The SQL Model Clause of Oracle Database 10*g*

An Oracle White Paper
August 2003

ORACLE

The SQL Model Clause of Oracle Database 10*g*

The SQL Model Clause of Oracle Database 10*g*

EXECUTIVE OVERVIEW

Complex calculations can be challenging in SQL, specially when they involve inter-row references. Such computations will often demand elaborate SQL Joins and Union statements, code that is difficult to develop, maintain and execute efficiently. Oracle Database 10*g* introduces an innovative approach to complex SQL calculations: the SQL Model clause.

An extension to SQL's Select statement, the Model clause enables developers to treat relational data as multidimensional arrays and define formulas on the arrays. It offers a concise, easy to read syntax and the functionality needed to handle demanding calculations. The Model clause resolves formula dependencies automatically, supporting large sets of interlinked formulas in sophisticated applications. In addition, Model clause processing employs advanced optimization techniques and data structures, for very high performance. The expressive power and ease of use of the Model clause, combined with the scalability and manageability of Oracle, provide a major advance for database applications.

INTRODUCTION

Many database applications, both in business and technical fields, require calculations which are difficult to build in SQL. Market share calculations involving different levels of a product or geography hierarchy or customized aggregations may need complex self-joins and union operations. Time series analyses and iterative calculations such as simultaneous equations can require moving data outside the database and into external calculation engines. Financial models like budgets and sales forecasts, with many logically interdependent formulas are specially challenging.

Although the limitations of SQL entail significant application development and maintenance burdens, there is a bigger issue to consider. When data is extracted from the database for external processing, administrative workload increases and manageability decreases. Picture financial projections based on extracted data and performed on PC spreadsheets. When many end users run their own copy of a standardized projection spreadsheet, *all* the machines must be updated whenever a change is made to the formulas. To fail in even one update means

incorrect financial calculations, a major risk in today's business environment. Even when results are based on consistent formulas, the output must be consolidated, a complex task demanding timely network access to many PC's. Reliable access to many PC's becomes ever more challenging as staffs shift to laptop computers that may not be connected to the network at all. Clearly, there would be major business advantages if complex calculations could be centralized within the database.

Oracle Database 10*g* addresses these needs with the SQL Model clause, a powerful new extension to the SQL SELECT statement. The Model clause simplifies and centralizes calculations for all types of applications. With the SQL Model clause, you can view query results in the form of multidimensional arrays and then apply formulas to calculate new array values. The formulas can be sophisticated interdependent calculations with inter-row and inter-array references. Applications built with the Model clause can replace PC spreadsheets and other external computation engines, providing more robust and efficient solutions. Integrating advanced calculations into the database significantly enhances performance, scalability and manageability compared to external computations:

- Performance – Model clause processing eliminates the need for many SQL join and union operations. It maximizes performance using advanced algorithms and data structures. At the most basic level, with Model clause data avoids the round trip required for external processing: copying data into separate applications, processing the data and then loading the results into the database.
- Scalability – Oracle's parallel query features support enterprise level scalability unmatched by external calculation tools such as PC spreadsheets. The Model clause leverages Oracle parallelism, efficiently using all system resources made available to it.
- Manageability – When computations are centralized close to the data, the inconsistency and poor security of calculations scattered across computational islands disappears. Also, data consolidation is simplified when applications share a common relational environment rather than a mix of calculation engines with incompatible data structures.

Concepts

The Model clause defines a multidimensional array by mapping the columns of a query into three groups: partitioning, dimension, and measure columns. These elements perform the following tasks:

- Partitions define logical blocks of the result set in a way similar to the partitions of the analytical functions (described in Oracle's *Data Warehousing Guide*). Each partition is viewed by the formulas as an independent array.

- Dimensions identify each measure cell within a partition. These columns are identifying characteristics such as date, region and product name.
- Measures are analogous to the measures of a fact table in a star schema. They typically contain numeric values such as sales units or cost. Each cell is accessed within its partition by specifying its full combination of dimensions.

To create formulas on these multidimensional arrays, you define computation rules expressed in terms of the dimension values. The rules are flexible and concise, and can use wild cards and FOR loops for maximum expressiveness. Calculations based on the Model clause improve on traditional MODEL calculations by integrating their analytical functions into the database, improving readability with symbolic referencing, and providing scalability and much better manageability.

The figure below gives a conceptual overview of Model using a hypothetical sales table. The table has columns for country, product, year and sales amount. The figure has three parts. The top segment shows the concept of dividing the table into partitioning, dimension and measure columns. The middle segment shows two rules that calculate the value of Prod1 and Prod2 for the year 2002. Finally, the third part shows the output of a query applying the rules to a table with hypothetical data. The unshaded output of is data retrieved from the database, while the shaded output shows rows calculated from rules. Note that the rules are applied within each partition.

Mapping of columns to model entities:

Country	Product	Year	Sales
Partition	Dimension	Dimension	Measure

Rules:

$$Sales(Prod1, 2002) = Sales(Prod1,2000)+Sales(Prod1,2001)$$
$$Sales(Prod2, 2002) = Sales(Prod2,2000)+Sales(Prod2,2001)$$

Output of model clause:

Country	Product	Year	Sales	
Partition	Dimension	Dimension	Measure	
A	Prod1	2000	10	Original Data
A	Prod1	2001	15	
A	Prod2	2000	12	
A	Prod2	2001	16	
B	Prod1	2000	21	
B	Prod1	2001	23	
B	Prod2	2000	28	
B	Prod2	2001	29	
A	Prod1	2002	25	Rule Results
A	Prod2	2002	28	
B	Prod1	2002	44	
B	Prod2	2002	57	

Note that the Model clause does not update existing data in tables, nor does it insert new data into tables: to change values in a table, the Model results are supplied to an INSERT or UPDATE or MERGE statement.

TECHNICAL DETAILS

The prior section provided an overview of the SQL Model clause. However, a solid understanding of the feature requires detailed examples. The rest of this paper presents a step-by-step presentation of essential Model concepts and keywords using examples. Not every feature is illustrated with an example, but we do provide at least a brief description of all major elements. For ease of understanding, all examples can be run with the sample data that ships with Oracle Database 10*g*. We use the sample schema SH, which is a star schema of the kind commonly used in business intelligence applications. SH has a fact table holding sales history for a consumer electronics vendor with international sales.

Basic Syntax

Here are the elements of the Model clause syntax which are discussed in this paper. While there are some additional syntax items, the set discussed here represents the core functionality. Note that the Model clause is processed after all other clauses of a SELECT statement except the final ORDER BY.

```
<prior clauses of SELECT statement>
MODEL [main]
 [reference models]
 [PARTITION BY (<cols>)]
  DIMENSION BY (<cols>)
  MEASURES (<cols>)
   [IGNORE NAV] | [KEEP NAV]
 [RULES
   [UPSERT | UPDATE]
   [AUTOMATIC ORDER | SEQUENTIAL ORDER]
   [ITERATE (n) [UNTIL <condition>]  ]
   (   <cell_assignment>  =  <expression>  ... )
```

Sample Data

To keep our examples concise, we will create a view using the Sales History (SH) schema of the sample schema set provided with Oracle10*g*. The view *sales_view* provides annual sums for product sales, in dollars and units, by country, aggregated across all channels. The view is built from a 1 million row fact table and defined as follows:

```
CREATE VIEW sales_view AS
SELECT country_name country, prod_name prod,
  calendar_year year,
  SUM(amount_sold) sale, COUNT(amount_sold) cnt
FROM sales, times, customers, countries, products
WHERE sales.time_id = times.time_id AND
  sales.prod_id = products.prod_id AND
  sales.cust_id = customers.cust_id AND
  customers.country_id = countries.country_id
GROUP BY country_name, prod_name, calendar_year;
```

If you wish to run the examples on your Oracle system with minimal execution time, you can create a materialized view of this statement. Oracle's summary management feature will automatically rewrite the examples to take advantage of the materialized view.

First Model Clause Example

As an initial example of Model, consider the following statement. It calculates the sales values for two products and defines sales for a new product based on the other two products.

```
SELECT SUBSTR(country,1,20) country, SUBSTR(prod,1,15) prod,
year, sales
FROM sales_view
WHERE country IN  ('Italy','Japan')
MODEL   RETURN UPDATED ROWS
  PARTITION BY (country)
  DIMENSION BY (prod, year)
  MEASURES (sale sales)
  RULES  (
    sales['Bounce', 2002] = sales['Bounce', 2001] +
      sales['Bounce', 2000],
    sales['Y Box', 2002] = sales['Y Box', 2001],
    sales['2_Products', 2002] = sales['Bounce', 2002] +
sales['Y Box', 2002])
ORDER BY country, prod, year;
```

The results are:

```
COUNTRY              PROD                 YEAR      SALES
-------------------- --------------- ---------- ----------
Italy                2_Products             2002   92613.16
Italy                Bounce                 2002    9299.08
Italy                Y Box                  2002   83314.08
Japan                2_Products             2002   103816.6
Japan                Bounce                 2002   11631.13
Japan                Y Box                  2002   92185.47
```

This statement partitions data by country, so the formulas are applied to data of one country at a time. Our sales fact data ends with 2001, so any rules defining values for 2002 or later will insert new cells. The first rule defines the sales of a video games called "Bounce" in 2002 as the sum of its sales in 2000 and 2001. The second rule defines the sales for Y Box in 2002 to be the same value they

were for 2001. The third rule defines a product called "2_Products," which is simply the sum of the Bounce and Y Box values for 2002. Since the values for 2_Products are derived from the results of the two prior formulas, the rules for Bounce and Y Box must be executed before the 2_Products rule.

Note the following characteristics of the example above:

- The "RETURN UPDATED ROWS" clause following the keyword MODEL limits the results returned to just those rows that were created or updated in this query. Using this clause is a convenient way to limit result sets to just the newly calculated values. We will use the RETURN UPDATED ROWS clause throughout our examples.
- The keyword "RULES," shown in all our examples at the start of the formulas, is optional, but we include it for easier reading.
- Likewise, many of our examples do not require ORDER BY on the Country column, but we include the specification for convenience in case readers want to modify the examples and use multiple countries.

REFERRING TO CELLS AND VALUES

This section examines the techniques for referencing cells and values in a SQL Model. The material on cell references is essential to understanding the power of the SQL Model clause.

Positional Cell Reference – Single cell access and upserts

What if we want to update the existing sales value for the product Bounce in the year 2000, in Italy, and set it to 10? We could do it with a query like this, which updates the existing cell for the value:

```
SELECT SUBSTR(country,1,20) country, SUBSTR(prod,1,15) prod,
  year, sales
FROM sales_view
WHERE country='Italy'
MODEL   RETURN UPDATED ROWS
  PARTITION BY (country)
  DIMENSION BY (prod, year)
  MEASURES (sale sales)
  RULES  (
    sales['Bounce', 2000] = 10 )
ORDER BY country, prod, year;

COUNTRY              PROD                 YEAR      SALES
-------------------- --------------- ---------- ----------
Italy                Bounce               2000         10
```

The formula in the query above uses "positional cell reference." The value for the cell reference is matched to the appropriate dimension based on its position

in the expression. The DIMENSION BY clause of the model determines the position assigned to each dimension: in this case, the first position is product ("prod") and the second position is year.

What if we want to create a forecast value of the sales for the product Bounce in the year 2005, in Italy, and set it to 20? We could do it with a query like this:

```
SELECT SUBSTR(country,1,20) country, SUBSTR(prod,1,15) prod,
  year, sales
FROM sales_view
WHERE country='Italy'
MODEL   RETURN UPDATED ROWS
  PARTITION BY (country)
  DIMENSION BY (prod, year)
  MEASURES (sale sales)
  RULES  (
    sales['Bounce', 2005] = 20 )
ORDER BY country, prod, year;
```

```
COUNTRY              PROD                 YEAR      SALES
-------------------- --------------- ---------- ----------
Italy                Bounce               2005         20
```

The formula in the query above sets the year value to 2005 and thus creates a new cell in the array.

NOTE: If we want to create new cells, such as sales projections for future years, we must use positional references or FOR loops (discussed later in this paper). That is, positional reference permits both updates and inserts into the array. This is called the "upsert" process.

Symbolic Cell Reference: Multi-cell access and updates

What if we want to update the sales for the product Bounce in all years after 1999 where we already have values recorded? Again, we will change values for Italy and set them to 10. We could do it with a query like this:

```
SELECT SUBSTR(country,1,20) country, SUBSTR(prod,1,15) prod,
  year, sales
FROM sales_view
WHERE country='Italy'
MODEL   RETURN UPDATED ROWS
  PARTITION BY (country)
  DIMENSION BY (prod, year)
  MEASURES (sale sales)
  RULES (
    sales[prod='Bounce', year>1999] = 10 )
ORDER BY country, prod, year;
```

```
COUNTRY              PROD                 YEAR      SALES
-------------------- -------------- ---------- ----------
Italy                Bounce               2000         10
Italy                Bounce               2001         10
```

The formula in the query above uses "symbolic cell reference." With symbolic cell references, the standard SQL conditions are used to determine the cells which are part of a formula. You can use conditions such as <,>, IN, and BETWEEN. In this example the formula applies to any cell which has product value equal to Bounce and a year value greater than 1999. The example shows how a single formula can access multiple cells.

NOTE: Symbolic references are very powerful, but they are solely for updating existing cells: they cannot create new cells such as sales projections in future years. If a cell reference uses symbolic notation in any of its dimensions, then its formula will perform only updates. Later we will discuss FOR loops in the Model clause, which provide a concise technique for creating multiple cells from a single formula.

Positional and Symbolic Cell References in a single query

What if we want a single query to update the sales for several products in several years for multiple countries, and we also want it to insert new cells? Placing several formulas into one query is more efficient than running multiple single-formula queries, since it reduces the number of times we need to access data. It also allows for more concise SQL, supporting higher developer productivity. Here is an example that updates two existing products and inserts a new product. To explain the notation, the query has three in-line comments.

```
SELECT SUBSTR(country,1,20) country, SUBSTR(prod,1,15) prod,
  year,   sales
FROM sales_view  WHERE country IN  ('Italy','Japan')
MODEL  RETURN UPDATED ROWS
  PARTITION BY (country)
  DIMENSION BY (prod, year)
  MEASURES (sale sales)
  RULES (
    sales['Bounce', 2002] = sales['Bounce', year = 2001] ,
      --positional notation:  can insert new cell
    sales['Y Box', year>2000] = sales['Y Box', 1999],
      --symbolic notation: can update existing cell
    sales['2_Products', 2005] = sales['Bounce', 2001] +
      sales['Y Box', 2000]  )
      --positional notation:  permits creation of new cell
      --for new product
  ORDER BY country, prod, year;
```

```
COUNTRY              PROD                 YEAR      SALES
-------------------- -------------- ---------- ----------
Italy                2_Products             2005   34579.63
Italy                Bounce                 2002    4928.65
Italy                Y Box                  2001    15177.7
Japan                2_Products             2005   52563.55
Japan                Bounce                 2002    6443.77
Japan                Y Box                  2001   22297.76
```

Since our example data has no values beyond the year 2001, any rule involving the year 2002 or later requires insertion of a new cell. The same applies to any new product name we define here. In the third formula we define a new product '2_Products' for 2005, so a cell will be inserted for it. The first rule, for Bounce in 2002, inserts new cells since it is positional notation. The second rule, for Y Box, uses symbolic notation, but since there are already values for 'Y Box' in the year 2001, it updates those values. The third rule, for '2_Products' in 2005, is positional, so it can insert new cells, and we see them in the output.

Multi-Cell References on the Right Side of a Formula

The earlier examples had multi-cell references only on the left side of the formulas. What if we want to refer to multiple cells on the right side of a formula? Multi-cell references can be used on the right side of formulas in which case an aggregate function needs to be applied on them to convert them to a single value. All existing aggregate functions including OLAP aggregates (inverse distribution functions, hypothetical rank and distribution functions etc.) and statistical aggregates, and user-defined aggregate functions can be used.

How can we forecast the sales of Bounce in Italy for the year 2005 to be 100 more than the maximum sales in the period 1999 to 2001?

```
SELECT SUBSTR(country,1,20) country, SUBSTR(prod,1,15) prod,
  year, sales
FROM sales_view
WHERE country='Italy'
MODEL   RETURN UPDATED ROWS
  PARTITION BY (country)
  DIMENSION BY (prod, year)
  MEASURES (sale sales)
  RULES (
    sales['Bounce', 2005] = 100 + MAX(sales)['Bounce',
      year BETWEEN 1998 AND 2002]  )
ORDER BY country, prod, year;

COUNTRY              PROD                 YEAR      SALES
-------------------- -------------- ---------- ----------
Italy                Bounce                 2005    5028.65
```

In the query above we use a BETWEEN condition to specify multiple cells on the right side of the formula, and these are aggregated to a single value with the MAX() function.

NOTE: Aggregate functions can appear only on the right side of formulas. Arguments used in the aggregate function can be constants, bind variables, measures of the MODEL clause, or expressions involving them.

CV() Function: Use left-side values in right side calculations

The CV() function is a very powerful tool used on the right side of formulas to copy left side specifications that refer to multiple cells. This allows for very compact and flexible multi-cell formulas. The CV() function is equivalent to a SQL join operation, but far more compact and readable.

What if we want to update the sales values for Bounce in Italy for multiple years, using a formula where: each year's sales is the sum of 'Mouse Pad' sales for that year plus 20% of the 'Y Box' sales for that year? We could do that with a query like this:

```
SELECT SUBSTR(country,1,20) country, SUBSTR(prod,1,15) prod,
  year, sales
FROM sales_view
WHERE country='Italy'
MODEL   RETURN UPDATED ROWS
  PARTITION BY (country)
  DIMENSION BY (prod, year)
  MEASURES (sale sales)
  RULES (
    sales['Bounce', year BETWEEN 1995 AND 2002] =
      sales['Mouse Pad', CV(year)] +
      0.2 * sales['Y Box', CV(year)]
  )
ORDER BY country, prod, year;

COUNTRY              PROD                 YEAR      SALES
-------------------- --------------- ---------- ----------
Italy                Bounce                 1999    7681.51
Italy                Bounce                 2000   9586.286
Italy                Bounce                 2001  21587.916
```

The two CV() functions used in the formula return the year dimension value of the cell currently referenced on the left side. When the left side of the formula above references the cell 'Bounce' and 1999, the right side expression would resolve to:

`sales['Mouse Pad', 1999] + 0.2 * sales['Y Box', 1999].`

Similarly, when the left side references the cell 'Bounce' and 2000, the right side expression we would evaluate is:

`sales['Mouse Pad', 2000] + 0.2 * sales['Y Box', 2000].`

CV() function takes a dimension key as its argument. It is also possible to use CV() without any argument as in CV() which causes positional referencing.

Therefore the formula above can be written as:

```
s ['Bounce', year BETWEEN 1995 AND 2002] =
  s['Mouse Pad', CV()] + 0.2 * s['Y Box', CV()]
```

Note that in the above results we see values for just years 1999-2001 although the condition would have accepted any year in the range 1995 to 2002. This is because our table has data for only the years 1999-2001. We used the wide time range in the formula to illustrate the formula flexibility.

Using CV() in expressions for inter-row calculations

CV() allows for very flexible expressions. For instance, by subtracting from the CV(year) value we can refer to other rows in our data set. If we have the expression "CV(year) –2" in a cell reference, we can access data from 2 years earlier.

What if we want to calculate the year over year percent growth in sales for products 'Y Box', 'Bounce' and 'Mouse Pad' in Italy? Here is a query for the task.

```
SELECT SUBSTR(country,1,10) country, SUBSTR(prod,1,10) prod,
year, sales, growth_pct
FROM sales_view
WHERE country='Italy'
MODEL     RETURN UPDATED ROWS
  PARTITION BY (country)
  DIMENSION BY (prod, year)
  MEASURES (sale sales, 0 growth_pct)
  RULES  (
    growth_pct[prod IN ('Bounce','Y Box','Mouse Pad'),
                year BETWEEN 1998 and 2001] =
        100* (sales[CV(prod), CV(year)] -
          sales[CV(prod), CV(year) -1] ) /
          sales[CV(prod), CV(year) -1] )
ORDER BY country, prod, year;
```

```
COUNTRY    PROD             YEAR       SALES GROWTH_PCT
---------- ---------- ---------- ----------- ----------
Italy      Bounce           1999    2,474.78
Italy      Bounce           2000    4,333.69      75.11
Italy      Bounce           2001    4,846.30      11.83
Italy      Mouse Pad        1998    3,055.69
Italy      Mouse Pad        1999    4,663.24      52.61
Italy      Mouse Pad        2000    3,662.83     -21.45
Italy      Mouse Pad        2001    4,747.90      29.62
Italy      Y Box            1999   15,215.16
Italy      Y Box            2000   29,322.89      92.72
Italy      Y Box            2001   81,207.55     176.94
```

It is important to note that the blank cells in the results are NULLs. The formula results in a null if there is no value for the product two years earlier. None of the products has a value for 1998, so in each case the 1999 growth calculation is NULL.

Wild Card with "ANY" keyword

A wild card operator is very useful for cell specification, and Model provides the ANY keyword for this purpose. We can use it with the prior example to replace the specification "year between 1998 and 2001" as shown below.

```
SELECT SUBSTR(country,1,10) country, SUBSTR(prod,1,10) prod,
year, sales, growth_pct
FROM sales_view
WHERE country='Italy'
MODEL     RETURN UPDATED ROWS
  PARTITION BY (country)
  DIMENSION BY (prod, year)
  MEASURES (sale sales, 0 growth_pct)
  RULES  (
    growth_pct[prod IN ('Bounce','Y Box','Mouse Pad'),
               ANY] =
        100* (sales[CV(prod), CV(year)] -
          sales[CV(prod), CV(year) -1] ) /
          sales[CV(prod), CV(year) -1] )
ORDER BY country, prod, year;


COUNTRY    PROD             YEAR       SALES GROWTH_PCT
---------- ---------- ---------- ----------- ----------
Italy      Bounce           1999    2,474.78
Italy      Bounce           2000    4,333.69      75.11
Italy      Bounce           2001    4,846.30      11.83
Italy      Mouse Pad        1998    3,055.69
Italy      Mouse Pad        1999    4,663.24      52.61
Italy      Mouse Pad        2000    3,662.83     -21.45
Italy      Mouse Pad        2001    4,747.90      29.62
Italy      Y Box            1999   15,215.16
Italy      Y Box            2000   29,322.89      92.72
Italy      Y Box            2001   81,207.55     176.94
```

This query gives the same results as the prior query because the full data set ranges from 1998 to 2001, and that is the range specified in the prior query.

ANY can be used in cell references to include all dimension values including NULLs. In symbolic reference notation, we use the phrase "IS ANY". Note that the ANY wildcard prevents cell insertion when used with either positional or symbolic notation.

FOR LOOPS – A CONCISE WAY TO SPECIFY NEW CELLS

The FOR construct enables a single formula to insert multiple new cells, acting like a wild card for the left side of formulas. (Note that the FOR constructs are allowed only on the left side of formulas.) As an example of FOR, consider the following formulas that estimate the sales of several products for year 2005 to be 30% higher than their sales for year 2001:

```
RULES
(  sales['Mouse Pad', 2005] = 1.3 * sales['Mouse Pad', 2001],
   sales['Bounce', 2005] = 1.3 * sales['Bounce', 2001],
   sales['Y Box', 2005] = 1.3 * sales['Y Box', 2001] )
```

By using positional notation on the left side of the formulas, we ensure that cells for these products in the year 2005 will be inserted if they are not already present in the array. This technique is bulky since it requires as many formulas as there are products. If we have to work with dozens of products, it becomes an

unwieldy approach. With FOR we can reword this computation so it is concise yet has exactly the same behavior?

```
SELECT SUBSTR(country,1,20) country, SUBSTR(prod,1,15) prod,
year, sales
FROM sales_view
WHERE country='Italy'
MODEL     RETURN UPDATED ROWS
  PARTITION BY (country)
  DIMENSION BY (prod, year)
  MEASURES (sale sales)
  RULES (
    sales[FOR prod IN  ('Mouse Pad', 'Bounce', 'Y Box'),
      2005] =  1.3 * sales[CV(prod), 2001]  )
ORDER BY country, prod, year;


COUNTRY              PROD                 YEAR          S
-------------------- --------------- ---------- ----------
Italy                Bounce               2005   6407.245
Italy                Mouse Pad            2005    6402.63
Italy                Y Box                2005 108308.304
```

If you write a specification similar to the above one, but without the FOR keyword, only cells which already exist would be updated, and no new cells would be inserted. In our data, that would mean no rows are returned. Here is that query:

```
SELECT SUBSTR(country,1,20) country, SUBSTR(prod,1,15) prod,
year, sales
FROM sales_view
WHERE country='Italy'
MODEL     RETURN UPDATED ROWS
  PARTITION BY (country)
  DIMENSION BY (prod, year)
  MEASURES (sale sales)
  RULES (
    sales[prod IN ('Mouse Pad', 'Bounce', 'Y Box'), 2005] =
      1.3 * sales[CV(prod), 2001]  )
ORDER BY country, prod, year;


no rows selected
```

The FOR construct can be thought of as a tool to make a single formula generate multiple formulas with positional references, thus enabling creation of new cells (UPSERT behavior).

FOR Loops which range over a value sequence

If the dimension values needed for a cell reference come from a sequence with regular intervals, you can use another form of the FOR construct:

```
 FOR dimension FROM <value1> TO <value2>
    [INCREMENT | DECREMENT] <value3>
```

This specification creates dimension values between value1 and value2 by starting from value1 and incrementing (or decrementing) by value3. Using a value range in a FOR loop enables extremely concise specification that apply to many existing cells or create many new cells.

OTHER CELL-HANDLING FEATURES

Order of evaluation of Formulas

By default, formulas are evaluated in the order they appear in the MODEL clause. The keywords "SEQUENTIAL ORDER" can be specified in the MODEL clause to make such an evaluation order explicit. To have models calculated so that all formula dependencies are processed in correct order, use the AUTOMATIC ORDER keywords. When a model has many formulas, it can be more efficient to use the AUTOMATIC ORDER option than to manually check that formulas are listed in a logically correct sequence. Using automatic order thus enables more productive development and maintenance of models.

NULL Measures and Missing Cells

Applications using SQL Models must work with two forms of non-deterministic values for a cell measure: cells which exist in the array but are assigned a NULL, and cells which are not in the array at all. A cell which is referred to by a cell reference but not found in the array is called a missing cell. MODEL clause default treatment for NULLs is the same as all other SQL tasks, and missing cells are treated by default as NULLs. Model also provides an alternate treatment of such cells. The keywords "IGNORE NAV" (where NAV stands for non-available values) can be added at the model or individual rule level. This phrase allows formulas to treat NULLs and missing cells as 0's in numeric calculations and as empty strings for character processing.

REFERENCE MODELS

The multi-dimensional array which has existing cells updated and new cells added is called the Main SQL Model. Along with the Main model, a Model clause can define one or more read-only multi-dimensional arrays, called Reference Models. The Reference Models serve as look-up tables. Using Reference Models, formulas can refer to arrays of different dimensionality. For instance, a profit projection could refer to a tax reference array and a costs reference array, where the tax is dimensioned by country and the cost is dimensioned by product. Another example would be a currency conversion calculation, with conversion factors treated as a reference model. Formulas which combine reference models, the wild card "ANY," and CV( ) functions are extremely flexible.

ITERATIVE MODELS

Using ITERATE option of the MODEL clause, you can evaluate formulas iteratively a specified number of times. We can use iterative models to calculate models where the formulas are interdependent. That is, the feature can solve simultaneous equations. The number of iterations is specified as an argument to the ITERATE clause.

Optionally, you can specify an early termination condition to stop formula evaluation before reaching the maximum iteration. This condition is specified in the UNTIL subclause of ITERATE and is checked at the end of an iteration. In some cases you may want the termination condition to be based on the change, across iterations, in value of a cell. Oracle provides a mechanism to specify such conditions by allowing you to access cell values as they existed before and after the current iteration in the UNTIL condition. You can also access the current iteration number for use in computations.

CONCLUSION

The challenges of specifying complex inter-row calculations in SQL have historically been met with solutions external to the database: programs in other languages and various types of calculation engines. With the SQL Model clause, Oracle Database 10*g* opens a whole new path to solving these challenges. The Model clause, an extension to the SELECT statement, treats relational data as multidimensional arrays in which every cell is accessible through a concise, flexible notation. As a result, complex SQL joins and unions are eliminated and processing is optimized. The Model clause automatically handles logical dependencies among formulas, further simplifying calculation development and maintenance. Oracle's parallel query processing powers are exploited by the Model clause, enabling enterprise level scalability.

The Model clause, combined with Oracle's analytic SQL enhancements in earlier releases of the database, makes Oracle a robust platform for advanced computations. Integrating advanced calculations into SQL improves manageability and simplifies data consolidation since data no longer needs to be extracted from the DBMS, processed externally, and have results inserted back into Oracle. Applications across the spectrum of database applications, and especially in Business Intelligence, will be able to leverage this important new feature.

APPENDIX: MODEL CLAUSE EXPLAIN PLANS

By performing an explain plan operation, you can find out the algorithm Oracle chooses to evaluate your Model. If your model has SEQUENTIAL ORDER formulas, then ORDERED is displayed. For AUTOMATIC ORDER Models, Oracle plans display ACYCLIC or CYCLIC based on whether the model has cyclic dependencies in its formulas. In addition, the plan output will have an annotation FAST in the case of ORDERED and ACYCLIC algorithms if all left side cell references are single cell references, and aggregates, if any, on the right side of formulas are simple arithmetic non-distinct aggregates like SUM, COUNT, AVG etc. Formula evaluation in this case would be highly efficient, so it is given the label "FAST."

Example Plan using ORDERED processing:

```
EXPLAIN PLAN FOR
SELECT SUBSTR(country,1,10) country,
  SUBSTR(prod,1,10) product, year, sales
FROM sales_view
WHERE country IN ('Italy','Brazil')
MODEL RETURN UPDATED ROWS
  PARTITION BY (country) DIMENSION BY (prod, year)
  MEASURES (sale sales)
  RULES SEQUENTIAL ORDER
  (
   sales['Bounce', 2003] =
     AVG(sales)[ANY, 2001] * 1.24,
   sales[prod != 'Y Box', 2000] =
     sales['Y Box', 2000] * 1.25
  );
```

The first five rows of results for this query are:

```
COUNTRY    PRODUCT          YEAR      SALES
---------- ---------- ---------- ----------
Italy      Bounce           2000 36653.6125
Italy      Mouse Pad        2000 36653.6125
Italy      Music CD-R       2000 36653.6125
Italy      Fly Fishin       2000 36653.6125
Italy      Deluxe Mou       2000 36653.6125
```

This query creates an explain plan starting with:

```
SELECT STATEMENT
  SQL MODEL ORDERED
```

Since the left side of the second formula is a multi-cell reference, Oracle will not choose the FAST method for the query.

Example plan using ACYCLIC FAST processing:

```
EXPLAIN PLAN FOR
SELECT SUBSTR(country,1,10) country,
  SUBSTR(prod,1,10) product, year, sales
FROM sales_view
WHERE country IN ('Italy','Brazil')
MODEL RETURN UPDATED ROWS
  PARTITION BY (country) DIMENSION BY (prod, year)
  MEASURES (sale sales)
  RULES  AUTOMATIC ORDER
  (
    sales['Bounce', 2004] =
      sales['Y Box', 2001] * 0.25,
    sales['Mouse Pad', 2004] =
      sales['Mouse Pad', 2001] / SUM(sales)[ANY, 2001] *
      2 *  sales['All Products', 2004],
    sales['All Products', 2004] = 200000
  );
```

The query returns:

```
COUNTRY    PRODUCT          YEAR      SALES
---------- ---------- ---------- ----------
Italy      All Produc       2004     200000
Italy      Bounce           2004 20301.8875
Italy      Mouse Pad        2004 1342.86407
Brazil     All Produc       2004     200000
Brazil     Bounce           2004
Brazil     Mouse Pad        2004  2344.7976
```

This query creates an explain plan starting with:

```
SELECT STATEMENT
 SQL MODEL ACYCLIC FAST
```

Formulas in this model are not cyclic and explain plan will show ACYCLIC. The FAST method is chosen in this case because it meets the two requirements listed at the start of the appendix.

The SQL Model Clause of Oracle Database 10*g*
August 2003
Author: John Haydu

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