NEW SQL OLAP FUNCTIONS FOR EVERYONE
Kenneth M. Guion, QED Solutions, Incorporated

Introduction
Did you ever want to find the top two salary earners in each department? Ever want to calculate a three-month moving average? Now starting with Oracle 8.1.6, you can write the SQL statements you have always wanted and the one’s you have never thought possible. Oracle has added a plethora of new analytical and statistical functions that can be called by any SQL statement in any application. This paper will help you learn about new OLAP (on-line analytical procession) and Business Intelligence functions that have been introduced since you first learned Oracle. It will specifically cover the basics of Oracle’s new Analytic Functions such as ROLLUP and CUBE; new functionality for our old Aggregate functions such as SUM and AVG; and how to use new “windowing” and “lag/lead” functionality to calculate cumulative totals, moving averages, and inter-row calculated values such as period-to-period growth. By utilizing these new built in functions, everyone can add a little DSS (Decision Support System) and Data Warehousing features into any application.

Environment
This paper is based on Oracle8i Release 2 (8.1.6) or later. Some of the commands and options in this paper such as “In-Line” views have been around since late Oracle 7. The examples in this paper use three simple tables: CLERKS, CLERK_DAILY SALES, and TOTAL SALES. The examples are also designed to demonstrate a specific feature, even if they are impractical or possibly a better way to achieve the same thing exists.

Example Schema

```
SQL> DESCR clerks
Name      Null?    Type
---------  --------  -----------
CLERK     NOT NULL VARCHAR2(6)
AGE       NOT NULL NUMBER

SQL> DESCR clerk_daily_sales
Name      Null?    Type
---------  --------  -----------
SDATE     NOT NULL DATE
CLERK     NOT NULL VARCHAR2(6)
QTY       NUMBER

SQL> DESCR total_sales
Name      Null?    Type
---------  --------  -----------
CLERK     NOT NULL VARCHAR2(6)
PRODUCT   VARCHAR2(7)
QTY       NUMBER
```
Aggregate Functions

Some of our applications have large amounts of data that simply cannot be interpreted at their lowest level. They must be analyzed and viewed at a summary or aggregate level to be able to make decisions efficiently. A common requirement of these DSS applications is to provide this aggregate data across many dimensions of the data. A dimension is a method of categorizing data such as geography, time, product, etc.

To do this, Oracle has added two new Aggregate Functions ROLLUP and CUBE. Each allows a standard SELECT statement to return subtotals at increasing levels of aggregation. These new functions will not only simplify your SQL code, but the resulting queries will be quicker and more efficient. Traditional methods were typically convoluted, contain multiple accesses to the same table(s), and are often difficult to optimize.

ROLLUPS

Suppose we wanted a report that showed the total sales for each product by clerk. Additionally, we wanted to see the total sales for each clerk and a grand total for all clerks. In the “good ol’ days” of COBOL, this was commonly referred to as a control-break report. The dimensions in this case would be both clerk and product. In order to achieve this report using a pure SQL solution, we would typically use three queries (UNION-ed together). The first query provides the clerk-product totals; the second, the clerk totals; and the third, the grand total.

Now with Oracle 8i we can accomplish the same thing using a ROLLUP. A ROLLUP is an extension to the GROUP BY clause used to calculate and return subtotals and a grand total as additional rows of the query efficiently. These additional rows are the rows that would be created by the two UNION SELECT portions of a pure SQL solution. The new ROLLUP operation creates these rows with only one access to the TOTAL_SALES table versus the traditional UNION method, which would have had to access the TOTAL_SALES table three separate times.

Simple ROLLUP Example

```sql
SQL> SELECT clerk, product, SUM(qty) qty
2  FROM total_sales
3  WHERE clerk IN ('SCOTT', 'FRED')
4  GROUP BY ROLLUP (clerk, product);

<table>
<thead>
<tr>
<th>CLERK</th>
<th>PRODUCT</th>
<th>QTY</th>
</tr>
</thead>
<tbody>
<tr>
<td>FRED</td>
<td>APPLE</td>
<td>5</td>
</tr>
<tr>
<td>FRED</td>
<td>BANANA</td>
<td>9</td>
</tr>
<tr>
<td>FRED</td>
<td>GRAPE</td>
<td>2</td>
</tr>
</tbody>
</table>
| FRED  |         | 16  | ← New Subtotal row created by ROLLUP function
| SCOTT | APPLE   | 5   |
| SCOTT | BANANA  | 2   |
| SCOTT | GRAPE   | 7   |
| SCOTT |         | 14  | ← New Subtotal row created by ROLLUP function
|      |         | 30  | ← New Grand total row created by ROLLUP function

9 rows selected.
```

A ROLLUP produces progressive subtotals for each column in the ROLLUP operation moving right to left. Again, in our example, ROLLUP will produce a subtotal for each product within a clerk, a subtotal for each clerk, and a grand total for all clerks. Although a ROLLUP can be achieved using client side tools such as SQL*PLUS using BREAK and COMPUTE, these tools can place a significant and unnecessary load on the middle or client tier. The ROLLUP command places the load on the database tier (where it belongs).
CUBEs

Consider the case where you want to get subtotals not only for each clerk, and product within clerk, but also for each product across clerks. The CUBE operator works similar to the ROLLUP operator, but creates subtotals for all possible combinations of the columns contained in the CUBE list. CUBE is particularly helpful when your dimensions are not part of the same hierarchy (i.e. day, month, year versus city, state, country). Note, however, that ROLLUPs and CUBEs are independent of any hierarchy meta-data that can now be stored in dictionary for query rewrites etc.

Simple CUBE Example

```
SQL> SELECT clerk, product, SUM(qty) qty
2  FROM total_sales
3  WHERE clerk IN ('SCOTT','FRED')
4  GROUP BY CUBE(clerk, product);
```

<table>
<thead>
<tr>
<th>CLERK</th>
<th>PRODUCT</th>
<th>QTY</th>
</tr>
</thead>
<tbody>
<tr>
<td>FRED</td>
<td>APPLE</td>
<td>5</td>
</tr>
<tr>
<td>FRED</td>
<td>BANANA</td>
<td>9</td>
</tr>
<tr>
<td>FRED</td>
<td>GRAPE</td>
<td>2</td>
</tr>
</tbody>
</table>
| FRED  |         | 16   | \(\uparrow\) Sub total row created by ROLLUP or CUBE function
| SCOTT | APPLE   | 5    |
| SCOTT | BANANA  | 2    |
| SCOTT | GRAPE   | 7    |
| SCOTT |         | 14   | \(\uparrow\) Sub total row created by ROLLUP or CUBE function
| APPLE |         | 10   | \(\uparrow\) NEW Sub total row created only by CUBE function
| BANANA|         | 11   | \(\uparrow\) NEW Sub total row created only by CUBE function
| GRAPE |         | 9    | \(\uparrow\) NEW Sub total row created only by CUBE function
|       |         | 30   | \(\uparrow\) Grand total row created by ROLLUP or CUBE function

12 rows selected.

Subtotals created by CUBE would be synonymous with those created for a cross-tab or matrix type report. In the example above, you see that three additional rows (the product sub totals) were created by the use of CUBE rather than ROLLUP. In a traditional pure SQL solution, even another table access would be needed for a total of four scans of the CLERKS table to create a comparable result versus using the CUBE function.

Grouping Functions

What happens when one of the columns that you are aggregating on allows a NULL value? The question then becomes “does the NULL value for a column indicate a newly created subtotal (or aggregate) row or is it a normal row that simply has a NULL value for that column?”

In order to help distinguish what rows are subtotals, Oracle created the GROUPING function. GROUPING returns the value “1” if the row is a subtotal or grand total row created by the ROLLUP or CUBE operator and returns a “0” if it is a normal row returned by the query. Consider the example below using the GROUPING function.
Distinguishing a NULL column value from an Aggregate Total row using the GROUPING function

```sql
SQL> SELECT 
    2  clerk, GROUPING(clerk) gc, 
    3  product, GROUPING(product) gp, 
    4  SUM(qty) qty 
    5  FROM total_sales 
    6  WHERE clerk IN ('JEFF', 'TIM') 
    7  GROUP BY ROLLUP(clerk, product);
```

<table>
<thead>
<tr>
<th>CLERK</th>
<th>GC</th>
<th>PRODUCT</th>
<th>GP</th>
<th>QTY</th>
</tr>
</thead>
<tbody>
<tr>
<td>JEFF</td>
<td>0</td>
<td>APPLE</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>JEFF</td>
<td>0</td>
<td>GRAPE</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>JEFF</td>
<td>0</td>
<td>&lt;NULL&gt;</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>JEFF</td>
<td>0</td>
<td>&lt;NULL&gt;</td>
<td>1</td>
<td>9</td>
</tr>
<tr>
<td>TIM</td>
<td>0</td>
<td>APPLE</td>
<td>0</td>
<td>7</td>
</tr>
<tr>
<td>TIM</td>
<td>0</td>
<td>BANANA</td>
<td>0</td>
<td>8</td>
</tr>
<tr>
<td>TIM</td>
<td>0</td>
<td>GRAPE</td>
<td>0</td>
<td>9</td>
</tr>
<tr>
<td>TIM</td>
<td>0</td>
<td>&lt;NULL&gt;</td>
<td>1</td>
<td>24</td>
</tr>
<tr>
<td>&lt;NULL&gt;</td>
<td>1</td>
<td>&lt;NULL&gt;</td>
<td>1</td>
<td>33</td>
</tr>
</tbody>
</table>

Even though PRODUCT column is NULL, the grouping function returns a zero indicating a normal row.

Grouping function returns a one; therefore this is a subtotal row created by the ROLLUP function.

The extra rows created by the ROLLUP and CUBE statements are created during the GROUP BY operation; and therefore, the HAVING clause can be used with the GROUPING function to filter results to include or exclude certain aggregate or (subtotal) rows.

Here is an example of a query that retrieves only the extra subtotal and grand total rows created by the CUBE statement.

**Using HAVING clause to filter out Non Aggregate CUBE rows**

```sql
SQL> SELECT clerk, product, SUM(qty) qty 
    2  FROM total_sales 
    3  WHERE clerk IN ('SCOTT', 'FRED') 
    4  GROUP BY CUBE(clerk, product) 
    5  HAVING GROUPING(clerk) = 1 OR GROUPING(product) = 1;
```

<table>
<thead>
<tr>
<th>CLERK</th>
<th>PRODUCT</th>
<th>QTY</th>
</tr>
</thead>
<tbody>
<tr>
<td>FRED</td>
<td></td>
<td>16</td>
</tr>
<tr>
<td>SCOTT</td>
<td></td>
<td>14</td>
</tr>
<tr>
<td>APPLE</td>
<td></td>
<td>10</td>
</tr>
<tr>
<td>BANANA</td>
<td></td>
<td>11</td>
</tr>
<tr>
<td>GRAPE</td>
<td></td>
<td>9</td>
</tr>
<tr>
<td></td>
<td></td>
<td>30</td>
</tr>
</tbody>
</table>

Sub total row created by ROLLUP or CUBE function
Sub total row created only by CUBE function
Sub total row created only by CUBE function
Sub total row created only by CUBE function
Grand total row created by ROLLUP or CUBE function

In all of our examples above, I have used the SUM function. While it is probably the most common, you can use other functions such as COUNT, AVG, MIN, MAX, etc. And since the ORDER BY is the last operation performed, the subtotal and grand total rows are sorted among the rest of the rows returned by the query. Be careful to sort these rows into a logical position so that they can be properly interpreted by the user.
Analytic Functions

Just beyond ROLLUP and CUBE is a new family of functions commonly referred to as Analytic Functions. These functions can be broken down into four groups: Ranking Functions, Reporting Functions, Window Functions, Lag/Lead Functions. As with CUBE and ROLLUP, the new analytic functions also increase developer productivity by minimizing the code that needs to be written. They also are optimized and actually perform better than traditional methods, which typically required convoluted, contained self-joins, and were often difficult to optimize.

RANKING FUNCTIONS

Ranking functions allow us to easily determine how a given row ranks or compares to other rows in the set. Functions included are RANK, DENSE_RANK, CUME_DIST, PERCENT_RANK, NTILE, RATIO_TO_REPORT. Each function allows the data to be ranked based on a single or multiple expressions thereby decreasing the chances of ties.

Now suppose our user wants a report that shows a list of clerks, their sales, and a ranking based on their ‘APPLE’ sales. The syntax looks a little strange, but it’s easy to understand.

Simple Ranking Functions Example

```sql
SQL> SELECT clerk, qty,
  2          RANK() OVER (ORDER BY qty DESC) AS rank,
  3      DENSE_RANK() OVER (ORDER BY qty DESC) AS dense,
  4          RANK() OVER (ORDER BY qty DESC, clerk) AS urank
  5  FROM total_sales
  6  WHERE product = 'APPLE'
  7  ORDER BY qty DESC, clerk;

<table>
<thead>
<tr>
<th>CLERK</th>
<th>QTY</th>
<th>RANK</th>
<th>DENSE</th>
<th>URANK</th>
</tr>
</thead>
<tbody>
<tr>
<td>TIM</td>
<td>7</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>FRED</td>
<td>5</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>SCOTT</td>
<td>5</td>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>JEFF</td>
<td>3</td>
<td>4</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>ALEX</td>
<td>1</td>
<td>5</td>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>
```

In the above example we see that using a non-unique sort specification of QTY causes both FRED and SCOTT who have sales of five to be ranked equally with a ranking of second. The difference between RANK and DENSE_RANK is that DENSE_RANK does not skip any ranking positions after a tie. Looking at our example we see that JEFF has a ranking of fourth using the RANK function, and no one has a ranking of third with the RANK functions since two clerks tied for second. With DENSE_RANK, JEFF is considered third even though two clerks tied for second. If we don’t want any ties, then more columns or expressions will have to be added to the ORDER BY clause to create a unique sort.

Processing Order

Queries with analytic functions are executed in three primary steps. The first step performs all WHERE, GROUP BY and HAVING clauses and sends the results to the second step where the analytic functions are calculated. Finally in the third step, the rows are ordered by the SELECT statements ORDER BY clause. This execution order allows for normal functions...
such as SUM, AVG, COUNT, etc. that are created during the GROUP BY phase to be used in or by the new analytic functions. In addition since ROLLUP and CUBE generated total rows are calculated during the GROUP BY phase, these rows will also be included in any Analytic function processing.

Here we rank clerks, by their total product sales rather than just their ‘APPLE’ sales, and we get the ratio of their individual sales to total sales for the group.

A demonstration of Processing Order and Raito to Report

```
SQL> SELECT clerk, SUM(qty) qty,
2  RANK() OVER (ORDER BY SUM(qty) DESC, clerk) AS rank,
3  RATIO_TO_REPORT(SUM(qty)) OVER () AS ratio
4  FROM total_sales
5  GROUP BY clerk
6  ORDER BY qty DESC, clerk;

<table>
<thead>
<tr>
<th>CLERK</th>
<th>QTY</th>
<th>RANK</th>
<th>RATIO</th>
</tr>
</thead>
<tbody>
<tr>
<td>TIM</td>
<td>24</td>
<td>1</td>
<td>0.33</td>
</tr>
<tr>
<td>FRED</td>
<td>16</td>
<td>2</td>
<td>0.22</td>
</tr>
<tr>
<td>SCOTT</td>
<td>14</td>
<td>3</td>
<td>0.19</td>
</tr>
<tr>
<td>ALEX</td>
<td>9</td>
<td>4</td>
<td>0.13</td>
</tr>
<tr>
<td>JEFF</td>
<td>9</td>
<td>5</td>
<td>0.13</td>
</tr>
</tbody>
</table>
```

Notice now that the RANK function is actually ordered by the Aggregate SUM function that is calculated during the GROUP BY phase. Also note the syntax of the RATIO_TO_REPORT function. There is no ORDER BY clause and the QTY column is a typical looking argument.

**Partitions**

Suppose that we wanted to also provide a ranking of sales within each clerk. To accomplish this task we would need to use the PARTITION BY clause. The first thing you will need to realize is that the PARTITION BY clause of these functions is totally unrelated and independent of Oracle 8’s table partition feature, etc. The PARTITION BY clause breaks the data into numerous datasets based on the list of columns, non-analytic functions, and/or expressions listed. The analytic function is then calculated independently on each partition, which means, for example, that the ranking functions reset their values etc. within each partition.

A single query can have multiple analytic functions, each with a different partitioning scheme. If a query contains an analytic function with no PARTITION BY clause, the whole query is actually treated as one partition.

The following query ranks the combination of clerks and products by their sales, ranks individual clerks by their sales within each product, and ranks individual product sales by each clerk. For those of you who pride themselves in writing very complex SQL, I will leave the pure traditional SQL solution (with no Analytic Functions) to you.
An example of Partitions

```sql
SELECT product, clerk, qty,
  RANK() OVER (ORDER BY qty DESC) AS rank,
  RANK() OVER (PARTITION BY product ORDER BY qty DESC) AS prank,
  RANK() OVER (PARTITION BY clerk ORDER BY qty DESC) AS crank
FROM total_sales
WHERE clerk IN ('TIM','ALEX')
ORDER BY product, clerk;
```

<table>
<thead>
<tr>
<th>PRODUCT</th>
<th>CLERK</th>
<th>QTY</th>
<th>RANK</th>
<th>PRANK</th>
<th>CRANK</th>
</tr>
</thead>
<tbody>
<tr>
<td>APPLE</td>
<td>ALEX</td>
<td>1</td>
<td>6</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>APPLE</td>
<td>TIM</td>
<td>7</td>
<td>3</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>BANANA</td>
<td>ALEX</td>
<td>3</td>
<td>5</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>BANANA</td>
<td>TIM</td>
<td>8</td>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>GRAPE</td>
<td>ALEX</td>
<td>5</td>
<td>4</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>GRAPE</td>
<td>TIM</td>
<td>9</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

As with ROLLUP and CUBE, the rows may be sorted when calculating the analytic function value, but that does not guarantee that the final results will be in the same order. Always use an ORDER BY clause to sort the rows as you desire. The PARTITION BY clause of each analytic function call causes the data to be sorted differently. Also remember that the analytic functions are calculated after the statement’s GROUP BY is calculated so that the partitioning itself could be based on a GROUP BY aggregate result such as SUM.

Ranking Nulls
Oracle treats NULLs as the largest value by default. The new analytic ranking functions, however, give us the choice of placing NULLs at either the top or the bottom of the rankings by specifying the NULLS FIRST or NULLS LAST keywords of the function’s ORDER BY clause. Review the following example, which ranks clerks by their GRAPE sales.

```sql
SELECT clerk, qty,
  RANK() OVER (ORDER BY qty DESC NULLS FIRST) AS DNF,
  RANK() OVER (ORDER BY qty NULLS FIRST) AS ANF,
  RANK() OVER (ORDER BY qty DESC NULLS LAST) AS DNL,
  RANK() OVER (ORDER BY qty NULLS LAST) AS ANL
FROM total_sales
WHERE product = 'GRAPE'
ORDER BY qty DESC;
```

<table>
<thead>
<tr>
<th>CLERK</th>
<th>QTY</th>
<th>DNF</th>
<th>ANF</th>
<th>DNL</th>
<th>ANL</th>
</tr>
</thead>
<tbody>
<tr>
<td>JEFF</td>
<td>1</td>
<td>1</td>
<td>5</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>TIM</td>
<td>2</td>
<td>5</td>
<td>1</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>SCOTT</td>
<td>3</td>
<td>4</td>
<td>2</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>ALEX</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>FRED</td>
<td>2</td>
<td>5</td>
<td>2</td>
<td>4</td>
<td>1</td>
</tr>
</tbody>
</table>
Investigating the DNF (Descending Nulls First) column and the ANF (Ascending Nulls First column, you can see regardless of whether the function is ordered in descending or ascending order, a Null value is always rated first. Likewise with NULLS LAST, a null value is always ranked last regardless of the sort order.

**Top N Statements**

In-Line views is a sub query that you place entirely in the FROM clause and that you give an alias. Any column that you list in the SELECT column list in the sub query, you can use in the parent or encapsulating query. In-line views in Oracle previous to version 8i allowed us to avoid creating unnecessary view schema objects. With Oracle 8i, in-line views now allow ordering.

Since you can use an order by clause in an in-line view it is now possible with Oracle 8i to find the top or bottom few rows of a table easily and efficiently. These queries are referred to as Top-N or Bottom-N queries. Examine the query below, which uses both an in-line view and a ROWNUM predicate to determine the top two clerks for APPLE sales.

```
Top 2 apple clerks using a TOP-N query

SQL> SELECT *
2  FROM
3     (SELECT clerk, qty
4          FROM total_sales
5          WHERE product = 'APPLE'
6          ORDER BY qty DESC) clerk_sales
7  WHERE
8     ROWNUM < 3;

CLERK   QTY
------ ----
TIM       7
SCOTT     5 ← Whoops. FRED also sold 5 apples,
         but ROWNUM predicate throws his row away.

```

Remember that if the ORDER BY clause was moved from the subquery (in-line view) to the top level query, there would be no guarantee that the top rows would be returned since Oracle sorts the rows after the row numbers are assigned. The ORDER BY in the in-line view, however, is executed before the rows are assigned a row number; and therefore, we are guaranteed to get the largest rows.

This query also executes quicker in Oracle 8i as compared to the traditional method used in prior versions. This increase is due to the fact that Oracle recognizes that only two rows are desired so it holds only the two biggest rows fetched so far in memory rather than sorting the entire table. When a bigger row is read, it discards the smaller row and that row is no longer considered or sorted any further.

However, the TOP-N statement approach above guarantees only two rows will be returned. In our case, FRED also sold five apples. We can use a combination of in-line views and analytic functions to produce a report that shows the top two clerks for each product with ties included.
Using an Analytic Function in an In-Line View

```
SQL> SELECT product, srank, clerk, qty
2 FROM
3 (SELECT product, clerk, qty, 
4 RANK() OVER (PARTITION BY product 
5 ORDER BY qty DESC NULLS LAST) AS srank 
6 FROM total_sales 
7 WHERE product IS NOT NULL) clerk_sales
8 WHERE srank < 3 
9 ORDER BY product, srank;
```

<table>
<thead>
<tr>
<th>PRODUCT</th>
<th>SRANK</th>
<th>CLERK</th>
<th>QTY</th>
</tr>
</thead>
<tbody>
<tr>
<td>APPLE</td>
<td>1</td>
<td>TIM</td>
<td>7</td>
</tr>
<tr>
<td>APPLE</td>
<td>2</td>
<td>SCOTT</td>
<td>5</td>
</tr>
<tr>
<td>BANANA</td>
<td>1</td>
<td>FRED</td>
<td>9</td>
</tr>
<tr>
<td>BANANA</td>
<td>2</td>
<td>TIM</td>
<td>8</td>
</tr>
<tr>
<td>GRAPE</td>
<td>1</td>
<td>TIM</td>
<td>9</td>
</tr>
<tr>
<td>GRAPE</td>
<td>2</td>
<td>SCOTT</td>
<td>7</td>
</tr>
</tbody>
</table>

Nested rank query allows us to include FRED with 5 apple sales to be included.

Reporting Aggregate Functions

Whereas the Ranking functions introduced totally new functions such as RANK and RATIO_TO_REPORT that could be calculated for a query or partition, existing Aggregate functions such as SUM and AVG have been enhanced and can now be calculated over a query or partition. Before, these functions caused only one row per group to be returned, now with the use of the OVER clause, these functions can be used to place these values as columns and as part of each row in the query.

As with the Ranking Functions, each Reporting Aggregate Functions can be partitioned differently; and therefore, the Reporting Aggregate Function would also be reset at each partition border.

Simple Aggregate Function Example

```
SQL> SELECT product, clerk, qty, 
2 SUM(qty) OVER () AS sum, 
3 SUM(qty) OVER (PARTITION BY product) AS psum 
4 FROM total_sales 
5 WHERE clerk IN ('SCOTT','FRED') 
6 ORDER BY product, clerk;
```

<table>
<thead>
<tr>
<th>PRODUCT</th>
<th>CLERK</th>
<th>QTY</th>
<th>SUM</th>
<th>PSUM</th>
</tr>
</thead>
<tbody>
<tr>
<td>APPLE</td>
<td>FRED</td>
<td>5</td>
<td>30</td>
<td>10</td>
</tr>
<tr>
<td>APPLE</td>
<td>SCOTT</td>
<td>5</td>
<td>30</td>
<td>10</td>
</tr>
<tr>
<td>BANANA</td>
<td>FRED</td>
<td>9</td>
<td>30</td>
<td>11</td>
</tr>
<tr>
<td>BANANA</td>
<td>SCOTT</td>
<td>2</td>
<td>30</td>
<td>11</td>
</tr>
<tr>
<td>GRAPE</td>
<td>FRED</td>
<td>2</td>
<td>30</td>
<td>9</td>
</tr>
<tr>
<td>GRAPE</td>
<td>SCOTT</td>
<td>7</td>
<td>30</td>
<td>9</td>
</tr>
</tbody>
</table>

Reviewing the above query, we see that a column SUM contains the total of all the products and clerks sales. By definition this column would have the same value for all columns of the query. The PSUM (Product Sum) column shows the total sales for that product.
With the use of In-Line views, comparisons can also be made between an aggregate value and the individual detail row values. For example, comparisons can be made between the average sales and an individual clerk’s sales. The following query returns all clerks who sold above the average.

### Comparing Reporting Aggregate values to detail values example

```sql
SQL> SELECT clerk, qty, clerk_avg
2  FROM (SELECT clerk, SUM(qty) qty,
3       AVG(SUM(qty)) OVER () AS clerk_avg
4       FROM total_sales
5       GROUP BY clerk) clerk_sales
6  WHERE qty > clerk_avg
7  ORDER BY qty DESC;
```

<table>
<thead>
<tr>
<th>CLERK</th>
<th>QTY</th>
<th>CLERK_AVG</th>
</tr>
</thead>
<tbody>
<tr>
<td>TIM</td>
<td>24</td>
<td>14.4</td>
</tr>
<tr>
<td>FRED</td>
<td>16</td>
<td>14.4</td>
</tr>
</tbody>
</table>

### Window Aggregate Functions

Windows are used to calculate cumulative and moving average values for each row. If you specify an ORDER BY in your Analytic Function, your Reporting Aggregate Function becomes a Window Aggregate Function. Specifically windows can be used to limit the rows in the query (or partition) that are used in calculating the analytic function. In fact all analytic functions operate in a window. If one is not defined, the whole partition or query is considered the window.

Simply specifying a starting point and/or ending point defines a window, and these points may move or be fixed. A window can be as large as the whole partition (or query) or as small as a one row. Windows are either Physical or Logical, and Logical windows are either based on a Time Interval or a Value Range.

Physical windows are identified by the `ROWS` keyword and logical by the `RANGE` keyword. The span of rows included in the window is defined using the `PRECEDING`, the `FOLLOWING`, or the `BETWEEN start AND end` phrase. To specify a window that always starts with the first row use the `UNBOUNDED` with the `PRECEDING` keyword as the start. Like wise use `UNBOUNDED FOLLOWING` to specify a window that always ends with the last row in the partition (or query). Finally, `CURRENT ROW` can be used to specify that the window begins or ends at the current row.

A sliding window would be used to calculate items such as a moving average, whereas a running total would be calculated by defining a window with a fixed start point of the first row (`UNBOUNDED PRECEDING`) and the last row being the current row (`CURRENT ROW`).

### Running Total

Simply simply specifying a unique ORDER BY clause in a normal Reporting Aggregate Function easily creates a running total. It will assume the default window definition, which is a logical window with the starting point of the first row and an ending point of the last row (i.e. `RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW`).
Simple Window example to calculate a Running Total

```sql
SQL> SELECT TO_CHAR(sdate,'Dy, Mon DD') s_date, qty,
2   SUM(qty) OVER () AS sum,
3   SUM(qty) OVER (ORDER BY sdate) AS rsum  /* Just added an order by clause */
4   FROM clerk_daily_sales
5   WHERE clerk = 'FRED'
6   ORDER BY sdate;

S_DATE       QTY  SUM  RSUM
-----------  ---- ---- -----  
Thu, Jan 11  2   30     2  
Fri, Jan 12  1   30     3  RSUM = 2 + 1
Tue, Jan 16  6   30     9  RSUM = 2 + 1 + 6
Wed, Jan 17  5   30    14  ...
Thu, Jan 18  4   30    18
Fri, Jan 19  3   30    21
Mon, Jan 22  9   30    30
```

The results are kind of surprising without any definition of the starting point or ending point of the window. In the above example, by simply adding an ORDER BY clause to the Aggregate Function, we have created a Window and therefore a running total.

**Physical Windows**

Physical windows are defined by using the ROWS keyword and by specifying the actual (“physical”) number of rows (called the “offset”) before and after the current row that should be included in the window. The offset must evaluate to a positive integer. Physical windows can be ordered based on multiple columns or expressions. For physical windows, the ORDER BY expression should result in a unique ordering so that the result given is “deterministic” (or repeatable).

Let’s say that we want a query that shows us a moving sum of the last four weekdays regardless of whether there were sales or not.
Physical Window Example – Running Total of Last Four Days of Sales

```sql
SQL> SELECT TO_CHAR(sdate,'Dy, Mon DD') s_date, qty,
    2  SUM(qty) OVER ( ORDER BY sdate
    3  ROWS 3 PRECEDING) AS saledy
    4  FROM clerk_daily_sales
    5  WHERE clerk IN ('FRED')
    6  ORDER BY sdate;
```

<table>
<thead>
<tr>
<th>S_DATE</th>
<th>QTY</th>
<th>SALEDY</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thu, Jan 11</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Fri, Jan 12</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>Tue, Jan 16</td>
<td>6</td>
<td>9</td>
</tr>
<tr>
<td>Wed, Jan 17</td>
<td>5</td>
<td>14</td>
</tr>
<tr>
<td>Thu, Jan 18</td>
<td>4</td>
<td>16</td>
</tr>
<tr>
<td>Fri, Jan 19</td>
<td>3</td>
<td>18</td>
</tr>
<tr>
<td>Mon, Jan 22</td>
<td>9</td>
<td>21</td>
</tr>
</tbody>
</table>

Whoops! We wanted Wed to include Tue, Mon and Fri, NOT the previous Thu (2) sales.

In the above example we see that no sales data is recorded for weekends; however, there were no sales for the government holiday MLK day (15-Jan-01). What we get with the above physical window is the last four days with sales recorded, not the last four weekdays. By their very nature, physical windows may not be well suited for sparse data (with gaps in dates etc.).

Time Interval Windows

Logical windows are defined by using the RANGE keyword and can only be ordered based on a single column or expression if either the PRECEDING or FOLLOWING keywords are used. For Logical Time Interval windows, the INTERVAL keyword is also used and the order by expression or column must evaluate to a date.

A Time Interval window includes all rows for the current rows day, month, or year as well as all rows whose interval value (sort expression) falls between the offset specified in the ROWS PRECEDING and ROWS FOLLOWING clause. For example, if a time interval window was defined as RANGE INTERVAL ‘2’ MONTHS PRECEDING then each row in the partition with the current month or the two preceding months would be included in the calculation of the function. DAY, MONTH, and YEAR are valid interval keywords.

Again, let’s try to write a query that shows us a moving sum of the last four weekdays regardless of whether there were sales or not.
Logical Time Interval Window Example – Running Total of Last Four Calendar Days

```sql
SELECT TO_CHAR(sdate,'Dy, Mon DD') s_date, qty,
      SUM(qty) OVER (PARTITION BY clerk
                   ORDER BY sdate RANGE INTERVAL '3' DAY PRECEDING) AS caldy
FROM clerk_daily_sales
WHERE clerk IN ('FRED')
ORDER BY sdate;
```

<table>
<thead>
<tr>
<th>S_DATE</th>
<th>QTY</th>
<th>SALEDY</th>
<th>CALDY</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thu, Jan 11</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Fri, Jan 12</td>
<td>1</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>Tue, Jan 16</td>
<td>6</td>
<td>9</td>
<td>6</td>
</tr>
<tr>
<td>Wed, Jan 17</td>
<td>5</td>
<td>14</td>
<td>11</td>
</tr>
</tbody>
</table>

Whoops! Now we successfully avoid Thu sales but now we are missing Fri sales.

Unfortunately the above query gets us the last four calendar days, but fails to recognize weekend. In addition, the above `PARTITION BY` clause was not needed, but added just to remind you of the possibilities that exist.

To conquer this request, we will need to base the actual window size for each row to account for weekends. If the current row is a Monday, Tuesday, or Wednesday the offset should be 5 days to allow for skipping over Saturday and Sunday. Thursday and Friday rows should still have an offset of 3 days. Below is a correct solution to the problem.
Varying Window Size Example – Running Total of Last Four Weekdays

```sql
SQL> SELECT TO_CHAR(sdate,'Dy, Mon DD') s_date, qty,
  2       SUM(qty) OVER (  
  3           ORDER BY sdate RANGE  
  4           (CASE WHEN (TO_CHAR(sdate,'DY') IN ('THU','FRI')) THEN 3 ELSE 5 END)  
  5           PRECEDING) AS weekdy  
  6      FROM clerk_daily_sales  
  7      WHERE clerk IN ('FRED')  
  8      ORDER BY sdate;

S_DATE       QTY  SALEDY  CALDY  WEEKDY
-----------  ---- ------- ------- -------
Thu, Jan 11  2       2       2  2 ← WEEKDY = Thu(2)
Fri, Jan 12  1       3       3  3 ← WEEKDY = Fri(1)+Thu(2)
Tue, Jan 16  6       9       6  9 ← WEEKDY = Tue(6)+Mon()+Sat()+Fri(1)+Thu(2)
                    Tue Includes Mon, Fri and Thu sales!
Wed, Jan 17  5      14      11 12 ← WEEKDY = Wed(5)+Tue(6)+Mon()+Sat()+Fri(1)
                    Wed Includes Tue, Mon and Fri sales!
Thu, Jan 18  4      16      15 15 ← WEEKDY = Thu(4)+Wed(5)+Tue(6)+Mon()
Fri, Jan 19  3      18      18 18 ← WEEKDY = Fri(3)+Thu(4)+Wed(5)+Tue(6)
Mon, Jan 22  9      21      12 21
```

In the above example, I used the new CASE function instead of using a DECODE or user defined function. Both of those would also work as well. The CASE function is new to Oracle 8i and allows all comparison operators to be used plus the ability to use ANDs, ORs, etc. in a single test. In addition, since simple date arithmetic is done in days, a Time Interval window with an INTERVAL of DAYS is the same as value range window. The above example is really a value range window based on days.

**Value Range Windows**

Logical Value Range windows can only be ordered based on a single column or expression if the PRECEDING or FOLLOWING clause is used and must evaluate to a positive integer.

A Value Range window contains all rows that have the same value as the order by expression of the current row plus all rows whose value falls between the offset specified in the values PRECEDING and values FOLLOWING clause. For example, if the current rows order by expression evaluates to 20 and the window is defined as ‘RANGE BETWEEN 5 PRECEDING AND 5 FOLLOWING’, then it would include all rows whose order by expression evaluates between 15 and 25.

The query below finds for each clerk, the average sales for all clerks that are within five years of his/her age.

**Value Range Example**
New SQL OLAP Functions...

```sql
SQL> SELECT s.clerk, age, SUM(qty) qty,
2  AVG(SUM(qty)) OVER (ORDER BY age
3       RANGE BETWEEN 5 PRECEDING AND 5 FOLLOWING) AS avg
4  FROM clerk_daily_sales s, clerks c
5  WHERE s.clerk = c.clerk
6  GROUP BY s.clerk, age
7  ORDER BY c.age;

CLERK   AGE  QTY    AVG
------ ---- ---- -----
JEFF     30   32  34.00
ALEX     33   36  36.00
TIM      36   40  37.00
SCOTT    39   35  35.00
FRED     42   30  32.50

Remember that the default window (created when just an ORDER BY clause is used) is defined as a logical window with the definition of RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW. In the example below we see that RSUM, which just uses an ORDER BY clause is the same as RASUM which is defined using the default keywords. It is important to remember that with a logical window, all rows with the current order by value are included. Looking at the first two rows, both have the order by value of January 11th and therefore both rows will have the same running total of six.

By simply making the ordering unique (i.e. adding CLERK to the ORDER BY clause), the column RSUMU looks more like our typical running total. It is still a logical window, but since the ordering is unique, no two rows will have the same order by value(s). ROSUM shows that we can still get the typical running total by specifying a physical window (using the ROWS keyword), but the results will not be deterministic (or repeatable).

Example of Non Unique sorts with Logical and Physical Windows

```sql
SQL> SELECT TO_CHAR(sdate,'Dy, Mon DD') s_date, clerk, qty,
2  SUM(qty) OVER (ORDER BY sdate)    AS rsum,
3  SUM(qty) OVER (ORDER BY sdate RANGE
4       BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW)   AS rasum,
5  SUM(qty) OVER (ORDER BY sdate,clerk)    AS rsumu,
6  SUM(qty) OVER (ORDER BY sdate ROWS UNBOUNDED PRECEDING )   AS rosum
7  FROM clerk_daily_sales
8  WHERE clerk IN ('SCOTT','FRED')
9  ORDER BY sdate, clerk;

S_DATE      CLERK   QTY     RSUM  =  RASUM      RSUMU  = ROSUM
-----------  ------ ----    -----    ------     ------   ------
Thu, Jan 11 FRED      2        6  =      6          2  =     2
Thu, Jan 11 SCOTT     4        6  =      6          6  =     6
Fri, Jan 12 FRED      1       10  =     10         10  =    10
Fri, Jan 12 SCOTT     3       10  =     10         20  =    20
Tue, Jan 16 FRED      6       24  =     24         24  =    24
```
Lag/Lead Functions

Lag/Lead Functions allows access to values in preceding or following rows and provide for inter row calculations such as period-to-period changes without expensive self-join operations. Example queries that could be written include “Growth in earnings from same quarter last year”. Access to a different row’s values are simply made by specifying the offset from the current row, and a default value can also be used to specify for rows where the offset goes beyond the beginning or end of the query.

The example below shows a basic use of the LAG and LEAD functions to access a preceding or following row.

```
SQL> SELECT TO_CHAR(sdate,'Dy, Mon DD') s_date, qty,
    2 LAG(qty,1,0) OVER (ORDER BY sdate) AS last_day,
    3 LEAD(qty,1,0) OVER (ORDER BY sdate)  AS next_day
    4 FROM clerk_daily_sales
    5 WHERE clerk IN ('FRED')
    6 ORDER BY sdate;
```

<table>
<thead>
<tr>
<th>S_DATE</th>
<th>QTY</th>
<th>LAST_DAY</th>
<th>NEXT_DAY</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thu, Jan 11</td>
<td>2</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Fri, Jan 12</td>
<td>1</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>Tue, Jan 16</td>
<td>6</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>Wed, Jan 17</td>
<td>5</td>
<td>6</td>
<td>4</td>
</tr>
<tr>
<td>Thu, Jan 18</td>
<td>4</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>Fri, Jan 19</td>
<td>3</td>
<td>4</td>
<td>9</td>
</tr>
<tr>
<td>Mon, Jan 22</td>
<td>9</td>
<td>3</td>
<td>0</td>
</tr>
</tbody>
</table>

In order to calculate period-to-period changes with lag / lead functions, the data would have to be dense or with no gaps thereby the physical offsets would correspond to logical offsets such as periods of time. For example, if the query is to calculate growth in sales from Q1 in one year to the next, then each quarter must have data so that the preceding fourth row is always for the same quarter. In the query below the sales growth from two days ago is calculated.

```
Period-to-Period Example - Growth in Sales from Two Days Ago
```

**Simple Lag/Lead Example**
New SQL OLAP Functions…

```
SQL> SELECT TO_CHAR(sdate,'Dy, Mon DD') s_date, qty, 
2    (qty - LAG(qty,2) OVER (ORDER BY sdate)) / 
3    (LAG(qty,2) OVER (ORDER BY sdate)) * 100 growth 
4  FROM clerk_daily_sales 
5  WHERE clerk IN ('FRED') 
6  ORDER BY sdate;

S_DATE       QTY   GROWTH
-----------  ---- --------
Thu, Jan 11    2
Fri, Jan 12    1
Tue, Jan 16    6   200.00  GROWTH = (6 – 2) / 2
Wed, Jan 17    5   400.00  GROWTH = (5 – 1) / 1
Thu, Jan 18    4   -33.33
Fri, Jan 19    3   -40.00
Mon, Jan 22    9   125.00
```

**First and Last Functions**

These two functions `FIRST_VALUE` and `LAST_VALUE` are really just two more Window Aggregate Functions. They return the first value in a window and the last value in a window.

Don’t forget that unless `UNBOUNDED FOLLOWING` is specified, the `LAST_VALUE` function will return the value of the current row since it operates over the window and by default a window ends with the current row as shown by the `RLQTY` column in the example below. `FIRST_VALUE` will be as expected, since by default a window is defined to start on the first row of the partition (or query, i.e. `UNBOUNDED PRECEDING`).

**First and Last Value Example**

```
SQL> SELECT TO_CHAR(sdate,'Dy, Mon DD') s_date, qty, 
2    FIRST_VALUE(qty) OVER (ORDER BY sdate) AS fqty, 
3    LAST_VALUE(qty) OVER (ORDER BY sdate) AS rlqty, 
4    LAST_VALUE(qty) OVER (ORDER BY sdate 
5    ROWS BETWEEN UNBOUNDED PRECEDING 
6    AND UNBOUNDED FOLLOWING) AS lqty 
7  FROM clerk_daily_sales 
8  WHERE clerk = 'FRED';

S_DATE       QTY FQTY  RLQTY    LQTY
-----------  ---- ----- ------   ----- 
Thu, Jan 11    2     2      2      9
Fri, Jan 12    1     2      1      9
Tue, Jan 16    6     2      6      9
Wed, Jan 17    5     2      5      9
Thu, Jan 18    4     2      4      9
Fri, Jan 19    3     2      3      9
Mon, Jan 22    9     2      9      9
```

**Conclusion**
If you made it to the end of this paper, I hope you have a good overview of some of the new OLAP functions provided for by Oracle 8i. With these new functions all kinds of complex queries could be formulated. In the example below the query ranks and returns by sales only the clerks who sold less than the average of the clerks who are within five years of his age.

```
SQL> SELECT clerk, qty,
          2  RANK() OVER (ORDER BY qty DESC) rank
          3  FROM (SELECT s.clerk, SUM(qty) qty,
          4  AVG(SUM(qty)) OVER (ORDER BY age
          5  RANGE BETWEEN 5 PRECEDING AND 5 FOLLOWING) AS avg
          6  FROM clerk_daily_sales s, clerks c
          7  WHERE s.clerk = c.clerk
          8  GROUP BY s.clerk, age) clerk_avg
          9  WHERE qty < avg
         10  ORDER BY rank;

  CLERK  QTY  RANK
  ------ ---- -----
   JEFF   32     1
   FRED   30     2
```

In addition, Oracle has also added some statistical functions including linear regression, covariance, and correlation computations. Please refer to the Oracle documentation for complete details.