

Analytical SQL in Oracle Database 12c

ORACLE WHITE PAPER | FEBRUARY 2015





Contents

Data Analysis with SQL	1
Flexible and Comprehensive Development Framework	2
In-database analytical SQL with Oracle Database 12c	4
Overview of in-database analytical SQL functionality	7
Conclusion	23
Further Reading	24



Data Analysis with SQL

Today information management systems along with operational applications need to support a wide variety of business requirements that typically involve some degree of analytical processing. These requirements can range from data enrichment and transformation during ETL workflows, creating time-based calculations like moving average and moving totals for sales reports, performing real-time pattern searches within logs files to building what-if data models during budgeting and planning exercises. Developers, business users and project teams can choose from a wide range of languages to create solutions to meet these requirements.

Over time many companies have found that the use so many different programming languages to drive their data systems creates five key problems:

1. Decreases the ability to rapidly innovate
2. Creates data silos
3. Results in application-level performance bottlenecks that are hard to trace and rectify
4. Drives up costs by complicating the deployment and management processes
5. Increases the level of investment in training

Development teams need to quickly deliver new and innovative applications that provide significant competitive advantage and drive additional revenue streams. Anything that stifles innovation needs to be urgently reviewed and resolved. The challenge facing many organizations is to find the right platform and language to securely and efficiently manage the data and analytical requirements while at the same time supporting the broadest range of tools and applications to maximize the investment in existing skills.

IT and project managers need an agile platform to underpin their projects and applications so that developers can quickly and effectively respond to ever-changing business requirements without incurring the issues listed above.

Flexible and Comprehensive Development Framework

Oracle's in-database analytical SQL offers an agile analytical development platform that enhances both query performance and developer productivity. It supports all types of processing, ranging from real-time analysis to interactive decision support to operational batch reporting. Using Oracle Database both corporate developers and independent software vendors can take advantage of a rich set of SQL based analytical features within their projects and applications.

The key benefits provided by Oracle's in-database analytical SQL are:

- Enhanced developer productivity
- Improved manageability
- Minimized learning effort
- Investment protection (through ANSI SQL compliance)
- Increased performance

Enhanced developer productivity

Using the latest built-in analytical SQL capabilities, developers can simplify their application code by replacing complex code – written in SQL and/or other languages - with analytical-SQL that is much clearer and more concise. Tasks that in the past required the use of procedural languages or multiple SQL statements can now be expressed using single, simple SQL statements. This simplified SQL (analytical SQL) is quicker to formulate, maintain and deploy compared to older approaches, resulting in greater developer productivity. The productivity benefits also extend to SQL-literate business users who are now able to write their own reports and workflows and manage their own enhancement requests.

In the example shown in figure 1, the left box shows the SQL used to calculate the average wait time between two specific events: 1001 and 1002. It requires the use of multiple select statements to extract the data for each event and create two result sets with the data correctly ordered so the main SQL statement can compute the average wait. The box on the right replaces the multiple select statements with a simple LAG function. This makes the code much cleaner, quicker to develop, simpler to test and it is much easier to understand.

	Multi-Select, Multi-Table Join Statement	Window Function based Statement
1	SELECT	SELECT
2	AVG(event_datetime - prior_event_datetime) AS	AVG(event_datetime - prior_event_datetime) AS avg_wait
3	avg_wait	FROM
4	FROM	(SELECT
5	(SELECT	new_event_cd as event_cd
6	a.*	, event_datetime
7	, rownum AS row_number	, LAG(new_event_cd) OVER (PARTITION BY order_id
8	FROM	ORDER BY event_datetime, new_event_cd)
9	(SELECT	AS prior_event_cd
10	order_id	, LAG(event_datetime) OVER (PARTITION BY order_id
11	, event_datetime AS prior_event_time	, LAG(new_event_cd) OVER (PARTITION BY order_id, new_event_cd)
12	, new_event_cd AS prior_event_cd	AS prior_event_datetime
13	FROM order_pipeline_events	FROM order_pipeline_events
14	WHERE warehouse_id = 'RNO1'	WHERE warehouse_id = 'RNO1'
15	ORDER BY order_id)
16	, prior_event_datetime	WHERE prior_event_cd = '1001'
17	, prior_event_cd	
18) a,	AND event_cd = '1002';
19) a,	
20	(SELECT	
21	b.*	
22	, rownum AS row_number	
23	FROM	
24	(SELECT	
25	order_id	
26	, event_datetime	
27	, new_event_cd AS event_cd	
28	FROM order_pipeline_events	
29	WHERE warehouse_id = 'RNO1'	
30	ORDER BY order_id	
31	, event_datetime	



Improved Manageability

Oracle's approach allows developers to efficiently layer their analysis using SQL because it can support a broad range of business requirements. For example, a requirement to select all the regions contributing more than 50% of total sales and where the salesperson is contributing more than 18% of regional sales can all be expressed in a single, simple SQL statement using analytical SQL:

Some of the key analytical SQL features are shown above in bold and these will be explored in more detail in the following sections of this whitepaper.

The amount of effort required to understand analytical SQL is minimized through the use of careful syntax design. Built-in analytical SQL constructs typically leverage existing SQL constructs, such as the aggregate functions SUM and AVG, to extend these well-understood keywords.

Most developers and business users have a reasonable level of proficiency with SQL and can quickly adopt and integrate analytical features, such as pareto-distributions, pattern matching, cube and rollup aggregations into their applications and reports. The amount of time required for enhancements, maintenance and upgrades is minimized: more people will be able to review and enhance the existing SQL code rather than having to rely on a few key people with specialized programming skills.

Most of Oracle's analytical SQL is part of the ANSI SQL standard. This ensures broad support for these features and rapid adoption of newly introduced functionality across applications and tools – both from Oracle's partner network and other independent software vendors. Oracle is continuously working with its many partners to assist them in exploiting the expanding library of analytic functions. Already many independent software vendors have integrated support for the new Database 12c in-database analytic functions into their products.

Oracle's in-database analytical SQL enables significantly better query performance. Not only does it remove the need for specialized data-processing silos but also the internal processing of these purpose-built functions is fully optimized. Using SQL unlocks the full potential of the Oracle database - such as parallel execution - to provide enterprise level scalability unmatched by external specialized processing engines.

Summary

This section has outlined how Oracle's in-database analytical SQL provides IT and application development teams with a robust and agile analytical development platform that enhances both query performance and developer productivity while providing investment protection by building on existing standards based skills.

The following sections will explore some of the key concepts behind Oracle's analytical SQL and provides an overview of the key analytical features and functions that use these key concepts.

In-database analytical SQL with Oracle Database 12c

This section outlines the high level processing concepts behind analytical SQL.

Processing concepts behind analytical SQL

Oracle's in-database analytical SQL – first introduced in Oracle Database 8i Release 1- has introduced several new elements to SQL processing. These elements build on existing SQL features to provide developers and business users with a framework that is both flexible and powerful in terms of its ability to support sophisticated calculations. There are four essential concepts used in the processing of Oracle's analytic SQL:

- Processing order
- Result set partitions
- Calculation windows
- Current Row

This four-step process is internally optimized and completely transparent but it does provide a high degree of flexibility in terms of being able to layer analytical features to create the desired result set without having to resort to long and complicated SQL statements. The following sections will explore these four concepts in more detail.

1. Processing order

Query processing using analytic SQL takes place in three stages:

Stage 1: All joins, WHERE, GROUP BY and HAVING clauses¹ are performed. Where customers are using Exadata storage servers the initial join and filtering operations for the query will be managed by the storage cells. This step can result in a significant reduction in the volume of data passed to the analytic function, which helps improve performance.

Stage 2: The result set is made available to the analytic function, and all the calculations are applied.

Stage 3: If the query has an ORDER BY clause then this is processed to allow for precise control of the final output.

2. Partitions – organizing your data sets

Analytic SQL allows users to divide query result sets into ordered groups of rows called "partitions"². Any aggregated results such as SUM's, AVG's etc. are available to the analytical functions. Partitions can be based upon any column(s) or expression. A query result set may have just one partition holding all the rows, a few large partitions, or many small partitions with each holding just a few rows.

¹ Please note that this applies mostly to a single query block

² Note that the term "partitions" used within the context of analytic functions is unrelated to Oracle Partitioning.

3. Calculation windows

Within each partition, a sliding window of data can be defined. The window determines the range of rows used to perform the calculations for the "current row" (defined in the next section). Window sizes can be based on either a physical number of rows or a logical interval such as time.

The window has a starting row and an ending row. Depending on its definition, the window may move at one or both ends.

For instance, a window defined for a cumulative sum function would have its starting row fixed at the first row of its window, and its ending row would slide from the starting point all the way to the last row of the window.

```
SELECT
  Qtrs
, Months
, Channels
, Revenue
, SUM(Revenue) OVER (PARTITION BY Qtrs) AS Qtr_Sales
, SUM(Revenue) OVER () AS Total_Sales
FROM sales_table
```

In contrast, a window defined for a moving average would have both its starting and end points slide so that they

QTRS	MONTHS	CHANNELS	REVENUE	QTR_SALES	TOTAL_SALES
1998-01	1998-01	Direct Sales	1630685.07	6480684	12074678.14
1998-01	1998-01	Internet	222086.08	6480684	12074678.14
1998-01	1998-01	Partners	424649.34	6480684	12074678.14
1998-01	1998-02	Direct Sales	1627680	6480684	12074678.14
1998-01	1998-02	Internet	267127.38	6480684	12074678.14
1998-01	1998-02	Partners	477883.49	6480684	12074678.14
1998-01	1998-03	Direct Sales	1257087.65	6480684	12074678.14
1998-01	1998-03	Internet	205005.52	6480684	12074678.14
1998-01	1998-03	Partners	315084.1	6480684	12074678.14
1998-01	1998-03	Tele Sales	53395.37	6480684	12074678.14
1998-02	1998-04	Direct Sales	1313683.91	5593994.14	12074678.14
1998-02	1998-04	Internet	223332.56	5593994.14	12074678.14
1998-02	1998-04	Partners	396249.81	5593994.14	12074678.14
1998-02	1998-04	Tele Sales	42712.02	5593994.14	12074678.14
1998-02	1998-05	Direct Sales	1228571.99	5593994.14	12074678.14
1998-02	1998-05	Internet	177134.61	5593994.14	12074678.14
1998-02	1998-05	Partners	321134.55	5593994.14	12074678.14
1998-02	1998-05	Tele Sales	21446.08	5593994.14	12074678.14
1998-02	1998-06	Direct Sales	1243938.66	5593994.14	12074678.14
1998-02	1998-06	Internet	232485.87	5593994.14	12074678.14
1998-02	1998-06	Partners	393304.08	5593994.14	12074678.14

FIGURE 2 – USING WINDOW FUNCTION TO CREATE DIFFERENT LEVELS OF AGGREGATION BASED ON TIME-BASED QUARTERS

maintained a constant physical or logical range. The example below demonstrates the use of physical window to create a 6-month moving average:

```
SELECT
  calendar_month_desc as "Month"
, amount_sold as "Sales"
, AVG(amount_sold) OVER(ORDER BY calendar_month_desc rows between 5 preceding
  and current row) as "6M_Avg_Sales"
FROM . . .
```

Alternatively, if the data set contains a date column then it is possible to use logical windows by taking advantage of Oracle's built-in time awareness. In this case we can use the "range interval '5' month preceding" syntax to create the 6-month moving average:

```

SELECT
calendar_month_desc as "Month"
, amount_sold as "Sales"
, AVG(amount_sold) OVER(ORDER BY calendar_month_desc range interval '5' month
preceding) as "6M_Avg_Sales"
FROM . . .

```

The output from these queries is shown here:

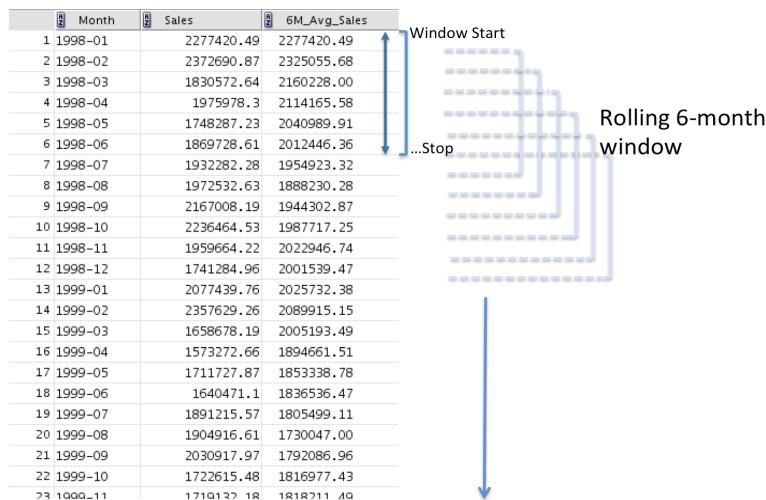


Figure 3 – An example showing the concept of rolling windows to compute a six month moving average

The concept of a window is very powerful and provides a lot of flexibility in terms of being able to interact with the data. A window can be set as large as all the rows in a partition. At the other extreme it could be just a single row. Users may specify a window containing a constant number of rows, or a window containing all rows where a column value is in a specified numeric range. Windows may also be defined to hold all rows where a date value falls within a certain time period, such as the prior month.

4. Current Row

Each calculation performed with an analytic function is based on a current row within a window. The current row serves as the reference point determining the start and end of the window. In the example below the calculation of a running total would be the result of the current row plus the values from the preceding two rows. At the end of the window the running total will be reset. The example shown below creates running totals within a result set showing the total sales for each channel within a product category within year:

```

SELECT
calendar_year
, prod_category_desc
, channel_desc
, country_name
, sales
, units
, SUM(sales) OVER (PARTITION BY calendar_year, prod_category_desc,
channel_desc order by country_name) sales_tot_cat_by_channel
FROM . . .

```

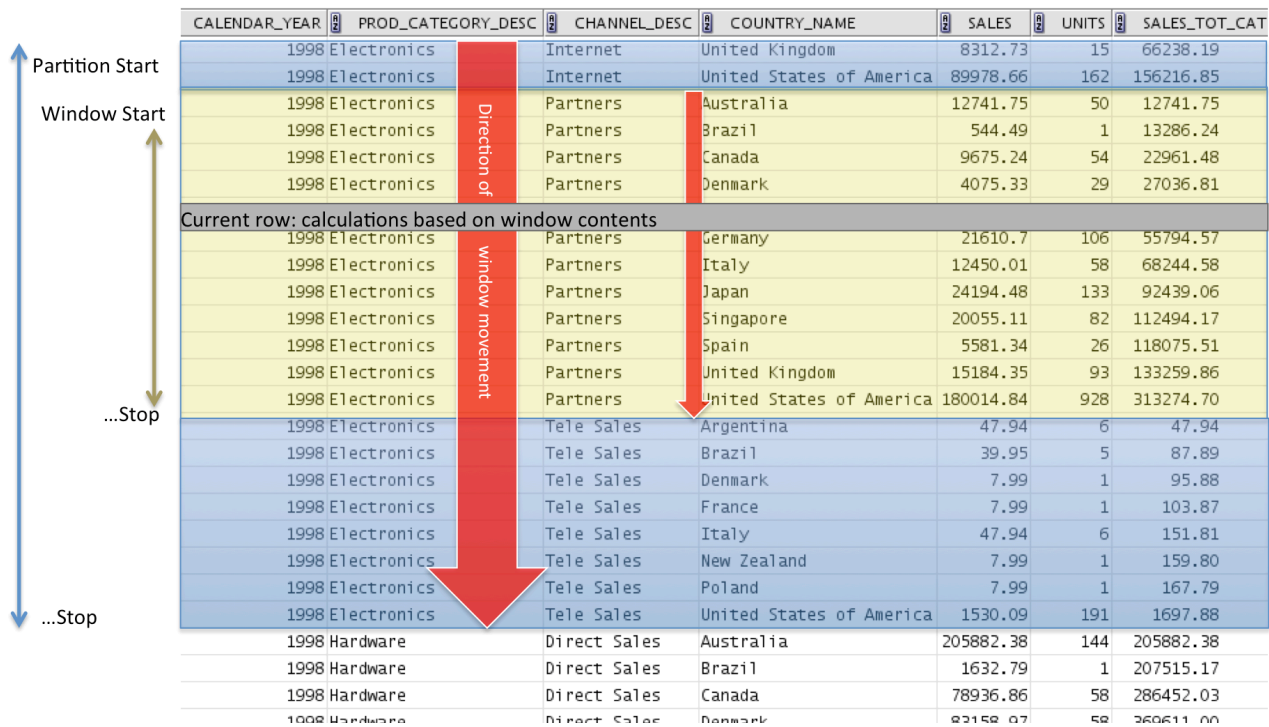



Figure 4 – An example showing the position of the current row within a partition based on rolling windows

Summary

This section outlined the four main processing concepts behind analytical SQL. The following sections provide an overview of the key analytical features and functions that use these concepts.

Overview of in-database analytical SQL functionality

There are many data warehouse and operational projects that would benefit from moving non-SQL based analytical code back inside the database and/or replacing complex SQL processing with simple SQL that leverages ANSI standard functions and features. The so-called workarounds that developers have used in the past, such as multiple self-joins and calls to external functions to deliver analytical calculations, can now be replaced with the following types in-database functions and features that are part of Oracle Database 12c:

- Enhanced reporting
- Data Sampling
- Advanced aggregations
- Statistical analysis
- Pattern matching
- Spreadsheet-like data modeling

The following sections will introduce these topics in more detail³.

Enhanced reporting

The following topics provide information about analytical SQL features and techniques that are linked to enhanced reporting. Although these topics are presented in terms of data warehousing, they are applicable to any activity needing analysis and reporting:

- Windowing, lag-lead and reporting aggregate functions
- Pivoting operations
- Data densification for reporting and time series calculations

Windowing, lag-lead and reporting aggregate functions

Windowing functions are used to compute moving and cumulative versions of SUM, AVERAGE, COUNT, MAX, MIN, and many more functions. They provide access to more than one row of a table and remove the need to use multiple self-joins. The functions can be used only in the SELECT and ORDER BY clauses of the query. Windowing aggregate functions include the term FIRST_VALUE, which returns the first value in the window; and the term LAST_VALUE, which returns the last value in the window. These two functions can only be used as analytic functions unlike FIRST and LAST which can both be used as aggregate functions and analytic functions.

The FIRST_VALUE and LAST_VALUE functions provide control over the way nulls are processed. If the IGNORE NULLS option is used with FIRST_VALUE, it returns the first non-null value in the set, or NULL if all values are NULL. If IGNORE NULLS is used with LAST_VALUE, it returns the last non-null value in the set, or NULL if all values are NULL. The example shown below demonstrates using the IGNORE NULLS option with FIRST_VALUE:

```
SELECT prod_id, channel_id, time_id,
       CASE WHEN MIN(amount_sold) > 9.5
            THEN MIN(amount_sold) ELSE NULL END amount_sold,
       FIRST_VALUE(CASE WHEN MIN(amount_sold) > 9.5
                      THEN min(amount_sold) ELSE NULL END)
              IGNORE NULLS OVER (PARTITION BY prod_id
                                ORDER BY channel_id DESC, time_id
                                ROWS BETWEEN UNBOUNDED PRECEDING
                                AND UNBOUNDED FOLLOWING) fv_nv
FROM sales
WHERE prod_id = 115 AND time_id BETWEEN '18-DEC-01' AND '22-DEC-01'
GROUP BY prod_id, channel_id, time_id
ORDER BY prod_id;
```

Which generates the following output:

PROD_ID	CHANNEL_ID	TIME_ID	AMOUNT_SOLD	FV_NV
115	4	18-DEC-01		9.66
115	4	19-DEC-01		9.66
115	4	20-DEC-01		9.66
115	4	22-DEC-01		9.66

³ Please consult the [Oracle Data Warehousing Guide](#) for an in-depth discussion of analytical SQL

115	3	18-DEC-01	9.66	9.66
115	3	19-DEC-01	9.66	9.66
115	3	20-DEC-01	9.66	9.66
115	3	21-DEC-01	9.66	9.66
115	3	22-DEC-01	9.66	9.66
115	2	18-DEC-01	9.67	9.66
115	2	19-DEC-01	9.67	9.66
115	2	21-DEC-01	9.67	9.66
115	2	22-DEC-01	9.67	9.66

13 rows selected.

In contrast, the `FIRST/LAST` aggregate functions allow a data set to be ranked and then continue to work with its top-ranked or bottom-ranked rows. After finding the top or bottom ranked rows, an aggregate function can be applied to any desired column. These analytical functions are valuable because they allow a developer to simplify their SQL by avoiding the need for a self-join or sub-query and this improves performance. The syntax for these functions begins with a regular aggregate function that produces a single return value per group which is then used to determine the ranking based on the `FIRST/LAST` functions and includes a new clause starting with the word `KEEP`.

The `LAG` and `LEAD` functions can be used to compare values within a data set when the relative positions of rows can be determined in advance. The `LAG` function accesses a row based on a given offset working backwards from the current position. The `LEAD` function does the same but looks beyond the current position. Both work by specifying the count of rows that separates the target row from the current row. These functions remove the need to use self-joins to fulfill this type of business requirement because they can access more than one row of a table at the same time. To ensure the most efficient performance, the Oracle Database has built-in optimizations to support this type of data processing.

Once the processing for a query has completed it is then possible to incorporate additional reporting aggregates, or measures, such as compute running totals or moving averages. These additional calculations allow business users to build sophisticated queries such as creating a report that lists each product category and the region in which it had maximum sales.

This is another great example of how using SQL as the language to deliver analytics allows developers to layer their SQL code with different functions while maintaining a level of simplicity that makes maintenance, support and enhancements much easier to manage compared to using bespoke and complex code (some of which might have to exist outside the database).

Pivoting operations

Pivoting is a key technique in many data warehouse related applications. Business intelligence queries typically transform multiple input rows into result sets that contain fewer and generally wider rows. The data returned by business intelligence queries is often most usable if presented in a cross-tabular format. The pivot clause of the `SELECT` statement lets application developers write cross-tabular queries that rotate rows into columns, aggregating data as part of the rotation process. Prior to this feature BI tools and report developers were forced to use self-joins as a workaround, which resulted in complex code that was extremely difficult to optimize.

Many ETL and data mining workflows include pivot operations within the transformation step. Pushing this type of processing back inside the database and executing it using native SQL will significantly increase the performance of workflows that require this type of feature by removing the need to deploy staging tables to manage the row-to-column or column-to-row transformation.

If we need to convert dimension values (rows) into columns to split out a specific set of data points then we can use the `PIVOT` clause to transform the result set from this:

	ORDER_DATE	ORDER_MODE	ORDER_TOTAL
9	02-07-2007	direct	600
10	02-09-2007	direct	6394.8
11	02-09-2007	direct	62303
12	02-10-2007	direct	6653.4
13	02-10-2007	direct	29669.9
14	04-06-2007	online	282694.3
15	04-06-2008	online	17620
16	04-10-2007	direct	129
17	06-10-2007	direct	12589
18	06-12-2007	online	21116.9
19	07-06-2007	online	38017.8
20	07-06-2007	online	11006.2
21	07-11-2006	direct	3878.4
22	08-01-2006	online	5543.1
23	08-01-2006	direct	59872.4
24	08-01-2008	direct	7826

to this:

```
SELECT * FROM
(SELECT EXTRACT(YEAR FROM order_date) year,
      order_mode,
      order_total
      FROM orders)
PIVOT
(SUM(order_total) FOR order_mode IN
('direct' AS Store, 'online' AS Internet));
```

	YEAR	STORE	INTERNET
1	2006	371895.5	100056.6
2	2007	1274078.8	1271019.5
3	2004	5546.6	(null)
4	2008	252108.3	393349.4

The UNPIVOT clause lets you rotate the data the other way – i.e. it is the reverse of pivot. For example:

```
SELECT * FROM pivot_table
UNPIVOT (yearly_total FOR order_mode IN (store AS 'direct',
      internet AS 'online'))
ORDER BY year, order_mode;
```

YEAR	ORDER_	YEARLY_TOTAL
2004	direct	5546.6
2006	direct	371895.5
2006	online	100056.6
2007	direct	1274078.8
2007	online	1271019.5
2008	direct	252108.3
2008	online	393349.4

Data densification for reporting and time series calculations

Data is normally stored in sparse form. That is, if no value exists for a given combination of dimension values, no row exists in the fact table. However, in some cases it may be necessary to view the data in a dense form, with rows for all combinations of dimension values displayed even when no data exists for specific data points. Time series calculations can be performed most easily when data is dense along the time dimension. This is because dense data will fill a consistent number of rows for each period, which in turn makes it easier to use the analytic windowing functions with physical offsets.


Data densification is the process of converting sparse data into dense form. To overcome the problem of data sparsity, Oracle provides the partitioned outer join to fill the gaps in a time series or any other dimension. Such a join extends the conventional outer join syntax by applying the outer join to each logical partition defined in a query. Oracle logically partitions the rows in the query based on the expression specified in the PARTITION BY clause. The result of a partitioned outer join is a UNION of the outer joins of each of the partitions in the logically partitioned table with the table on the other side of the join.

The result is a much richer data set that is easier to use in terms of applying analytical functions such as moving averages and moving totals.

Data sampling

Some business intelligence reports and advanced analytical processing often need to limit the number of rows returned by a query either by asking for a specific number of rows (for example top 5 and/or bottom 10) or a percentage of rows (for example 25% of all rows). This type of query is often called a Top-N query and it offers a straightforward way of creating reports that show "Who are the ten most profitable customers?". It is also useful for user interfaces that provide the first few rows of a data set for browsing.

This type of requirement highlights the key benefits of switching to Oracle's SQL analytical functions. To date many developers, report writers and software vendors have satisfied this type requirement by using the ROW_NUMBER window function. The SQL code needed is not overly complicated but by using the ANSI SQL standard syntax, the new "row limiting" clause that appears as the last part of a SELECT, the SQL statement is greatly simplified and easier to understand as shown in the following example:

<pre>SELECT empno, ename, deptno FROM (SELECT empno, ename, deptno, sal, comm, row_number() OVER (ORDER BY sal,comm) rn FROM emp) WHERE rn <=5 ORDER BY sal, comm;</pre>		<pre>SELECT empno, ename, deptno FROM emp ORDER BY sal, comm FETCH FIRST 5 ROWS ONLY;</pre>
--	---	---

Advanced SQL aggregations

Oracle has extended the processing capabilities of the GROUP BY clause to provide fine-grained control over the creation of totals derived from the initial result set. This includes the following features:

- Rollup – calculates multiple levels of subtotals across a specified group of dimensions

- Cube - calculates subtotals for all possible combinations of a group of dimensions and it calculates a grand total
- Grouping – helps identify which rows in a result set have been generated by a rollup or cube operation
- Grouping sets - is a set of user defined groupings that are generated as part of the result set

ROLLUP creates subtotals that "roll up" from the most detailed level to a grand total, following a grouping list specified in the ROLLUP clause. ROLLUP takes as its argument an ordered list of grouping columns.

ROLLUP is very helpful for subtotaling along a hierarchical dimension such as time or geography and it simplifies and speeds the population and maintenance of summary tables. This is especially useful for ETL developers and DBAs.

CUBE can calculate a cross-tabular report with a single SELECT statement. Like ROLLUP, CUBE is a simple extension to the GROUP BY clause, and its syntax is also easy to learn. CUBE takes a specified set of grouping columns and creates the required subtotals for all possible combinations. This feature is very useful in situations where summary tables need to be created. CUBE adds most value to query processing where the query is based on columns from multiple dimensions rather than columns representing different levels of a single dimension.

While ROLLUP and CUBE are very powerful features they can seem a little inflexible. Developers often need to determine which result set rows are subtotals and the exact level of aggregation for a given subtotal. This allows them to use subtotals in calculations such as percent-of-totals.

To help resolve data quality issues it is often important to differentiate between stored NULL values and "NULL" values created by a ROLLUP or CUBE. The GROUPING function resolves these problems. Using a single column as its argument, GROUPING returns 1 when it encounters a NULL value created by a ROLLUP or CUBE operation. That is, if the NULL indicates the row is a subtotal, GROUPING returns a 1. Any other type of value, including a stored NULL, returns a 0. Even though this information is only of value to the developer it is a very powerful feature. It is not only useful for identifying NULLs, it also enables sorting subtotal rows and filtering results.

Grouping sets allow the developer and business user to precisely define the groupings of key dimensions. It produces a single result set which is equivalent to a UNION ALL of differently grouped rows. This allows efficient analysis across multiple dimensions without computing the whole CUBE. Since computing all the possible permutations for a full CUBE creates a heavy processing load, the precise control enabled by grouping sets translates into significant performance gains.

For example, consider the following statement:

```
SELECT
channel_desc
, calendar_month_desc, country_iso_code,
TO_CHAR(SUM(amount_sold), '9,999,999,999') SALES$
FROM sales, customers, times, channels, countries
WHERE sales.time_id=times.time_id
AND sales.cust_id=customers.cust_id
AND sales.channel_id= channels.channel_id
AND channels.channel_desc IN ('Direct Sales', 'Internet')
AND times.calendar_month_desc IN ('2000-09', '2000-10')
AND country_iso_code IN ('GB', 'US')
GROUP BY GROUPING SETS(
(channel_desc, calendar_month_desc, country_iso_code)
,(channel_desc, country_iso_code)
,(calendar_month_desc, country_iso_code));
```

The above statement calculates aggregates over the following three groupings:

```
(channel_desc, calendar_month_desc, country_iso_code)
(channel_desc, country_iso_code)
(calendar_month_desc, country_iso_code)
```

Now compare the above statement to the following statement that uses CUBE and GROUPING ID to create the same result set:

```
SELECT
    channel_desc
  , calendar_month_desc
  , country_iso_code,
  TO_CHAR(SUM(amount_sold), '9,999,999,999') SALES$,
  GROUPING_ID(channel_desc, calendar_month_desc, country_iso_code) gid
FROM sales, customers, times, channels, countries
WHERE sales.time_id=times.time_id
AND sales.cust_id=customers.cust_id
AND sales.channel_id= channels.channel_id
AND channels.channel_desc IN('Direct Sales', 'Internet')
AND times.calendar_month_desc IN('2000-09', '2000-10')
AND country_iso_code IN ('GB', 'US')
GROUP BY CUBE(channel_desc, calendar_month_desc, country_iso_code)
HAVING GROUPING_ID(channel_desc, calendar_month_desc, country_iso_code)=0
      OR GROUPING_ID(channel_desc, calendar_month_desc, country_iso_code)=2
      OR GROUPING_ID(channel_desc, calendar_month_desc, country_iso_code)=4;
```

This statement computes all the 8 (2 *2 *2) groupings, though only three of these groupings are actually required.

Concatenated groupings offer a concise way to generate useful combinations of groupings. Groupings specified with concatenated groupings yield the cross product of groupings from each grouping set. Developers can use this feature to specify a small number of concatenated groupings, which in turn actually generates a large number of final groups. This helps to both simplify and reduce the length and of the SQL statement making it easier to understand and maintain. Concatenated groupings are specified by listing multiple grouping sets, cubes, and rollups, and separating them with commas. The example below contains concatenated grouping sets:

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

which defines the following groupings:

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

Concatenation of grouping sets is very helpful for a number of reasons. Firstly, it reduces the complexity of query development because there is no need to enumerate all groupings within the SQL statement. Secondly, it allows application developers to push more processing back inside the Oracle Database. The SQL typically generated by OLAP-type applications often involves the concatenation of grouping sets, with each grouping set defining groupings needed for a dimension.

Composite columns

A composite column is a collection of columns that are treated as a unit during the computation of groupings. In general, composite columns are useful in ROLLUP, CUBE, GROUPING SETS, and concatenated groupings. For example, in CUBE or ROLLUP, composite columns would mean skipping aggregation across certain levels.

You specify the columns in parentheses as in the following statement:

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

In this statement, the data is not rolled up across year and quarter. What is actually produced is equivalent to the following groupings of a UNION ALL:

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

There is more information about advanced SQL aggregations in the Oracle SQL Language Reference documentation⁴.

Oracle Database 12c (12.1.0.2) has introduced a key new feature for aggregation and data discovery to support the requirements for approximate answers. In some cases the level of precision needed within an analytical query can be reduced – i.e. good enough is in fact the perfect answer. An approximate answer that is for example within 1% of the actual value can be sufficient.

Oracle has added this feature by implementing HyperLogLog algorithms in SQL for 'approximate count distinct' operations. Processing of large volumes of data is significantly faster than the exact aggregation, especially for data sets with a large number of distinct values, with negligible deviation from the exact result.

The following statement returns the approximate number of distinct customers for each product:

```
SELECT
  prod_id,
  APPROX_COUNT_DISTINCT(cust_id) AS "Number of Customers"
FROM sales
GROUP BY prod_id
ORDER BY prod_id;
```

producing the following output:

PROD_ID	Number of Customers
13	2516
14	2030
15	2105
16	2367
17	2093
18	2975
19	2630
20	3791

APPROX_COUNT_DISTINCT processes large amounts of data significantly faster than COUNT, with negligible deviation from the exact result. There is more information about this new feature in the Oracle SQL Language Reference documentation⁵.

Statistical analysis

Many data warehouse and operational projects start with an evaluation of the available data to determine if the business requirements, or hypothesis, are feasible and which data points are likely to provide the most valuable information. Data exploration and evaluation requires the use of mathematical features such as: percentile distributions, standard deviation, linear regression and many other statistical tests.

⁴ [HTTP://DOCS.ORACLE.COM/CD/E16655_01/SERVER.121/E17209/FUNCTIONS186.HTM#I1279931](http://docs.oracle.com/cd/E16655_01/SERVER.121/E17209/FUNCTIONS186.HTM#I1279931)

⁵ [HTTP://DOCS.ORACLE.COM/DATABASE/121/SQLRF/FUNCTIONS013.HTM#SQLRF56900](http://docs.oracle.com/database/121/SQLRF/FUNCTIONS013.HTM#SQLRF56900)

In the past projects have typically delivered these requirements through the use of external specialized mathematical engines. The Oracle Database makes it possible to push this type of exploration and analysis back inside the database. Oracle provides a rich library of statistical functions including support for:

- Linear algebra
- Linear regression
- Statistical aggregates for hypothesis tests

Linear algebra

Linear algebra has a wide range of practical applications such as analysis of microarray data in bioinformatics, operations research in supply chains, analysis of consumer demand data in econometrics, asset allocation problems in finance and other general optimization problems.

Performing matrix manipulation in PL/SQL in the past required inventing a matrix representation based on PL/SQL's native data types and then writing bespoke data manipulation routines. This required substantial programming effort and the performance of the resulting implementation was challenging to optimize. Many developers have opted to use external packages to support linear algebra use cases. This resulted in data being pushed to external packages for processing and then pulled back into the Oracle Database for further analysis using SQL. This back-and-forth data transfer can be very time consuming.

In Oracle Database 12c the `UTL_NLA` package exposes matrix data types and wrapper subprograms within PL/SQL to help developers build linear algebra models. This removes the need to move data out the Oracle Database. It allows developers to easily integrate results with other types of analysis and maintains the required layers of data security

Retailers often use linear algebra to analyze the effectiveness of their marketing programs. For example, individual stores will typically allocate their marketing budget over a number of possible programs such as: media advertisements, promotions, discount coupons and direct mailers. Using regression analysis it is possible to build a linear relationship between the amount of sales that an average store has in a given year and the spending on the four components of the marketing program.

A sales-marketing linear model for the above would look like this:

```
Sales Revenue = a + b Media Advisements
                + c Promotions
                + d Discount Coupons
                + e Direct Mailer
```

This model can be implemented as the following view, which refers to the OLS regression object⁶:

```
CREATE OR REPLACE VIEW sales_marketing_model (year, ols)
AS SELECT year,
        OLS_Regression(
            /* mean_y => */ AVG(sales),
            /* variance_y => */ var_pop(sales),
            /* MV mean vector => */
            UTL_NLA_ARRAY_DBL (AVG(media),AVG(promo),
```

⁶ Please note that a multiple linear regression application built using the `UTL_NLA` package is implemented in an object called `OLS_Regression`. Note that sample files for the OLS Regression object can be found in `$ORACLE_HOME/plsql/demo`.

```

                                AVG(disct),AVG(dmail)),
/* VCM variance covariance matrix => */
                                UTL_NLA_ARRAY_DBL (var_pop(media),covar_pop(media,promo),
covar_pop(media,disct),covar_pop(media,dmail),
                                var_pop(promo),covar_pop(promo,disct),
                                covar_pop(promo,dmail),var_pop(disct),
                                covar_pop(disct,dmail),var_pop(dmail)),
/* CV covariance vector => */
                                UTL_NLA_ARRAY_DBL
(covar_pop(sales,media),covar_pop(sales,promo),
                                covar_pop(sales,disct),covar_pop(sales,dmail)))
FROM sales_marketing_data
GROUP BY year;

```

Using this view, a marketing program manager can perform an analysis such as "Is this sales-marketing model reasonable for year 2012 data?" This question would translate to: "is the multiple-correlation greater than some acceptable value, say, 0.9". The SQL for such a query might be as follows:

```

SELECT model.ols.getCorrelation(1)
       AS "Applicability of Linear Model"
FROM sales_marketing_model model
WHERE year = 2012;

```

Using this approach it is then possible to answer additional questions such as:

- What is the expected base-line sales revenue of a store without any marketing programs in 2012?
- Which component of the marketing program was the most effective in 2012? That is, a dollar increase in which program produced the greatest expected increase in sales?

Linear regression

Linear regression can be used in a number of different ways depending on the type of analysis required. For example, one approach is to use it to predict a missing value or forecast a future data point. Linear regression does this by fitting a predictive model to a set of data points. The resulting model can then be used to determine the value of a missing data point given that other data point(s) exist. This "fitted" model can also be used to make predictions regarding future data points.

For example it is possible to compute an ordinary-least-squares regression line that expresses the quantity sold of a product as a linear function of the product's list price. In the code sample listed below the calculations are grouped by sales channel. The values SLOPE, INTCPT, RSQR in the result set are: slope, intercept, and coefficient of determination of the regression line, respectively. The value COUNT is the number of products in each channel for which both quantity sold and list price data is available:

```

SELECT
  s.channel_id
, REGR_SLOPE(s.quantity_sold, p.prod_list_price) SLOPE
, REGR_INTERCEPT(s.quantity_sold, p.prod_list_price) INTCPT
, REGR_R2(s.quantity_sold, p.prod_list_price) RSQR
, REGR_COUNT(s.quantity_sold, p.prod_list_price) COUNT
, REGR_AVGX(s.quantity_sold, p.prod_list_price) AVGLISTP
, REGR_AVGY(s.quantity_sold, p.prod_list_price) AVQSOLD
FROM sales s, products p

```

```

WHERE s.prod_id=p.prod_id
AND p.prod_category='Electronics'
AND s.time_id=to_DATE('10-OCT-2000')
GROUP BY s.channel_id;

```

CHANNEL_ID	SLOPE	INTCPT	RSQR	COUNT	AVGLISTP	AVGQSOLD
2	0	1	1	39	466.656667	1
3	0	1	1	60	459.99	1
4	0	1	1	19	526.305789	1

Another use case for linear regression is that given a specific column of data along with a series of possibly related columns of data it is possible to quantify the strength of the relationship between the data columns to determine where there are columns that may have no relationship with the specific column and/or to isolate columns that may in fact contain redundant information. This can be used to help simplify analysis by removing unwanted data columns from reports and/or data mining models.

Hypothesis testing

The use of hypothesis testing by management teams, along with other complex statistical analysis, is growing rapidly. One of the most often quoted works in this area is the analysis done by Glebbeek and Bax⁷ in 2004. They tested the hypothesis that the relationship between employee turnover and firm performance was based on an inverted U-shaped relationship. This implied that both excessively high and low staff turnover was harmful to overall business performance. The optimum level of staff turnover was estimated as being somewhere in the middle of these two extremes. The research found that at the best possible level of performance and office would expect to have a 6% annual turnover of staff with anything above or below this figure likely to lead to poorer performance. This specific type of analysis can be very useful for HR managers.

Oracle provides statistical functions to support the following types of hypothesis testing:

- Hypothesis Testing - Parametric Tests
- Crosstab statistics
- Hypothesis Testing - Non-Parametric Tests
- Non-Parametric Correlation

Included in the release of 12c was the PL/SQL package DBMS_STAT_FUNCS. It contains the descriptive statistical function SUMMARY along with functions to support distribution fitting. The SUMMARY function summarizes a numerical column of a table with a variety of descriptive statistics. The five distribution fitting functions support normal, uniform, Weibull, Poisson, and exponential distributions.

The following example determines the significance of the difference between the average sales to men and women where the hypothesis could be that the distributions of average sales are assumed to have similar (pooled) variances (note the use of the ROLLUP command within the GROUP BY clause):

```

SELECT SUBSTR(cust_income_level, 1, 22) income_level,
       AVG(DECODE(cust_gender, 'M', amount_sold, null)) sold_to_men,
       AVG(DECODE(cust_gender, 'F', amount_sold, null)) sold_to_women,

```

⁷ [Is High Employee Turnover Really Harmful? An Empirical Test Using Company Records](#), Arie C. Glebbeek¹ and Erik H. Bax¹

```

        STATS_T_TEST_INDEP(cust_gender, amount_sold, 'STATISTIC', 'F')
t_observed,
        STATS_T_TEST_INDEP(cust_gender, amount_sold) two_sided_p_value
FROM sh.customers c, sh.sales s
WHERE c.cust_id = s.cust_id
GROUP BY ROLLUP(cust_income_level)
ORDER BY income_level, sold_to_men, sold_to_women, t_observed;

```

The output from this command is shown below. The calculated value “p-value” is the probability that the null hypothesis - the amount purchased by men is equal to the amount purchased by women - can be rejected. The figures in the table below that are highlighted in red show those rows that satisfy that criterion (< .05—95% confidence level):

INCOME_LEVEL	SOLD_TO_MEN	SOLD_TO_WOMEN	T_OBSERVED	TWO_SIDED_P_VALUE
A: Below 30,000	105.28349	99.4281447	-1.9880629	.046811482
B: 30,000 - 49,999	102.59651	109.829642	3.04330875	.002341053
C: 50,000 - 69,999	105.627588	110.127931	2.36148671	.018204221
D: 70,000 - 89,999	106.630299	110.47287	2.28496443	.022316997
E: 90,000 - 109,999	103.396741	101.610416	-1.2544577	.209677823
F: 110,000 - 129,999	106.76476	105.981312	-.60444998	.545545304
G: 130,000 - 149,999	108.877532	107.31377	-.85298245	.393671218
H: 150,000 - 169,999	110.987258	107.152191	-1.9062363	.056622983
I: 170,000 - 189,999	102.808238	107.43556	2.18477851	.028908566
J: 190,000 - 249,999	108.040564	115.343356	2.58313425	.009794516
K: 250,000 - 299,999	112.377993	108.196097	-1.4107871	.158316973
L: 300,000 and above	120.970235	112.216342	-2.0642868	.039003862
	107.121845	113.80441	.686144393	.492670059
	106.663769	107.276386	1.08013499	.280082357

14 rows selected.

There is more information about these statistical features in the Oracle SQL Language Reference documentation⁸.

Pattern matching

Oracle now provides a completely new native SQL syntax for pattern matching. It has adopted the regular expression capabilities of Perl by implementing a core set of rules to define patterns in sequences (streams of rows) using SQL. This new inter-row pattern search capability complements the already existing capabilities of regular expressions that match patterns within character strings of a single record. The 12c `MATCH_RECOGNIZE` feature allows the definition of patterns, in terms of characters or sets of characters, and provides the ability to search for those patterns across row boundaries.

The `MATCH_RECOGNIZE` clause builds on the syntax used in other analytical SQL functions and features reusing many of the basic concepts such as `PARTITION BY` and `ORDER BY`. The `MATCH_RECOGNIZE` clause uses these concepts to logically partition and order the data stream ready for analysis. As can be seen in the code extracts used later in this section the syntax for this new clause is very rich and comprehensive yet, because it reuses existing concepts, it is easy to understand.

The `PATTERN` clause of the `MATCH_RECOGNIZE` construct defines the patterns, which need to be matched within a stream of records and it is based on regular expression syntax. Each pattern variable is then described in the `DEFINE` clause using SQL-like syntax to identify individual rows or inter-row changes in behavior (events). The

⁸ http://docs.oracle.com/cd/E16655_01/server.121/e17209/functions186.htm#i1279931

structure used to define the patterns will be well known to developers who are familiar with regular expression declarative languages such as PERL.

There are four basic steps for building a `MATCH_RECOGNIZE` clause:

- Define the partitions and ordering needed to identify the pattern encapsulated within the ‘streams of events’
- Define the pattern of events and pattern variables identifying the individual events within the overall pattern
- Define measures (outputs): source data points, pattern data points, calculations and aggregates related to a pattern
- Determine how the output will be generated

Pattern matching can help enterprises derive significant value from understanding how their customers use their website. This classic use case is often referred to as analysis of user “sessionization” and it can relate to website clickstream data. Within the telco industry a similar approach is used with CDR data to measure call quality. Pattern matching can help business teams define service offerings and enhancements, differential pricing strategies, marketing campaigns and more. The example below is an example of using pattern matching to examine sessionization relating to web site clickstreams.

Using pattern matching - sessionization

Within a raw clickstream data set the goal is to detect the sessions, assign a session ID to each session, and to display each input row with its session ID. The input data comes from the web server system log that tracks all page requests. Each row in the data set is the event of a user requesting a page. These logs show when a user requested a given page but there is no indication of when the user stopped looking at the page. Using pattern matching it is possible to determine how long the person spent on each page by comparing the timestamps between events.

The pattern-matching query listed below will display each input row with its `Session_ID`. In this case, events are considered to be part of the same session if they are ten or fewer time units apart. That session threshold is expressed in the `DEFINE` clause for pattern variables and is highlighted in bold:

```
SELECT time_stamp
       , user_id
       , session_id
FROM Events MATCH_RECOGNIZE
(PARTITION BY User_ID
 ORDER BY Time_Stamp
 MEASURES match_number() AS session_id
 ALL ROWS PER MATCH
 PATTERN (b s*)
 DEFINE
       s AS (s.Time_Stamp - prev(Time_Stamp) <= 10))
ORDER BY user_id, time_stamp;
```

For the purposes of this example a session is defined as a sequence of one or more time-ordered rows with the same partition key. If the rows have a timestamp greater than ten units apart, they are considered to be in different sessions⁹.

⁹ Note that the 10-unit time threshold used here is an arbitrary number: each real-world case requires the analyst's judgment to determine the most suitable threshold time gap. Historically, a 30-minute gap has been a commonly used threshold for separating sessions of website visits.

The output from this query would be as follows:

TIME_STAMP	USER_ID	SESSION_ID
1	Mary	1
11	Mary	1
23	Mary	2
34	Mary	3
44	Mary	3
53	Mary	3
63	Mary	3
3	Richard	1
13	Richard	1
23	Richard	1
33	Richard	1
43	Richard	1
2	Sam	1
12	Sam	1
22	Sam	1
.	.	.

This is the first step in our process of analyzing sessionization data. The query has now assigned session numbers to the detail-level rows. The next step in our analysis is to aggregate these results by session. The next code example aggregates the data to give one row per session. The output shows the session id, the user id, the number of aggregated events per session and total session duration. This makes it easy to see how many clicks each user has made per session and how long each session lasts.

```
SELECT session_id
, user_id
, start_time
, no_of_events
, duration
FROM Events MATCH_RECOGNIZE
(PARTITION BY User_ID
ORDER BY Time_Stamp
MEASURES MATCH_NUMBER() session_id,
COUNT(*) AS no_of_events,
FIRST(time_stamp) start_time,
LAST(time_stamp) - FIRST(time_stamp) duration
PATTERN (b s*)
DEFINE
s AS (s.Time_Stamp - PREV(Time_Stamp) <= 10) )
ORDER BY user_id, session_id;
```

Within the MEASURE syntax it is possible to include standard SQL functions and features such as aggregate functions. In addition the MATCH_RECOGNIZE clause includes two built-in functions that explain how the pattern matching criteria have been applied. The first of these built-in features, MATCH_NUMBER, is used above.

The MATCH_NUMBER function identifies matches within a row pattern partition. Each match is numbered sequentially starting with 1 in the order they are found. Note that match numbering starts over again at 1 in each row pattern partition, because there is no inherent ordering between row pattern partitions.

Along with knowing which MATCH_NUMBER has been assigned, it is possible to determine which component of a pattern applies to a specific row. This is done using the CLASSIFIER function. The classifier of a row is the pattern variable that the row is mapped to by a row pattern match. The CLASSIFIER function returns a character string whose value is the name of the variable the row is mapped to and in this example the classifier function would

simply return a single value of "s". This is because we have only defined one event – s as (s.Time_Stamp – PREV(Time_Stamp) <= 10).

The above query only uses the first of these two built-in functions and its output is listed in column 1 of the output shown below:

SESSION_ID	USER_ID	START_TIME	NO_OF_EVENTS	DURATION
1	Mary	1	2	10
2	Mary	23	1	0
3	Mary	34	4	29
1	Richard	3	5	40
2	Richard	54	2	9
1	Sam	2	4	30
2	Sam	43	3	5
3	Sam	59	3	9

8 rows selected.

This data set can now be used to drive many other types of analyses such as maximum, minimum, and average session duration.

The general syntax for SQL pattern matching is very rich which means it is easy to define complex patterns. The above example only provides an overview of the most basic elements of this functionality. More information is available in the Data Warehousing Guide and the SQL Language Reference Guide for Oracle Database 12c which is available via the [OTN website](http://www.oracle.com/technetwork/indexes/documentation/index.html)¹⁰. There is an Oracle-by-Example tutorial available on the Oracle Learning Library¹¹.

Spreadsheet-like data modeling and what-if analysis

Many business users like the flexibility provided by spreadsheets. However, these tools have severe limitations in terms of the amount of data they can manage. What business users really want is a solution that combines access to the huge pools of data inside their data warehouse with the flexibility of analysis offered by traditional spreadsheets. Oracle can now offer business users a SQL based solution: the MODEL clause. Oracle's vision is that for many applications the MODEL clause will replace the need to use desktop spreadsheets.

The MODEL clause allows business users to create a sophisticated multidimensional array from query results and then apply business calculations (rules) to this array to derive new calculated values. These rules can range from simple arithmetic operations to more complex and sophisticated requirements such as recursive simultaneous equations. The MODEL clause leverages the Oracle Database's core strengths in terms of scalability, manageability, collaboration, and security. The core query engine can work with unlimited quantities of data. By defining and executing models within the database, users avoid transferring large data sets to and from separate modeling environments. Models can be shared easily across workgroups, ensuring that calculations are consistent for all applications. Access can also be controlled precisely with Oracle's security features. With its rich functionality, the MODEL clause can enhance all types of applications.

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

¹⁰ Link to Database documentation library: <http://www.oracle.com/technetwork/indexes/documentation/index.html>

¹¹ http://www.oracle.com/webfolder/technetwork/tutorials/obe/db/12c/r1/misc/sql_pattern/sql_pattern.html

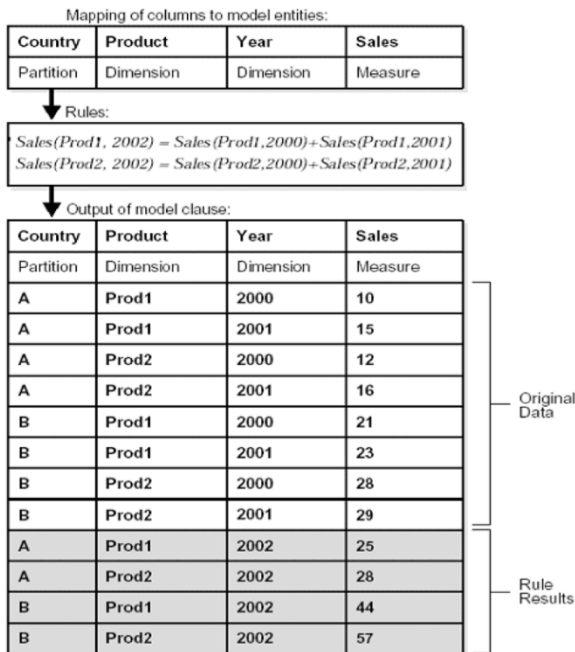


FIGURE 5 – USING THE SQL MODEL CLAUSE

1) Partitions define logical blocks within the result set in a way similar to the partitions of the analytical functions (see section on partitions in the section Processing concepts behind analytical SQL). Each partition is viewed by the formulae as an independent array.

2) Dimensions identify each measure cell within a partition. These identifying characteristics include data points such as date, region and product name.

3) Measures are equivalent 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.

Figure 5 shows how these three concepts would be applied. The diagram is broken down into three parts:

Stage 1: maps the columns to model entities. This divides the source table into three components: partition, dimension and measure columns.

Stage 2: defines the required business rules that define the new data points. Finally, the last stage creates a unified output by applying the rules to the source table to create a unified data set containing the original data and the hypothetical data derived from the business rules.

Stage 3: merge data with calculated rows - the unshaded area in Figure 5 shows the output block and contains the data retrieved from the database, while the shaded output shows rows calculated using the rules (note that the rules are applied within each partition).

The SQL Model clause is able to support very sophisticated calculations. The SQL query below shows a typical example of how to use the SQL model clause:

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

The above query partitions the source data, the table sales_view, on country so that the model computation, as defined by the three RULES, is performed on each country. The model calculates the sales of "Bounce" in "2002" as

the sum of its sales in “2000” and “2001” and sets the sales for “Y Box” in “2002” to the same value as they were in “2001”. In addition it introduces a new product category called “All_Products” for year “2002” to be the sum of sales of “Bounce” and “Y Box” for that year.

The output from the above query is shown below and the bold text indicates new values:

COUNTRY	PRODUCT	YEAR	SALES
Italy	Bounce	1999	2474.78
Italy	Bounce	2000	4333.69
Italy	Bounce	2001	4846.3
Italy	Bounce	2002	9179.99
...			
Italy	Y Box	1999	15215.16
Italy	Y Box	2000	29322.89
Italy	Y Box	2001	81207.55
Italy	Y Box	2002	81207.55
...			
Italy	All_Products	2002	90387.54

For more information please refer to the Oracle whitepaper “The SQL Model Clause of Oracle Database”¹². There is an Oracle-by-Example tutorial available on the Oracle Learning Library¹³.

The Model clause treats relational data as a multidimensional array in which every cell is accessible through a concise, flexible notation. This means that complex SQL joins and unions are eliminated and processing is optimized - Oracle’s parallel query processing powers are exploited by the Model clause and this allows applications to benefit from enterprise level scalability. The Model clause automatically handles logical dependencies among formulas and further simplifies the development and maintenance of sophisticated business calculations.

Conclusion

Oracle’s analytical SQL provides business users and SQL developers with a simplified way to support the most important operational and business intelligence reporting requirements. By moving processing inside the database developers can benefit from increased productivity and business users can benefit from improved query performance across a broad range business calculations.


The use of Oracle’s in-database analytical features delivers the following benefits to IT teams and business users:

- Increased developer productivity
- Minimizes learning effort
- Improves manageability
- Provides investment protection (adheres to industry standards based syntax)
- Delivers increased query speed

The flexibility and power of Oracle’s analytic functions, combined with their adherence to international SQL standards, makes them an important tool for all SQL users: DBAs, application developers, data warehouse

¹² [HTTP://WWW.ORACLE.COM/TECHNETWORK/MIDDLEWARE/BI-FOUNDATION/10GR1-TWP-BI-DW-SQLMODEL-131067.PDF](http://www.oracle.com/technetwork/middleware/bi-foundation/10GR1-TWP-BI-DW-SQLMODEL-131067.PDF)

¹³ [HTTP://WWW.ORACLE.COM/WEBFOLDER/TECHNETWORK/TUTORIALS/OBE/DB/10G/R2/PROD/BIDW/SQLMODEL/SQLMODEL_OTN.HTM](http://www.oracle.com/webfolder/technetwork/tutorials/obe/db/10g/r2/prod/bidw/sqlmodel/sqlmodel_otn.htm)



developers and business users. In addition, many business intelligence tool vendors have recognized the importance of these features and functions by incorporating support for them directly in to their products.

Overall, the SQL analytic functions and features in Oracle Database 12c make it the most effective platform for delivering analytical results directly into operational, data warehousing and business intelligence projects.

Further Reading

The following Oracle Database features are referenced in the text:

1. [Database SQL Language Reference - Oracle and Standard SQL](#)
2. [Oracle Analytical SQL Features and Functions](#) - a compelling array of analytical features and functions that are accessible through SQL
3. [Oracle Statistical Functions](#) - eliminate movement and staging to external systems to perform statistical analysis.
4. [Oracle Database 12c Query Optimization](#) - providing innovation in plan execution and stability.

The following Oracle whitepapers, articles, presentations and data sheets are essential reading:

1. [Oracle Magazine SQL 101 Columns](#)
2. [Oracle Database SQL Language Reference—T-test Statistical Functions](#)
3. [Oracle Statistical Functions Overview](#)
4. [SQL Analytics Data Sheet](#)

You will find links to the above papers, and more, on the “Oracle Analytical SQL ” web page hosted on the Oracle Technology Network:



<http://www.oracle.com/technetwork/database/bi-datawarehousing/sql-analytics-index-1984365.html>



Oracle Corporation, World Headquarters
500 Oracle Parkway
Redwood Shores, CA 94065, USA

Worldwide Inquiries
Phone: +1.650.506.7000
Fax: +1.650.506.7200

CONNECT WITH US

-  blogs.oracle.com/datawarehousing
-  [facebook/BigRedDW](https://facebook.com/BigRedDW)
-  [twitter/BigRedDW](https://twitter.com/BigRedDW)
-  oracle.com/sql
-  [github/oracle/analytical-sql-examples](https://github.com/oracle/analytical-sql-examples)

Hardware and Software, Engineered to Work Together

Copyright © 2015, Oracle and/or its affiliates. All rights reserved. This document is provided for information purposes only, and the contents hereof are subject to change without notice. This document is not warranted to be error-free, nor subject to any other warranties or conditions, whether expressed orally or implied in law, including implied warranties and conditions of merchantability or fitness for a particular purpose. We specifically disclaim any liability with respect to this document, and no contractual obligations are formed either directly or indirectly by this document. This document may not be reproduced or transmitted in any form or by any means, electronic or mechanical, for any purpose, without our prior written permission.

Oracle and Java are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

Intel and Intel Xeon are trademarks or registered trademarks of Intel Corporation. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. AMD, Opteron, the AMD logo, and the AMD Opteron logo are trademarks or registered trademarks of Advanced Micro Devices. UNIX is a registered trademark of The Open Group. 0215



Oracle is committed to developing practices and products that help protect the environment