**A Comprehensive Guide to Oracle Partitioning with Samples**

The research presented focuses on partitioning strategy options for each currently available Oracle version, in particular Oracle11g. The study guideline is on implementing the new partitioning options available in 11g, while enhancing performance tuning for the physical model and applications involved. Therefore, basic strategies— such as range, hash and list—, composite partitioning strategies including all possible combinations of Basic strategies, and Partition Extensions such as Reference and Interval partitioning strategies are covered. The research covers both middle-to-large-size and VLDB databases with significant implications for consolidation, systems integration, high-availability, and virtualization support. Topics covered emphasize subsequent performance tuning and specific application usage such as IOT-partitioning and composite partitioning choices.  
  
1. Introduction

Encountering that performance for a table whose segments have exceeded a good number of Gigabytes or even Terabytes is probably the main consideration to partition that table. Indeed, that table will require special segment access control, and thus, special index architecture and administration. Ultimately, the usage of multiple block size tablespaces and buffer caches become a database technology that greatly enhances a partitioning approach.  
In principle, tables (including materialized views), indexes, and Index-Organized Tables (IOT) are the objects used as target in any valid partitioning strategy utilized. Partitioning strategies can involve primarily Basic, Composite and Partition Extensions.

2. Partitioning Strategies

Oracle Partitioning strategies include, namely:

2.1 Basic Partitioning (Single-Level)

This strategy involves one of the following options: Partition by Range (Establishes ranges within the domain used as partitioning key), List (Provides a list of values matching one partition in the partition key domain and a default partition for those not matched) or Hash (which transforms the partitioning key value and maps it to given partition).   
  
2.2 Composite Partitioning

The following combinations of basic partitioning result into valid composite partitioning strategies, namely:

* Range-Range
* Range-List
* Range-Hash
* List-List
* List-Range
* List-Hash
* Interval Partitioning

The Interval Partitioning strategy is fundamentally a special implementation of Range partitioning, which maps primarily a DATE, TIMESTAMP data type to a numeric interval, using the INTERVAL keyword, use as a partition range marker. The functions NUMTOYMINTERVAL and NUMTODSINTERVAL are commonly used. Interval partitioning can occur as a single-level strategy or composite option in combination with all other options, namely, Range, Hash and List.

2.3 Partition Extensions

This partitioning strategy involves:

2.3.1 Reference Partitioning

This strategy normally uses the referential integrity constraint between to table, and uses the key in the details table to attain partition on the referenced key, which points to a candidate primary key in another partitioned table, the master table. The referential integrity constraint must be enabled and enforced.

2.3.2 Virtual Column Partitioning

This option permits to partition of a column on a virtual column, which is usually the outcome of a mathematical operation on two or more actual columns on the same table. This option extends every basic partitioning strategy.

3. Object Partitioning

In addition to the overall partitioning strategies, it is important to look at partitioning from the database objects to be partitioned, which include:

3.1 Tables (including Materialized Views)  
  
Tables strictly support each Basic and Composite strategy and all Partition Extension, constrained by any SQL DDL rule.

3.2 Indexes

Indexes can have local or global partitions. Locally partitioned indexes can be pre-fixed (if they partitioned on the left portion of the key) or non-prefixed, otherwise. A local index has a one-to-one correspondence with the underlying table, and can reside in the same or on a different tablespace, which could even be of a different block size, a strategy that often enhances database performance. Index Partitioning can support Basic Partitioning strategies in general.  
  
3.3 Index-Organized Tables

Like tables, Index-Organized Tables support all partitioning Basic strategies, and –in general– the partitioning key must be a subset of the primary key.

4. Creating Partitions  
  
It is exceptionally good to have the experience of creating partitions under any possible strategy, namely:  
  
4.1 Creating Range-Partitioned Tables  
  
This sample code creates a table with four partitions and enables row movement:  
  
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;

4.2 Creating Global Indexes

Creating a range-partitioned global index is similar to creating range-partitioned table. The following 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));  
  
4.3 Creating an Interval Range-Partitioned Table  
  
An Interval Range-Partitioned table is a special case of a Range Partitioned Table. The following are important features and relevant issues:

* 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.  
  The following sample code 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.

4.4 Creating Hash-Partitioned Tables

Hash-Partitioned tables map the insertion location for any row via a hashing algorithm that determines the appropriate tablespace for the partition key instance. The following example illustrates a typical case:  
  
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 needed 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.  
  
4.5 Creating List-Partitioned Tables  
  
List-partitioned tables follow the following rules among others:  
  
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.) In fact, 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.  
  
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);

4.6 Creating Reference-Partitioned Tables

In many cases, it is recommended to use referential integrity constraint in a strategic way to partition a master-detail table scenario, accordingly. The following concerns are quite relevant:

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.  
  
Besides, 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.

The following sample code illustrates the creation of a reference-partitioned table preceded by the creation a of master table and a details table. The referential integrity constraint is highlighted as a key partition marking strategy.  
  
4.6.1 Master Table DDL  
  
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')) );

4.6.2 Details Table DDL  
  
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);

4.7 Creating Local Partitioned Indexes

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

Similarly, 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 SCHOOL\_DIRECTORY table:  
  
CREATE INDEX ndx\_gd ON school\_directory (email) LOCAL  
PARTITIONS 4 STORE IN (t1, t2, t3, t4);

Naturally, 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.

4.8 Creating a Hash-Partitioned Global Index

The following sample code illustrates the creation of a global hash-partitioned index.

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);  
  
The syntax is similar to that used for a hash partitioned table.  
In most instances, 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.  
  
4.9 Creating Range-Hash Partitioned Tables  
  
Range-Hash partitioned tables are probably the most common type among the composite partitioning strategies. 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 SUBPARTITION. In fact, it is important to consider the following issues, namely:

* 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 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'))  
);  
  
4.10 Creating Range-List Partitioned Tables  
  
Range-List partitioned tables are subject to range rules at the first partitioning level and list rules at second, list partitioning level, accordingly.  
  
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')  
) );

4.11 Creating List-Hash Partitioned Tables

This example shows the 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);

4.12 Creating List-Range Partitioned Tables

The following 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)  
)  
);

4.13 Creating 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')  
)  
, 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')  
);

4.14 Creating Range-Hash Partitioned Tables Using a Subpartition Template  
  
The following sample code illustrates how to use a subpartition template to create a composite Range-Hash Partition Table.  
  
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'))  
);

4.15 Creating a Multicolumn Range-Partitioned Table

The following sample code illustrates the use of a multicolumn partitioned approach for table BI\_AUTO\_RENTALS\_SUMMARY. 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. The purpose of this type of partition is to avoid dispersion of data within the range and balance the physical usage of each partition entries for performance tuning optimization purposes.

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));

4.16 Creating a Virtual Column Based Partitioned Table

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.

The next sample code shows the DIRECT\_MARKETING 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.

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;  
  
The reader could note that the hist\_avg\_sales and sales\_forecast are two real columns in the DIRECT\_MARKETING table of NUMBER type, which are used in the description of the virtual column.

4.17 Using Compression and Partitioning

The following important issues apply to partitioning when using compression as well:

* 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.

The following 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.  
  
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\_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;

4.18 Using Partitioned Index Key Compression

The following recommendations apply to partitioned-index key compression, namely:

* 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  
);

Likewise, 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.

4.19 Creating Range-Partitioned Index-Organized Tables

In general, it is possible to partition index-organized tables, and their secondary indexes, by the range method. The following 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);

4.20 Creating 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);

4.21 Creating a List-Partitioned Index-Organized Table

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

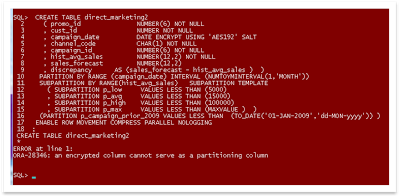
CREATE TABLE current\_mktg\_campaigns  
( campaign\_id NUMBER(8)  
, period\_code INTEGER CONSTRAINT fpclst\_ck  
CHECK (period\_code BETWEEN 1 AND 26)  
, campaign\_name VARCHAR2(20)  
, projected\_sales NUMBER(12,2)  
, campaign\_desc VARCHAR2(4000),  
PRIMARY KEY (campaign\_id, period\_code))  
ORGANIZATION 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);  
  
  
4.22 Creating Composite Interval-RangeHashList Partitioned Tables

In comparison to previous examples, the composite interval range-partitioned tables subpartitioned by any of range, hash or list provides a very powerful strategy in using a time domain, such as those associate with a DATE OR TIMESTAMP column. The following 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;

4.23 Using Encryption in Partitioned Tables

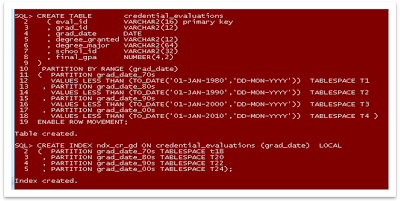
While encryption is supported in partitioned tables, it is not possible to partitioned a table on an encrypted column under any partitioning strategy.

[](http://4.bp.blogspot.com/_wWbZXrYfXrE/SjfIRTrBHCI/AAAAAAAABO0/V1wZeTTMeZI/s1600-h/image001.png)*Exhibit 1. Partitioning Key and Encryption on the Same Column are Incompatible*.[[1]](http://www.blogger.com/post-edit.g?blogID=4535123449935735221&postID=5345978750357959934" \l "_ftn1" \o ")

[[1]](http://www.blogger.com/post-edit.g?blogID=4535123449935735221&postID=5345978750357959934" \l "_ftnref1" \o ") Removing the encryption option will allow for the table to be created.

4.24 Using Multiple Block Size Caches

When using multiple block size databases, i.e., those using more than one tablespace block size and associated buffer cache, there are many congruent features to take advantages of to a higher performance optimization.

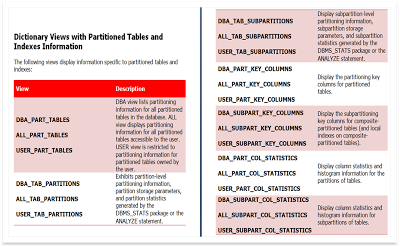
[](http://3.bp.blogspot.com/_wWbZXrYfXrE/SjfGhb70ttI/AAAAAAAABOs/vrw8aGCEIIA/s1600-h/image003.png)*Exhibit 2. Using multiple Block*Size *to Optimize Index Performance.*

Creating indexes on a tablespace with a larger block size will increasing performance in DSS and in most OLTP scenarios.

The following 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.

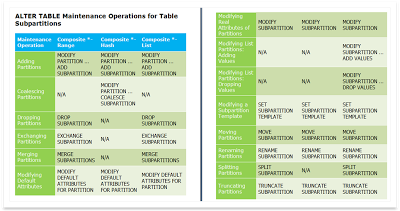
5. Partitioning- Related Data Dictionary Views

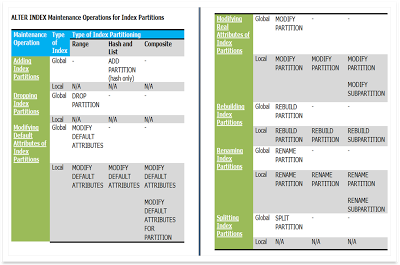
The following exhibit illustrates the list of data dictionary views that handle partitioning-related metadata.

[](http://3.bp.blogspot.com/_wWbZXrYfXrE/SjfFQS6M62I/AAAAAAAABOk/xxiUbWIx1ts/s1600-h/image005.png)

*Exhibit 3. Oracle Partitioning-related Data Dictionary Views*

6. Maintenance Operations of Partitions and Subpartitions

[](http://4.bp.blogspot.com/_wWbZXrYfXrE/SjfD4G6eDqI/AAAAAAAABOc/7UxNBpfzp2k/s1600-h/image007.png)*Exhibit 4. Maintenance Operations of Subpartitions*

[](http://2.bp.blogspot.com/_wWbZXrYfXrE/SjfCg3e9OII/AAAAAAAABOU/SU2c8xZDhSA/s1600-h/image009.png)*Exhibit 5. Maintenance Operations of Index Partitions.*

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

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.  
The following set of exhibits illustrates various scenarios of maintenance operations of partitions and subpartitions.

[http://1.bp.blogspot.com/_wWbZXrYfXrE/Sje8DgJJ5JI/AAAAAAAABOM/P_HF2AsdV7w/s400/image011.png](http://1.bp.blogspot.com/_wWbZXrYfXrE/Sje8DgJJ5JI/AAAAAAAABOM/P_HF2AsdV7w/s1600-h/image011.png)

*Exhibit 6. Adding a partition to a range partition table when MAXVALUE is not specified.*

[](http://1.bp.blogspot.com/_wWbZXrYfXrE/Sje6EM0XAaI/AAAAAAAABOE/91Ra0zkoN7Q/s1600-h/image013.png)*Exhibit 7. Coalescing Operation Reserved for the Hash Partitioning strategy.*

[](http://4.bp.blogspot.com/_wWbZXrYfXrE/Sje5wi88MsI/AAAAAAAABN8/ig2WtCRA1_o/s1600-h/image015.png)

*Exhibit 8. The Coalescing Maintenance Operation is Reserved to the Hash Partitioning Strategy.*

[](http://3.bp.blogspot.com/_wWbZXrYfXrE/Sje3xQ4grjI/AAAAAAAABN0/TGoI8XxFSis/s1600-h/image017.png)

*Exhibit 9. Truncating a Partitioning and Resetting the High Water Mark by Dropping the Storage for Exchange Preparedness.*

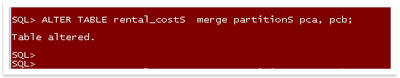
[http://1.bp.blogspot.com/_wWbZXrYfXrE/Sje3OgbB-QI/AAAAAAAABNs/B79kUJg6sfU/s400/image019.png](http://1.bp.blogspot.com/_wWbZXrYfXrE/Sje3OgbB-QI/AAAAAAAABNs/B79kUJg6sfU/s1600-h/image019.png)*Exhibit 10. Exchanging the Empty Partition with a Preloaded Table.*

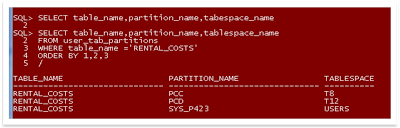
[http://1.bp.blogspot.com/_wWbZXrYfXrE/SjdjzHjzCOI/AAAAAAAABNk/8wamurUYYvg/s400/image021.png](http://1.bp.blogspot.com/_wWbZXrYfXrE/SjdjzHjzCOI/AAAAAAAABNk/8wamurUYYvg/s1600-h/image021.png)*Exhibit 11. Exchanging the Empty Partition with a Preloaded Table and Updating the Indexes (Rebuild Operation).*

[](http://2.bp.blogspot.com/_wWbZXrYfXrE/Sjdio5A5A_I/AAAAAAAABNc/ht-gM4dLXDk/s1600-h/image023.png)*Exhibit 12. Moving a Table Partition (When a Composite Partition is Being Used) is not Supported*

[](http://4.bp.blogspot.com/_wWbZXrYfXrE/SjdiPhGsbzI/AAAAAAAABNU/WC6rzD7Tf0M/s1600-h/image025.png)

*Exhibit 13. Moving a Table Partition when Supported.*

[](http://1.bp.blogspot.com/_wWbZXrYfXrE/SjdhywJMs6I/AAAAAAAABNI/nKYZrynCCk0/s1600-h/image027.png)*Exhibit 14. Moving two Table Partitions into a New System-Generated Partition.*

[](http://1.bp.blogspot.com/_wWbZXrYfXrE/SjdhP6Xs0tI/AAAAAAAABNA/1qN4LIG2ccI/s1600-h/image029.png)

*Exhibit 15. System-Generated Merged Partition Uses Default Tablespace.*

[http://2.bp.blogspot.com/_wWbZXrYfXrE/Sjdgsn1il-I/AAAAAAAABM4/L8JOK6Twuhc/s400/image031.png](http://2.bp.blogspot.com/_wWbZXrYfXrE/Sjdgsn1il-I/AAAAAAAABM4/L8JOK6Twuhc/s1600-h/image031.png)*Exhibit 16. Partitions PSW and PSE are Merged into Partition PSS in the Same Tablespace.*

The next exhibit shows the query and result set showing that the merged partition is set in the same tablespace as one of the merged partitions. In contrast, the previous example shows that the system-named partition was set in a default tablespace, since neither the target partition name nor the tablespace name was provided.

[](http://4.bp.blogspot.com/_wWbZXrYfXrE/SjdgMIK2bsI/AAAAAAAABMw/XJMO0FFonGQ/s1600-h/image033.png)

*Exhibit 17. Merged Partitions List for Table REGIONAL\_RENTALS.*

[](http://2.bp.blogspot.com/_wWbZXrYfXrE/Sjde3P3QecI/AAAAAAAABMo/nnMCwHvQ8Yw/s1600-h/image035.png)

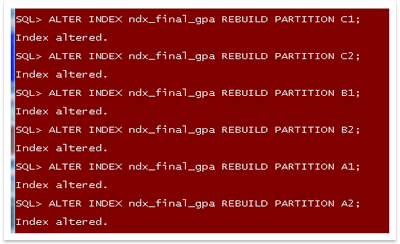
*Exhibit 18. Splitting a Range Partition with Date Datatype Partition Key.*

[](http://3.bp.blogspot.com/_wWbZXrYfXrE/SjdeTwUGtGI/AAAAAAAABMg/pg6h0aWZj5M/s1600-h/image037.png)

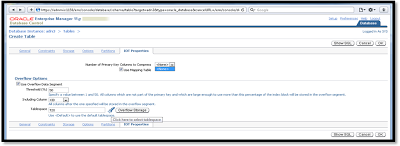
*Exhibit 19. Splitting a Range Partition with Date Partition Key.*



*Exhibit 20. Truncating a Table Partition with Various Options.*

[](http://3.bp.blogspot.com/_wWbZXrYfXrE/SjdbOaT8FMI/AAAAAAAABMQ/QqoCrCn8Gh4/s1600-h/image041.png)*Exhibit 21. Rebuilding a Global Index Partitions.*

7. Manageability  
  
DBAs can use Oracle Enterprise Manager Database and Grid Control to create, maintain, and verify accuracy of SQL, using the Schema tag and then selecting the desired partitioning options on the relevant object, namely, tables, indexes, and index-organized tables. There is extensive support to use standard, unstructured, and user-defined datatypes.

[](http://1.bp.blogspot.com/_wWbZXrYfXrE/SjdaVYiOhbI/AAAAAAAABMI/EDjckLb7C9Y/s1600-h/image043.png)*Exhibit 22. Manageability Applied to Partitioned Index-Organized Tables.*

[](http://3.bp.blogspot.com/_wWbZXrYfXrE/SjdZ_zZyFxI/AAAAAAAABMA/8ggw6uMP7WI/s1600-h/image045.png)

*Exhibit 23. Creating a Sample Partitioned Index-Organized Table using Database Control.*

8. Partitioning to Attain the Best Availability, Manageability and Performance

The following are valid partitioning options to improve availability, manageability and optimize performance, namely:

8.1 Partition Pruning

The following concerns regarding partition pruning can be highlighted:

* 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 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.

8.2 Partition-Wise Joins

The next paragraph asserts the most important issues regarding partition-wise joins, as follows:

* 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.
* 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.
* 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.
* 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.

8.3 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 the DIRECT\_MARKETING table and the CUSTOMERS table on cust\_id.

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;

Furthermore, the most commonly highlights are listed below:

* 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 interconnection, 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.

8.3.1 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.

8.3.2 Partial Partition-Wise Joins: Single-Level Partitioning

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.

8.3.3 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.

8.4 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.  
  
The three Oracle-supported Local Index partitioning types are:

8.4.1 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. Other important aspects of local partitioned indexes are as follows:

* Oracle constructs the local index so that it is equipartitioned 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.
* 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 equipartitioned 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

8.4.1.1 Local Prefixed Indexes

A local index is prefixed if it is partitioned on a left prefix of the index columns.

8.4.1.2 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.

8.4.2 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. It is also possible to highlight the following features:

* 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.
* Normally, a global index is not equipartitioned with the underlying table and usually 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.

The following are distinctive features differentiating global indexes on prefixing, namely:

* 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.loba

The most practical aspects of global partitioned indexes are emphasized in the following statements:

* 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. Thus, 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 normally a trade-off between performance and availability, and manageability.

8.5 Guidelines to Index Partitioning

The most important guidelines for OLTP and DSS are listed below respectively, as follows:

8.5.1 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.

8.5.2 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.

8.6 Tuning and Mixing objects in Multiple Block Size Database Models

The option to set index in larger block size tablespaces and caches has proven to greatly improve performance in Decision Support Systems (DSS), as introduced in the author’s paper “OMBDB, Oracle Multiple Block Size Databases: An Innovative Paradigm for Datawarehousing Architectures”, as verified by several field researchers and leading consultants. This option has also performed quite well in many Online Transaction Processing (OLTP) Systems.

9. Partitioning and Table Compression

When using partitioning and compression options together, it is possible to use compress in most cases, and the clause COMPRESS FOR ALL OPERATIONS can be specified at table creation time. Partitions can be individually set as compressed or not compressed, as needed.

Likewise, when using table compression on partitioned tables with bitmap indexes, the DBA needs 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.

10. Partition Strategy Recommendations

10.1 When to Use Range or Interval Partitioning

Range partitioning is a convenient method for partitioning historical data. Besides, other reasons include:

* 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.  
  
10.2 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. The next guidelines should be followed carefully:

* 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.
* Hash-partitioning is beneficial 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.

10.3 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.

10.4 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.

10.5 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. The following are relevant issues, namely:

* 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.

10.6 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.

10.7 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. Likewise, 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.

10.8 When to Use Composite List-Hash Partitioning

Composite list-hash partitioning is applicable to 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.

10.9 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.

10.10 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. In general, 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. Besides, 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.

10.11 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-rangehashlist partitioning, accordingly.

10.12 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.

10.13 When to Partition on Virtual Columns  
  
Virtual column partitioning enables partitioning on an expression, which may use data from other columns, and perform calculations with these columns.

The most relevant highlights are:

* 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.

11. Oracle Database Partitioning for ILM

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

11.1 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.

11.2 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.  
  
11.3 Low-Cost   
  
Low cost storage is a key factor in implementing ILM.  
  
11.4 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. In general, Enforceable Compliance Policies where Oracle Partitioning becomes mission-critical involve:

* Data Retention
* Immutability
* Privacy
* Auditing
* Expiration

12. 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:

12.1 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.

12.2 Performance

12.2.1 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.  
  
Besides, partition pruning greatly reduces the amount of data retrieved from disk and shortens processing time, thus improving query performance and optimizing resource utilization.

12.2.1.1 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.

12.2.1.2 Advanced Partition Pruning Techniques

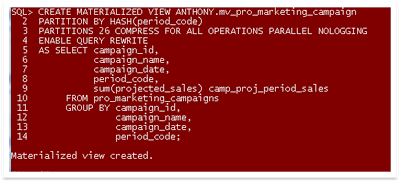
Oracle also prunes in the presence of more complex predicates or SQL statements involving partitioned tables. For instance, when a partitioned table is joined to the subset of another table, constrained by a WHERE clause condition.

12.3 Manageability

Manageability is greatly improved of the usage of Oracle Enterprise Manager Database and Grid Control and the usage of supplied packages.

13. 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. Similarly, 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.  
The next sample code illustrates how to effectively create a compressed materialized view partitioned by hash, which uses an aggregation on period\_code.

[](http://3.bp.blogspot.com/_wWbZXrYfXrE/SjdJOwHKidI/AAAAAAAABL4/dv7l51Ztpac/s1600-h/image047.png)*Exhibit 24. SQL DDL Creating a Partition Materialized View*

13.1 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.

13.2 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

Besides, to enable query rewrites, set the QUERY\_REWRITE\_INTEGRITY initialization parameter.

Likewise, in order to manually keep materialized views up to date, the init.ora parameter QUERY\_REWRITE\_INTEGRITY must be set to either TRUSTED or STALE\_TOLERATED. 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.

Besides, there is a need for a DBA to execute ALTER MATERIALIZED VIEW CONSIDER FRESH for every materialized view updated using this strategy.  
Likewise, partitioning also effectively addresses OLTP features and characterization 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

13.3 Storage Management

The following approaches can be used, namely:

13.3.1 High Availability: Implementing storage redundancy.

* Hardware-based mirroring
* Using ASM for mirroring
* Software-based mirroring not using ASM.

13.3.2 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

13.3.3 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.

13.3.4 Partition Placement

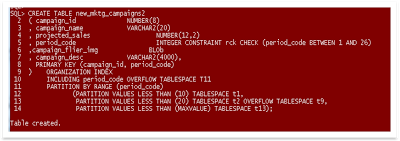
The following scenarios imply special considerations for partition placement, namely:

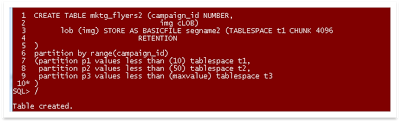
* Using Bigfile Tablespaces (due to one actual big file is used.)
* Customization (since other Oracle options are used in conjunction with it.)
* Oracle Exadata (a new VLDB Oracle database infrastructure with superfast performance capabilities. )

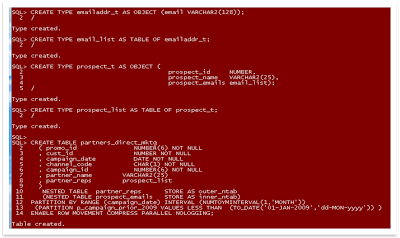
14. Partitioning Support for LOB

Oracle Partitioning support the storage of LOB types, such as BLOBs and BFILE types, and user-defined object datatypes.

The next exhibits illustrate some sample code involving large objects (LOBs) and user-defined object datatypes, as shown:

[](http://3.bp.blogspot.com/_wWbZXrYfXrE/SjdFZobr_LI/AAAAAAAABLo/hYSMXuuSbx8/s1600-h/image049.png)*Exhibit 25. One option for LOB support can be attained via Index Organized Tables, storing LOBs in a separate tablespace.*

*Exhibit 26. Partitioning Support for a CLOB Stored as Basicfile with Chunk and Retention set.*

[](http://4.bp.blogspot.com/_wWbZXrYfXrE/SjdDqkg8iMI/AAAAAAAABLY/aWi4ehbHQfQ/s1600-h/image053.png)*Exhibit 27. Partitioning Support for Object User-Defined Datatypes.*

[](http://1.bp.blogspot.com/_wWbZXrYfXrE/SjdDNmDZwFI/AAAAAAAABLQ/-BTPre1bLKI/s1600-h/image055.png)*Exhibit 28. Partitioning Support for Object User-Defined Datatypes using a primary key (see next 2 exhibits).*

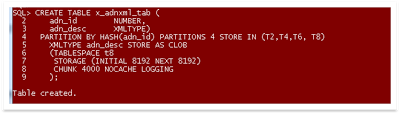
*Exhibit 29. Partitioning Support for Object User-Defined Datatypes using the object Identifier as a primary key.*

[http://1.bp.blogspot.com/_wWbZXrYfXrE/Sjc83uOll7I/AAAAAAAABK4/IMy87zgleX4/s400/image059.png](http://1.bp.blogspot.com/_wWbZXrYfXrE/Sjc83uOll7I/AAAAAAAABK4/IMy87zgleX4/s1600-h/image059.png)*Exhibit 30. Partitioning Support for Object User-Defined Datatypes using the object Identifier as a primary key.*

[](http://2.bp.blogspot.com/_wWbZXrYfXrE/Sjc6ldsXJEI/AAAAAAAABKw/SuugMvPgdYM/s1600-h/image061.png)

*Exhibit 31. Partitioning Support for Nested Tables using Object User-Defined Datatypes*

[](http://4.bp.blogspot.com/_wWbZXrYfXrE/Sjc6FR1RehI/AAAAAAAABKo/R8XfkwZwGmA/s1600-h/image063.png)*Exhibit 32. Partitioning Support for Varrays.*

[](http://4.bp.blogspot.com/_wWbZXrYfXrE/Sjc5cNgedDI/AAAAAAAABKg/pwGqI2Ihi_g/s1600-h/image065.png)*Exhibit 33. Partitioning Support for XML.*

15. Best Practices

In relation to best practices, it could be relevant to apply important considerations, tips and techniques, and specific constraints.

15.1 Special Considerations

* 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.
* 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 system’s platform, among others.

15.2 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.

15.3 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 storage constraints such as LOBs or large VARCHAR2 definitions.

16. Concluding Remarks

Upon completion of this resarch study, the author has arrived to the following remarkable conclusions based on his experience in the field:

* 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.
* Oracle partitioning can easily mix with other technologies, in particular, with multiple block size caches, which enable performance optimization by allowing optimal usage of I/O and memory resources with no contention.