A COMPREHENSIVE GUIDE TO ORACLE PARTITIONING WITH SAMPLES

SUCCESSFUL ORACLE DATAWAREHOUSING AND BUSINESS INTELLIGENCE

ANTHONY D NORIEGA
anthony.noriega@adnmis.com
ADN R & D www.adnmis.com

ORACLE CERTIFIED PROFESSIONAL

ORACLE Technology NETWORK

SPEAKER QUALIFICATIONS

- Independent Consultant, ADN
- Speaker at NYOUG meetings, IOUG LIVE and Collaborate
- 24 years of IT experience
- 18 years of Oracle experience, 13 as a DBA (v6 thru 11g)
- RMAN experience with Oracle8i,9i, 10g, and 11g, since 1999.
- BS Systems Engineering, Universidad del Norte, 1987.
- MS Computer Science, NJIT, 1993
- PhD CIS candidate, NJIT, 1997
- MBA MIS, Montclair State University, 2006
- College Math Professor and former HS Math Teacher Principal.

OBJECTIVES

- Present the various types of object partitioning options with sample code.
- Derive a series of guidelines to existing and newly defined best practices.
- Provide a consolidated framework to partition-based performance tuning.
- Analyze and synthesize Oracle recommendations to object partitioning.
- Discuss various partitioning scenario-driven cases.

WHY AND WHEN TO PARTITION

- Table Size
- Table Access requires specific control
- Improve Index Performance
- Combining Technologies
  (Multiple block size caches)

OBJECT PARTITIONING

- Table (Includes Materialized Views)
- Index
- Index-Organized Table (IOT)

- **PARTITIONING STRATEGIES**

TABLE PARTITIONING

- Basic Partitions
- Composite Partitions
- Partition Extensions

TABLE PARTITIONING

- BASIC (Single Level)
  - Range (Includes Interval Partitioning)
  - List
  - Hash

TABLE PARTITIONING

- COMPOSITE
  - Range-Range
  - Range-Hash
  - Range-List
  - List-Range
  - List-Hash
  - List-List

INTERVAL PARTITIONING

- INTERVAL PARTITIONING
  - Interval-Range

PARTITION EXTENSIONS

- PARTITIONING KEY EXTENSION
  - Reference Partitioning
  - Virtual Column-Based Partition

INDEX PARTITIONING

- GLOBAL
  - Global Range Partitioned Indexes
  - Global Hash Partitioned Indexes
- LOCAL
  - Default
  - Customized (Tablespace and Block Size)

INDEX PARTITION MAINTENANCE

- **Operations on heap-organized tables marking all global indexes as unusable.**
  - **ADD (HASH)**
  - **COALESCE (HASH)**
  - **DROP**
  - **EXCHANGE**
  - **MERGE**
  - **MOVE**
  - **SPLIT**
  - **TRUNCATE**

IOT PARTITIONING

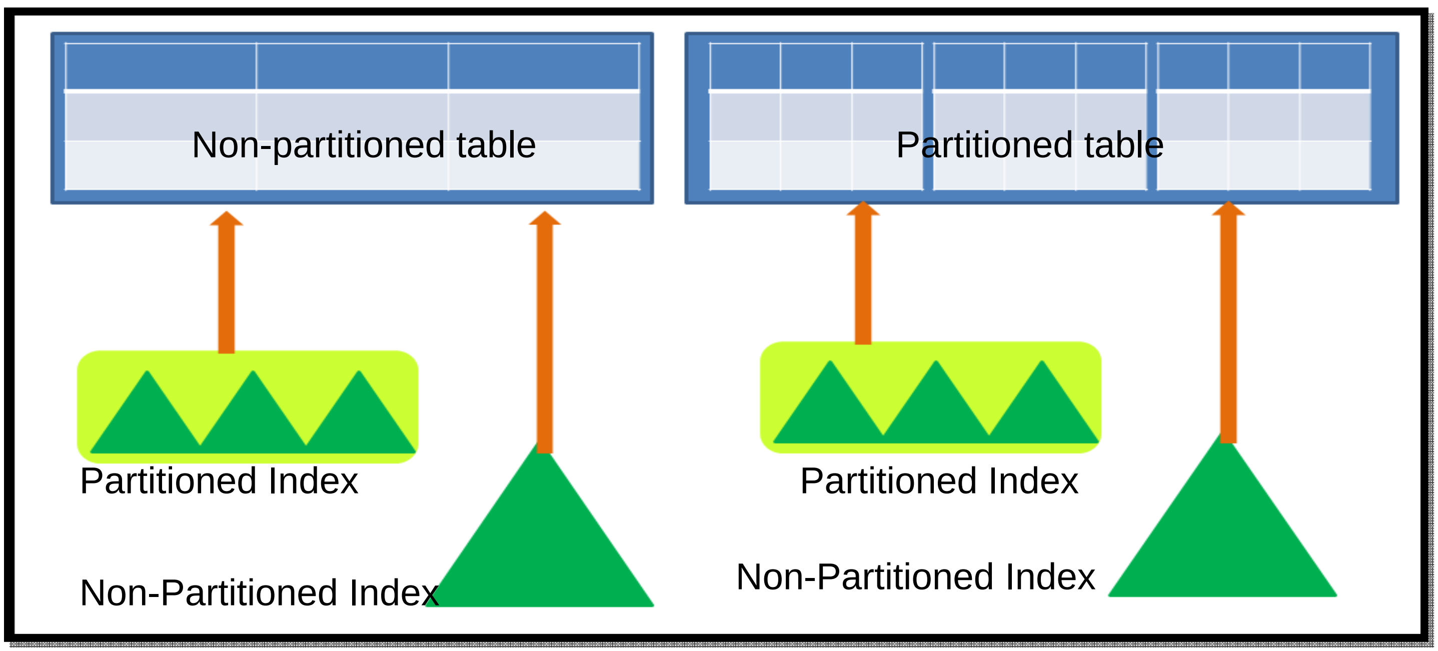
- INDEX-ORGANIZED TABLE (IOT) PARTITIONING
- Follows nearly the same options as table partitioning with some constraints.

CREATING PARTITIONS

- Creating a partitioned table or index is similar to creating a non-partitioned table or index, by adding a partitioning clause (and its subclauses, if any) to the CREATE TABLE statement.
- Partitioning is possible on both regular, i.e., heap organized tables, and index-organized tables (IOT), except for those containing LONG or LONG RAW columns. It is possible to create non-partitioned global indexes, range or hash-partitioned global indexes, and local indexes on partitioned tables.
- Specifying either ENABLE ROW MOVEMENT or DISABLE ROW MOVEMENT may be done when creating or altering a partitioned table.

CREATING PARTITIONS

- A partitioned table can have both partitioned and non-partitioned indexes.
- Likewise, a non-partitioned table can have both partitioned and non-partitioned indexes.



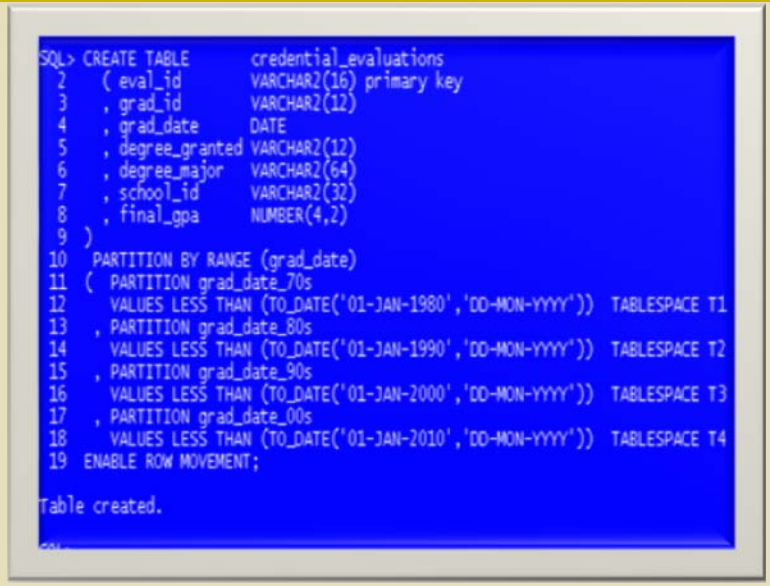
PARTITIONING COMBINATION STRATEGY

- Creating Range-Partitioned Tables and Global Indexes
- Creating Interval-Partitioned Tables
- Creating Hash-Partitioned Tables and Global Indexes
- Creating List-Partitioned Tables
- Creating Reference-Partitioned Tables
- Creating Composite Partitioned Tables
- Creating Partitioned Index-Organized Tables
- Using
  - Subpartition Templates to Describe Composite Partitioned Tables
  - Using Multicolumn Partitioning Keys
  - Using Virtual Column-Based Partitioning
  - Using Table Compression with Partitioned Tables
  - Using Key Compression with Partitioned Indexes
- Partitioning Restrictions for Multiple Block Sizes

CREATING A RANGE PARTITIONED TABLE

This sample code creates a table with four partitions and enables row movement:

```
CREATE TABLE      credential_evaluations
  ( eval_idVARCHAR2(16) primary key
  , grad_id             VARCHAR2(12)
  , grad_date           DATE
  , degree_granted  VARCHAR2(12)
  , degree_major      VARCHAR2(64)
  , school_id           VARCHAR2(32)
  , final_gpa            NUMBER(4,2))
 PARTITION BY RANGE (grad_date)
(  PARTITION grad_date_70s
   VALUES LESS THAN (TO_DATE('01-JAN-1980','DD-MON-YYYY'))  TABLESPACE T1
 , PARTITION grad_date_80s
   VALUES LESS THAN (TO_DATE('01-JAN-1990','DD-MON-YYYY'))  TABLESPACE T2
 , PARTITION grad_date_90s
   VALUES LESS THAN (TO_DATE('01-JAN-2000','DD-MON-YYYY'))  TABLESPACE T3
 , PARTITION grad_date_00s
   VALUES LESS THAN (TO_DATE('01-JAN-2010','DD-MON-YYYY'))  TABLESPACE T4 )
ENABLE ROW MOVEMENT;
```

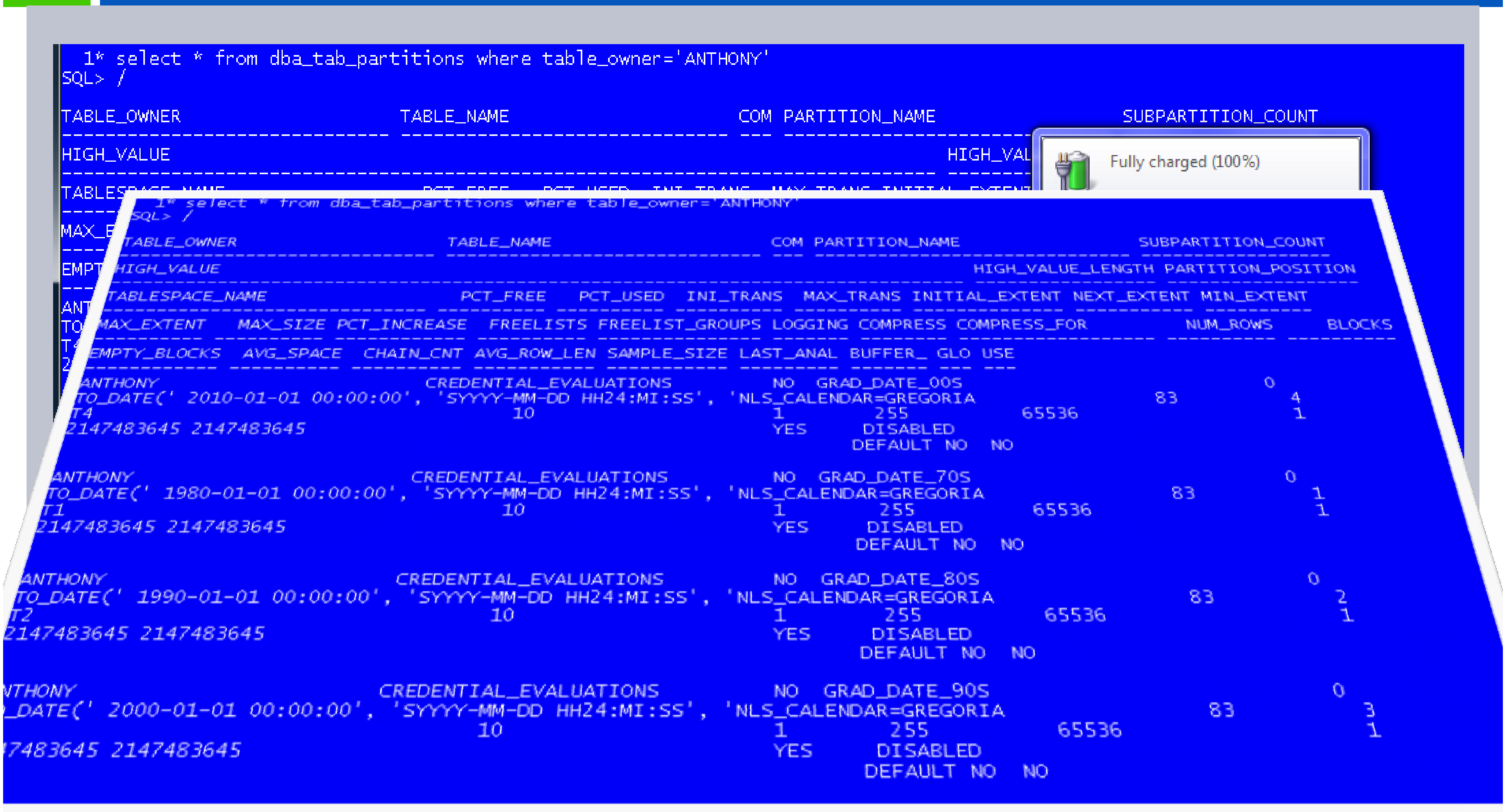


RANGE-PARTITIONED GLOBAL INDEX

- Creating a range-partitioned global index is similar to creating range-partitioned table. This example creates a range-partitioned global index on final_gpa for credential_evaluations. Each index partition is named but is stored in the default tablespace for the index.

```
CREATE INDEX ndx_final_gpa ON credential_evaluations
   (final_gpa)
  GLOBAL PARTITION BY RANGE(final_gpa)
    ( PARTITION c1 VALUES LESS THAN (2.5)
    , PARTITION c2 VALUES LESS THAN (3.0)
    , PARTITION b1 VALUES LESS THAN (3.4)
    , PARTITION b2 VALUES LESS THAN (3.7)
    , PARTITION a1 VALUES LESS THAN (3.9)
    , PARTITION a2 VALUES LESS THAN (MAXVALUE));
```

QUERYING DICTIONARY VIEWS



QUERYING DICTIONARY VIEWS

```
  1* SELECT * FROM dba_ind_partitions where index_owner='ANTHONY'
SQL> /

INDEX_OWNER                    INDEX_NAME                     COM PARTITION_NAME                 SUBPARTITION_COUNT
------------------------------ ------------------------------ --- ------------------------------ ------------------
HIGH_VALUE                                                                       HIGH_VALUE_LENGTH PARTITION_POSITION
-------------------------------------------------------------------------------- ----------------- ------------------
STATUS   TABLESPACE_NAME                  PCT_FREE  INI_TRANS  MAX_TRANS INITIAL_EXTENT NEXT_EXTENT MIN_EXTENT
-------- ------------------------------ ---------- ---------- ---------- -------------- ----------- ----------
MAX_EXTENT   MAX_SIZE PCT_INCREASE  FREELISTS FREELIST_GROUPS LOGGING COMPRESS     BLEVEL LEAF_BLOCKS DISTINCT_KEYS
---------- ---------- ------------ ---------- --------------- ------- -------- ---------- ----------- -------------
AVG_LEAF_BLOCKS_PER_KEY AVG_DATA_BLOCKS_PER_KEY CLUSTERING_FACTOR   NUM_ROWS SAMPLE_SIZE LAST_ANAL BUFFER_ USE
----------------------- ----------------------- ----------------- ---------- ----------- --------- ------- ---
PCT_DIRECT_ACCESS GLO DOMIDX
----------------- --- ------
PARAMETERS
------------------------------------------------------------------------------------------------------------------------
ANTHONY                        NDX_FINAL_GPA                  NO  C1                                              0
2.5                                                                                                      3          1
USABLE   USERS                                  10          2        255          65536                      1
2147483645 2147483645                                             YES     DISABLED          0           0              0
                      0                       0                     0          0          28-MAY-09 DEFAULT NO
                  NO


ANTHONY                        NDX_FINAL_GPA                  NO  C2                                              0
3.0                                                                                                      3          2
USABLE   USERS                                  10          2        255          65536                      1
2147483645 2147483645                                             YES     DISABLED          0           0              0
                      0                       0                     0          0          28-MAY-09 DEFAULT NO
                  NO


ANTHONY                        NDX_FINAL_GPA                  NO  B1                                              0
3.4                                                                                                      3          3
USABLE   USERS                                  10          2        255          65536                      1
2147483645 2147483645                                             YES     DISABLED          0           0              0
                      0                       0                     0          0          28-MAY-09 DEFAULT NO
                  NO


ANTHONY                        NDX_FINAL_GPA                  NO  B2                                              0
3.7                                                                                                      3          4
USABLE   USERS                                  10          2        255          65536                      1
2147483645 2147483645                                             YES     DISABLED          0           1              1
                      1                       1                     1          1        1 28-MAY-09 DEFAULT NO
                  NO


ANTHONY                        NDX_FINAL_GPA                  NO  A1                                              0
3.9                                                                                                      3          5
USABLE   USERS                                  10          2        255          65536                      1
2147483645 2147483645                                             YES     DISABLED          0           0              0
```

RANGE-PARTITIONED GLOBAL INDEXES

- Creating a range-partitioned global index is similar to creating range-partitioned table. This example creates a range-partitioned global index on final_gpa for credential_evaluations. Each index partition is named but is stored in the default tablespace for the index.

```
CREATE INDEX ndx_final_gpa ON credential_evaluations (final_gpa)
  GLOBAL PARTITION BY RANGE(final_gpa)
    ( PARTITION c1 VALUES LESS THAN (2.5)
    , PARTITION c2 VALUES LESS THAN (3.0)
    , PARTITION b1 VALUES LESS THAN (3.4)
    , PARTITION b2 VALUES LESS THAN (3.7)
    , PARTITION a1 VALUES LESS THAN (3.9)
    , PARTITION a2 VALUES LESS THAN (MAXVALUE));
```

RANGE-PARTITIONED GLOBAL INDEX

- Creating a range-partitioned global index is similar to creating range-partitioned table. This example creates a range-partitioned global index on final_gpa for credential_evaluations. Each index partition is named but is stored in the default tablespace for the index.

```
CREATE INDEX ndx_final_gpa ON credential_evaluations
    (final_gpa)
   GLOBAL PARTITION BY RANGE(final_gpa)
     ( PARTITION c1 VALUES LESS THAN (2.5)
     , PARTITION c2 VALUES LESS THAN (3.0)
     , PARTITION b1 VALUES LESS THAN (3.4)
     , PARTITION b2 VALUES LESS THAN (3.7)
     , PARTITION a1 VALUES LESS THAN (3.9)
     , PARTITION a2 VALUES LESS THAN (MAXVALUE));
```

INTERVAL-PARTITIONED TABLES

- The INTERVAL clause of the CREATE TABLE statement sets interval partitioning for the table. At least one range partition must be specified using the PARTITION clause.
- The range partitioning key value determines the high value of the range partitions (transition point) and the database automatically creates interval partitions for data beyond that transition point.
- For each interval partition, the lower boundary is the non-inclusive upper boundary of the previous range or interval partition.
- The partitioning key can only be a single column name from the table and it must be of NUMBER or DATE type.
- The optional STORE IN clause lets you specify one or more tablespaces.

INTERVAL-PARTITIONED TABLES

This example sets four partitions with varying widths. It also specifies that above the transition point of January 1, 2009, partitions are created with a width of one month.

```
CREATE TABLE rental_costs (
   item_id      NUMBER(6),
   time_intv    DATE,
   unit_cost    NUMBER(12,2),
   unit_price   NUMBER(12,2) )
PARTITION BY RANGE (time_intv)
  INTERVAL(NUMTOYMINTERVAL(1, 'MONTH'))
    ( PARTITION pca VALUES LESS THAN (TO_DATE('1-1-2006', 'DD-MM-YYYY')) tablespace  t2,
      PARTITION pcb VALUES LESS THAN (TO_DATE('1-1-2007', 'DD-MM-YYYY')) tablespace  t4,
      PARTITION pcc VALUES LESS THAN (TO_DATE('1-1-2008', 'DD-MM-YYYY')) tablespace  t8,
      PARTITION pcd VALUES LESS THAN (TO_DATE('1-1-2009', 'DD-MM-YYYY')) tablespace t12
);
```

The high bound of partition pcd establishes the transition point. pcd and all partitions below it, namely, (pca, pcb, and pcc) are in the range section while all partitions above it fall into the interval section.

HASH-PARTITIONED TABLES


```
CREATE TABLE school_directory
   (stid  NUMBER PRIMARY KEY,
    lname  VARCHAR2 (50),
    fname  VARCHAR2 (50),
    phone VARCHAR2(16),
    email VARCHAR2(128),
    class_year VARCHAR2(4))
  PARTITION BY HASH (stid)   PARTITIONS 4 STORE IN (t1, t2, t3, t4);
```

- The PARTITION BY HASH clause of the CREATE TABLE statement identifies that the table is to be hash-partitioned.
- The PARTITIONS clause can then be used to specify the number of partitions to create, and optionally, the tablespaces to store them in. Otherwise, PARTITION clauses can be used to name the individual partitions and their tablespaces .
- The only attribute TO specify for hash partitions is TABLESPACE. All of the hash partitions of a table must share the same segment attributes (except TABLESPACE), which are inherited from the table level.

CREATING LIST-PARTITIONED TABLES


```
CREATE TABLE regional_rentals
    (divno       NUMBER,
     divname   VARCHAR2(40),
     rentals_quarterly    NUMBER(12, 2),
     state         VARCHAR2(2))
  PARTITION BY LIST (state)
    (PARTITION pnw VALUES  ('OR', 'WA', 'WY') TABLESPACE T1,
     PARTITION psw  VALUES  ('AZ', 'CA', 'UT') TABLESPACE T3,
     PARTITION pne  VALUES  ('CT', 'NY', 'NJ') TABLESPACE T5,
     PARTITION pse   VALUES  ('FL', 'GA', 'SC') TABLESPACE T7);
```

- A PARTITION BY LIST clause is used in the CREATE TABLE statement to create a table partitioned by list, by specifying lists of literal values,(the discrete values of the partitioning columns qualifying rows matching the partition's single column partitioning key.) There is no sense of order among partitions.
- The DEFAULT keyword is used to describe the value list for a partition that will accommodate rows that do not map into any of the other partitions.
- Optional subclauses of a PARTITION clause can specify physical and other attributes specific to a partition segment. If not overridden at the partition level, partitions inherit the attributes of their parent table.

REFERENCE-PARTITIONED TABLES

- The PARTITION BY REFERENCE clause is used with the CREATE TABLE statement, specifying the name of a referential constraint, which becomes the partitioning referential constraint used as the basis for reference partitioning in the table. The referential integrity constraint must be enabled and enforced.
- It is possible to set object-level default attributes, and optionally specify partition descriptors that override the object-level defaults on a per-partition basis.
- When providing partition descriptors, the number of partitions described should match the number of partitions or subpartitions in the referenced table, i.e., the table will have one partition for each subpartition of its parent when the parent table is composite; otherwise the table will have one partition for each partition of its parent.
- No partition bounds can be set for the partitions of a reference-partitioned table.
- The partitions of a reference-partitioned table can be named, inheriting their name from the respective partition in the parent table, unless this inherited name conflicts with one of the explicit names given. In this scenario, the partition will have a system-generated name.
- Partitions of a reference-partitioned table will collocate with the corresponding partition of the parent table, if no explicit tablespace is set accordingly.

REFERENCE-PARTITIONED TABLES

Master Table

CREATE TABLE order_hist
(ord_id NUMBER(16),
ord_date TIMESTAMP WITH LOCAL TIME ZONE,
ord_mode VARCHAR2(8),
cust_id NUMBER(9),
ord_status VARCHAR2(4),
ord_total NUMBER(12,2),
act_mgr_id NUMBER(9),
promo_id NUMBER(8),
CONSTRAINT ord_pk PRIMARY KEY(ord_id) USING INDEX TABLESPACE INDX) **PARTITION BY RANGE(ord_date)**
(PARTITION pq1 VALUES LESS THAN (TO_TIMESTAMP_TZ('01-APR-2008 07:00:00 -5:00' , 'DD-MON-YYYY HH:MI:SS TZH:TZM')),
PARTITION pq2 VALUES LESS THAN (TO_TIMESTAMP_TZ('01-JUL-2008 07:00:00 -5:00' , 'DD-MON-YYYY HH:MI:SS TZH:TZM')),
PARTITION pq3 VALUES LESS THAN (TO_TIMESTAMP_TZ('01-OCT-2008 07:00:00 -5:00' , 'DD-MON-YYYY HH:MI:SS TZH:TZM')),
PARTITION pq4 VALUES LESS THAN (TO_TIMESTAMP_TZ('01-JAN-2009 07:00:00 -5:00' , 'DD-MON-YYYY HH:MI:SS TZH:TZM')));

Details Table

CREATE TABLE order_details
(ord_id NUMBER(16) NOT NULL,
line_item_id NUMBER(3) NOT NULL,
prod_id NUMBER(8) NOT NULL,
unit_price NUMBER(12,2),
qty NUMBER(8),
CONSTRAINT ord_det_fk
FOREIGN KEY(ord_id)
REFERENCES order_hist(ord_id)
)
PARTITION BY REFERENCE(ord_det_fk);

REFERENCE-PARTITIONED TABLES


```
SQL>
SQL> CREATE TABLE order_hist
      ( ord_id               NUMBER(16),
        ord_date             TIMESTAMP WITH LOCAL TIME ZONE,
        ord_mode             VARCHAR2(8),
        cust_id              NUMBER(9),
        ord_status           VARCHAR2(4),
        ord_total            NUMBER(12,2),
        act_mgr_id           NUMBER(9),
        promo_id             NUMBER(8),
        CONSTRAINT ord_pk PRIMARY KEY(ord_id) USING INDEX TABLESPACE INDX
      )
    PARTITION BY RANGE(ord_date)
      ( PARTITION pq1 VALUES LESS THAN (TO_TIMESTAMP_TZ('01-APR-2008 07:00:00 -5:00' , 'DD-MON-YYYY HH:MI:SS TZH:TZM')),
        PARTITION pq2 VALUES LESS THAN (TO_TIMESTAMP_TZ('01-JUL-2008 07:00:00 -5:00' , 'DD-MON-YYYY HH:MI:SS TZH:TZM')),
        PARTITION pq3 VALUES LESS THAN (TO_TIMESTAMP_TZ('01-OCT-2008 07:00:00 -5:00' , 'DD-MON-YYYY HH:MI:SS TZH:TZM')),
        PARTITION pq4 VALUES LESS THAN (TO_TIMESTAMP_TZ('01-JAN-2009 07:00:00 -5:00' , 'DD-MON-YYYY HH:MI:SS TZH:TZM'))
      );

Table created.

SQL>
SQL> CREATE TABLE order_details
      ( ord_id               NUMBER(16) NOT NULL,
        line_item_id         NUMBER(3)  NOT NULL,
        prod_id              NUMBER(8)  NOT NULL,
        unit_price           NUMBER(12,2),
        qty                  NUMBER(8),
        CONSTRAINT ord_det_fk
        FOREIGN KEY(ord_id) REFERENCES order_hist(ord_id)
      )
      PARTITION BY REFERENCE(ord_det_fk);

Table created.

SQL>
SQL>
```

LOCAL PARTITIONED INDEXES

When creating a local index for a table, the database constructs the index so that it is equipartitioned (1-to-1 correspondence) with the underlying table.

The database also ensures that the index is maintained automatically when maintenance operations are performed on the underlying table. This sample code creates a local index on the table dept:

```
CREATE INDEX ndx_gd ON school_directory (email) LOCAL
  PARTITIONS 4 STORE IN (t1, t2, t3, t4);
```

It is possible to optionally name the hash partitions and tablespaces into which the local index partitions are to be stored, otherwise, the database uses the name of the corresponding base partition as the index partition name, and stores the index partition in the same tablespace as the table partition.

HASH-PARTITIONED GLOBAL INDEX

- Hash-partitioned global indexes can improve the performance of indexes where a small number of leaf blocks in the index have high contention in multiuser OLTP environments.
- Hash-partitioned global indexes can also limit the impact of index skew on monotonously increasing column values. Queries involving the equality and IN predicates on the index partitioning key can efficiently use hash-partitioned global indexes.

The syntax is similar to that used for a hash partitioned table.

```
CREATE UNIQUE INDEX ndx_sch_dir ON
   school_directory (stid,phone,email) GLOBAL
    PARTITION BY HASH (stid,phone)
    (PARTITION psp1  TABLESPACE t1,
     PARTITION psp2  TABLESPACE t2,
     PARTITION psp3  TABLESPACE t4,
     PARTITION psp4  TABLESPACE t8);
```

RANGE-HASH PARTITIONED TABLES

In general, to create a composite partitioned table, use the PARTITION BY [**RANGE** | LIST] clause of a CREATE TABLE statement. Next, you specify a SUBPARTITION BY [RANGE | LIST | **HASH**] clause that follows similar syntax and rules as the PARTITION BY [RANGE | LIST | HASH] clause. The PARTITION and SUBPARTITION or SUBPARTITIONS clauses, and optionally a SUBPARTITION TEMPLATE clause.

```
CREATE TABLE  credential_evaluations
( eval_id  VARCHAR2(16)  primary key
 , grad_id          VARCHAR2(12)
 , grad_date        DATE
 , degree_granted VARCHAR2(12)
 , degree_major   VARCHAR2(64)
 , school_id        VARCHAR2(32)
 , final_gpa        NUMBER(4,2))
PARTITION BY RANGE (grad_date)
  SUBPARTITION BY HASH (grad_id)
SUBPARTITIONS 8 STORE IN (T1,T2,T3,T4)
(  PARTITION grad_date_70s
  VALUES LESS THAN (
TO_DATE('01-JAN-1980','DD-MON-YYYY'))
, PARTITION grad_date_80s
  VALUES LESS THAN (
TO_DATE('01-JAN-1990','DD-MON-YYYY'))
 , PARTITION grad_date_90s
  VALUES LESS THAN (
TO_DATE('01-JAN-2000','DD-MON-YYYY')),
PARTITION grad_date_00s
  VALUES LESS THAN (
TO_DATE('01-JAN-2010','DD-MON-YYYY'))
);
```

RANGE-HASH PARTITIONED TABLES

- The partitions of a range-hash partitioned table are logical structures only, as their data is stored in the segments of their subpartitions.
- As with partitions, these subpartitions share the same logical attributes.
- Unlike range partitions in a range-partitioned table, the subpartitions cannot have different physical attributes from the owning partition, but they can reside another tablespace.
- Attributes specified for a range partition apply to all subpartitions of that partition.
- Specify different attributes for each range partition
  Specify a STORE IN clause at the partition level if the list of tablespaces across which the subpartitions of that partition should be spread is different from those of other partitions.

```
CREATE TABLE emp ( deptno NUMBER ,   lname VARCHAR(32),
                                fname VARCHAR2(32) ,  grade NUMBER)
   PARTITION BY RANGE(deptno)  SUBPARTITION BY HASH(empname)
     SUBPARTITIONS 8 STORE IN (ts1, ts3, ts5, ts7)
  (PARTITION p1 VALUES LESS THAN (20000),
   PARTITION p2 VALUES LESS THAN (40000)
     STORE IN (t2, t4, t6, t8),  PARTITION p3 VALUES LESS THAN  (MAXVALUE)
    (SUBPARTITION p1_s1 TABLESPACE ts4,
     SUBPARTITION p3_s2 TABLESPACE ts5));
```

RANGE-LIST PARTITIONED TABLES


```
CREATE TABLE q_territory_sales
(   divno  VARCHAR2(12),  depno NUMBER,
      itemno  VARCHAR2(16),    accrual_date DATE,
      sales_amount NUMBER,     state VARCHAR2(2),
      constraint pk_q_dvdno primary key(divno,depno)
)    TABLESPACE t8   PARTITION BY RANGE
  (accrual_date)     SUBPARTITION BY LIST (state)
    (PARTITION q1_2000 VALUES LESS THAN
  (TO_DATE('1-APR-2000','DD-MON-YYYY'))
       (SUBPARTITION q1_2000_nw VALUES ('OR', 'WY'),
         SUBPARTITION q1_2000_sw VALUES ('CA', 'NM'),
         SUBPARTITION q1_2000_ne VALUES ('NY', 'CT'),
         SUBPARTITION q1_2000_se VALUES ('FL', 'GA'),
         SUBPARTITION q1_2000_nc VALUES ('SD', 'WI'),
         SUBPARTITION q1_2000_sc VALUES ('TX', 'LA') ),
PARTITION q2_2000 VALUES LESS THAN
  (TO_DATE('1-JUL-2000','DD-MON-YYYY'))
       (SUBPARTITION q2_2000_nw VALUES ('OR', 'WY'),
         SUBPARTITION q2_2000_sw VALUES ('CA', 'NM'),
         SUBPARTITION q2_2000_ne VALUES ('NY', 'CT'),
SUBPARTITION q2_2000_se VALUES ('FL', 'GA'),
         SUBPARTITION q2_2000_nc VALUES ('SD', 'WI'),
         SUBPARTITION q2_2000_sc VALUES ('TX', 'LA')
  ),  PARTITION q3_2000 VALUES LESS THAN
 (TO_DATE('1-OCT-2000','DD-MON-YYYY'))
       (SUBPARTITION q3_2000_nw VALUES ('OR', 'WY'),
         SUBPARTITION q3_2000_sw VALUES ('CA', 'NM'),
         SUBPARTITION q3_2000_ne VALUES ('NY', 'CT'),
         SUBPARTITION q3_2000_se VALUES ('FL', 'GA'),
         SUBPARTITION q3_2000_nc VALUES ('SD', 'WI'),
         SUBPARTITION q3_2000_sc VALUES ('TX', 'LA')
      ),  PARTITION q4_2000 VALUES LESS THAN (
 TO_DATE('1-JAN-2001','DD-MON-YYYY'))
       (SUBPARTITION q4_2000_nw VALUES ('OR', 'WY'),
         SUBPARTITION q4_2000_sw VALUES ('CA', 'NM'),
         SUBPARTITION q4_2000_ne VALUES ('NY', 'CT'),
         SUBPARTITION q4_2000_se VALUES ('FL', 'GA'),
         SUBPARTITION q4_2000_nc VALUES ('SD', 'WI'),
         SUBPARTITION q4_2000_sc VALUES ('TX', 'LA')
      ) );
```

This example illustrates the creation of a range-list partitioned table.

RANGE-LIST PARTITIONED TABLES


```
SQL> CREATE TABLE q_territory_sales
          (divno            VARCHAR2(12),
           depno            NUMBER,
           itemno           VARCHAR2(16),
           accrual_date DATE,
           sales_amount NUMBER,
           state            VARCHAR2(2),
           constraint pk_q_dvdno primary key(divno,depno)
          )
     TABLESPACE t8
     PARTITION BY RANGE (accrual_date)
       SUBPARTITION BY LIST (state)
          (PARTITION q1_2000 VALUES LESS THAN (TO_DATE('1-APR-2000','DD-MON-YYYY'))
             (SUBPARTITION q1_2000_nw VALUES ('OR', 'WY'),
              SUBPARTITION q1_2000_sw VALUES ('CA', 'NM'),
              SUBPARTITION q1_2000_ne VALUES ('NY', 'CT'),
              SUBPARTITION q1_2000_se VALUES ('FL', 'GA'),
              SUBPARTITION q1_2000_nc VALUES ('SD', 'WI'),
              SUBPARTITION q1_2000_sc VALUES ('TX', 'LA')
             ),
           PARTITION q2_2000 VALUES LESS THAN ( TO_DATE('1-JUL-2000','DD-MON-YYYY'))
             (SUBPARTITION q2_2000_nw VALUES ('OR', 'WY'),
              SUBPARTITION q2_2000_sw VALUES ('CA', 'NM'),
              SUBPARTITION q2_2000_ne VALUES ('NY', 'CT'),
              SUBPARTITION q2_2000_se VALUES ('FL', 'GA'),
              SUBPARTITION q2_2000_nc VALUES ('SD', 'WI'),
              SUBPARTITION q2_2000_sc VALUES ('TX', 'LA')
             ),
   PARTITION q3_2000 VALUES LESS THAN (TO_DATE('1-OCT-2000','DD-MON-YYYY'))
             (SUBPARTITION q3_2000_nw VALUES ('OR', 'WY'),
              SUBPARTITION q3_2000_sw VALUES ('CA', 'NM'),
              SUBPARTITION q3_2000_ne VALUES ('NY', 'CT'),
              SUBPARTITION q3_2000_se VALUES ('FL', 'GA'),
              SUBPARTITION q3_2000_nc VALUES ('SD', 'WI'),
              SUBPARTITION q3_2000_sc VALUES ('TX', 'LA')
             ),
           PARTITION q4_2000 VALUES LESS THAN ( TO_DATE('1-JAN-2001','DD-MON-YYYY'))
             (SUBPARTITION q4_2000_nw VALUES ('OR', 'WY'),
              SUBPARTITION q4_2000_sw VALUES ('CA', 'NM'),
              SUBPARTITION q4_2000_ne VALUES ('NY', 'CT'),
              SUBPARTITION q4_2000_se VALUES ('FL', 'GA'),
              SUBPARTITION q4_2000_nc VALUES ('SD', 'WI'),
              SUBPARTITION q4_2000_sc VALUES ('TX', 'LA')
             )
          )
  /

Table created.
```

CREATING LIST-HASH PARTITIONED TABLES

This example shows a car_rentals table that is list partitioned by territory and subpartitioned using hash by customer identifier.

```
CREATE TABLE car_rentals
( car_id              VARCHAR2(16)
, account_number      NUMBER
, customer_id         NUMBER
, amount_paid         NUMBER
, branch_id           NUMBER
, territory           VARCHAR(2)
, status              VARCHAR2(1)
)
PARTITION BY LIST (territory)
SUBPARTITION BY HASH (customer_id) SUBPARTITIONS 8
( PARTITION p_nw VALUES ('OR', 'WY') TABLESPACE T1
, PARTITION p_sw VALUES ('AZ', 'CA') TABLESPACE T2
, PARTITION p_ne VALUES ('NY', 'CT') TABLESPACE T3
, PARTITION p_se VALUES ('FL', 'GA') TABLESPACE T4
, PARTITION p_nc VALUES ('SD', 'WI') TABLESPACE T5
, PARTITION p_sc VALUES ('OK', 'TX') TABLESPACE T6
);
```

CREATING LIST-RANGE PARTITIONED TABLES

This sample code shows a car_rentals table that is list by territory and subpartitioned by range using the rental paid amount. Note that row movement is enabled.

```
CREATE TABLE car_rentals
 ( car_id         VARCHAR2(16)
 , account_number NUMBER
 , customer_id    NUMBER
 , amount_paid    NUMBER
 , branch_id      NUMBER
 , territory      VARCHAR(2)
 , status         VARCHAR2(1) )
 PARTITION BY LIST (territory)
 SUBPARTITION BY RANGE (amount_paid)
 ( PARTITION p_nw VALUES ('WA', 'WY')
   ( SUBPARTITION snwlow   VALUES LESS THAN (1000)
   , SUBPARTITION snwmid   VALUES LESS THAN (10000)
   , SUBPARTITION snwhigh  VALUES LESS THAN
      (MAXVALUE)   )
 ,  PARTITION p_ne VALUES ('NY', 'CT')
   ( SUBPARTITION snelow   VALUES LESS THAN (1000)
   , SUBPARTITION snemid   VALUES LESS THAN (10000)
   , SUBPARTITION snehigh  VALUES LESS THAN
      (MAXVALUE)
   )
 ,  PARTITION p_sw VALUES ('CA', 'AZ')
   ( SUBPARTITION sswlow   VALUES LESS THAN
      (1000)
   , SUBPARTITION sswmid   VALUES LESS THAN
      (10000)
   , SUBPARTITION sswhigh  VALUES LESS THAN
      (MAXVALUE)
   )
 ,  PARTITION p_se VALUES ('FL', 'GA')
   ( SUBPARTITION sselow   VALUES LESS THAN
      (1000)
   , SUBPARTITION ssemid   VALUES LESS THAN
      (10000)
   , SUBPARTITION ssehigh  VALUES LESS THAN
      (MAXVALUE)
   )
 );
```

LIST-LIST PARTITIONED TABLES


```
CREATE TABLE car_rentals_acct
 ( car_id        VARCHAR2(16)
 , account_number NUMBER
 , customer_id    NUMBER
 , amount_paid    NUMBER
 , branch_id      NUMBER
 , territory      VARCHAR(2)
 , status         VARCHAR2(1)
 , rental_date    TIMESTAMP WITH LOCAL TIME ZONE
 , constraint pk_car_rhist primary
      key(car_id,account_number,branch_id,rental_date)
 )
 PARTITION BY LIST (territory)
 SUBPARTITION BY LIST (status)
 ( PARTITION p_nw VALUES ('WA', 'WY')
   ( SUBPARTITION snw_low   VALUES ('C')
   , SUBPARTITION snw_avg   VALUES ('B')
   , SUBPARTITION snw_high  VALUES ('A')
   )
 ,
 PARTITION p_ne VALUES ('NY', 'CT')
   ( SUBPARTITION sne_low   VALUES ('C')
   , SUBPARTITION sne_avg   VALUES ('B')
   , SUBPARTITION sne_high  VALUES ('A')
   )
 , PARTITION p_sw VALUES ('CA', 'AZ')
   ( SUBPARTITION ssw_low   VALUES ('C')
   , SUBPARTITION ssw_avg   VALUES ('B')
   , SUBPARTITION ssw_high  VALUES ('A')
   )
 , PARTITION p_se VALUES ('FL', 'GA')
   ( SUBPARTITION sse_low   VALUES ('C')
   , SUBPARTITION sse_avg   VALUES ('B')
   , SUBPARTITION sse_high  VALUES ('A')
   )
 )
```

This sample code shows an car_rentals_acct table that is list-partitioned by territory and subpartitioned by list using account status column.

LIST-LIST PARTITIONED TABLES


```
SQL> ;
CREATE TABLE car_rentals_acct
( car_id          VARCHAR2(16)
, account_number NUMBER
, customer_id    NUMBER
, amount_paid    NUMBER
, branch_id      NUMBER
, territory      VARCHAR(2)
, status         VARCHAR2(1)
, rental_date    TIMESTAMP WITH LOCAL TIME ZONE
, constraint pk_car_rhist primary key(car_id,account_number,branch_id,rental_date)
)
PARTITION BY LIST (territory)
SUBPARTITION BY LIST (status)
( PARTITION p_nw VALUES ('WA', 'WY')
  ( SUBPARTITION snw_low   VALUES ('C')
  , SUBPARTITION snw_avg   VALUES ('B')
  , SUBPARTITION snw_high  VALUES ('A')
  )
,  PARTITION p_ne VALUES ('NY', 'CT')
  ( SUBPARTITION sne_low   VALUES ('C')
  , SUBPARTITION sne_avg   VALUES ('B')
  , SUBPARTITION sne_high  VALUES ('A')
  )
,  PARTITION p_sw VALUES ('CA', 'AZ')
  ( SUBPARTITION ssw_low   VALUES ('C')
  , SUBPARTITION ssw_avg   VALUES ('B')
  , SUBPARTITION ssw_high  VALUES ('A')
  )
,  PARTITION p_se VALUES ('FL', 'GA')
  ( SUBPARTITION sse_low   VALUES ('C')
  , SUBPARTITION sse_avg   VALUES ('B')
  , SUBPARTITION sse_high  VALUES ('A')
  )
  ) enable row movement
SQL> /

Table created.
```

RANGE-HASH PARTITIONED TABLE USING A SUBPARTITION TEMPLATE


```
CREATE TABLE      credential_evaluations
  ( eval_id  VARCHAR2(16) primary key
  , grad_id  VARCHAR2(12)
  , grad_date            DATE
  , degree_granted     VARCHAR2(12)
  , degree_major        VARCHAR2(64)
  , school_id             VARCHAR2(32)
  , final_gpa             NUMBER(4,2)
)
 PARTITION BY RANGE (grad_date)
   SUBPARTITION BY HASH (grad_id)
   SUBPARTITION TEMPLATE
   ( SUBPARTITION S_a TABLESPACE t1,
     SUBPARTITION S_b TABLESPACE t2,
     SUBPARTITION S_c TABLESPACE t3,
     SUBPARTITION S_d TABLESPACE t4
(  PARTITION grad_date_70s
   VALUES LESS THAN (
TO_DATE('01-JAN-1980','DD-MON-YYYY'))
 , PARTITION grad_date_80s
   VALUES LESS THAN (
TO_DATE('01-JAN-1990','DD-MON-YYYY'))
 , PARTITION grad_date_90s
   VALUES LESS THAN (
TO_DATE('01-JAN-2000','DD-MON-YYYY'))
 , PARTITION grad_date_00s
   VALUES LESS THAN (
TO_DATE('01-JAN-2010','DD-MON-YYYY'))
);
```

MULTICOLUMN RANGE-PARTITIONED TABLE

This example shows a multicolumn range-partitioned table, storing the actual DATE information in three separate columns: year, month, and day with partition quarterly granularity

```
CREATE TABLE  bi_auto_rentals_summary
(  acctno                NUMBER,
   rental_date           TIMESTAMP WITH LOCAL TIME ZONE,
   year                  NUMBER,
   month                 NUMBER,
   day                   NUMBER,
   total_amount          NUMBER,
   CONSTRAINT pk_actdate PRIMARY KEY (acctno, rental_date))
PARTITION BY RANGE (year,month)
  (PARTITION prior2008       VALUES LESS THAN (2008,1),
   PARTITION pq1_2008        VALUES LESS THAN (2008,4),
   PARTITION pq2_2008        VALUES LESS THAN (2008,7),
   PARTITION pq3_2008        VALUES LESS THAN (2008,10),
   PARTITION pq4_2008        VALUES LESS THAN (2009,1),
   PARTITION p_current       VALUES LESS THAN (MAXVALUE,1));
```

MULTICOLUMN RANGE-PARTITIONED TABLE

This sample code illustrates the use of a multicolumn partitioned approach for table supplier_parts, storing the relevant data including price. Partition the table on (supid, partno) to enforce equally sized partitions.

```
CREATE TABLE sp_price (
  supid          NUMBER,
  partno         NUMBER,
  unitprice      NUMBER,
  status         VARCHAR2(1))
PARTITION BY RANGE (supid, partno)
 (PARTITION p1 VALUES LESS THAN  (10000,1000),
  PARTITION p2 VALUES LESS THAN (50000,2000),
  PARTITION p3 VALUES LESS THAN (MAXVALUE,MAXVALUE));
```

USING VIRTUAL COLUMN-BASED PARTITIONING

- **In the context of partitioning, a virtual column can be used as any regular column.**
- **All partition methods are supported when using virtual columns, including interval partitioning and all different combinations of composite partitioning.**
- **There is no support for calls to a PL/SQL function on the virtual column used as the partitioning column.**

VIRTUAL COLUMN-BASED PARTITIONING


```
CREATE TABLE direct_marketing
 ( promo_id                 NUMBER(6) NOT NULL
 , cust_id          NUMBER NOT NULL
 , campaign_date            DATE NOT NULL
 , channel_code             CHAR(1) NOT NULL
 , campaign_id              NUMBER(6) NOT NULL
 , hist_avg_sales           NUMBER(12,2) NOT NULL
 , sales_forecast           NUMBER(12,2) NOT NULL
 , discrepancy      AS (sales_forecast - hist_avg_sales )  )
 PARTITION BY RANGE (campaign_date) INTERVAL (NUMTOYMINTERVAL(1,'MONTH'))
 SUBPARTITION BY RANGE(discrepancy)   SUBPARTITION TEMPLATE
  ( SUBPARTITION p_low     VALUES LESS THAN (5000)
  , SUBPARTITION p_avg     VALUES LESS THAN (15000)
  , SUBPARTITION p_high    VALUES LESS THAN (100000)
  , SUBPARTITION p_max     VALUES LESS THAN (MAXVALUE )  )
 (PARTITION p_campaign_prior_2009 VALUES LESS THAN  (TO_DATE('01-JAN-2009','dd-MON-yyyy'))
)
ENABLE ROW MOVEMENT COMPRESS PARALLEL NOLOGGING;
```

This sample code shows the sales table partitioned by range-range using a virtual column for the subpartitioning key. The virtual column calculates the difference between the historic average sales and the forecasted potential sales. As a rule, at least one partition must be specified.

USING COMPRESSION AND PARTITIONING

- For heap-organized partitioned tables, compress some or all partitions using table compression.
- The compression attribute can be declared for a tablespace, a table, or a partition of a table.
- Whenever the compress attribute is not specified, it is inherited like any other storage attribute.

USING COMPRESSION AND PARTITIONING


```
CREATE TABLE        credential_evaluations
  ( eval_id   VARCHAR2(16) primary key
  , grad_id   VARCHAR2(12)
  , grad_date            DATE
  , degree_granted       VARCHAR2(12)
  , degree_major         VARCHAR2(64)
  , school_id            VARCHAR2(32)
  , final_gpaNUMBER(4,2))
 PARTITION BY RANGE (grad_date)
   SUBPARTITION BY HASH (grad_id) SUBPARTITIONS 8 STORE IN (T1,T2,T3,T4)
(  PARTITION grad_e_70s
   VALUES LESS THAN (TO_DATE('01-JAN-1980','DD-MON-YYYY'))  TABLESPACE T1 COMPRESS
 , PARTITION grad_date_80s
   VALUES LESS THAN (TO_DATE('01-JAN-1990','DD-MON-YYYY'))  TABLESPACE T2 COMPRESS
 , PARTITION grad_date_90s
   VALUES LESS THAN (TO_DATE('01-JAN-2000','DD-MON-YYYY'))  TABLESPACE T3 NOCOMPRESS
 , PARTITION grad_date_00s
   VALUES LESS THAN (TO_DATE('01-JAN-2010','DD-MON-YYYY'))  TABLESPACE T4 NOCOMPRESS)
ENABLE ROW MOVEMENT;
```

This sample code creates a list-partitioned table with both compressed and uncompressed partitions. The compression attribute for the table and all other partitions is inherited from the tablespace level.

USING PARTITIONED INDEX KEY COMPRESSION

- Compress some or all partitions of a B-tree index using key compression.
- Key compression is applicable only to B-tree indexes.
- Bitmap indexes are stored in a compressed manner by default.
- An index using key compression eliminates repeated occurrences of key column prefix values, thus saving space and I/O.
- This sample code creates a local partitioned index with all partitions except the most recent one compressed:

```
CREATE INDEX ndx_grad_date ON credential_evaluations (grad_date)
COMPRESS LOCAL
  (
   PARTITION grad_date_70s,
   PARTITION grad_date_80s,
   PARTITION grad_date_90s,
   PARTITION grad_date_00s NOCOMPRESS
  );
```

It is NOT possible to specify COMPRESS (or NOCOMPRESS) explicitly for an index subpartition. The compression setting in a partition is inherited for a child subpartition. Attribute. Each index subpartition of a parent partition inherits its key compression setting .

CREATING RANGE-PARTITIONED INDEX-ORGANIZED TABLES

It is possible to partition index-organized tables, and their secondary indexes, by the range method. This sample code creates the range-partitioned index-organized table new_mktg_campaings. The INCLUDING clause specifies that all columns after period_code are to be stored in an overflow segment. There is one overflow segment for each partition, all stored in the same tablespace (T11). Optionally, OVERFLOW TABLESPACE is specified at the individual partition level, in which case some or all of the overflow segments could have separate TABLESPACE attributes.

```
CREATE TABLE new_mktg_campaigns
( campaign_id        NUMBER(8)
, period_code                      INTEGER CONSTRAINT rck CHECK (period_code BETWEEN
1 AND 26)
, campaign_name                    VARCHAR2(20)
, projected_sales                  NUMBER(12,2)
, campaign_desc                    VARCHAR2(4000),
  PRIMARY KEY (campaign_id, period_code)
)    ORGANIZATION INDEX
     INCLUDING period_code OVERFLOW TABLESPACE T11
     PARTITION BY RANGE (period_code)
          (PARTITION VALUES LESS THAN (10) TABLESPACE t1,
           PARTITION VALUES LESS THAN (20) TABLESPACE t2 OVERFLOW TABLESPACE
t9,
           PARTITION VALUES LESS THAN (MAXVALUE) TABLESPACE t13);
```

HASH-PARTITIONED INDEX-ORGANIZED TABLES

Another option for partitioning index-organized tables is to use the hash method. In the following example, the future_mktg_campaings index-organized table is partitioned by the hash method.

```
CREATE TABLE future_mktg_campaigns
( campaign_id       NUMBER(8)
, period_code       INTEGER
            CONSTRAINT fnock CHECK (period_code BETWEEN 1 AND 26)
, campaign_name  VARCHAR2(20)
, projected_sales   NUMBER(12,2)
, campaign_desc   VARCHAR2(2000),
  PRIMARY KEY (campaign_id,  period_code)
)
    ORGANIZATION INDEX
    INCLUDING period_code OVERFLOW TABLESPACE T11
    PARTITION BY HASH (period_code)
         PARTITIONS 8
         STORE IN (T1,T2,T3,T4,T5,T6,T7,T8)
         OVERFLOW STORE IN (T9,T10,T11);
```

LIST-PARTITIONED INDEX-ORGANIZED TABLES

The other option for partitioning index-organized tables is to use the list method.

```
CREATE TABLE current_mktg_campaigns
( campaign_id       NUMBER(8)
, period_code       INTEGER CONSTRAINT fpclst_ck
        CHECK (period_code BETWEEN 1 AND 26)
, campaign_name               VARCHAR2(20)
, projected_sales   NUMBER(12,2)
, campaign_desc     VARCHAR2(4000),
  PRIMARY KEY (campaign_id, period_code))
        ORGANIZATION INDEX
        INCLUDING period_code OVERFLOW TABLESPACE T11
        PARTITION BY LIST (period_code)
        (PARTITION  A VALUES (2, 4, 8, 10,12,14,16) TABLESPACE t12,
        PARTITION  B VALUES (1,3,5,7,9,11,13,15,17) TABLESPACE t14
        OVERFLOW TABLESPACE t15,
        PARTITION C VALUES (DEFAULT) TABLESPACE t10  );
```

COMPOSITE INTERVAL-* PARTITIONED TABLES


```
TABLE_OWNER                    TABLE_NAME                     COM PARTITION_NAME
------------------------------ ------------------------------ --- ------------------------------
SUBPARTITION_COUNT HIGH_VALUE
------------------ --------------------------------------------------------------------------------
HIGH_VALUE_LENGTH PARTITION_POSITION TABLESPACE_NAME                  PCT_FREE   PCT_USED  INI_TRANS
----------------- ------------------ ------------------------------ ---------- ---------- ----------
 MAX_TRANS INITIAL_EXTENT NEXT_EXTENT MIN_EXTENT MAX_EXTENT   MAX_SIZE PCT_INCREASE  FREELISTS
---------- -------------- ----------- ---------- ---------- ---------- ------------ ----------
FREELIST_GROUPS LOGGING COMPRESS COMPRESS_FOR         NUM_ROWS     BLOCKS EMPTY_BLOCKS  AVG_SPACE
--------------- ------- -------- ------------------ ---------- ---------- ------------ ----------
 CHAIN_CNT AVG_ROW_LEN SAMPLE_SIZE LAST_ANAL BUFFER_ GLO USE
---------- ----------- ----------- --------- ------- --- ---
ANTHONY                        PRO_MARKETING_CAMPAIGNS        YES BEFORE_2009
                 4 TO_DATE(' 2009-01-01 00:00:00', 'SYYYY-MM-DD HH24:MI:SS', 'NLS_CALENDAR=GREGORIA
               83                  1 USERS                                 0                      1
       255
                NONE    ENABLED  DIRECT LOAD ONLY
                                                            DEFAULT NO  NO

SQL> insert into pro_marketing_campaigns values (100,'ADN Campaign',sysdate-100,2,10000000,'Composite Interval-* sample')
  2  /

1 row created.

SQL> commit;

Commit complete.

SQL> select * from dba_tab_partitions where table_name ='PRO_MARKETING_CAMPAIGNS';

TABLE_OWNER                    TABLE_NAME                     COM PARTITION_NAME
------------------------------ ------------------------------ --- ------------------------------
SUBPARTITION_COUNT HIGH_VALUE
------------------ --------------------------------------------------------------------------------
HIGH_VALUE_LENGTH PARTITION_POSITION TABLESPACE_NAME                  PCT_FREE   PCT_USED  INI_TRANS
----------------- ------------------ ------------------------------ ---------- ---------- ----------
 MAX_TRANS INITIAL_EXTENT NEXT_EXTENT MIN_EXTENT MAX_EXTENT   MAX_SIZE PCT_INCREASE  FREELISTS
---------- -------------- ----------- ---------- ---------- ---------- ------------ ----------
FREELIST_GROUPS LOGGING COMPRESS COMPRESS_FOR         NUM_ROWS     BLOCKS EMPTY_BLOCKS  AVG_SPACE
--------------- ------- -------- ------------------ ---------- ---------- ------------ ----------
 CHAIN_CNT AVG_ROW_LEN SAMPLE_SIZE LAST_ANAL BUFFER_ GLO USE
---------- ----------- ----------- --------- ------- --- ---
ANTHONY                        PRO_MARKETING_CAMPAIGNS        YES BEFORE_2009
                 4 TO_DATE(' 2009-01-01 00:00:00', 'SYYYY-MM-DD HH24:MI:SS', 'NLS_CALENDAR=GREGORIA
               83                  1 USERS                                 0                      1
       255
                NONE    ENABLED  DIRECT LOAD ONLY
                                                            DEFAULT NO  NO

ANTHONY                        PRO_MARKETING_CAMPAIGNS        YES SYS_P289
                 4 TO_DATE(' 2009-03-01 00:00:00', 'SYYYY-MM-DD HH24:MI:SS', 'NLS_CALENDAR=GREGORIA
               83                  2 USERS                                 0                      1
```

COMPOSITE INTERVAL-* PARTITIONED TABLES

- Include the INTERVAL definition.
- Specify at least one range partition using the PARTITION clause.
- Note that:
  - The range partitioning key value determines the high value of the range partitions, which is called the transition point, and the database automatically creates interval partitions for data beyond that transition point.
  - The subpartitions for intervals in an interval-* partitioned table will be created when the database creates the interval. You can specify the definition of future subpartitions only through the use of a subpartition template.
- Create an interval-hash partitioned table with multiple hash partitions using one of the following methods:
- Either specify a number of hash partitions in the PARTITIONS clause or Use a subpartition template: Future interval partitions will only get a single hash subpartition.

COMPOSITE INTERVAL-* PARTITIONED TABLES

This sample code shows the pro_marketing_campaigns table as interval-partitioned using monthly intervals on campaign_date, with hash subpartitions by period_code.

```
CREATE TABLE pro_marketing_campaigns
( campaign_id         NUMBER(8)
, campaign_name     VARCHAR2(20)
, campaign_date       DATE
, period_code       INTEGER CONSTRAINT fcopck CHECK (period_code BETWEEN 1 AND 26)
, projected_sales     NUMBER(12,2)
, campaign_desc      VARCHAR2(4000),
  PRIMARY KEY (campaign_id, period_code))
 PARTITION BY RANGE (campaign_date) INTERVAL (NUMTOYMINTERVAL(1,'MONTH'))
 SUBPARTITION BY HASH (period_code) SUBPARTITIONS 4
 ( PARTITION p_prior_2009 VALUES LESS THAN (TO_DATE('01-JAN-2009','dd-mon-yyyy')))
PARALLEL COMPRESS FOR ALL OPERATIONS;
```

PARTITIONING AND ENCRYPTION

- There is no encryption support for a column used a partitioning key

```
SQL>  CREATE TABLE direct_marketing2
  2      ( promo_id                NUMBER(6) NOT NULL
  3      , cust_id                 NUMBER NOT NULL
  4      , campaign_date           DATE ENCRYPT USING 'AES192' SALT
  5      , channel_code            CHAR(1) NOT NULL
  6      , campaign_id             NUMBER(6) NOT NULL
  7      , hist_avg_sales          NUMBER(12,2) NOT NULL
  8      , sales_forecast          NUMBER(12,2)
  9      , discrepancy       AS (sales_forecast - hist_avg_sales )  )
 10     PARTITION BY RANGE (campaign_date) INTERVAL (NUMTOYMINTERVAL(1,'MONTH'))
 11     SUBPARTITION BY RANGE(hist_avg_sales)   SUBPARTITION TEMPLATE
 12       ( SUBPARTITION p_low      VALUES LESS THAN (5000)
 13       , SUBPARTITION p_avg      VALUES LESS THAN (15000)
 14       , SUBPARTITION p_high     VALUES LESS THAN (100000)
 15       , SUBPARTITION p_max      VALUES LESS THAN (MAXVALUE )  )
 16     (PARTITION p_campaign_prior_2009 VALUES LESS THAN  (TO_DATE('01-JAN-2009','dd-MON-yyyy')) )
 17    ENABLE ROW MOVEMENT COMPRESS PARALLEL NOLOGGING
 18   ;
 CREATE TABLE direct_marketing2
 *
ERROR at line 1:
ORA-28346: an encrypted column cannot serve as a partitioning column


SQL>
```

OMBSDB: Multiple Block Size Caches


```
NAME
--------------------------------------------------------------------------------
C:\APP\ORACLE\ORADATA\ADN3\SYSTEM01.DBF
C:\APP\ORACLE\ORADATA\ADN3\SYSAUX01.DBF
C:\APP\ORACLE\ORADATA\ADN3\UNDOTBS01.DBF
C:\APP\ORACLE\ORADATA\ADN3\USERS01.DBF
C:\APP\ORACLE\ORADATA\ADN3\T1.DBF
C:\APP\ORACLE\ORADATA\ADN3\T2.DBF
C:\APP\ORACLE\ORADATA\ADN3\T3.DBF
C:\APP\ORACLE\ORADATA\ADN3\T4.DBF
C:\APP\ORACLE\ORADATA\ADN3\T5.DBF
C:\APP\ORACLE\ORADATA\ADN3\T6.DBF
C:\APP\ORACLE\ORADATA\ADN3\T7.DBF
C:\APP\ORACLE\ORADATA\ADN3\T8.DBF
C:\APP\ORACLE\ORADATA\ADN3\T9.DBF
C:\APP\ORACLE\ORADATA\ADN3\T10.DBF
C:\APP\ORACLE\ORADATA\ADN3\T11.DBF
C:\APP\ORACLE\ORADATA\ADN3\T12.DBF
C:\APP\ORACLE\ORADATA\ADN3\T13.DBF
C:\APP\ORACLE\ORADATA\ADN3\T14.DBF
C:\APP\ORACLE\ORADATA\ADN3\T15.DBF
C:\APP\ORACLE\ORADATA\ADN3\T16.DBF

20 rows selected.

SQL> alter system set db_16k_cache_size = 16m;

System altered.

SQL> show parameter db_

NAME                                 TYPE        VALUE
------------------------------------ ----------- ------------------------------
db_16k_cache_size                    big integer 16M
db_2k_cache_size                     big integer 0
db_32k_cache_size                    big integer 0
db_4k_cache_size                     big integer 0
db_8k_cache_size                     big integer 0
db_block_buffers                     integer     0
db_block_checking                    string      FALSE
db_block_checksum                    string      TYPICAL
db_block_size                        integer     8192
db_cache_advice                      string      ON
```

USING MUTIPLE BLOCK SIZE CACHES

- Creating indexes on a tablespace with a larger block size will increasing performance in DSS and in most OLTP scenarios.
- This sample code creates the credential_tables in the 8k block size T1,T2,T3,and T4 tablespaces, and local indexes on the 16k T18,T20,T22,T24 tablespaces, as cached respectively.

```
SQL> CREATE TABLE        credential_evaluations
       ( eval_id         VARCHAR2(16) primary key
       , grad_id         VARCHAR2(12)
       , grad_date       DATE
       , degree_granted  VARCHAR2(12)
       , degree_major    VARCHAR2(64)
       , school_id       VARCHAR2(32)
       , final_gpa       NUMBER(4,2)
     )
      PARTITION BY RANGE (grad_date)
     (  PARTITION grad_date_70s
        VALUES LESS THAN (TO_DATE('01-JAN-1980','DD-MON-YYYY'))  TABLESPACE T1
      , PARTITION grad_date_80s
        VALUES LESS THAN (TO_DATE('01-JAN-1990','DD-MON-YYYY'))  TABLESPACE T2
      , PARTITION grad_date_90s
        VALUES LESS THAN (TO_DATE('01-JAN-2000','DD-MON-YYYY'))  TABLESPACE T3
      , PARTITION grad_date_00s
        VALUES LESS THAN (TO_DATE('01-JAN-2010','DD-MON-YYYY'))  TABLESPACE T4 )
     ENABLE ROW MOVEMENT;

Table created.

SQL> CREATE INDEX ndx_cr_gd ON credential_evaluations (grad_date)  LOCAL
     (  PARTITION grad_date_70s TABLESPACE t18
      , PARTITION grad_date_80s TABLESPACE T20
      , PARTITION grad_date_90s TABLESPACE T22
      , PARTITION grad_date_00s TABLESPACE T24);

Index created.

SQL> _
```

USING MUTIPLE BLOCK SIZE CACHES


```
SQL> SET AUTOTRACE ON
SQL> SELECT * FROM CREDENTIAL_EVALUATIONS WHERE grad_date > sysdate-150;

no rows selected

Execution Plan
----------------------------------------------------------
Plan hash value: 3993119364

--------------------------------------------------------------------------------
-----------------------------------

| Id  | Operation                | Name                    | Rows  | Bytes | Cost
 (%CPU)| Time     | Pstart| Pstop |

--------------------------------------------------------------------------------
-----------------------------------

|   0 | SELECT STATEMENT         |                         |     1 |   100 |
2   (0)| 00:00:01 |       |       |

|   1 |  PARTITION RANGE ITERATOR|                         |     1 |   100 |
2   (0)| 00:00:01 |   KEY |     4 |

|*  2 |   TABLE ACCESS FULL      | CREDENTIAL_EVALUATIONS |     1 |   100 |
2   (0)| 00:00:01 |   KEY |     4 |

--------------------------------------------------------------------------------
-----------------------------------


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

   2 - filter("GRAD_DATE">SYSDATE@!-150)

Note
-----
   - dynamic sampling used for this statement


Statistics
----------------------------------------------------------
          0  recursive calls
          0  db block gets
          3  consistent gets
          0  physical reads
          0  redo size
        739  bytes sent via SQL*Net to client
        513  bytes received via SQL*Net from client
          1  SQL*Net roundtrips to/from client
          0  sorts (memory)
```

Analyzing Performance from Optimizer access path as explained.

RELEVANT DATA DICTIONARY VIEWS

Dictionary Views with Partitioned Tables and Indexes Information

The following views display information specific to partitioned tables and indexes:

View	Description
DBA_PART_TABLES ALL_PART_TABLES USER_PART_TABLES	DBA view lists partitioning information for all partitioned tables in the database. ALL view displays partitioning information for all partitioned tables accessible to the user. USER view is restricted to partitioning information for partitioned tables owned by the user.
DBA_TAB_PARTITIONS ALL_TAB_PARTITIONS USER_TAB_PARTITIONS	Exhibits partition-level partitioning information, partition storage parameters, and partition statistics generated by the DBMS_STATS package or the ANALYZE statement.
DBA_TAB_SUBPARTITIONS ALL_TAB_SUBPARTITIONS USER_TAB_SUBPARTITIONS	Display subpartition-level partitioning information, subpartition storage parameters, and subpartition statistics generated by the DBMS_STATS package or the ANALYZE statement.
DBA_PART_KEY_COLUMNS ALL_PART_KEY_COLUMNS USER_PART_KEY_COLUMNS	Display the partitioning key columns for partitioned tables.
DBA_SUBPART_KEY_COLUMNS ALL_SUBPART_KEY_COLUMNS USER_SUBPART_KEY_COLUMNS	Display the subpartitioning key columns for composite-partitioned tables (and local indexes on composite-partitioned tables).
DBA_PART_COL_STATISTICS ALL_PART_COL_STATISTICS USER_PART_COL_STATISTICS	Display column statistics and histogram information for the partitions of tables.
DBA_SUBPART_COL_STATISTICS ALL_SUBPART_COL_STATISTICS USER_SUBPART_COL_STATISTICS	Display column statistics and histogram information for subpartitions of tables.

PARTITIONED TABLE MAINTENANCE

ALTER TABLE Maintenance Operations for Table Subpartitions

Maintenance Operation	Composite *-Range	Composite *-Hash	Composite *-List
Adding Partitions	MODIFY PARTITION ... ADD SUBPARTITION	MODIFY PARTITION ... ADD SUBPARTITION	MODIFY PARTITION ... ADD SUBPARTITION
Coalescing Partitions	N/A	MODIFY PARTITION ... COALESCE SUBPARTITION	N/A
Dropping Partitions	DROP SUBPARTITION	N/A	DROP SUBPARTITION
Exchanging Partitions	EXCHANGE SUBPARTITION	N/A	EXCHANGE SUBPARTITION
Merging Partitions	MERGE SUBPARTITIONS	N/A	MERGE SUBPARTITIONS
Modifying Default Attributes	MODIFY DEFAULT ATTRIBUTES FOR PARTITION	MODIFY DEFAULT ATTRIBUTES FOR PARTITION	MODIFY DEFAULT ATTRIBUTES FOR PARTITION
Modifying Real Attributes of Partitions	MODIFY SUBPARTITION	MODIFY SUBPARTITION	MODIFY SUBPARTITION
Modifying List Partitions: Adding Values	N/A	N/A	MODIFY SUBPARTITION ... ADD VALUES
Modifying List Partitions: Dropping Values	N/A	N/A	MODIFY SUBPARTITION ... DROP VALUES
Modifying a Subpartition Template	SET SUBPARTITION TEMPLATE	SET SUBPARTITION TEMPLATE	SET SUBPARTITION TEMPLATE
Moving Partitions	MOVE SUBPARTITION	MOVE SUBPARTITION	MOVE SUBPARTITION
Renaming Partitions	RENAME SUBPARTITION	RENAME SUBPARTITION	RENAME SUBPARTITION
Splitting Partitions	SPLIT SUBPARTITION	N/A	SPLIT SUBPARTITION
Truncating Partitions	TRUNCATE SUBPARTITION	TRUNCATE SUBPARTITION	TRUNCATE SUBPARTITION

PARTITIONED INDEX MAINTENANCE

ALTER INDEX Maintenance Operations for Index Partitions

Maintenance Operation	Type of Index	Type of Index Partitioning		
		Range	Hash and List	Composite
Adding Index Partitions	Global	-	ADD PARTITION (hash only)	-
	Local	N/A	N/A	N/A
Dropping Index Partitions	Global	DROP PARTITION	-	-
	Local	N/A	N/A	N/A
Modifying Default Attributes of Index Partitions	Global	MODIFY DEFAULT ATTRIBUTES	-	-
	Local	MODIFY DEFAULT ATTRIBUTES	MODIFY DEFAULT ATTRIBUTES	MODIFY DEFAULT ATTRIBUTES MODIFY DEFAULT ATTRIBUTES FOR PARTITION
Modifying Real Attributes of Index Partitions	Global	MODIFY PARTITION	-	-
	Local	MODIFY PARTITION	MODIFY PARTITION	MODIFY PARTITION MODIFY SUBPARTITION
Rebuilding Index Partitions	Global	REBUILD PARTITION	-	-
	Local	REBUILD PARTITION	REBUILD PARTITION	REBUILD SUBPARTITION
Renaming Index Partitions	Global	RENAME PARTITION	-	-
	Local	RENAME PARTITION	RENAME PARTITION	RENAME PARTITION RENAME SUBPARTITION
Splitting Index Partitions	Global	SPLIT PARTITION	-	-
	Local	N/A	N/A	N/A

MAINTENANCE OPERATIONS

The following operations support the UPDATE INDEXES clause:

ADD PARTITION | SUBPARTITION

COALESCE PARTITION | SUBPARTITION

DROP PARTITION | SUBPARTITION

EXCHANGE PARTITION | SUBPARTITION

MERGE PARTITION | SUBPARTITION

MOVE PARTITION | SUBPARTITION

SPLIT PARTITION | SUBPARTITION

TRUNCATE PARTITION | SUBPARTITION

SKIP_UNUSABLE_INDEXES Initialization Parameter

As of Oracle10g, SKIP_UNUSABLE_INDEXES is an initialization parameter with a default value of TRUE. This setting disables error reporting of indexes and index partitions marked UNUSABLE. To avoid choosing an alternative execution plan to evading the unusable elements, set this parameter to FALSE.

MAINTENANCE OPERATIONS

ALTER TABLE credential_evaluatons **ADD** PARTITION grad_date_10s
VALUES LESS THAN (TO_DATE('01-JAN-2020','DD-MON-YYYY')) **TABLESPACE** T10;

```
SQL>
SQL> ALTER TABLE credential_evaluations ADD PARTITION grad_date_10s VALUES LESS THAN (TO_DATE('01-JAN-2020','DD-MON-YYYY')) TABLESPACE T10;

Table altered.
```

```
SQL> ALTER TABLE xmliot ADD PARTITION xmliot_p5 TABLESPACE T5;
ALTER TABLE xmliot ADD PARTITION xmliot_p5 TABLESPACE T5
            *
ERROR at line 1:
ORA-25182: feature not currently available for index-organized tables
```

```
SQL>
SQL> ALTER TABLE rental_costs ADD PARTITION pce TABLESPACE T5;
ALTER TABLE rental_costs ADD PARTITION pce TABLESPACE T5
            *
ERROR at line 1:
ORA-14760: ADD PARTITION is not permitted on Interval partitioned objects
```

MAINTENANCE OPERATIONS

ALTER TABLE school_directory COALESCE PARTITION PARALLEL;

```
SQL> ALTER TABLE regional_rentals COALESCE PARTITION PARALLEL;
ALTER TABLE regional_rentals COALESCE PARTITION PARALLEL
            *
ERROR at line 1:
ORA-14259: table is not partitioned by Hash method
```

```
SQL> ALTER TABLE xmliot COALESCE PARTITION PARALLEL;
ALTER TABLE xmliot COALESCE PARTITION PARALLEL
            *
ERROR at line 1:
ORA-25182: feature not currently available for index-organized tables


SQL> ALTER TABLE X_adnxml_tab COALESCE PARTITION PARALLEL;

Table altered.

SQL>
SQL> ALTER TABLE school_directory COALESCE PARTITION PARALLEL;

Table altered.

SQL>
```

MAINTENANCE OPERATIONS

ALTER TABLE order_hist DROP PARTITION pq4 UPDATE INDEXES;

```
SQL>
SQL> ALTER TABLE order_hist DROP PARTITION pq4 UPDATE INDEXES
  2  /

Table altered.

SQL>
```

MAINTENANCE OPERATIONS

ALTER TABLE bi_auto_rentals_summary EXCHANGE PARTITION pq1_2008 WITH TABLE bi_auto_rentals_summary _1 UPDATE INDEXES;

```
SQL>
SQL> ALTER TABLE BI_AUTO_RENTALS_SUMMARY TRUNCATE PARTITION pq1_2008 DROP STORAGE;

Table truncated.

SQL>
```

```
SQL> ALTER TABLE BI_AUTO_RENTALS_SUMMARY EXCHANGE PARTITION pq1_2008 WITH TABLE BI_AUTO_RENTALS_SUMMARY_1;

Table altered.

SQL>
SQL>
```

```
SQL>
SQL> ALTER TABLE BI_AUTO_RENTALS_SUMMARY EXCHANGE PARTITION pq1_2008 WITH TABLE BI_AUTO_RENTALS_SUMMARY_1 UPDATE INDEXES;

Table altered.

SQL>
```

MAINTENANCE OPERATIONS

ALTER TABLE bi_auto_rentals_summary **MOVE PARTITION** pq1_2008 **TABLESPACE** T9 **UPDATE INDEXES;**

```
SQL>
SQL> ALTER TABLE credential_evaluations MOVE PARTITION grad_date_90s TABLESPACE T9 UPDATE INDEXES;
ALTER TABLE credential_evaluations MOVE PARTITION grad_date_90s TABLESPACE T9 UPDATE INDEXES
                                                  *
ERROR at line 1:
ORA-14257: cannot move partition other than a Range, List, System, or Hash
partition
```

```
SQL> ALTER TABLE bi_auto_rentals_summary MOVE PARTITION pq1_2008 TABLESPACE T9 UPDATE INDEXES;

Table altered.

SQL>
SQL>
```

MAINTENANCE OPERATIONS


```
SQL> ALTER TABLE rental_costS  merge partitionS pca, pcb;

Table altered.

SQL>
SQL>
```

```
SQL> SELECT table_name,partition_name,tabespace_name
  2
SQL> SELECT table_name,partition_name,tablespace_name
  2  FROM user_tab_partitions
  3  WHERE table_name ='RENTAL_COSTS'
  4  ORDER BY 1,2,3
  5  /

TABLE_NAME                     PARTITION_NAME                 TABLESPACE
------------------------------ ------------------------------ ----------
RENTAL_COSTS                   PCC                            T8
RENTAL_COSTS                   PCD                            T12
RENTAL_COSTS                   SYS_P423                       USERS
```

MAINTENANCE OPERATIONS


```
SQL> ALTER TABLE REGIONAL_RENTALS MERGE PARTITIONS PSW,PSE INTO PARTITION PSS;

Table altered.
```

```
SQL> SELECT table_name,partition_name
  2  FROM user_tab_partitions
  3  WHERE table_name = 'REGIONAL_RENTALS'
  4  ORDER BY 1,2
  5  /

TABLE_NAME                                PARTITION_NAME
----------------------------------------- ------------------------------
REGIONAL_RENTALS                          PNE
REGIONAL_RENTALS                          PNW
REGIONAL_RENTALS                          PSS

SQL> _
```

MAINTENANCE OPERATIONS

ALTER TABLE q_territory_sales **SPLIT PARTITION** q4_2000 **AT** (TO_DATE('15-MOV-2000','DD-MON-YYYY'));

```
SQL> ALTER TABLE q_territory_sales SPLIT PARTITION q4_2000  AT (TO_DATE('15-NOV-2000','DD-MON-YYYY'))
  2  /

Table altered.

SQL>
```

```
TABLE_NAME                     PARTITION_NAME                 SUBPARTITION_NAME
------------------------------ ------------------------------ ------------------------------
Q_TERRITORY_SALES              Q1_2000                        Q1_2000_NC
Q_TERRITORY_SALES              Q1_2000                        Q1_2000_NE
Q_TERRITORY_SALES              Q1_2000                        Q1_2000_NW
Q_TERRITORY_SALES              Q1_2000                        Q1_2000_SC
Q_TERRITORY_SALES              Q1_2000                        Q1_2000_SE
Q_TERRITORY_SALES              Q1_2000                        Q1_2000_SW
Q_TERRITORY_SALES              Q2_2000                        Q2_2000_NC
Q_TERRITORY_SALES              Q2_2000                        Q2_2000_NE
Q_TERRITORY_SALES              Q2_2000                        Q2_2000_NW
Q_TERRITORY_SALES              Q2_2000                        Q2_2000_SC
Q_TERRITORY_SALES              Q2_2000                        Q2_2000_SE
Q_TERRITORY_SALES              Q2_2000                        Q2_2000_SW
Q_TERRITORY_SALES              Q3_2000                        Q3_2000_NC
Q_TERRITORY_SALES              Q3_2000                        Q3_2000_NE
Q_TERRITORY_SALES              Q3_2000                        Q3_2000_NW
Q_TERRITORY_SALES              Q3_2000                        Q3_2000_SC
Q_TERRITORY_SALES              Q3_2000                        Q3_2000_SE
Q_TERRITORY_SALES              Q3_2000                        Q3_2000_SW
Q_TERRITORY_SALES              SYS_P467                       SYS_SUBP455
Q_TERRITORY_SALES              SYS_P467                       SYS_SUBP456
Q_TERRITORY_SALES              SYS_P467                       SYS_SUBP457
Q_TERRITORY_SALES              SYS_P467                       SYS_SUBP458
Q_TERRITORY_SALES              SYS_P467                       SYS_SUBP459
Q_TERRITORY_SALES              SYS_P467                       SYS_SUBP460
Q_TERRITORY_SALES              SYS_P468                       SYS_SUBP461
Q_TERRITORY_SALES              SYS_P468                       SYS_SUBP462
Q_TERRITORY_SALES              SYS_P468                       SYS_SUBP463
Q_TERRITORY_SALES              SYS_P468                       SYS_SUBP464
Q_TERRITORY_SALES              SYS_P468                       SYS_SUBP465
Q_TERRITORY_SALES              SYS_P468                       SYS_SUBP466

30 rows selected.
```

MAINTENANCE OPERATIONS

ALTER TABLE credential_evaluations **TRUNCATE PARTITION** grad_date_10s **DROP STORAGE UPDATE INDEXES;**

```
SQL> ALTER TABLE DIRECT_MARKETING  TRUNCATE SUBPARTITION P_CAMPAIGN_PRIOR_2009_P_AVG;

Table truncated.
```

```
  1  SELECT table_name,partition_name,composite
  2  from user_tab_Partitions
  3  WHERE table_name='CREDENTIAL_EVALUATIONS'
  4* ORDER BY 1,2,3
SQL> /

TABLE_NAME                     PARTITION_NAME                 COM
------------------------------ ------------------------------ ---
CREDENTIAL_EVALUATIONS         GRAD_DATE_10S                  YES
CREDENTIAL_EVALUATIONS         GRAD_DATE_70S                  YES
CREDENTIAL_EVALUATIONS         GRAD_DATE_80S                  YES
CREDENTIAL_EVALUATIONS         GRAD_DATE_90S                  YES
CREDENTIAL_EVALUATIONS         SYS_P421                       YES
CREDENTIAL_EVALUATIONS         SYS_P422                       YES

6 rows selected.
```

```
SQL> ALTER TABLE credential_evaluations TRUNCATE PARTITION grad_date_10s UPDATE INDEXES;

Table truncated.

SQL> ALTER TABLE credential_evaluations TRUNCATE PARTITION grad_date_10s DROP STORAGE UPDATE INDEXES;

Table truncated.

SQL>
```

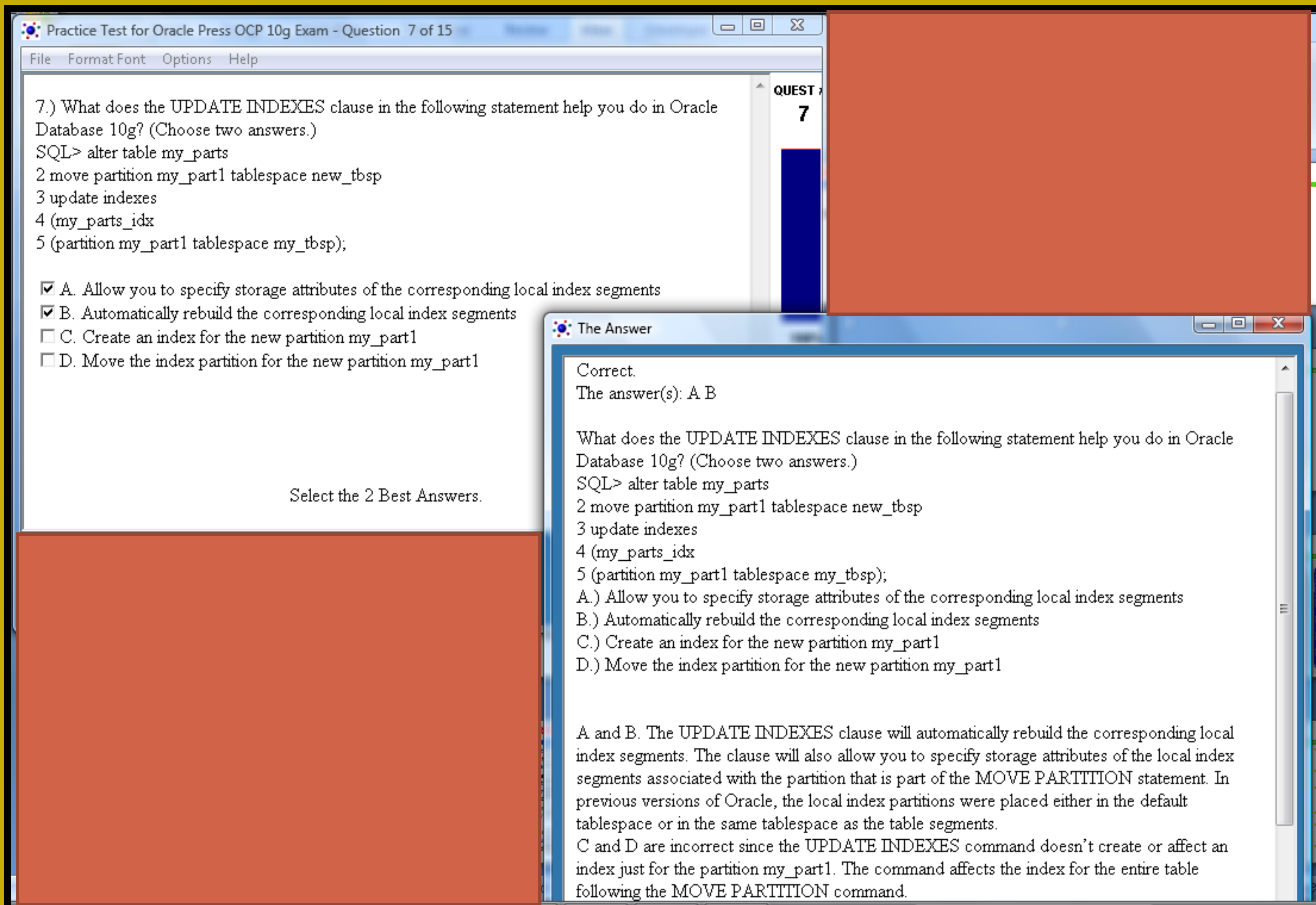
MAINTENANCE OPERATIONS

ALTER INDEX ndx_final_gpa REBUILD PARTITION C1;
ALTER INDEX ndx_final_gpa REBUILD PARTITION C2;
ALTER INDEX ndx_final_gpa REBUILD PARTITION B1;
ALTER INDEX ndx_final_gpa REBUILD PARTITION B2;
ALTER INDEX ndx_final_gpa REBUILD PARTITION A1;
ALTER INDEX ndx_final_gpa REBUILD PARTITION A2;

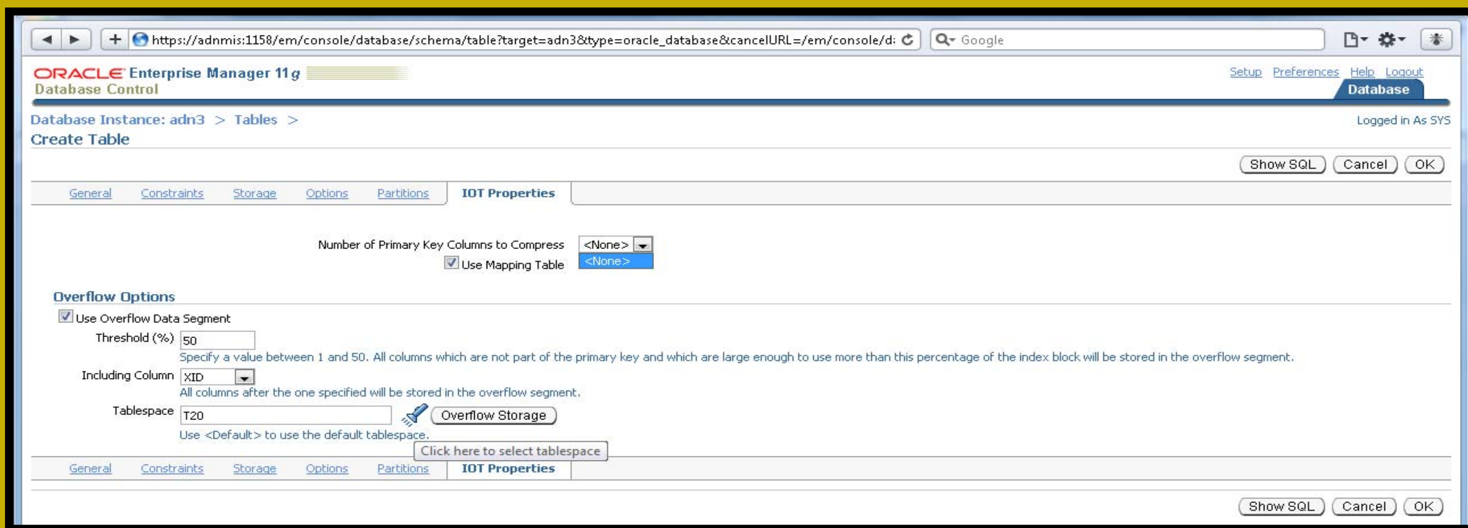
```
SQL> ALTER INDEX ndx_final_gpa REBUILD PARTITION C1;
Index altered.
SQL> ALTER INDEX ndx_final_gpa REBUILD PARTITION C2;
Index altered.
SQL> ALTER INDEX ndx_final_gpa REBUILD PARTITION B1;
Index altered.
SQL> ALTER INDEX ndx_final_gpa REBUILD PARTITION B2;
Index altered.
SQL> ALTER INDEX ndx_final_gpa REBUILD PARTITION A1;
Index altered.
SQL> ALTER INDEX ndx_final_gpa REBUILD PARTITION A2;
Index altered.
SQL>
```

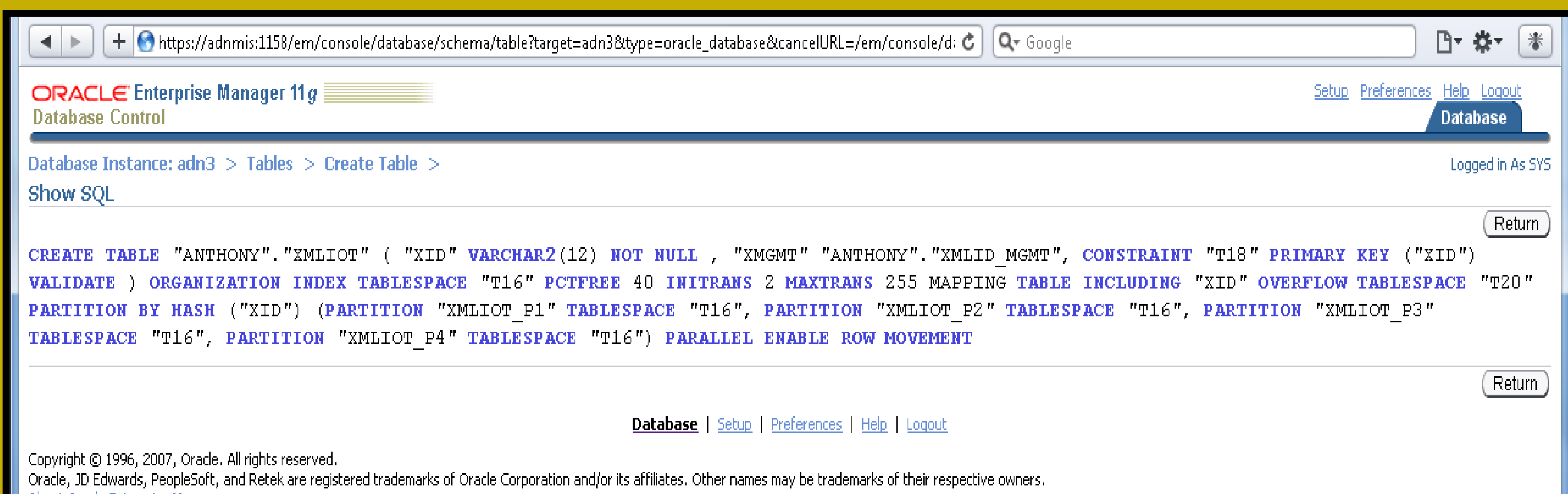
MAINTENANCE OPERATIONS

This is a Sample questions from the OCP DBA certification test preparation.



MANAGEABILITY





PARTITIONING FOR AVAILABILITY, MANAGEABILITY, AND PERFORMANCE

- Partition Pruning
- Partition-Wise Joins
- Index Partitioning
- Partitioning and Table Compression
- Tuning and Mixing objects in Multiple Block Size Database Models

PARTITION PRUNING

- Partition pruning is a foundational performance feature to both DSS and OLTP, enabling the Oracle Database to perform operations only on those partitions that are relevant to the SQL.
- The optimizer analyzes FROM and WHERE clauses in SQL statements to eliminate unneeded partitions.
- Partition pruning greatly optimizes time and resources when retrieving data from disk, thus improving query performance.
- When partitioning an index and a table on different columns (with a global partitioned index), then partition pruning also eliminates index partitions even when the partitions of the underlying table cannot be eliminated.
- Either static or dynamic pruning could be used, depending on SQL statement.
- Static pruning occurs at compile-time, with the information about the partitions accessed beforehand while dynamic pruning occurs at run-time.

PARTITION PRUNING

- Partition pruning affects the statistics of the objects involved and therefore also the execution plan of the statement.
- Oracle Database prunes partitions when using range, LIKE, equality, and IN-list predicates on the range or list partitioning columns, and when using equality and IN-list predicates on the hash partitioning columns.
- When using composite partitioned objects, Oracle can prune at both levels using the relevant predicates.

PARTITION-WISE JOINS

- Partition-wise joins minimize query response time by reducing the amount of data exchanged among parallel execution servers when joins execute in parallel, thus reducing response time and improving the use of both CPU and memory resources.
- In Oracle Real Application Clusters (RAC) environments, partition-wise joins also avoid or at least limit the data traffic over the interconnect, which is the key to achieving good scalability for massive join operations.

PARTITION-WISE JOINS

- Partition-wise joins can be full or partial. Oracle decides which one to use.
- **Full Partition-Wise Joins**
- A full partition-wise join divides a large join into smaller joins between a pair of partitions from the two joined tables. To use this feature, you must equipartition both tables on their join keys, or use reference partitioning. For example, consider a large join between a sales table and a customer table on cust_id.
- The query "find the records of all customers who were part of the campaign more than 200 potential sales items in the 3rd Quarter of 2008" is a typical example of a SQL statement performing such a join. This example shows a partition-wise join:

```
SELECT c.cust_lname COUNT(*)
FROM direct_marketing  dm, customers c
WHERE dm.cust_id = c.cust_id
   AND   dm.campaign_init_date = c.campaign_date
  AND dm.campaign_date  BETWEEN TO_DATE('01-JUL-2008', 'DD-MON-YYYY')  AND
     (TO_DATE('01-OCT-2008', 'DD-MON-YYYY'))
GROUP BY c.cust_lname HAVING COUNT(*) > 200;
```

PARTITION-WISE JOINS

- Partition-wise joins reduce query response time and optimizing CPU and memory resources by minimizing the amount of data exchanged among parallel execution servers when joins execute in parallel.
- In RAC environments, partition-wise joins also avoid or at least limit the data traffic over the interconnect, which is the key to achieving good scalability for massive joins.
- To avoid remote I/O, both matching partitions should have affinity to the same node.
- Partition pairs should be spread over all nodes to use all CPU resources available and avoid bottlenecks .
- Nodes can host multiple pairs when there are more pairs than nodes, e.g., for an 8-node system and 16 partition pairs, each node receives two pairs.

PARTITION-WISE JOINS

- **Full Partition-Wise Joins: Composite - Single-Level**
- This method is a variation of the single-level - single-level method. In this scenario, one table (typically the larger table) is composite partitioned on two dimensions, using the join columns as the subpartition key.
- **Partial Partition-Wise Joins: Single-Level Partitioning**
- The simplest method to enable a partial partition-wise join is to partition sales by hash on cust_id.
- The number of partitions determines the maximum degree of parallelism, because the partition is the smallest granule of parallelism for partial partition-wise join operations.

PARTITION-WISE JOINS

- **Full Partition-Wise Joins: Composite - Composite**
- When necessary, it is possible to also partition a table by a composite method.
- It is possible to get full partition-wise joins on all combinations of partition and subpartition partitions: partition - partition, partition - subpartition, subpartition - partition, and subpartition - subpartition.

PARTITION-WISE JOINS

- **Partial Partition-Wise Joins**
- Oracle Database can perform partial partition-wise joins only in parallel.
- Unlike full partition-wise joins, partial partition-wise joins require partitioning only one table on the join key.
- The partitioned table is referred to as the reference table. The other table may or may not be partitioned. Partial partition-wise joins are more common than full partition-wise joins.
- To execute a partial partition-wise join, the database dynamically repartitions the other table based on the partitioning of the reference table. Then, the execution becomes similar to a full partition-wise join.

PARTITION-WISE JOINS

- The performance advantage that partial partition-wise joins have over joins in non-partitioned tables is that the reference table is not moved during the join operation.
- The parallel joins between non-partitioned tables require both input tables to be redistributed on the join key. This redistribution operation involves exchanging rows between parallel execution servers.
- This is a CPU-intensive operation that can lead to excessive interconnect traffic in RAC environments.

RULES TO INDEX PARTITIONING

- The rules for partitioning indexes are similar to those for tables:
- An index can be partitioned unless:
  - The index is a cluster index.
  - The index is defined on a clustered table.
- It is possible to mix partitioned and nonpartitioned indexes with partitioned and nonpartitioned tables:
- A partitioned table can have partitioned or nonpartitioned indexes.
- A nonpartitioned table can have partitioned or nonpartitioned indexes.
- Bitmap indexes on nonpartitioned tables cannot be partitioned.
- A bitmap index on a partitioned table must be a local index.
- Nonprefixed indexes are particularly useful in historical databases.

RULES TO LOCAL INDEX PARTITIONING

- The three Oracle-supported Local Index partitioning types are:
- **Local Partitioned Indexes**
- In a local index, all keys in a particular index partition refer only to rows stored in a single underlying table partition. A local index is created by specifying the LOCAL attribute.
- Oracle constructs the local index so that it is equi-partitioned with the underlying table.
- Oracle also maintains the index partitioning automatically when partitions in the underlying table are added, dropped, merged, or split, or when hash partitions or subpartitions are added or coalesced, ensuring that the index remains equipartitioned with the table.
- A local index can be created UNIQUE if the partitioning columns form a subset of the index columns. This restriction guarantees that rows with identical index keys always map into the same partition, where uniqueness violations can be detected.

LOCAL INDEXES ADVANTAGES

- Only one index partition needs to be rebuilt when a maintenance operation other than SPLIT PARTITION or ADD PARTITION is performed on an underlying table partition.
- The duration of a partition maintenance operation is proportional to partition size.
- Local indexes support partition independence.
- Local indexes support smooth roll-out of old data and roll-in of new data in historical tables.
- Oracle can take advantage of the fact that a local index is equi-partitioned with the underlying table to generate improved query access plans.
- Local indexes simplify the task of tablespace incomplete recovery. In order to recover a partition or subpartition of a table to a point in time, the corresponding index entries must be recovered to the same point in time.
- Oracle Database PL/SQL Packages and Types Reference for a description of the DBMS_PCLXUTIL package

INDEX PARTITIONING TYPES

- **Local Prefixed Indexes**
- A local index is prefixed if it is partitioned on a left prefix of the index columns.
- **Local Nonprefixed Indexes**
- A local index is nonprefixed if it is not partitioned on a left prefix of the index columns. Therefore, it is not possible to have a unique local nonprefixed index unless the partitioning key is a subset of the index key.
- **Global Partitioned Indexes**
- In a global partitioned index, the keys in a particular index partition may refer to rows stored in more than one underlying table partition or subpartition.
- A global index can be range or hash partitioned, though it can be defined on any type of partitioned table.
- A global index is created by specifying the GLOBAL attribute.
- Index partitions can be merged or split as necessary.

GUIDELINES TO INDEX PARTITIONING

- **Global Partitioned Indexes (continued)**
- Normally, a global index is not equipartitioned with the underlying table and usualy nothing could prevent this. An index that must be equi-partitioned with the underlying table should be created as LOCAL.
- A global partitioned index contains a single B-tree with entries for all rows in all partitions. Each index partition may contain keys that refer to many different partitions or subpartitions in the table.
- The highest partition of a global index must have a partition bound all of whose values are MAXVALUE.
- **Prefixed and Non-Prefixed Global Partitioned Indexes**
- A global partitioned index is prefixed if it is partitioned on a left prefix of the index columns.
- Global prefixed partitioned indexes can be unique or nonunique.
- Nonpartitioned indexes are treated as global prefixed nonpartitioned indexes.

GUIDELINES TO INDEX PARTITIONING

- **Management of Global Partitioned Indexes**
- Global partitioned indexes are harder to manage than local indexes.
- When the data in an underlying table partition is moved or removed (SPLIT, MOVE, DROP, or TRUNCATE), all partitions of a global index are affected. So, global indexes do not support partition independence.
- When an underlying table partition or subpartition is recovered to a point in time, all corresponding entries in a global index must be recovered to the same point in time. Because these entries may be scattered across all partitions or subpartitions of the index, mixed in with entries for other partitions or subpartitions that are not being recovered, there is no way to accomplish this except by re-creating the entire global index.
- When deciding how to partition indexes on a table, consider the mix of applications that need to access the table.
- There is a trade-off between performance and availability, and manageability.

GUIDELINES TO INDEX PARTITIONING

- **For OLTP applications**
- Global indexes and local prefixed indexes provide improved performance over local non-prefixed indexes because they minimize the number of index partition probes.
- Local indexes support more availability when there are partition or subpartition maintenance operations on the table.
- Local non-prefixed indexes are very useful for historical databases.
- **For DSS applications**
- Local non-prefixed indexes can improve performance because many index partitions can be scanned in parallel by range queries on the index key.
- For historical tables, indexes should be local if possible. This limits the impact of regularly scheduled drop partition operations.
- Unique indexes on columns other than the partitioning columns must be global because unique local non-prefixed indexes whose key does not contain the partitioning key are not supported.

TYPES OF INDEX PARTITIONING: SUMMARY

Type of Index	Index Equipartitioned with Table	Index Partitioned on Left Prefix of Index Columns	UNIQUE Attribute Allowed	Example: Table Partitioning Key	Example: Index Columns	Example: Index Partitioning Key
Local Prefixed (any partitioning method)	Yes	Yes	Yes	A	A, B	A
Local Nonprefixed (any partitioning method)	Yes	No	Yes[Foot 1]	A	B, A	A
Global Prefixed (range partitioning only)	No[Foot 2]	Yes	Yes	A	B	

[Footnote 1] For a unique local nonprefixed index, the partitioning key must be a subset of the index key.

[Footnote 2] Although a global partitioned index may be equipartitioned with the underlying table, Oracle does not take advantage of the partitioning or maintain equipartitioning after partition maintenance operations such as DROP or SPLIT PARTITION.

TABLE COMPRESSION AND BITMAP INDEXES

- When using table compression on partitioned tables with bitmap indexes, you need to do the following before introducing the compression attribute for the first time:

1. Mark bitmap indexes unusable.
2. Set the compression attribute.
3. Rebuild the indexes.

PARTITION STRATEGY CHOICE: RECOMMENDATIONS

- **When to Use Range or Interval Partitioning**
- Range partitioning is a convenient method for partitioning historical data.
- The boundaries of range partitions define the ordering of the partitions in the tables or indexes.
- Interval partitioning is an extension to range partitioning in which, beyond a point in time, partitions are defined by an interval. Interval partitions are automatically created when the data is inserted into the partition.
- Range or interval partitioning is often used to organize data by time intervals on a column of type DATE.
- For instance, keeping the past 48 months' worth of data online, Range partitioning simplifies this process. To add data from a new month, the DBA will load it into a separate table, clean it, index it, and then add it to the range-partitioned table using the EXCHANGE PARTITION statement, all while the original table remains online.
- After adding the new partition, the DBA can drop the trailing month with the DROP PARTITION statement.

PARTITION STRATEGY CHOICE: RECOMMENDATIONS

- **When to Use Hash Partitioning**
- There are scenarios when it is not trivial into which partition data should reside, although the partitioning key can be identified. With hash partitioning, a row is placed into a partition based on the result of passing the partitioning key into a hashing algorithm.
- When using this approach, data is randomly distributed across the partitions rather than grouped together.
- Hence, this is a great approach for some data, but may not be an effective way to manage historical data.
- Partition pruning is limited to equality predicates.
- Hash partitioning also supports partition-wise joins, parallel DML and parallel index access.
- Excellent when the DBA needs to enable partial or full parallel partition-wise joins with very likely equi-sized partitions or distribute data evenly among the nodes of an MPP platform using RAC, thus minimizing interconnect traffic when processing internode parallel statements.

PARTITION STRATEGY CHOICE: RECOMMENDATIONS

- **When to Use List Partitioning**
- It is recommended to use list partitioning when you want to specifically map rows to partitions based on discrete values.
- **When to Use Composite Partitioning**
- Composite partitioning offers the benefits of partitioning on two dimensions. From a performance perspective, it benefits from partition pruning on one or two dimensions depending on the SQL statement, taking advantage of both full or partial partition-wise joins on either dimension, as needed.
- It can benefit from parallel backup and recovery of a single table (manageability perspective).
- The DBA can split up backups of your tables and you can decide to store data differently based on identification by a partitioning key.
- The database stores every subpartition in a composite partitioned table as a separate segment.
- Thus, the subpartitions may have properties that differ from the properties of the table or from the partition to which the subpartitions belong.

PARTITION STRATEGY CHOICE: RECOMMENDATIONS

- **When to Use Composite Range-Hash Partitioning**
- Composite range-hash partitioning is particularly common for tables that store history, are very large as a result, and are frequently joined with other large tables,
  - Then, composite range-hash partitioning provides the benefit of partition pruning at the range level
  - Opportunity to perform parallel full or partial partition-wise joins at the hash level. Specific cases can benefit from partition pruning on both dimensions for specific SQL statements.
- Composite range-hash partitioning can also be utilized for tables that traditionally use hash partitioning, but also use a rolling window approach.

PARTITION STRATEGY CHOICE: RECOMMENDATIONS

- **When to Use Composite Range-List Partitioning**
- Composite range-list partitioning is mostly used for large tables that store historical data and are usually accessed on more than one dimension.
- **When to Use Composite Range-Range Partitioning**
- Composite range-range partitioning is helpful for applications that store time-dependent data on more than one time dimension.
- Business cases for composite range-range partitioning could include ILM scenarios, and applications that store historical data and need to categorize its data by range on another dimension.
- **When to Use Composite List-Hash Partitioning**
- Composite list-hash partitioning is utilized for large tables that are usually accessed on one dimension, but because of their size need yet to take advantage of parallel full or partial partition-wise joins.
- **When to Use Composite List-List Partitioning**
- Composite list-list partitioning is helpful for large tables that are often accessed on different dimensions. The DBA can explicitly map rows to partitions on those dimensions on the basis of discrete values.

PARTITION STRATEGY CHOICE: RECOMMENDATIONS

- **When to Use Composite List-Range Partitioning**
- Composite list-range partitioning is advantageous for large tables that are accessed on different dimensions. For the most commonly used dimension, the DBA can explicitly map rows to partitions on discrete values.
- List-range partitioning is likely to be used for tables that use range values within a list partition; in contrast range-list partitioning is mostly used for discrete list values within a range partition.
- List-range partitioning is less likely to be used to store historical data, although equivalent scenarios all work. Range-list partitioning can be implemented using interval-list partitioning, while list-range partitioning does not support interval partitioning.

PARTITION STRATEGY CHOICE: RECOMMENDATIONS

- **When to Use Interval Partitioning**
- Interval partitioning can be used for every table that is range partitioned and uses fixed intervals for new partitions. The database automatically creates interval partitions as data for that partition is loaded. Until this happens, the interval partition exists but no segment is created for the partition.
- The benefit of interval partitioning is that there is no need to create your range partitions explicitly. Therefore, a DBA could consider using interval partitioning unless there is a need to create range partitions with different intervals, or a need to specific partition attributes when creating range partitions.
- When upgrading an application it is recommended to use range partitioning or composite range-* partitioning, accordingly.

PARTITION STRATEGY CHOICE: RECOMMENDATIONS

- **When to Use Reference Partitioning**
- Reference partitioning is effective in the following scenarios:
- When denormalizing or planning to denormalize, a column from a master table into a child table in order to get partition pruning benefits on both tables.
- If two large tables are joined often, then the tables are not partitioned on the join key, but you want to take advantage of partition-wise joins.
- Indeed, reference partitioning implicitly enables full partition-wise joins.
- If data in multiple tables has a related life cycle, then reference partitioning can provide significant manageability benefits.
- Partition management operations against the master table are automatically cascaded to its descendents. For example, when adding a partition to the master table, that creation is automatically propagated to all its descendents.
- In order to use reference partitioning, the DBA has to enable and enforce the foreign key relationship between the master table and the reference table in place.
- It is also possible to cascade reference-partitioned tables based on the data model used.

PARTITION STRATEGY CHOICE: RECOMMENDATIONS

- **When to Partition on Virtual Columns**
- Virtual column partitioning enables you to partition on an expression, which may use data from other columns, and perform calculations with these columns.
- There is no support for PL/SQL function calls on a virtual column definitions as a partitioning key.
- Virtual column partitioning supports all partitioning methods as well as performance and manageability features.
- Virtual columns could be used when tables are frequently accessed using a predicate that is not directly captured in a column, but can be derived, in order to get partition pruning benefits.
- The virtual column does not require any storage.

ORACLE PARTITIONING FOR ILM SUPPORT

- **Oracle Database for ILM**

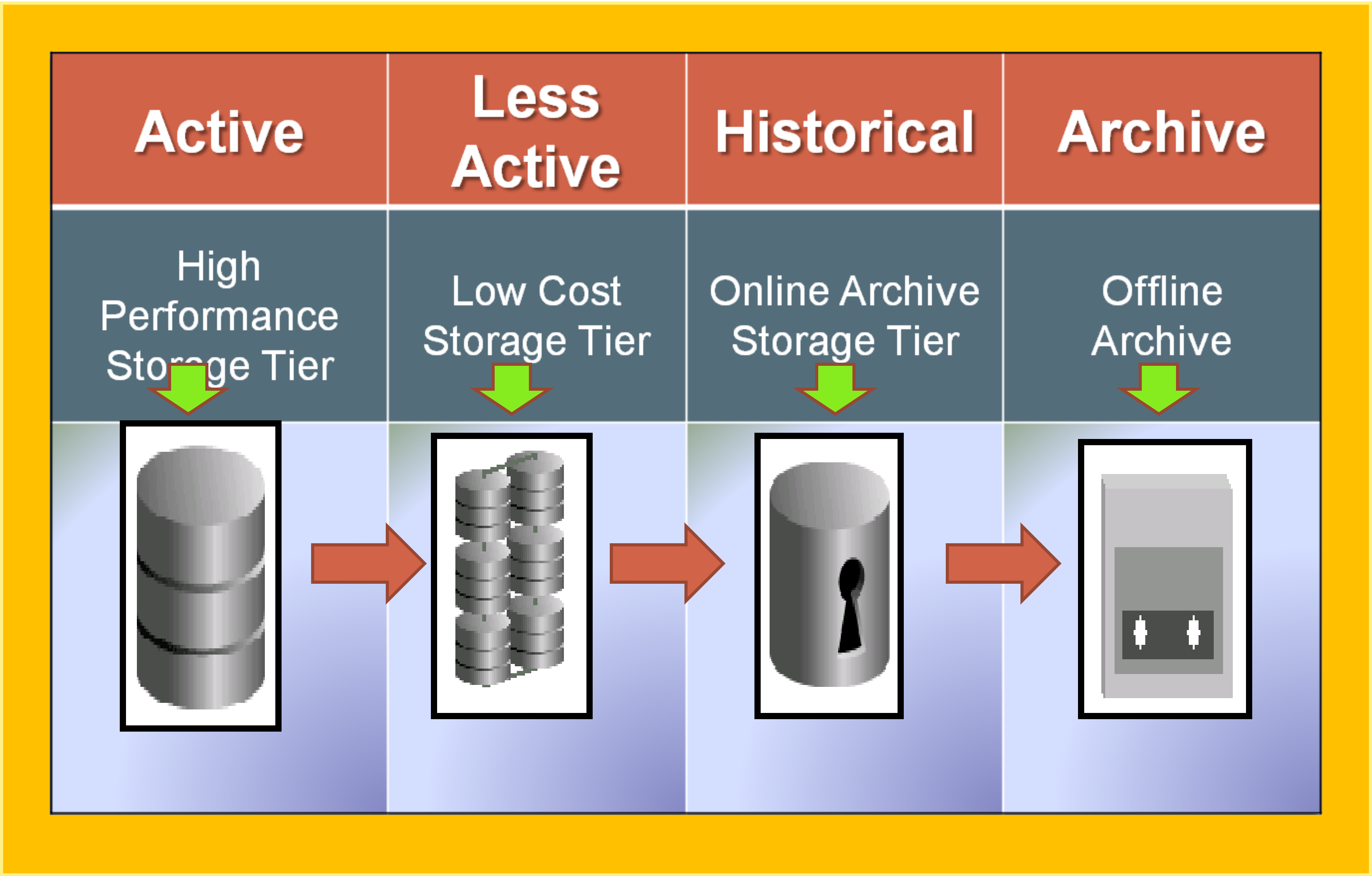
The Oracle Database Partitioning option provides an uniquely ideal platform for implementing an ILM solution offering:

- **Application Transparency**
  - There is no need to customize applications
  - Data can easily be moved and accessed at the different stages of its lifecycle.
  - Flexibility required to quickly adapt to any new regulatory compliance.
- **Fine-grained**
  - View data at a very fine-grained level as well as group related data together, whereas storage devices only see bytes and blocks.
- **Low-Cost**
  - Low cost storage is a key factor in implementing ILM.
- **Enforceable Compliance Policies**
  - It is imperative to show to regulatory bodies that data is being retained and managed in accordance with the regulations defining security and audit policies, which enforce and log all access to data.

ORACLE PARTITIONING FOR ILM SUPPORT

- In general, Enforceable Compliance Policies where Oracle Partitioning is valuable involve:
  - Data Retention
  - Immutability
  - Privacy
  - Auditing
  - Expiration

ORACLE PARTITIONING FOR ILM SUPPORT



ILM PARTITIONING STRATEGY

1. Define Logical Tiers
2. Define the Data Life Cycle
3. Generate Partition Advice, if not table not partitioned
4. Assign Table to a Lifecycle

ORACLE PARTITIONING FOR DATAWAREHOUSING

- Datawarehouses often require techniques both for managing large tables and providing good query optimization.
- Oracle Partitioning is beneficial in attaining the following Datawarehousing goals, namely:
  - Scalability
  - Performance
  - Manageability

ORACLE PARTITIONING FOR DATAWAREHOUSING

- **Scalability**
- Partitioning is effective scaling a data warehouse by dividing database objects into smaller pieces, enabling access to smaller, more manageable objects. Providing direct access to smaller objects addresses the scalability requirements of data warehouses:
  - Bigger Database
  - Bigger Individual tables: More Rows in Tables
  - More Users Querying the System
  - More Complex Queries

ORACLE PARTITIONING FOR DATAWAREHOUSING

- **More Users Querying the System**
  - With partitioning, users are more likely to hit isolated and smaller data sets and the database returns results faster with less data contention.
  - More Complex Queries
  - Smaller data sets help perform complex queries faster (in memory processing and less I/O overhead.)
  - Performance
  - Optimal performance is a key to success for a data warehouse. Analyses run against the database should return within a reasonable amount of time even on terabyte-size tables.

ORACLE PARTITIONING FOR DATAWAREHOUSING

- **Partition Pruning**
  - Partition pruning is an essential performance feature since the optimizer analyzes FROM and WHERE clauses in SQL statements to eliminate unneeded partitions when building the partition access list.
  - Partition pruning greatly reduces the amount of data retrieved from disk and shortens processing time, thus improving query performance and optimizing resource utilization.
- **Basic Partition Pruning Techniques**
  - The optimizer utilizes a wide variety of predicates for pruning. The three predicate types, equality, range, and IN-list, are the most commonly used cases of partition pruning.
- **Advanced Partition Pruning Techniques**
  - Oracle also prunes in the presence of more complex predicates or SQL statements involving partitioned tables. For instance, when a partitioned table is joined to the subset of another table, constrained by a WHERE clause condition.

ORACLE PARTITIONING FOR DATAWAREHOUSING

- **Partial Partition-Wise Joins**
  - Oracle Database can perform partial partition-wise joins only in parallel.
  - To execute a partial partition-wise join, the database dynamically repartitions the other table based and the execution is similar to a full partition-wise join.
- **Benefits of Partition-Wise Joins**
  - Reduction of Communications Overhead
  - Reduction of Memory Requirements

ORACLE PARTITIONING FOR DATAWAREHOUSING

- **Partitioning Materialized Views**
- The underlying storage for a materialized view is a table structure,and therefore partitioning materialized views is quite similar.
- When the database rewrites a query to run against materialized views, the query can take advantage of the same performance features as those queries running against tables MV's directly benefit from.
- A rewritten query may eliminate materialized view partitions and it can take advantage of partition-wise joins, when joins back to tables or with other materialized views are necessary.

ORACLE PARTITIONING FOR DATAWAREHOUSING

- **Partitioning Materialized Views (continued)**
  - This sample code illustrates how to effectively create a compressed materialized view partitioned by hash, which using an aggregation on period_code.

```
SQL> CREATE MATERIALIZED VIEW ANTHONY.mv_pro_marketing_campaign
     PARTITION BY HASH(period_code)
     PARTITIONS 26 COMPRESS FOR ALL OPERATIONS PARALLEL NOLOGGING
     ENABLE QUERY REWRITE
     AS SELECT campaign_id,
               campaign_name,
               campaign_date,
               period_code,
               sum(projected_sales) camp_proj_period_sales
          FROM pro_marketing_campaigns
          GROUP BY campaign_id,
                   campaign_name,
                   campaign_date,
                   period_code;

Materialized view created.
```

ORACLE PARTITIONING FOR DATAWAREHOUSING

- **Partitioning Materialized Views (continued)**
  - **Partition Exchange Load (PEL)**
  - Partitions can be added using Partition Exchange Load (PEL). When using PEL, a separate identical table to a single partition is created, including the same indexes and constraints, if any.
  - **Partitioning and Materialized View Refresh Strategies**
    - Full refresh
    - Fast (incremental) refresh based on materialized view logs against the base tables
    - Manually using DML, followed by ALTER MATERIALIZED VIEW CONSIDER FRESH
  - To enable query rewrites, set the QUERY_REWRITE_INTEGRITY initialization parameter.
    - To manually keep materialized views up to date, the init.ora parameter QUERY_REWRITE_INTEGRITY must be set to either TRUSTED or STALE_TOLERATED.

ORACLE PARTITIONING FOR DATAWAREHOUSING

- **Partitioning Materialized Views (continued)**
- When using materialized views and base tables with comparable partitioning strategies, then PEL can be an extremely powerful way to keep materialized views up-to-date manually.
- Here is how PEL can work:
  - Create tables to enable PEL against the tables and materialized views
  - Load data into the tables, build the indexes, and implement any constraints
  - Update the base tables using PEL
  - Update the materialized views using PEL
  - Execute ALTER MATERIALIZED VIEW CONSIDER FRESH for every materialized view you updated using this strategy

ORACLE PARTITIONING FOR OLTP

- Partitioning is often used for Online Transaction Processing (OLTP) systems to reduce contention in order to support a very large user population since (OLTP) systems are one of the most common data processing systems in today's enterprises, including, for instance, financial and retail systems. Partitioning also helps in addressing regulatory requirements facing OLTP systems, including storing larger amounts of data in a cost-effective manner.
- **Oracle partitioning effectively optimizes OLTP focus on:**
  - Performance
  - Manageability
  - Availability

ORACLE PARTITIONING FOR OLTP

- **Partitioning also effectively addresses OLTP features such as, namely:**
  - Short response time
  - Small transactions
  - Data maintenance operations
  - Large user populations
  - High concurrency
  - Large data volumes
  - High availability
  - Lifecycle related data usage

STORAGE MANAGEMENT

- **High Availability: Implementing storage redundancy.**
  - Hardware-based mirroring
  - Using ASM for mirroring
  - Software-based mirroring not using ASM
- **Performance: optimum throughput from storage devices, multiple disks must work in parallel.**
  - Hardware-based striping
  - Software-based striping using ASM
  - Software-based striping not using ASM

STORAGE MANAGEMENT

- **ILM**
- In an Information Lifecycle Management environment, it is not possible to use striping across all devices, because all data would then be distributed across all storage pools, in contrast with different storage pools typically involving different performance characteristics.
- **Partition Placement**
  - Using Bigfile Tablespaces
  - Customization
  - Oracle Exadata

LOB PARTITIONING SUPPORT

- Oracle Partitioning support the storage of LOB types, such as BLOBs and BFILE types.
- One option for LOB support can be attained via Index Organized Tables, storing LOBs in a separate tablespace.

```
SQL>
SQL>
SQL> CREATE TABLE new_mktg_campaigns2
  2  ( campaign_id                 NUMBER(8)
  3  , campaign_name               VARCHAR2(20)
  4  , projected_sales                     NUMBER(12,2)
  5  , period_code                         INTEGER CONSTRAINT rck CHECK (period_code BETWEEN 1 AND 26)
  6  ,campaign_flier_img                 BLOb
  7  , campaign_desc               VARCHAR2(4000),
  8    PRIMARY KEY (campaign_id, period_code)
  9  )    ORGANIZATION INDEX
 10       INCLUDING period_code OVERFLOW TABLESPACE T11
 11       PARTITION BY RANGE (period_code)
 12              (PARTITION VALUES LESS THAN (10) TABLESPACE t1,
 13               PARTITION VALUES LESS THAN (20) TABLESPACE t2 OVERFLOW TABLESPACE t9,
 14               PARTITION VALUES LESS THAN (MAXVALUE) TABLESPACE t13);

Table created.

SQL>
```

LOB PARTITIONING SUPPORT

Partitioning LOB Support can also be via explicit LOB storage, e.g. as BFILE, CLOB, or BLOB.

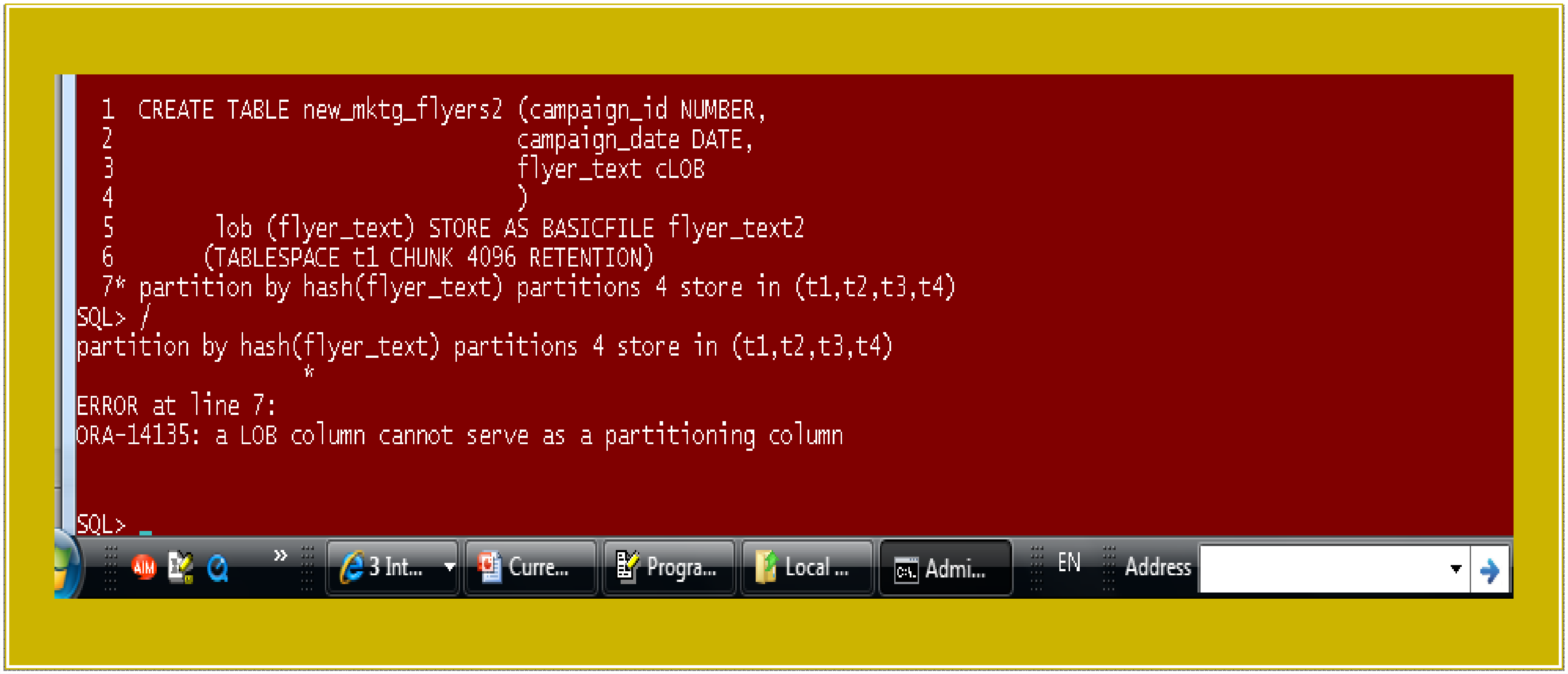
```
  1  CREATE TABLE new_mktg_flyers (campaign_id NUMBER,
  2                               campaign_date DATE,
  3                               flyer_text cLOB
  4                               )
  5        lob (flyer_text) STORE AS BASICFILE flyer_text
  6       (TABLESPACE t1 CHUNK 4096 RETENTION)
  7  partition by range(campaign_id)
  8  (partition p1 values less than (10) tablespace t1,
  9   partition p2 values less than (50) tablespace t2,
 10   partition p3 values less than (maxvalue) tablespace t3
 11* )
SQL> /

Table created.

SQL>
```

LOB PARTITIONING SUPPORT

However, an LOB column should not be used as a partition key column itself.



PARTITIONING SUPPORT FOR USER-DEFINED DATATYPES

User-datatype s can be used in a partitioned table.

```
SQL> CREATE TYPE emailaddr_t AS OBJECT (email VARCHAR2(128));
     /

Type created.

SQL> CREATE TYPE email_list AS TABLE OF emailaddr_t;
     /

Type created.

SQL> CREATE TYPE prospect_t AS OBJECT (
                                        prospect_id     NUMBER,
                                        prospect_name   VARCHAR2(25),
                                        prospect_emails email_list);
     /

Type created.

SQL> CREATE TYPE prospect_list AS TABLE OF prospect_t;
     /

Type created.

SQL>
SQL> CREATE TABLE partners_direct_mktg
       ( promo_id              NUMBER(6) NOT NULL
       , cust_id               NUMBER NOT NULL
       , campaign_date         DATE NOT NULL
       , channel_code          CHAR(1) NOT NULL
       , campaign_id           NUMBER(6) NOT NULL
       , partner_name       VARCHAR2(25)
       , partner_reps       prospect_list
       )
        NESTED TABLE  partner_reps      STORE AS outer_ntab
        (NESTED TABLE prospect_emails   STORE AS inner_ntab)
     PARTITION BY RANGE (campaign_date) INTERVAL (NUMTOYMINTERVAL(1,'MONTH'))
      (PARTITION p_campaign_prior_2009 VALUES LESS THAN  (TO_DATE('01-JAN-2009','dd-MON-yyyy')) )
     ENABLE ROW MOVEMENT COMPRESS PARALLEL NOLOGGING;

Table created.

SQL>
```

PARTITIONING SUPPORT FOR USER-DEFINED DATATYPES

User-defined object ype using the object id (OID) as primary key .

```
SQL>
SQL> rem creating phone type (phone_typ)
SQL> CREATE TYPE phone_typ AS OBJECT
  2     (phone_no NUMBER, phone_contact CHAR(30));
  3   /

Type created.

SQL>
SQL> rem creating phone obj table (phone_obj_t)
SQL> CREATE TABLE phone_obj_t OF phone_typ (phone_no PRIMARY KEY)
  2     OBJECT IDENTIFIER IS PRIMARY KEY;

Table created.
```

```
SQL> rem creating phone table (phone_list_tab)
SQL> CREATE TABLE phone_list_tab
  2     (contact_no     NUMBER,
  3      mgr_contact REF phone_typ SCOPE IS phone_obj_t)
  4     PARTITION BY HASH (contact_no) partitions 4
  5     STORE IN (t1,t2,t3,t4);

Table created.
```

```
SQL> CREATE TABLE phone_list_tab2
  2     (contact_no     NUMBER,
  3      mgr_contact REF phone_typ
  4      CONSTRAINT mgr_contact_listed REFERENCES phone_obj_t)
  5     PARTITION BY HASH (contact_no) partitions 4
  6     STORE IN (t1,t2,t3,t4);

Table created.

SQL> _
```

PARTITIONING SUPPORT FOR NESTED TABLES

Nested tables can be used in a partitioned table.

```
CREATE TABLE business_prospect (
  partner_name VARCHAR2(25),
  partner_reps prospect_list)
  NESTED TABLE partner_reps STORE AS prospect_outer_ntab
  (NESTED TABLE prospect_emails STORE AS prospect_inner_ntab)
 PARTITION BY HASH(partner_name) PARTITIONS 4 STORE IN
(t1,t2,t3,t4);
```

```
SQL>
SQL>
SQL> CREATE TABLE business_prospect (
  2      partner_name VARCHAR2(25),
  3      partner_reps prospect_list)
  4      NESTED TABLE partner_reps STORE AS prospect_outer_ntab
  5      (NESTED TABLE prospect_emails STORE AS prospect_inner_ntab)
  6    partition by hash(partner_name) partitions 4 store in (t1,t2,t3,t4)
  7   /

Table created.

SQL> _
```

PARTITIONING SUPPORT FOR VARRAYS

Like Nested tables, Varrays can be also used in a partitioned table.

```
CREATE TYPE member_emails IS VARRAY(30) OF VARCHAR2(128);
/
CREATE TABLE partners_online_dir (partner_id  number,
                                  territory        VARCHAR2(4),
                                  edom             VARCHAR2(30),
                                  partner_        emails member_emails)
PARTITION BY RANGE(edom)
  (PARTITION p1 VALUES LESS THAN  ('aol.com') TABLESPACE T1,
   PARTITION p2 VALUES LESS THAN  ('msn.com') TABLESPACE T2,
   PARTITION p3 VALUES LESS THAN  ('yahoo.com') TABLESPACE T3,
   PARTITION p4 VALUES LESS THAN  (MAXVALUE) TABLESPACE T4  )
ENABLE ROW MOVEMENT  COMPRESS FOR ALL OPERATIONS;
/
```

```
SQL> CREATE TYPE member_emails IS VARRAY(30) OF VARCHAR2(128);
  2  /

Type created.

SQL> CREATE TABLE partners_online_dir (partner_id          number,
  2                                    territory           VARCHAR2(4),
  3                                    edom                VARCHAR2(30),
  4                                    partner_emails member_emails
  5                                    )
  6  PARTITION BY RANGE(edom)
  7     (PARTITION p1 VALUES LESS THAN  ('aol.com') TABLESPACE T1,
  8      PARTITION p2 VALUES LESS THAN  ('msn.com') TABLESPACE T2,
  9      PARTITION p3 VALUES LESS THAN  ('yahoo.com') TABLESPACE T3,
 10      PARTITION p4 VALUES LESS THAN  (MAXVALUE) TABLESPACE T4
 11     )
 12  ENABLE ROW MOVEMENT  COMPRESS FOR ALL OPERATIONS;

Table created.
```

PARTITIONING SUPPORT FOR XML

XML Data types can be included in partitioned tables, as illustrated below.

```
SQL>
SQL>
SQL>
SQL>
SQL> CREATE TABLE x_adnxml_tab (
  2      adn_id          NUMBER,
  3      adn_desc        XMLTYPE)
  4    PARTITION BY HASH(adn_id) PARTITIONS 4 STORE IN (T2,T4,T6, T8)
  5      XMLTYPE adn_desc STORE AS CLOB
  6      (TABLESPACE t8
  7       STORAGE (INITIAL 8192 NEXT 8192)
  8       CHUNK 4000 NOCACHE LOGGING
  9      );

Table created.

SQL> _
```

BEST PRACTICES

From the business and functional point of view, a partitioning strategy is normally identified with a functional goal-seeking perspective, and therefore it needs to be mapped to an Oracle partitioning technical recommendation or specific partitioning strategy matching those business requirements, regulatory compliance , or systems platform, among others.

BEST PRACTICES

- Use Oracle partitioning strategic recommendations for each database system environment accordingly.
- When in doubt refer to sample code, forum discussions, and case studies.
- Consolidate recommendations made in this presentation into a practical enterprise policy framework.

TIPS AND TECHNIQUES

- Using multiple block size caches can increase load throughput in DSS, in particular, when using indexes in a block size larger than the table.
- This is more important volumes are based on a (Stripe and mirror everything) SAME-approach (i.e.,RAID 0+1).
- Likewise, performance optimization and contention reduction can be attained in OLTP systems using the same approach, when the appropriate partitioning strategy is being used, in accordance to the strategic recommendations previously made.

CONSTRAINTS

- As previously stated, there is no support for LONG and LONG RAW data types on any Oracle partitioned object or any partitioning strategy discussed.
- Likewise, an encrypted column cannot serve as partitioning key.
- When migrating to Oracle11g or any other recent release, consider changing LONG and LONG RAW datatypes into CLOB, BLOB accordingly for current and future release forward compatibility and improved manageability.
- A VARRAY of XML data types cannot be set in a partitioned table (via an SQL DDL statement.)
- Certain datatypes have size and store constraints like LOBs or large VARCHAR2 definitions.

CONCLUDING REMARKS

- Oracle partitioning provides effective strategies to attain time and resource optimization, including CPU and memory.
- Oracle Partitioning option is extremely practical to achieve regulatory compliance.
- Oracle partitioning is mission-critical to attain most needed scalability, manageability, performance, and high-availability in any system platform.