Don’t Be In a Funk: Use Analytic Functions

Philip Rice, University of California Santa Cruz

Abstract
Analytic Functions have been available for years, but have not gained good visibility. These power tools should not be a secret! Real world examples show how complex problems can be solved more easily. Know when to consider analytic functions at your site. Report writers and developers can be excited by the power of the feature set. DBAs will want to know about performance benefits. All of this is without extra cost licensing costs.

TARGET AUDIENCE
This session paper is Introductory, and assumes the reader has SQL experience with the GROUP BY clause, i.e. aggregates. Regardless of years of SQL experience, anyone with complex reporting requirements can benefit from seeing why and how to use Analytic Functions.

EXECUTIVE SUMMARY
This paper will:
• Demonstrate how the Analytic Function rich feature set offers flexibility benefits in solving complex reporting problems
• Show how major runtime performance gains are possible
• Show how the "WITH" clause (Common Table Expression) offers benefits for relatively complex SQL, often helpful in analytic function situations

BACKGROUND
Analytic Functions are Ancient History in IT time. They have been available since 8.1.6(!), and in Standard Edition starting with 9iR2. These are available at no extra cost. Data Mining, Advanced Analytics and OLAP are extra cost options, not pertinent to our topic, so we will not look at those. Seeing "analytic" might make you think that there is a focus on statistics, but the feature set is much more inclusive and helpful for reporting. In spite of all of that cost benefit (free), and all of that time (available for well over a decade), Analytic Functions have not become well known. Microsoft SQL Server 2012 has some of this feature set, but don’t expect to find this in MySQL.

The goal in this paper is to raise awareness of possibilities. In this case study, with real world examples, consider how you might apply the ideas for functionality and performance at your site. This will not be a comprehensive view of all functions, but it can get you thinking about the potential. Even though analytic functions cannot be a magic answer for everything, they can be a powerful tool to use. You do not need to be an expert to share your newly gained awareness. Use the slides from the presentation at your site if you wish. Tell others -- they will appreciate it!

TECHNICAL DISCUSSIONS AND EXAMPLES

BASICS -- STARTING ON THE SAME PAGE
The title for this paper comes from a real world situation. A developer called out "Hey, does anyone know SQL?!?" She was having frustration (In a Funk) after trying to solve a problem for an hour, based on conventional awareness of SQL. After she explained the requirements, an Analytic Function solved the problem easily, and she was quite relieved and pleased with the simple solution.
Before getting to the real world examples, we should start with some basics. There is some overlap with “traditional” aggregate functions. For example, max, avg and count have aggregate and analytic variations. They have the same function keyword, and there is similar syntax.

Analytics are used in SELECT statement, and they are most likely to be most helpful in reporting situations. You'll see that analytics offer better functionality. In some situations, the traditional approach can be much more difficult for the same result. You'll see benefits of performance improvement, which is likely to be more obvious with larger datasets. There is potential to bring runtime of hours down to minutes.

**GENERAL SYNTAX**

Watch for the “OVER” keyword. That is the indicator of an analytic function, e.g. used with max, avg and count.

```
Function(arg1, ..., argn) OVER (  
    [PARTITION BY <...>]
    [ORDER BY <...>]
    [window_clause]  )
```

It's probably easiest to think of PARTITION BY as being comparable to GROUP BY. PARTITION has no connection with partitions in tables. The use here deals with how resultsets are grouped.

The window clause can be very helpful. For example, you can look at different time periods on same row of output. You will see examples later. The window clause (partial) syntax is

```
[ROW or RANGE] BETWEEN <start> AND <end>
```

For the ROW keyword, it can refer to one or more rows, prior or following. The LAG or LEAD function can be used for handling ROWs in a window, without needing the ROW keyword. An example in this paper uses LAG.

For the RANGE keyword, think in terms of a logical offset. The RANGE type of window can be anchored or sliding. It needs to be operating on a date or numeric column. As an example in using the DATE datatype, RANGE 30 PRECEDING would look at the last 30 days. This would be sliding, because the 30 day window would be shifting as each row comes in. The anchored variety would start with the partition’s first row, and end with the current row being processed.

In some situations, it can be important to know about timing of execution in SQL statements. There are a few major stages:

1) Joins, and clauses are applied for WHERE, GROUP BY, HAVING
2) Analytics are applied “OVER” the resultset from step 1
3) The main ORDER BY of a query is applied after analytic functions. Analytic functions can only appear in:

• select list (not clauses in step 1)
• main ORDER BY clause of the query

Limitations in the last item should not normally be significant. An analytic function result set can be supplied to the outer query, and the WHERE clause can make use of that inner resultset. You'll see examples later. However, be very aware of how a view can change your results, as detailed near the end of this paper. Similarly, be aware when you combine GROUP BY with analytics.

The “OVER” keyword might not feel especially intuitive. One way to think of it is that an analytic is applied in stage 2 (in our numbered list above) OVER the entire resultset that was derived from the conditions in stage 1.
**COMPARISON: “TRADITIONAL” AGGREGATE VS. ANALYTIC**

We'll start by looking at an example of the “traditional” aggregate count function. Some old-timers will recognize that the OUTLN owner is an old 9i feature, but it is good here for example purposes.

```sql
select count(*), OBJECT_TYPE
from all_objects
where owner = 'OUTLN'
group by OBJECT_TYPE;
```

<table>
<thead>
<tr>
<th>COUNT(*)</th>
<th>OBJECT_TYPE</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 INDEX</td>
<td></td>
</tr>
<tr>
<td>1 PROCEDURE</td>
<td></td>
</tr>
<tr>
<td>3 TABLE</td>
<td></td>
</tr>
</tbody>
</table>

Note that non-aggregated columns must be in the GROUP BY clause. What if we want to show detail at same time as the aggregate? With analytics (when we see the OVER keyword), we can show both detail and aggregate at once. In the first use of the count function in the next example, the “OVER” clause has nothing in the parentheses. That means we are not PARTITIONing (grouping) on anything, and thus our function will act on the whole result set. In the column after that, we are PARTITIONing (grouping) on each object_type. This time we can get the different result set partitions, comparable to the “traditional” aggregate count function, but we see all the detail rows.

```sql
select object_name, object_type,
    count(*) OVER () tot_count,
    count(*) OVER (PARTITION BY object_type) type_count
from all_objects where owner = 'OUTLN';
```

<table>
<thead>
<tr>
<th>OBJECT_NAME</th>
<th>OBJECT_TYPE</th>
<th>TOT_COUNT</th>
<th>TYPE_COUNT</th>
</tr>
</thead>
<tbody>
<tr>
<td>OL$NAME</td>
<td>INDEX</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>OL$HNT_NUM</td>
<td>INDEX</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>OL$SIGNATURE</td>
<td>INDEX</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>OL$NODE_OL_NAME</td>
<td>INDEX</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>ORA$GRANT_SYS_SELECT</td>
<td>PROCEDURE</td>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td>OL$NODES</td>
<td>TABLE</td>
<td>8</td>
<td>3</td>
</tr>
<tr>
<td>OL$HINTS</td>
<td>TABLE</td>
<td>8</td>
<td>3</td>
</tr>
<tr>
<td>OL$</td>
<td>TABLE</td>
<td>8</td>
<td>3</td>
</tr>
</tbody>
</table>

**ROW_NUMBER AND RANK**

We'll build on the count example above to show three more functions, looking at the Index and Table objects. As shown below, some have the same LAST_DDL_TIME. This demonstrates how RANK works. Note that all three functions have same PARTITION BY and ORDER BY clauses.

```sql
SELECT object_type "ObjTyp", substr(object_name,1,10) "ObjName",
    to_char(last_ddl_time,'yyyymmdd hh24miss') last_ddl_time,
    row_number() over
    (PARTITION BY object_type ORDER BY last_ddl_time) RN,
    rank() over
    (PARTITION BY object_type ORDER BY last_ddl_time) R,
    dense_rank() over
    (PARTITION BY object_type ORDER BY last_ddl_time) DR
FROM all_objects
WHERE owner = 'OUTLN' AND object_type IN ('TABLE', 'INDEX');
```

Below we see results for Row_number, Rank and Dense Rank. If two records have the same value in the ORDER BY, the two records get a different ROW_NUMBER. RANK and DENSE_RANK do not work like that. If two records have the same value in the ORDER BY, they both get the same RANK or DENSE_RANK. The difference between RANK and
DENSE_RANK is how they are counted. DENSE_RANK uses sequential numbers, RANK does not. In the INDEX set, the fourth line is different from the first three. RANK jumps to display "4", and DENSE_RANK is "2".

\[ RN = \text{Row number}, \quad R = \text{Rank}, \quad DR = \text{Dense Rank} \]

<table>
<thead>
<tr>
<th>ObjTyp</th>
<th>ObjName</th>
<th>LAST_DDL_TIME</th>
<th>RN</th>
<th>R</th>
<th>DR</th>
</tr>
</thead>
<tbody>
<tr>
<td>INDEX</td>
<td>OL$NAME</td>
<td>20031001 173156</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>INDEX</td>
<td>OL$HINT_NUM</td>
<td>20031001 173156</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>INDEX</td>
<td>OL$SIGNATU</td>
<td>20031001 173156</td>
<td>3</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>INDEX</td>
<td>OL$NODE_OL</td>
<td>20080906 102159</td>
<td>4</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>TABLE</td>
<td>OL$NODES</td>
<td>20080906 102610</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>TABLE</td>
<td>OL$</td>
<td>20080906 102610</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>TABLE</td>
<td>OL$HINTS</td>
<td>20080906 102610</td>
<td>3</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

ROW_NUMBER is similar to the ROWNUM pseudo-column. One key difference is that ROWNUM gets incremented as rows are returned from the query. That means that we cannot say "WHERE ROWNUM = 5". But testing showed that ROW_NUMBER can be used that way, giving you an extra piece of functionality.

Be aware of non-deterministic queries giving inconsistent results in more than one execution. That is a good awareness to have in general, and does not apply specifically to analytics. An example would be in asking for Top 3 of something, such as highest widget sales rounded to the nearest thousand dollars. If there are 5 widgets with the same sales amounts, you can’t be sure of getting the same Top 3 in each run.

The next example shows that we can sort both ways in the same SQL statement. We traditionally think of just one ORDER BY clause, affecting the entire statement. For each analytic function, the ORDER BY clause is independent for the column value. That provides the extra flexibility. In the example below, note that LAST_DDL_TIME is the same for some rows, but ROW_NUMBER does not show them as matching row numbers. This is different from Ranking.

```sql
SELECT object_type "ObjTyp",
       to_char(last_ddl_time,'yyyymmdd hh24miss') last_ddl_time,
       row_number() OVER ( partition by object_type order by last_ddl_time) SORTUP,
       row_number() OVER ( partition by object_type order by last_ddl_time DESC NULLS LAST) SORTDOWN
FROM  all_objects
WHERE owner = 'OUTLN'
  AND object_type IN ('TABLE','INDEX');
```

<table>
<thead>
<tr>
<th>ObjTyp</th>
<th>LAST_DDL_TIME</th>
<th>SORTUP</th>
<th>SORTDOWN</th>
</tr>
</thead>
<tbody>
<tr>
<td>INDEX</td>
<td>20031001 173156</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>INDEX</td>
<td>20031001 173156</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>INDEX</td>
<td>20031001 173156</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>INDEX</td>
<td>20080906 102159</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>TABLE</td>
<td>20080906 102610</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>TABLE</td>
<td>20080906 102610</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>TABLE</td>
<td>20080906 102610</td>
<td>3</td>
<td>1</td>
</tr>
</tbody>
</table>

So far, we have seen relatively basic uses, primarily showing flexibility. Now we see a real world example where performance was the major factor. Some readers will recognize EFFDT and EFFSEQ columns that are prevalent in PeopleSoft. The requirement for the original report was to show detail columns, along with maximum effective date and maximum effective sequence. First, we see the original statement, as generated by a reporting tool. In the “traditional” slow way to see mixed detail and summary levels, we need separate statements for each maximum value.
Why did this statement need tuning attention? Because it required immense amounts of temp space. At first, it gave an error that ran out of space in the 32GB temp file. But after adding four more temp files for a total of 160GB of temp space, it still never finished! Clearly, an alternative was required. In the example below, there is abbreviated syntax, because we have no extra value in showing column names or WHERE predicates. We only care about the structure involved in comparing the old and the new way.

```sql
SELECT <detail columns>, max_effdt, max_effseq
FROM
  ( SELECT <detail columns>,
    MAX (DISTINCT t2.APLAN_EFFDT) max_effdt
    FROM t3, t1 LEFT OUTER JOIN t2
    WHERE [...] GROUP BY t1.CTERM_EMPLID, t1.CTERM_TERM_CD) d5,
  ( SELECT <detail columns>,
    MAX (t2.APLAN_EFFSEQ) max_effseq
    FROM t3, t1 LEFT OUTER JOIN t2
    WHERE [...] GROUP BY t1.CTERM_EMPLID, t1.CTERM_TERM_CD,
    t2.APLAN_EFFDT) d4,
  ( SELECT <detail columns only, no aggregate!!!>
    FROM t3, t1 LEFT OUTER JOIN t2
    WHERE [...] < NO group by clause!!!> ) d3
WHERE < predicates for outer select >
ORDER BY < columns for outer select >;
```

Our Analytic Function approach made this FASTER!! Improvement went down to minutes, rather than never completing. This particular example is the oldest in this paper, and improvements in newer releases may have less dramatic runtime differences. However, this example at least shows that analytics can offer potential for significant performance gains.

```sql
SELECT <detail columns>, max_effdt, max_effseq
FROM
  ( SELECT <detail columns>,
    max(t2.APLAN_EFFDT) OVER
      (PARTITION BY t1.cterm_emplid, t1.cterm_term_cd)
      AS max_effdt,
    max(t2.APLAN_EFFSEQ) OVER
      (PARTITION BY t1.cterm_emplid, t1.cterm_term_cd,
      t2.APLAN_EFFDT)
      AS max_effseq
    FROM t3, t5 (t1 LEFT OUTER JOIN t2 [...] )
    t4) WHERE [...] LEFT OUTER JOIN t4 [...] WHERE [...] )
WHERE < predicates for outer select >
ORDER BY < columns for outer select >;
```

**Example of UPDATE**

Originally, this example was a two-step approach. Developers took the easiest way they knew with traditional SQL methods:

1) Update the entire table, setting a flag column to a single value for all rows.
2) Go back and “fix” a subset, which was also a significant part of the table. This required much more time and redo generation than necessary.

A better way is to find just the rows we need. The inner select gets values used as predicates in the next SELECT statement, and the UPDATE acts on those rowid values.

```sql
UPDATE FZBRFCX SET FZBRFCX_ZERO_FLAG = 2
WHERE rowid IN
  (SELECT rowid FROM
```
(SELECT rowid, FZBRFCX_ZERO_FLAG Flag, 
  sum(FZBRFCX_TRANS_AMT) OVER 
  (PARTITION BY FZBRFCX_ACCT_CODE, 
   FZBRFCX_FUND_CODE, 
   FZBRFCX_DOC_REF_NUM) Sum_Amt 
FROM FZBRFCX 
) WHERE Sum_Amt = 0 AND Flag <> 0 ) ;

HELPFUL SIDETRACK: QUERY SUBFACTORING, AKA COMMON TABLE EXPRESSION
We'll step away from analytic functions briefly, for a feature that is often very helpful with our topic. Oracle calls it Query Subfactoring. In ANSI standards, it is known as Common Table Expression (CTE). You could just refer to it as the WITH clause. You will find this in the SELECT statement documentation. The WITH clause also has PL/SQL declarations as an optional phrase, but we are not interested in that here.

Analytic functions often need an inline view, i.e. a subquery. As shown in the UPDATE example above, we need an extra layer so values can be used in a WHERE clause. Sometimes the inline views are nested, with multiple layers for complex reporting situations. In the following statement, indentation is helpful, but it is less than ideal. Most notably, aliases for the outermost statements are separated from the statement itself. In a complex reporting situation, an outermost WHERE clause might be separated from the related SELECT clause by a half page or more. The same can be true for separation of statement aliases.

Traditional inline: layers with indentation
SELECT MID_LVL.po_code, MID_LVL.seq, [...] 
FROM 
  (select INNER_LVL.po_code, INNER_LVL.seq, [...] 
  FROM 
    (select po_code, seq, [...] 
    FROM fprpoda 
    where po_code in 
      (select b.po_code 
      FROM fprpoda 
      where activity_date 
        between '01-NOV-09' and '09-NOV-09') CODE_LIST 
    ) INNER_LVL 
  ) MID_LVL 
WHERE < [MID_LVL.column] predicates...>

Using Common Table Expression (CTE) allows a Top-Down style, which is more readable. In the slide presentation for this paper, Query Subfactoring (CTE) allows related SQL statements to be split across multiple slides, so code is easier to follow. This can be considered a Top-Down style, because each alias can be used for the FROM clause in a statement below it.

The WITH keyword says we will have one or more statements, with an associated alias. When we stop seeing commas to separate the aliased statements, the WITH clause is done. Then we have the final statement, making use of the work preceding it. This allows us to easily see the main purpose of the code, nicely separated at the end.

WITH 
  CODE_LIST AS 
    (select po_code 
    from fprpoda 
    where activity_date 
      between '01-NOV-09' and '09-NOV-09'), 
  INNER_LVL AS 
    (select po_code, seq, [...] 
    from fprpoda 
    where po_code in CODE_LIST), 

  ...
MID_LVL AS
  (select po_code, seq, [...] from INNER_LVL )
SELECT * FROM MID_LVL
WHERE < predicates...>

Be aware that all aliases must be used at least once further down, or you will get an error.

In the example above, we only use each alias statement once. Subfactoring offers another benefit, by easily allowing multiple use of any alias. A real world example is a report that had five UNION ALL statements. Each of the five had a few predicates in common, along with a couple predicates that were different in each individual statement. Pulling those common elements into a statement with an alias allowed the alias to be used five times. The code became much simpler, more readable. In addition, the execution plan showed that the alias was materialized to a temporary view, which was used five times, giving the best efficiency.

In 11gR2 and above, a newer feature called Recursive Query Subfactoring is available. This paper does not have reason to use that feature, but some advanced reporting can benefit from it.

Now we get back to our main topic again, using the Query Subfactoring feature.

**Running Totals and Windowing**

This next example has a Financial Auditing requirement. For any Purchase Order, there can be multiple sequence numbers, as amounts are entered. Above the $50,000 level, more stringent approvals are required. A monthly report shows whether anyone was trying to get around the audit rules, by entering artificially smaller amounts. The report will show values from current and previous rows, to find a PO where the running total went above $50,000.

Originally, the developer thought it would require PL/SQL, with multiple cursors starting and stopping. Even though our code below is non-trivial, a PL/SQL approach would probably have been a daunting task. For a large table, the PL/SQL approach would probably be slower, due to the “row by row” processing, rather than a set operation. In our example, we'll use Query Subfactoring (CTE) to see the pieces build on each other.

**Running Totals and Windowing**

WITH
code_list AS ( -- [Purchase Order codes used in next stmt]
  SELECT distinct po_code
  FROM  fprpoda
  WHERE trunc(activity_date)
    BETWEEN '01-NOV-09' AND '09-NOV-09'
    AND seq is not null ),
  -- Alias #2, using what was defined above
INNER_LVL AS ( -- [sum each code and seq combo]
  SELECT po_code,
    LAG(po_code, 1) OVER
    (ORDER BY po_code) "PrevCode",
    seq , amt "CurrAmt", activity_date,
    SUM(amt) OVER (PARTITION BY po_code
    ORDER BY po_code, seq, activity_date) running_tot
  FROM fprpoda
  WHERE po_code IN
    ( select po_code from CODE_LIST ) ),
  -- Alias #3, using what was defined above
MID_LVL AS ( -- get curr/prev row values
  SELECT po_code, seq ,
)

...
(CASE WHEN "PrevCode" != po_code THEN NULL
  -- 1 is the default for LAG.
  ELSE LAG(running_tot, 1) OVER
    (ORDER BY po_code, seq) END) "PrevRunTot",
running_tot "RunningTot",
activity_date curr_actv,
(CASE WHEN "PrevCode" != po_code THEN NULL
  ELSE  LAG(activity_date)
    OVER
    (ORDER BY po_code, seq) END)  prev_actv
FROM INNER_LVL)

/*
Query subfactoring (CTE) above is done. Now we can use one nicely
isolated statement to show what we're ultimately doing.
As shown in the results below, we see a Running Total of at least $50,000, and a Previous Running Total of less than $50,000.
Four Purchase Orders would be subject to investigation.
*/

SELECT po_code, seq, "PrevRunTot",
  "RunningTot" - "PrevRunTot" "DiffChange",
  "RunningTot" , prev_actv, curr_actv
FROM MID_LVL
WHERE "PrevRunTot" <  50000
  AND "RunningTot" >= 50000;

<table>
<thead>
<tr>
<th>PO_CODE</th>
<th>SEQ</th>
<th>PrevRunTot</th>
<th>DiffChange</th>
<th>RunningTot</th>
<th>PREV_ACTV</th>
<th>CURR_ACTV</th>
</tr>
</thead>
<tbody>
<tr>
<td>B0142584</td>
<td>7</td>
<td>46,800.00</td>
<td>5,500.00</td>
<td>52,300.00</td>
<td>05-FEB-09</td>
<td>05-NOV-09</td>
</tr>
<tr>
<td>B0181676</td>
<td>1</td>
<td>38,142.00</td>
<td>23,856.34</td>
<td>61,998.34</td>
<td>26-NOV-07</td>
<td>17-NOV-08</td>
</tr>
<tr>
<td>S0176940</td>
<td>1</td>
<td>43,371.00</td>
<td>42,156.00</td>
<td>85,527.00</td>
<td>17-JUN-05</td>
<td>23-MAR-06</td>
</tr>
<tr>
<td>S0181330</td>
<td>1</td>
<td>1.00</td>
<td>302,069.91</td>
<td>302,070.91</td>
<td>20-JUL-07</td>
<td>28-AUG-07</td>
</tr>
</tbody>
</table>

Take a look at the details for Purchase Order B0142584. We see that Sequence 7 jumped past the threshold. For Sequence 0, 2, and 7, we see that they have the exact same date/time, so the running total is not gradually increasing. Even though that may look a bit odd, we still get valid results for the threshold in the report.

<table>
<thead>
<tr>
<th>PO_CODE</th>
<th>SEQ</th>
<th>CurrAmt</th>
<th>RunningTot</th>
<th>Activity_Date_Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>B0142584</td>
<td>0</td>
<td>1.00</td>
<td>5,001.00</td>
<td>22-JAN-2003 10:27:00</td>
</tr>
<tr>
<td>B0142584</td>
<td>0</td>
<td>5,000.00</td>
<td>5,001.00</td>
<td>22-JAN-2003 10:27:00</td>
</tr>
<tr>
<td>B0142584</td>
<td>1</td>
<td>6,500.00</td>
<td>11,501.00</td>
<td>09-OCT-2003 14:36:01</td>
</tr>
<tr>
<td>B0142584</td>
<td>2</td>
<td>-1.00</td>
<td>18,500.00</td>
<td>27-OCT-2004 15:51:01</td>
</tr>
<tr>
<td>B0142584</td>
<td>2</td>
<td>7,000.00</td>
<td>18,500.00</td>
<td>27-OCT-2004 15:51:01</td>
</tr>
<tr>
<td>B0142584</td>
<td>3</td>
<td>9,500.00</td>
<td>28,000.00</td>
<td>05-OCT-2006 13:27:01</td>
</tr>
<tr>
<td>B0142584</td>
<td>4</td>
<td>4,000.00</td>
<td>32,000.00</td>
<td>25-OCT-2007 09:45:02</td>
</tr>
<tr>
<td>B0142584</td>
<td>5</td>
<td>5,500.00</td>
<td>37,500.00</td>
<td>27-NOV-2007 10:12:03</td>
</tr>
<tr>
<td>B0142584</td>
<td>6</td>
<td>9,300.00</td>
<td>46,800.00</td>
<td>05-FEB-2009 11:12:01</td>
</tr>
<tr>
<td>B0142584</td>
<td>7</td>
<td>-7,000.00</td>
<td>52,300.00</td>
<td>05-NOV-2009 12:27:01</td>
</tr>
<tr>
<td>B0142584</td>
<td>7</td>
<td>7,000.00</td>
<td>52,300.00</td>
<td>05-NOV-2009 12:27:01</td>
</tr>
<tr>
<td>B0142584</td>
<td>7</td>
<td>-9,300.00</td>
<td>52,300.00</td>
<td>05-NOV-2009 12:27:01</td>
</tr>
<tr>
<td>B0142584</td>
<td>7</td>
<td>9,500.00</td>
<td>52,300.00</td>
<td>05-NOV-2009 12:27:01</td>
</tr>
<tr>
<td>B0142584</td>
<td>7</td>
<td>-4,000.00</td>
<td>52,300.00</td>
<td>05-NOV-2009 12:27:01</td>
</tr>
<tr>
<td>B0142584</td>
<td>7</td>
<td>-11,500.00</td>
<td>52,300.00</td>
<td>05-NOV-2009 12:27:01</td>
</tr>
<tr>
<td>B0142584</td>
<td>7</td>
<td>9,500.00</td>
<td>52,300.00</td>
<td>05-NOV-2009 12:27:01</td>
</tr>
<tr>
<td>B0142584</td>
<td>7</td>
<td>-9,300.00</td>
<td>52,300.00</td>
<td>05-NOV-2009 12:27:01</td>
</tr>
<tr>
<td>B0142584</td>
<td>7</td>
<td>11,500.00</td>
<td>52,300.00</td>
<td>05-NOV-2009 12:27:01</td>
</tr>
<tr>
<td>B0142584</td>
<td>7</td>
<td>9,300.00</td>
<td>52,300.00</td>
<td>05-NOV-2009 12:27:01</td>
</tr>
</tbody>
</table>
**Running Totals and Windowing: Notes**

- The LAG function puts current and previous values on same row.
- That allows us to easily use a WHERE clause to find our threshold.
- We could not put LAG in the statement with our running total. We needed extra layering because current/previous row values were not available until the running total was done.
- We got a new running total for each Purchase Order Code, because that is in the “PARTITION BY” clause.

**LISTAGG, available starting with 11gR2**

This recently available function concatenates values from rows into a string, i.e. a LIST AGGregation. It can be used as a Simple Aggregate OR an Analytic variety. Our example is a continuation of our Running Total. We saw duplicate dates for some Sequences. In our prior detail level example, each date was on a separate line. As shown below, LISTAGG offers the convenience of displaying multiple dates for each sequence on the same line of output. Part of the syntax is that you supply the separator character, with a semicolon used here.

### 11.2 LISTAGG: Simple Aggregate (GROUP BY)

```sql
-- SEQ 0 & 2 has same date/time, grouped on same line:
SELECT seq,
   LISTAGG(to_char(activity_date,'MON-YYYY'), '; ') WITHIN GROUP (ORDER BY seq) "Activity_Dates"
FROM fprpoda WHERE po_code = 'B0142584' AND seq < 3
GROUP BY seq;
```

<table>
<thead>
<tr>
<th>SEQ</th>
<th>Activity_Dates</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>JAN-2003; JAN-2003</td>
</tr>
<tr>
<td>1</td>
<td>OCT-2003</td>
</tr>
<tr>
<td>2</td>
<td>OCT-2004; OCT-2004</td>
</tr>
</tbody>
</table>

### 11.2 LISTAGG: Analytic

For the Simple Aggregate variety of LISTAGG, we needed GROUP BY. The Analytic variety operates at the detail level. LISTAGG does not have a way to eliminate duplicates. To do that, DISTINCT is required at the statement level after the SELECT keyword. That method allows looking across column values to find duplicate rows. We need DISTINCT for our example below of the Analytic variety of LISTAGG. Otherwise, it would show all 19 rows of detail, comparable to what we saw in that detail section further above.

```sql
SELECT DISTINCT seq,
   SUM(amt) OVER (ORDER BY seq, activity_date) "RunTot",
   SUM(amt) OVER (PARTITION BY seq ORDER BY seq, activity_date) "SeqTot",
   LISTAGG(amt, ';') WITHIN GROUP (ORDER BY seq)
   OVER (PARTITION BY seq) "Amts"
FROM fprpoda WHERE po_code = 'B0142584'
   AND seq IS NOT NULL ORDER BY seq;
```

-- Sequence 7 has ten Amount entries. Some cancel each other out, but we see all ten.

<table>
<thead>
<tr>
<th>Seq</th>
<th>RunTot</th>
<th>SeqTot</th>
<th>Amts</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>5001</td>
<td>5001</td>
<td>5000;1</td>
</tr>
<tr>
<td>1</td>
<td>11501</td>
<td>6500</td>
<td>6500</td>
</tr>
<tr>
<td>2</td>
<td>18500</td>
<td>6999</td>
<td>7000;-1</td>
</tr>
<tr>
<td>3</td>
<td>28000</td>
<td>9500</td>
<td>9500</td>
</tr>
<tr>
<td>4</td>
<td>32000</td>
<td>4000</td>
<td>4000</td>
</tr>
<tr>
<td>5</td>
<td>37500</td>
<td>5500</td>
<td>5500</td>
</tr>
<tr>
<td>6</td>
<td>46800</td>
<td>9300</td>
<td>9300</td>
</tr>
<tr>
<td>7</td>
<td>52300</td>
<td>5500</td>
<td>9300;-9500;9500;-11500;-4000;</td>
</tr>
</tbody>
</table>
**Moving Average**

In our next example, we have three variations on a Moving Average. This is similar to what LAG and LEAD will do, looking to Preceding (prior) or Following (next) row values. We will use column aliases called MA1, MA2 and MA3. All three aliases use comparable syntax. The only difference is which rows we want averaged, as shown in this table:

<table>
<thead>
<tr>
<th>Rows characteristic</th>
<th>MA1</th>
<th>MA2</th>
<th>MA3</th>
</tr>
</thead>
<tbody>
<tr>
<td># PRECEDING</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td># FOLLOWING</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td># being averaged</td>
<td>3</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

WITH SGLCODE AS (
  select seq, amt
  from fprpoda
  where po_code in ('BA177629')
    and seq IS NOT NULL )

SELECT   seq, amt,
          avg(amt) OVER (order by seq rows between 1 PRECEDING and 1 FOLLOWING ) ma1,
          avg(amt) OVER (order by seq rows between 0 PRECEDING and 1 FOLLOWING ) ma2,
          avg(amt) OVER (order by seq rows between 1 PRECEDING and 0 FOLLOWING ) ma3
FROM SGLCODE order by seq;

To make more sense of what is happening, consider the amount column (AMT) below in the first row of output. In each case, we include the current row AMT value in our calculation.

MA1 wants 1 PRECEDING, but there is no prior row. Since it wants 1 FOLLOWING, we have
\[
\frac{.01 + 2137.70}{2} = 1068.86
\]

MA2 wants 0 PRECEDING and 1 FOLLOWING, so we have the same result as MA1:
\[
\frac{.01 + 2137.70}{2} = 1068.86
\]

MA3 wants 1 PRECEDING; as we saw with MA1, there is no prior row. Since it wants 0 FOLLOWING, we have
\[
\frac{.01}{1} = .01
\]

From there on down, we have the comparable logic in how it looks for PRECEDING and FOLLOWING values to calculate, along with the current row value.

<table>
<thead>
<tr>
<th>Moving Average: Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>SEQ</td>
</tr>
<tr>
<td>-----</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
</tbody>
</table>
NTILE

Our final example uses manufactured simple data, obviously not a real world example this time! We have 6 rows of test scores, and we are asking for 4 buckets, i.e. we want Quartiles in this example.

We tell the NTILE definition to be ordered DESCENDING, which results in highest test scores being in buckets 1 and 2. If “DESC” were taken out of SQL, ranking would be reversed, i.e. lowest scores would be in the 1st quartile rather than 4th. Two extra values (6/4) are allocated to buckets 1 and 2.

```
NTILE: Example with 4 buckets
SELECT name, score,
       NTILE(4) OVER (ORDER BY score DESC)
       AS quartile
FROM test_scores ORDER BY name;
NAME               SCORE  QUARTILE
-----------------    ------  ------
Barry Bottomly        12         4
Felicity Fabulous     99         1
Felix Fair            41         2
Mildred Middlin       55         2
Paul Poor             24         3
Sharon Swell          86         1
```

CAUTION: DON’T LOOK THROUGH ROSE COLORED GLASSES WHEN USING A VIEW!

As mentioned earlier, analytics are evaluated near the end of the sequence of statement execution: 1) query resultset is returned, 2) analytics are applied, and 3) ORDER BY in the main query is used. This might seem like a minor idea, but you should be aware of this when using a view. You might write a report and have it used for months, and then someone comes back to say there are some wrong answers in there.

In this final section of the paper, we go in depth to understand why we can get different behavior from a view that has analytics inside. To get a handle on this, we will create a test case.

Our new table is called CRS_YR, which is a subset (10886 rows) from the PeopleSoft table that tracks course attendance. For simplicity, we will use Year values, rather than effective dates with granularity down to the day.

Column CRSTYPE is a 3 character code, e.g. SEM for Seminar.

Column CRSMODE is a 2 character code (which happen to be number values), e.g. 17.

Column YR is the Year of attendance, e.g. 2013.

```
CREATE TABLE crs_yr
(crstype varchar2(3), crsmode varchar2(2), YR NUMBER);
Table created.
```

The to_char and to_date conversion is simply to give Year values for our test case. Analytics are not involved yet.

```
INSERT INTO crs_yr
WITH crsyr AS
(select to_date(to_char(effdt,'RRRR'),'RRRR') YR
from sysadm.ps_crse_attendance
GROUP BY to_date(to_char(effdt,'RRRR'),'RRRR')
HAVING COUNT(*) > 20
ORDER BY YR)
SELECT ssr_component, instruction_mode, substr(to_char(effdt,'YYYY'),1,4) YR
FROM sysadm.ps_crse_attendance
WHERE to_date(to_char(effdt,'RRRR'),'RRRR')
```
IN (select YR from crsyr);
commit;

select count(*) from crs_yr;
  COUNT(*)
----------
  10886

create index type_idx on CRS_YR(CRSTYPE);
Index created.

EXEC DBMS_STATS.GATHER_TABLE_STATS ('price', 'CRS_YR')
PL/SQL procedure successfully completed.

What is the question we want to answer with a report?

• We are only interested in Seminars: CRSTYPE of SEM.
• For each row in our test table, show the most recent year for Type and for Mode.

SELECT CRSTYPE, CRSMODE, YR,
  max(YR) OVER (PARTITION BY CRSTYPE) MAX_BY_TYPE,
  max(YR) OVER (PARTITION BY CRSMODE) MAX_BY_MODE
FROM CRS_YR
where CRSTYPE = 'SEM';
/* Showing detail of all rows will not be helpful here. For the moment, seeing the row count is all we need. Out of the 10886 total rows in the table, we get a subset of the table when requesting only the Seminar rows.
*/
3093 rows selected.

Now, we create a view without the CRSTYPE specified as a WHERE clause predicate. This is meant to offer simplicity in reporting, so we can ask for any Course Type, not just Seminar.

CREATE OR REPLACE VIEW CRS_YR_VW
AS
SELECT CRSTYPE, CRSMODE, YR,
  max(YR) OVER (PARTITION BY CRSTYPE) MAX_BY_TYPE,
  max(YR) OVER (PARTITION BY CRSMODE) MAX_BY_MODE
FROM CRS_YR;
View created.

We get the same subset of the table when requesting only the Seminar rows, so that looks promising.

SELECT * FROM CRS_YR_VW
WHERE CRSTYPE = 'SEM';
3093 rows selected.

Using each method (with and without the view) we look at the max year value for Course Type, when we ask for Seminar.
First, the non-view method:

select count(*), MAX_BY_TYPE from (
  SELECT CRSTYPE, CRSMODE, YR,
    max(YR) OVER (PARTITION BY CRSTYPE) MAX_BY_TYPE,
    max(YR) OVER (PARTITION BY CRSMODE) MAX_BY_MODE
  FROM CRS_YR
  where CRSTYPE = 'SEM')
group by MAX_BY_TYPE
order by MAX_BY_TYPE;
  COUNT(*)  MAX_BY_TYPE
----------
1 10886
We get the same answer with the view, so that looks good:

```sql
select count(*), MAX_BY_TYPE from (  
    SELECT * FROM CRS_YR_VW  
    WHERE CRSTYPE = 'SEM')  
group by MAX_BY_TYPE  
order by MAX_BY_TYPE;  
COUNT(*) MAX_BY_TYPE  
---------- -----------  
3093 2013
```

Now we continue the same idea, but look at MAX_BY_MODE, instead of MAX_BY_TYPE.

Using MINUS, we find a few differences on MAX_BY_MODE. The non-view method has some extra values:

```sql
select MAX_BY_MODE from (  
    SELECT CRSTYPE, CRSMODE, YR,  
    max(YR) OVER (PARTITION BY CRSTYPE) MAX_BY_TYPE,  
    max(YR) OVER (PARTITION BY CRSMODE) MAX_BY_MODE  
    FROM CRS_YR  
    where CRSTYPE = 'SEM')  
MINUS  
select MAX_BY_MODE from (  
    SELECT * FROM CRS_YR_VW  
    WHERE CRSTYPE = 'SEM') ;  
MAX_BY_MODE  
----------  
2008 2010 2011 2012
```

With the view, we only get 2013 for our most recent year. We clearly have different results, even though we applied the same WHERE clause. We need to explore more to see which is correct.

```sql
select count(*), MAX_BY_MODE from (  
    SELECT * FROM CRS_YR_VW  
    WHERE CRSTYPE = 'SEM')  
group by MAX_BY_MODE  
order by MAX_BY_MODE;  
COUNT(*) MAX_BY_MODE  
---------- -----------  
3093 2013
```

Without the view, out of 3093 rows, 144 are prior to year 2013:

```sql
select count(*), MAX_BY_MODE from (  
    SELECT CRSTYPE, CRSMODE, YR,  
    max(YR) OVER (PARTITION BY CRSTYPE) MAX_BY_TYPE,  
    max(YR) OVER (PARTITION BY CRSMODE) MAX_BY_MODE  
    FROM CRS_YR  
    where CRSTYPE = 'SEM')  
group by MAX_BY_MODE  
order by MAX_BY_MODE;  
COUNT(*) MAX_BY_MODE  
---------- -----------  
2949 2013
```
We see above that there are only 5 rows with a MAX_BY_MODE value of 2008, so this is a nice subset to examine. We see below that 2008 is the most recent year for this subset, validating results that we see above:

```sql
select MAX_BY_MODE, CRSTYPE, CRSMODE, YR, MAX_BY_TYPE from (  
    SELECT CRSTYPE, CRSMODE, YR,  
    max(YR) OVER (PARTITION BY CRSTYPE) MAX_BY_TYPE,  
    max(YR) OVER (PARTITION BY CRSMODE) MAX_BY_MODE  
    FROM CRS_YR  
    WHERE CRSTYPE = 'SEM')  
where MAX_BY_MODE IN (2008)  
order by YR;
```

<table>
<thead>
<tr>
<th>MAX_BY_MODE</th>
<th>CRSTYPE</th>
<th>CR</th>
<th>YR</th>
<th>MAX_BY_TYPE</th>
</tr>
</thead>
<tbody>
<tr>
<td>2008</td>
<td>SEM</td>
<td>17</td>
<td>2005</td>
<td>2013</td>
</tr>
<tr>
<td>2008</td>
<td>SEM</td>
<td>17</td>
<td>2007</td>
<td>2013</td>
</tr>
<tr>
<td>2008</td>
<td>SEM</td>
<td>17</td>
<td>2008</td>
<td>2013</td>
</tr>
<tr>
<td>2008</td>
<td>SEM</td>
<td>17</td>
<td>2008</td>
<td>2013</td>
</tr>
<tr>
<td>2008</td>
<td>SEM</td>
<td>17</td>
<td>2008</td>
<td>2013</td>
</tr>
</tbody>
</table>

In this case the analytic method is skewing results in some way. We can use the traditional aggregates method to get more information on Type and Mode, for rows that are Seminar ('SEM'). Consistent with what we saw above, the max year for Type is 2013:

```sql
select max(YR), CRSTYPE from CRS_YR where CRSTYPE = 'SEM'  
group by CRSTYPE order by CRSTYPE;  
```

<table>
<thead>
<tr>
<th>MAX(YR)</th>
<th>CRS</th>
</tr>
</thead>
<tbody>
<tr>
<td>2013</td>
<td>SEM</td>
</tr>
</tbody>
</table>

We have a range of years for Mode when we only ask for Seminar. In the 5 rows above, we see that we only had Mode 17. That matches our finding here (second from the last line below, for CRSMODE 17), validating further that MAX_BY_MODE should not be 2013. So our view is not giving an answer we want:

```sql
select max(YR), CRSMODE from CRS_YR where CRSTYPE = 'SEM'  
group by CRSMODE order by CRSMODE;  
```

<table>
<thead>
<tr>
<th>MAX(YR)</th>
<th>CR</th>
</tr>
</thead>
<tbody>
<tr>
<td>2011</td>
<td>02</td>
</tr>
<tr>
<td>2010</td>
<td>05</td>
</tr>
<tr>
<td>2012</td>
<td>07</td>
</tr>
<tr>
<td>2012</td>
<td>08</td>
</tr>
<tr>
<td>2013</td>
<td>09</td>
</tr>
<tr>
<td>2013</td>
<td>11</td>
</tr>
<tr>
<td>2012</td>
<td>12</td>
</tr>
<tr>
<td>2013</td>
<td>14</td>
</tr>
<tr>
<td>2013</td>
<td>15</td>
</tr>
<tr>
<td>2013</td>
<td>16</td>
</tr>
<tr>
<td>2008</td>
<td>17</td>
</tr>
<tr>
<td>2010</td>
<td>18</td>
</tr>
</tbody>
</table>

12 rows selected.
To investigate, let's look at the execution plans. First, the non-view plan. We see that our index on CRSTYPE is used. The Row count of 3093 is an exact match here with what we saw previously for the Seminar CRSTYPE. In Predicate Information, we see the "access" method. As noted earlier, the WHERE clause is applied before any analytics are used.

```sql
SELECT * from table(dbms_xplan.display_cursor(SQL_ID=>'91sb7bv70ujhh'));
```

**SQL_ID 91sb7bv70ujhh, child number 0**

```sql
SELECT CRSTYPE, CRSMODE, YR,
max(YR) OVER (PARTITION BY CRSTYPE) MAX_BY_TYPE,
max(YR) OVER (PARTITION BY CRSMODE) MAX_BY_MODE
FROM CRS_YR where CRSTYPE = 'SEM'
```

Plan hash value: 3840167244

<table>
<thead>
<tr>
<th>Id</th>
<th>Operation</th>
<th>Name</th>
<th>Rows</th>
<th>Bytes</th>
<th>Cost (%CPU)</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>SELECT STATEMENT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>WINDOW SORT</td>
<td></td>
<td>3093</td>
<td>34023</td>
<td>19 (100)</td>
<td>00:00:01</td>
</tr>
<tr>
<td>2</td>
<td>WINDOW BUFFER</td>
<td></td>
<td>3093</td>
<td>34023</td>
<td>19 (27)</td>
<td>00:00:01</td>
</tr>
<tr>
<td>3</td>
<td>TABLE ACCESS BY INDEX ROWID</td>
<td>CRS_YR</td>
<td>3093</td>
<td>34023</td>
<td>15 (7)</td>
<td>00:00:01</td>
</tr>
<tr>
<td>* 4</td>
<td>INDEX RANGE SCAN</td>
<td>TYPE_IDX</td>
<td>3093</td>
<td></td>
<td>2 (0)</td>
<td>00:00:01</td>
</tr>
</tbody>
</table>

Predicate Information (identified by operation id):
----------------------------------------------

4 - access("CRSTYPE"='SEM')

Now compare that with the plan for the view, using the same WHERE clause:

```sql
SELECT * from table(dbms_xplan.display_cursor(SQL_ID=>'cz1hjfhfbwn9d'));
```

**SQL_ID cz1hjfhfbwn9d, child number 0**

```sql
SELECT * FROM CRS_YR_VW WHERE CRSTYPE = 'SEM'
```

Plan hash value: 2520468137

<table>
<thead>
<tr>
<th>Id</th>
<th>Operation</th>
<th>Name</th>
<th>Rows</th>
<th>Bytes</th>
<th>Cost (%CPU)</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>SELECT STATEMENT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>* 1</td>
<td>VIEW</td>
<td>CRS_YR_VW</td>
<td>10886</td>
<td>478K</td>
<td>40 (100)</td>
<td>00:00:01</td>
</tr>
<tr>
<td>2</td>
<td>WINDOW SORT</td>
<td></td>
<td>10886</td>
<td>116K</td>
<td>40 (75)</td>
<td>00:00:01</td>
</tr>
<tr>
<td>3</td>
<td>WINDOW SORT</td>
<td></td>
<td>10886</td>
<td>116K</td>
<td>40 (75)</td>
<td>00:00:01</td>
</tr>
<tr>
<td>4</td>
<td>TABLE ACCESS FULL</td>
<td></td>
<td>10886</td>
<td>116K</td>
<td>14 (29)</td>
<td>00:00:01</td>
</tr>
</tbody>
</table>

Predicate Information (identified by operation id):
----------------------------------------------

1 - filter("CRSTYPE"='SEM')

The index is not used for the view. In Predicate Information, we have a filter on the view, not "access" on the table. The view does not have any WHERE clause. The execution plan shows that there is nothing to apply before analytics, and no reason for the index to be used. When scanning the entire table (based on the view definition with no WHERE clause), the max value for Year is 2013. Any filtering on any CRSTYPE value becomes meaningless at that point, because it only sees a single answer of 2013, for any row in the table.
Note that we did get a valid answer for CRSTYPE. Our SQL statement has PARTITION BY on that column. Our test case can show that we get the same answer with or without the view. When the PARTITION BY and WHERE clauses use the same column, we see the same result whether the predicate is applied first or not.

To show more detail on CRSTYPE, we first show all possible Type values, with most recent year:

```sql
select distinct MAX_BY_TYPE, CRSTYPE from (  
  SELECT CRSTYPE, CRSMODE, YR,  
      max(YR) OVER (PARTITION BY CRSTYPE) MAX_BY_TYPE,  
      max(YR) OVER (PARTITION BY CRSMODE) MAX_BY_MODE  
  FROM CRS_YR)  
order by CRSTYPE;
```

<table>
<thead>
<tr>
<th>MAX_BY_TYPE CRS</th>
</tr>
</thead>
<tbody>
<tr>
<td>2013 DIS</td>
</tr>
<tr>
<td>2013 FLD</td>
</tr>
<tr>
<td>2013 FLI</td>
</tr>
<tr>
<td>2010 FLS</td>
</tr>
<tr>
<td>2013 IND</td>
</tr>
<tr>
<td>2013 LAB</td>
</tr>
<tr>
<td>2009 LBS</td>
</tr>
<tr>
<td>2013 LEC</td>
</tr>
<tr>
<td>2013 SEM</td>
</tr>
<tr>
<td>2009 STI</td>
</tr>
<tr>
<td>2007 STS</td>
</tr>
<tr>
<td>2013 STU</td>
</tr>
</tbody>
</table>

12 rows selected.

To show that CRSTYPE can be valid regardless of the predicate value, we'll use the 'LBS', rather than SEM that we had used previously. First, look at the non-view variety:

```sql
SELECT CRSTYPE, CRSMODE, YR,  
      max(YR) OVER (PARTITION BY CRSTYPE) MAX_BY_TYPE,  
      max(YR) OVER (PARTITION BY CRSMODE) MAX_BY_MODE  
FROM CRS_YR  
WHERE CRSTYPE = 'LBS'  
ORDER BY YR;
```

<table>
<thead>
<tr>
<th>CRS CR</th>
<th>YR</th>
<th>MAX_BY_TYPE</th>
<th>MAX_BY_MODE</th>
</tr>
</thead>
<tbody>
<tr>
<td>LBS 07</td>
<td>1993</td>
<td>2009</td>
<td>2009</td>
</tr>
<tr>
<td>LBS 08</td>
<td>1997</td>
<td>2009</td>
<td>2006</td>
</tr>
<tr>
<td>LBS 07</td>
<td>2005</td>
<td>2009</td>
<td>2009</td>
</tr>
<tr>
<td>LBS 08</td>
<td>2006</td>
<td>2009</td>
<td>2006</td>
</tr>
<tr>
<td>LBS 07</td>
<td>2007</td>
<td>2009</td>
<td>2009</td>
</tr>
<tr>
<td>LBS 07</td>
<td>2008</td>
<td>2009</td>
<td>2009</td>
</tr>
<tr>
<td>LBS 07</td>
<td>2009</td>
<td>2009</td>
<td>2009</td>
</tr>
</tbody>
</table>

7 rows selected.

Using 'LBS' again, the view has the different result for MAX_BY_MODE, but MAX_BY_TYPE consistently gives 2009 as the answer:

```sql
SELECT * FROM CRS_YR_VW  
WHERE CRSTYPE = 'LBS'  
ORDER BY YR;
```

<table>
<thead>
<tr>
<th>CRS CR</th>
<th>YR</th>
<th>MAX_BY_TYPE</th>
<th>MAX_BY_MODE</th>
</tr>
</thead>
<tbody>
<tr>
<td>LBS 07</td>
<td>1993</td>
<td>2009</td>
<td>2013</td>
</tr>
<tr>
<td>LBS 08</td>
<td>1997</td>
<td>2009</td>
<td>2013</td>
</tr>
</tbody>
</table>
All of this is to show that you'll want to be cautious about using analytics in a view. That can change your results, compared with using analytics without a view, depending on which columns are used. And even if you get valid results with a view, you can have a performance problem. As shown in the execution plans above, the view did a full table scan and then applied a filter. On a large table, that could become a noticeable runtime problem.

CONCLUSION

Analytic Functions are not a magic answer to all SQL problems, but they have great potential to put in your toolkit for greater flexibility and performance. Don't keep analytics a secret, spread the word!

APPENDICES

Excerpts from Ask Tom regarding differing results from analytics in a view:
http://asktom.oracle.com/pls/apex/f?p=100:11:0::NO::P11_QUESTION_ID:3469884600671 posting from April 19, 2002
"The predicate will always be applied AFTER the analytics buried in the view, they must be."

"Analytics are applied after predicates. The view -- it has no predicate. The query -- it has a predicate. You'll find that you have DIFFERENT result sets."
[...]
"Note that when the predicate CAN be pushed, it most certainly is."

REFERENCES


WITH clause (Query Subfactoring, aka Common Table Expression) in SELECT statement documentation:
http://docs.oracle.com/cd/E16655_01/server.121/e17209/statements_10002.htm#SQLRF56374