Lecture Notes
By Ammar Sajdi
Oracle 8I new features
(Draft)

CONSTRAINTS BEHAVIOUR

Primary key constraint is always associated with an index. By this I meant that whenever you create a primary key constraint, Oracle automatically create an index with it

Example

```
SQL> CREATE TABLE AMMAR (F1   NUMBER,
CONSTRAINT AMMAR_pk PRIMARY KEY (F1));
```

To verify that Oracle indeed created an index, run the following command

```
SQL> select index_name from user_indexes where
table_name='AMMAR';

INDEX_NAME
------------------------------
AMMAR_PK
```

Disable the primary key

```
SQL>ALTER TABLE AMMAR DISABLE PRIMARY KEY;

Table altered.

SQL> select index_name from user_indexes where
table_name='AMMAR';

no rows selected
```

The primary key is disabled and the index is clearly dropped

```
SQL> Insert into ammar values (1);

1 row created.

SQL> insert into ammar values (1);

1 row created.
```

The duplication is accepted because the primary key is disabled.

Let us try to enable the primary key

```
SQL> alter table ammar enable primary key;
alter table ammar enable primary key
*
ERROR at line 1:
ORA-02437: cannot validate (SCOTT.AMMAR_PK) - primary key
violated
```

Now, assume that you want to enable the primary key to check any future entry, but you do not this enablement to check the existing data in the database. For this, Oracle provided the ENABLE NOVALIDATE clause

Let us try to implement this feature

```
SQL> alter table AMMAR enable novalidate primary key
alter table AMMAR enable novalidate primary key
*
ERROR at line 1:
ORA-02437: cannot validate (SCOTT.AMMAR_PK) - primary key
violated
```

It is still trying to validate the existing data and ignoring the NOVALIDATE. Actually, it need the index that was associated with the primary key, after all, a primary key is going to be enabled and it is necessary for it to have an index. Therefore, let us create an index manually on F1

```
SQL>  create index ammar_ind on ammar(f1);

Index created.

SQL> alter table AMMAR enable novalidate primary key;

Table altered.
```

Try to verify that the primary key is actually enabled

```
SQL> insert into ammar values (1);
insert into ammar values (1)
*
ERROR at line 1:
ORA-00001: unique constraint (SCOTT.AMMAR_PK) violated
```

If you create a DEFERRABLE Primary Key constraint, then the index associated with it does not get dropped when the constraint is disabled. The following illustrates

```
SQL> create table ammar  (F1 number,
  2    constraint ammar_pk primary key (f1) deferrable);

Table created.

SQL> select index_name from user_indexes where
table_name='AMMAR';

INDEX_NAME
------------------------------
AMMAR_PK

SQL> alter table ammar disable primary key;

Table altered.
```

Let us verify that the index is still there.

```
SQL> select index_name from user_indexes where
table_name='AMMAR';

INDEX_NAME
------------------------------
AMMAR_PK
```

In this case therefore, we can enable novalidate primary indexes with creating an index in the manner that was done previously.

You can validate the data later on by

```
Alter table AMMAR modify primary key validate [exceptions
into EXCEPTIONS]
```

Since we used deferred constraints, let us study their behaviour. A deferrable constraint means that its effect can be deferred (postponed) until commit time. If a constraint is defined as deferrable, then the user can have the option of making the constraint either to act in the immediate mode, or in the deferred mode. If the constraint is not created as deferrable, then it can only act as immediate.

The following example illustrates:

```
SQL> select * from ammar;

        F1
----------
         1

SQL> insert into ammar values (1);
insert into ammar values (1)
*
ERROR at line 1:
ORA-00001: unique constraint (SCOTT.AMMAR_PK) violated
```

Even though the constraint is deferrable, but its validation took place immediately.

This is the intended behaviour. If you direct the constraint to act in the deferred mode then, it will give you what you want

```
SQL> set constraint ammar_pk deferred;

Constraint set.

SQL> insert into ammar values (1);

1 row created.
```

The insert is accepted because the constraint is not checked yet

```
SQL>commit;
```

```
ORA-00001: unique constraint (SCOTT.AMMAR_PK) violated
```

TEMPORARY TABLES

Oracle can create temporary tables to hold **session-private** data that exists only for the duration of

- Transaction
- Session.

The command to create a temporary Table, we use the command:-

CREATE GLOBAL TEMPORARY TABLE

The definition (or structure if you like) of the temporary table is visible to all sessions. However, the data in a temporary table is visible only to the session that inserts the data into the table. Other session can use the definition, but their data will be completely independent.

A temporary table has a definition that persists the same as the definitions of regular tables, but it contains either session-specific or transaction-specific data. You specify whether the data is session- or transaction-specific with the ON COMMIT keywords.

A TRUNCATE statement issued on a session-specific temporary table truncates data in its own session; it does not truncate the data of other sessions that are using the same table.

Segment Allocation

Temporary tables use temporary segments in the user's default temporary tablespace. Unlike permanent tables, temporary tables and their indexes do not automatically allocate a segment when they are created. Instead, segments are allocated when the first INSERT (or CREATE TABLE AS SELECT) is performed.

This means that if a SELECT, UPDATE, or DELETE is performed before the first INSERT, then the table appears to be empty.

Such temporary segments, if created in tablespace of type permanent, are deallocated at the end of the transaction for transaction-specific temporary tables and at the end of the session for session-specific temporary tables.
If created in tablespace of type temporary, then the segments will be available for reuse until the database is shutdown.

Example

```
SQL> create Global temporary table TEMP_TRANS
    (X1      number)
    On commit delete rows;

Table created.
```

This temporary table will delete its data automatically when the user commits the transaction

NOTE: YOU CANNOT SPECIFY STORAGE PARAMETERS OR TABLESPACE CLAUSE FOR TEMPORARY TABLES. IT WILL USE YOUR TEMPORARY TABLESPACE.

For example

```
SQL>Create Global temporary table TEMP_TRANS_test
    (X1      number) On commit delete rows
      storage (initial 100K)

create Global temporary table TEMP_TRANS_test
*
ERROR at line 1:
ORA-14451: unsupported feature with temporary table


SQL>create Global temporary table TEMP_SESSION
 (X1      number)
 On commit PRESERVE rows
/
```

This temporary table will delete its data automatically when the user exists the transaction.

Examples

In this example, we will insert a record to the TEMP_TRANS table which is a temporary table with transaction scope

```
SQL> INSERT INTO TEMP_TRANS VALUES (1000);

1 row created.
```

Now, we will insert another record to the TEMP_SESSION table, which is a temporary table with session scope

```
SQL> INSERT INTO TEMP_SESSION VALUES(1000);

1 row created.
```

The following statement will show that each table has only one record

```
SQL> SELECT * FROM TEMP_TRANS;

        X1
----------
 1000

SQL> SELECT * FROM TEMP_SESSION;

        X1
----------
     1000

SQL> COMMIT;

Commit complete.
```

UPON COMMIT, THE TEMP_TRANS SHOULD BE EMPTY, BUT THE TEMP_SESSION SHOULD STILL HOLD ITS DATA

```
SQL> SELECT * FROM TEMP_TRANS;

no rows selected

SQL> SELECT * FROM TEMP_SESSION;

        X1
----------
```

```
      1000
```

If another session queries the TEMP_SESSION, it will not be able to view this record because each session temporary data is kept separately

Now connect to a new session

```
SQL> connect scott/tiger@o8i
Connected.
SQL> SELECT * FROM TEMP_SESSION;

no rows selected
```

Clearly the data is cleared from the Session specified Temporary table

NOTE: You can build indexes and constraints on Temporary tables.

NOTE: ANALYZE Table has no effect on Temporary tables.
NOTE: Temporary tables generate less redo than a normal table because Oracle does not protect actual data or index in the temporary extent. The only redo generated is that associated with Rollback generated by DML for the temporary tables

Note: Try to export a schema that contains a temporary table. What do you think will happen? Since an export session is a new database session, it will export the temporary table structure, but no data will be export. This is an expected behaviour because the visibility for temporary tables does not extend beyond the scope of the session’s data. The same hold for replicating temporary tables

EXTERNAL TABLES (9i)

```
SQL> create table PALCO
  2  (comp_name  varchar2(10),
  3   HQ         varchar2(10),
  4   reg_no     number(10));

Table created.

SQL> create table palco_load
  2  (company_name  varchar2(10),
  3   Head_q  varchar2(10),
  4   registration number(10))
  5  organization external (Type Oracle_loader default
directory palco_Dir
  6  Access parameters (fields terminated by ",")
  7  location ('palco.dat'));

(Type Oracle_loader default directory palco_Dir
                                     *

ERROR at line 6:
ORA-06564: object PALCO_DIR does not exist
```

Therefore, we must create a directory object

```
SQL> connect system/manager
Connected.
SQL>
SQL> Create directory palco_Dir as 'c:\';

Directory created.
```

Therefore, palco_dir is a logical directory name that points to C:\.
We need to give SOCTT user READ writes on this directory.

```
SQL> grant read on directory palco_Dir to scott;

Grant succeeded.
```

You may also need to grant write privilege on the directory

```
SQL> grant write on directory palco_Dir to scott;


SQL>  create table palco_load
  2   (company_name  varchar2(10),
  3    Head_q  varchar2(10),
  4    registration number(10))
  5   organization external
  6   (Type Oracle_loader default directory palco_Dir
  7   Access parameters (fields terminated by ",")
  8   location ('palco.dat'));
```

The palco.dat file content is

```
C:\>type palco.dat
PALCO,AMMAN,1200
REALSOFT,AMMAR,3000
```

To load the data from PALCO_LOAD to PALCO execute the following standard SQL

```
SQL> insert into palco select * from palco_load;

2 rows created.

SQL> select * from palco;

COMP_NAME  HQ             REG_NO
---------- ---------- ----------
PALCO      AMMAN            1200
REALSOFT   AMMAR            3000
```

OPTIMIZER STABILITY

Optimizer stability is a feature introduced in Oracle8i to make SQL statement execution unaffected by changes in the factors the influence optimizer decisions. Such changes are new Oracle releases, changes in Init.ora parameters, index creations ,etc . In other words, it is an attempt to stabilize the execution plan with respect to elements that fuel changes in the execution plan.

The following example illustrates

```
SQL> select * from palco where reg_no=1234;


Execution Plan
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=CHOOSE
   1    0   TABLE ACCESS (FULL) OF 'PALCO'
```

The above execution plan illustrates that the table was accessed sequentially to fetch certain ROWS. Assume that we like this execution plan and we want Oracle to always choose this execution plan

1- Create an outline that will store the statements that you want to stabilize:

```
SQL>CREATE OR REPLACE OUTLINE CUST
     FOR CATEGORY TESTING ON
     SELECT * FROM PALCO WHERE REG_NO=1234;
```

2- Verify that the outline exists in the data dictionary

```
SQL> SELECT * FROM USER_OUTLINES;

NAME  CATEGORY   USED      TIMESTAMP VERSION    SQL_TEXT
----- ---------- --------- --------- ---------- -----------
---------------------------------
CUST  TESTING    UNUSED    10-DEC-01 8.1.6.0.0  SELECT *
FROM PALCO WHERE REG_NO=1234
```

3- Create and index on REG_NO of PALCO table.

```
SQL> create index sss on palco (reg_no);

Index created.
```

After creating the IND1 index, the Query will now use the index as shown below

```
SQL> select * from palco where reg_no=123;

Execution Plan
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=CHOOSE
   1    0   TABLE ACCESS (BY INDEX ROWID) OF 'PALCO'
   2    1     INDEX (RANGE SCAN) OF 'SSS' (NON-UNIQUE)
```

Obviously, Oracle optimizer was affected by the existence of this index

Using the stored outline we will make sure that the query will behave the same way it did when the outline was created. (ie before the index was created)

```
SQL>Alter session set use_stored_outlines=TESTING;
```

Now test if the query is really stabilized against the creation of the index.

```
SQL> select * from palco where reg_no=1234;
```

Note: The statement must be written exactly in the same way it was written when the outline was created (Similar case, spaces etc ..) to ensure the expected results

```
Execution Plan
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=CHOOSE (Cost=1 Card=1
Bytes=27)
   1    0   TABLE ACCESS (FULL) OF 'PALCO' (Cost=1 Card=1
Bytes=27)
```

It is now confirmed that the query works according to the same execution plan that the optimizer exhibited before the index was created

Where is the outline stored?

There is an Oracle created user called OUTLN. This user is automatically created upon installation.

```
SQL> CONNECT OUTLN/OUTLN
Connected

SQL> SELECT * FROM TAB;

TNAME                          TABTYPE  CLUSTERID
------------------------------ ------- ----------
OL$                            TABLE
OL$HINTS                       TABLE


SQL> select ol_name,category,signature, sql_text,
hash_Value from ol$;


SQL> col ol_name format a8
SQL> col category format a10
SQL> col sql_text format a40
```

```
OL_NAME  CATEGORY   SIGNATURE
-------- ---------- --------------------------------
SQL_TEXT                                 HASH_VALUE
---------------------------------------- ----------
CUST     TESTING    AE292D6535034ED1BBD21184E66B7CF4
select * from palco where reg_no=1234    2898428904
```

Oracle stores the statement syntax and includes appropriate hints to it that make sure that the statement will execute according to the current execution plan.

How can you stabilize the statement across different databases?

By export the OUTLN user and importing it in the other databases

Note: Never drop the OUTLN user

==

FUNCTION BASED INDEXES

INDEXES do not work if the indexed column is modified by Function or expression

*SQL> select * from ord where ord_id=10;*

```
Execution Plan
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=CHOOSE
   1    0   TABLE ACCESS (BY INDEX ROWID) OF 'ORD'
   2    1     INDEX (RANGE SCAN) OF 'SYS_C00862' (NON-
UNIQUE)
```

now modify the Ord_id by using an addition operation as shown below

*SQL> select * from ord where ord_id+0=10;*

```
Execution Plan
```

```
--------------------------------------------------------
   0      SELECT STATEMENT Optimizer=CHOOSE
   1    0   TABLE ACCESS (FULL) OF 'ORD'



SQL> create index ind2 on employees (first_name);

Index created.


SQL> select * from employees where first_name='AMMAR';

no rows selected


Execution Plan
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=CHOOSE
   1        TABLE ACCESS (BY INDEX ROWID) OF 'EMPLOYEES'
   2          INDEX (RANGE SCAN) OF 'IND2' (NON-UNIQUE)



SQL> select * from employees
     where lower(first_name)='ammar'


no rows selected


Execution Plan
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=CHOOSE
   1      0   TABLE ACCESS (FULL) OF 'EMPLOYEES'
   2

SQL> connect system/manager
Connected.

Make Sure that the user is granted the QUERY REWRITE Priv

SQL> GRANT QUERY REWRITE TO SCOTT;

Make sure that Query Rewrite is enabled for the user who is
going to create the index
```

```
SQL> connect scott/tiger
Connected.

SQL> ALTER SESSION SET QUERY_REWRITE_ENABLED=TRUE;

Session altered.

SQL> Create index ind3 on Employees (lower(First_name));



Now run the query again

SQL> select * from employees
     where lower(first_name)='ammar'


no rows selected


Execution Plan
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=CHOOSE (Cost=2
Card=152 Bytes=6080)
   1    0   TABLE ACCESS (BY INDEX ROWID) OF 'EMPLOYEES'
(Cost=2 Card= 152 Bytes=6080)
   2    1     INDEX (RANGE SCAN) OF 'IND3' (NON-UNIQUE)
(Cost=1 Card=152)
```

IMPORTANT NOTE:
Function based indexes do not work if Cost Based optimizer is not enabled. Therefore, make sure that your tables are analyzed. The following example illustrates

```
SQL> Create index ind3 on Emp (lower(ename))
SQL> /

Index created

SQL> analyze table emp compute statistics;

Table analyzed.

SQL> select * from emp where lower(ename)='ammar';

no rows selected
```

```
Execution Plan
----------------------------------------------------------
SELECT STATEMENT Optimizer=CHOOSE (Cost=1 Card=1 Bytes=32)
   TABLE ACCESS (FULL) OF 'EMP' (Cost=1 Card=1 Bytes=32)
```

The table contains around 100,000 records

```
SQL> select count(*) from emp;

  COUNT(*)
----------
   144688
```

Why is not the Ind3 index used?

The table is analyzed, but are the indexes analyzed? Let us analyze them

```
SQL>ANALYZE TABLE emp compute statistics
FOR ALL INDEXED COLUMNS


SQL>select * from emp where lower(ename)='ammar';

Execution Plan
----------------------------------------------------------
  SELECT STATEMENT Optimizer=CHOOSE
    TABLE ACCESS (BY INDEX ROWID) OF 'EMP'
     INDEX (RANGE SCAN) OF 'IND3' (NON-UNIQUE)
```

Now the index is used

INDEXES and SORTING

```
SQL> create index ind5 on ord (F1);

Index created.

SQL> select * from ord where f1 > 0 order by f1;

   ORD_ID   CUST_ID ORD_DATE  FILLED_DA    AMOUNT  STAFF_NO        F1
--------- --------- --------- --------- --------- --------- ---------
        1         2 19-APR-01 19-APR-01      1234        12         1
       14         4 19-APR-01 19-APR-01       800        19        14
       20         5 14-APR-01 15-APR-01       400        19        20
       21         5 14-APR-01 15-APR-01       560        24        21
       22         4 18-APR-01 18-APR-01       700        32        22
```

The following execution plan does not show an ORDER By Operation

```
Execution Plan
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=CHOOSE
   1    0   TABLE ACCESS (BY INDEX ROWID) OF 'ORD'
   2    1     INDEX (RANGE SCAN) OF 'IND5' (NON-UNIQUE)
```

Note: The ORDER BY Operation used the index to order the data. It did not repeat the sort operation. The execution plan above does not show any sort operation.

SQL> drop index ind5;

Index dropped.

*SQL> select * from ord where f1 > 0 order by f1;*

```
   ORD_ID    CUST_ID ORD_DATE  FILLED_DA    AMOUNT  STAFF_NO        F1
--------- --------- --------- --------- --------- --------- ---------
        1         2 19-APR-01 19-APR-01      1234        12         1
       14         4 19-APR-01 19-APR-01       800        19        14
       20         5 14-APR-01 15-APR-01       400        19        20
       21         5 14-APR-01 15-APR-01       560        24        21
       22         4 18-APR-01 18-APR-01       700        32        22
```

```
Execution Plan
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=CHOOSE
   1    0   SORT (ORDER BY)
   2    1     TABLE ACCESS (FULL) OF 'ORD'
```

Note: after the index is removed, the data is still ordered. The ordering is due to the order by Statement and not to the index. Note that the execution plan has SORT operation

OBJECT-RELATIONAL MODEL

This section will give an example driven explanation of Oracle’s Object-Relational Model. This model was introduced in Oracle8.

```
 SQL>Create table  student_info
 (St_no   number(5),
  st_name varchar2(20),
  pobox   varchar2(10),
  city    varchar2(10),
  country varchar2(10));
```

It is a good idea to think about the pobox, city and country fields as one piece of information describing the address of the student. This is how we can now create an object type called ADDRESS

```
SQL> create type Address as object
  2  ( pobox        varchar2(10),
  3   city          varchar2(10),
  4   country       varchar2(10))
  5  /

Type created.


SQL> drop table student_info;

Table dropped.
```

The student_info table is recreated and the 3 fields representing the address are substituted by one field called St_addr of ADDRESS type

```
SQL> create table student_info
  2  (st_no         number,
  3   st_name       varchar2(20),
  4   st_addr       address);

Table created.
```

st_addr is now called Object Column

```
SQL> desc student_info
 Name                            Null?    Type
 ------------------------------- -------- ----
 ST_NO                                    NUMBER
 ST_NAME                                  VARCHAR2(20)
 ST_ADDR                                  ADDRESS
```

```
SQL>INSERT INTO STUDENT_INFO VALUES
(1,'ammar',ADDRESS('17187','AMMAN','JORDAN'));
```

We insert data into the new object by called its CONSTRUCTOR FUNCTION, Which is a function automatically created with the object in order to create a new address

ADDRESS('17187','AMMAN','JORDAN')

Selecting

```
  1* select st_addr.pobox from student_info
SQL> /
select st_addr.pobox from student_info
               *
ERROR at line 1:
ORA-00904: invalid column name
```

Introducing an Alis and qualifying the column name

```
  1* select d.st_addr.pobox from student_info D
SQL> /

ST_ADDR.PO
----------
17187
```

```
SQL> Create or replace type family_member as Object
  2  (mem_id number,
  3   mem_name varchar2(30),
  4   mem_age  number);
  5  /

Type created.

SQL>
SQL> create table family of family_member;

Table created.
```

The above table is called **OBJECT TABLE.** It contains rows of objects only and not mixed with regular columns

```
SQL> desc family;
 Name                            Null?    Type
 ------------------------------- -------- ----
 MEM_ID                                   NUMBER
 MEM_NAME                                 VARCHAR2(30)
 MEM_AGE                                  NUMBER

SQL> Insert into family values (1,'Ammar',10);

1 row created.
```

NESTED TABLE EXAMPLE

```
SQL> create type item_type as object
  2  (prodid    number(5),
  3   descr     varchar2(30));
  4  .
SQL> /

Type created.

SQL> create type item_nst_type as table of item_type;
  2  /

Type created.

SQL> create table sales_order
  2  ( order_id number,
  3   supplier number,
  4  ship_Date date,
  5   items   item_nst_type)
  6   nested table items store as  AMMAR;

Table created.

SQL> desc ammar
 Name                            Null?    Type
 ------------------------------- -------- ----
 PRODID                                   NUMBER(5)
 DESCR                                    VARCHAR2(30)

SQL> desc sales_order
 Name                            Null?    Type
 ------------------------------- -------- ----
 ORDER_ID                                 NUMBER
 SUPPLIER                                 NUMBER
 SHIP_DATE                                DATE
 ITEMS                                    ITEM_NST_TYPE


SQL> insert into sales_order (order_id, supplier,
ship_Date) values (1,1,sysdate);

1 row created.

SQL> commit;

Commit complete.


insert into table (select d.items from sales_order d where
d.order_id=2) values (102,'DELL')


1*  select * from table (select d.items from sales_order d
where d.order_id=2)
```

DROPPING COLUMNS

```
SQL> alter table test set unused column f1;
```

The above statement will remove the def. Of f1 From the table definition. The data will remain in the database, but it is disassociated from its column. Therefore, you would expect this command to be fast

```
SQL> desc test;
 Name                            Null?    Type
 ------------------------------- -------- ----
 F2                                       NUMBER

SQL> select * from test;

       F2
---------
      100
```

Can you identify tables with columns marked as unused?

```
SQL> SELECT * FROM USER_UNUSED_COL_TABS;

TABLE_NAME                          COUNT
------------------------------ ----------
TEST                                    1
```

Can I add a column the table with the same name as the column that was just marked?

```
SQL> ALTER TABLE TEST ADD F1 NUMBER;

Table altered.
```

It means yes you can.

Note: When you export a table with columns marked as unused, the unused column will not be exported

The following statement will allow you to remove the column data from the database blocks

```
SQL> alter table test drop unused columns;

Table altered.
```

The above statement removes that actual data of the column from the database tablespaces.

The above two statements can be combined into one:-

```
SQL> alter table test drop column f1;
```

Assume that there is a check constraint on F2

```
SQL> alter table test drop column f2;

Table altered.


SQL>create table test (
f1 number,
f2 number,
constraint chk check (f1 > f2 ))




SQL> ALTER TABLE TEST DROP COLUMN F1;
ALTER TABLE TEST DROP COLUMN F1
                             *
ERROR at line 1:
ORA-12991: column is referenced in a multi-column
constraint
```

If Oracle allows you to drop f1, then how is the check constraint will be evaluated when it references a non existing column

```
SQL> r
  1* select constraint_name, search_condition from
user_constraints where constraint_name='CHK'

CONSTRAINT_NAME                SEARCH_CONDITION
------------------------------ --------------------
CHK                            f1 > f2


SQL> ALTER TABLE TEST DROP COLUMN F1 CASCADE CONSTRAINT;

Table altered.
```

The above statement will drop the column and any associated constraint, also it will drop any check constraints that refers to it.

```
SQL> SELECT CONSTRAINT_NAME, SEARCH_CONDITION FROM
```

```
USER_CONSTRAINTS WHERE CONSTRAINT_NAME='CHK';

no rows selected
```

Question: What if the dropped column is associated with an index.

Let us answer the question by an example

```
SQL>  CREATE TABLE ABC
  2   (F1 NUMBER(5),
  3    F2 VARCHAR2(10));

Table created.

SQL> CREATE INDEX ABC_IND ON ABC(F1);

Index created.

SQL> SELECT INDEX_NAME FROM USER_INDEXES WHERE
TABLE_NAME='ABC';

INDEX_NAME
------------------------------
ABC_IND

SQL> ALTER TABLE ABC DROP COLUMN F1;

Table altered.

SQL> SELECT INDEX_NAME FROM USER_INDEXES WHERE
TABLE_NAME='ABC';

no rows selected
```

Therefore, the index is dropped with its column. The same applies if you use `ALTER TABLE ABC SET UNUSED COLUMN`

Question: What if the dropped column is associated with a Primary Key

```
SQL> Drop table ABC
  2  ;
```

```
Table dropped.

SQL> CREATE TABLE ABC
  2  (F1 NUMBER(5),
  3   F2 VARCHAR2(10),
  4  CONSTRAINT ABC_PK  PRIMARY KEY (F1));

Table created.

SQL> SELECT CONSTRAINT_NAME FROM USER_CONSTRAINTS WHERE
TABLE_NAME='ABC';

CONSTRAINT_NAME
------------------------------
ABC_PK

SQL> ALTER TABLE ABC DROP COLUMN F1;

Table altered.

SQL> SELECT CONSTRAINT_NAME FROM USER_CONSTRAINTS WHERE
TABLE_NAME='ABC';

no rows selected
```

Therefore, the Primary Key is dropped with its column as well. The same applies if you use ALTER TABLE ABC SET UNUSED COLUMN

Question: What if the dropped column is associated with a Primary Key that is referenced by a foreign key

```
SQL> drop table ABC;

Table dropped.

SQL> CREATE TABLE ABC
 (F1 NUMBER(5),
  F2 VARCHAR2(10),
  CONSTRAINT ABC_PK  PRIMARY KEY (F1));

CREATE TABLE XYZ
```

```
 (X1    NUMBER(5),
  X2    VARCHAR2(10),
  X3    NUMBER(5) ,
 CONSTRAINT XYZ_FK FOREIGN KEY (X3) REFERENCES ABC (F1))
```

Verify that the foreign key constraint exists:-

```
SQL> SELECT CONSTRAINT_NAME FROM USER_CONSTRAINTS WHERE
TABLE_NAME='XYZ';

CONSTRAINT_N   AME
------------------------------
XYZ_FK

SQL> ALTER TABLE ABC DROP COLUMN F1;
ALTER TABLE ABC DROP COLUMN F1
                            *
ERROR at line 1:
ORA-12992: cannot drop parent key column
```

Obviously, the statement failed due to the reference from table XYZ.

Try the following

```
SQL> ALTER TABLE ABC DROP COLUMN F1 CASCADE CONSTRAINTS;

Table altered.
```

Verify the existence of the Foreign key

```
SQL>  SELECT CONSTRAINT_NAME FROM USER_CONSTRAINTS WHERE
TABLE_NAME='XYZ';

no rows selected
```

The foreign key was clearly dropped

Verify the primary key at the ABC table

```
SQL> SELECT CONSTRAINT_NAME FROM USER_CONSTRAINTS WHERE
TABLE_NAME='ABC';
```

```
no rows selected
```

It is also dropped.

So the effect of the command ALTER TABLE ABC DROP COLUMN F1 CASCADE CONSTRAINTS, was dropping the foreign key on the referencing table (XYZ), dropping the primary key (on ABC) and then dropping the actual column. The same applies if you use ALTER TABLE ABC SET UNUSED COLUMN.

NOTE: Dropping column generates a large amount of Rollback and Redo. You can limit the amount of Rollback by using the CHECKPOINT option (Alter table ABC drop column f1 CHECKPOINT 500) The integer after CHECKPOINT indicates the number of rows that will be processed before commit. If you use CHECKPOINT by did not specify an integer, Oracle will commit every 512 rows

MONITORING DML ACTIVITY
Monitoring the number of Inserts and updates and deletes.

```
CREATE TABLE TEST_MONITOR
 (T1    NUMBER,
  T2    VARCHAR2(10)) MONITORING;
```

The *approximate* count of inserts, updates, and deletes is kept in the SGA for the table TEST_MONITOR create above. Be careful with this kind of statistics, because if you issue a rollback against transaction that you made, the statistics will not reflect the rollback effect.

There is a dictionary view called USER_TAB_MODIFICATIONS (or DBA_/ALL_) where Oracle keeps the running total of the changes. This is done by SMON every 3 hours or at clean shutdown.

```
SQL> DESC user_tab_modifications;
 Name                                      Null?    Type
 ----------------------------------------- -------- -------
 TABLE_NAME
 VARCHAR2(30)
 PARTITION_NAME
 VARCHAR2(30)
 SUBPARTITION_NAME
 VARCHAR2(30)
 INSERTS                                                  NUMBER
 UPDATES                                                  NUMBER
 DELETES                                                  NUMBER
 TIMESTAMP                                                DATE
 TRUNCATED
 VARCHAR2(3)
```

BITMAP INDEXES

ANALYZE TABLE ... Compute statistics FOR ALL INDEXED COLUMNS
Rule base optimizer cannot use Bitmap indexex

Hint INDEX(table bitmap_index_name)

TRUST

```
PRAGMA RESTRICT_REFERENCES ( Function_name, WNDS [, WNPS]
[, RNDS] [, RNPS] [, TRUST] );
```

Where:

WNDS
Writes no database state (does not modify database tables).

RNDS
Reads no database state (does not query database tables).

WNPS
Writes no package state (does not change the values of packaged variables).

RNPS
Reads no package state (does not reference the values of packaged variables).

TRUST

Allows easy calling from functions that do have RESTRICT_REFERENCES declarations to those that do not.

Note. The usage of RESTRICT REFERENCES is allowed in packages only

Let us try to create a package using PRAGMA RESTRICT_REFERENCES WNDS (basically does allow writing to database. Within this package, we will create a function that tries to update the DEPT table

```
SQL>create or replace package PALCO1 is
  function get_name return varchar2;
  pragma restrict_references (get_name, WNDS);
end;
```

The function get_name uses and Update statement, and therefore violates the WNDS restriction

```
SQL> create or replace package body PALCO1 is
     function get_name return varchar2 is
      begin
       update dept set dname='xxx' where deptno=10;
     end;
    end;
    /

Warning: Package Body created with compilation errors.

SQL> show error
Errors for PACKAGE BODY PACK1:

LINE/COL ERROR
-------- -----------------------------------------------------------
0/0      PL/SQL: Compilation unit analysis terminated
2/2      PLS-00452: Subprogram 'GET_NAME' violates its
associated pragma
```

```
CREATE OR REPLACE PACKAGE PALCO2 IS
   FUNCTION X (X1 NUMBER) RETURN NUMBER;
   FUNCTION Y (Y1 NUMBER) RETURN NUMBER;
   PRAGMA RESTRICT_REFERENCES(Y,WNDS,TRUST); -- Only for Y
END;

CREATE OR REPLACE PACKAGE BODY P1a IS
   FUNCTION Y (Y1 NUMBER) RETURN NUMBER IS
   BEGIN
      RETURN X(X1);  -- THIS STATEMENT CALLED FUNCTION X
   END;
END;
```

In this case Oracle will not check the Body of PALCO2 for verify it adherence to WNDS restriction, rather it will trust that it actually adheres. Therefore, TRUST keyword is just a promise.

This way Y can call X even though X does not have RESTRICT_REFERECE PRAGMA

NEW AGGREGATE SQL FUNCTION

```
  1* select deptno, job, sum(salary) from staff group by
rollup(deptno , job)
SQL> /

   DEPTNO JOB                             SUM(SALARY)
--------- ------------------------------ -----------
       10 BELLBOY                               1800
       10 CLERK                                 1200
       10 MANAGER                               4300
       10                                       7300
       20 BELLBOY                               3500
       20 CHEF                                  3700
       20 CLERK                                 1800
       20 MANAGER                               1500
       20 STEWARD                                750
       20                                      11250
       30 BELLBOY                                650
       30 CASHIER                               1200
       30 CLERK                                 1700
       30 STEWARD                               3500
       30                                       7050
       40 BELLBOY                               2450
       40 CASHIER                                800
       40 CHEF                                  2250
       40                                       5500
                                               31100
```

```
20 rows selected.
```

```
  1* select deptno, job, sum(salary) from staff group by
rollup(job,deptno)
SQL> /

   DEPTNO JOB                              SUM(SALARY)
--------- ------------------------------ -----------
       10 BELLBOY                               1800
       20 BELLBOY                               3500
       30 BELLBOY                                650
       40 BELLBOY                               2450
          BELLBOY                               8400
       30 CASHIER                               1200
       40 CASHIER                                800
          CASHIER                               2000
       20 CHEF                                  3700
       40 CHEF                                  2250
          CHEF                                  5950
       10 CLERK                                 1200
       20 CLERK                                 1800
       30 CLERK                                 1700
          CLERK                                 4700
       10 MANAGER                               4300
       20 MANAGER                               1500
          MANAGER                               5800
       20 STEWARD                                750
       30 STEWARD                               3500
          STEWARD                               4250
                                               31100
```

```
1* select deptno, job, sum(salary) from staff group by
cube(deptno, job)
SQL> /

   DEPTNO JOB                 SUM(SALARY)
--------- ----------------- -----------
       10 BELLBOY                  1800
       10 CLERK                    1200
       10 MANAGER                  4300
       10                          7300
       20 BELLBOY                  3500
       20 CHEF                     3700
```

SUM(SAL)
Group by
DEPTNO

```
20 CLERK              1800
20 MANAGER            1500
20 STEWARD            750
20                    11250
30 BELLBOY            650
30 CASHIER            1200
30 CLERK              1700
30 STEWARD            3500
30                    7050
40 BELLBOY            2450
40 CASHIER            800
40 CHEF               2250
40                    5500
   BELLBOY            8400
   CASHIER            2000
   CHEF                5950
   CLERK               4700
   MANAGER             5800
   STEWARD             4250
                                        31100
```

Sum(Sal) group by JOB

Ranking Functions

```
  1* select job, salary , rank() over (order by salary
desc) rank from staff
SQL> /

JOB                               SALARY      RANK
------------------------------ --------- ---------
MANAGER                             2200         1
CHEF                                2200         1
STEWARD                             2200         1
CLERK                               1800         4
BELLBOY                             1800         4
BELLBOY                             1800         4
BELLBOY                             1700         7
MANAGER                             1500         8
CHEF                                1500         8
CLERK                               1500         8
CHEF                                1450        11
BELLBOY                             1250        12
CLERK                               1200        13
BELLBOY                             1200        13
```

```
CASHIER                              1200        13
MANAGER                              1200        13
MANAGER                               900        17
CASHIER                               800        18
CHEF                                  800        18
STEWARD                               750        20
STEWARD                               700        21

JOB                                SALARY      RANK
------------------------------ --------- ---------
BELLBOY                               650        22
STEWARD                               600        23
CLERK                                 200        24
```

What is the OVER() function.
The over() function is a window. Just imagine that the qualifying records are grouped in windows based on the order by clause inside the window)

```
Execution Plan
----------------------------------------------------------
SELECT STATEMENT Optimizer=CHOOSE (Cost=1038 Card=114688)
 WINDOW (SORT)
  TABLE ACCESS (FULL) OF 'EMP' (Cost=102 Card=114688 Bytes)


SQL)  select deptno, sum(salary), rank() over(order by
sum(salary) desc) rank from staff
  2* group by deptno

   DEPTNO SUM(SALARY)      RANK
--------- ----------- ---------
       20       11250         1
       10        7300         2
       30        7050         3
       40        5500         4
```

Windowing Function

```
SQL> select deptno, staff_name , salary, sum(salary) over
     (order by staff_no desc) running_Tot
     from staff
```

```
   DEPTNO STAFF_NAME     SALARY RUNNING_TOT
--------- ---------- --------- -----------
       10 Dian            1800        1800
       40 Chris           1450        3250
       30 Debbie          2200        5450
       20 Tom             1500        6950
       10 Jerry           1200        8150
       20 Corrine         1800        9950
       30 Ronald           200       10150
       40 Anne            1250       11400
       30 Ron             1200       12600
       20 Davin            750       13350
       10 Susan            900       14250
       20 Karen           1800       16050
       30 Jake             650       16700
       40 Kathy            800       17500
       30 Paul             600       18100
       20 Judith          2200       20300
       10 John            2200       22500
       20 Simone          1700       24200
       30 Todd            1500       25700
       40 Brian           1200       26900
       40 Don              800       27700

   DEPTNO STAFF_NAME     SALARY RUNNING_TOT
--------- ---------- --------- -----------
       30 Jessie           700       28400
       20 Steve           1500       29900
       10 Martin          1200       31100

24 rows selected.


SQL> Select deptno, staff_name , salary, count(*) over
     (order by staff_no desc) running_Tot
     from staff

   DEPTNO STAFF_NAME     SALARY RUNNING_TOT
--------- ---------- --------- -----------
       10 Dian            1800           1
       40 Chris           1450           2
       30 Debbie          2200           3
       20 Tom             1500           4
       10 Jerry           1200           5
       20 Corrine         1800           6
       30 Ronald           200           7
       40 Anne            1250           8
       30 Ron             1200           9
       20 Davin            750          10
       10 Susan            900          11
       20 Karen           1800          12
       30 Jake             650          13
       40 Kathy            800          14
       30 Paul             600          15
       20 Judith          2200          16
       10 John            2200          17
       20 Simone          1700          18
       30 Todd            1500          19
       40 Brian           1200          20
       40 Don              800          21

   DEPTNO STAFF_NAME    SALARY RUNNING_TOT
--------- ---------- --------- -----------
       30 Jessie           700          22
       20 Steve           1500          23
       10 Martin          1200          24
```

```
Select job , staff_name  , max(salary) over (partition by
     job) as max1 from staff
```

```
JOB                            STAFF_NAME      MAX1
------------------------------ ---------- ---------
BELLBOY                        Brian           1800
BELLBOY                        Simone          1800
BELLBOY                        Jake            1800
BELLBOY                        Corrine         1800
BELLBOY                        Dian            1800
BELLBOY                        Anne            1800
CASHIER                        Don             1200
CASHIER                        Ron             1200
CHEF                           Judith          2200
CHEF                           Kathy           2200
CHEF                           Chris           2200
CHEF                           Tom             2200
CLERK                          Martin          1800
CLERK                          Karen           1800
CLERK                          Ronald          1800
CLERK                          Todd            1800
MANAGER                        Steve           2200
MANAGER                        John            2200
MANAGER                        Jerry           2200
MANAGER                        Susan           2200
STEWARD                        Jessie          2200
STEWARD                        Paul            2200
STEWARD                        Debbie          2200
STEWARD                        Davin           2200
```

```
SQL>SELECT  A.DEPTNO,  A.DEPT_COUNT/B.TOT_COUNT COUNT1,
A.DEPT_SAL/B.TOT_SAL TOT1 FROM
(SELECT DEPTNO , COUNT(*) DEPT_COUNT, SUM(SALARY) DEPT_SAL
FROM STAFF GROUP BY DEPTNO) A,
 (SELECT COUNT(*) TOT_COUNT, SUM(SALARY) TOT_SAL FROM
STAFF) b

   DEPTNO    COUNT1      TOT1
--------- --------- ---------
       10 .20833333 .23472669
       20 .29166667 .36173633
       30 .29166667  .2266881
       40 .20833333 .17684887
```

```
SQL>SELECT ROWNUM AS RANK , DEPTNO , TOTAL  FROM (SELECT
     DEPTNO, SUM(SALARY) TOTAL FROM  STAFF
     GROUP BY DEPTNO ORDER BY 2 DESC)
  WHERE ROWNUM < 3


     RANK    DEPTNO     TOTAL
--------- --------- ---------
        1        20     11250
        2        10      7300
```

```
SELECT STAFF_NAME , SALARY FROM STAFF A
WHERE  7 >   (SELECT COUNT(*) FROM STAFF B
WHERE a.SALARY <= B.SALARY)
ORDER BY 2 DESC

STAFF_NAME    SALARY
---------- ---------
John            2200
Judith          2200
Debbie          2200
Karen           1800
Corrine         1800
```

```
Dian            1800
```

More RANK FUNCTION

A single query block can contain more than one ranking function, each partitioning the data into different groups (that is, reset on different boundaries). The groups can be mutually exclusive. The following query ranks products based on their dollar sales within each region (rank_of_product_per_region) and over all regions (rank_of_product_total).

```
SELECT r_regionkey, p_productkey, SUM(s_amount) AS
SUM_S_AMOUNT,
  RANK() OVER (PARTITION BY r_regionkey
              ORDER BY SUM(s_amount) DESC)
AS rank_of_product_per_region,
  RANK() OVER (ORDER BY SUM(s_amount) DESC)
               AS rank_of_product_total
FROM product, region, sales
WHERE r_regionkey = s_regionkey AND p_productkey =
s_productkey
GROUP BY r_regionkey, p_productkey
ORDER BY r_regionkey;
```

The query produces this result:

```
R_REGIONKEY    P_PRODUCTKEY    SUM_S_AMOUNT  RANK_OF_PRODUCT_PER_REGION RK_PRODUCT_TOTAL
-----------    ------------    ------------  --------------------        -------------
EAST           SHOES                    130                    1                1
EAST           JACKETS                   95                    2                4
EAST           SHIRTS                    80                    3                6
EAST           SWEATERS                  75                    4                7
EAST           T-SHIRTS                  60                    5                1
EAST           TIES                      50                    6                2
EAST           PANTS                     20                    7                4
WEST           SHOES                    100                    1                2
WEST           JACKETS                   99                    2                3
WEST           T-SHIRTS                  89                    3                5
WEST           SWEATERS                  75                    4                7
WEST           SHIRTS                    75                    4                7
WEST           TIES                      66                    6               10
WEST           PANTS                     45                    7               13
```

Cube- and Rollup-group Ranking

Analytic functions, RANK for example, can be reset based on the groupings provided by a CUBE or ROLLUP operator.

It is useful to assign ranks to the groups created by CUBE and ROLLUP queries. See the CUBE/ROLLUP section, which includes information about the GROUPING function for further details. A sample query is:

```
SQL>SELECT r_regionkey, p_productkey, SUM(s_amount) AS
SUM_S_AMOUNT,
 RANK() OVER (PARTITION BY GROUPING(r_regionkey),
GROUPING(p_productkey)
ORDER BY SUM(s_amount) DESC) AS rank_per_cube
FROM product, region, sales
WHERE r_regionkey = s_regionkey AND p_productkey =
s_productkey
GROUP BY CUBE(r_regionkey, p_productkey)
ORDER BY GROUPING(r_regionkey), GROUPING(p_productkey),
r_regionkey;
```

It produces this result:

```
R_REGIONKEY  P_PRODUCTKEY   SUM_S_AMOUNT    RANK_PER_CUBE
-----------  ------------   ------------    -------------
EAST         SHOES                   130                1
EAST         JACKETS                  50               12
EAST         SHIRTS                   80                6
EAST         SWEATERS                 75                7
EAST         T-SHIRTS                 60               11
EAST         TIES                     95                4
EAST         PANTS                    20               14
WEST         SHOES                   100                2
WEST         JACKETS                  99                3
WEST         SHIRTS                   89                5
WEST         SWEATERS                 75                7
WEST         T-SHIRTS                 75                7
WEST         TIES                     66               10
WEST         PANTS                    45               13
EAST         NULL                    510                2
WEST         NULL                    549                1
NULL         SHOES                   230                1
NULL         JACKETS                 149                5
NULL         SHIRTS                  169                2
NULL         SWEATERS                150                4
NULL         T-SHIRTS                135                6
NULL         TIES                    161                3
NULL         PANTS                    65                7
NULL         NULL                   1059                1
```

Treatment of NULLs

NULLs are treated like normal values. Also, for the purpose

of rank computation, a NULL value is
assumed to be equal to another NULL value. Depending on the
ASC | DESC options provided for
measures and the NULLS FIRST | NULLS LAST option, NULLs
will either sort low or high and hence,
are given ranks appropriately. The following example shows
how NULLs are ranked in different cases:

```
SELECT s_productkey, s_amount,      RANK() OVER (ORDER BY
s_amount ASC NULLS FIRST) AS rank1, RANK() OVER (ORDER BY
s_amount ASC NULLS LAST)  AS rank2,  RANK() OVER (ORDER BY
s_amount DESC NULLS FIRST)AS rank3,RANK() OVER (ORDER BY
s_amount DESC NULLS LAST) AS rank4
FROM sales;
```

The query gives the result:

```
S_PRODUCTKEY   S_AMOUNT   RANK1    RANK2    RANK3    RANK4
------------   --------   -----    -----     ----    -----
SHOES               100       6        4        3        1
JACKETS             100       6        4        3        1
SHIRTS               89       5        3        5        3
SWEATERS             75       3        1        6        4
T-SHIRTS             75       3        1        6        4
TIES               NULL       1        6        1        6
PANTS              NULL       1        6        1        6
```

CUME_DIST

The CUME_DIST function (defined as the inverse of
percentile in some statistical books) computes the
position of a specified value relative to a set of values.
The order can be ascending or descending.
Ascending is the default. The range of values for CUME_DIST
is from greater than 0 to 1. To compute the
CUME_DIST of a value x in a set S of size N, we use the
formula:

```
SELECT r_regionkey, p_productkey, SUM(s_amount) AS
SUM_S_AMOUNT,
   CUME_DIST() OVER
    (PARTITION BY r_regionkey
        ORDER BY SUM(s_amount))
     AS cume_dist_per_region
```

```
FROM region, product, sales
WHERE r_regionkey = s_regionkey AND p_productkey =
s_productkey
GROUP BY r_regionkey, p_productkey
ORDER BY r_regionkey, s_amount DESC;
```

It will produce this result:

```
R_REGIONKEY  P_PRODUCTKEY   SUM_S_AMOUNT     CUME_DIST_PER_REGION
-----------  ------------   ------------     --------------------
EAST         SHOES                   130                     1.00
EAST         JACKETS                  95                      .84
EAST         SHIRTS                   80                      .70
EAST         SWEATERS                 75                      .56
EAST         T-SHIRTS                 60                      .42
EAST         TIES                     50                      .28
EAST         PANTS                    20                      .14
WEST         SHOES                   100                     1.00
WEST         JACKETS                  99                      .84
WEST         T-SHIRTS                 89                      .70
WEST         SWEATERS                 75                      .56
WEST         SHIRTS                   75                      .28
WEST         TIES                     66                      .28
WEST         PANTS                    45                      .14
```

```
SQL>select deptno, job,sum(sal) ,
     cume_dist () over (partition by deptno order by
     sum(sal)) as cum_per_deptno
     from emp
     group by deptno,job

    DEPTNO JOB         SUM(SAL) CUM_PER_DEPTNO
---------- --------- ---------- --------------
        10 CLERK           1300           .333
        10 MANAGER         2450           .667
        10 PRESIDENT       5000          1.000
        20 CLERK           1900           .333
        20 MANAGER         2975           .667
        20 ANALYST         6000          1.000
        30 CLERK            950           .333
        30 MANAGER         2850           .667
        30 SALESMAN        5600          1.000

9 rows selected.
```

ROW_NUMBER

The ROW_NUMBER function assigns a unique number (sequentially, starting from 1, as defined by ORDER BY) to each row within the partition. It has the following syntax:

```
ROW_NUMBER() OVER
  ([PARTITION BY <value expression1> [, ...]]
    ORDER BY <value expression2> [collate clause]
[ASC|DESC]
      [NULLS FIRST | NULLS LAST] [, ...])
```

As an example, consider this query:

```
SELECT p_productkey, s_amount,
     ROW_NUMBER() (ORDER BY s_amount DESC NULLS LAST) AS
srnum
FROM product, sales
WHERE p_productkey = s_productkey;
```

It would give:

```
P_PRODUCTKEY      S_AMOUNT     SRNUM
------------      --------     -----
SHOES                  100         1
JACKETS                 90         2
SHIRTS                  89         3
T-SHIRTS                84         4
SWEATERS                75         5
JEANS                   75         6
TIES                    75         7
PANTS                   69         8
BELTS                   56         9
SOCKS                   45        10
SUITS                 NULL        11
```

```
 SQL>select deptno, sal ,ename ,
   row_number () over (order by ename) ser from emp
```

```
    DEPTNO        SAL ENAME             SER
---------- ---------- ---------- ----------
        20       1100 ADAMS               1
        30       1600 ALLEN               2
        30       2850 BLAKE               3
        10       2450 CLARK               4
        20       3000 FORD                5
        30        950 JAMES               6
        20       2975 JONES               7
        10       5000 KING                8
        30       1250 MARTIN              9
        10       1300 MILLER             10
        20       3000 SCOTT              11

    DEPTNO        SAL ENAME             SER
---------- ---------- ---------- ----------
        20        800 SMITH              12
        30       1500 TURNER             13
        30       1250 WARD               14
```

```
  SQL> select deptno, sal ,ename ,
   row_number () over (order by rowid) ser from emp
```

```
SQL> break on deptno skip 1
SQL>select deptno, sal ,ename ,
  row_number () over (partition by deptno order by rowid)
  AS ER from emp;

    DEPTNO        SAL ENAME             SER
---------- ---------- ---------- ----------
        10       2450 CLARK               1
                 5000 KING                2
                 1300 MILLER              3

        20        800 SMITH               1
                 2975 JONES               2
                 3000 SCOTT               3
                 1100 ADAMS               4
                 3000 FORD                5

        30       1600 ALLEN               1
                 1250 WARD                2
                 1250 MARTIN              3
                 2850 BLAKE               4
                 1500 TURNER              5
                  950 JAMES               6
```

```
  SQL>select deptno , ename ,
     sum(sal) over (order by deptno)
     from emp

    DEPTNO ENAME      SUM(SAL)OVER(ORDERBYDEPTNO)
---------- ---------- ---------------------------
        10 CLARK                             8750
           KING                              8750
           MILLER                            8750
        20 SMITH                                19625
           ADAMS                                19625
           FORD                                 19625
           SCOTT                                19625
           JONES                                19625

        30 ALLEN                                29025
           BLAKE                                29025
           MARTIN                               29025
           JAMES                                29025
           TURNER                               29025
           WARD                                 29025
```

```
SQL>select deptno , ename , sal,
   sum(sal) over (order by empno)
   from emp
```

```
    DEPTNO ENAME      SUM(SAL)OVER(ORDERBYEMPNO)
---------- ---------- --------------------------
        20 SMITH                             800
        30 ALLEN                            2400
        30 WARD                             3650
        20 JONES                            6625
        30 MARTIN                           7875
        30 BLAKE                           10725
        10 CLARK                           13175
        20 SCOTT                           16175
        10 KING                            21175
        30 TURNER                          22675
        20 ADAMS                           23775
        30 JAMES                           24725
        20 FORD                            27725
        10 MILLER                          29025

14 rows selected.
```

Explanation:= The first row is fetched, since there is order by on empno, all other employees with the same empno are fetched, and then a SUM operation is executed. This time it is executed on only one empno because empno is unique. The second empno is fetched, the same takes place, the window of calculation now slides from the first emp to the second employee. Therefore the summation takes place on the first and the second, etc that is why is doing running total.

```
SQL> break on deptno skip 1
 select deptno, sal,
  sum(sal) over (partition by deptno order by sal
               rows unbounded preceding)  xx from emp
```

```
    DEPTNO        SAL         XX
---------- ---------- ----------
        10       1300       1300
                 2450       3750
                 5000       8750

        20        800        800
                 1100       1900
                 2975       4875
                 3000       7875
                 3000      10875

        30        950        950
                 1250       2200
                 1250       3450
                 1500       4950
                 1600       6550
                 2850       9400
```

In this example, the analytic function SUM defines, for each row, a window that starts at the beginning of the partition(UNBOUNDED PRECEDING) and ends, by default, at the current row.

The following is an example of a cumulative balance per account ordered by deposit date.

```
SELECT Acct_number, Trans_date, Trans_amount,
       SUM(Trans_amount) OVER (PARTITION BY Acct_number
       ORDER BY Trans_date ROWS UNBOUNDED PRECEDING) AS
       Balance
     FROM Ledger
     ORDER BY Acct_number, Trans_date;
```

```
Acct_number    Trans_date    Trans_amount       Balance
-----------    ----------    ------------       -------
      73829    1998-11-01          113.45        113.45
      73829    1998-11-05          -52.01         61.44
      73829    1998-11-13           36.25         97.69
      82930    1998-11-01           10.56         10.56
      82930    1998-11-21           32.55         43.11
      82930    1998-11-29           -5.02         38.09
```

There is an example of a time-based window that shows, for each transaction, the moving average of transaction amount for the preceding 7 days of transactions:

```
SELECT Account_number, Trans_date, Trans_amount,
  AVG (Trans_amount) OVER
       (PARTITION BY Account_number ORDER BY Trans_date
            RANGE INTERVAL '7' DAY PRECEDING) AS mavg_7day
FROM Ledger;
```

```
Acct_number    Trans_date    Trans_amount    mavg_7day
-----------    ----------    ------------    ---------
      73829    1998-11-03          113.45       113.45
      73829    1998-11-09          -52.01        30.72
      73829    1998-11-13           36.25        -7.88
      73829    1998-11-14           10.56        -1.73
      73829    1998-11-20           32.55        26.45
      82930    1998-11-01          100.25       100.25
      82930    1998-11-10           10.01        10.01
      82930    1998-11-25           11.02        11.02
      82930    1998-11-26          100.56        55.79
      82930    1998-11-30           -5.02        35.52
```

Windowing Aggregate Functions with Logical Offsets

The following example illustrates how window aggregate functions compute values in the presence of duplicates.

```
SELECT r_rkey, p_pkey, s_amt
    SUM(s_amt) OVER
        (ORDER BY p_pkey RANGE BETWEEN 1 PRECEDING AND
CURRENT ROW) AS current_group_sum
FROM product, region, sales
WHERE r_rkey = s_rkey AND p_pkey = s_pkey AND r_rkey =
'east'
ORDER BY r_rkey, p_pkey;
```

```
R_RKEY   P_PKEY     S_AMT      CURRENT_GROUP_SUM  /*Source numbers for
the current_group_sum column*/
------   ------     -----      -----------------  /*-----          */
EAST          1       130      130                /* 130            */
EAST          2        50      180                /*130+50          */
EAST          3        80      265                /*50+(80+75+60)   */
EAST          3        75      265                /*50+(80+75+60)   */
EAST          3        60      265                /*50+(80+75+60)   */
EAST          4        20      235                /*80+75+60+20     */
```

```
SELECT t_timekey, s_amount,
       FIRST_VALUE(s_amount) OVER
              (ORDER BY t_timekey ROWS 1 PRECEDING) AS
LAG_physical,
       SUM(s_amount) OVER
```

```
          (ORDER BY t_timekey ROWS 1 PRECEDING) AS
MOVINGSUM,
FROM sales, time
WHERE sales.s_timekey = time.t_timekey
ORDER BY t_timekey;
```

It can yield either of the following:

```
T_TIMEKEY   S_AMOUNT     LAG_PHYSICAL      MOVINGSUM
---------   --------     -----------       ---------
92-10-11           1               1               1
92-10-12           4               1               5
92-10-12           3               4               7
92-10-12           2               3               5
92-10-15           5               2               7
```

Explanation := In the moving Sum, each row is added to the previous row

FIRST_VALUE AND LAST_VALUE FUNCTIONS

The FIRST_VALUE and LAST_VALUE functions help users derive full power and flexibility from the window aggregate functions. They allow queries to select the first and last rows from a window. These rows are specially valuable since they are often used as the baselines in calculations. For instance, with a partition holding sales data ordered by day, we might ask "How much was each day's sales compared to the first sales day (FIRST_VALUE) of the period?" Or we might wish to know, for a set of rows in increasing sales order, "What was the percentage size of each sale in the region compared to the largest sale (LAST_VALUE) in the region?"

REPROTING FUNCTION:

After a query has been processed, aggregate values like the number of resulting rows or an average value in a column can be easily computed within a partition and made available to other reporting functions.
Reporting aggregate functions return the same aggregate value for every row in a partition

RATIO_TO_REPORT

```
SELECT s_productkey, SUM(s_amount) AS sum_s_amount,
       SUM(SUM(s_amount)) OVER () AS sum_total,
       RATIO_TO_REPORT(SUM(s_amount)) OVER () AS
ratio_to_report
FROM sales
GROUP BY s_productkey;
```

with this result:

```
S_PRODUCTKEY    SUM_S_AMOUNT      SUM_TOTAL      RATIO_TO_REPORT
------------    ------------      ---------      ---------------
SHOES                    100            520                 0.19
JACKETS                   90            520                 0.17
SHIRTS                    80            520                  0.15
SWEATERS                  75            520                 0.14
SHIRTS                    75            520                 0.14
TIES                      10            520                 0.01
PANTS                     45            520                  0.08
SOCKS                     45            520                 0.08
```

CASE Statement

```
SQL>Select CASE when sal > 2000 then sal else sal*1.1 end
from emp
```

```
CASEWHENSAL>2000THENSALELSESAL*1.1END
-------------------------------------
                                  880
                                 1760
                                 1375
                                 2975
                                 1375
                                 2850
                                 2450
                                 3000
                                 5000
                                 1650
                                  1210
                                 1045
                                 3000
                                 1430
```

FINE GRAINED SECURITY

```
Connect dev02/oracle
```

```
CREATE OR REPLACE PACKAGE rest_security AS
 FUNCTION custid_sec (D1 VARCHAR2, D2 VARCHAR2)
 RETURN VARCHAR2;
 END;
 /

Package created.


CREATE OR REPLACE PACKAGE BODY rest_security AS
 /* limits SELECT  statements based on customer id */
 FUNCTION custid_sec (D1 VARCHAR2, D2 VARCHAR2)
 RETURN VARCHAR2
 IS
    d_predicate VARCHAR2(2000);
    BEGIN
       d_predicate := 'lname = SYS_CONTEXT(''userenv'',
''session_user'')';
       RETURN d_predicate;
 END custid_sec;
 END rest_security;
 .
 /

Package body created



SQL> grant execute on rest_security to public;

Grant succeeded.



begin
   DBMS_RLS.ADD_POLICY ('DEV02', 'customer', '', 'DEV02',
'rest_security.custid_sec', 'select')

 end;
ERROR at line 1:
ORA-28106: input value for argument #3 is not valid
ORA-06512: at "SYS.DBMS_RLS", line 0
ORA-06512: at line 2
Error due to NULL value in the third argument which is
supposed to be Policy name.
```

```
SQL>begin
     DBMS_RLS.ADD_POLICY ('DEV02', 'customer', 'XX',
'DEV02', 'rest_security.custid_sec’,’select’);
   end;
SQL>

PL/SQL procedure successfully completed.
```

Note: a dummy policy called xx is added

PL/SQL BULK LOADING:

Normal loading of 10000 records

```
SQL> create table test1 (F1 number);

Table created.

SQL> begin
  2    for i in 1 .. 10000 loop
  3        insert into test1 values (i);
  4    end loop;
  5  end;
  6
  7  .
SQL> set timing on
SQL> /

PL/SQL procedure successfully completed.

 real: 109126
SQL> commit;

Commit complete.
```

Using Bulk loading techniques

```
SQL> declare
  2     type arr is table of number index by
binary_integer;
  3     my_arr   arr;
  4  begin
  5     for i in 1 .. 10000 loop
  6         my_arr(i) := i;
  7     end loop;
  8  -- Bulk loading
  9  FORALL I IN MY_ARR.FIRST .. MY_ARR.LAST
 10      insert into test1 values (my_arr(i));
 11* end;
```

The performance should be much better

In the above example, we first declare an array called ARR, then we create a physical array of ARR type.
This physical array is called MY_ARR.
In Lies 5 till 9 , we initialize the array with values from 1 .. 10000. At line 9, we use the syntax for bulk loading which is the keyword **FORALL.** Note that the FORALL is not a loop statement

PL/SQL BULK Select

```
declare
   type array is table of emp.ENAME%TYPE index by
binary_integer;
   AMMAR  array;
 begin
   SELECT ENAME BULK COLLECT into AMMAR FROM EMP;
 end;
```

using Cursor approach

```
declare
   type arr1 is table of number index by binary_integer;
   ammar arr1;
   cursor c1 is select sal from emp;
 begin
    open c1;
    fetch c1 bulk collect into ammar; -- this bulk collect
    for i in ammar.first .. ammar.last loop
      dbms_output.put_line(ammar(i));
    end loop;
 end;
```

In the declaration section, a cursor and a PL/SQL table containing one column are declared as usual;

The fetch statement is not embedded in a loop, it is done in one round trip. After the statement is executed, Ammar PL/SQL table is populated with the SAL column.

The following example shows that you collect a specific number or rows(8.1.6)

```
DECLARE
TYPE ARRAY IS TABLE OF STRING(255) INDEX BY BINARY_INTEGER;
EMP_NAMES  ARRAY;
CURSOR C IS SELECT ENAME  FROM EMP;
BULK_LIMIT NUMBER;
BEGIN
OPEN C;

-- FIRST BULK 10
DBMS_OUTPUT.PUT_LINE('-- FIRST BULK 10');
BULK_LIMIT := 10;
FETCH C BULK COLLECT INTO EMP_NAMES LIMIT BULK_LIMIT;

FOR I IN EMP_NAMES.FIRST..EMP_NAMES.LAST LOOP
DBMS_OUTPUT.PUT_LINE(EMP_NAMES(I));
END LOOP;

-- SECOND BULK 15
DBMS_OUTPUT.PUT_LINE('-- SECOND BULK 15');
BULK_LIMIT := 15;
FETCH C BULK COLLECT INTO EMP_NAMES LIMIT BULK_LIMIT;

FOR I IN EMP_NAMES.FIRST..EMP_NAMES.LAST LOOP
  DBMS_OUTPUT.PUT_LINE(EMP_NAMES(I)); END LOOP;

CLOSE C;

END;
/
```

Collecting a record into a PL/SQL table of Records

```
  1  declare
  2    type array is table of emp%rowtype index by
  3        binary_integer;
  3    AMMAR  array;
  4    Cursor c1 is select * from emp;
  5  begin
  6    open c1;
  7    fetch c1 BULK COLLECT into AMMAR;
  8* end;
SQL> /
declare
*
ERROR at line 1:
ORA-06550: line 7, column 30:
PLS-00597: expression 'AMMAR' in the INTO list is of wrong
type
ORA-06550: line 7, column 3:
PL/SQL: SQL Statement ignored
```

Oracle 8i does not support the feature of bulk loading into PL/SQL table of Records.

BULK delete or Update

```
declare
  type array is table of number index by binary_integer;
  type array_char is table of varchar2(15) index by
binary_integer;
  v_deptno  array;
  v_sal     array;
  v_ename   array_char;
begin
   v_Deptno(1) := 10;
   v_deptno(2) := 20;
 forall i in 1 .. v_Deptno.count
    delete from emp where deptno = v_Deptno(i)
    returning ename, sal bulk collect into v_ename, v_Sal;
  for i in 1 .. 10 loop
    dbms_output.put_line(v_Ename(i));
 end loop;
```

```
end;
```

The example above we created a pl/sql the is populated with the department numbers to be deleted

The delete statement is a normal delete statement, except that it is embedded in FORALL and the deptno = array.
We are also taking advantage of a feature where you can fetch the values of the currently deleted record (using Bulk Collect) into PL/SQL tables. In the above example, each of v_ename, and V_sal is separate PL/SQL table because we cannot populate on PL/SQL table of Records as mentioned above

Dynamic SQL

Before Oracle 8I

```
create or replace procedure   test_it (x  varchar2 )
is
   rows        number;
   c1          integer;
   statement1  varchar2(100);
begin
   statement1   := 'CREATE TABLE '|| x ||'(f1
number)';
   c1 := dbms_sql.open_Cursor;
   dbms_sql.parse(c1,statement1,dbms_Sql.native);
   rows := dbms_sql.execute(c1);
end;
```

After 8I

```
create or replace procedure new_test (x varchar2) is
begin
  execute immediate 'create table ' ||x || '(f2 number)';
end;
```

INSERT EXAMPLE after 8I

Another example

```
CREATE TABLE dept_new
           (deptno NUMBER(2), dname VARCHAR2(14), loc
VARCHAR2(13));

Example of DBMS_SQL

CREATE OR REPLACE PROCEDURE dbms_example
  (deptnum IN dept_new.deptno%TYPE,
   deptname IN dept_new.dname%TYPE,
   location IN dept_new.loc%TYPE) IS

  stmt_str varchar2(100);
  rows_processed NUMBER;
  cur_hdl NUMBER;

BEGIN
  stmt_str := 'INSERT INTO dept_new VALUES(:deptno, :dname,
:loc)';
  cur_hdl := DBMS_SQL.OPEN_CURSOR;
  DBMS_SQL.PARSE(cur_hdl,stmt_str,DBMS_SQL.NATIVE);
  DBMS_SQL.BIND_VARIABLE(cur_hdl,':deptno',deptnum);
  DBMS_SQL.BIND_VARIABLE(cur_hdl,':dname', deptname);
  DBMS_SQL.BIND_VARIABLE(cur_hdl,':loc',location);

  rows_processed := dbms_sql.execute(cur_hdl);

  DBMS_SQL.CLOSE_CURSOR(cur_hdl);
END;
/
```

Example of Native Dynamic SQL

```
CREATE OR REPLACE PROCEDURE native_example
  (deptnum dept_new.deptno%TYPE,
   deptname dept_new.dname%TYPE,
   location dept_new.loc%TYPE) IS

  stmt_str varchar2(100);

BEGIN
  stmt_str :='INSERT INTO dept_new VALUES(:deptno, :dname,
:loc)';
  EXECUTE IMMEDIATE stmt_str USING deptnum, deptname,
location;
```

```
END;
/
```

```
Example of Fetching a single row using Native Dynamic SQL
declare
     stat varchar2(200);
     v_empno number;
     ret varchar2(20);
 begin
     stat := 'select ename from emp where empno = :b1';
     v_empno := 6408;
     EXECUTE IMMEDIATE stat INTO ret USING v_empno;
     dbms_output.put_line('Value fetched from table:
'||ret);
 end;
```

```
Value fetched from table: AMMAR
```

PL/SQL procedure successfully completed.
Repeating the example and returning result into PL/SQL record

```
declare
     stat varchar2(200);
     v_empno number;
     ret     emp%rowtype;
 begin
     stat := 'select * from emp where empno = :b1';
     v_empno := 6408;
     EXECUTE IMMEDIATE stat INTO  ret USING v_empno;
     dbms_output.put_line('Value fetched from table:
'||ret.ename);
 end;
```

```
Value fetched from table: AMMAR

PL/SQL procedure successfully completed.
```

```
Delete Example
declare
     stat varchar2(200);
     v_empno number;
     ret     varchar2(10);
 begin
     stat := ‘delete from emp where empno = :b1 returning
             empno into :b2';
     v_empno := 7499;
     EXECUTE IMMEDIATE stat USING v_empno, OUT ret;
     dbms_output.put_line('Successfully deleted
‘||SQL%ROWCOUNT || ‘ rows for empno: '||ret);
 end;
```

If you get the following error

```
declare
*
ERROR at line 1:
ORA-00904: invalid column name
ORA-06512: at line 8
```

Make sure that after the ‘Returning ‘ clause in the delete statement, you use a valid column name

Using Native Dynamic SQL to Call a stored function

Function : GET_NAME (empno IN number) Return varchar2

```
Declare
  VAL  number;
  RET  varchar2(100);
Begin
stat := 'begin :b1 := get_name(:b2); end;';
    val := 6408;
    EXECUTE IMMEDIATE stat USING OUT ret, IN val;
    dbms_output.put_line('The Name of employee ‘|| val ||’
is ‘||ret);
end;
```

```
The Name of employee 6408 is AMMAR

PL/SQL procedure successfully completed.
```

OR Alternatively you can use

```
stat := 'call get_name(:b2) into :b1';
    val := 6408;
    EXECUTE IMMEDIATE stat USING IN val, out ret;
```

- Please note above the IN val, Out ret have been swapped

Multi – row fetch

```
declare
    type    my_curs_type is REF CURSOR;  -- must be weakly typed
    curs    my_curs_type;
    str     varchar2(200);
    ret     varchar2(20);
begin
    str := 'select msg from msg';
    -- No placeholders so no USING clause
    OPEN curs FOR str;
    loop
        FETCH curs INTO ret;
        exit when curs%notfound;
        dbms_output.put_line(ret);
    end loop;
    CLOSE curs;
end;
```

Bulk dynamic SQL can be simulated using native dynamic SQL by placing the
bulk SQL within a BEGIN..END block and executing the block dynamically.

The following is an example of Bulk SQL using Native Dynamic SQL:

1. First create the necessary tables:

```
CREATE TABLE bulk1 (ename VARCHAR2(50))
CREATE TABLE bulk2 (ename VARCHAR2(50))
```

2. Next, insert data into bulk1:

```
INSERT INTO bulk1 VALUES('MARY JANE');
INSERT INTO bulk1 VALUES('JOHN DOE');
INSERT INTO bulk1 VALUES('MICHAEL DAVIS');
```

3. Next, create a VARRAY:

```
CREATE OR REPLACE TYPE name_array_type IS
   VARRAY(100) of VARCHAR2(50);
/
```

4. Next, create the procedure:

```
    CREATE OR REPLACE PROCEDURE copy_ename_column
      (table1 VARCHAR2, table2 VARCHAR2) IS
     ename_col NAME_ARRAY_TYPE;

     BEGIN
--bulk fetch the 'ename' column into a VARRAY of VARCHAR2s.
          EXECUTE IMMEDIATE
           'BEGIN
              SELECT ENAME BULK COLLECT INTO :tab
                FROM ' || table1 || ';
            END;'
         USING OUT ename_col;

   --bulk fetch the 'ename' column into another table.
       EXECUTE IMMEDIATE
        'BEGIN
          FORALL i IN :tab.first..:tab.last
             INSERT INTO ' || table2 || ' VALUES (:tab(i));
          END;'
         USING ename_col.first, ename_col.last, ename_col;
       END;
      /
```

5. Next, execute the procedure:

```
SQL> exec copy_ename_column('bulk1','bulk2');
```

6. Finally, verify results:

```
SQL> select * from bulk2;
```

MATERIALIZED VIEWS

Necessary Grants

SYSTEM/MANAGER

```
SQL> grant create materialized view to scott;

Grant succeeded.
```

```
SQL> grant query rewrite to scott;

Grant succeeded.

SQL>Connect Scott/tiger
```

(PLEASE NOTE: You cannot you the Syntax CREATE OR REPLACE MATERIALIZED VIEW, Only CREATE MATRIALIZED VIEW)

```
CREATE MATERIALIZED VIEW SAL_SUMMARY
ON PREBUILT TABLE
NEVER REFRESH
ENABLE QUERY REWRITE
AS
 SELECT DEPTNO, SUM(SAL), COUNT(*) FROM EMP
  GROUP BY DEPTNO

                 *
ERROR at line 6:
ORA-12059: prebuilt table "SCOTT"."SAL_SUMMARY" does not exist
```

```
CREATE MATERIALIZED VIEW SAL_SUMMARY
--ON PREBUILT TABLE
BUILD DEFERRED     -- will be created at the  next referesh
NEVER REFRESH
ENABLE QUERY REWRITE
AS
 SELECT DEPTNO, SUM(SAL), COUNT(*) FROM EMP
  GROUP BY DEPTNO



MATRIALEZED VIEW CREATED
```

Trying to use

```
SQL> alter session set query_rewrite_enabled=true;
SQL> alter session set query_rewrite_integrity=trusted;

QL> select deptno, count(*) from emp group by deptno;

   DEPTNO   COUNT(*)
--------- ----------
       30      40960
```

```
xecution Plan
----------------------------------------------------------
SELECT STATEMENT Optimizer=CHOOSE (Cost=682 Card=3 Bytes=6)
 SORT (GROUP BY) (Cost=682 Card=3 Bytes=6)
   TABLE ACCESS (FULL) OF 'EMP' (Cost=102 Card=114688 Bytes)
```

`Problem, the materialized group is not used`

```
CREATE MATERIALIZED VIEW SAL_SUMMARY
--ON PREBUILT TABLE
--BUILD DEFERRED     -- will be created at the  next referesh
--NEVER REFRESH
REFRESH FAST ON COMMIT
ENABLE QUERY REWRITE
AS
 SELECT DEPTNO, SUM(SAL), COUNT(*) FROM EMP
  GROUP BY DEPTNO
```

Or you can REFRESH COMPLETE instead of Refresh Fast

Refresh Modes

FAST	Specifies a fast (incremental) refresh mode, which uses only the updated data stored in the **materialized view log** associated with the master or detail table. The appropriate log must exist for the fast refresh to succeed unless you use direct-path load. Materialized view log has got SQL statement to create it. Oracle can perform a fast refresh only if all of the following conditions are true: • The materialized view's master table has a materialized view log or you used direct-load INSERT. (Oracle creates the direct loader log automatically. No user intervention is needed.) • The necessary log was created before the materialized view was last refreshed or created
COMPLETE	Specifies a complete refresh mode, or a refresh that reexecutes the materialized view's query. If you specify a complete refresh, Oracle performs a complete refresh regardless of whether a fast refresh is possible.

FORCE	Specifies a fast refresh if one is possible or complete refresh if a fast refresh is not possible. Oracle decides whether a fast refresh is possible at refresh time

	If you omit FAST, COMPLETE, and FORCE, Oracle uses FORCE by default

```
The following commands shows how you can refresh using a defined time
intervals

   REFRESH FAST
   START WITH 1-JUL-98
   NEXT SYSDATE +7 AS
```

To Sum up,

You create a materialized view and specify one of the refresh modes specified above, then you can specify ON COMMIT or ON DEMAND or START WITH

To create a materialized view log. A **materialized view log** is a table associated with the master table of a materialized view. When changes are made to the master table's data, Oracle stores rows describing those changes in the materialized view log and then uses the materialized view log to refresh materialized views based on the master table. This process is called a *fast refresh*. Without a materialized view log, Oracle must reexecute the materialized view query to refresh the materialized view. This process is called a *complete refresh*. Usually, a fast refresh takes less time than a complete refresh.