Analytic SQL Features in Oracle9*i*

An Oracle Technical White Paper
December 2001

Analytic SQL Features in Oracle9*i*

Analytic SQL Features in Oracle9*i*

INTRODUCTION

Business intelligence processing requires advanced calculations including complex rankings, subtotals, moving averages, and lead/lag comparisons. These aggregation and analysis tasks are essential to queries that provide businesses with new insights. To meet the needs of enterprise-scale business intelligence, a data server must execute aggregation and analysis efficiently and scalably. Historically, the SQL language lacked features in this area, and this shortcoming forced developers to create inefficient and cumbersome code to achieve their goals. Oracle has significantly enhanced SQL to address the challenges of business intelligence. The enhancements have been added over several releases of the database:

- 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 have been incorporated into the international SQL-99 standard.
- Oracle9*i* adds several new families of analytic functions, plus important extensions to the GROUP BY clause such as Concatenated Groupings and GROUPING SETS. The new GROUP BY extensions add flexibility to aggregate processing and are integrated with Materialized Views for enhanced performance. In addition, Oracle9*i* enhances SQL processing for outstanding performance in complex analytic queries.

This paper discusses the analytic functions, GROUP BY extensions and query processing enhancements introduced in Oracle9*i*. For readers new to this topic, we also provide background material on the related Oracle8*i* features. Note that many business intelligence tools and applications already exploit these enhancements. Likewise, Oracle customers around the world are taking advantage of the new features in their internal applications.

ANALYTIC FUNCTIONS

Oracle provides eight families of analytic functions, three of which are new in Oracle9*i*. This paper offers details about the three new families added in Oracle9*i*. To learn the concepts and details of the analytic functions introduced before Oracle9*i*, please see Chapter 19 of the Oracle9*i* Data Warehousing Guide, available on the Oracle Technology Network.

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. Actions which before required self-joins or complex procedural processing may now be performed in simple SQL. The performance enhancements enabled by the new functions enhance query speeds for every kind of business intelligence task, ranging from production reports to ad hoc queries to complex OLAP queries.
- *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 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 international SQL standard, the new functions are attractive for independent software vendors: vendors have an incentive to adjust their products to take advantage of the new functions. Oracle is working with vendors of query, reporting and OLAP products to assist them in exploiting analytic functions. In the past, several database vendors have offered proprietary extensions in the same areas as these functions. However, those extensions did not achieve major market share, and few software vendors adjusted their products to support them. In contrast, the new analytic functions will be supported by a large number of independent software vendors.

Analytic Functions Family List

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 up 25% of the sales". The functions examine the entire output before producing an answer. 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 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 sorts of calculations simple: it lets one row contain values calculated at different aggregation levels. The 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 Oracle9*i*, 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 invoking the PERCENTILE_DISC function with parameter of 0.5.
- Hypothetical Rank and Distribution family - These functions, new in Oracle9*i*, 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 - An Oracle9*i* enhancement, this family enables queries to specify sorted aggregate groups and return the first or last value of each group. For instance, in a banking system storing daily balances for each customer's account, you could obtain the beginning or ending monthly balance.

Note that the analytic functions enhance the power of the database platform for decision support processing but they are not intended as a substitute for specialized OLAP facilities. In fact, OLAP products benefit greatly from using the analytic functions in the SQL they create. The Oracle9*i* OLAP option, for instance, leverages the power of the analytic functions to enhance its query performance.

The next sections describe key features of the analytic functions introduced in Oracle9*i* and provide basic examples. Since this paper is a descriptive and conceptual discussion of the functions, it does not include syntax diagrams. However, it does offer practical cases to show the value of the new functions. Most of the examples involve data from a sales table, where each row contains detail or aggregated sales data.

Inverse Percentile Family

One very common analytic question is to find the value in a data set that corresponds to a specific percentile. For example, what if we ask "what value is the median (50th percentile) of my data?" This question is the inverse of the information provided by the Oracle8*i* Release 2 CUME_DIST function, which answers the question "what is the percentile value for each row?"

Two new Oracle9*i* functions, PERCENTILE_CONT and PERCENTILE_DISC, compute inverse percentile. These functions require a sort specification and a percentile parameter value between 0 and 1. For instance, if a user needs the median value of income data, he would specify that the data set be sorted on income, and specify a percentile value of 0.5. The functions can be used as either aggregate functions or reporting aggregate functions. When used as aggregate functions, they return a single value per ordered set; and when used as reporting aggregate functions, they repeat the data on each row. PERCENTILE_DISC function returns the actual "discrete" value which is closest to the specified percentile values while the PERCENTILE_CONT function calculates a "continuous" percentile value using linear interpolation. The functions use a new WITHIN GROUP clause to specify the data ordering.

Example of Inverse Percentile as an aggregate function:

Consider the table HOMES which has the following data:

Area	Address	Price
Uptown	15 Peak St	456,000
Uptown	27 Primrose Path	349,000
Uptown	44 Shady Lane	341,000
Uptown	23301 Highway 64	244,000
Uptown	34 Design Rd	244,000
Uptown	77 Sunset Strip	102,000
Downtown	72 Easy St	509,000
Downtown	29 Wire Way	402,000

Area	Address	Price
Downtown	45 Diamond Lane	203,000
Downtown	76 Blind Alley	201,000
Downtown	15 Tern Pike	199,000
Downtown	444 Kanga Rd	102,000

To find the average and median value for each area, we could use the query below:

```
SELECT Homes.Area, AVG(Homes.Price),
  PERCENTILE_DISC (0.5) WITHIN GROUP
    (ORDER BY Homes.Price DESC),
  PERCENTILE_CONT (0.5) WITHIN GROUP
    (ORDER BY Homes.Price DESC)
  FROM Homes  GROUP BY Area;
```

Area	AVG	PERCENTILE_ DISC	PERCENTILE_ CONT
Uptown	289,333	341,000	292,500
Downtown	269,333	203,000	202,000

In the example above, to compute the median price of homes, the data is ordered on home price within each area and the median price is computed using the "discrete" and "interpolation" methods. The results above show how PERCENTILE_DISC returns actual values from the table, while PERCENTILE_CONT calculates new interpolated values.

Example of Inverse Percentile as a reporting function:

The inverse percentile functions can be used as reporting functions. In that usage, they will return a value that is repeated on multiple rows, allowing easy calculations. Consider the same HOMES table shown above. To find the median value for each area and display the results as a reporting function , we could use the query below:

```
SELECT Homes.Area, Homes.Price,
  PERCENTILE_DISC (0.5)
    WITHIN GROUP (ORDER BY Homes.Price DESC)
      OVER (PARTITION BY Area),
  PERCENTILE_CONT (0.5)
    WITHIN GROUP (ORDER BY Homes.Price DESC)
      OVER (PARTITION BY Area)
  FROM Homes;
```

Area	Price	PERCENTILE_ DISC	PERCENTILE_ CONT
Uptown	456,000	341,000	292,500
Uptown	349,000	341,000	292,500
Uptown	341,000	341,000	292,500
Uptown	244,000	341,000	292,500
Uptown	244,000	341,000	292,500

Area	Price	PERCENTILE_ DISC	PERCENTILE_ CONT
Uptown	102,000	341,000	292,500
Downtown	509,000	203,000	202,000
Downtown	402,000	203,000	202,000
Downtown	203,000	203,000	202,000
Downtown	201,000	203,000	202,000
Downtown	199,000	203,000	202,000
Downtown	102,000	203,000	202,000

Hypothetical Rank and Distribution Family

In certain analyses, such as financial planning, we may wish to know how a data value would rank if it were added to our data set. For instance, if we hired a worker at a salary of $50,000, where would his salary rank compared to the other salaries at our firm? The hypothetical rank and distribution functions support this form of what-if analysis: they return the rank or percentile value which a row would be assigned if the row was hypothetically inserted into a set of other rows. The hypothetical functions can calculate RANK, DENSE_RANK, PERCENT_RANK and CUME_DIST. Like the inverse percentile functions, the hypothetical rank and distributions functions use a WITHIN GROUP clause containing an ORDER BY specification.

Example of Hypothetical Rank function:

Here is a query using our real estate data introduced above. It finds the hypothetical ranks and distributions for a house with a price of $400,000.

```
SELECT Area,
  RANK (400000) WITHIN GROUP (ORDER BY Price DESC),
  DENSE_RANK (400000)
    WITHIN GROUP (ORDER BY Price DESC),
  PERCENT_RANK (400000)
    WITHIN GROUP (ORDER BY Price DESC),
  CUME_DIST (400000) WITHIN GROUP (ORDER BY Price DESC)
  FROM Homes
  GROUP BY Area;
```

Area	RANK	DENSE_RANK	PERCENT_ RANK	CUME_ DIST
Uptown	2	2	0.166	0.285
Downtown	3	3	0.333	0.428

Unlike Inverse Percentile functions, Hypothetical Rank functions cannot be used as reporting functions.

First/Last Aggregate Family

The FIRST/LAST aggregate functions apply an aggregate function over the rows that rank as the first or last with respect to a given order specification. FIRST/LAST lets us order the rows with one expression but apply the aggregate to a separate column. For instance, with a banking database we can take checking account transactions and GROUP BY customer, order each customer's transactions by date, and then return the amount of the first or last transaction for each customer. An aggregate function is applied to the transaction amount, so if there are multiple transactions on the first day we will return a single value.

While such queries can be created using a join or subquery, the SQL syntax is cumbersome and performance can be inefficient. The FIRST and LAST functions do this work with simpler SQL syntax and better performance. (Note that the analytic functions FIRST_VALUE and LAST_VALUE are not aggregate functions. They will return multiple rows per group.)

The FIRST and LAST aggregates introduce a new clause starting with the keyword KEEP. The KEEP clause specifies the ordering of a group and whether the first or last rows of that group are needed. In many cases, the ordering of the group will result in tie values for the first or last rows. Therefore, it is very important to choose the appropriate aggregate function to apply to the column which is actually returned. The MIN or MAX functions are often used with FIRST and LAST, since they act as tiebreakers when the ordering results in tie values. FIRST and LAST aggregates can also be used with the OVER clause as reporting functions.

Example of FIRST/LAST:

Consider the following Sales_Employee table that shows the salaries and commissions of sales people in the Hardware and Software departments of a hypothetical company.

Name	Department	Salary	Commission
Jones	Hardware	1000	200
Jack	Hardware	1100	350
Martin	Hardware	2500	500
Brown	Hardware	3000	650
Scott	Hardware	3200	650
Young	Software	900	200
Adams	Software	1200	200
Clark	Software	2000	480
Miles	Software	2500	600
White	Software	2400	650

Now consider the following query that returns the salaries of the employees who earn the highest and lowest commissions:

```
SELECT department,
  MIN(salary) KEEP (DENSE_RANK FIRST ORDER BY
     commission) as LOW_COMM_SALARY,
  MAX(salary) KEEP (DENSE_RANK LAST ORDER BY
     commission) as  HIGH_COMM_SALARY
  FROM Sales_Employee
  GROUP BY department;
```

Here, the MIN function is used as a tiebreaker to find the minimum salary paid to employees who earn the lowest commission. The MAX function is used as a tiebreaker to find the maximum salary paid to employees who earn the highest commission. The result for this query would be:

Department	LOW_COMM_SALARY	HIGH_COMM_SALARY
Hardware	1000	3200
Software	900	2400

Note that in the Hardware department, Brown and Scott are tied for the highest commission (650), but we choose the maximum salary of 3200 (Scott's) among the two for HIGH_COMM_SALARY. In the same department, only one person (Jones) earns the minimum commission, so we return 1000 as the LOW_COMM_SALARY in Hardware department. In the Software department, both Young and Adams are tied for the minimum commission (200), but Young earns the minimum salary (900) among the two, so we return his salary as the LOW COMM_SALARY for Software department. In the same department (software), only White makes the highest commission (650), so his salary (2400) is returned as the HIGH_COMM_SALARY.

AGGREGATION ENHANCEMENTS

To leverage the power of the database server, the SQL engine should offer powerful aggregation commands. Oracle's extensions to SQL's GROUP BY clause provide this power and bring many benefits including:

- Quicker and more efficient query processing
- Reduced client processing loads and network traffic: aggregation work is shifted to servers
- Simplified programming: less SQL code needed for many tasks
- Opportunities for caching aggregations: similar queries can leverage prior work

Without the aggregation enhancements in Oracle9*i*, many aggregation tasks require multiple queries against the same tables. A multi-query approach is inefficient, for it requires multiple scans of the same data. The extensions to the GROUP BY clause in Oracle9*i* allow the user to specify exactly what aggregations are needed in a single query, and they enable the optimizer to choose quicker execution plans. The GROUP BY extensions enable subtotal and grand total calculations such as those shown in the reference scenario below. They allow us to specify the following in a single GROUP BY clause:

- GROUPING SETS - perform multiple independent groupings for just the subtotals needed.
- CUBE and ROLLUP – specify complex grouping sets with efficient and convenient shortcuts
- Composite Columns - skip unneeded aggregation levels
- Concatenated groupings - concisely specify many complex groupings by automatically generating needed combinations.

Oracle first added SQL aggregation extensions in Oracle8*i*, with the CUBE and ROLLUP extensions to the GROUP BY clause. Oracle9*i* enhances this support by adding the GROUPING SETS and Concatenated Groupings extensions to the GROUP BY clause. All these extensions adhere to the ANSI SQL-99 standard. Many customers today use the extensions for a wide range of processing. These features are extremely useful in data warehousing, which usually involves extensive aggregate calculations. Materialized view creation and maintenance specially benefit from the GROUP BY extensions.

The feature descriptions begins with a brief reference scenario. GROUPING SETS are presented, followed by ROLLUP and CUBE. We build on these concepts with discussions of composite columns and concatenated grouping sets. Finally we cover hierarchical cubes and grouping functions, powerful features supporting OLAP tasks.

Reference Scenario

To illustrate the concepts of the GROUP BY extensions, this section uses a hypothetical videotape sales and rental company. The examples for aggregation enhancements will refer to data from this scenario. The videotape company has stores in several regions, and the data we use tracks profit information. The data is categorized by three dimensions: Time, Department and Region. For our examples, the time covers 2000 and 2001, the departments are Video Sales and Video Rentals, , and the Regions East, West, Central. The report below is a sample cross-tabular report showing the total profit by region and department in 2001:

2001			
Region	Department		Department Totals
	Video Rental Profit	Video Sales Profit	
Central	82,000	85,000	167,000
East	101,000	137,000	238,000
West	96,000	97,000	193,000
Regional Totals	279,000	319,000	598,000

Simple cross-tabular report, with subtotals shaded

To retrieve its core data, the six unshaded numbers, the report requires a SUM function with GROUP BY on Region and Department. In addition, the report needs five subtotals plus a grand total. The subtotals and grand total are the shaded numbers such as Video Rental Profits across regions (279,000) and Eastern region profits across department (238,000). Fully half the values needed for this report would not be calculated by a single query that did a GROUP BY Region, Department. Without the GROUP BY extensions, retrieving the needed data would require *four* queries with differing GROUP BY clauses: GROUP BY Region, Department; GROUP BY Region; GROUP BY Department; and no GROUP BY at all for the grand total..

GROUPING SETS

A grouping set is a precisely specified set of groups that the user wants the system to form. The feature enables users to specify multiple groupings in the GROUP BY clause of a single SELECT statement. Grouping sets produce a single result set which is equivalent to a UNION ALL of differently grouped rows. By specifying grouping sets, the query can be significantly more efficient than the UNION ALL approach. This efficiency gain occurs because GROUPING SETS enables the query optimizer to use lower-level aggregation groups as building blocks for higher levels groups.

Grouping sets are specified very easily by following the GROUP BY keywords with the term "GROUPING SETS" and a column specification list.

Example of GROUPING SETS

The SQL below calculates aggregates for 3 groupings - (Time, Region, Department), (Time, Department), and (Region, Department).

```
SELECT Time, Region, Department, sum(Profit)
  FROM Sales
  GROUP BY GROUPING SETS ((Time, Region, Department),
    (Time,Department), (Region,Department));
```

The output is:

Time	Region	Department	Profit
2000	Central	VideoRental	75,000
2000	Central	VideoSales	74,000
2000	East	VideoRental	89,000
2000	East	VideoSales	115,000
2000	West	VideoRental	87,000
2000	West	VideoSales	86,000
2000	[NULL]	VideoRental	251,000
2000	[NULL]	VideoSales	275,000
2001	Central	VideoRental	82,000
2001	Central	VideoSales	85,000
2001	East	VideoRental	101,000
2001	East	VideoSales	137,000
2001	West	VideoRental	96,000
2001	West	VideoSales	97,000
2001	[NULL]	VideoRental	279,000
2001	[NULL]	VideoSales	319,000
[NULL]	Central	VideoRental	157,000
[NULL]	Central	VideoSales	159,000
[NULL]	East	VideoRental	190,000
[NULL]	East	VideoSales	252,000
[NULL]	West	VideoRental	183,000
[NULL]	West	VideoSales	183,000

Compare this with the multi-query approach with UNION ALL:

```
SELECT Time, Region, Department, sum(Profit)
  FROM salesTable
  GROUP BY year, region, product
UNION ALL
  SELECT Time, NULL, Department, sum(Profit)
    FROM salesTable
    GROUP BY Time, Department
UNION ALL
  SELECT NULL, Region, Department, sum(Profit)
    FROM salesTable
    GROUP BY Region, Product;
```

The SQL above would require 3 scans of the base table, making it very inefficient.

The enhanced convenience and performance enabled by grouping sets can be applied in many areas. For data warehousing in particular the feature adds great value. Grouping sets can be used for pro-active caching of frequently accessed aggregates and for updating multiple summary tables in one pass using the new multi-table insert feature of Oracle9*i*.

Interpreting “[NULL]” Values in Results

The NULL values returned by the GROUP BY extensions do not always have the traditional meaning of “value unknown.” Instead, a NULL may indicate that its row is a subtotal. For instance, the first NULL value shown in the GROUPING SETS results above (in the seventh data row) is in the Region column. This NULL means that the row is a subtotal for "All Regions" for the VideoRental department in 2000. To avoid introducing another non-value in the database system, these subtotal values are not given a special tag. See the section on the GROUPING functions family for details on how NULLs representing subtotals are distinguished from NULLs stored in the data. Note that the NULLs shown in the figures of this paper are displayed only for clarity: in standard Oracle output, these cells would be blank.

ROLLUP

ROLLUP provides an extremely efficient shortcut for specifying a frequently needed pattern of groups. It creates subtotals which "roll up" from the most detailed level to the most summarized level, following the sequence of a grouping list specified in the ROLLUP clause. You can use the GROUPING SETS syntax to specify equivalent results, but working with ROLLUP will simplify the expression.

ROLLUP’s form is:

```
SELECT . . . GROUP BY
   ROLLUP( <grouping column reference list> )
```

ROLLUP takes as its argument an ordered list of grouping columns. First, it calculates the standard aggregate values specified in the GROUP BY clause. Then it creates progressively higher-level subtotals, moving from right to left through the list of grouping column reference list.

ROLLUP will create subtotals at n+1 levels, where n is the number of grouping columns. For instance, if a query specifies ROLLUP on grouping columns of Time, Region, Department (n=3), the result set will include rows at four aggregation levels. As mentioned earlier, ROLLUPs can be expressed as GROUPING SETs. For instance, ROLLUP(a, b, c) is equivalent to: GROUPING SETS ((a, b, c), (a, b),(a) ())

Note that the grouping “()” above defines the grand total.

Example of ROLLUP

Here is an example of ROLLUP using the data in the video store database.

```
SELECT Time, Region, Department,
 sum(Profit) AS Profit FROM sales
 GROUP BY ROLLUP(Time, Region, Dept)
```

This query returns the following sets of rows, shown below:

- Grouping by Time, Region and Dept: this is the regular aggregation that would be produced by GROUP BY without using ROLLUP
- Grouping by Time and Region: this is the first-level subtotal aggregating across Department for each combination of Time and Region
- Grouping by Time: this is the second-level subtotal aggregating across Region and Department for each Time value
- Grouping by all: a grand total row

Time	Region	Department	Profit
2000	Central	VideoRental	75,000
2000	Central	VideoSales	74,000
2000	Central	[NULL]	149,000
2000	East	VideoRental	89,000
2000	East	VideoSales	115,000
2000	East	[NULL]	204,000
2000	West	VideoRental	87,000
2000	West	VideoSales	86,000
2000	West	[NULL]	173,000
2000	[NULL]	[NULL]	526,000
2001	Central	VideoRental	82,000
2001	Central	VideoSales	85,000
2001	Central	[NULL]	167,000
2001	East	VideoRental	101,000
2001	East	VideoSales	137,000
2001	East	[NULL]	238,000
2001	West	VideoRental	96,000
2001	West	VideoSales	97,000
2001	West	[NULL]	193,000
2001	[NULL]	[NULL]	598,000
[NULL]	[NULL]	[NULL]	1,124,000

Suggested Uses of ROLLUP

The ROLLUP extension is widely used in tasks involving subtotals:

- ROLLUP is very helpful for subtotaling along a hierarchical dimension such as time or geography. For instance, a query could specify a ROLLUP(year, month, day) or ROLLUP(country, state, city).
- ROLLUP simplifies and speeds the population and maintenance of summary tables. Data warehouse administrators will want to make extensive use of it.

Note that the subtotals created by ROLLUP are only a fraction of possible subtotal combinations. If you wish to generate a cross tabular report that requires many subtotals, a single ROLLUP clause will not usually be sufficient. For instance, in the cross-tab shown in our reference scenario, the departmental totals across all regions (279,000 and 319,000) are not calculated by a ROLLUP(Time, Region, Department) clause. Generating those numbers would require a ROLLUP clause with the grouping columns specified in a different order: ROLLUP(Time, Department, Region). The easiest way to generate the full set of subtotals needed for cross-tabular reports is to use the CUBE extension.

CUBE

CUBE is another powerful shortcut for a specifying a frequently needed pattern of groups. CUBE finds subtotals based on all possible combinations of a set of grouping columns. This is the set of information typically needed for all cross-tabular reports, so CUBE can calculate the subtotals for a cross-tabular report with a single SELECT statement.

CUBE's form is:

```
SELECT . . .  GROUP BY
   CUBE ( <grouping column reference list> )
```

CUBE takes a specified set of grouping columns and creates subtotals for all possible combinations of them. In terms of multi-dimensional analysis, CUBE generates all the subtotals that could be calculated for a data cube with the specified dimensions. If you have specified CUBE(Time, Region, Department), the result set will include all the values that would be included in an equivalent ROLLUP statement plus additional combinations. If there are n columns specified for a CUBE, there will be 2^n combinations of subtotals returned.

As with ROLLUP, CUBEs can be expressed as GROUPING SETs. For instance, `CUBE(a, b, c)` is equivalent to:

```
GROUPING SETS ((a, b, c), (a, b), (a, c), (b, c),
  (a), (b), (c) ())
```

Example of CUBE

The query below gives an example of a three-column CUBE which would generate rows holding eight different (two to the third power) aggregate groups.

```
SELECT Time, Region, Department,
  SUM(Profit) AS Profit
  FROM sales
  GROUP BY CUBE  (Time, Region, Dept);
```

The CUBE query above would create the following four groupings that were found in the ROLLUP example: (Time, Region, Dept), (Time, Region), (Time), (). It would also create the following groupings: (Region), (Dept), (Time, Dept), (Region, Dept)

Suggested Uses of CUBE

In any situation requiring cross-tabular reports, CUBE is an important aid. Like ROLLUP, CUBE will be helpful in generating summary tables. Note that CUBE adds most value in queries that use columns from multiple dimensions rather than columns representing different levels of a single dimension.

For instance, a commonly requested cross-tabulation might need subtotals for all the combinations of (month, state, product). These are three independent dimensions, and analysis of all possible subtotal combinations will be commonplace. In contrast, a cross-tabulation showing all possible combinations of (year, month, day) would have several values of limited interest, since there is a natural hierarchy in the time dimension. Subtotals such as profit by day of month summed across year would be unnecessary in many analyses. Another important use of CUBE is the creation and maintenance of materialized views.

Hierarchy Handling with GROUP BY Extensions

The ROLLUP and CUBE extensions work independently of any hierarchy metadata in your system. Their calculations are based entirely on the columns specified in the SELECT statement in which they appear. This approach enables the extensions to be used whether or not hierarchy metadata is available. The simplest way to handle levels in hierarchical dimensions is by using the ROLLUP extension and indicating levels explicitly through separate columns. The code below shows a simple example of this with months rolled up to quarters and quarters rolled up to years.

```
SELECT Year, Quarter, Month,
  SUM(Revenues) AS Revenues
  FROM Revenues
  GROUP BY ROLLUP(Year, Quarter, Month);
```

This query returns the rows below. Note that the sample data below is unrelated to sample data sets in other queries.

Year	Quarter	Month	Revenues
2001	Winter	Jan	55,000
2001	Winter	Feb	64,000
2001	Winter	March	71,000
2001	Winter	[NULL]	190,000
2001	Spring	April	75,000
2001	Spring	May	86,000
2001	Sprint	June	88,000

2001	Spring	[NULL]	249,000
2001	Summer	July	91,000
2001	Summer	August	87,000
2001	Summer	September	101,000
2001	Summer	[NULL]	279,000
2001	Fall	October	109,000
2001	Fall	November	114,000
2001	Fall	December	133,000
2001	Fall	[NULL]	356,000
2001	[NULL]	[NULL]	1,074,000

Composite Columns

A composite column is a collection of columns, specified within parentheses, that is treated as a single unit during the computation of groupings. For example, by enclosing a subset of the columns in a ROLLUP list within parenthesis, some levels will be skipped during ROLLUP. Here is a simple case where (quarter, month) is a composite column:

```
GROUP BY ROLLUP (year, (quarter, month), day)
```

Because of its composite column of (quarter, month), the SQL above never separates the quarter and month columns in its ROLLUP. This means that it never shows a rollup at the (year, quarter) level - we have skipped the quarter level aggregates. Instead we will get the following groupings:

```
(year, quarter, month, day),
(year, quarter, month),
(year)
()
```

Example of Composite Columns

Here is a numeric example at the month and year level which skips the quarters:

```
SELECT Year, Quarter, Month,
    SUM(Revenues) AS Revenues FROM Revenues
    GROUP BY ROLLUP(Year, (Quarter, Month))
```

This query returns the rows below:

Year	Quarter	Month	Revenues
2001	Winter	Jan	55,000
2001	Winter	Feb	64,000
2001	Winter	March	71,000
2001	Spring	April	75,000
2001	Spring	May	86,000
2001	Sprint	June	88,000
2001	Summer	July	91,000

2001	Summer	August	87,000
2001	Summer	September	101,000
2001	Fall	October	109,000
2001	Fall	November	114,000
2001	Fall	December	133,000
2001	[NULL]	[NULL]	1,074,000

Composite columns are useful in ROLLUP, CUBE and concatenated groupings (described below). They can enhance performance and reduce the amount of result set filtering needed for queries.

Concatenated Grouping

Concatenated grouping offers a concise way to generate large combinations of groupings. Concatenated grouping creates a set of groups from the *cross-product* of groups generated by each grouped set, where a grouped set is an expression enclosed by a ROLLUP, CUBE or GROUPING SET. The cross-product is analogous to a Cartesian product performed on groups. Therefore the operation enables even a small number of concatenated groups to generate a large number of final groups. The concatenated groupings are specified simply by listing multiple ROLLUPs, or CUBEs or GROUPING SETs and separating them with commas. Here is an example of concatenated grouping set syntax:

```
  GROUP BY GROUPING SETS (a, b), GROUPING SETS (c, d)
```

The first GROUPING SET creates groups (a) and (b), while the second GROUPING SET creates groups (c) and (d). The cross product of these groups is:

`(a, c), (a, d), (b, c)` and `(b, d).`

Concatenated grouping is very helpful for these reasons:

- Ease of query development and maintenance – Developers and applications need not enumerate all groupings manually, a major issue when dealing with the hierarchical cubes described in the next section.
- Use by applications – Many applications can leverage the feature. In particular, SQL generated by OLAP applications often involves concatenated groups.

Hierarchical Cubes

One of the most important uses for concatenated grouping is to generate the aggregates needed for a *hierarchical cube* of data. A hierarchical cube is a data set where the data is aggregated along the rollup hierarchy of each of its dimensions and these aggregations are combined across dimensions. Note that a hierarchical cube is not the same as a CUBE aggregation for all key columns of multiple dimensions. A hierarchical cube is a smaller set of groups which includes the set of aggregations needed for business intelligence queries. We can generate all the aggregations needed for a hierarchical cube by using concatenated rollup: we define a ROLLUP expression along the hierarchy of each dimension and then concatenate the ROLLUP's to create many groups. Below we show a simple hierarchical cube example.

Consider a sales data set with three dimensions, each of which has a 3-level hierarchy:

Time: year, quarter, month

Product: category, brand, item

Geography: region, state, city

For our business intelligence needs, we would like to calculate and store the aggregates for various combinations of the three dimensions. We can use ROLLUP on each dimension to specify its useful aggregates. Once we have the ROLLUP-based aggregates of each dimension, we concatenate them with each other. This will generate our hierarchical cube.

Example of Concatenated GROUPING SETS

Below we show a GROUP BY clause for our sales data set. It creates a hierarchical cube using concatenated ROLLUP operations:

```
GROUP BY ROLLUP(year, quarter, month),
  ROLLUP(category, brand, item),
  ROLLUP(region, state, city)
```

The ROLLUP's in the GROUP BY specification above generate the following groups, 4 for each dimension:

ROLLUP by Time	ROLLUP by Product	ROLLUP by Geography
year, quarter, month	Category, brand, item	region, state, city
year, quarter	Category, brand	region, state
year	Category	region
all times	all products	all geographies

The concatenated rollup specified in the SQL above will perform a cross-product on the ROLLUP aggregations listed in the table. The cross-product will create the 64 (4x4x4) aggregate groups needed for a hierarchical cube of the data. There are major advantages in using three ROLLUP expressions to replace what would otherwise require 64 grouping set expressions: the concise SQL is far less error-prone to develop and far easier to maintain, and it enables much better query optimization. Readers can picture how a cube with more dimensions and more levels would make the use of concatenated groupings even more advantageous. For instance, if we added just one extra 4-level dimension and lacked concatenated rollup, we would need to specify 256 grouping sets.

GROUPING Functions Family

Two challenges arise with the use of GROUP BY extension. First, how can we programmatically determine which result set rows are subtotals, and how do we find the exact level of aggregation of a given subtotal? We will often need to use subtotals in calculations such as percent-of-totals, so we need an easy way to determine which rows are the subtotals we seek. Second, what happens if query results contain both stored NULL values and "NULL" values created by a GROUP BY extension? How does an application or developer differentiate between the two?

To handle these issues, Oracle 8*i* introduced a function called GROUPING, and Oracle9i introduces the GROUPING_ID and GROUP_ID functions

GROUPING Function

Using a single column as its argument, GROUPING returns 1 when it encounters a NULL value created by a GROUP BY extension. That is, if the NULL indicates the row is a subtotal, GROUPING returns a 1. Any other type of value, including a stored NULL, will return a 0. GROUPING() appears in the selection list portion of a SELECT statement. Its form is:

```
SELECT …  [GROUPING(dimension column)…]  …
  GROUP BY …    {CUBE | ROLLUP}
```

Example of Grouping Function

This example uses GROUPING to create a set of mask columns for the result set shown in the prior example.   The mask columns are easy to analyze programmatically.

```
SELECT Time, Region, Department, SUM(Profit) AS Profit,
  GROUPING (Time) as T,
  GROUPING (Region) as R,
  GROUPING (Department) as D
  FROM Sales
GROUP BY ROLLUP (Time, Region, Department);
```

Time	Region	Department	Profit	T	R	D
2000	Central	Video Rental	75,000	0	0	0
2000	Central	Video Sales	74,000	0	0	0
2000	Central	[NULL]	149,000	0	0	1
2000	East	Video Rental	89,000	0	0	0
2000	East	Video Sales	115,000	0	0	0
2000	East	[NULL]	204,000	0	0	1
2000	West	Video Rental	87,000	0	0	0
2000	West	Video Sales	86,000	0	0	0
2000	West	[NULL]	173,000	0	0	1
2000	[NULL]	[NULL]	526,000	0	1	1
2001	Central	Video Rental	82,000	0	0	0
2001	Central	Video Sales	85,000	0	0	0
2001	Central	[NULL]	167,000	0	0	1
2001	East	Video Rental	101,000	0	0	0
2001	East	Video Sales	137,000	0	0	0
2001	East	[NULL]	238,000	0	0	1
2001	West	VideoRental	96,000	0	0	0
2001	West	VideoSales	97,000	0	0	0
2001	West	[NULL]	193,000	0	0	1
2001	[NULL]	[NULL]	598,000	0	1	1
[NULL]	[NULL]	[NULL]	1,124,000	1	1	1

An application program can easily identify the detail rows above by a testing for the GROUPING() pattern "0 0 0" on the T, R, and D columns. The first level subtotal rows can be found with the pattern "0 0 1," the second level subtotal rows have a pattern of "0 1 1," and the grand total row's pattern is "1 1 1." Note that we need to use the GROUPING family function inside a HAVING clause if we want to use it to filter query results.

GROUPING_ID Function

To find the GROUP BY level of a particular row, our query must return GROUPING function information for each of the GROUP BY columns. If we do this using the GROUPING function, *every* GROUP BY column requires another column using the GROUPING function. For instance, a four-column GROUP BY clause needs to be analyzed with four GROUPING functions. This is inconvenient to write in SQL and significantly increases the number of columns required in the query. When we wish to store the query result sets in tables, as with materialized views, the extra columns waste storage space.

To address these problems, Oracle9*i* introduces the GROUPING_ID() function. GROUPING_ID returns a single number that lets us determine the exact GROUP BY level. For each row, GROUPING_ID takes the set of 1's and 0's that would be generated if we used the appropriate GROUPING functions and concatenates them, forming a "bit-vector." The bit vector is treated as a binary number, and the number's base-10 value is returned by the GROUPING_ID function. For instance, if we group with the expression "CUBE(a, b)" the possible values are as shown below:

Aggregation Level	Bit Vector	GROUPING_ID
a, b	0 0	0
a	0 1	1
b	1 0	2
grand total	1 1	3

GROUPING_ID clearly distinguishes groupings created by grouping set specification, and it is very useful during refresh and rewrite of materialized views.

GROUP_ID Function

While the extensions to GROUP BY give us great power and flexibility, they also allow complex result sets that may include duplicate groupings. The GROUP_ID function lets us distinguish among duplicate groupings. If there are multiple sets of rows calculated for a given level, GROUP_ID assigns the value of 0 to all the rows in the first set. All other sets of duplicate rows for a particular grouping are assigned higher values, starting with 1.

Example of GROUP_ID

The query below generates a duplicate grouping which is identified with the GROUP_ID function. Note that this example uses a different data set from our video store to illustrate the concept more clearly.

```
SELECT Region, State, sum(sales) as sum_sales,
  GROUPING_ID(state, region),
  GROUP_ID()
FROM salesTable
GROUP BY GROUPING SETS (region, ROLLUP(region, state));
```

The query above generates the following groupings: (region, state), (region), (region), and (). Note that the grouping (region) is repeated twice. Unlike the GROUPING_ID() function, GROUP_ID function distinguishes such duplicate groupings. The following table shows this difference:

Region	State	sum_sales	GROUPING_ID	GROUP_I
E	NY	1000	0	0
W	CA	2000	0	0
E	[NULL]	1000	2	0
E	[NULL]	1000	2	1
W	[NULL]	2000	2	0
W	[NULL]	2000	2	1
[NULL]	[NULL]	3000	3	0

This function lets users filter out duplicate groupings from result sets. For example, we can filter out duplicate (region) groupings from the above example by adding a HAVING clause with the condition "GROUP_ID() = 0" to the query.

Integration with Materialized Views

Materialized views, also called summary tables, are an important technique for maximizing the query performance of any decision support system. In Oracle9*i*, the CUBE, ROLLUP and GROUPING SETS extensions to GROUP BY may be used in the creation and maintenance of materialized views. This feature allows multiple levels of groupings to be contained in a single materialized view. By allowing the replacement of many (potentially hundreds) of small single-level materialized views with a small number of multi-level materialized views, Oracle9*i* simplifies database management. In addition, materialized view creation and maintenance become faster when using this feature.

Example of Materialized View created with ROLLUP

Here is an example of a materialized view created using the ROLLUP extension:

```
CREATE MATERIALIZED VIEW mv AS
  SELECT productline, productfamily, productid, region,
    state, city, sum(sales) as sum_sales,
    grouping_id(productline,
    productfamily,productid,region,state,city)
  FROM salesTable
  GROUP BY productline, ROLLUP(productfamily,
    productid), region, ROLLUP(state, city);
```

The SQL above computes nine subtotals across the product and geography dimension hierarchy and stores them into a single materialized view. The subtotals are:

- productline, productfamily, productid, region, state, city
- productline, productfamily, productid, region, state
- productline, productfamily, productid, region
- productline, productfamily, region, state, city
- productline, productfamily, region, state

- productline, productfamily, region
- productline, region, state, city
- productline, region, state
- productline, region

Contrast the code above with creating and managing nine separate materialized views each containing a single-level summary (GROUP BY without any extension). The multiple materialized view approach would require multiple scans of the base data. The other advantage is in the computation of ROLLUP, CUBE, and GROUPING SETS. The optimizer internally detects whether certain set of groups can be derived together without re-sorting the data and computes them together, thereby minimizing sort time and maximizing performance.

Note that multi-level materialized views like the one above can be partitioned like any other materialized view. For example, they can be partitioned on a grouping_id column and thus store different summary levels in separate partitions. If a materialized view containing multiple summary levels is suitably partitioned, it has the same query rewrite capabilities as multiple materialized views each containing a single summary level. For some types of queries, the multi-level materialized view offers performance advantages over multiple single-level materialized views.

Many data warehouses will be able to take advantage of the manageability and performance benefits provided by multi-level materialized views.

SQL PROCESSING ENHANCEMENTS FOR COMPLEX ANALYTIC QUERIES

Along with the its new SQL syntax, Oracle9*i* adds powerful enhancements to its internal SQL processing. These enhancements enable highly efficient processing of complex analytic queries. OLAP tools and analytic applications alike will benefit from these improvements. The Oracle9*i* OLAP product already takes full advantage of the SQL query structure described in this section.

Since internal processing changes are not visible to developers in the form of new SQL keywords, we need to look under the covers of query processing. This section presents the enhancements through a detailed study of the processing steps for a sample analytic query. The material here uses a relatively simple query to illustrate the concepts, but the techniques apply to extremely complex cases. To keep the presentation as clear as possible and focus on the essential points, the SQL shown does not contain every word required for the executable code. For instance, it does not show join conditions.

Complex Analytic Query Specification

Earlier in the paper, we discussed how Oracle9*i* can specify hierarchical cubes in a simple and efficient SQL query. These hierarchical cubes represent the logical cubes referred to in many OLAP products. To handle complex analytic queries, the core technique is to enclose a hierarchical cube query in an outer query which specifies the exact slice needed from the cube. Oracle9*i* optimizes the processing of hierarchical cubes nested inside slicing queries. By applying many powerful algorithms, these queries can be processed at unprecedented speed and scale. This enables OLAP tools and analytic applications to use a consistent style of queries to handle the most complex questions.

For clarity, the presentation that follows divides processing into three parts. Queries are first modified for high efficiency with a transformation phase. Then an optimization phase occurs. Finally, the query execution phase leverages powerful data structures and system management features of Oracle9*i*.

Sample Query

Consider the analytic query below. It consists of a hierarchical cube query nested in a slicing query. We will examine how it is transformed and optimized step-by-step.

```
SELECT Month, Division, sum_sales FROM
  (SELECT Year, Quarter, Month, Division,
      Brand, Item, Sum(Sales) sum_sales,
      GROUPING_ID(Year, Quarter, Month,
                  Division, Brand, Item) gid
   FROM Sales, Products, Time
   WHERE <join-condition>
   GROUP BY
     ROLLUP(Year, Quarter, Month),
     ROLLUP(Division, Brand, Item)
  )
WHERE Division = 25
  AND Month = 200001
  AND gid = 3;
```

The inner hierarchical cube specified above defines a relatively simple cube, with two dimensions and four levels in each dimension. It would generate 16 groups (4 Time levels * 4 Product levels). The GROUPING_ID function in the query identifies the specific group each row belongs to, based on the aggregation level of the <grouping-columns> in its argument.

The outer query applies the constraints needed for our specific query, limiting Division to a value of 25 and Month to a value of 200001 (representing January 2000 in this case). In conceptual terms, it slices a small chunk of data from the cube. The outer query's constraint on the gid column, gid = 3, indicates that we seek rows with both the Brand and Item columns aggregated. For such rows, the GROUPING_ID function creates a bit vector of 000011, which is 3. The gid constraint selects only those rows that are aggregated at the level of a "GROUP BY Month, Division" clause. Note that all the business intelligence constraints would typically be numeric key values generated by the application which created the SQL.

Query Transformation Phase

The first set of changes to the query is referred to as query transformation. Query transformation simplifies the original request to enhance the efficiency of the query processing. These steps include group pruning, predicate push-down and move-around, and materialized view rewrite. All of these transformations are entirely transparent to queries, since they are done automatically within the RDBMS.

Group pruning

Group pruning removes unneeded aggregation groups from query processing based on the <BI-query-conditions>. The outer conditions of our query above limit the result set to a single group aggregating Division and Month. Any other groups involving Year, Month, Brand and Item are unnecessary here. The group pruning transformation recognizes this and transforms the query into:

```
SELECT Month, Division, sum_sales  FROM
  (SELECT  Year, Quarter, Month, Division,
      null, null, Sum(Sales) sum_sales,
      GROUPING_ID(Year, Quarter, Month,
                  Division, Brand, Item) gid
   FROM Sales, Products, Time
   WHERE <join-condition>
   GROUP BY
     Year, Quarter, Month, Division )
WHERE Division = 25
  AND Month = 200001
  AND gid = 3;
```

The bold italic items above highlight the changed SQL. The inner query now has a simple GROUP BY clause of Month, Division. The columns Year, Quarter, Brand and Item have been converted to null to match the simplified GROUP BY clause. Since the query now requests just one group, fifteen out of sixteen groups are removed from the processing, greatly reducing the work. For a cube with more dimensions and more levels, the savings possible through group pruning can be far greater. Note that the group pruning transformation works with all the GROUP BY extensions: ROLLUP, CUBE, and GROUPING SETS.

Predicate Push-Down

Along with group pruning, we should transform the query to apply query constraints as early as possible in the processing sequence. When we take constraints, formally referred to as "predicates," and move them into an inner query, the transformation is referred to as predicate push-down. Constraining earlier in the processing reduces the number of rows that must be handled by later steps in the sequence, greatly speeding query performance. For our sample query, the system could move the constraints on Division, Month and grouping ID from the outer query to the inner query, creating:

```
SELECT Month, Division, sum_sales  FROM
  (SELECT Year, Quarter, Month, Division,
      null, null, Sum(Sales) sum_sales,
      GROUPING_ID(Year, Quarter, Month,
                  Division, Brand, Item) gid
   FROM Sales, Products, Time
   WHERE <join-condition>
      AND Division = 25
      AND Month = 200001
      AND gid = <gid-for-Division-Month>
   GROUP BY Year, Quarter, Month, Division
   HAVING gid = 3);
```

Now that the inner query is constrained on Division and Month, the processing returns only the relevant month and division to the outer query. Since the data access can take advantage of indexes to find just the required rows, the query is far more efficient.

View Merging

Yet another way to improve performance is through the technique of view merging. A query with in-line views can be analyzed to see if the views could be merged to simplify processing. While complex queries may involve complex view merging, even our very simple example offers an opportunity to perform this transformation. We can merge the inner query to create the following SQL:

```
SELECT Month, Division,
      Sum(Sales) sum_sales,
      GROUPING_ID(Year, Quarter, Month,
                  Division, Brand, Item) gid
FROM Sales, Products, Time
WHERE <join-condition>
      AND Division = 25
      AND Month = 200001
GROUP BY Year, Quarter, Month, Division
HAVING gid = 3;
```

Our sample query has eliminated its in-line view and is now a much simpler query.

Materialized View Rewrite

Our query can be further transformed to leverage materialized views. Instead of using the sales table, the query now directly accesses the appropriate materialized view, here labeled "MatView":

```
SELECT Month, Division,
       sum_sales, gid
FROM   MatView
WHERE
       Division = 25
       AND Month = 200001
       AND gid = 3;
```

The MatView materialized view holds pre-aggregated sales data in the column "sum_sales" so there is no time spent calculating totals. Likewise, the <join-condition> of the prior query version is eliminated, since the materialized view holds all needed information. The materialized view can hold many groupings generated by the GROUP BY extensions. These groupings can be identified with a column using the GROUPING_ID() function. The GROUPING_ID values enable group pruning and partition pruning. When a single materialized view holds many different groupings, it simplifies rewrite and data maintenance. In this example, the GROUPING_ID values are stored in the column "gid."

Predicate Move-Around

Predicate move-around can be considered an advanced form of predicate push-down. While predicate push-down places a predicate in an inner query to reduce the amount of rows passed to later stages of the query, predicate move-around applies predicates wherever they can be used to enhance processing efficiency. It will reach into an inner query block and move a predicate into a completely different query block, if that will improve performance. Predicate move-around can be called in several places during query transformation. Our sample query is too simple to use predicate move-around, but there are many situations in which the transformation can enhance processing efficiency.

Additional Passes of Group Pruning and Materialized View Rewrite

At this point, the processing will evaluate whether it is worth performing additional group pruning and materialized view rewrites. This evaluation is useful because the various transformations discussed above may create a form of the query that supports yet more transformations.

Query Optimization Phase

Our sample query has already been transformed into a much more efficient statement. However, we can enhance its speed still further with several query optimizations. These include partition pruning, index selection and analytic SQL optimization.

Partition Pruning Optimizations

By accessing only the relevant partitions of the MatView materialized view, we can sharply reduce the processing load of our query. If MatView was partitioned by the 16 possible groups of our cube, then there are 16 partitions based on a stored GROUPING_ID() value. All partitions not needed for our query are automatically pruned from the processing. This step eliminates a great deal of unnecessary data access: in the case of the sample query, 15 of 16 partitions never need to be accessed.

Bitmap Indexes

Once partitions are pruned, the query can be analyzed to see if any relevant bitmap indexes exist. Our query would use these access structures, and many analytic queries can take advantage of such indexes at this point. Using bitmap indexes limits I/O processing to just the qualifying rows of the partitions, enabling extremely fast performance.

SQL Optimizations for Analytic Functions

If a query uses analytic functions, the Oracle9*i* optimizer will now adjust the processing for maximal efficiency. Data sorting stands out as one of the critical optimization areas. Since each analytic function in a query includes its own sort specification, the same sort order may be specified multiple times in a single query. For these cases, it is important to avoid performing the same data sort multiple times. Therefore, the optimizer tracks exactly which sort orders are requested and reuses the sorted data set wherever possible. Another optimization for avoiding sorts is to exploit the data access method such as index access or sort-merge. If the data is accessed in exactly the sort order needed by an analytic function, no extra sorting is performed.

Query Execution Phase

Finally, the query execution phase of processing takes place. The Oracle9*i* optimizer, resource manager and memory management facilities will work together to ensure that the query is handled in the most efficient possible way, with full access to the necessary memory and processing cycles. When we consider the full set of transformations, optimizations and execution techniques that are available to our analytic queries, it becomes clear that Oracle9*i* can achieve excellent performance for extremely challenging queries.

CONCLUSION

Business intelligence processing in relational databases has historically been a challenging task. Analytic queries have required elaborate programming that led to slow performance. With its analytic functions, GROUP BY extensions and powerful optimizations for analytic queries, Oracle9*i* overcomes these challenges and speedily performs the hardest business intelligence queries.

Oracle9*i*'s analytic enhancements to the SQL language are readily accessible and easily incorporated into new and existing applications. The analytic functions and GROUP BY extensions benefit the full spectrum of business intelligence processing and conform to the international SQL standard. Developers of business intelligence tools and vertical market applications are leveraging these features. Oracle9*i* also enhances internal query processing to dramatically improve performance of complex analyses. Together, the analytic functions, GROUP BY extensions and query processing enhancements help make Oracle9*i* the leading platform for business intelligence and data warehousing.

Analytic SQL Features in Oracle9*i*
December 2001
Author: John Haydu
Contributing Author: Sankar Subramanian

Oracle Corporation
World Headquarters
500 Oracle Parkway
Redwood Shores, CA 94065
U.S.A.

Worldwide Inquiries:
Phone: +1.650.506.7000
Fax: +1.650.506.7200
www.oracle.com