

# EXCITING SQL & PL/SQL NEW FEATURES

## Introduction

This paper is to explore various new features introduced in recent releases of Oracle 9i and 10g. These features are not commonly known and this paper is to explore these new features. SQL tuning is also possible using these new features.

This paper is designed to provide an outline of features. Not an exhaustive educational material. Scripts are provided wherever possible to help improve the understanding of these features.

## Subquery factoring

Subquery factoring is a SQL feature in which, results of a subquery can be queried as if the subquery is another row source. This feature is implemented using 'with' SQL clause. Performance wise, this query is useful as a complex subquery need to be evaluated once and can be queried in the later part of the SQL, as if we are querying a pre-built table.

```

WITH
    dept_avg as                                ..1
    (select deptno, avg(sal) avg_sal from emp group by deptno ),
    all_avg as                                  ..2
    (select avg(avg_sal) avg_sal from dept_avg )
select d.deptno, d_avg.avg_sal , a_avg.avg_sal
from
    dept d,
    dept_avg d_avg,                            ..3
    all_avg a_avg
where
    d_avg.avg_sal >= a_avg.avg_sal and
    d.deptno=d_avg.deptno;

```

In the SQL above,

Line 1 defines subquery dept\_avg. Line 2 defines a subquery all\_avg and all\_avg subquery queries the dept\_avg defined in line 1. Subsequent lines are joining a table dept with these two row sources defined earlier as dept\_avg and all\_avg. Notice that the subqueries defined earlier are referred as if they are table row sources.

Let us reveal what goes on underneath what meets eye. Explain plan for this query almost spills out underlying operations:

Explain plan :

Id	Operation	Name	Rows	Bytes	
0	SELECT STATEMENT		1	52	
1	TEMP TABLE TRANSFORMATION				..1
2	LOAD AS SELECT				
3	SORT GROUP BY		14	364	
4	TABLE ACCESS FULL	EMP	14	364	
* 5	HASH JOIN		1	52	
6	NESTED LOOPS		1	39	
7	VIEW		1	13	
8	SORT AGGREGATE		1	13	
9	VIEW		14	182	
10	TABLE ACCESS FULL	SYS_TEMP_0FD9D6604_740118	14	364	..2
* 11	VIEW		1	26	
12	TABLE ACCESS FULL	SYS_TEMP_0FD9D6604_740118	14	364	
13	TABLE ACCESS FULL	DEPT	4	52	

Figure 1: Explain plan for subquery factoring

In Figure 1, line 1 has temp table transformation. Indeed, Oracle is materializing the subquery, creating a temporary table like object. Line 2 shows the name of the temporary object as SYS\_TEMP\_0FD9D6604\_740118.

To explain it a bit further, to materialize the dept\_avg subquery, Oracle created a temporary table and that temporary table was queried again to create the row source for all\_avg subquery. This temporary table seems to exist during query execution and disappears at the end of query execution.

Turning on sqltrace confirms the above observation. Following lines are from the trace file. This shows that a new global temporary table is created on the fly and accessed later to create subsequent row sources:

```
CREATE GLOBAL TEMPORARY TABLE "SYS"."SYS_TEMP_0FD9D6605_740118" ("C0"
NUMBER(2),"C1" NUMBER) IN_MEMORY_METADATA CURSOR_SPECIFIC_SEGMENT
STORAGE (OBJNO 4254950917) NOPARALLEL
```

There are few exotic options in these lines such as 'in\_memory\_metadata' and 'cursor\_specific\_segment' etc. References to these options are not readily available, but we could guess that cursor\_specific\_segment implies that this segment exists during the execution of this cursor.

[Script: subquery\\_factoring.sql](#)

### **Subquery factoring and dblink**

Subquery factoring is useful even if the subquery is accessing the remote tables over the database link. If the remote table is accessed more than once, then this feature is quite handy for tuning performance of the query, as rows from remote database can be materialized.

Following query uses loopback database link to access emp table.

```
WITH dept_avg as
  (select deptno, avg(sal) avg_sal from emp@loopback group by deptno ),
all_avg as
  (select avg(avg_sal) avg_sal from dept_avg )
select d.deptno, d_avg.avg_sal , a_avg.avg_sal
  from
    dept d,
    dept_avg d_avg,
    all_avg a_avg
where
  d_avg.avg_sal >= a_avg.avg_sal and
  d.deptno=d_avg.deptno
;
```

Step 1 belows shows that temp table is created to materialize data from remote table and is used subsequently.

Id	Operation	Name	Rows	Bytes
0	SELECT STATEMENT		1	42
1	TEMP TABLE TRANSFORMATION			
2	LOAD AS SELECT			
3	SORT GROUP BY		3	78
4	REMOTE		14	364
* 5	HASH JOIN		1	42
6	NESTED LOOPS		1	39
7	VIEW		1	13
8	SORT AGGREGATE		1	13
9	VIEW		3	39
10	TABLE ACCESS FULL	SYS_TEMP_0FD9D6602_8103BDC3	3	78
* 11	VIEW		1	26
12	TABLE ACCESS FULL	SYS_TEMP_0FD9D6602_8103BDC3	3	78
13	TABLE ACCESS FULL	DEPT	4	12

-----  
 Remote SQL Information (identified by operation id):  
 -----

4 - SELECT "SAL","DEPTNO" FROM "EMP" "EMP" (accessing 'LOOPBACK' )

**Script:** subquery\_factoring\_dbblink.sql

### **Subquery factoring and materialize hint**

While line from the trace file confirms that a global temporary table was created in the aforementioned SQL, Oracle can decide NOT to materialize the subquery.

Inspect the following SQL:

```
with
  dept_avg as
    (select deptno, avg(sal) avg_sal from emp group by deptno ),
  all_avg as
    (select avg(sal) avg_sal from emp )
select d.deptno, d_avg.avg_sal , a_avg.avg_sal
from
  dept d,
  dept_avg d_avg,
  all_avg a_avg
where
  d_avg.avg_sal >= a_avg.avg_sal and
  d.deptno=d_avg.deptno;
```

Difference between this query and earlier query is that, in this query we are not referring to dept\_avg from all\_avg subquery. Instead all\_avg subquery directly queries the emp table. Explain plan for the above query shows that step TEMP\_TABLE\_TRANSFORMATION is missing. Oracle did not materialize the subquery in this case.

Id	Operation	Name	Rows	Bytes
0	SELECT STATEMENT		1	52
* 1	HASH JOIN		1	52
2	NESTED LOOPS		1	39
3	VIEW		1	13
4	SORT AGGREGATE		1	13
5	TABLE ACCESS FULL	EMP	14	182
* 6	VIEW		1	26
7	SORT GROUP BY		14	364
8	TABLE ACCESS FULL	EMP	14	364
9	TABLE ACCESS FULL	DEPT	4	52

It is possible to materialize the subquery using an undocumented and unsupported hint:  
 /\*+ materialize \*/ (Thanks to Jonathan Lewis)

SQL:

```
With dept_avg as
  (select /*+ materialize */ deptno, avg(sal) avg_sal from emp group by deptno ),
...
```

Id	Operation	Name	Rows	Bytes
0	SELECT STATEMENT		1	52
1	TEMP TABLE TRANSFORMATION			
2	LOAD AS SELECT			
3	SORT GROUP BY		14	364
4	TABLE ACCESS FULL	EMP	14	364
5	LOAD AS SELECT			
6	SORT AGGREGATE		1	13
7	TABLE ACCESS FULL	EMP	14	182
* 8	HASH JOIN		1	52
9	NESTED LOOPS		1	39
10	VIEW		1	13
11	TABLE ACCESS FULL	SYS_TEMP_0FD9D660D_740118	1	13
* 12	VIEW		1	26
13	TABLE ACCESS FULL	SYS_TEMP_0FD9D660C_740118	14	364
14	TABLE ACCESS FULL	DEPT	4	52

Above explain plan proves that these queries are indeed materialized as the ‘TEMP TABLE TRANSFORMATION’ step is present. For read only databases, temporary tables are not created even if the hint is provided.

[Script](#): subquery\_factoring\_with\_hint.sql

In essence, a complex SQL subquery can be written with better performance using this subquery factoring feature.

### Connect by enhancements

Prior to Oracle versions 9i, ‘connect by’ clause always resulted in nested loops join. Other more suitable join techniques were never used [at least, in author’s experience]. Oracle 9i removes that restrictions and other suitable join techniques are also considered and used.

In addition to that, a new step also appears in the ‘connect by’ query explain plans.

We will consider the following SQL to test this new feature. This query is accessing emp table connecting by empno and mgr columns.

```
select lpad(' ',2*level-1 ,ename) ename
from emp
connect by prior empno = mgr
start with mgr is null
/
```

### Explain plan for Oracle version 8i:

Execution Plan

```
0 SELECT STATEMENT Optimizer=CHOOSE
1 0 CONNECT BY
2 1 TABLE ACCESS (FULL) OF 'EMP'
3 1 TABLE ACCESS (BY USER ROWID) OF 'EMP'
4 1 TABLE ACCESS (FULL) OF 'EMP'
```

Statistics

```
0 recursive calls
180 db block gets
89 consistent gets
0 physical reads
0 redo size
692 bytes sent via SQL*Net to client
358 bytes received via SQL*Net from client
2 SQL*Net roundtrips to/from client
0 sorts (memory)
0 sorts (disk)
14 rows processed
```

From the above statistics, we can see that 269 buffer gets consumed to execute this SQL.

### Explain plan for Oracle version 10g:

Following figure shows the explain plan for Oracle version 10g. A new step 'connect by pump' is introduced and HASH JOIN technique used. Performance of this query is greatly improved reducing the buffer gets to just 15 instead of 269 buffer gets as in release 8i.

#### Execution Plan

```
-----
0      SELECT STATEMENT Optimizer=ALL_ROWS (Cost=2 Card=14 Bytes=196)
1      0      CONNECT BY (WITH FILTERING)
2      1      FILTER
3      2      TABLE ACCESS (FULL) OF 'EMP' (TABLE) (Cost=2 Card=14 Bytes=196)
4      1      HASH JOIN
5      4      CONNECT BY PUMP
6      4      TABLE ACCESS (FULL) OF 'EMP' (TABLE) (Cost=2 Card=14 Bytes=196)
7      1      TABLE ACCESS (FULL) OF 'EMP' (TABLE) (Cost=2 Card=14 Bytes=196)
```

#### Statistics

```
-----
1 recursive calls
0 db block gets
15 consistent gets
0 physical reads
0 redo size
716 bytes sent via SQL*Net to client
664 bytes received via SQL*Net from client
2 SQL*Net roundtrips to/from client
5 sorts (memory)
0 sorts (disk)
14 rows processed
```

Interestingly, this feature is available in 8i also, but disabled by default. Setting an undocumented (and so unsupported) initialization parameter `_new_connect_by_enabled` to true will enable this feature in 8i. Also, this feature can be disabled in 10g, by setting an undocumented initialization parameter `_old_connect_by_enabled` to true. This parameter also can be modified at the session level using 'alter session' sql statement.

While this feature generally improves the performance, there are few cases, in which this feature hinders performance. If there is a need to disable this feature, then it is recommended to disable at the session level, instead of instance level.

### New functions

A new function `sys_connect_by_path` is introduced. This function is useful in retrieving full hierarchy of data. Values are connected by the second argument passed to the function:

```
select lpad(' ',2*level-1) || sys_connect_by_path (ename, '/') ename
from emp
connect by prior empno = mgr ;
```

ename

```
-----
/king
 /king/jones
  /king/jones/scott
   /king/jones/scott/adams
    /king/jones/ford
     /king/jones/ford/smith
 /king/blake
  /king/blake/allen
   /king/blake/ward
    /king/blake/martin
     /king/blake/turner
      /king/blake/james
 /king/clark
  /king/clark/miller
```

### New pseudo columns

Few pseudo columns are also introduced in Oracle 10g improving 'connect by' clause usability.

#### Cycle issue and Connect by iscycle

In Oracle 8i, cyclic relationship in data causes error ORA-1436 in 'connect by' queries. Emp table is used here to illustrate this issue. From the following SQL output, CLARK is the manager of MILLER. There is no cyclic relationship in data, yet.

```
select empno, mgr, ename from emp where ename in ('MILLER','CLARK')
```

```
/
  EMPNO      MGR ENAME
-----
   7782      7839 CLARK
   7934      7782 MILLER
```

So, following SQL works fine as there is no cyclic relationship.

```
select empno, mgr, ename from emp
connect by prior empno = mgr
start with empno=7782
```

```
/
  EMPNO      MGR ENAME
-----
   7782      7839 CLARK
   7934      7782 MILLER
  ...
```

Let's update this table to create a cyclic relationship.

```
update emp set mgr=7934 where ename='CLARK';
```

```
select empno, mgr, ename from emp where ename in ('MILLER','CLARK');
```

```
  EMPNO      MGR ENAME
-----
   7782      7934 CLARK
   7934      7782 MILLER
```

After the update, we can see that there is a cyclic relationship between these two rows. CLARK is the manager of MILLER, who is a manager of CLARK. We will execute the query, in 8i version of the database.

```
select empno, mgr, ename from emp
connect by prior empno = mgr
start with empno=7782
```

```
SQL> /
```

```
ERROR:
```

```
ORA-01436: CONNECT BY loop in user data
```

This error can be avoided in 10g version of the database using a new keyword nocycle. Further it is possible to see whether there is any cyclic relationship or not, using a new 'connect\_by\_iscycle' keyword. Value of 0 for that expression indicates that there is no cyclic relationship in that row and value of 1 indicates that there exists a cyclic relationship.

```

Select lpad(' ',2*level-1) || sys_connect_by_path (ename, '/') ename, connect_by_iscycle
from emp
connect by nocycle prior empno = mgr
start with empno=7782;

```

ENAME	CONNECT_BY_ISCYCLE
/CLARK	0
/CLARK/MILLER	1

Second row shows that connect\_by\_iscycle is 1 indicating that there is a cyclic relationship. Keyword nocycle resolve the ORA-1436 error in 9i/10g.

### Connect by isleaf

Connect\_by\_isleaf pseudo column shows whether a specific row is a leaf row or not. Leaf row in a hierarchical query is defined as node without any children.

```

Select
lpad(' ',2*level-1) || sys_connect_by_path (ename, '/') ename, connect_by_isleaf
from emp
connect by prior empno = mgr
start with mgr is null;

```

ENAME	CONNECT_BY_ISLEAF
/KING	0
/KING/JONES	0
/KING/JONES/SCOTT	0
/KING/JONES/SCOTT/ADAMS	1 ← Leaf row
/KING/JONES/FORD	0
/KING/JONES/FORD/SMITH	1
/KING/BLAKE	0
/KING/BLAKE/ALLEN	1
/KING/BLAKE/WARD	1
/KING/BLAKE/MARTIN	1
/KING/BLAKE/TURNER	1
/KING/BLAKE/JAMES	1

Value of zero in connect\_by\_isleaf column indicates that row is not a leaf row and 1 indicates that row is a leaf row.

## 'Model' feature

Model is a new SQL feature introduced that emulates a spreadsheet like functionality. This feature is very powerful and code is compact thanks mostly to its features. Just like a cell can be independently referred in an excel spreadsheet a value can be independently referred and complex calculations possible. We will consider few examples to explain this new feature.

### Example: Inventory report

Inspect the following table and data. Requirement is to create an inventory report. We will program an SQL for this report using model clause. This table is to hold sales and receipt quantities for a product, location and week. This is a fact table and item, location and week are dimensions to this fact table. In our example we take just 10 weeks of data.

```
CREATE TABLE ITEM_DATA
(
  ITEM          VARCHAR2(10 BYTE),
  LOCATION     NUMBER,
  WEEK         NUMBER,
  SALES_QTY    NUMBER,
  RCPT_QTY     NUMBER
) ;
```

ITEM	LOCATION	WEEK	SALES_QTY	RCPT_QTY
...				
Shirt	1	1	623	55
Shirt	1	2	250	469
Shirt	1	3	882	676
Shirt	1	4	614	856
Shirt	1	5	401	281
Shirt	1	6	163	581
Shirt	1	7	324	415
Shirt	1	8	409	541
Shirt	1	9	891	790
Shirt	1	10	759	465
Shirt	2	1	261	515
Shirt	2	2	664	67
...				

Further, formula to calculate the inventory is

$$\text{Inventory} = \text{Last week inventory} + \\ \text{This week receipts} - \\ \text{This week sales}$$

Above formula alludes that we need to access prior row to calculate current row's inventory. Current week inventory is last week inventory plus quantity received this week and quantity sold this week, per item, location and week. Accessing prior row in Oracle versions prior to 8i is possible only self join of table. This self join is inefficient and can lead to performance issues. Model feature allows us to access the prior week values with compact code:

```
select item, location, week, inventory, sales_qty, rcpt_qty.....1
from item_data
model return updated rows.....2
partition by (item).....3
dimension by (location, week).....4
measures ( 0 inventory , sales_qty, rcpt_qty).....5
rules (.....6
inventory [location, week]= nvl(inventory[cv(location), cv(week)-1],0) .....7
- sales_qty [cv(location), cv(week) ] + .....8
+rcpt_qty [cv(location), cv(week) ]
)
order by item , location, week;
```

We will explain the above SQL, line by line.

1. inventory is the derived column we calculate in this SQL. Even though there is a column with same name in the table, we are referring to a derived column, discussed below.
2. model keyword indicates that this SQL uses certain structures of model clause.
3. partition by (item) indicates that data to be partitioned by item column values. Oracle will partition the data using this column. This data partitioning is much different from Oracle table partitioning. This is analogous to a sheet in Excel workbook, each sheet for a specific item, say pants, shirts etc.

4. dimension by (location, week). These are the dimensions of this spreadsheet. This is analogous to excel spreadsheet's x and y coordinates.
5. measures indicates various derived columns that we will refer in the following rules.
- 7-8. Here are the rules defined. In this case, inventory for a week is (last week inventory + receipt qty for the current week - sales qty for current week.).Formula becomes, then,

$$\text{inventory [location, week]} = \text{nvl}(\text{inventory}[\text{cv}(\text{location}), \text{cv}(\text{week})-1], 0) - \text{sales\_qty} [\text{cv}(\text{location}), \text{cv}(\text{week}) ] + \text{rcpt\_qty} [\text{cv}(\text{location}), \text{cv}(\text{week}) ]$$

In this formula, second line is accessing the inventory from the prior row. Third line accesses the sales\_qty from current row. Fourth line accesses the rcpt\_qty from current row. Cv function is used to refer to the current value from the left side of the equation.

For e.g. for the location 10, week 6 inventory calculations will be:

$$\text{Inventory [ 10, 6]} = \text{inventory}[10, 5 ] + \text{sales\_qty} [10,6] + \text{receipt\_qty} [10,6]$$

With above data condition cv(location) will return 10 and cv(week ) will return 6 in the right side of equation. Note CV function can only be used in the right side of equation. Output of the above SQL is:

ITEM	LOCATION	WEEK	INVENTORY	SALES_QTY	RCPT_QTY
Pants	1	1	33	199	232
Pants	1	2	387	433	787
Pants	1	3	213	888	714
Pants	1	4	884	165	836 ← 884 = 213 +836 -165
Pants	1	5	791	338	245
Pants	1	6	879	611	699
Pants	1	7	510	858	489
Pants	1	8	1316	37	843
Pants	1	9	1325	778	787
Pants	1	10	1859	140	674
Pants	2	1	-21	697	676
Pants	2	2	-172	263	112

Refer to the highlighted row above,

$$\text{For pants, inventory [1,4]} = \text{inventory} [1,3] + \text{rcpt\_qty} [1,3] - \text{sales\_qty} [1,3]$$

$$884 = 213 + 836 - 165$$

In above formula, [1,4] is for location =1 and 4<sup>th</sup> week.

**Script:** model\_example2.sql

### **Example: Running sales qty total report**

Similar to the previous example, we will attempt to calculate running sales quantity and running rcpt quantity total. Following listing shows the SQL:

```
select item, location, week, inventory, sales_qty, rcpt_qty , sales_so_for, rcpt_so_for
from item_data
model return updated rows
partition by (item)
dimension by (location, week)
measures ( 0 inventory , sales_qty, rcpt_qty, 0 sales_so_for, 0 rcpt_so_for )
rules (
inventory [location, week] = nvl(inventory [cv(location), cv(week)-1 ] ,0)
- sales_qty [cv(location), cv(week) ] +
+ rcpt_qty [cv(location), cv(week) ],
```

```

sales_so_for [location, week] = sales_qty [cv(location), cv(week)] +
                               nvl(sales_so_for [cv(location), cv(week)-1],0),
rcpt_so_for [location, week] = rcpt_qty [cv(location), cv(week)] +
                               nvl(rcpt_so_for [cv(location), cv(week)-1],0)
)
order by item , location, week ;

```

We discussed inventory rule already. We are adding two more rules to our SQL to calculate running total for sales\_qty and rcpt\_qty columns. Highlighted rules are to calculate sales\_qty and rcpt\_qty for a location and product. Formula implemented is

Sales\_so\_for for this location, week = sales\_qty in this week +  
Sales\_so\_for for this location and previous week

So, formula becomes

$$\text{sales\_so\_for [location, week]} = \text{sales\_qty [cv(location), cv(week)]} + \dots\dots\dots 1$$

$$\text{nvl(sales\_so\_for [cv(location), cv(week)-1],0)} \dots\dots\dots 2$$

1. cv(location) refers to the value of location from the left side of equation. Sales\_qty [cv(location), cv(week) ] refers to the sales\_qty value from the current row.
2. nvl(sales\_so\_for [cv(location), cv(week)-1],0) refers to the prior row's sales\_so\_for column value. If this is the first week for a product and location combination, then accessing prior row returns null value. Nvl function returns 0 if null is returned. For e.g. for location 10 and week 1, formula is

$$\text{sales\_so\_for [location, week]} = \text{sales\_qty [cv(location), cv(week)]} +$$

$$\text{nvl(sales\_so\_for [cv(location), cv(week)-1],0),}$$

$$\text{sales\_so\_for [ 10, 1]} = \text{sales\_qty [ 10, 1 ]} +$$

$$\text{nvl ( sales\_so\_for [ 10, 0]. 0 )}$$

sales\_so\_for [ 10, 0 ] accesses a non-existent row and so null value returned. NVL function masks that by returning zero value.

Similar formula applied to rcpt\_qty column too.

Script: model\_example3.sql

### Example: Moving sales avg report

Complex calculations are just a matter of using correct formula with model. Here is another example for the use of model clause. In this, we calculate moving average of sales from current sales qty and previous two weeks sales qty per location and product.

```

select item, location, week, sales_qty, rcpt_qty , moving_sales_avg
from item_data
model return updated rows
partition by (item)
dimension by (location, week)
measures ( 0 moving_sales_avg , sales_qty, rcpt_qty)
rules (
moving_sales_avg [location, week] =
(sales_qty [cv(location), cv(week)]+
nvl(sales_qty [cv(location), cv(week)-1] , sales_qty [cv(location), cv(week)] ) +
nvl(sales_qty [cv(location), cv(week)-2] ,sales_qty [cv(location), cv(week)] ) )
/3
) order by item, location, week;

```

Refer to the highlighted section of SQL above. Formula used in the above example is

Moving sales avg for this location, week =  

$$(\text{sales qty for this week} + \text{sales qty for last week} + \text{sales qty for 2 weeks ago}) / 3$$

In essence, it is sum of current week and past two week's sales qty divided by 3. For first two weeks for la location, null values are returned. NVL function returns current week's value if the prior week sales\_qty is null. So, the formula becomes

```
moving_sales_avg [location, week] =
  (sales_qty [cv(location), cv(week)]+ .....1
   nvl(sales_qty [cv(location), cv(week)-1 ] , sales_qty [cv(location), cv(week)]) + .....2
   nvl(sales_qty [cv(location), cv(week)-2 ] ,sales_qty [cv(location), cv(week)]) )/3.....3
```

1. Refers to current week sales\_qty
2. Accesses the last week sales\_qty by subtracting 1 from current value of week from the left side. If null, then current week sales\_qty itself is used to find average.
3. Accesses the last week sales\_qty by subtracting 2 from current value of week from the left side. If null, then current week sales\_qty itself is used to find average.

Referring to non-existent values will return null values. NVL function is used to handle those null values in boundary condition.

[Script](#): model\_example4.sql

### Explain plan

Explain plan for model clause shows a new keyword MODEL.

```
Rows      Row Source Operation
-----
1500      SORT ORDER BY (cr=9 pr=0 pw=0 time=156936 us)
1500      SQL MODEL ORDERED (cr=9 pr=0 pw=0 time=102891 us)
1500      TABLE ACCESS FULL OBJ#(29061) (cr=9 pr=0 pw=0 time=85 us)
```

### Analytic functions

Analytic functions are new type of functions introduced in 9i with extensive features such as ability to access other rows, partition rows, row ordering etc. We will explore these functions with examples in this section.

#### Example: Running sales total query

We will reuse item\_data table we discussed in model section. For easier reference, table structure is printed here also. This is a fact table and item, location and week are dimensions to this fact table. In our example we take just 10 weeks of data.

In this example, we will create an SQL to query and track running total of sales\_qty column per item, location and week combinations. This running total must reset to zero at item/location boundaries.

```
CREATE TABLE ITEM_DATA
(
  ITEM      VARCHAR2(10 BYTE),
  LOCATION  NUMBER,
  WEEK      NUMBER,
  SALES_QTY NUMBER,
  RCPT_QTY  NUMBER
```

);

ITEM	LOCATION	WEEK	SALES_QTY	RCPT_QTY
...				
Shirt	1	1	623	55
Shirt	1	2	250	469
Shirt	1	3	882	676
Shirt	1	4	614	856
Shirt	1	5	401	281
Shirt	1	6	163	581
Shirt	1	7	324	415
Shirt	1	8	409	541
Shirt	1	9	891	790
Shirt	1	10	759	465
Shirt	2	1	261	515
Shirt	2	2	664	67
...				

SQL:

```
select item, location, week, sales_qty, rcpt_qty,
sum (sales_qty) over (partition by item, location order by week.....1
                      rows between unbounded preceding and current row.....2
                      ) running_sales_total.....3
from item_data;
```

Above SQL calculates the running total of sales\_qty so far for this item and location.

1. sum(sales\_qty) is capturing the sum of sales\_qty column. Following SQL clause 'over(partition by item, location order by week' partitioning the rows by item and location and then sorts the rows by week column.
2. SQL clause 'rows between unbounded preceding and current row' delimits the rows that will be used for summing up. Sum function discussed in (1) is applied over the set of rows delimited by this condition. 'Unbounded preceding' means that rows from first row for this item and location combination.
3. running\_sales\_total is the column alias for this derived column.

Output of the SQL is:

ITEM	LOCATION	WEEK	SALES_QTY	RCPT_QTY	RUNNING_SALES_TOTAL
..					
Pants	1	1	638	524	638
Pants	1	2	709	353	1347
Pants	1	3	775	734	2122
Pants	1	4	344	879	2466
Pants	1	5	990	744	3456
Pants	1	6	635	409	4091
Pants	1	7	524	801	4615
Pants	1	8	14	593	4629
Pants	1	9	630	646	5259
Pants	1	10	61	241	5320
Pants	2	1	315	737	315
Pants	2	2	176	282	491
Pants	2	3	999	314	1490
..					

In the above output, last column running\_sales\_total shows the running total of sales\_qty column. For e.g. for item='PANTS', location=1 and week=6 running total of sales\_qty (4091) is sum of values from week 1 through week 6 (638 + 709 + 775 + 344 + 990 + 635), for the same item='PANTS', location=1.

Also note that, for the next location running\_sales\_total column reset to 315, just sales\_qty from that row.

**Example: Moving average sales\_qty query**

In this example, we calculate the average of sales quantity for the current and past two weeks.

```
select item, location, week, rcpt_qty, sales_qty,
       avg(sales_qty) over ( partition by item, location order by week.....1
                           rows between 2 preceding and current row.....2
                           ) moving_sales_avg.....3
from item_data
```

1. Calculates average of sales\_qty by calling avg function. Partition by clause partitions the rows by item and location.
2. SQL clause 'rows between 2 preceding and current row' delimits the rows the avg function operate as current row and 2 preceding row. Rows are partitioned by item, location and ordered by week. Accessing 2 preceding row is equivalent to accessing the prior 2 weeks' sales\_qty.
3. moving\_sales\_avg is the alias for this derived column.

ITEM	LOCATION	WEEK	RCPT_QTY	SALES_QTY	MOVING_AVG
...					
Pants	1	1	524	638	638.00
Pants	1	2	353	709	673.50
Pants	1	3	734	775	707.33
Pants	1	4	879	344	609.33
Pants	1	5	744	990	703.00
Pants	1	6	409	635	656.33
Pants	1	7	801	524	716.33
Pants	1	8	593	14	391.00
Pants	1	9	646	630	389.33
Pants	1	10	241	61	235.00
Pants	2	1	737	315	315.00
Pants	2	2	282	176	245.50
Pants	2	3	314	999	496.67
...					

In the above example, last column moving\_avg is the sum of rcpt\_qty column values from current row and prior rows. For example, for item='PANTS', location=1 and week =5 moving\_avg value of 703 is average of (775, 344, 990) which are prior two weeks and current week sales\_qty column values.

**Function : lag**

Lag is an analytic function that allows us to access the values from prior rows in an ordered set of rows. Lag can access any prior row within that data partition and by default accesses the last row. Consider the following SQL:

```
select item, location, week, sales_qty, rcpt_qty,
       lag(sales_qty) over ( partition by item, location order by week ) sales_prior_week
from item_data;
```

In the above SQL, second line uses a function lag. This line retrieves the sales\_qty from prior week. As discussed earlier, 'partition by item, location' clause specifies the data to be partitioned per item, location and ordered by week column. Lag returns null value if the referred row does not exists. For example, for the first row lag(sales\_qty) will return null values. In this following output, we can see that sales\_prior\_week column retrieves values from prior row's sales\_qty.

In this output below, sales\_prior\_week column value for item='PANTS', location=2, week=2 value of 453 is derived from prior week row ( item='PANTS', location=2 and week=1 ) sales\_qty. First row is null for week=1 in each of the data partition.

ITEM	LOCATION	WEEK	SALES_QTY	RCPT_QTY	SALES_PRIOR_WEEK
...					
Pants	1	10	859	888	465
Pants	2	1	453	427	
Pants	2	2	281	674	453
Pants	2	3	855	182	281
Pants	2	4	764	366	855
Pants	2	5	864	362	764
Pants	2	6	843	599	864
Pants	2	7	99	513	843
Pants	2	8	901	74	99
Pants	2	9	363	678	901
Pants	2	10	17	692	363
Pants	3	1	495	364	
Pants	3	2	956	15	495
...					

Syntax for lag is: Lag (value\_expr, offset, default). Second and third parameters are optional parameters. Second parameter is to access any prior row by providing a physical offset, defaults to 1. Third parameter can be used to provide value if the prior value\_exp is null.

### Example: sales change percent query

In this example, we calculate sales\_change\_percent, defined as percent change in sales from prior week.

```
select item,location,week, sales_qty, rcpt_qty ,
trunc ( (sales_qty-sales_prior_week)/sales_prior_week,4)*100 sales_change_percent.....1
from (.....2
select item, location, week, sales_qty, rcpt_qty,
lag(sales_qty) over ( partition by item, location order by week ) sales_prior_week
from item_data
);
```

1. Formula for this calculation is :

$$\text{Sales\_change\_percent} = \frac{\text{This week's Sales\_qty} - \text{last week sales\_qty}}{\text{Last week sales qty}} * 100$$

Last week sales\_qty is a derived column sales\_prior\_week in the subquery

2. Sub query discussed earlier.

Output of the query is:

ITEM	LOCATION	WEEK	SALES_QTY	RCPT_QTY	SALES_CHANGE_PERCENT
Pants	1	1	638	524	
Pants	1	2	709	353	11.12
Pants	1	3	775	734	9.30
Pants	1	4	344	879	-55.61
Pants	1	5	990	744	187.79
Pants	1	6	635	409	-35.85
Pants	1	7	524	801	-17.48
Pants	1	8	14	593	-97.32
Pants	1	9	630	646	4400.00
Pants	1	10	61	241	-90.31
Pants	2	1	315	737	
Pants	2	2	176	282	-44.12
Pants	2	3	999	314	467.61

**Function : lead**

Lead is an analytic function to access subsequent row in a data partition. Functionality of this function is very much like lag, except that lag is to access prior row and lead is to access later row.

Consider the following SQL:

```
Select item,week, location , sales_qty,
      lead (sales_qty, 2, sales_qty) over ( partition by item, week order by sales_qty desc ) sls_qty
from item_data;
```

Second line above is to access the sales\_qty column value from a row 2 weeks later. Third parameter is returned if the lead function accesses nonexistent row.

ITEM	WEEK	LOCATION	SALES_QTY	SLS_QTY
...				
Pants	1	43	990	957
Pants	1	10	976	941
Pants	1	31	957	941
Pants	1	5	941	888
Pants	1	45	941	874
Pants	1	44	888	843
Pants	1	27	874	836
Pants	1	14	843	832
Pants	1	42	836	827
....				

For example, for the row item='PANTS', week=1 and location=43 sls\_qty is derived from sales\_qty column value for the row with item='PANTS', week=1 and location 31.

Note that, in this example we modified the partitioning columns to be item, week order by sales\_qty desc. This SQL is returning the top locations by sales.

**Example: query to use both lead and lag**

In this example, we query sales for the preceding two locations, current location and succeeding two locations order by week desc.

```
Select item, location , week,
      lag (sales_qty, 2, sales_qty)
      over ( partition by item, location order by week asc ) prec_week_2_sls_qty,
      lag (sales_qty, 1, sales_qty)
      over ( partition by item, location order by week asc ) prec_week_1_sls_qty,
      sales_qty,
      lead (sales_qty, 1, sales_qty)
      over (partition by item, location order by week asc ) succ_week_1_sls_qty,
      lead (sales_qty, 2, sales_qty)
      over (partition by item, location order by week asc ) succ_week_2_sls_qty
from item_data
order by item,location,week ;
```

We can query 2 rows preceding and 2 rows succeeding using this SQL, without a self join.

ITEM	LOCATION	WEEK	Preceding week 2	Preceding week 1	This week	Succeeding week 1	Succeeding week 2
...							
Pants	1	1	250	250	250	634	747
Pants	1	2	634	250	634	747	777
Pants	1	3	250	634	747	777	379
Pants	1	4	634	747	777	379	309
Pants	1	5	747	777	379	309	387
Pants	1	6	777	379	309	387	199
Pants	1	7	379	309	387	199	465
Pants	1	8	309	387	199	465	859
Pants	1	9	387	199	465	859	465

Pants	1	10	199	465	859	859	859
Pants	2	1	453	453	453	281	855
Pants	2	2	281	453	281	855	764
Pants	2	3	453	281	855	764	864

**Script:** anfn\_lead.sql

### **Function : first value & last value**

First\_value and last\_value functions are useful in finding values for top 1 row and bottom 1 row from a data partition. For example, queries to find the top location by sales and bottom location by sales can be written using these functions.

### **Example: top seller and worst seller**

In this example, we will query to find the location and sales\_qty with highest and lowest sales\_qty for an item and week.

```
select distinct item, week,
  first_value (location) over (partition by item, week order by sales_qty desc.....1
    rows between unbounded preceding and unbounded following ) top_seller,.....2
  first_value (sales_qty) over (partition by item, week order by sales_qty desc.....3
    rows between unbounded preceding and unbounded following ) top_sales_qty,
  last_value (location) over (partition by item, week order by sales_qty desc.....4
    rows between unbounded preceding and unbounded following ) worst_seller,
  last_value (sales_qty) over (partition by item, week order by sales_qty desc
    rows between unbounded preceding and unbounded following ) worst_sales_qty
from item_data
order by item, week
/
```

1. Data is partitioned by item, week ordered by sales\_qty descending. This will bring the row with highest sales\_qty within that item, week partition. First\_value(location) retrieves the location associated with that first row.
2. SQL clause 'rows between unbounded preceding and unbounded following' delimits set of rows first\_value function can operate. Unbounded specifies that there is no boundary for the set of rows within that data partition. In essence, first\_value will operate on all the rows for that item and week. Rows are sorted by item, week, sales\_qty desc and for each unique item, week combination top row is selected and location associated with that row is returned.
3. first\_value(sales\_qty) returns the sales\_qty associated with the top row by sales\_qty.
4. last\_value(location) retrieves the last row from the data partition and returns the location associated with that row.

Output of the above SQL:

ITEM	WEEK	TOP_SELLER	TOP_SALES_QTY	WORST_SELLER	WORST_SALES_QTY
Pants	1	43	990	49	39
Pants	2	50	980	28	10
Pants	3	2	999	50	43
Pants	4	7	988	44	81
Pants	5	1	990	40	11
Pants	6	42	982	47	37
Pants	7	39	977	14	18
Pants	8	36	995	1	14
Pants	9	26	981	21	30

From the above we can see that location 43 is top seller and sold 990 Pants in that week. Also location 49 is the worst seller and sold just 39 Pants.

**Script:** anfn\_first\_value.sql

**Function : Dense rank**

Dense\_rank function returns rank of a row in a given data partition. Following SQL illustrates this function.

```
select distinct item,week, location , sales_qty,
dense_rank () over (partition by item, week order by sales_qty desc ) sls_rnk .....1
from item_data
order by item, week,sls_rnk;.....2
```

1. dense\_rank returns rank of the current row in a given data partitioned by item, week and ordered by sales\_qty descending.
2. Sorting is done at item, week, sls\_rnk and this returns top sellers first.

Output of the query is:

ITEM	WEEK	LOCATION	SALES_QTY	SLS_RNK
Pants	1	35	984	1
Pants	1	26	925	2
Pants	1	39	922	3
Pants	1	40	897	4
Pants	1	34	889	5
Pants	1	7	870	6
Pants	1	14	866	7

....

**Example: top 2 seller by sales qty**

In this example, we will retrieve top two locations by sales\_qty per item/week and their rank in receipt\_qty. A subquery is used to retrieve top two sellers.

```
select * from (
select item,week, location , sales_qty,
dense_rank () over (partition by item, week order by sales_qty desc ) sls_rank,.....1
dense_rank () over (partition by item, week order by rcpt_qty desc ) rcpt_rank.....2
from item_data
) where sls_rank <3.....3
;
```

1. dense\_rank calculates the rank of the current row within the data partition for item and week, ordered by sales\_qty descending.
2. dense\_rank calculates the rank of the current row within the data partition for item and week, ordered by rcpt\_qty descending.
3. Limits rows to top 2 rows by sls\_rank derived column

Output of the query is:

ITEM	WEEK	LOCATION	SALES_QTY	SLS_RANK	RCPT_RANK
Pants	1	43	990	1	34
Pants	1	10	976	2	31
Pants	2	50	980	1	7
Pants	2	12	969	2	14
Pants	2	40	969	2	35
Pants	3	2	999	1	37
Pants	3	30	965	2	12
Pants	4	7	988	1	43
Pants	4	9	986	2	48
.....					
Pants	8	36	995	1	4
Pants	8	43	980	2	34

Pants	9	26	981	1	6
Pants	9	33	918	2	9
Pants	10	42	940	1	20
Pants	10	40	907	2	39
Shirt	1	21	980	1	17
Shirt	1	18	977	2	43
Shirt	2	39	918	1	44
.....					

**Script:** anfn\_dense\_rank.sql

### **Function : NTILE**

Ntile function divides ordered set of rows in to bucket, with every bucket holding same # of rows. Very useful in grouping ordered rows in to different sets.

Inspect the following SQL. Following SQL splits the locations in to 10 different buckets based upon their sales\_qty, bucket 1 has top seller locations and bucket 10 has worst seller locations for that item.

```
select item, location, total_sls_qty,
       ntile(10) over ( partition by item order by total_sls_qty desc ) sales_group.....1
from (select item, location, sum(sales_qty) total_sls_qty from item_data.....2
     group by item, location)
```

1. Rows are partitioned by item and ordered by total\_sls\_qty descending. Ntile function operates on this ordered set of rows and groups them in to 10 different buckets
2. Grouping at item and location.

Output of the SQL is:

ITEM	LOCATION	TOTAL_SLS_QTY	SALES_GROUP
Pants	30	7106	1
Pants	11	7056	1
Pants	38	6884	1
Pants	22	6846	1
Pants	5	6309	1
Pants	21	6293	2
Pants	39	6115	2
Pants	26	6027	2
Pants	15	6004	2
Pants	46	5928	2
Pants	14	5899	3
...			

This shows that locations with sales\_group = 1 are top sellers of pants.

### **Example: Find locations to improve**

Suppose, we are trying to find all locations that we can concentrate to improve sales. Report is to find all stores and their respective sales\_qty which are falling in the mid range, neither top sellers, nor worst sellers.

Following query is to find those locations:

```
select * from (
  select item, location, total_sls_qty,
         ntile(10) over ( partition by item order by total_sls_qty desc ) sales_group
  from
    (select item, location, sum(sales_qty) total_sls_qty from item_data
     group by item, location)
)
where sales_group in (5,6);
```

Output:

ITEM	LOCATION	TOTAL_SLS_QTY	SALES_GROUP
Pants	10	5319	5
Pants	46	5229	5
Pants	48	5192	5
Pants	47	5146	5
Pants	27	5098	5
Pants	40	5080	6
Pants	50	5050	6
Pants	5	5030	6
Pants	44	4948	6
Pants	38	4844	6
shirt	38	5296	5
shirt	11	5260	5

...  
[Script](#): anfn\_ntile.sql

## **Regular Expressions**

Regular expressions are very useful in searching for patterns in the database columns. Unix style regular expression [POSIX standard] are now available in SQL. There are three new functions introduced to support this new regular expression feature.

1. `regexp_like` – Analogous to 'like' keyword in SQL.
2. `regexp_substr` – Analogous to `substr` function
3. `regexp_instr` – Analogous to `instr` function
4. `regexp_replace` – Analogous to `replace` function and replace the string with supplied string.

We will use the following data to illustrate the power of regular expressions.

```
select name from book_master
SQL> /
NAME
-----
The Very Hungry Caterpillar board book
Brown Bear, Brown Bear, What Do You See?
Panda Bear, Panda Bear, What Do You See?
Cost-Based Oracle Fundamentals
Oracle wait Interface
```

### **Function : regexp\_like**

Function `regexp_like` retrieves the rows with matching patterns. Syntax for the function `regexp_like` is

`Regexp_like (source, pattern, match parameter)`

Only first two parameters are mandatory and third parameter is optional. Parameter `source` is the column or string to search. Second parameter `pattern` is the string to search for and `match parameter` provides ability such as case insensitive searches etc.

### **Function : regexp\_instr**

`Regexp_instr` searches for a pattern and returns the position of the pattern, in the source string.

Syntax for `regexp_instr` is:

`Regexp_instr (source, pattern, position, occurrence, return_option, match_parameter)`

Only first two parameters are mandatory and other parameters are optional. Third parameter specifies the position to start the search from, fourth parameter searches for occurrence of that parameter.

### **Function : regexp\_substr**

Regexp\_substr searches for a pattern and returns the sting that matched pattern, in the source string. Syntax for regexp\_substr is:

```
Regexp_substr (source, pattern, position, occurrence, match_parameter)
```

Only first two parameters are mandatory and other parameters are optional. Third parameter specifies the position to start the search from, fourth parameter searches for occurrence of that parameter.

### **Example: Search for patterns with repeating words**

We will create a query to search for books with repeating words in their names. Code must also rejects words with less than 10 characters. We can see that following two books must be returned using this query:

```
Brown Bear, Brown Bear, What Do You See?
Panda Bear, Panda Bear, what Do You See?
```

This is extremely complex to write this query using traditional SQL features. But regular expressions can be used very easily to achieve this.

SQL:

```
select regexp_substr (name, '([[:alnum:]] | [[:space:]]{10,})*(\1)' sbstr_out,
      regexp_instr (name, '([[:alnum:]] | [[:space:]]{10,})*(\1)' instr_out
from book_master
where regexp_like (name, '.*([[:alnum:]] | [[:space:]]{10,})*(\1).*');
```

In our example, name is the source string and pattern we are trying to match is represented in regular expression as:

```
'([[:alnum:]] | [[:space:]]{10,})*(\1)'
```

Let us split this expression in to smaller chunks and explore it. In the following expression ([[:alnum:]] | [[:space:]]{10,}),

[[:alnum:]] matches any alpha numeric characters.

[[:space:]] matches a space character.

[[:alnum:]] | [[:space:]] matches one alphanumeric or space character

([[:alnum:]] | [[:space:]]{10,}) matches 10 or more alphanumeric or space characters. {10,} means that 10 or more prior characters. First parameter with in curly bracket, specifies the minimum # of characters and second parameter specifies maximum # of characters.

([[:alnum:]] | [[:space:]]{10,})\* matches 10 or more alphanumeric or space characters followed by zero or more occurrence of any character. Here '.' Matches with any character and '\*' means zero or more characters.

Surrounding the string with paranthesis [ (..) ] allows us to reference the string later, without explicitly using the string itself.

'([[:alnum:]]|[:space:]){10,}.\*(\1)' String (\1) matches previously matched first string. In this case it is 10 or more alphanumeric or space characters.

In essence, this string matches

10 or more alphanumeric or space characters

followed by zero or more any character

followed by 10 or more alphanumeric or space characters.

Output shows that book names with words repeated are returned.

Output of this query is:

SBSTR_OUT	INSTR_OUT
Brown Bear, Brown Bear	1
Panda Bear, Panda Bear	1

[Script](#): populate\_data.sql and regexp\_01.sql

### Example: Search and replace pattern

In this example, we will search for a pattern and replace with a different pattern using regexp\_replace and back referencing.

Consider the following data:

```
select name from class_roster;
```

```
NAME
-----
George :: Adams
Scott  :: Davey
Brian  :: Williams
```

In the above data, format of name is first\_name followed by two colons and last\_name. We will write a query to display the name with the format last\_name, first\_name format.

Following query can replace the pattern:

```
Select name,
       regexp_replace(name, '([[:alnum:]]|[:space:]){1,}::([[:alnum:]]|[:space:]){1,}', '\2,\1')
       last_comma_first
from class_roster
;
```

In the above SQL, second parameter to regexp\_replace is a regular expression pattern. We will decipher them by breaking it apart. Consider the pattern ([[:alnum:]]|[:space:]){1,}.

[[:alnum:]] matches any alpha numeric characters.

[[:space:]] matches a space character.

[[:alnum:]]|[:space:] matches one alphanumeric or space character

[[:alnum:]]|[:space:]){1,} matches one or more alphanumeric or space characters

([[:alnum:]]|[:space:]){1,}) Parentheses are used so that we can back reference them later.

Pattern ([[:alnum:]]|[:space:]){1,}::([[:alnum:]]|[:space:]){1,} essentially matches one or more characters of alphanumeric or space, followed by two colons, followed by one or more characters of alphanumeric or space characters.

Pattern '\2,\1' back references 2<sup>nd</sup> matched string, followed by ',' followed by 1<sup>st</sup> matched string and returns last\_name, first\_name.

Result of above query:

NAME	LAST_COMMA_FIRST
George :: Adams	Adams,George
Scott :: Davey	Davey,Scott
Brian :: Williams	Williams,Brian

**Script:** regexp\_02.sql

## **Flashback features**

Flashback feature allows us to access past image of data without performing an incomplete recovery. Tables can be queried to find past versions of rows and tables can be rolled back to past image. There are many different variations available

- Flashback query
- Flashback transaction query
- Flashback table
- Flashback drop
- Flashback database

We will discuss first three features, here.

### **Flashback query**

Flashback query allows us to access past image of a row. Flashback feature uses undo segments to reconstruct the past image. Initialization parameter undo\_retention plays critical role in deciding how long past images are kept. In most cases, prior images can be queried as old as undo\_retention parameter value.

Consider following table and data:

```
create table emp_salary
(emp_id number,
 salary number
);

insert into emp_salary (emp_id, salary) values ( 1, 2000);
insert into emp_salary (emp_id, salary) values ( 2, 3000);
insert into emp_salary (emp_id, salary) values ( 3, 4000);
insert into emp_salary (emp_id, salary) values ( 4, 5000);
insert into emp_salary (emp_id, salary) values ( 5, 6000);
```

We will consider last row alone for this example. Initial version of the row is:

```
select emp_id, salary from emp_salary where emp_id=5;
EMP_ID SALARY
-----
5      6000
```

We will update this table adding 25% to the salary value for emp\_id=5.

```
update emp_salary set salary=salary*1.25 where emp_id=5 ;
Commit;
```

```
select emp_id, salary from emp_salary where emp_id=5;
EMP_ID      SALARY
-----
5           7500
```

We will update again adding another 25% to salary column value.

```
update emp_salary set salary=salary*1.25 where emp_id=5 ;
Commit;
```

```
select emp_id, salary from emp_salary where emp_id=5;
EMP_ID      SALARY
-----
5           9375
```

Now, we want to see all different versions of the row and we can see that using the following query:

```
select emp_id, salary, versions_xid, versions_starttime, versions_endtime
from emp_salary
versions between scn minvalue and maxvalue
where emp_id=5
order by versions_endtime nulls last;
```

EMP_ID	SALARY	VERSIONS_XID	VERSIONS_STARTTIME	VERSIONS_ENDTIME
5	6000			28-JUL-06 04.21.53 PM
5	7500	0001002200001E32	28-JUL-06 04.21.53 PM	28-JUL-06 04.21.53 PM
5	9375	0003001F00003526	28-JUL-06 04.21.53 PM	

First row is the initial row with salary value of 6000. Versions\_endtime is a pseudo column indicates, that version of row ended at 04.21.53 PM.

Second row shows that salary was updated to 7500. Versions\_starttime shows that this version existed between 04.21.53PM and 04.21.53 PM.

Third version of row is current row and updated to 9375. versions\_endtime is null, meaning version is still alive.

We used clause 'versions between scn minvalue and maxvalue' in above SQL. We provided minvalue and maxvalue to see all possible versions. It is also possible to see only specific versions of the row. Versions can be seen using timestamps also. Here is an example to query the versions of row during past 1 hour.

```
select emp_id, salary, versions_xid, versions_starttime, versions_endtime
from emp_salary
versions between timestamp systimestamp - interval '60' minute and systimestamp-interval '1'
minute
where emp_id=5
order by versions_endtime nulls last
/
```

Another variation of flashback query is 'as of' SQL clause. You can query the past image of row specifying past time.

```
select emp_id, salary
from emp_salary
as of timestamp systimestamp-(5/(24*60*60))
where emp_id=5
```

```

/
EMP_ID      SALARY
-----
5           6000

```

Above query shows the past image of row as of 5 minutes ago. It is also possible to use SCN based query: This SCN was captured 5 minutes ago. Refer to the script below for more details.

```

select emp_id, salary from emp_salary
as of scn 5615683784747
where emp_id=5;

```

[Script: flashback\\_version\\_query.sql](#)

### **Flashback transaction query**

This variant of flashback query can be used to query all DML activity from a specific transaction. This needs an XID or Transaction ID to be passed as the parameter.

For convenience we will repeat a prior query:

```

select emp_id, salary, versions_xid, versions_starttime, versions_endtime
from emp_salary
versions between scn minvalue and maxvalue
where emp_id=5
order by versions_endtime nulls last;

```

```

EMP_ID      SALARY  VERSIONS_XID          VERSIONS_STARTTIME          VERSIONS_ENDTIME
-----
5           6000
5           7500  0001002200001E32     28-JUL-06 04.21.53 PM      28-JUL-06 04.21.53 PM
5           9375  0003001F00003526     28-JUL-06 04.21.53 PM

```

We can see that two transactions modified these rows. Pseudo column versions\_xid retrieves the transaction id associated with that change. We can query to find all DML activity from that specific transaction using following flashback transaction query:

```

select start_timestamp , logon_user, undo_sql
from flashback_transaction_query
where xid=hexraw('0001002200001E32' );

```

```

START_TIM LOGON_USER
-----
UNDO_SQL
-----
02-AUG-06 SQLNF
update "SQLNF"."EMP_SALARY" set "SALARY" = '5000' where ROWID = 'AAAH5aAAEAAACgSAAD' ;

02-AUG-06 SQLNF
update "SQLNF"."EMP_SALARY" set "SALARY" = '7500' where ROWID = 'AAAH5aAAEAAACgSAAE' ;

```

This shows that two rows were updated by this transaction. Flashback\_transaction\_query also has other column start\_timestamp which can be used to order the rows from this table.

[Script: flashback\\_version\\_tx.sql](#)

### **Flashback table**

Flashback table is useful in recovering every row in a table to a past time.

Here is the initial version of rows in emp\_salary table.

```

EMP_ID      SALARY
-----
1           2000

```

```

2      3000
3      4000
4      5000
5      6000

```

Let us capture current scn value to use later. We will flashback the table to this SCN.

```

select current_scn from v$database;
      CURRENT_SCN
-----
5615683871107

```

We will update the table adding 25% to emp\_id in (4,5). We will execute another transaction to update a row followed by a commit.

Final modified versions of row

```

      EMP_ID      SALARY
-----
1              2000
2              3000
3              4000
4              6250 *
5              9375 *  These rows are modified.

```

We will flashback the table to an earlier SCN.

```
flashback table emp_salary to scn 5615683871107;
```

Querying the table, we see that table has been recovered back to a past point in time.

```

select * from emp_salary;
      EMP_ID      SALARY
-----
1              2000
2              3000
3              4000
4              5000
5              6000

```

[Script](#): flashback\_version\_table.sql

## **Pipelined functions**

Pipelined functions provide ability to access output of a function, as if it is a table. Table functions were introduced in 8i. But there were few drawbacks with table functions, such as inability to use table functions directly in SQL in any useful way. Pipelined functions remove those restrictions and now the functions can be accessed using SQL as if it is a table.

### **Example: Pipelined functions**

We will use a simple example to illustrate this feature. Table item\_data described in earlier section is reused. Following is an object type returned by the pipelined function.

```

-- Creating object types to be returned by the function.
create or replace type item_data_typ as object
( item varchar2(30),
  location number,
  qty number
);

create or replace type item_data_tabtyp as table of item_data_typ
/

```

Now, we will write a pipelined function to return above types when called from SQL.

```

create or replace function plsql_piplined_noparallel
return item_data_tabtyp PIPELINED .....1
as
  l_item_data_tabtyp item_data_tabtyp := item_data_tabtyp (); .....2
  indx number:=1;
  l_fib number :=0;
begin
  l_item_data_tabtyp.extend;
  for l_csr_items in (select item, location, qty from item_data )
  loop
    l_fib := next_Fib_N (round(l_csr_items.location) ); .....3
    l_item_data_tabtyp (indx) :=item_data_typ ( l_csr_items.item, l_fib, l_csr_items.qty); .....4
    pipe row ( l_item_data_tabtyp (indx) ); .....5
    l_item_data_tabtyp.extend;
    indx := indx+1;
  end loop;
  return;
end;
/

```

In the above function, return type is declared as item\_data\_tabtyp, which is a table of item\_data\_typ.

1. Keyword PIPELINED indicates that this is a pipelined function.
2. Local variable is declared and initialized by calling the table type constructor.
3. Function next\_Fib\_N is called to find the next Fibonacci series number greater than location.
4. One of array member is populated with a row.
5. Array member is piped so that calling program can consume that.

Pipelined functions operate nearly like producer, consumer model. After declaring the function, we can call this function from SQL, as if this function is a table. In the following SQL, above function is called in from clause as if this is a table. Cast function casts the function output as item\_data\_tabtyp and table keyword is used to access this function, in a table like format.

```

select item, location, sum(qty) from
  table (cast (plsql_piplined_noparallel as item_data_tabtyp))
group by item, location
/

```

ITEM	LOCATION	SUM(QTY)
Pants	1	1091215.73
Pants	13	700517.6
Pants	21	763532.279
Pants	34	706139.503
Pants	55	1495631.24
Pants	89	1534359.5
...		

[Script](#): plsql\_populate\_table.sql, plsql\_table\_fnc\_01.sql

### **Pipelined parallel functions**

Pipelined functions can even operate as classic producer/consumer model. Multiple parallel query processes can operate as producer and receiving code acts as consumer. Multi level producer/consumer model also possible in complex situations.

Following shows the parallel pipelined function. We will also populate sid of the current connection that is processing this row, so that we can see producer/consumer model. Parallel\_enable keyword is to parallelize the item\_data table rows by range (location) column.

```
create or replace function plsqli_piplined_parallel (
  l_item_data_ref in item_data_pkg.item_data_ref)
return item_data_tabtyp PIPELINED
parallel_enable (partition l_item_data_ref by range (location) )
as
  l_item_data_tabtyp item_data_tabtyp := item_data_tabtyp ();
  indx number:=1;
  l_fib number :=0;
  l_item varchar2(30);
  l_location number;
  l_qty number;

  l_sid number;
  l_dummy_sid number;
begin
  l_item_data_tabtyp.extend;
  select sid into l_sid from v$mystat where rownum <2;
  loop
    fetch l_item_data_ref into l_item , l_location, l_qty , l_dummy_sid ;
    exit when l_item_data_ref%NOTFOUND;
    l_fib := next_Fib_N (round(l_location) );
    l_item_data_tabtyp (indx) := item_data_typ ( l_item, l_fib, l_qty ,l_sid);
    pipe row ( l_item_data_tabtyp (indx) );
    l_item_data_tabtyp.extend;
    indx := indx+1;
  end loop;
  return;
end;
/
```

Accessing above function using SQL returns following:

```
select /*+ parallel (5) */ item, location ,sid, sum(qty) from
  table (
    cast (
      plsqli_piplined_parallel(cursor (select item, location, qty ,0 from item_data) )
      as item_data_tabtyp
    )
  )
group by item, location, sid order by sid
/
```

ITEM	LOCATION	SID	SUM(QTY)
...			
Scarves	987	182	5226861.79
Pants	987	182	5243932.25

Pants	21	182	763532.279
Shirts	55	195	1507026.7
Shirts	2584	195	8130601.01
Shirts	610	195	3692315.45
....			

In the above lines, note SID is different for various rows. This shows that indeed rows are processed by different processes, implementing producer/consumer model.

[Script](#): `plsql_table_fnc_parallel.sql`

More complex business applications are possible, where data goes through multiple transformations.

## **PL/SQL warnings**

Oracle 10g introduces PL/SQL warning feature. While creating a stored object, PL/SQL can review the code and print out warnings about code issues, such as dead code, performance issues etc. These errors are generally not errors, rather warnings or potential performance issues during run time. This feature can be enabled at session level, a good usability feature.

To enable PL/SQL warning at session level, for all possible warnings, following SQL can be used.

```
alter session set plsql_warnings='enable:all';
```

General syntax for enabling PL/SQL warning is

```
Alter session set plsql_warnings = ‘
{ ENABLE | DISABLE | ERROR } :
{ ALL | SEVERE | INFORMATIONAL | PERFORMANCE | { integer | (integer [, integer] ..) } }’
```

For example to enable `plsql_warnings` for performance related warnings following SQL can be used.

```
alter session set plsql_warnings='enable:performance';
```

Refer to `$ORACLE_HOME/plsql/mesg/plwus.mesg` for few error codes that can be used with this feature.

### **Plsql warning: dead code**

We will use the following code to test this feature for dead code detection.

```
create or replace procedure plsql_warnings_prc_01 is
  l_id number :=1;
begin
  if (l_id =1) then
    dbms_output.put_line ('Value of l_id is ' || l_id);
  else
    dbms_output.put_line ('Unreachable code '); -- DEAD Code
  end if;
end;
/
```

In above code fragment, we set the `l_id` variable is initialized to 1, but never modified. Later this variable is referenced in the 'if...else' condition clause. 'Else' part of the code will never be executed as `l_id` is always will be 1.

Creating the above procedure generates the following warning message. Clearly PL/SQL engine has identified issue with code

Show errors

Errors for PROCEDURE PLSQL\_WARNINGS\_PRC\_01:  
LINE/COL ERROR

-----  
9/2 PLW-06002: Unreachable code

[Script](#): plsql\_warnings\_01.sql

**Plsql warning: Inefficient assignment**

In this test case, we assign a static value to a variable inside a loop. That assignment must be kept outside the loop for performance reasons.

```
create or replace procedure plsql_warnings_prc_02
is
  l_id number :=1;
begin
  for i in 1 .. 5
  loop
    l_id := 2; -- Inefficient assignment
  end loop;
end;
```

Variable l\_id is initialized to 1 in the declaration section and set to 2 inside the loop. In each iteration of the loop, we always set the variable l\_id value to 2. This is not entirely efficient as the assignment can be moved out of loop.

Creating this procedure, generates this warning:

LINE/COL ERROR

-----  
7/2 PLW-06002: Unreachable code

While the PL/SQL engine prints error in correct line, but the error message itself is not user friendly.

Actually, error message is fine. Reason here is that, PL/SQL moved the assignment outside the loop during optimization and so that part of code just became unreachable or useless code.

[Script](#): plsql\_warnings\_02.sql

**Plsql warning: Inefficient assignment – Part 2**

Let's add some dbms\_output to print the value inside the loop. This is essentially making the execution of loop code necessary.

Difference between following code and above code is that a dbms\_output line is added.

```
create or replace procedure plsql_warnings_prc_02
is
  l_id number :=1;
begin
```

```

for i in 1 .. 5
loop
  l_id := 2;
  dbms_output.put_line ('i = ' || i); -- Added this line
end loop;
end;
/
Procedure created.

```

show errors  
No errors.

So, PL/SQL correctly identified that loop is necessary as there are no warning messages.

### **Plsql warning: Pass by reference**

By definition, parameters passed as 'IN OUT' mode can be modified by caller and called programs. Parameters defined as IN OUT are copied to a temporary variable in the called program and at the end of normal completion of the called program values are returned from the temporary variable, which is copied back to the actual parameter in the calling program.

If the parameter size is huge, this tend to create performance issue as there is additional processing involved and memory must be allocated for this temporary variable etc.

This can be avoided by passing the variable by reference. This test case probes whether PL/SQL can warn us about this issue.

prompt This procedure accepts clob as in out paramteter

prompt

```
create or replace procedure plsql_warnings_prc_05
```

```
( l_emp_jpeg in out blob ) -- BLOB as in out parameter, potentially huge size
```

```
is
```

```
  l_id number :=1;
```

```
  l_lc number :=1;
```

```
begin
```

```
  l_id := 'ORCL';
```

```
  l_id := l_id + 1;
```

```
end;
```

```
/
```

SP2-0804: Procedure created with compilation warnings

Show errors

Errors for PROCEDURE PLSQL\_WARNINGS\_PRC\_05:

LINE/COL ERROR

```
-----
2/5   PLW-07203: parameter 'L_EMP_JPEG' may benefit from use of the
      NOCOPY compiler hint

```

PL/SQL correctly identified that this variable can benefit fro use of NOCOPY compiler hint.

**Script:** plsql\_warnings\_05.sql

### **Plsql warning: Infinite loop**

In this test case, we will create variations of code with infinite loop and test whether this feature can detect infinite loops in code.

In the following code, exit condition for while loop is that  $l\_lc \leq 100$ , which will never occur, leading to infinite loop in processing.

```
create or replace procedure plsql_warnings_prc_03
is
  l_id number :=1;
  l_lc number :=1;
begin
  while (l_lc <= 100)
  loop
    l_id := l_id *l_id;
  end loop;
end;
/
Procedure created.
```

Procedure was created with out any errors. So, question arises, is it easy to detect every possible infinite loop. This problem is known as ‘halting problem’ and solution is not so mundane. Wikipedia says that:

“[Alan Turing](#) proved in [1936](#) that a general [algorithm](#) to solve the halting problem for *all* possible program-input pairs cannot exist. “

There are various other test cases in this script file testing this feature.

**Script:** plsql\_warnings\_05.sql

### **Plsql warning: Use of keywords**

It is not very uncommon for developers to name their variables using keywords. This is very hard to detect unless specialized tools are written to look for keywords. Plsql\_warning feature detects this keyword reuse.

```
create or replace procedure plsql_warnings_prc_06
  ( l_emp_id in number )
is
  l_id number :=1;
  l_lc number :=1;
  cv number :=0;
begin
  l_id := 'ORCL';
  l_id := l_id + 1;
  cv :=1;
end;
/
SP2-0804: Procedure created with compilation warnings
```

Errors for PROCEDURE PLSQL\_WARNINGS\_PRC\_06:  
LINE/COL ERROR

```
-----
6/3    PLW-05004: identifier CV is also declared in STANDARD or is a SQL
        builtin
```

[Script](#): `plsql_warnings_06.sql`

## **PL/SQL native compilation**

PL/SQL is an interpretive language, at least until 9i. Now, PL/SQL code can be natively compiled. When a PL/SQL stored procedure is created, Oracle converts that to an intermediate m-code. This code is converted to machine instructions during run time.

For computationally intensive processing, this tends to be slower. Performance can be improved by compiling code in to native machine instructions and run directly. Oracle provides ability to convert the stored procedures to executable as shared libraries. This feature is known as PL/SQL native compilation.

There needs to be a basic setup before native compilation is possible.

-- Native compilation needs these three parameters set.

```
alter session set plsql_code_type=native;
```

```
alter session set plsql_debug=false;
```

-- This will create shared libraries in /tmp directory

```
alter system set plsql_native_library_dir='/tmp';
```

-- Now, let's create procedures and see what happens

```
create or replace procedure plsql_compile_prc_01
```

```
is
```

```
  l_id number :=1;
```

```
begin
```

```
  if (l_id =1) then
```

```
    dbms_output.put_line ('Value of l_id is ' || l_id);
```

```
  else
```

```
    dbms_output.put_line ('Unreachable code ');
```

```
  end if;
```

```
end;
```

```
/
```

We can see that this PL/SQL block is converted to 'C' code. Here is the partial listing of PL/SQL code.

```
/*----- Implementation of Procedure PLSQL_COMPILE_PRC_01 -----*/
```

```
# ifdef __cplusplus
```

```
extern "C" {
```

```
# endif
```

```
# ifndef PEN_ORACLE
```

```
# include <pen.h>
```

```
# endif
```

```

/* Types used in generated code */

typedef union {ub1 st[344]; size_t _si; void * _vs;} PEN_State;
typedef union {ub1 cup[328]; size_t _cu; void * _vc;} PEN_Cup;
typedef union {ub1 slg[112]; pen_buffer p;} PEN_Buffer;

/* Macros used in generated code */

#define dl0 ((void ***) (PEN_Registers[ 3]))
#define dpf ((void ****) (PEN_Registers[ 5]))

#define bit(x, y) ((x) & (y))
#define PETisstrnull(strhdl) \
    (!PMUflgnotnull(PETmut(strhdl)) || !PETdat(strhdl) || !PETlen(strhdl))
.....

```

Due to issues with compiler, author was not able to test this feature thoroughly.

[Script](#): `plsql_compile_01.sql`

### **About the author**

*Riyaj Shamsudeen has 15+ years of experience in Oracle and 12+ years as an Oracle DBA. He currently works for JCPenney, specializes in performance tuning and database internals. He has authored few articles such as internals of locks, internals of hot backups, redo internals etc. He also teaches in community colleges in Dallas such as North lake college. He is a board member for DOUG (Dallas Oracle User Group).*

*When he is not dumping the database blocks, he can be seen playing soccer with his kids.*

*Special thanks to Nisha Riyaj for reviewing this document critically. Any problem with the document is mine, not hers though.*

## References

1. Oracle support site. Metalink.oracle.com. Various documents
2. Internal's guru Steve Adam's website  
[www.ixora.com.au](http://www.ixora.com.au)
3. Jonathan Lewis' website  
[www.jlcomp.daemon.co.uk](http://www.jlcomp.daemon.co.uk)
4. Tom Kyte's website  
Asktom.oracle.com
5. Oracle Data Cartridge developer's guide
6. What's New in PL/SQL in Oracle Database 10g?  
[http://www.oracle.com/technology/tech/pl\\_sql/htdocs/New\\_In\\_10gR1.htm](http://www.oracle.com/technology/tech/pl_sql/htdocs/New_In_10gR1.htm)
7. PL/SQL Pipelined and Parallel Table Functions in Oracle9i: Metalink note 136909.1
8. New Features For Oracle10g PL/SQL 10.x . Metalink Note:311971.1
9. PL/SQL just got faster. Follow links from  
[http://www.oracle.com/technology/tech/pl\\_sql/htdocs/New\\_In\\_10gR1.htm](http://www.oracle.com/technology/tech/pl_sql/htdocs/New_In_10gR1.htm)
10. Freedom, order, PL/SQL optimization  
[http://www.oracle.com/technology/tech/pl\\_sql/htdocs/New\\_In\\_10gR1.htm](http://www.oracle.com/technology/tech/pl_sql/htdocs/New_In_10gR1.htm)
11. PL/SQL performance – debunking myths



**Appendix #1: Environment details**

Sun Solaris 2.9

Oracle version 10.1.0.4

No special configurations such as RAC/Shared server etc.

Locally managed tablespaces

No ASM

No ASSM

And

Linux CentOS 4.3

Oracle version 10.1.0.4

No special configurations such as RAC/Shared server etc.

Locally managed tablespaces

No ASM

No ASSM

**Appendix #2: Scripts**

Sl #	Script_name	Topic	Description
1	Subquery_factoring.sql	Subquery factoring	To test subquery_factoring using item_data table.
2	Subquery_factoring_dblink.sql	Subquery factoring	Subquery factoring with database links
3	Subquery_factoring_with_hint.sql	Subquery factoring	Subquery factoring with hint
4	Model_example2.sql	Model clause	Model clause to calculate inventory
5	Model_example3.sql	Model clause	Running sales quantity
6	Model_example4.sql	Model clause	Moving sales average report
7	Anfn_lead.sql	Analytic functions	Use of lag and lead function for the past two weeks and next two weeks
8	Anfn_first_value.sql	Analytic functions	SQL to find top seller and worst seller
9	Anfn_dense_rank.sql	Analytic functions	SQL to find rank by sales_qty and rpt_qty
10	Anfn_ntile.sql	Analytic functions	SQL to find mid band of locations
11	Populate_data.sql	Regular expressions	Populate data to test regular expressions
12	Regexp_01.sql	Regular expressions	Search for repeating words
13	Regexp_02.sql	Regular expressions	SQL to search and replace
14	Flashback_version_query.sql	Flashback	Flashback query
15	Flashback_version_tx.sql	Flashback	Flashback TX
16	Flashback_version_table.sql	Flashback	Flashback table
17	Plsql_table_fnc_01.sql	Pipelined functions	To test pipelined function
18	Plsql_table_fnc_parallel.sql	Pipelined parallel functions	To test pipelined functions in parallel
19	Plsql_warnings_01.sql	Plsql_warning	To detect dead code
20	Plsql_warnings_02.sql	Plsql_warning	Detect inefficient assignment
21	Plsql_warnigns_03.sql	Plsql_warning	Detect inefficient assignment with dbms_output
22	Plsql_warnings_05.sql	Plsql_warning	Pass by reference
23	Plsql_warnings_06.sql	Plsql_warning	Use of keywords
24	Plsql_compile_01.sql	Native compilation	Test native compilation.

Appendix #3: Scripts

script: subquery\_factoring.sql

```
-----
-- Script : subquery_factoring.sql
-----
-- This script is to test subquery factoring feature
-- Author : Riyaj Shamsudeen
-- No implied or explicit warranty !
-- Note:
-----
with dept_avg as
  (select deptno, avg(sal) avg_sal from emp group by deptno ),
all_avg as
  (select avg(avg_sal) avg_sal from dept_avg )
select d.deptno, d_avg.avg_sal , a_avg.avg_sal
  from
    dept d,
    dept_avg d_avg,
    all_avg a_avg
where
  d_avg.avg_sal >= a_avg.avg_sal and
  d.deptno=d_avg.deptno
/
set autotrace off
-- autotrace does not print the explain plan correctly
explain plan set statement_id='d1' for
with dept_avg as
  (select deptno, avg(sal) avg_sal from emp group by deptno ),
all_avg as
  (select avg(avg_sal) avg_sal from dept_avg )
select d.deptno, d_avg.avg_sal , a_avg.avg_sal
  from
    dept d,
    dept_avg d_avg,
    all_avg a_avg
where
  d_avg.avg_sal >= a_avg.avg_sal and
  d.deptno=d_avg.deptno
/
pause Press any key to contonue
set lines 160
set pages 40
undef stid
select * from table(dbms_xplan.display(null,null,'all'));
rem select * from table(dbms_xplan.display(null,'&stid','all'));
```

**Script:** subquery\_factoring\_dblink.sql

```
-----
-- Script : subquery_factoring_dblink.sql
-----
-- This script is to test sub query factoring feature with database link
-- Author : Riyaj Shamsudeen
-- No implied or explicit warranty !
-- Note:
-----
--
-- Need database link loopback to be created.
--
-- create database link loopback
-- connect to <username>
-- identified by <password>
-- using '<tnsstring>';
--
prompt Create database link before running this query
set autotrace off
-- autotrace does not print the explain plan correctly
with dept_avg as
  (select deptno, avg(sal) avg_sal from emp@loopback group by deptno ),
all_avg as
  (select avg(avg_sal) avg_sal from dept_avg )
select d.deptno, d_avg.avg_sal , a_avg.avg_sal
      from
      dept d,
      dept_avg d_avg,
      all_avg a_avg
where
      d_avg.avg_sal >= a_avg.avg_sal and
      d.deptno=d_avg.deptno
/
explain plan set statement_id='d1' for
with dept_avg as
  (select deptno, avg(sal) avg_sal from emp@loopback group by deptno ),
all_avg as
  (select avg(avg_sal) avg_sal from dept_avg )
select d.deptno, d_avg.avg_sal , a_avg.avg_sal
      from
      dept d,
      dept_avg d_avg,
      all_avg a_avg
where
      d_avg.avg_sal >= a_avg.avg_sal and
      d.deptno=d_avg.deptno
/
pause Press any key to contonue
set lines 160
set pages 40
undef stid
select * from table(dbms_xplan.display(null,null,'all'));
rem select * from table(dbms_xplan.display(null,'&&stid','all'));
```

**Script:** subquery\_factoring\_with\_hint.sql

```
-----  
-- Script : subquery_factoring_with_hint.sql  
-----  
-- This script is to test sub query factoring feature with database link  
--  
--  
-- Author : Riyaj Shamsudeen  
-- No implied or explicit warranty !  
-- Note:  
-----  
set autotrace on  
with dept_avg as  
  (select /*+ materialized */ deptno, avg(sal) avg_sal from emp group by deptno ),  
all_avg as  
  (select /*+ materialized */ avg(sal) avg_sal from emp )  
select d.deptno, d_avg.avg_sal , a_avg.avg_sal  
  from  
    dept d,  
    dept_avg d_avg,  
    all_avg a_avg  
where  
  d_avg.avg_sal >= a_avg.avg_sal and  
  d.deptno=d_avg.deptno  
/
```

**Script:** model\_example2.sql

```
-----
-- Script : model_example2.sql
-----
--
-- Author : Riyaj Shamsudeen
-- No implied or explicit warranty !
-- Note:
--       To explain model clause using real life example
-----

prompt Recreating tables...
drop table item_data;
create table item_data (
    item varchar2(10),
    location number,
    week number,
    sales_qty number ,
    rcpt_qty number )
/
prompt populating tables
declare
    type product_array is table of varchar2(32);
    l_product_array product_array := product_array ('Shirt','Pants','Ties');
begin
    for prodind in 1..3 loop
        for location in 1..50 loop
            for week in 1..10 loop
                insert into item_Data
                values (
                    l_product_array(prodind),
                    location,
                    week,
                    trunc(dbms_random.value(10, 1000)),
                    trunc(dbms_random.value(10, 1000))
                );
            end loop;
        end loop;
    end loop;
    commit;
end;
/
select * from item_data where rownum <11;
set pause on
set pagesize 40
select item, location, week, inventory, sales_qty, rcpt_qty
    from item_data
    model return updated rows
    partition by (item)
    dimension by (location, week)
    measures ( 0 inventory , sales_qty, rcpt_qty)
    rules (
    inventory [location, week] = nvl(inventory [cv(location), cv(week)-1 ] ,0)
        - sales_qty [cv(location), cv(week) ] +
        + rcpt_qty [cv(location), cv(week) ]
    )
)
```

```

order by item , location,week
/
set pages 40
pause on
select item, location, week, inventory, sales_qty, rcpt_qty , sales_so_far
  from item_data
  model return updated rows
  partition by (item)
  dimension by (location, week)
  measures ( 0 inventory , sales_qty, rcpt_qty, 0 sales_so_far )
  rules (
    inventory [location, week] = nvl(inventory [cv(location), cv(week)-1 ] ,0)
      - sales_qty [cv(location), cv(week) ] +
      + rcpt_qty [cv(location), cv(week) ],
    sales_so_far [location, week] = sales_qty [cv(location), cv(week)] +
      nvl(sales_so_far [cv(location), cv(week)-1],0)
  )
order by item , location,week
/

```

**Script:** model\_example3.sql

```

-----
-- Script : model_example3.sql
-----
-- Author : Riyaj Shamsudeen
-- No implied or explicit warranty !
-- Note:
--       To explain model clause using real life example -3
--       Should be run after running mode_example2.sql
-----
prompt Please resize window as the line size is set to 160 now
prompt This script will show running total of sales for that item and store
set pages 40
set pause on
set lines 160
select item, location, week, inventory, sales_qty, rcpt_qty , sales_so_for, rcpt_so_for
  from item_data
  model return updated rows
  partition by (item)
  dimension by (location, week)
  measures ( 0 inventory , sales_qty, rcpt_qty, 0 sales_so_for, 0 rcpt_so_for )
  rules (
    inventory [location, week] = nvl(inventory [cv(location), cv(week)-1 ] ,0)
      - sales_qty [cv(location), cv(week) ] +
      + rcpt_qty [cv(location), cv(week) ],
    sales_so_for [location, week] = sales_qty [cv(location), cv(week)] +
      nvl(sales_so_for [cv(location), cv(week)-1],0),
    rcpt_so_for [location, week] = rcpt_qty [cv(location), cv(week)] +
      nvl(rcpt_so_for [cv(location), cv(week)-1],0)
  )
order by item , location,week

```

```

Script: model_example4.sql
-----
-- Script : model_example4.sql
-----
-- Author : Riyaj Shamsudeen
-- No implied or explicit warranty !
-- Note:
--       To explain model clause using real life example 4
--       Should be run after running mode_example2.sql
-----
prompt Please resize window as the line size is set to 160 now
prompt This script will show running total of sales for that item and store
set pages 40
set pause on
set lines 160
set feedback on
set echo on
prompt Press any key to continue
select item, location, week, sales_qty, rcpt_qty , moving_sales_avg
  from item_data
  model return updated rows
  partition by (item)
  dimension by (location, week)
  measures ( 0 moving_sales_avg , sales_qty, rcpt_qty)
  rules (
  moving_sales_avg [location, week] = (sales_qty [cv(location), cv(week)]+
    nvl(sales_qty [cv(location), cv(week)-1 ] ,sales_qty [cv(location), cv(week)] ) +
    nvl(sales_qty [cv(location), cv(week)-2 ] ,sales_qty [cv(location), cv(week)] ) )
  /3
  )
order by item , location,week
/

```

**Script:** anfn\_lead.sql

```
-----
-- Script : anfn_example2.sql
-----
-- Author : Riyaj Shamsudeen
-- No implied or explicit warranty !
-- Note:
--       To explain analytics function. Uses the data from anfn_example1.sql script
--       dense_rank explained here..
-----
set pause on pagesize 40
set lines 120
pause 'Press any key to continue'
col prec_week_1_sls_qty heading 'Preceding|week 1'
col prec_week_2_sls_qty heading 'Preceding|week 2'
col sales_qty heading 'This|week'
col succ_week_1_sls_qty heading 'Succeeding|week 1'
col succ_week_2_sls_qty heading 'Succeeding|week 2'
select
  item, location , week,
  lag (sales_qty, 2, sales_qty ) over (
    partition by item, location order by week asc ) prec_week_2_sls_qty,
  lag (sales_qty, 1, sales_qty ) over (
    partition by item, location order by week asc ) prec_week_1_sls_qty,
  sales_qty,
  lead (sales_qty, 1, sales_qty ) over (
    partition by item, location order by week asc ) succ_week_1_sls_qty,
  lead (sales_qty, 2, sales_qty ) over (
    partition by item, location order by week asc ) succ_week_2_sls_qty
from item_data
order by item,location,week
/
```

**Script:** anfn\_first\_value.sql

```
-----
-- Script : anfn_first_value.sql
-----
-- Author : Riyaj Shamsudeen
-- No implied or explicit warranty !
-- Note:
--       To explain analytics function. Uses the data from anfn_example1.sql script
--       dense_rank explained here..
-----
set pause on pagesize 40
select distinct item, week,
  first_value (location) over (partition by item, week order by sales_qty desc
    rows between unbounded preceding and unbounded following ) top_seller,
  first_value (sales_qty) over (partition by item, week order by sales_qty desc
    rows between unbounded preceding and unbounded following )
top_sales_qty,
  last_value (location) over (partition by item, week order by sales_qty desc
    rows between unbounded preceding and unbounded following ) worst_seller,
  last_value (sales_qty) over (partition by item, week order by sales_qty desc
    rows between unbounded preceding and unbounded following )
worst_sales_qty
from item_data
order by item, week
```

**Script:** anfn\_dense\_rank.sql

```
-----
-- Script : anfn_example2.sql
-----
-- Author : Riyaj Shamsudeen
-- No implied or explicit warranty !
-- Note:
--       To explain analytics function. Uses the data from anfn_example1.sql script
--       dense_rank explained here..
-----
set pause on pagesize 40
column sales_change_percent format 999999999.99
-- To get the top 2 locations by store and their receipt rank for all weeks
select * from (
  select
    item,week, location , sales_qty,
    dense_rank () over (
      partition by item, week order by sales_qty desc ) sls_rnk,
    dense_rank () over (
      partition by item, week order by rcpt_qty desc ) rcpt_rnk
  from item_data
) where sls_rnk <3
/
```

**Script:** populate\_data.sql

```
-----
-- Script : populate_data.sql
-----
-- Author : Riyaj Shamsudeen
-- No implied or explicit warranty !
-- Note:
-----
create table book_master
( name varchar2 (255),
  author varchar2(255),
  isbn varchar2(100)
);

insert into book_master values
('The Very Hungry Caterpillar board book',
 'Eric Carle',
 '0399226907' );

insert into book_master values
('Brown Bear, Brown Bear, What Do You See?',
 'Bill Martin Jr',
 '0805047905' );

insert into book_master values
('Panda Bear, Panda Bear, What Do You See?',
 'Bill Martin Jr',
 '0805017585' );

insert into book_master values
('Cost-Based Oracle Fundamentals', 'Jonathan Lewis',1590596366);

insert into book_master values
```

```

        ('Oracle wait Interface', 'Richmond Shee', '007222729X')
;
commit;
create table class_roster
(
    name varchar2(40),
    grade number
);
insert into class_roster values ( 'George :: Adams' , 8);
insert into class_roster values ( 'Scott :: Davey' , 8);
insert into class_roster values ( 'Brian :: Williams' , 8);
commit;

```

**Script:** regexp\_01.sql

```

-----
-- Script : regexp_01.sql
-----
-- This script is to use regular expression : use regexp_like
-- Author : Riyaj Shamsudeen
-- No implied or explicit warranty !
-- Note:
-----

set lines 120 pages 40
col substr_out format a40
select
regexp_substr (name, '([[:alnum:]]|[:space:]){10,}).*(\1)') substr_out,
regexp_instr (name, '([[:alnum:]]|[:space:]){10,}).*(\1)') instr_out
from book_master
where
regexp_like (name, '.*([[:alnum:]]|[:space:]){10,}).*(\1).*')
/

```

**Script:** regexp\_02.sql

```

-----
-- Script : regexp_02.sql
-----
-- This script is to use regular expression : use regexp_replace
-- Author : Riyaj Shamsudeen
-- No implied or explicit warranty !
-- Note:
-----

set lines 120 pages 40
column name format A40
column last_comma_first format A40
select
name,
regexp_replace(name,
'([[:alnum:]]|[:space:]){1,}::([[:alnum:]]|[:space:]){1,})' ,
'\2,\1' ) last_comma_first
from class_roster
/

```

```

Script: flashback_version_query.sql
-----
-- Script : flashback_version_query.sql
-----
-- This script is to populate tables to describe flashback
-- Author : Riyaj Shamsudeen
-- No implied or explicit warranty !
-- Note:
-----

prompt
prompt Populating emp_salary table...
prompt
set feedback off
drop table emp_salary;
create table emp_salary
  ( emp_id number,
    salary number
  );
prompt
prompt Sleeping for 4 seconds
prompt SCN to mapping is at 3 seconds precision
prompt We sleep for 4 seconds so that we can get different timestamps
prompt
begin
  dbms_lock.sleep(4);
end;
/
insert into emp_salary (emp_id, salary) values ( 1, 2000);
insert into emp_salary (emp_id, salary) values ( 2, 3000);
insert into emp_salary (emp_id, salary) values ( 3, 4000);
insert into emp_salary (emp_id, salary) values ( 4, 5000);
insert into emp_salary (emp_id, salary) values ( 5, 6000);
commit;
REM set feedback on
prompt Note down the current scn and time
column current_scn format 99999999999999
column current_time format A40
select current_scn, scn_to_timestamp (current_scn) current_time
from v$database;
prompt
prompt Sleeping for 4 seconds
prompt
begin
  dbms_lock.sleep(4);
end;
/
column curscn noprint new_value currentscn
select to_char(current_scn,'99999999999999') curscn , scn_to_timestamp (current_scn)
current_time
from v$database;
pause 'press any key to continue'
!tput clear
prompt Initial version of the row
select emp_id, salary
from emp_salary

```

```

where emp_id=5;
prompt We will update the table adding 25% to emp_id=5 salary
update emp_salary set salary=salary*1.25 where emp_id=5 ;
commit;
prompt
prompt Querying the data we see that value is updated!!
prompt
select * from emp_salary where emp_id=5;
prompt
set lines 120
prompt Let us query the older versions of the row
prompt Since we are querying within 3 seconds, flashback does not return rows
prompt correctly
column versions_xid format A20
column versions_starttime format A25
column versions_endtime format A25
select emp_id, salary, versions_xid, versions_starttime, versions_endtime
  from emp_salary
versions between scn minvalue and maxvalue
where emp_id=5;
prompt We will sleep for 4 more seconds and try again
begin
  dbms_lock.sleep(4);
end;
/
prompt Let us also update the row again to clarify the updates to that row
update emp_salary set salary=salary*1.25 where emp_id=5 ;
commit;

prompt Flashback query uses undo technology and so undo_retention must
prompt be sufficiently kept high
prompt
prompt QUERY #1: Following query is to see all versions of that row
prompt
set echo on
select emp_id, salary, versions_xid, versions_starttime, versions_endtime
from emp_salary
versions between scn minvalue and maxvalue
where emp_id=5
order by versions_endtime nulls last ;
set echo off
prompt
prompt QUERY #2: Following query is to see version of that row as of 5 seconds ago
prompt
set echo on
select emp_id, salary
from emp_salary
as of timestamp systimestamp-(5/(24*60*60))
where emp_id=5
/
set echo off
prompt
prompt QUERY #3: Following query is to see version of that row as of scn captured
earlier.
prompt
set echo on

```

```

select emp_id, salary
from emp_salary
as of scn &currentscn
where emp_id=5;
set echo off

```

**Script:** flashback\_version\_tx.sql

```

-----
-- Script : flashback_version_tx.sql
-----
-- This script is to populate tables to describe flashback
-- Author : Riyaj Shamsudeen
-- No implied or explicit warranty !
-- Note:
-----

prompt
prompt Populating emp_salary table....
prompt
set feedback off
drop table emp_salary;
create table emp_salary
( emp_id number,
  salary number
);
prompt
prompt Sleeping for 4 seconds
prompt SCN to mapping is at 3 seconds precision
prompt We sleep for 4 seconds so that we can get different timestamps
prompt
begin
  dbms_lock.sleep(4);
end;
/
insert into emp_salary (emp_id, salary) values ( 1, 2000);
insert into emp_salary (emp_id, salary) values ( 2, 3000);
insert into emp_salary (emp_id, salary) values ( 3, 4000);
insert into emp_salary (emp_id, salary) values ( 4, 5000);
insert into emp_salary (emp_id, salary) values ( 5, 6000);
commit;
REM set feedback on
prompt Note down the current scn and time
column current_scn format 99999999999999
column current_time format A40
select current_scn, scn_to_timestamp (current_scn) current_time
from v$database;
prompt
prompt Sleeping for 4 seconds
prompt
begin
  dbms_lock.sleep(4);
end;
/

select to_char(current_scn,'99999999999999') curscn , scn_to_timestamp (current_scn)
current_time
from v$database;
pause 'press any key to continue'

```

```

!tput clear
prompt Initial version of the row
select emp_id, salary
  from emp_salary
 where emp_id=5;
prompt We will update the table adding 25% to emp_id=5 salary
update emp_salary set salary=salary*1.25 where emp_id=5 ;
commit;
prompt
prompt Querying the data we see that value is updated!!
prompt
select * from emp_salary where emp_id=5;
prompt
set lines 120
prompt Let us query the older versions of the row
prompt Since we are querying within 3 seconds, flashback does not return rows
prompt correctly
column versions_starttime format A25
column versions_endtime format A25
select emp_id, salary, versions_xid ,
       versions_starttime, versions_endtime
 from emp_salary
versions between scn minvalue and maxvalue
where emp_id=5;
prompt We will sleep for 4 more seconds and try again
begin
  dbms_lock.sleep(4);
end;
/
prompt Let us also update the row again to clarify the updates to that row
update emp_salary set salary=salary*1.25 where emp_id=5 ;
update emp_salary set salary=salary*1.25 where emp_id=4 ;
commit;
select emp_id, salary, versions_xid ,
       versions_starttime, versions_endtime
 from emp_salary
versions between scn minvalue and maxvalue
where emp_id=5;
prompt We will sleep for 4 more seconds and try again
prompt we will save the xid in to a variable
prompt and then query the activity from that transaction
prompt
variable l_versions_xid varchar2(30)
begin
select versions_xid into :l_versions_xid
from (
select versions_xid
from emp_salary
versions between scn minvalue and maxvalue
where emp_id=5
and versions_xid is not null
) where rownum <2
;
end;
/

```

```

prompt Flashback query uses undo technology and so undo_retention must
prompt be sufficiently kept high
prompt
prompt QUERY #1 : Print SQLs from a transaction
prompt
prompt Following xid from the previous query will be passed
select :l_versions_xid XID from dual;
set echo on
select start_timestamp , logon_user, undo_sql
from flashback_transaction_query
where xid = hexraw (:l_versions_xid )
;
set echo off

```

**Script:** flashback\_version\_table.sql

```

-----
-- Script : flashback_table.sql
-----
-- This script is to populate tables to describe flashback
-- Author : Riyaj Shamsudeen
-- No implied or explicit warranty !
-- Note:
-----

prompt
prompt Populating emp_salary table....
prompt
set feedback off
drop table emp_salary;
create table emp_salary
( emp_id number,
  salary number
)
;
prompt For the flashback table to work, we need to have row movement enabled
prompt
alter table emp_salary enable row movement;
prompt
prompt Sleeping for 4 seconds
prompt SCN to mapping is at 3 seconds precision
prompt We sleep for 4 seconds so that we can get different timestamps
prompt
begin
  dbms_lock.sleep(4);
end;
/
insert into emp_salary (emp_id, salary) values ( 1, 2000);
insert into emp_salary (emp_id, salary) values ( 2, 3000);
insert into emp_salary (emp_id, salary) values ( 3, 4000);
insert into emp_salary (emp_id, salary) values ( 4, 5000);
insert into emp_salary (emp_id, salary) values ( 5, 6000);
commit;
REM set feedback on
prompt Initial version of rows..
select * from emp_salary;
prompt
prompt Sleeping for 4 seconds

```

```

prompt
begin
  dbms_lock.sleep(4);
end;
/
prompt
prompt We will capture the current scn, so that we can refer to this point in time
prompt
column curscn noprint new_value currentscn
select to_char(current_scn,'999999999999') curscn , scn_to_timestamp (current_scn)
current_time
from v$database;
pause 'press any key to continue'
prompt We will update the table adding 25% to emp_id in (4,5)
update emp_salary set salary=salary*1.25 where emp_id=5 ;
update emp_salary set salary=salary*1.25 where emp_id=4 ;
commit;
prompt
prompt Another transaction to update a row again!
prompt
update emp_salary set salary=salary*1.25 where emp_id=5 ;
commit;
prompt Modified versions of row
select * from emp_salary;
set lines 120
begin
  dbms_lock.sleep(4);
end;
/
prompt Flashback query uses undo technology and so undo_retention must
prompt be sufficiently kept high
prompt
prompt Flashback the table to prior scn
prompt
set echo on
flashback table emp_salary to scn &currentscn;
select * from emp_salary;
set echo off

```

**Script:** plsql\_table\_fnc\_01.sql

```

-----
-- Script : plsql_table_fnc_01a.sql
-----
-- This script is to show the table functions in 10g
-- No parallelism
-- Author : Riyaj Shamsudeen
-- No implied or explicit warranty !
-- Note:
-----
set timing on
drop function plsql_piplined_noparallel;
drop type item_data_tabtyp;

create or replace type item_data_typ
as object
( item varchar2(30),

```

```

        location number,
        qty number
    )
/

create or replace type item_data_tabtyp as table of item_data_typ
/

--create or replace function plsqli_piplined_noparallel ( l_item varchar2)
create or replace function plsqli_piplined_noparallel
return item_data_tabtyp PIPELINED
as
    l_item_data_tabtyp item_data_tabtyp := item_data_tabtyp ();
    indx number:=1;
    l_fib number :=0;
begin
    l_item_data_tabtyp.extend;
    for l_csr_items in (select item, location, qty from item_data )
    loop
        l_fib := next_Fib_N (round(l_csr_items.location) );
        l_item_data_tabtyp (indx) := item_data_typ ( l_csr_items.item, l_fib,
l_csr_items.qty );
        pipe row ( l_item_data_tabtyp (indx) );
        l_item_data_tabtyp.extend;
        indx := indx+1;
    end loop;
    return;
end;
/
select item, location, sum(qty) from
    table (cast (plsqli_piplined_noparallel as item_data_tabtyp))
group by item, location
/

```

**Script:** plsqli\_table\_fnc\_parallel.sql

```

-----
-- Script : plsqli_table_fnc_parallel.sql
-----
-- This script is to show the table functions in 10g
-- with parallelism
-- Author : Riyaj Shamsudeen
-- No implied or explicit warranty !
-- Note:
--   Need access to v$mystat.
--   grant select on v_$mystat to <user>
-----
-- We need one strongly typed ref cursor to be passed
-- Creating a package specs to specify ref cursor
set timing on
alter table item_Data parallel (degree 5) ;
drop function plsqli_piplined_parallel;
drop type item_data_tabtyp;
create or replace package item_data_pkg
is
    type item_data_typ is RECORD (
        item varchar2(30),

```

```

        location number,
        qty number,
        sid number
    );
    type item_data_ref is ref cursor return item_data_typ;
end;
/
create or replace type item_data_typ
as object
( item varchar2(30),
  location number,
  qty number ,
  sid number
)
/

create or replace type item_data_tabtyp as table of item_data_typ
/

--create or replace function plsql_piplined_parallel
create or replace function plsql_piplined_parallel (
    l_item_data_ref in item_data_pkg.item_data_ref )
return item_data_tabtyp PIPELINED
parallel_enable (partition l_item_data_ref by range (location) )
as
    l_item_data_tabtyp item_data_tabtyp := item_data_tabtyp ();
    indx number:=1;
    l_fib number :=0;
    l_item varchar2(30);
    l_location number;
    l_qty number;

    l_sid number;
    l_dummy_sid number;
begin
    l_item_data_tabtyp.extend;

    select sid into l_sid from v$mystat where rownum <2;
    loop
        fetch l_item_data_ref into l_item , l_location, l_qty , l_dummy_sid ;
        exit when l_item_data_ref%NOTFOUND;
        l_fib := next_Fib_N (round(l_location) );
        l_item_data_tabtyp (indx) := item_data_typ ( l_item, l_fib, l_qty ,l_sid);
        pipe row ( l_item_data_tabtyp (indx) );
        l_item_data_tabtyp.extend;
        indx := indx+1;
    end loop;
    return;
end;
/
select /*+ parallel (5) */ item, location ,sid, sum(qty) from
    table (
        cast (
            plsql_piplined_parallel(
                cursor (
                    select item, location, qty ,0 from item_data

```

```

        )
      ) as item_data_tabtyp
    )
  )
group by item, location, sid
order by sid
/
alter table item_Data noparallel;

```

**Script:** plsql\_warnings\_01.sql

```

-----
-- Script : plsql_warnings_01.sql
-----
-- This script is to explain PL/SQL new feature: plsql_warnings
-- Author : Riyaj Shamsudeen
-- No implied or explicit warranty !
-- Note:
-----
rem Enabling plsql_warnings for all warnings.
REM This can be done at session level

alter session set plsql_warnings='enable:all';

REM
REM Procedure with dead code
REM
prompt First let us drop the procedure
prompt PL/SQL warnings seems to work only for first time creation.
prompt Recreating procedure with same code does not generate
prompt any warning.
prompt
prompt *** Drop and recreate procedure ***
prompt

drop procedure plsql_warnings_prc_01;

create or replace procedure plsql_warnings_prc_01
is
  l_id number :=1;
begin

  if (l_id =1) then
    dbms_output.put_line ('value of l_id is '|| l_id);
  else
    dbms_output.put_line ('Unreachable code ');
  end if;
end;
/

show errors
prompt
prompt *** Recreating second time ***
prompt

create or replace procedure plsql_warnings_prc_01
is

```

```

    l_id number :=1;
begin

    if (l_id =1) then
        dbms_output.put_line ('Value of l_id is '|| l_id);
    else
        dbms_output.put_line ('Unreachable code ');
    end if;
end;
/
show errors

```

**Script:** plsql\_warnings\_02.sql

```

-----
-- Script : plsql_warnings_02.sql
-----
-- This script is to explain PL/SQL new feature: plsql_warnings
-- Author : Riyaj Shamsudeen
-- No implied or explicit warranty !
-- Note:
-----
rem Enabling plsql_warnings for all warnings.
REM This can be done at session level

alter session set plsql_warnings='enable:all';

REM
REM Procedure with inefficient assignment
REM
prompt First let us drop the procedure
prompt PL/SQL warnings seems to work only for first time creation.
prompt Recreating procedure with same code does not generate
prompt any warning.
prompt
prompt *** Drop and recreate procedure ***
prompt

set serveroutput on
drop procedure plsql_warnings_prc_02;

create or replace procedure plsql_warnings_prc_02
is
    l_id number :=1;
begin
    for i in 1 .. 5
    loop
        l_id := 2;
    end loop;
end;
/

show errors
prompt
prompt *** Recreating second time ***
prompt

create or replace procedure plsql_warnings_prc_02

```

```

is
  l_id number :=1;
begin
  for i in 1 .. 5 loop
    l_id := 2;
  end loop;
end;
/
show errors
prompt
prompt *** Drop and recreate third time ***
prompt In this test, we add dbms_output and we can see that
prompt there are no warnings
prompt
drop procedure plsql_warnings_prc_02;
create or replace procedure plsql_warnings_prc_02
is
  l_id number :=1;
  l_second number;
begin
  for i in 1 .. 5
  loop
    l_id := 2;
    dbms_output.put_line ('i ='||i);
  end loop;
end;
/
show errors

```

**Script:** plsql\_warnings\_03.sql

```

-----
-- Script : plsql_warnings_03.sql
-----
-- This script is to explain PL/SQL new feature: plsql_warnings
-- Author : Riyaj Shamsudeen
-- No implied or explicit warranty !
-- Note:
-----
rem Enabling plsql_warnings for all warnings.
REM This can be done at session level

prompt alter session set plsql_warnings='enable:all';
alter session set plsql_warnings='enable:all';

REM
REM Procedure with infinite loop
REM
prompt First let us drop the procedure
prompt PL/SQL warnings seems to work only for first time creation.
prompt
prompt *** Drop and recreate procedure ***
prompt

set serveroutput on
drop procedure plsql_warnings_prc_03;

create or replace procedure plsql_warnings_prc_03

```

```

is
  l_id number :=1;
  l_lc number :=1;
begin
  while (l_lc <= 100)
  loop
    l_id := l_id *l_id;
  end loop;
end;
/
show errors
rem    dbms_output.put_line(l_id);

prompt
prompt Let's try different variation, of infinite loop
prompt

drop procedure plsqli_warnings_prc_03;

create or replace procedure plsqli_warnings_prc_03
is
  l_id number :=1;
  l_lc number :=1;
begin
  while (l_lc=1)
  loop
    l_id := l_id *l_id;
  end loop;
end;
/
show errors
rem    dbms_output.put_line(l_id);
prompt
prompt Same issue.. Engine did not detect infinite loop
drop procedure plsqli_warnings_prc_03;

create or replace procedure plsqli_warnings_prc_03
is
  l_id number :=1;
  l_lc number :=1;
begin
  while (l_lc=1)
  loop
    select 1 into l_id from dual;
  end loop;
end;
/
show errors
rem    dbms_output.put_line(l_id);
prompt
prompt Same issue.. Engine did not detect infinite loop in this case either.
prompt

```

Script: plsql\_warnings\_04.sql

```
-----
-- Script : plsql_warnings_04.sql
-----
-- This script is to explain PL/SQL new feature: plsql_warnings
-- Author : Riyaj Shamsudeen
-- No implied or explicit warranty !
-- Note:
-----
-----
rem Enabling plsql_warnings for all warnings.
REM This can be done at session level

prompt alter session set plsql_warnings='enable:all';
alter session set plsql_warnings='enable:all';

REM
REM Procedure with character to number assignments
REM
prompt First let us drop the procedure
prompt PL/SQL warnings seems to work only for first time creation.
prompt Recreating procedure with same code does not generate
prompt any warning.
prompt
prompt *** Drop and recreate procedure ***
prompt

set serveroutput on
drop procedure plsql_warnings_prc_04;
prompt
prompt This procedure assigns non-numeric value to a variable
prompt defined as number
prompt
create or replace procedure plsql_warnings_prc_04
is
    l_id number :=1;
    l_lc number :=1;
begin
    l_id := 'ORCL';
    l_id := l_id + 1;
end;
/
show errors
prompt
prompt This procedure assigns text larger than the variable can hold
prompt
drop procedure plsql_warnings_prc_04;
create or replace procedure plsql_warnings_prc_04
is
    l_id varchar2(2) ;
    l_lc number :=1;
begin
    l_id := 'ORCL';
    l_id := l_id + 1;
end;
```

```

/
show errors
prompt
prompt There are no warnings raised ;-(
prompt

Script: plsql_warnings_05.sql
-----
-- Script : plsql_warnings_05.sql
-----
-- This script is to explain PL/SQL new feature: plsql_warnings
-- Author : Riyaj Shamsudeen
-- No implied or explicit warranty !
-- Note:
-----
rem Enabling plsql_warnings for all warnings.
REM This can be done at session level

prompt alter session set plsql_warnings='enable:all';
alter session set plsql_warnings='enable:all';

prompt This test case measures whether we get any warnings
prompt if we try to pass huge variable i.e. blob
prompt
prompt First let us drop the procedure
prompt PL/SQL warnings seems to work only for first time creation.
prompt Recreating procedure with same code does not generate
prompt any warning.
prompt
prompt *** Drop and recreate procedure ***
prompt

set serveroutput on
drop procedure plsql_warnings_prc_05;
prompt
prompt This procedure accepts clob as in out paramteter
prompt defined as number
prompt
create or replace procedure plsql_warnings_prc_05
  ( l_emp_jpeg in out blob )
is
  l_id number :=1;
  l_lc number :=1;
begin
  l_id := 'ORCL';
  l_id := l_id + 1;
end;
/
show errors

```

```

Script: plsql_warnings_07.sql
-----
-- Script : plsql_warnings_07.sql
-----
-- This script is to explain PL/SQL new feature: plsql_warnings
-- Author : Riyaj Shamsudeen
-- No implied or explicit warranty !

```

```

-- Note:
-----
rem Enabling plsql_warnings for all warnings.
REM This can be done at session level

prompt alter session set plsql_warnings='enable:all';
alter session set plsql_warnings='enable:all';

prompt This test case measures whether we get any warnings
prompt if we try to pass huge variable i.e. blob
prompt
prompt First let us drop the procedure
prompt PL/SQL warnings seems to work only for first time creation.
prompt Recreating procedure with same code does not generate
prompt any warning.
prompt
prompt *** Drop and recreate procedure ***
prompt

set serveroutput on
drop procedure plsql_warnings_prc_07;
prompt
prompt This procedure uses key word CV and let us see Oracle
prompt can detect this
prompt
create or replace procedure plsql_warnings_prc_07
( l_emp_id in number )
is
  l_id number :=0;
  l_lc number :=1;
begin
  if (l_id <1) then
    goto test_var;
  end if;
  <<test_var >>
  begin
    if (l_id =0 ) then
      dbms_output.put_line('l_id is zero');
    end if;
  end;
end;
/
show errors

Script: plsql_warnings_07.sql
-----
-- Script : plsql_compile_01.sql
-----
-- This script is to explain PL/SQL new feature: plsql native compilation
-- Author : Riyaj Shamsudeen
-- No implied or explicit warranty !
-- Note:
-----
rem Enabling plsql_compile for all compile.
REM This can be done at session level
REM

```

```
alter session set plsql_code_type=native;
alter session set plsql_debug=false;
REM
prompt This must be altered at system level, you might need alter system privilege
REM
alter system set plsql_native_library_dir='/tmp';
REM
prompt Also native compiler must be set spnc_commands file in
prompt $ORACLE_HOME/plsql directory
show errors
prompt
prompt Creating procedure
create or replace procedure plsql_compile_prc_01
is
    l_id number :=1;
begin

    if (l_id =1) then
        dbms_output.put_line ('value of l_id is '|| l_id);
    else
        dbms_output.put_line ('Unreachable code ');
    end if;
end;
/
show errors
```