Building Analytic Reports for Decision Support Systems – Aggregate vs. Analytic Functions

Prof. Ion LUNGU PhD., assist. Adela BÂRA, assist. Vlad DIACONIȚA
Academy of Economic Studies, Bucharest

In order to build analytic reports for Decision Support Systems (DSS) in an organization we can choose to do it by using Business Intelligence techniques such as OLAP and data warehousing or by using traditional reports based on SQL queries. The cost and developing time for BI tools is greater than those for SQL Reports and these factors are important in taking decision on what type of techniques we used for DSS. But some of the requirements of DSS such as prediction or roll-up can make SQL reports development a difficult job. This paper presents some of the analytic functions that are involved in developing SQL reports.
Keywords*: Decision Support Systems (DSS), Structured Query Language (SQL), aggregate functions, analytic functions.*

Introduction

The main goal of Decision Support Systems (DSS) is to assist managers, at different levels in the organization, in taking decisions and to provide in real time representative information, to help and support them in their activities such as analyzing departamental data, planning and forecasting activities for their decision area [LIBA01].
Throught Decision Support Systems managers can manipulate large sets of data in a short period of time or in real time systems. In essence, managers at every departamental level can have a customized view that extracts information from transactional sources and summarizes it into meaningful indicators. DSS can gather data from ERP systems implemented in an organization from different functional areas or modules such as: financials, inventory, purchase, order management, production. Information from this functional modules within a ERP system is managed by a relational software database such as Oracle Database. In the lates versions in addition to aggregate functions Oracle implemented analitycal functions to help developerers building decision support reports [ORMG01]. These functions will be presented in the following sections.

Aggregate versus analytic functions

Aggregate functions applied on a set of records return a single result row based on groups of rows. Aggregate functions such as SUM, AVG and COUNT can appear in SELECT statement and they are commonly used with the GROUP BY clauses. In this case Oracle divides the set of records into groups, specified in the GROUP BY clause. Contrary, if the user does not specified any GROUP BY clause then Oracle returns only a single record as a result of the function that are applied.
Aggregate functions are used in analytic reports to divide data in groups and analyze these groups separately and for building subtotals or totals based on groups.
Oracle provides an extended list of aggregate functions such as: *AVG, CORR_X, COUNT, COVAR_X, DENSE_RANK, FIRST, LAST, MAX, MEDIAN, MIN, PERCENTILE_X, RANK, REGR_X, STATS_X, STDDEV_X, SUM, VAR_X, VARIANCE*, where _X represents an extension or different types of similar functions [ORA10g].
Analytic functions process data based on a group of records but they differ from aggregate functions in that they return multiple rows for each group. The group of rows is called a *window* and is defined by the *analytic clause*. For each row, a sliding window of rows is defined and it determines the range of rows used to process the current row. Window sizes can be based on either a physical number of rows or a logical interval, based on conditions over values.
Analytic functions are performed after completing operations such joins, WHERE, GROUP BY and HAVING clauses, but before ORDER BY clause. Therefore, analytic

functions can appear only in the select list or ORDER BY clause.

Analytic functions are commonly used to compute cumulative, moving and reporting aggregates [ORA10g].

The structure of an analytic function is:

ANALYTIC_FUNCTION (arguments) OVER (analytic clause)

If you want to indicate that the function operates on a query result set then you should use *OVER analytic clause,* where *analytic clause* may have the following options:

PARTITION BY (expression1, expression2,…) ORDER [SIBLINGS] BY expression / position /alias [ASC/DESC] [nulls first/last] WINDOW CLAUSE

In order to partition the query result set into groups based on one or more expressions use the PARTITION BY clause. If you omit this clause, then the function treats all rows of the query result set as a single group.

Window clause can be:

ROWS/RANGE [BETWEEN] {UNBOUNDED PRECEDING}/ {CURRENT ROW}/ {expression PRECEDING/FOLLOWING}[AND] {UNBOUNDED PRECEDING}/ {CURRENT ROW}/ {expression PRECEDING/FOLLOWING}

If no *window clause* is present then Oracle considers it to ***RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW.***

In most cases aggregate functions have the analytic version, that's way the list of analytic function is: *AVG, CORR, COVAR_X, COUNT, DENSE_RANK, FIRST, FIRST_VALUE, LAG, LAST, LAST_VALUE, LEAD, MAX, MIN, NTILE, PERCENT_RANK, PERCENTILE_CONT, PERCENTILE_DISC, RANK, RATIO_TO_REPORT, REGR_X, ROW_NUMBER, STDDEV_X, SUM, VAR_X, VARIANCE.*

In the following section we'll try to use these analytic functions in queries that can be applied in decision support reports.

Building analytic queries

We'll consider a set of examples based on the following tables: *FURNIZORI, COMENZI_APROV, PRODUSE* and *RULAJE_BALANTA.* The structure of these tables can be found at: http://bd.ase.ro and also a set of records for testing.

1) ***AVG function*** is used to calculate the average of values of rows specified in window clauses. For example, in order to calculate the average of net period over 3 consecutives days group by company we can write:

```
select e.compania, e.cont, e.moneda,
e.data, e.rulaj_d,
        avg(e.rulaj_d) over (partition by
        e.compania order by e.data
        rows between 1 preceding and 1 fol-
        lowing) as medie_rulaj_debitor
        from rulaje_balanta e;
```

Using ***ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING*** clause then we analyze the neighbours of the current row relative to one position in days and for every company. Modifying the clause we can analyze the average of net period over 3 months (preceding, current and following month):

```
select e.compania, e.cont, e.moneda,
e.data, e.rulaj_d,
avg(e.rulaj_d) over (partition by
e.compania order by extract(month
from e.data)
rows between 1 preceding and 1 fol-
lowing) as medie_rulaj_debitor
from rulaje_balanta e
order by e.data;
```

2) ***COUNT function*** in analytic version is used to calculate the number of rows involved in the window clause. In the following example we count the number of products for each vendor having order value between [current value-1000 and current value +1000] or in the range 1000 less than through 1000 greater than the current order value:

```
select c.vendor_id,
c.inventory_item_id,
count(*)
over (order by
(c.unit_price*c.quantity_received)
range between 1000 preceding and
1000 following) as
nr_comenzi_interval
from expr_detalii_comenzi_apr_v c
```

Another example counts the number of companies that have the period net in the range 1000 less than trough 1000 greater than the current unit for class 6 accounts and where currency is RON:

```
select e.cont,
e.compania,sum(rulaj_d),
count(*)
over (order by sum(rulaj_d) range
between 1000 preceding and 1000
following) as nr_vecini
from rulaje_balanta e
where e.moneda ='RON' and cont
like'6%'
group by e.cont, e.compania
order by cont
```

3) ***MIN and MAX function*** in the analytic version are used to classify and build ratings over the rows in the window clause. In the following example the products are classified over order values for each vendor:

```
select c.vendor_id, c.organization_id,
min(c.unit_price*c.quantity_received
) keep (dense_rank first order by
unit_price)
over (partition by
c.inventory_item_id) as inferior,
max(c.unit_price*c.quantity_received
) keep (dense_rank last order by
unit_price)
over (partition by
c.inventory_item_id) as superior
from
EXPR_DETALII_COMENZI_APR_V
c
order by c.vendor_id
```

Another example is used to classify all expenses for 6 class accounts chart between the current account limits:

```
select e.compania,
e.cont,sum(e.rulaj_d),
min(sum(e.rulaj_d)) keep
(dense_rank first order by e.cont)
over (partition by e.cont) as inferior,
max(sum(e.rulaj_d)) keep
(dense_rank last order by e.cont)
over (partition by e.cont) as superior
from rulaje_balanta e
where cont like'6%' and
moneda='RON'
group by e.compania, e.cont
order by cont, sum(e.rulaj_d)
```

4) ***FIRST_VALUE*** and ***LAST_VALUE*** functions are used as another method for classifying over a range. The main advantage is that through these functions can be processed rows with similar limits, but with different classifications criteria. The above examples can be done like this:

```
select e.compania,
e.cont,sum(e.rulaj_d),
first_value(sum(e.rulaj_d)) over (par-
tition by e.cont order by
sum(e.rulaj_d) rows unbounded pre-
ceding) as inferior,
last_value(sum(e.rulaj_d)) over (par-
tition by e.cont order by
sum(e.rulaj_d) rows between un-
bounded preceding and unbounded
following) as superior
from rulaje_balanta e
where cont like'6%' and
moneda='RON'
group by e.compania, e.cont
order by cont, sum(e.rulaj_d)
```

Modifying the window clause we can find out we are the minimum and maximum value between current period net ± 1000000 RON:

```
select e.compania,
e.cont,sum(e.rulaj_d),
first_value(sum(e.rulaj_d)) over (par-
tition by e.cont order by
sum(e.rulaj_d) range between
1000000 preceding and 1000000 fol-
lowing) as inferior,
last_value(sum(e.rulaj_d)) over (par-
tition by e.cont order by
sum(e.rulaj_d) range between
1000000 preceding and 1000000 fol-
lowing) as superior
from rulaje_balanta e
where cont like'6%' and
moneda='RON'
group by e.compania, e.cont
order by cont, sum(e.rulaj_d)
```

5) ***LAG and LEAD functions*** report the current value to previous/next values of each row specified in the window clause. The functions provide access to more than one row of a table at the same time without a self join. Given a series of rows returned from a query and a position of the cursor, LAG and LEAD provide access to a row at a given physical offset prior /beyond that position. If you do not specify offset, then its default is 1. The optional default value is returned if the offset goes beyond the scope of the window. If you do not specify default, then its default is null [ORA10g].

Using LAG function in the next example we retrieve the current and the previous values of period net with an offset equal to 1:

select e.cont,e.data, e.compania,sum(e.rulaj_d) ru-laj_curent,
lag(sum(e.rulaj_d),1,0) over (order by cont, data, compania)
as rulaj_anterior
from rulaje_balanta e
where cont like'6%' and moneda='RON'
group by e.cont, e.data, e.compania

An with LEAD function we can retrieve the current and the next values of period net with an offset equal to 1:

select e.cont,e.data, e.compania,sum(e.rulaj_d) ru-laj_curent,
lag(sum(e.rulaj_d),1,0) over (order by cont, data, compania)
as rulaj_anterior,
lead(sum(e.rulaj_d),1,0) over (order by cont, data, compania)
as rulaj_urmator
from rulaje_balanta e
where cont like'6%' and moneda='RON'
group by e.cont, e.data, e.compania

LAG and LEAD functions are very useful for forecasts and predictions over the historical and current data with the specified offset.

6) ***RANK function*** is used also for classification, returning the current position in the window. For example we can retrieve the position of each type of expense in the amount of expenses:

select e.compania, e.cont, sum(e.rulaj_d),
rank() over (partition by e.compania order by sum(e.rulaj_d) desc) pozitie
from rulaje_balanta e
where e.moneda='ron' and e.cont like '6%'
group by e.compania, e.cont
order by e.compania, pozitie

7) ***SUM function*** used in an analytical manner calculates a cumulative total based on each group in the window. Next, we can calculate subtotals for order valuein the range 100000 less than trough 100000 greater than the current order:

*select c.vendor_id,c.inventory_organization_id, c.inventory_item_id, sum(c.unit_price*c.quantity_received)*
over (partition by c.inventory_organization_id
*order by c.unit_price*c.quantity_received*
range between 100000 preceding and 100000 following) cumulat
from expr_detalii_comenzi_apr_v c

Or, we can change the window clause to calculate subtotals for orders that have values less than or equal to current order:

*select c.vendor_id,c.inventory_organization_id, c.inventory_item_id, sum(c.unit_price*c.quantity_received)*
over (partition by c.inventory_organization_id
*order by c.unit_price*c.quantity_received*
range unbounded preceding) cumu-lat
from expr_detalii_comenzi_apr_v c

Conclusions

Decision Support Systems are regularly based on a set of views that extract, join and aggregate rows from many tables from the ERP systems. In order to develop such types of systems we can choose BI tools or Reports based on SQL queries. The last solution is low consuming developing time and costs. For developing reports easiest an important option is to choose analytic functions for predictions (LAG and LEAD), subtotals over current period (SUM and COUNT), classifications or ratings (MIN, MAX, RANK, FIRST_VALUE and LAST_VALUE).

References

[LIBA01] Lungu Ion, Bara Adela, Fodor Anca - *Business Intelligence tools for building the Executive Information Systems*, 5thRoEduNet International Conference, Universitatea Lucian Blaga, Sibiu, june 2006

[ORA10g] Oracle Corporation - *Database Performance Tuning Guide 10g Release 2 (10.2),* Part Number B14211-01, 2005

[ORMG01] Oracle Corporation - *Oracle Magazine*, 2006

[NET**] www.oracle.com