Analytic Functions in Oracle 8*i*

Srikanth Bellamkonda, Tolga Bozkaya, Bhaskar Ghosh
Abhinav Gupta, John Haydu, Sankar Subramanian, Andrew Witkowski

Oracle Corporation, 500 Oracle Parkway 4op7,Redwood Shores, CA 94065

{sbellamk, tbozkaya, abgupta, bghosh, jhaydu, srsubram, awitkows}@us.oracle.com

Abstract

One of the deficiencies of SQL is the lack of support for analytic calculations like moving averages, cumulative sums, ranking, percentile and lead and lag which are critical for OLAP applications. These functions work on ordered sets of data. Expressing these functions in current SQL is not elegant, requires self-joins and is difficult to optimize.

To address this issue, Oracle has introduced SQL extensions called Analytic functions in Oracle8*i* Release 2. In this paper we discuss the motivation behind these functions, their structure and semantics, their usage for answering reasonably complex business questions and various algorithms for their optimization and execution in the RDBMS server.

Several implementation techniques are discussed in the paper. It presents heuristic algorithms used in the optimizer for reducing the number of sorts required by multiple Analytic functions. It also covers specialized processing of Top-N Rank queries to reduce buffer usage in the iterator model and to improve communication costs for parallel execution.

The scope of OLAP support afforded by these new functions far exceeds existing SQL functionality. Oracle and IBM have proposed these SQL extensions to ANSI to be included in the SQL-99 standard.

1 Introduction

Analytic calculations like moving averages, cumulative sums, ranking, percentile and lag/lead are critical for decision support applications. In order for ROLAP tools and applications to have access to such calculations in an efficient and scalable manner, it is important that relational databases provide a succinct representation in SQL which can be easily optimized. Analytic functions have been introduced in Oracle 8*i* Release 2 to meet these requirements.

These functions typically operate on a window defined on an ordered set of rows. That is, for each row in the input, a window is defined to encompass rows satisfying certain condition relative to the current row and the function is applied over the data values in the window. The function can be an aggregate function or a function like rank, ntile etc.

Expressing these functions in the existing SQL is quite complicated, may require definition of views and joins and is difficult to optimize. In commercial databases, this functionality can be simulated through the use of subqueries, rownum[1] and procedural packages[2]. These simulations often result in inefficient and non-scalable execution plans as the query's semantics are dispersed across multiple query blocks or procedural extensions which are non-parallelizable.

Oracle has proposed extensions to SQL that express analytic functions succinctly in a single query block, thus paving the way for efficient and

(1) An Oracle specific function which generates a monotonically increasing sequence number for the result rows of a query
(2) PL/SQL in Oracle

scalable execution and better support for OLAP tools and analytic applications. A syntax proposal has been made to ANSI jointly with IBM[OraIBM99].

1.1 Business Needs

The business needs for analytic calculations fall into the following categories.

1. **Ranking Functions:** These functions rank data based on some criteria and address business questions like "rank products based on their annual sales", "find top 10 and bottom 5 salespersons in each region".
2. **Window Functions:** These functions operate on a sliding window of rows defined on an ordered data set and for each row *r* return some aggregate over the window rooted at *r*. These functions can either compute moving aggregates or cumulative aggregates. Common business queries include "find the 13-week moving average of a stock price", or "find the cumulative monthly sales of a product in each region".
3. **Reporting Functions:** These functions address business questions requiring comparisons of aggregates at different levels. A typical query would be "for each region, find the sales for cities which contribute at least 10% of regional sales". This query needs access to aggregated sales per (region) and (region, city) levels.
4. **Lag/Lead Functions:** These functions provide access to other rows from any given row without the need to perform a self-join. Queries addressed by these functions include "find monthly change in the account balance of a customer" where we can access a row one month before the current row.
5. **Inverse Distribution and First/Last Functions:** Inverse Distribution functions allow computation of median and similar functions on an ordered data set. First/Last functions allow us to compute aggregates on the first or last value of an ordered set. For example, find the average balance for the first month of each year. Details of these functions are given in the Appendix.

2 Outline of the Paper

The rest of the paper is organized as follows. In Section 3 we present queries for some interesting business questions using SQL with and without our extensions. We also present the syntax and semantics of Analytic functions there. The computational model is presented in Section 4. In Section 5, we present several families of Analytic functions.

Section 6 presents some optimization and execution techniques, and Section 7 presents results of experiments comparing our approach with using existing SQL for computing the queries given in Section 3. We present related work in Section 8. In the Appendix we present some more examples and a few more Analytical functions.

3 Motivating Examples

In the rest of the paper, we will use the following SALESTABLE schema.

```
salesTable(region, state, product,
  salesperson, date, sales)
```

The key of the table is (region, state, product, salesperson, date). The table records each transaction made by any salesperson. We show here some queries which are difficult to express in SQL92 but are elegant and intuitive with our extensions.

Q1 Find Top-3 salespersons within each region based on sales

This query can either be expressed using procedural extensions to SQL (e.g. PL/SQL in Oracle) or using complicated non-equi joins in SQL92. We give the PL/SQL representation here because this is the better approach.

```
SELECT region,salesperson, s_sales,
    RANK(region,s_sales) rnk
FROM (
    SELECT region, salesperson,
            SUM(sales) s_sales
    FROM salesTable
    GROUP region, salesperson
     ORDER BY region, SUM(sales) DESC
 WHERE RANK(region, s_sales) <= 3;
```

The data from the in-line view is passed "in order" to the user defined PL/SQL function (RANK). The operation of assigning ranks inherently becomes sequential, whereas we could have easily parallelized by partitioning by regional level and keeping only the Top-3 values within each region.

Q2 Compute the moving sum of sales of each product within the last month

This query would require an expensive non-equi self-join where each row is joined with rows 1 month before it:

```
 SELECT s1.product, SUM(s2.sales) as m_avg
 FROM salesTable s1, salesTable s2
 WHERE s1.product = s2.product AND
       s2.date <= s1.date AND
       s2.date >= ADD_MONTHS(s1.date, -1)
 GROUP BY s1.product, s1.date;
```

Clearly, the query execution time would grow linearly with the size of window.

Q3 For each year, find products with maximum annual sales

This query requires defining the view V1 for computing yearly sales for each product and another view V2 (over V1) to compute maximum yearly sales.

```
 CREATE VIEW V1
 AS SELECT product, year(date) yr,
       SUM(sales) s_sales
  FROM salesTable
  GROUP BY product, year(date);

 CREATE VIEW V2
 AS SELECT yr, MAX(s_sales) m_sales
  FROM V1
  GROUP BY yr;

 SELECT product, yr, s_sales
 FROM V1, V2
 WHERE V1.yr = V2.yr
    AND V1.s_sales = V2.m_sales
```

Q4 Give the Top-10 products for states which contribute more than 25% of regional sales.

This query requires not only access to 2 different levels of aggregation, but also filtering based on rank values.

```
 CREATE VIEW V4
 AS SELECT region, state, SUM(sales)
    s_sales
 FROM salesTable
 GROUP BY region, state

 CREATE VIEW V5
 AS SELECT region, SUM(sales) s_sales
 FROM salesTable
 GROUP BY region;

 SELECT region, state, product,
    RANK(region, state, X.s_sales) rank
 FROM (
    SELECT region, state, product,
           SUM(sales) s_sales
    FROM salesTable, V4, V5
    WHERE salesTable.region = V4.region
       AND salesTable.state = V4.state
       AND V4.region = V5.region
       AND V4.s_sales > 0.25 * V5.s_sales
    GROUP BY region,state, product
    ORDER BY region,state,SUM(sales) DESC)
    X
 WHERE
    RANK(region, state, X.s_sales) <= 10;
```

Clearly, it is not very intuitive to define views for a particular instance of a query. Also, in a typed language with positional arguments, a different RANK function would have to be defined for each new instance. Not only is the representation complex, but multiple aggregations and joins would make it quite inefficient as well.

Q5 For each product, compare sales of a month to its previous month

This query requires a self-join of monthly aggregated salesTable.

```
CREATE VIEW V3
AS SELECT product, month(date) mn,
   SUM(sales) s_sales
FROM salesTable
GROUP BY product, month(date);

SELECT product, X.mn, X.s_sales -
   Y.s_sales dif_sales
FROM V3 X LEFT OUTER JOIN V3 Y
     ON (X.product = Y.product
         AND X.mn = Y.mn + 1);
```

The query assumes dense data. The self-join would be unnecessary if there was a function to look at some other row in the data set.

The queries in the above examples require partitioning the data into some groups, ordering data within each group, defining a window for each row (applicable to some functions), and specifying an aggregate/function that is applied on rows in the window. Our challenge in the language design is to have a unified framework which encompasses all these classes of functions. The resulting syntax for these analytic functions used in queries 1-5 is:

```
function(<arguments>) OVER
   ([<partition by clause>]
    [<order by clause>
    [aggregate group clause]])
```

<partition by > and <order by> clauses in the function definition determine the partitioning and ordering columns of the query. The <partition by> clause divides the dataset into groups which are then ordered internally based on the <order by> clause. Note that the <order by> clause of the function does not necessarily guarantee the final ordering of the result set. The final ordering of the result set is dictated by the <order by> clause of the query block.

The <aggregate group clause> selects a subset of rows from this ordered set by defining the end points for each row. The row for which we are defining the end points is called as the current row. For example, the aggregate group clause ROWS 1 PRECEDING defines a "*physical*" window of size two rows i.e. the previous row and the current row.[3] Similarly, the clause ROWS BETWEEN 1 PRECEDING and 1 FOLLOWING defines a "*physical*" window of size three rows i.e. a window centered at the current row and includes one previous and one following row. Physical windows, however, are not suitable for sparse data as not every value may be present in the result set. We therefore allow the definition of *logical* windows. Consider the <aggregate group clause> RANGE INTERVAL '1' MONTH PRECEDING. It defines a *"logical"* window including all rows within the last month and the current month. Similarly, the clause RANGE INTERVAL '1' MONTH PRECEDING AND '1' MONTH FOLLOWING defines a *"logical"* window including all rows with the last month, current month and next month. A *"physical"* window is determined strictly based on the number of rows. However, a *"logical"* window is obtained by applying the condition in the <aggregate group clause> to the <order by> expression. A logical window can have only one expression in the ORDER BY clause.

For example, to determine the end points of a window defined by the aggregate group clause (ORDER BY date DESC RANGE INTERVAL '1' MONTH PRECEDING), we take the value of date in the current row and subtract '1' MONTH from it. We include all rows whose value falls with the last month and current month.

After a window has been defined for every row of the input, the function is applied to that set.

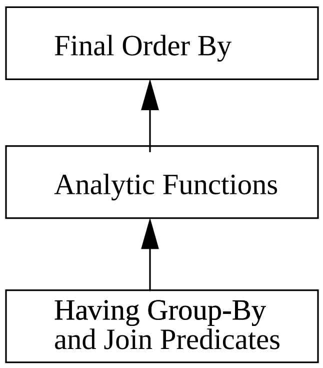
For more details on the syntax, please refer to ANSI SQL submission [OraIBM99]

4 Computational Model

Analytic functions are evaluated after all other query clauses, except distinct and order-by, have been evaluated. In the following query,

(3) A physical window is tagged with the keyword ROWS.

```
SELECT region, product, SUM(sales) s1,
   RANK() OVER (PARTITION BY region
      ORDER BY SUM(sales) DESC) s2
FROM salesTable
GROUP BY region, product
ORDER BY region, s2;
```



Analytic (rank) function s2 is evaluated after computing the aggregate function s1 over the group (region, product). The query block ordering (region, s2) is evaluated after the function s2 is computed.

We will now illustrate the analytical calculations under each category where we demonstrate the flexibility of the aggregate group clause.

5 Families of Analytical Functions

5.1 Ranking Functions

These functions assign ranks to rows based on some ordering criteria. Functions in this category are rank, dense_rank, row_number, ntile, percent_rank, and cume_dist. For example, rewriting query Q1 as:

```
SELECT * FROM
 (SELECT region, salesperson,
    SUM(sales) sum_sales,
    RANK() OVER
       (PARTITION BY region
        ORDER BY sum(sales) DESC) rank
 FROM salesTable
 GROUP BY region, salesperson)
WHERE rank <= 3;
```

may result in the following data:

Table 1: Ranking Functions

Region	SalesPerson	sum_sales	rank
East	John	1000	1
East	Mary	500	2
East	Sheri	200	3
West	Robert	2000	1
West	Annie	2000	1
West	Tom	1100	3

Row_number is a very simple function which assigns consecutive numbers (starting with 1) to rows after ordering the rows. Ntile bucketizes data into specified number of buckets. Percent_rank and cume_dist, like ntile, are tiling functions dividing ordered data into buckets.

5.2 Window Functions

All aggregate functions (sum, count, max etc.) are extended with the window syntax to perform computations over sliding window of rows within a partition. The aggregate group clause is used to precisely define this window. ROWS or RANGE keyword specifies the type of window and specification BETWEEN x PRECEDING and y FOLLOWING determines the size of the window. For example, query Q2 would be represented by:

```
SELECT product, date,
   SUM(sales) sum_sales,
   SUM(SUM(sales)) OVER
      (PARTITION BY product ORDER BY date
         RANGE INTERVAL ’1’ MONTH PRECED-
    ING) mavg
FROM salesTable
GROUP BY product, date;
```

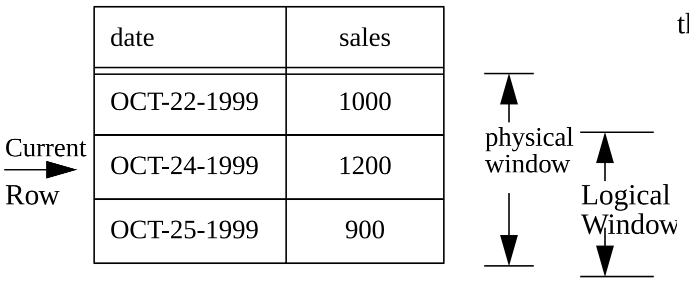
Aggregate function Sum(sales) is computed over the group (product, date). Then the data is partitioned on product and within each partition ordered by date. Next, the aggregate group clause “RANGE

INTERVAL '1' MONTH PRECEDING" is applied to the ordered set for each row, creating windows holding all rows within the last one month of the value in the current row. For each row, the average sales is computed over its corresponding window.

While ROWS specified windows in terms of physical offsets, RANGE specifies a logical window. With the ROWS option, x PRECEDING would imply x rows preceding the current row while with the RANGE option, it implies rows in the logical range [current value - x, current value] assuming ascending order. For example, consider the following two functions:

```
AVG(sales) OVER
   (ORDER BY date
    ROWS 1 PRECEDING
       AND 1 FOLLOWING) phy_avg,
AVG(sales) OVER
   (ORDER BY date
    RANGE INTERVAL '1' day PRECEDING
      AND INTERVAL '1' day FOLLOWING)
       log_avg
```

The figure below illustrates difference between phy_avg and log_avg for a given position of current row. For the former we average three rows, and for the latter only two as there are no rows preceding 'oct-24-1999' by 1 day.



date	sales
OCT-22-1999	1000
OCT-24-1999	1200
OCT-25-1999	900

5.3 Reporting Functions

These are a special kind of window functions where the window ranges from beginning of the partition to the end of the partition. In this case, each row in the partition is reported with an aggregate over the entire partition. By skipping the order-by and aggregate group clauses, reporting function semantics are obtained. For example, query Q3 can be expressed as:

```
SELECT *
FROM (SELECT year(date), product,
             SUM(sales) s_sales,
             MAX(SUM(sales)) OVER
                   (PARTITION BY product)
                    m_sales
      FROM salesTable
     GROUP BY year(date), product)
WHERE s_sales = m_sales;
```

The above query avoids the view definitions and joins that the traditional approach would have required. This is achieved by reporting aggregates at different levels inside an in-line view and performing comparisons in the outer view as shown above. A similar approach has been proposed in [CR96]. The

region	product	s_sales	m_sales
East	P1	1500	2000
East	P2	1300	2000
East	P3	2000	2000
East	P4	1000	2000

result of Query Q3 is shown in the highlighted row of the table above.

Similary, query Q4 can be expressed as:

```
SELECT *
FROM (SELECT region, state, product,
           SUM(sales) s_sales,
           SUM(SUM(sales)) over
             (PARTITION BY region)     s1,
           SUM(SUM(sales)) OVER
         (PARTITION BY region, state) s2,
           RANK() OVER
             (PARTITION BY region, state
             ORDER BY SUM(sales) DESC) rank
      FROM salesTable
     GROUP BY region, state, product)
WHERE s2 >= 0.25 * s1 AND rank <= 10;
```

5.4 Lag/Lead Functions

Lag/Lead functions provide an access to any row at a certain offset from the current row in the ordered dataset. These functions can be used to compare measures across different rows without requiring expensive self-joins. For example, SQL for query Q5 would be:

```
SELECT product, month(date),
   SUM(sales) sum_sales,
   SUM(sales) - LAG(SUM(sales), 1) OVER
                (PARTITION BY product
               ORDER BY month(date))  diff
FROM salesTable
GROUP BY product, month(date);
```

Lag (Lead) allow access to rows at an offset before (after) the current row in an ordered sequence. Note that Lag/Lead functions are most suitable for dense data where physical offsets correspond to logical units.

6 Optimized Execution of Analytic Functions

In this section, we present optimizations[4] employed in the implementation of the analytic functions in Oracle 8*i* Release 2.

6.1 Minimization of number of sorts

A given SQL query can contain multiple analytic functions, each requiring its own ordering specification, e.g:

```
SELECT region, state,
   SUM(sales) OVER (PARTITION BY region)
   sum_region,
   SUM(sales) OVER (PARTITION BY region,
   state) sum_region_state,
   RANK() OVER (PARTITION BY region ORDER
   BY sales DESC) rank
FROM salesTable;
```

Here, each of the 3 functions requires different sortings of the base data. But, we can evaluate all the 3 functions by sorting the base data on:

- `(region, state)` which satisfies requirements for functions (1) and (2).
- `(region, sales DESC)` which satisfies function (3).

In all, we require at least 2 sorts.

The analytic functions are evaluated after the computation of "group by" clause but prior to the "order by" clause in the main query block. The sorting is also required for "group by"[5] and "order by" clauses. The ordering done for "group by" could be used by analytic functions and ordering done for analytic functions could be used by "order by" clause of the query. e.g

```
SELECT SUM(sales) s1,
   SUM(SUM(sales)) OVER
      (PARTITION BY region
       ORDER BY state) s2,
   RANK() OVER (ORDER BY SUM(sales)) r1
FROM sales_table
GROUP BY region, state
ORDER BY SUM(sales);
```

This query can be computed using 2 sorts:

- `(region, state)` to compute the group by aggregate s1 and analytic function s2
- `(SUM(sales))` to compute analytic function r1 and query "order by" clause.

The number of sorts could significantly affect the execution time of a query. The problem we are trying to solve is to find a minimal number of sorts which can satisfy a SQL query block.

An ordering Group (OG) is a set of analytic functions which can be satisfied by a single sort. A minimal set of OGs covering all functions is important for efficient execution of the query.

For computation of this set, we need to consider

(4) Patents for these optimizations have been submitted to the US Patents Office.

(5) assuming sort-based aggregation

only the expressions in the PARTITION BY and ORDER BY clauses of the analytic functions. Computation of an analytic function requires ordering (sorting) the data on *<p1>, <p2>,..., <pn>,<o1>, <o2>,...<om>*, where

- the PARTITION BY expressions *<p1>, <p2>, ...<pn>* can be commuted in any order,
- the ORDER BY expressions *<o1>, <o2>, .., <om>* should follow the PARTITION BY expressions and appear in the specified order.

We would represent the query GROUP BY as a dummy analytic function with the PARTITION BY expressions containing the query GROUP BY expressions. This can be done as GROUP by can be computed by ordering on any commutation of GROUP BY expressions and the same property is satisfied by an analytic function with only PARTITION by columns.

Similarly, the query ORDER BY is represented as another dummy analytic function having the same ORDER BY expressions. This is only for the purposes of computing the minimal orderings.

The algorithm involves 2 stages :

```
1. Find a set of OGs which cover all
   the functions.
2. Select a minimal set of OGs from
   the set found in (1) such that all
   the functions are covered.
```

We illustrate the algorithm by an example where we find a minimal set of OGs for a set of analytic functions. Consider 5 functions with the following partition by P() and order by O() clauses:

```
1. P(x, y, z)
2. P(y, x)
3. P(a)
4. O(a, b)
5. P(x, y, a, b) O(z)
```

Here, `a, b, x, y, z` are different expressions which in various combinations dictate the ordering requirements of a given analytic function. For simplicity, we do not discuss the ASC/DESC options in the algorithm, but these can be incorporated easily.

With each OG, we also store "commute_index" which is the set of (prefix) columns common to all the functions in the OG. This can be useful in parallel execution to decide the partitioning of data. For example, an OG (x, y, z) for functions 1 and 2 above would have commute_index = 2 implying that the data can be partitioned on columns (x, y) and each partition would require ordering on (x, y, z).

The algorithm works as follows:

6.1.1 Stage 1

A function with the east number of partition by expressions is most restrictive in terms of ordering requirements. For example, function (2) requires ordering starting with `(x, y)` in any permutation and can possibly be used to satisfy ordering requirement for function (1) if the third expression is `(z)`. In case of 2 functions with same number of `P()` expressions, a function with greater number of `O()` expressions is more restrictive. So, we select such functions and construct OGs around them. This is used to prune the set of OGs.

So, we sort the functions in ascending order of number of P() expressions as primary key and descending order of number of ordering expressions as secondary key.

```
1. O(a, b)
2. P(a)
3. P(y, x)
4. P(x, y, z)
5. P(x, y, a, b) O(z)
```

From this set, we select a set of functions such that OGs containing those functions would cover all functions. This is done by traversing the above list in order and picking any function which would not belong to any OG considered so far.

In this example, we

```
1. choose (1),
2. ignore (2) (can be satisfied by OG
   of (1)),
3. choose (3),
4. ignore (4) (can be satisfied by OG
   of (3)),
5. ignore (5) (can either be satisfied
   by OG of (1) or (3))
```

So, the chosen OGs are the ones starting with `(a, b)` and `(y, x)`. We would form OGs starting with `(y, x)` by considering the subset of functions which can be satisfied by such an ordering after removing `(y, x)` from the `P()/O()` expressions. Functions (3), (4) and (5) satisfy the criterion:

3) P()
4) P(z)
5) P(a, b) O(z)

The same algorithm is applied recursively, and in all we get 2 OGs starting with (y, x):

OG1: `(y, x, z)` - functions (3, 4)
OG2: `(y, x, a, b, z)` - functions (3, 5)

Same algorithm starting with functions (1), (2) and (5) to construct OGs starting with `(a, b)` yields:

OG3 : `(a, b, x, y, z)` - functions (1, 2, 5)

Now, we have 3 OGs. In all, these satisfy all the functions.

6.1.2 Stage 2

Now, we choose a minimal set of OGs.

- `(a, b, x, y, z)` satisfies functions {3,4,5},
- `(y, x, z)` satisfies functions {1,2},
- `(y, x, a, b, z)` satisfies functions {2,5},

We can follow a greedy approach to select the minimal set of OGs. In this example `(a, b, x, y, z)` can be used to compute 3 functions. Now, we remove functions {3,4,5} from all the other OGs to get:

- `(y, x, z)` satisfies function {1,2}
- `(y, x, a, b, z)` satisfies {2}

Next, we would select OG `(y, x, z)` which results in covering all the functions.

The proof that this set is a minimal set follows from construction. Each OG which is selected for the minimal set of OGs satisfies some functions which are not satisfied by already selected OGs. This algorithm of recursively finding the minimum number of partitioning columns and constructing OGs followed by a greedy approach to select a set of OGs can be used to get a minimal set of orderings (OGs).

6.2 Predicate Pushdown Optimizations

Predicates on ranking functions appearing in the outer query block can be pushed into the corresponding sorts (i.e., the predicate is applied when sorting the data), thus resulting in an efficient execution of "top-N" queries. Consider SQL for query Q1 in Section 3:

```
SELECT *
FROM (SELECT region, salesperson,
        SUM(sales) sum_sales,
        RANK() OVER
          (PARTITION BY region
            ORDER BY sum(sales) DESC) rank
      FROM salesTable
      GROUP BY region, salesperson)
WHERE rank <= 3;
```

This optimization pushes the predicate "rank <= 3" into the sort that orders data on (region, sum(sales) desc). Sort applies this predicate per region keeping only top 3 salespersons per region. This not only reduces the chances of sort spilling to disk, but also minimizes the number of sort runs and hence merge passes. This results in an asymptotic complexity of O(n*log N), where n is the number of rows in the dataset and N is the "top-N" value. Functions that are pushed into sort currently are RANK, DENSE_RANK and ROW_NUMBER.

The rules governing as to when rank functions with predicates should be evaluated and when it is legal to push rank predicates into the sort are non-trivial. This is because, in the presence of other analytic functions, this predicate pushdown should not filter rows needed later. The following example shows why the execution order is important:

```
SELECT *
FROM (SELECT region, salesperson,
        SUM(sales) sum_sales,
        RANK() OVER (PARTITION BY region
           ORDER BY sum(sales) DESC) rank,
        SUM(SUM(sales))OVER
        (PARTITION BY region) rep_sum_sales
    FROM salesTable
    GROUP BY region, salesperson)
WHERE rank <= 3;
```

The execution sequence where in step 2 we filter out ranks:

```
1. order data on (region, sum(sales)
   desc)
2. compute ranks and filter out
   salespersons other than top 3
3. compute sum(sum(sales)) over
   (PARTITION BY region)
```

is incorrect as the reporting aggregate function "sum" needs to access all rows per region to compute rep_sum_sales. Observe that computing rep_sum_sales first and then rank with filtering would result in correct semantics. We follow a simple rule that schedules the evaluation rank functions with top N predicates in the end. Also, the ordering group optimization described in section 3.1.1 puts rank and other analytic functions in the same ordering group - which means data will be ordered on (region, sum(sales) desc) and rank and rep_sum_sales would be computed together. In this scenario, rank predicates can not be pushed into sort as it filter out rows necessary to compute rep_sum_sales. The rules deciding when it is legal to push rank predicates are:

Consider a rank function R = rank() over (PARTITION BY x ORDER BY y) and suppose there is a predicate C_r on that function in the outer query block. Without loss of generality we assume that x and y are single columns (extension to the multi column keys is obvious). We assume that the predicate is of the form "R {<, <=, =} <constant>" as then sort filters out records early.

Suppose further that the query block has other analytic functions F_1, ..., F_n which are in the same ordering group OG as R. Recall that R, F_1, ..., F_n can be evaluated with a single sort. Let P_r, O_r represent PARTITION BY and ORDER BY keys of R respectively and let $P_r||O_r$ denote concatenation of P_r and O_r.

The rules below are sufficient conditions to push predicate C_r into the sort for the ordering group OG. They implement the intuition that filtering on a rank predicate must not remove rows that are required to compute other functions in the same ordering group.

1. All reporting functions in the OG must have same or finer granularity PARTITION BY clause than $P_r||O_r$. Thus C_r will not filter out rows inside partitions of these reporting functions. Instead it may filter out the entire partitions which is correct. For example, predicate C_r can be pushed with functions like "sum(m) over (PARTITION BY x, y)" but it can not be pushed with "sum(m) over (PARTITION BY x)".
2. All ranking functions in the OG must either have finer granularity PARTITION BY clause than P_r, or have same PARTITION BY clause but same or finer ORDER BY clause than O_r. The justification is as above.

For example, predicate C_r can be pushed with "dense_rank() over (PARTITION BY x, y ORDER BY z)" or " rank() over (PARTITION BY x ORDER BY y, z)", but can not be pushed with "rank() over (ORDER BY x, y)".

3. All window functions in the OG with the ROWS option must have same or finer granularity PARTITION BY clause than P_r and their windows should not extend to rows past the current row. Observe that C_r is of the form R {<, <=, =} <constant>, thus it doesn't filter rows above some <constant> rank, thus window functions can extend before the current row. If they extended past it, the predicated could have filter out the needed rows. For example, predicate C_r can be pushed in case of functions like "sum(m) over (PARTITION BY x ORDER BY y ROWS 1 PRECEDING", or "sum(m) over (PARTITION BY x, y ORDER BY z ROWS BETWEEN 10 PRECEDING AND CURRENT ROW)". It can not be pushed for functions like "sum(m) over (PARTITION BY x ORDER BY y ROWS 10 FOLLOWING)".
4. All window functions in the OG with RANGE option must have same or finer granularity PARTITION BY clause than G_r and their windows must not extend to rows past the current row. Justification is as above. That is, pushing can happen with functions like "sum(m) over (PARTITION BY x ORDER BY y RANGE 1 PRECEDING)" but it can not happen with functions like "sum(m) over (PARTITION BY x ORDER BY y RANGE 1 FOLLOWING)".

Similar rules apply for pushing predicates on reporting functions. Let C_r be a predicate on a reporting *R* with a partition by clause P_r and let R belong to ordering group OG. C_r can be pushed into the sort for OG if analytic functions from *OG* must have the same or higher granularity partition by clause than P_r. J Observe that since R partitions by on a same or coarser granularity than other functions in the group, then C_r will be filtering groups which include entire groups of other functions. For example: a predicate on 'sum(m) over (PARTITION BY X)' can be pushed into the sort if other analytic functions are 'sum(m) over (PARTITION BY x, y)', or 'rank() over (PARTITION BY x, ORDER BY y)', etc.

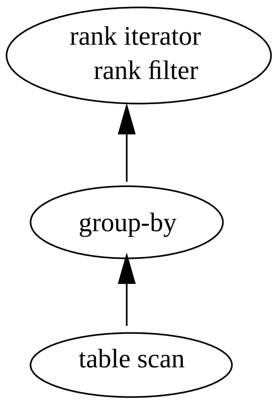
6.3 Parallel Execution

The basic building block of Oracle's SQL execution plan and engine is an iterator. A iterator is an object-oriented mechanism for manipulating rows of data. Associated with each iterator is a start, fetch, and close method. Examples of iterator include table scans, joins, count, union. Iterator nodes are combined together in a tree that implement the query logic and the decisions (e.g. join order, join method, access method) made by the optimizer. At execution time, the serial plan is evaluated in a demand-driven fashion, by fetching all rows from the root node of the iterator tree.

For more on the iterator model of SQL execution please refer to [GG93].

For the parallel plan, consider an execution model with iterators as nodes, edges representing flow and repartitioning of rows up a tree and horizontal data partitioning to distribute load amongst parallel processes at each level. Each edge incurs a communication cost depending on the parallel hardware architecture on which the tree is executed.

Consider an iterator tree for the example query Q1 in Section 3:

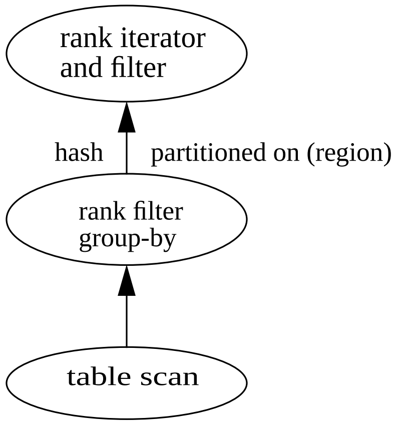


Hash partitioning: The re-distribution of rows flowing from the GROUP BY node to the RANK node is done by a hash on PARTITION BY column of the RANK function. In our experience, the number of unique values of the PARTITION BY columns is usually large enough for hashing to give good enough load balancing amongst the parallel processes computing RANK.

Pushdown Optimization:

For the query Q1, the last row source in the query tree is the Top-3 RANK computation. An optimization to reduce communication costs in the redistribution from GROUP BY to RANK is to perform RANK-based filtering in the GROUP BY node itself since the only the Top-3 ranking rows from each GROUP-BY process could be candidates for the Top-3 ranking rows in the overall result. That is, each process computing the GROUP BY aggregate also computes RANK locally and only communicates the Top-3 ranking rows upwards.

Thus only a subset of rows are communicated across the link between the GROUP-BY and RANK nodes. To make sure that this subset is small enough, we turn this optimization on only if the size of each group (in the GROUP-BY) is at least 'k' times that of N (as in Top-N, i.e. N =3 here). So, we filter out on an average at least (1-1/k)-fraction of the rows in each group in the GROUP-BY node before communicating them. The factor 'k' is tunable based on how reduction in communication cost offsets the additional computation of RANK in the GROUP-BY node. Of course, the final RANK computation is done in the top RANK node as before. The resultant execution plan looks as follows.

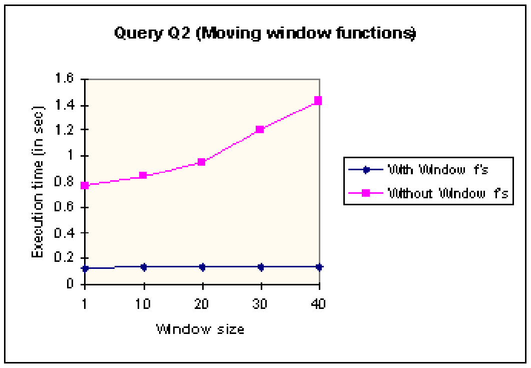


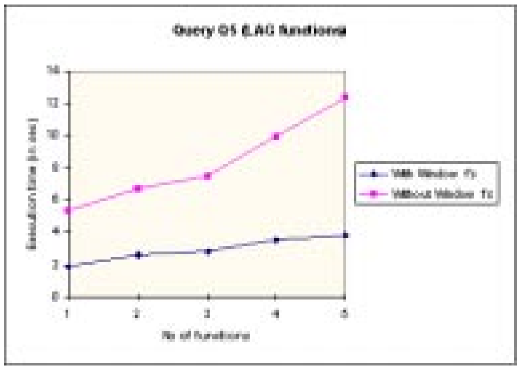
7 Experimental Evaluation

We implemented the functions with associated optimization algorithms in Oracle 8i Release 2. We ran the analytic function queries Q1-Q5 described in Section 3 on the salesTable schema, both by using the new functions and by simulating those in SQL92, sometimes using PL/SQL. The tests were conducted on a 16-processor Sun E4000 on a salesTable of size 20MB having 1 million tuples. The results are shown in Table 2. We measured only the elapsed time in running the query (result output time was excluded). The table had no indices on it as we did not want to influence the execution times by having indices which would favor one of the two approaches. The queries were run in parallel with degree of parallelism set to 48 (this is the maximum number of processes active at any (pipelined) stage of execution).

Table 2: Query execution times

Query id	**Approach 1**: With Analytic function (in sec)	**Approach 2:** Without Analytic functions (in sec)	Simulation method for the queries
Q1	4.57	18.48	Procedural package (PL/SQL)
Q2	3.43	5.73	Non-equi self join
Q3	3.72	7.37	Joins of 2 views with different levels of aggregation
Q4	5.45	26.79	Joins of 2 views and base salesTable, along with PL/SQL for rank
Q5	2.59	5.49	Equi self join





The observations are as follows:

- In Q1, rank function in PL/SQL makes rank assignment inherently serial (ordering can still be done in parallel) and hence, we lose out on the opportunity to push the Top-3 rank selection condition into the sort.

- In Q2, nested loop non-equi self join to compute the moving window function makes the performance worse as compared to using the analytic function directly which just partitions and sorts the data based on PBY and OBY in the window specification.

- Q3 in the second approach requires aggregates SUM at (product, year) and MAX at (year) level followed by a join. Equivalent query using reporting functions avoid the last join.

- Q4 without analytic functions suffers from the problems mentioned in Q1 and Q3.

- Q5 requires access to sales value of previous year and self join can be easily avoided using LAG function.

We also tried to compare performance degradation with increase in the window size using both of the above mentioned approaches. This could lead to increased fan-out of non-equi joins (as in Q2). Query Q2 was run with varying window sizes (we used physical windows here so that we know the exact size of window while trying to compare the performance). The window sizes were small enough for the whole window to fit in memory. In some other tests, we observed that even when the window doesn't fit into memory the execution times asymptotically approach the same value (within 15% of running times for those with small window sizes). As can be seen in the graph, increase in window size leads to degradation in execution times without using window functions.

We also studied the performance degradation by increasing the number of analytic functions which could lead to increase in number of self-joins (as in Q5). Query Q5 was run by increasing the number of LAG/LEAD functions. Increased number of LAG/ LEAD functions correspond to greater number of self-joins in the second approach. View V3 used in this query was materialized on disk for purposes of self-join. We can observe much higher negative impact on query performance due to self-joins from the graph.

8 Related Work

SQL extensions with limited analytical functionality exists in other commercial RDBMSs. Redbrick RISQL [Red60] provides functions for computing moving aggregates, ranking and ratiotoreport. However, moving aggregates can be computed only on physical windows (determined by number of rows included in the set) which is not suitable for sparse data. Further, the data set can only be partitioned once using the RESET BY clause since it is tied to the ORDER BY clause of the query. Teradata [TerR30] also provides support of physical window functions and ranking functions. Microsoft SQL Server [MS70] does support limited ranking functionality (TopN), but none of the window functions and partitioning capabilities presented in the paper. For any ROLAP engine which is served by a back-end RDBMS, the SQL needed to support such queries would be very complex and quite inefficient. Sybase[Syb120] allows selection on non-GROUP BY columns and hence, some of the reporting aggregates queries mentioned in this paper. The SQL extensions in [CR96] using grouping variables allow succinct representation of queries involving repeated aggregation over the same groups and can be executed efficiently. The extensions proposed in this paper are similar to those of [CR96] with respect to allowing multiple passes over the data. The proposal goes a step farther by allowing aggregates on ordered sets of data.

9 Conclusion

We have presented a unified framework for analytic functions in SQL which operate on partitioned and ordered data. These include five types of functions - ranking, windowing, reporting functions, lag/lead functions and inverse distribution functions. We have also discussed algorithms for minimizing the number of sorts required by the Analytic Functions and shown optimizations like pushing rank predicates into the sort. Finally, we have provided experimental evidence showing that our optimized implementation provides substantial performance gains over existing implementation of same functionality in SQL. These efforts are an initial step towards supporting a rich and efficient set of OLAP queries in the Oracle RDBMS server.

10 References

[1] [Red60] Informix Redbrick Decision Server 6.0
[2] [TerR30] NCR Teradata V2R3.0.3
[3] [MS70] Microsoft SQL Server 7.0
[4] [Syb120] Sybase Adaptive Server Enterprise 12.0
[5] [CR96] Damianos Chatziantoniou, Kenneth Ross. Querying Multiple Features of Groups in Relational

Databases. In Proceedings of the 22nd VLDB conference Mumbai (India), 1996.

[6] [OraIBM99] Fred Zemke, Krishna Kulkarni, Andy Witkowski, Bob Lyle. Proposal for OLAP functions for ANSI-NCTS

[7] [GG93] Goetz Graefe: Query Evaluation Techniques for Large Databases. Computing Surveys 25(2): 73-170 (1993).

11 APPENDIX (More Analytic Functions)

11.1 Motivating Examples

Here are other queries which are difficult and inefficient to express in SQL.

Q6 For each product, find total sales on the first day a sale is recorded

This can be done by first computing the first day of sales for each product and then, joining it back to the salesTable on (product, date).

```
CREATE VIEW V4
AS SELECT product, MIN(date) m_date
FROM  salesTable
GROUP BY product;

SELECT V4.product, SUM(sales) s_sales
FROM V4, salesTable
WHERE V4.product = salesTable.product AND
   V4.m_date = salesTable.date
GROUP BY V4.product;
```

Q7 For each (region, product), find the median sales figure.

This query does not have a representation in SQL or PL/SQL. We require support for order-dependent aggregate functions inside the RDBMS to compute them.

Hence, we define inverse distribution and first/last functions in the following section. These aggregate functions can also be used as reporting functions (defined in Section 5.3).

11.2 Inverse Distribution Functions

This includes percentile function in two flavors.

PERCENTILE_DISC function assumes a discrete distribution model and returns an element from the ordered set which has the smallest discrete percentile value that is larger than or equal to the percentile value given as an argument of the function. For example, query Q7 can be easily expressed using PERCENTILE_DISC as:

```
SELECT
PERCENTILE_DISC(0.5) WITHIN GROUP
   (ORDER BY sales) median_sales,
FROM salesTable
GROUP BY region, product;
```

PERCENTILE_CONT function assumes a continuous distribution model and computes the result by doing linear interpolation of the two elements whose continuous percentile values enclose the percentile value given as the argument of the function. Since the returned value is a computed value, only numeric or date values are allowed in the ORDER BY clause.

11.3 FIRST/LAST aggregates

These functions allow us to compute aggregates on top-most (first) or bottom-most (last) rows of an ordered set within a group.

The KEEP clause ranks the ordered data and applies the specified aggregate function to the rows that rank first (or last).

For example, query Q6 can be expressed as:

```
SELECT product, YEAR(date),
  SUM(sales)
     KEEP (DENSE_RANK FIRST ORDER BY
         day(date)) first_day_sales
FROM salesTable
GROUP BY product, YEAR(date);
```