ORACLE'S SQL ANALYTIC FUNCTIONS IN 8i AND 9i

David L. Fuston, dfuston@vlamis.com
Vlamis Software Solutions, Inc., www.vlamis.com

INTRODUCTION

The SQL language has traditionally provided little support for business intelligence processing. Many aggregation tasks and business intelligence calculations such as subtotals, moving averages, rankings and lead/lag comparisons were cumbersome at best prior to Oracle 8i.

ANSWERING BUSINESS QUESTIONS

The challenge is in deriving answers to business questions from the available data, so that decision makers at all levels can quickly respond to changes in the business climate. While a standard transactional query might ask, "When did order 84305 ship?" an analytical query might ask, "How do sales in the Southwestern region for this month compare with plan? Or with sales a year ago?" The first question involves simple data selection and retrieval. However, the second question involves inter-row calculations, time series analysis, and access to aggregated historical and current data. This is online analytical processing — OLAP. The data processing required to answer analytical questions is fundamentally different from the data processing required to answer transactional questions. The following table highlights the major differences.

Characteristic	Transactional Query	Analytical Query
Typical operation	Update	Analyze
Age of data	Current	Historical
Level of data	Detail	Aggregate
Data required per query	Minimal	Extensive
Querying pattern	Individual queries	Iterative queries

TRANSFORMING TABLES INTO MULTIDIMENSIONAL DATA STRUCTURES

DIFFERENCES IN DATA MODELS

When an OLAP application runs, OLAP Services fetches the required data from an Oracle*9i* database into a temporary cache. Within this cache, the data is stored in multidimensional data objects. For the data to be fetched

correctly, you must identify which columns will be fetched and what role they will play. The basic data model in a relational database is a table composed of one or more columns of data. All of the data is stored in columns. In contrast, the basic data model in a multidimensional cache is a cube, which is composed of measures, dimensions, and attributes. Once you determine that you will want to access a particular column through OLAP Services, you must identify whether the data from that column will function in it as a measure, a dimension, or an attribute. You also

identify which columns are keys. These decisions are stored as metadata and constraints.

TYPES OF DATA STRUCTURES

Your Oracle RDBMS and OLAP Services use different data structures, as shown in the following table. Note that even though both use dimensions, their implementations are different.

Oracle RDBMS & Data Warehouse	**OLAP Services**
Tables	Levels
Materialized Views	Attributes
Dimensions	Dimensions
	Measures
	Cubes

IDENTIFYING YOUR DATA REQUIREMENTS

Before you can begin mapping columns to multidimensional structures, you must know what data users want to view and at what levels they want to view it. If you have already created a data warehouse, you have already done most of this research. You only need to verify that the requirements haven't changed for the analytical applications that will be run using OLAP Services. You can use Oracle Enterprise Manager to explore the existing schemas. Then make a note of the columns that you are going to use and the types of multidimensional objects you want to define them as: measures, dimensions, or attributes. Keep in mind that the OLAP API only has access to objects in the database through the metadata definitions. Thus, if an object (such as a column in a table) has not been defined in the metadata, it is not available to OLAP applications.

MEASURES

Measures are the same as facts. The term "fact" is typically used in relational databases, and the term "measure" is typically used in multidimensional applications. You will encounter both terms in Oracle Enterprise Manager, since the creation of metadata is the process of associating relational objects with their multidimensional counterparts.

Measures are thus located in fact tables. A fact table typically has two types of columns: measures (or facts) and foreign keys to dimension tables. Measures contain the data that you wish to analyze, such as Sales or Cost. Oracle

Enterprise Manager requires that a column have a numerical or date data type to be identified as a measure. Most frequently, a measure is numerical and additive. One or more columns in the dimension tables form constraints on the fact tables. Foreign keys in the fact tables define these constraints, by the metadata, or both.

DIMENSIONS

Dimensions identify and categorize your data. Dimension members are stored in a dimension table. Each column represents a particular level in a hierarchy. In a star schema, the columns are all in the same table; in a snowflake schema, the columns are in separate tables for each level. Because measures are typically multidimensional, a single value in a measure must be qualified by a member of each dimension to be meaningful. For example, Sales measure might have dimensions for Product, Geographic Area, and Time. A value in the Sales measure (37854) is only meaningful when it is qualified by a product (DVD Player), a geographic area (Southwest), and Time (March 2001). Defining a dimension in your data warehouse creates a database dimension object, in addition to creating metadata. A dimension object contains the details of the parent-child relationship between columns in a dimension table; it does not contain data. The database dimension object is used by the Summary Advisor and query rewrite to optimize your data warehouse. However, in the OLAP API, a dimension does contain data, such as the names of individual products, geographic areas, and time periods. The OLAP API uses the metadata, dimension objects, and dimension tables to construct its dimensions.

LEVELS

Dimensions are structured hierarchically so that data at different levels of aggregation can be manipulated together efficiently for analysis and display. Each dimension must have at least one level. Each *level* represents a position in the hierarchy. Levels group the data for aggregation and are used internally for computation. For example, in a Time dimension hierarchy, you might group weeks into quarters into years. Week, Quarter, and Year are the levels of your Time dimension. If data for the Sales measure is stored in weeks, then the higher levels of the Time dimension allow the Sales data to be aggregated correctly into quarters and years. The members of a hierarchy at different levels have a *parent-child* relationship. For example, “QTR1” is the child of “YR2001,” thus “YR2001” is the parent of “QTR1.” The dimension members at the lowest level of a hierarchy often are used as a foreign key in a fact table. All levels of a dimension are stored in dimension tables.

ATTRIBUTES

Attributes provide supplementary information about the dimension members at a particular level. Attributes are often used for display, since the dimension members themselves may be meaningless, such as a value of “T2965” for a time period. For example, you might have columns for employee number (ENUM), last name (LAST_NAME), first name (FIRST_NAME), and telephone extension (TELNO). ENUM is the best choice for a level, since its values uniquely identify employees while the other columns may not. ENUM also has a NUMBER data type, which makes it more efficient than a text column. LAST_NAME, FIRST_NAME, and TELNO are attributes. Even though they are dimensioned by ENUM, they do not make suitable measures because they are descriptive text rather than business measurements. Attributes are associated with a particular level of a dimension hierarchy and must be stored in the same table as that level.

RECENT CHANGES IN SQL

The recent releases of Oracle have addressed both the differences in data models and types of data structures by significantly enhancing SQL for business intelligence processing:

- Oracle8*i* Release 1 added support for the CUBE and ROLLUP extensions to the SELECT statement's GROUP BY clause. These extensions enable more efficient and convenient aggregations, a key part of data warehousing and business intelligence processing.
- Oracle8*i* Release 2 introduced a powerful new set of SQL analytic functions to address essential business intelligence calculations. The analytic functions provide enhanced performance and higher developer productivity for many calculations. In addition, the functions are being considered for incorporation into the ANSI SQL: 1999 standard.
- Oracle8*i* Release 8.2 adds powerful new families of analytic functions and important extensions for the GROUP BY clause.

ANALYTIC FUNCTIONS

Oracle provides 8 families of analytic functions, 3 of which are new in Release 8.2. Here is a brief description of each family listing the specific functions:

- Ranking family - This family supports business questions like "show the top 10 and bottom 10 salesperson per region" or "show, for each region, salespersons that make 25% of the sales". Oracle provides RANK, DENSE_RANK, PERCENT_RANK, CUME_DIST and NTILE functions.
- Window Aggregate family - This family addresses questions like "What is the 13-week moving average of a stock price?" or "What was the cumulative sum of sales per each region?" The new features provide moving and cumulative processing for all the SQL aggregate functions including AVG, SUM, MIN, MAX, COUNT, VARIANCE and STDDEV.
- Reporting Aggregate family - One of the most common types of calculations is the comparison of values at different levels of aggregation. For instance, we might want to know regional sales levels as a percent of national sales. All percent-of-total and market share calculations require this processing. The reporting aggregate family makes these sort of calculations simple: it lets one row contain values calculated at different aggregation levels. The new family provides reporting aggregate processing for all SQL aggregate functions including AVG, SUM, MIN, MAX, COUNT, VARIANCE and STDDEV.
- LAG/LEAD family - Studying change and variation is at the heart of analysis. Necessarily, this involves comparing the values of different rows in a table. While this has been possible in SQL, usually through self-joins, it has not been efficient or easy to formulate. The LAG/LEAD family enables queries to compare different rows of a table simply by specifying an offset from the current row.
- Linear Regression family - Oracle provides functions for linear regression calculations, including slope, intercept, correlation coefficient and many other key values.
- Inverse Percentile family - Added in Release 8.2, these functions allow queries to find the data, which corresponds to a specified percentile value. For instance, users may find the median value of a data set by querying PERCENTILE_DISC(0.5).

- Hypothetical Rank and Distribution family - These functions, new in Release 8.2, allow queries to find what rank or percentile value a hypothetical data value would have if it were added to an existing data set.
- FIRST/LAST Aggregates family - A Release 8.2 enhancement, this family enables queries to return the first or last value of a sorted aggregate group.

Note that the analytic functions are intended to supplement the power of the relational database platform for decision support processing: they are not intended to supplant the role of specialized OLAP environments. Any OLAP product, such as Oracle Express, can leverage the power of the analytic functions to enhance its query performance.

BENEFITS

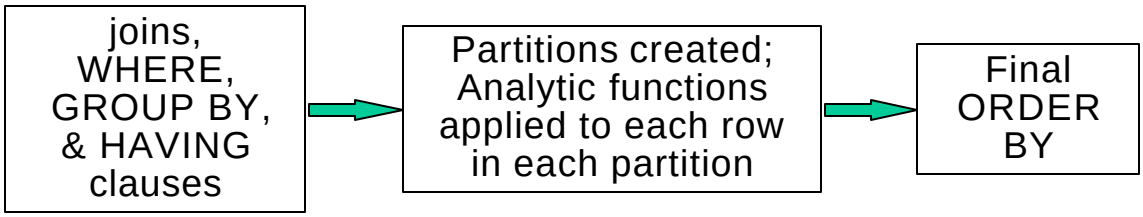
The analytic functions enhance both database performance and developer productivity. They are valuable for all types of processing, ranging from interactive decision support to batch report jobs. Corporate developers and independent software vendors alike will be able to take advantage of the features. Here are key benefits provided by the new functions:

- *Improved Query Speed* - The processing optimizations supported by these functions enable significantly better query performance. An action which before required self-joins or complex procedural processing may now be performed with far fewer table scans in native SQL. The performance enhancements enabled by the new functions enhance query speeds for Oracle's Express system and other ROLAP products.
- *Enhanced Developer Productivity* - The functions enable developers to perform complex analyses with much clearer and more concise SQL code. Tasks, which in the past required multiple SQL statements or the use of Express procedural languages, can now be expressed using single SQL statements. The new SQL is quicker to formulate and maintain than the older approaches, resulting in greater productivity.
- *Minimized Learning Effort* - Through careful syntax design, the analytic functions minimize the need to learn new keywords. The syntax leverages existing aggregate functions, such as SUM and AVG, so that these well-understood keywords can be used in extended ways.
- *Standardized Syntax* - As part of the ANSI SQL standard, the new functions are attractive for independent software vendors: vendors will have an incentive to adjust their products to take advantage of the new functions.

ANALYTIC FUNCTION CONCEPTS

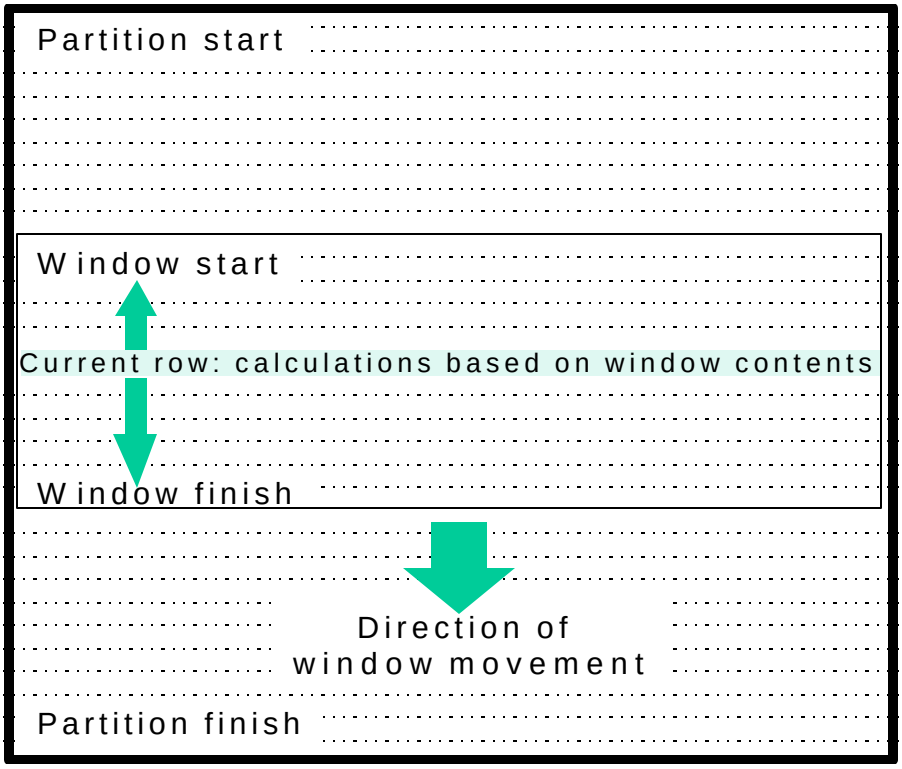
To perform their operations, the analytic functions add several new elements to SQL processing. These elements build on existing SQL to allow flexible and powerful calculation expressions. Here are the essential concepts used in the analytic functions:

- *Processing Order* - Query processing using analytic functions takes place in three stages. All joins, WHERE, GROUP BY and HAVING clauses are performed first. Second, the result set is made available to the analytic functions, and all their calculations take place. Third, if the query has an ORDER BY clause at its end, the ORDER BY is processed to allow for precise output ordering. The processing order is shown below.



- *Result Set Partitions* - The analytic functions allow users to divide query result sets into ordered groups of rows called partitions. Note that the term "*partitions*" used with analytic functions is unrelated to Oracle's table partitions feature. Throughout this paper, we use "partitions" only in the meaning related to analytic functions. Partitions are created after the groups defined with GROUP BY clauses, so any aggregate results such as SUM's and AVG's are available to them. Partition divisions may be based upon any desired columns or expression. A query result set may have just one partition holding all the rows, a few large partitions, or many small partitions holding just a few rows each.
- *Window* - For each partition, a sliding window of data may be defined. The window determines the range of rows used to perform the calculations for the "current row" (defined in the next bullet). Window sizes can be based on either a physical number of rows or a logical interval such as time. The window has a starting row and an ending row. Depending on its definition, the window may move at one or both ends. For instance, a window defined for a cumulative sum function would have its starting row fixed at the first row of its partition, and its ending row would slide from the starting point all the way to the last row of the partition. In contrast, a window defined for a moving average would have both its starting and end points slide so that they maintained a constant physical or logical range.

  A window can be set as large as all the rows in a partition. At the other extreme, it could be just a single row. Users may specify a window containing a constant number of rows, or a window containing all rows where a column value is in a specified numeric range. Windows may also be defined to hold all rows where a date value falls within a certain time period, such as the prior month.
- *Current Row* - Each calculation performed with an analytic function is based on a current row within a window. The current row serves as the reference point determining the start and end of the window. For instance, a centered moving average calculation could be defined with a window that holds the current row, the 5 preceding rows and the 6 rows ahead of it. This would create a sliding window of 12 rows, as shown below.



RELEASE 8.2 ANALYTIC FUNCTIONS - FEATURES AND EXAMPLES

This section describes the key features of the analytic functions introduced in Release 8.2 and provides basic examples. However, it does offer practical cases to show the value of the new functions.

In the query below I used three queries (union-ed together). The first query provides the department-job totals; the second, the department totals; and the third, the grand total.

```
SQL> -- Here is the pure SQL way with accesses to the employee table
SQL>
SQL> SELECT deptno, job, SUM(sal) sal ← Provides the dept-job totals
  2  FROM emp
  3  GROUP BY deptno, job
  4  UNION
  5  SELECT deptno, NULL job, SUM(sal) sal ← Provides the dept sub totals
  6  FROM emp
  7  GROUP BY deptno
  8  UNION
  9  SELECT TO_NUMBER(NULL) deptno, NULL job, SUM(sal) sal ← The grand total
 10  FROM emp
 11  ORDER BY deptno, job;

     DEPTNO JOB              SAL
----------- --------- ---------
         10 CLERK          1300
         10 MANAGER        2450
         10 PRESIDENT      5000
         10                8750
         20 ANALYST        6000
         20 CLERK          1900
         20 MANAGER        2975
         20               10875
         30 CLERK           950
         30 MANAGER        2850
         30 SALESMAN       5600
         30                9400
                          29025

13 rows selected.
```

Now in 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 were created by the two extra UNION SELECT statement in the above traditional SQL solution; however, the new ROLLUP operation can create these rows with only one table access versus the above example, which accessed the table three separate times.

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 job within a department, a subtotal for each department, and a grand total for all departments. Notice how simple the syntax is using the ROLLUP operation.

```
SQL>
SQL> -- Here is the new way using ROLLUP and a single table access
SQL>
SQL> SELECT deptno, job, SUM(sal) sal
  2  FROM emp
  3  GROUP BY ROLLUP(deptno, job);

    DEPTNO JOB              SAL
---------- --------- ---------
        10 CLERK          1300
        10 MANAGER        2450
        10 PRESIDENT      5000
        10                8750
        20 ANALYST        6000
        20 CLERK          1900
        20 MANAGER        2975
        20               10875
        30 CLERK           950
        30 MANAGER        2850
        30 SALESMAN       5600
        30                9400
                         29025

13 rows selected.
```

Although ROLLUPs can be achieved using client side tools such as SQL*Plus that calculate these subtotals on the client, these tools can place a significant load on the client machine.

But what happens when one of the columns that you are aggregating on allows a `NULL` value? The question then becomes "is the `NULL` column row returned a subtotal row or a normal group row?" Consider the following query output.

```
SQL> -- But what if someone isn't assigned a JOB or Dept?  Confusing NULLS!
SQL>
SQL> SELECT deptno, job, SUM(sal) sal
  2  FROM emp
  3  GROUP BY ROLLUP(deptno, job);

    DEPTNO JOB              SAL
---------- --------- ---------
        10 CLERK          1300
        10 MANAGER        2450
        10                3750
        20 ANALYST        6000
        20 CLERK          1900
        20 MANAGER        2975
        20               10875
        30 CLERK           950
        30 MANAGER        2850
        30 SALESMAN       4350
        30                1250   ← Is this a subtotal?
        30                9400   ← What about this row?
           PRESIDENT      5000
                          5000
                         29025

15 rows selected.
```

In order to help our programs and us 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 operator and returns a “0” if it is a normal row returned by the query.

Consider the example below using the GROUPING function. Notice that I used the traditional DECODE syntax to determine the job subtotals, and the new Oracle 8i CASE syntax to determine the department subtotals. While the DECODE statement uses only repetitive equality checks, the CASE statement allows all comparison operators plus the ability to use ANDs, ORs, etc. in a single test. Anyway, on with our example of ROLLUPs.

```
SQL>
SQL> -- Use GROUPING Function.      Example uses new 8i CASE function
SQL> -- and traditional old DECODE function
SQL>
SQL> SELECT
  2        CASE WHEN    GROUPING(deptno) = 1
  3             THEN 'All Depts'
  4           WHEN (GROUPING(deptno) = 0 AND deptno IS NULL )
  5             THEN 'No Dept'
  6           ELSE TO_CHAR(deptno) END AS deptno,
  7        DECODE(GROUPING(job),
  8             1, 'All Jobs',
  9             0, NVL(job,'No Job'))  AS job,
 10        SUM(sal) sal
 11  FROM
 12        emp
 13  GROUP BY
 14        ROLLUP(deptno,job);

DEPTNO     JOB             SAL
---------- --------- ---------
10         CLERK          1300
10         MANAGER        2450
10         All Jobs       3750
20         ANALYST        6000
20         CLERK          1900
20         MANAGER        2975
20         All Jobs      10875
30         CLERK           950
30         MANAGER        2850
30         SALESMAN       4350
30         No Job         1250
30         All Jobs       9400
No Dept    PRESIDENT      5000
No Dept    All Jobs       5000
All Depts  All Jobs      29025

15 rows selected.
```

But what if you wanted to get subtotals not only for each department and job within department, but also for each job (across departments)? 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. week, month, and year). Note, however, that `ROLLUP`s and CUBEs are independent of any hierarchy meta-data stored in dictionary.

Subtotals created by `CUBE` would be synonymous with those created for a cross-tab or matrix type report. Here is the example using `CUBE`. I'll leave the traditional SQL solution for you to do.

```
SQL>
SQL> -- Example using CUBE
SQL>
SQL> SELECT deptno, job, SUM(sal) sal
  2  FROM emp
  3  GROUP BY CUBE(deptno, job);

    DEPTNO JOB              SAL
---------- --------- ---------
        10 CLERK          1300
        10 MANAGER        2450
        10 PRESIDENT      5000
        10                8750
        20 ANALYST        6000
        20 CLERK          1900
        20 MANAGER        2975
        20               10875
        30 CLERK           950
        30 MANAGER        2850
        30 SALESMAN       5600
        30                9400
           ANALYST        6000
           CLERK          4150
           MANAGER        8275
           PRESIDENT      5000
           SALESMAN       5600
                         29025

18 rows selected.
```

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 subtotals, or to include only the subtotal rows. In addition, since the `ORDER BY` is the last operation performed, the subtotal and grand total lines are sorted among the rest of the rows returned by the query.

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

```
SQL>
SQL> -- And if we only wanted the subtotal and grand total lines
SQL>
SQL> SELECT
  2        DECODE(GROUPING(deptno),1,'All Depts',deptno) deptno,
  3        DECODE(GROUPING(job),1,'All Jobs',job) job,
  4        SUM(sal) sal
  5  FROM
  6        emp
  7  GROUP BY
  8        CUBE(deptno, job)
  9  HAVING
 10        GROUPING(deptno) = 1 OR GROUPING(job) = 1;

DEPTNO     JOB              SAL
---------- --------- ---------
10         All Jobs        8750
20         All Jobs       10875
30         All Jobs        9400
All Depts  ANALYST         6000
All Depts  CLERK           4150
All Depts  MANAGER         8275
All Depts  PRESIDENT       5000
All Depts  SALESMAN        5600
All Depts  All Jobs       29025

9 rows selected.
```

In all of our examples 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.

Analytic Functions

Just beyond `ROLLUP` and `CUBE` are some new functions for determining how a given row ranks or compares to other rows in the set. Continuing on, suppose that our favorite user wants a report that shows a list of employee names, their salary, and a ranking based on their salary. Oracle 8i provides the `RANK` function to calculate such information. The syntax looks a little strange, but it's easy to understand. Before we begin, we need to discuss how `NULL`s are ranked within Oracle. Oracle treats `NULL`s as the largest value by default. The new ranking functions, however, give us the choice of placing `NULL`s at either the top or the bottom of the list.

Here is the example using `RANK` twice; once with `NULL`s at the top and once with `NULL`s at the bottom. Again, I will leave the pure SQL solution for you to do (no beginner task, mind you).

```
SQL>
SQL> -- Rank example
SQL>
SQL> SELECT ename, sal,
  2   RANK() OVER (ORDER BY sal DESC NULLS LAST) AS null_rank_last,
  3   RANK() OVER (ORDER BY sal DESC NULLS FIRST) AS null_rank_first
  4  FROM emp;

ENAME            SAL NULL_RANK_LAST NULL_RANK_FIRST
---------- --------- -------------- ---------------
KING                            14               1
JONES           3000             1               2
SCOTT           3000             1               2
FORD            3000             1               2
BLAKE           2850             4               5
CLARK           2450             5               6
ALLEN           1600             6               7
TURNER          1500             7               8
MILLER          1300             8               9
WARD            1250             9              10
MARTIN          1250             9              10
ADAMS           1100            11              12
JAMES            950            12              13
SMITH            800            13              14

14 rows selected.
```

Investigating the last column “null rank first”, you notice that three people are tied for second and the next person has the place of fifth. If you wanted to not skip any numbers after a tie, use the `DENSE_RANK` function (example below).

Now suppose that we wanted to also provide a ranking of salaries within each department. To accomplish this task use the `PARTITION BY` clause followed by a list of columns (separated by commas). The `PARTITION BY` clause is totally unrelated and independent of Oracle 8’s table partition feature, etc. The `PARTITION BY` clause breaks the data into numerous datasets for analytic functions to perform its’ calculations.

```
SQL>
SQL> -- Rank within department example (No skipping numbers i.e. DENSE)
SQL>
SQL> SELECT deptno, ename, sal,
  2   DENSE_RANK() OVER (ORDER BY sal DESC) AS overall_rank,
  3   DENSE_RANK() OVER (PARTITION BY deptno
  4              ORDER BY sal DESC) AS dept_rank
  5  FROM emp
  6  ORDER BY deptno, sal DESC;

    DEPTNO ENAME            SAL OVERALL_RANK DEPT_RANK
---------- ---------- --------- ------------ ---------
        10 KING            5000            1         1
        10 CLARK           2450            4         2
        10 MILLER          1300            7         3
```

```
         20 JONES           3000            2         1
         20 FORD            3000            2         1
         20 SCOTT           3000            2         1
         20 ADAMS           1100            9         2
         20 SMITH            800           11         3
         30 BLAKE           2850            3         1
         30 ALLEN           1600            5         2
         30 TURNER          1500            6         3
         30 WARD            1250            8         4
         30 MARTIN          1250            8         4
         30 JAMES            950           10         5

14 rows selected.
```

Suppose further that we want to calculate the percent of the departments' total salary each person receives. This calculation is performed by the `RATIO_TO_REPORT` function. Using the `PARTITION BY` clause causes the `RATIO_TO_REPORT` function to provide the ratio of the partition instead of the whole report.

```
SQL>
SQL> -- Want to see what percentage of payroll each person is in their dept?
SQL>
SQL> SELECT
  2        deptno, ename, sal,
  3        RATIO_TO_REPORT(sal)
  4              OVER (PARTITION BY deptno) AS pct_of_dept
  5  FROM
  6        emp
  7  ORDER BY
  8        deptno, sal DESC;

    DEPTNO ENAME            SAL PCT_OF_DEPT
---------- ---------- --------- -----------
         10 KING           5000         .57
         10 CLARK          2450         .28
         10 MILLER         1300         .15
         20 JONES          3000         .28
         20 SCOTT          3000         .28
         20 FORD           3000         .28
         20 ADAMS          1100         .10
         20 SMITH           800         .07
         30 BLAKE          2850         .30
         30 ALLEN          1600         .17
         30 TURNER         1500         .16
         30 WARD           1250         .13
         30 MARTIN         1250         .13
         30 JAMES           950         .10

14 rows selected.
```

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. You can also use a window to define a range (or moving range) of rows that are used to perform the analytic function calculation. With windows you can, therefore, calculate running totals, moving averages, etc.

IN-LINE VIEWS AND 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.

```
SQL>
SQL> -- Top-N example - get top two salary earners; notice ties are split
SQL>
SQL> SELECT *
  2  FROM (SELECT ename, sal
  3        FROM emp
  4        ORDER BY sal DESC) emp
  5  WHERE rownum < 3;

ENAME            SAL
---------- ---------
KING            5000
JONES           3000
```

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 three rows are desired so it holds only the three 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.

```
SQL>
SQL> -- And to get the top 2 salary earners in each group - ties not split
SQL>
SQL> SELECT deptno, ename, sal
  2  FROM (SELECT deptno, ename, sal,
  3        RANK() OVER (PARTITION BY deptno
  4              ORDER BY sal DESC) AS dept_rank
  5        FROM emp) emp
  6  WHERE dept_rank < 3;

    DEPTNO ENAME            SAL
---------- ---------- ---------
        10 KING            5000
        10 CLARK           2450
        20 JONES           3000
        20 SCOTT           3000
        20 FORD            3000
        30 BLAKE           2850
        30 ALLEN           1600

7 rows selected.
```

CONCLUSION

With the introduction of analytic functions, Oracle solved many problems of using SQL in business intelligence tasks. The added analytic functions of Oracle8*i* Release 8.2 enable faster query performance and greater developer productivity for even more calculations. The value of analytic functions has already been recognized, and major business intelligence tool vendors are using them in their products. The power of the analytic functions, combined with their status as international SQL standards, make them an important tool for all SQL users.

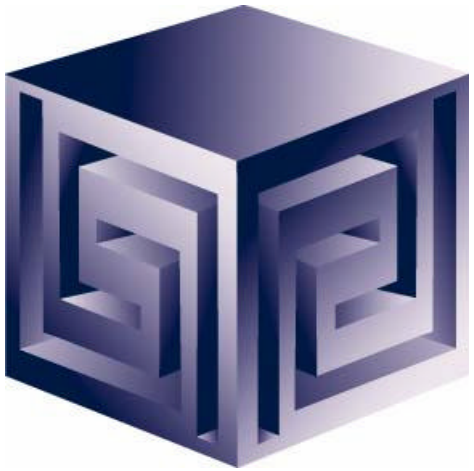
Footnote: As of the writing of this paper, Oracle 9i beta was still in development and not yet released to a select group of beta testers and developers. By the time of the IOUG-A Live! 2001 conference, 9i analytic functions will be summarized in the MS PowerPoint presentation.

Footnote: For information on the analytic functions introduced in earlier release, Oracle8*i* Release 2, please see the technical white paper "Analytic Functions for Oracle8*i*," available at the URL http://technet.oracle.com/products/oracle8i/ on the Oracle Technology Network.

Oracle's SQL Analytic Functions in 8i and 9i

Presentation #278

IOUG-A Live! 2001



Presented by:

David Fuston

Vlamis Software Solutions, Inc.
(816) 781-2880

dfuston@vlamis.com

http://www.vlamis.com

Vlamis Software Solutions, Inc.

- **Founded in 1992 in Kansas City, Missouri**
- **A Member of Oracle Partner Program since 1995 along with various Oracle Beta Programs, including 9i**
- **Designs and implements databases/data marts/data warehouses using RDBMS and Multidimensional tools**
- **Specializes in Data Transformation, Data Warehousing, Business Intelligence, Oracle Financials and Applications Development**
- **Founder Dan Vlamis is former developer at Oracle-Waltham office for Sales Analyzer Application**
- **Oracle Certified Solutions Provider**

ORACLE | CERTIFIED SOLUTION PARTNER

Agenda

- **Answering Business Questions**
- **Transforming Tables into Multidimensional Data Structures**
- **Recent Changes in SQL**
- **Analytic Functions and Families**
- **Analytic Functions - Features and Examples**
- **New in 9i**
- **Summary**

Answering Business Questions

- **Standard transactional query might ask, “When did order 84305 ship?”**
- **The transactional query involves simple data selection and retrieval.**

- **An analytical query might ask, “How do sales in the Southwestern region for this month compare with plan? Or with sales a year ago?”**
- **The analytical query involves inter-row calculations, time series analysis, and access to aggregated historical and current data.**

Transactional Query versus Analytical Query

Characteristic	Transactional Query	Analytical Query
Typical Operation	Update	Analyze
Age of Data	Current	Historical, current & projected
Level of Data	Detail	Aggregate
Data Required per Query	Minimal	Extensive
Querying Pattern	Individual Queries	Iterative Queries

Types of Data Structures

Oracle RDBMS and Data Warehouse	Advanced Analytic Services
Tables	Levels
Materialized Views	Attributes
Dimensions	Dimensions
	Measures
	Cubes

Metadata Definitions and Objects

- **Measures (Multi-dimensional) = Facts (relational)**
  - **Sales units or dollars, unit cost**
- **Dimensions = identify and categorize data**
  - **Product, Geography, Time, Sales Channel**
- **Level = position in hierarchy of a dimension**
  - **Week rolls into Quarter, which rolls into Year**
- **Attributes = supplementary info about a dimension member**
  - **Color, telephone number, size, shape, weight**

Analytic Function Families

- **Ranking family - RANK, DENSE_RANK, PERCENT_RANK, CUME_DIST, ROW_NUMBER, and NTILE functions.**
- **Moving Aggregate family - SUM, AVG, MIN, MAX, COUNT, VARIANCE, STDDEV, FIRST_VALUE, LAST_VALUE**
- **Reporting Aggregate family – SUM, AVG, MIN, MAX, COUNT (with/without DISTINCT), VARIANCE, STDDEV, RATIO_TO_REPORT**
- **LAG/LEAD family**
- **Linear Regression family - slope, intercept, correlation coefficient**
- **Inverse Percentile family - PERCENTILE_DISC**

Analytic Function Families—Sample Questions

- **Ranking (“Find the top 10 sales reps in each region.”)**
- **Moving aggregates (“What is the 200-day moving average of our company’s stock price?”)**
- **Period-over-period comparisons (“What is the percentage growth of January 1999 over January 1998?”)**
- **Ratio-to-report (“What are January’s sales as a percentage of the entire year’s?”)**

Analytic Function Benefits

- **Analytic functions are not intended to replace OLAP environments; rather, they may be used by OLAP products like Oracle's Express to:**
  - **Improved Query Speed**
  - **Enhanced Developer Productivity**
  - **Minimized Learning Effort**
  - **Standardized Syntax**

Analytic Function Benefits

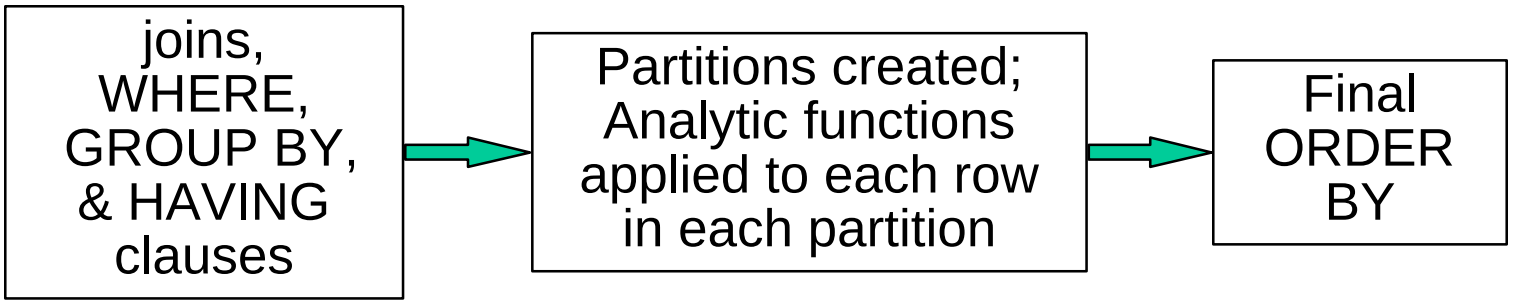
- **Analytic functions lend statistical muscle that has in the past called for joins, unions, and complex programming.**
- **Performance is improved (sometimes significantly) because the functions are performing work that previously required self-joins and unions.**
- **Using Analytic functions requires far less SQL coding than previously required to accomplish the same task because one SQL statement takes the place of many.**
- **Analytic functions allow division of results into ordered groups**

Analytic SQL Processing Concepts

- **Processing Order – 3 stages**
- **Result Set Partitions -- unrelated to Oracle's table partitions feature**
- **Window -- For each partition**
- **Current Row**

The Current Row is inside a Window, a Window is inside a Partition, and a Partition is inside of the Result Set.

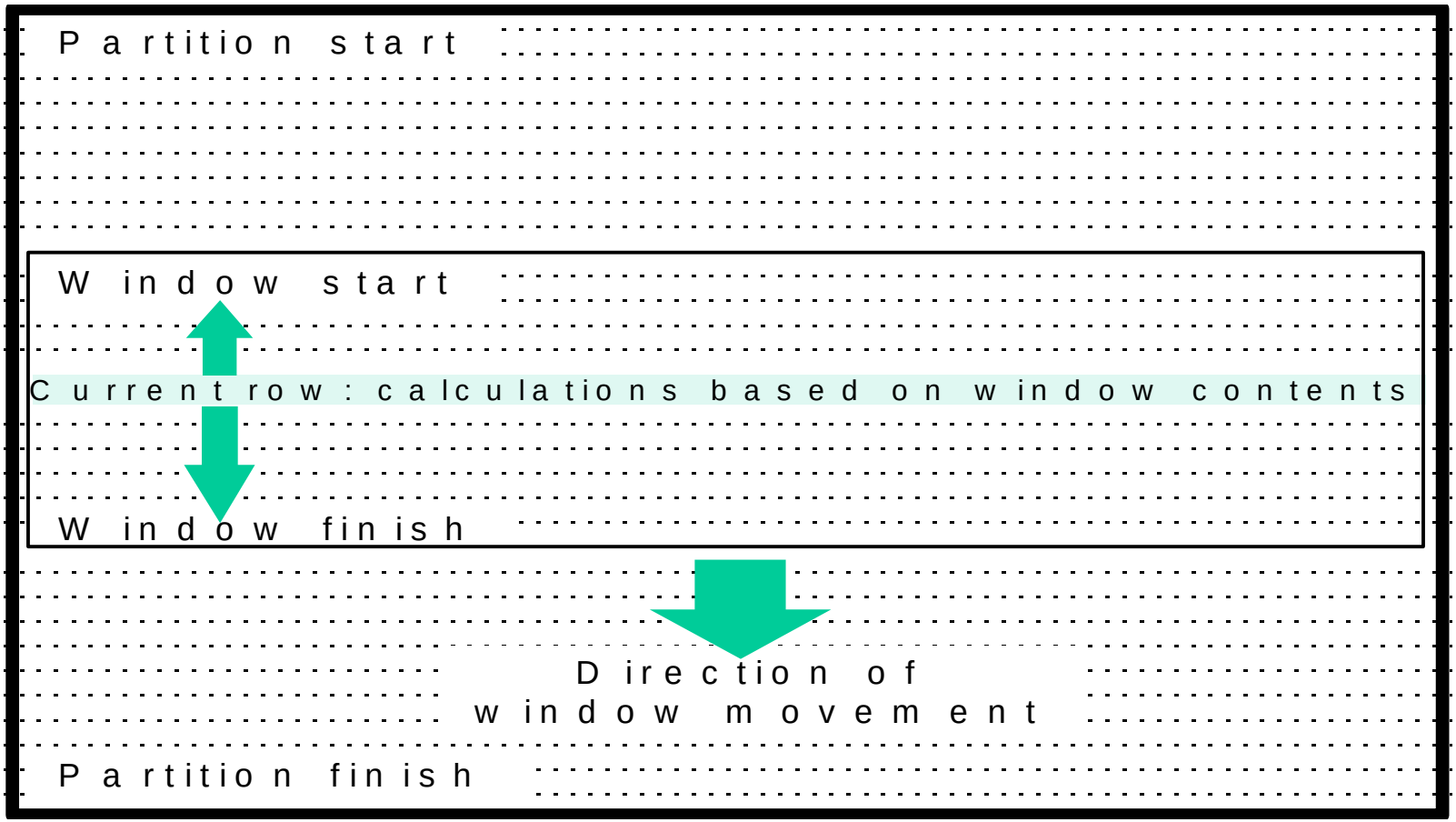
Processing Order – 3 stages



Analytic Function Query Partitions

- **Query result sets are divided into ordered groups called Partitions***
- **Partitioning takes place after GROUP BY.**
- **Result sets may be divided into as many partitions as makes sense for the values being derived.**
- **Partitioning may be performed using expressions or column values.**
- **Each result set may represent a single Partition, a few larger Partitions, or many small Partitions.**
- **Each Partition may be represented by a sliding Window defining the range of rows used for calculations on the Current Row.**
- ***unrelated to database table partitioning**

Window and Current Row



Analytic Function Windows

- **Windows may be defined representing a number of physical rows**
- **Each Window has a starting row and an ending row and may slide either direction**
- **A moving average would slide both directions so that the averaging made sense.**
- **Windows may represent 1 or more rows in a partition (or the entire partition).**

Current Row

- **Each analytic function is based upon a current row within a Window (defined by OVER or ORDER BY clause)**
- **Current Row is the reference point setting the start and end of a window**
- **A moving average defines a window that begins in a range or rows surrounding the current row.**

Standard SQL Example – three unions

```
SQL> -- Here is the pure SQL way with accesses to the employee table
SQL>
SQL> SELECT deptno, job, SUM(sal) sal ← Provides the dept-job totals
  2  FROM emp
  3  GROUP BY deptno, job
  4  UNION
  5  SELECT deptno, NULL job, SUM(sal) sal ← Provides the dept sub totals
  6  FROM emp
  7  GROUP BY deptno
  8  UNION
  9  SELECT TO_NUMBER(NULL) deptno, NULL job, SUM(sal) sal ← The
grand total
 10  FROM emp
 11  ORDER BY deptno, job;
```

Standard SQL Example – three unions result set

DEPTNO	JOB	SAL
10	CLERK	1300
10	MANAGER	2450
10	PRESIDENT	5000
10		8750
20	ANALYST	6000
20	CLERK	1900
20	MANAGER	2975
20		10875
30	CLERK	950
30	MANAGER	2850
30	SALESMAN	5600
30		9400
		29025

13 rows selected.

Analytic SQL Example – same three unions

```
SQL>
SQL> -- Here is the new way using ROLLUP and a single table access
SQL>
SQL> SELECT deptno, job, SUM(sal) sal
  2  FROM emp
  3  GROUP BY ROLLUP(deptno, job);
```

Analytic SQL Example – same three unions result set

DEPTNO	JOB	SAL
10	CLERK	1300
10	MANAGER	2450
10	PRESIDENT	5000
10		8750
20	ANALYST	6000
20	CLERK	1900
20	MANAGER	2975
20		10875
30	CLERK	950
30	MANAGER	2850
30	SALESMAN	5600
30		9400
		29025

13 rows selected.

Analytic SQL Example – some complications from NULL values

- SQL> -- But what if someone isn't assigned a JOB or Dept? Confusing NULLS!
- SQL>
- SQL> SELECT deptno, job, SUM(sal) sal
- 2 FROM emp
- 3 GROUP BY ROLLUP(deptno, job);

Analytic SQL Example – NULL values in result set

DEPTNO	JOB	SAL	
10	CLERK	1300	
10	MANAGER	2450	
10		3750	
20	ANALYST	6000	
20	CLERK	1900	
20	MANAGER	2975	
20		10875	
30	CLERK	950	
30	MANAGER	2850	
30	SALESMAN	4350	
30		1250	← Is this a subtotal?
30		9400	← What about this row?
	PRESIDENT	5000	
		5000	
		29025	

15 rows selected.

Analytic SQL Example – Grouping Function

```
SQL>
SQL> -- Use GROUPING Function.      Example uses new 8i CASE function
SQL> -- and traditional old DECODE function
SQL>
SQL> SELECT
  2         CASE WHEN         GROUPING(deptno) = 1
  3                           THEN 'All Depts'
  4                         WHEN (GROUPING(deptno) = 0 AND deptno IS NULL )
  5                           THEN 'No Dept'
  6                         ELSE TO_CHAR(deptno) END AS deptno,
  7         DECODE(GROUPING(job),
  8                           1, 'All Jobs',
  9                           0, NVL(job,'No Job'))  AS job,
 10         SUM(sal) sal
 11  FROM
 12         emp
 13  GROUP BY
 14         ROLLUP(deptno,job);
```

Analytic SQL Example – Grouping Function Result Set

DEPTNO	JOB	SAL
----------	---------	---------
10	CLERK	1300
10	MANAGER	2450
10	All Jobs	3750
20	ANALYST	6000
20	CLERK	1900
20	MANAGER	2975
20	All Jobs	10875
30	CLERK	950
30	MANAGER	2850
30	SALESMAN	4350
30	No Job	1250
30	All Jobs	9400
No Dept	PRESIDENT	5000
No Dept	All Jobs	5000
All Depts	All Jobs	29025

15 rows selected.

Analytic SQL Example – Cube Function

```
SQL>
SQL> -- Example using CUBE
SQL>
SQL> SELECT deptno, job, SUM(sal) sal
  2  FROM emp
  3  GROUP BY CUBE(deptno, job);
```

Analytic SQL Example – Cube Function Result Set

DEPTNO	JOB	SAL
10	CLERK	1300
10	MANAGER	2450
10	PRESIDENT	5000
10		8750
20	ANALYST	6000
20	CLERK	1900
20	MANAGER	2975
20		10875
30	CLERK	950
30	MANAGER	2850
30	SALESMAN	5600
30		9400
	ANALYST	6000
	CLERK	4150
	MANAGER	8275
	PRESIDENT	5000
	SALESMAN	5600
		29025

18 rows selected.

Analytic SQL Example – Cube Function Subtotal and Total only

```
SQL>
SQL> -- And if we only wanted the subtotal and grand total lines
SQL>
SQL> SELECT
  2       DECODE(GROUPING(deptno),1,'All Depts',deptno) deptno,
  3       DECODE(GROUPING(job),1,'All Jobs',job) job,
  4       SUM(sal) sal
  5  FROM
  6       emp
  7  GROUP BY
  8       CUBE(deptno, job)
  9  HAVING
 10       GROUPING(deptno) = 1 OR GROUPING(job) = 1;
```

Analytic SQL Example – Cube Function Subtotal and Total only Result Set

DEPTNO	JOB	SAL
----------	---------	---------
10	All Jobs	8750
20	All Jobs	10875
30	All Jobs	9400
All Depts	ANALYST	6000
All Depts	CLERK	4150
All Depts	MANAGER	8275
All Depts	PRESIDENT	5000
All Depts	SALESMAN	5600
All Depts	All Jobs	29025

9 rows selected.

Rollup and Cube Limitations

- **Total Rows created during GROUP BY operation.**
- **Total Rows can be filtered with HAVING operation.**
- **Total Rows sorted by ORDER BY operation.**
- **WHERE clause does not apply to Total Rows.**

Analytic SQL Example – Rank Function

```
SQL>
SQL> -- Rank example
SQL>
SQL> SELECT ename, sal,
  2   RANK() OVER (ORDER BY sal DESC NULLS LAST) AS null_rank_last,
  3   RANK() OVER (ORDER BY sal DESC NULLS FIRST) AS null_rank_first
  4  FROM emp;
```

Analytic SQL Example – Rank Function Result Set

ENAME	SAL	NULL_RANK_LAST	NULL_RANK_FIRST
KING		14	1
JONES	3000	1	2
SCOTT	3000	1	2
FORD	3000	1	2
BLAKE	2850	4	5
CLARK	2450	5	6
ALLEN	1600	6	7
TURNER	1500	7	8
MILLER	1300	8	9
WARD	1250	9	10
MARTIN	1250	9	10
ADAMS	1100	11	12
JAMES	950	12	13
SMITH	800	13	14

14 rows selected.

Analytic SQL Example – PARTITION BY clause

```
SQL>
SQL> -- Rank within department example (No skipping numbers i.e. DENSE)
SQL>
SQL> SELECT deptno, ename, sal,
  2   DENSE_RANK() OVER (ORDER BY sal DESC) AS overall_rank,
  3   DENSE_RANK() OVER (PARTITION BY deptno
  4                      ORDER BY sal DESC) AS dept_rank
  5  FROM emp
  6  ORDER BY deptno, sal DESC;
```

Analytic SQL Example – PARTITION BY clause Result Set

DEPTNO	ENAME	SAL	OVERALL_RANK	DEPT_RANK
10	KING	5000	1	1
10	CLARK	2450	4	2
10	MILLER	1300	7	3
20	JONES	3000	2	1
20	FORD	3000	2	1
20	SCOTT	3000	2	1
20	ADAMS	1100	9	2
20	SMITH	800	11	3
30	BLAKE	2850	3	1
30	ALLEN	1600	5	2
30	TURNER	1500	6	3
30	WARD	1250	8	4
30	MARTIN	1250	8	4
30	JAMES	950	10	5

14 rows selected.

Analytic SQL Example – RATIO_TO_REPORT function

```
SQL>
SQL> -- Want to see what percentage of payroll each person is in their dept?
SQL>
SQL> SELECT
  2        deptno, ename, sal,
  3        RATIO_TO_REPORT(sal)
  4                 OVER (PARTITION BY deptno) AS pct_of_dept
  5  FROM
  6        emp
  7  ORDER BY
  8        deptno, sal DESC;
```

Analytic SQL Example – RATIO_TO_REPORT function Result Set

DEPTNO	ENAME	SAL	PCT_OF_DEPT
10	KING	5000	.57
10	CLARK	2450	.28
10	MILLER	1300	.15
20	JONES	3000	.28
20	SCOTT	3000	.28
20	FORD	3000	.28
20	ADAMS	1100	.10
20	SMITH	800	.07
30	BLAKE	2850	.30
30	ALLEN	1600	.17
30	TURNER	1500	.16
30	WARD	1250	.13
30	MARTIN	1250	.13
30	JAMES	950	.10

14 rows selected.

Analytic SQL Example – In-Line Views and Top-N Statements

```
SQL>
SQL> -- Top-N example - get top two salary earners; notice ties are split
SQL>
SQL> SELECT *
  2  FROM (SELECT ename, sal
  3        FROM emp
  4        ORDER BY sal DESC) emp
  5  WHERE rownum < 3;
```

Analytic SQL Example – In-Line Views and Top-N Statements Result Set

ENAME	SAL
----------	---------
KING	5000
JONES	3000

Analytic SQL Example – In-Line Views and Top-N Statements Revised

```
SQL>
SQL> -- And to get the top 2 salary earners in each group - ties not split
SQL>
SQL> SELECT deptno, ename, sal
  2  FROM (SELECT deptno, ename, sal,
  3        RANK() OVER (PARTITION BY deptno
  4                     ORDER BY sal DESC) AS dept_rank
  5        FROM emp) emp
  6  WHERE dept_rank < 3;
```

Analytic SQL Example – In-Line Views and Top-N Statements Revised Result Set

DEPTNO	ENAME	SAL
10	KING	5000
10	CLARK	2450
20	JONES	3000
20	SCOTT	3000
20	FORD	3000
30	BLAKE	2850
30	ALLEN	1600

7 rows selected.

"Top N" Queries using RANK/DENSE_RANK

- **"Top N" queries may be solved easily by using RANK or DENSE_RANK in dynamic view (query in FROM clause).**
- **NULLs are treated like normal values and for ranking are treated as equal to other NULLs.**
- **The ORDER BY clause may specify NULLS FIRST or NULLS LAST.**
- **If unspecified, NULLS are treated as larger than any other value and appear depending upon the ASC or DESC part of the ORDER BY.**

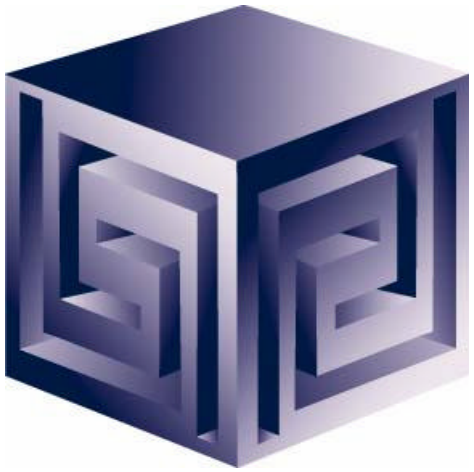
Conclusions

- **Analytic functions lend statistical muscle that has in the past called for joins, unions, and complex programming.**
- **Performance is improved (sometimes significantly) because the functions are performing work that previously required self-joins and unions.**
- **Using Analytic functions requires far less SQL coding than previously required to accomplish the same task because one SQL statement takes the place of many.**
- **Analytic functions allow division of results into ordered groups**

Oracle's SQL Analytic Functions in 8i and 9i

Presentation #278

IOUG-A Live! 2001



Presented by:

David Fuston

Vlamis Software Solutions, Inc.
(816) 781-2880

dfuston@vlamis.com

http://www.vlamis.com