclusion* clause, ANCHOR_ROW would be available for comparative analysis. Consider the following query for example.

Q4

```
SELECT ticker, day, price,
       MIN(CASE WHEN
           price >= INDEX(price, ANCHOR_ROW)
           THEN price ELSE NULL END)
OVER (PARTITION BY ticker
     ORDER BY day
     ROWS BETWEEN 10 PRECEDING
           AND 10 FOLLOWING
     EXCLUDE CURRENT ROW) as freq
FROM stocks s1;
```

The window “*10 preceding and 10 following*” used in this query spans before and after the CURRENT ROW (i.e., the ANCHOR_ROW) and includes the current row. With the exclusion clause “*exclude current row*”, the CURRENT ROW will be ignored while calculating the window aggregate MIN. However, access to the ANCHOR_ROW as in INDEX (price, ANCHOR_ROW) would be legal and retrieve the column “*price*” value from the ANCHOR_ROW. It’s just that we don’t process the window aggregate for the current row.

3.3 Cursor

Another extension is the notion of “*cursor*” that moves from the FIRST_ROW to the LAST_ROW in the window as we are computing the window aggregate. The marker “*CURSOR_ROW*” gives access to the row the cursor is at. For example CURSOR_ROW moves from row (ORCL, 2, 11) to (ORCL, 5, 12) for a window “*between 4 preceding and 1 preceding*” when the ANCHOR_ROW is at (ORCL, 6, 11).

```
orcl 1 10
orcl 2 11 ←FIRST_ROW ←CURSOR_ROW
orcl 3 12 ←CURSOR_ROW
orcl 4 12 ←CURSOR_ROW
orcl 5 12 ←LAST_ROW ←CURSOR_ROW
orcl 6 11 ←ANCHOR_ROW
orcl 7 12
orcl 8 12
```

As with other markers, INDEX notation can be used to get column values. Note that INDEX (<column>, CURSOR_ROW+1) gives access to the row next to the cursor and

INDEX (<column>, CURSOR_ROW-1) gives the row previous to the cursor row.

Using this “*cursor*” mechanism, queries like “**how many times in a 30day moving period, a stock price has exceeded a day’s value and maintained (either stayed the same or has increased) it for 2 days**” can be answered efficiently using window function. SQL would be something like:

Q5

```
SELECT ticker, day, price,
       SUM(CASE WHEN
           INDEX(price, CURSOR_ROW) >
           INDEX(price, ANCHOR_ROW)
           AND INDEX(price, CURSOR_ROW+1) >=
           INDEX(price, CURSOR_ROW)
           THEN 1 ELSE 0 END)
OVER (PARTITION BY ticker
     ORDER BY day
     ROWS BETWEEN 30 PRECEDING
           AND 1 PRECEDING) as freq
FROM stocks s1;
```

Whenever an offset goes beyond the window, the value returned by INDEX function would be NULL.

Another useful function can be to find the distance (in number or rows) between two marker rows. For example to find the distance between CURSOR_ROW and LAST_ROW, one can say “*DISTANCE (LAST_ROW, CURSOR_ROW)*”.

3.4 LOGICAL windows

The semantics of *makers* and *offsets* as described in the previous section for physical windows (ROWS specification) would work for logical windows (RANGE specification) when there are no duplicates. With duplicates (in Order-By expression value), the markers FIRST_ROW, LAST_ROW and ANCHOR_ROW will not uniquely identify a single row and non-determinism will creep in. This is in contrast with the default logical window behavior where in all duplicate rows will have the same window function result and there is no non-determinism. However, it needs to be noted that logical windows will have non-determinism with EXCLUDE TIES | EXCLUDE CURRENT ROW options.

To address this issue, we can have new markers FIRST_GROUP, LAST_GROUP and ANCHOR_GROUP that give access to all duplicates and then have aggregates as arguments to INDEX. For example, one can write:

```
INDEX(max(sal), FIRST_GROUP)
```

ROW markers would give runtime error when they do not qualify a row uniquely.

³ May be we should use the keyword CURRENT ROW instead of ANCHOR_ROW.