proligence
Empowering Intelligence

Partitioning Demystified

by **Arup Nanda** Proligence, Inc.

This is a companion paper for the presentation on Advanced Partitioning Concepts delivered at *Oracle Technology Symposium* at Stamford, CT on March 27th and 28th, 2003 and is not intended to be an independent article and not guaranteed to be error free. A more current copy of this article may be found at www.proligence.com.

Topic Covered

- Subpartitioning Challenges
- Plan Table Revisited
- Dbms_Xplan
- Partition Pruning
- Partition-Wise Joins
- Character Value In Range
- Multi-Column Keys
- Conversion To Partitioned Table
- Partition Exchange
- Analyzing Subpartitions
- Parallel Index Rebuilding
- The Rule Based Optimizer
- Coalesce -Vs- Merge
- Rebuilding Indexes

Subpartitioning Challenges

The rows from a subpartition of a table are obtained the same way as in case of a partition, with a keyword after the table. For instance the to select rows from a subpartition SP1 of table TAB1, you would issue

```
SELECT … FROM TAB1 SUBPARTITION (SP1);
```

This same construct can be applied to DML statements and utilities like export/import and SQL*Loader, too. For instance to insert into a subpartition, you would issue

```
INSERT INTO TAB1 SUBPARTITION (SP1) …
```

In Export/Import, you would write

```
TABLE=TAB1:SP1
```

Information on storage of subpartitions can be obtained from the DBA_SEGMENTS view. The column, PARTITION_NAME is actually a misnomer; it refers to both partitions and subpartitions. When the view was first envisioned, the concept of subpartitions was not present and hence, the column was named such and never changed. In order to get information about a subpartition SP1 in table TAB1, use the query as follows.

```
SELECT COLUMNS
FROM USER_SEGMENTS
WHERE SEGMENT_NAME = 'TAB1'
AND PARTITION_NAME = 'SP1';
```

While creating subpartitions, you could use a subpartition template as follows to specify certain parameters for the subpartitions as follows.

```
PARTITION BY RANGE (COL1)
SUBPARTITION BY HASH (COL2)
SUBAPARTITION TEMPLATE
(
        SUBPARTITION SP1 TABLESPACE T1,
        SUBPARTITION SP2 TABLESPACE T2
)
(
        PARTITION P1 VALUES LESS THAN (101),
        PARTITION P2 VALUES LESS THAN (201),
…
```

In this example, you specified only partition names and a template for subpartition. The subpartitions are then named like P1_SP1, P1_SP2, P2_SP1, P2_SP2 and so on.

In 8i, only the tablespace names can be specified, nothing else. In 9i you could specify even storage parameters. The information on subpartitioning templates can be found in the data dictionary view DBA_SUBPARTITION_TEMPLATES.

Plan Table Revisited

You have been using plan_table to identify the optimizer plan of a statement. In this table, three columns are important for the partitioning option. The examples used in the document will use this table and more details on the columns will be provided.

PARTITION_START	The ID of beginning partition where the optimizer searches first.
PARTITION_STOP	The ID of the partition until which the optimizer searches for a match.

PARTITION_ID	The step id in plan_table that decided the partition start and stop
FILTER_PREDICATES	The exact condition used to evaluate partitions. (9i only)

The New Tool DBMS_XPLAN()

This is a new package useful for the querying the plan_table data. Instead of writing a complicated sql statement to query the plan_table, the call to the package returns the rows in a tabular format. To select the optimizer plan for the last explain plan statement, use the query.

```
SELECT * FROM TABLE(DBMS_XPLAN.DISPLAY())
```

Note the use of the `CAST TABLE()`. The function `DBMS_XPLAN.DISPLAY` is a *pipelined* function, i.e. it returns rows in a tabular format like a cursor. The `CAST TABLE()` makes it behave just like a table so that it can be queried as one. This returns the results as the formatted query from plan_table.

```
PLAN_TABLE_OUTPUT
---------------------------------------------

-------------------------------------------
| ID  | OPERATION             |  NAME       |
-------------------------------------------
|   0 | SELECT STATEMENT      |             |
|   1 |  SORT AGGREGATE       |             |
|   2 |   NESTED LOOPS        |             |
|   3 |    TABLE ACCESS FULL  | PTEST3HA    |
|   4 |    TABLE ACCESS FULL  | PTEST3HB    |
-------------------------------------------
```

The package has only this function, `display()` which takes three arguments

TABLE_NAME	The name of the table where the optimization plan is stored; defaults to PLAN_TABLE.
STATEMENT_ID	The statement id from the plan table mentioned earlier. By default, it takes the last one, or NULL.
FORMAT	This controls the way the display is formatted. Explained later in detail .

Let's examine the last option FORMAT in detail. It accepts four values as follows. The first value `BASIC` provides only the minimum amount of information, as in case of the example above. The value `TYPICAL`, which is the default, provides a variety of the information useful for understanding how the optimizer works. For instance, in case of partitioned table operation, the columns `PARTITION_START` and `PARTITION_STOP` are also displayed, in addition to `COST` for that step, the number of row expected to be retrieved and number of bytes those rows may have. A setting of `ALL` displays all the information that TYPICAL does but in addition also explains the sql statements generate for parallel queries, too. The last value `SERIAL` is similar to TYPICAL setting, but the queries are explained in serial even if parallel query will be used.

Partition Pruning

The main advantage of partitioning comes when the optimizer chooses the data in a specific partition only, where the requested data will be found and not all the partitions. For instance, consider a table SALES partitioned on `ORDER_DATE`, with one partition per quarter. When the following query is issued

```
SELECT … FROM SALES
WHERE ORDER_DATE = '1/1/2003'
```

the optimizer does not go through the entire table, but only the partition that houses the rows for the order date, which is 2003 Quarter 1. This way, the full table scans are limited to a specific partition only, saving significant IO. But how can you can make sure that the partition pruning has occurred? You can do so by querying the `plan_table`. Consider a table created as follows

```
CREATE TABLE PTEST1
(
       COL1 NUMBER,
       COL2 VARCHAR2(200),
       COL3 VARCHAR2(200)
)
PARTITION BY RANGE (COL1)
(
PARTITION P1 VALUES LESS THAN (1001),
PARTITION P2 VALUES LESS THAN (2001),
… and so on
PARTITION P9 VALUES LESS THAN (9001),
PARTITION PM VALUES LESS THAN (MAXVALUE)
);
```

Now we inserted several records into this table so that each partition will have at least one record and then analyzed the table.

```
INSERT INTO PTEST1
SELECT ROWNUM, OBJECT_TYPE, OBJECT_NAME
FROM ALL_OBJECTS
WHERE ROWNUM < 10001;
COMMIT;
```

Then we will examine the optimization plan for a query that will be issued on the table PTEST1 as follows.

```
EXPLAIN PLAN FOR
SELECT * FROM PTEST1
WHERE COL1 = 1500;
```

This populates the PLAN_TABLE with the optimization plan records. Now select the plan using the query

```
SELECT ID, LPAD(' ',LEVEL*1-1)||OPERATION||'
'||OPTIONS||' ON '||OBJECT_NAME OPERATION,
       PARTITION_START, PARTITION_STOP,
       PARTITION_ID, FILTER_PREDICATES
FROM PLAN_TABLE
CONNECT BY PARENT_ID = PRIOR ID
START WITH PARENT_ID IS NULL;
```

The result is as follows

```
ID OPERATION                          PB PE  PI
FILTER_PREDICATES
--- ------------------------------ -- -- --- -
-------------------
  0 SELECT STATEMENT  ON
  1  TABLE ACCESS FULL ON PTEST1    2  2    1
"PTEST1"."COL1"=1500
```

Note: This could have been done via dbms_xplan, too; but to make it version independent, we will stick to plan_table. Look at the PARTITION_START and PARTITION_STOP columns; the values are 2 each - indicating that the data will be selected from partition# 2 only. This is correct, since the value 1500 will be available in partition 2 only. How does the optimizer know which partition to look for? It does so at Step 1, as indicated by the column PARTITION_ID in PLAN_TABLE. Finally, we also know that the optimizer applied a filter to retrieve rows as in the column FILTER_PREDICATES. This explains how the optimizer came up with the plan and which segments it will select from. While testing the different partition pruning scenarios, this will be most helpful.

Let's introduce another complexity to the mix - subpartitioning. Consider a table created as follows.

```
CREATE TABLE PTEST2
(
       COL1 NUMBER,
       COL2 VARCHAR2(200),
       COL3 VARCHAR2(200)
)
PARTITION BY RANGE (COL1)
SUBPARTITION BY HASH (COL2)
SUBPARTITIONS 4
(
PARTITION P1 VALUES LESS THAN (1001),
PARTITION P2 VALUES LESS THAN (2001),
and so on…
PARTITION P9 VALUES LESS THAN (9001),
PARTITION PM VALUES LESS THAN (MAXVALUE)
);
```

We will insert rows in the same manner as the example earlier and analyze the table. Then we will issue the query as follows.

```
EXPLAIN PLAN
SET STATEMENT_ID = 'PTEST2'
FOR
SELECT COL2 FROM PTEST2 WHERE COL1 = 9500
AND COL2 = 'PROCEDURE';
```

Here the query is forced to select from a subpartition, as the filter is based on the partitioning as well as the subpartitioning key. The query on PLAN_TABLE as earlier shows the following output.

```
  ID OPERATION                           PB PE  PI
FILTER_PREDICATES
---- ----------------------------- -- -- ---
-------------------------------------------
   0 SELECT STATEMENT  ON
   1  TABLE ACCESS FULL ON PTEST2  38 38   1
"PTEST2"."COL1"=9500 AND "PTEST2"."COL2"
='PROCEDURE'
```

Note the PARTITION_START column; it shows 38. But we don't have that many partitions. Actually, it's the count of *subpartitions*, not partitions. Since the number of subpartitions in a partition is 4, the first 9 partitions contain the first 36 subpartitions, making the 38th subpartition the 2nd one in the 10th partition, i.e. partition PM.

Partition-wise Joins

When a partitioned table is joined to another partitioned table in such a way that partitioning keys determine the filtering, the optimizer can determine that it does not need to search the whole table, but just the partitions where the data resides. For instance, consider the tables SALES range partitioned on SALES_DATE column and table REVENUE range partitioned on BOOKED_DATE column and the partitioning schemes are the same. If the user queries using the following

```
SELECT … FROM SALES S, REVENUE R
WHERE S.SALES_DATE = R.BOOKED_DATE
```

The optimizer knows that for each row in SALES, only rows in a particular partition in REVENUE need to be searched, not all. The important thing is to identify if such elimination is indeed happening. Consider two tables created as follows.

```
CREATE TABLE PTEST3A
(
       COL1A NUMBER,
       COL2A VARCHAR2(200),
       COL3A VARCHAR2(200)
)
```

```
PARTITION BY RANGE (COL1A)
(
PARTITION P1 VALUES LESS THAN (1001),
PARTITION P2 VALUES LESS THAN (2001),
```
and so on…
```
PARTITION P9 VALUES LESS THAN (9001),
PARTITION PM VALUES LESS THAN (MAXVALUE)
);

CREATE TABLE PTEST3B
(
       COL1B NUMBER,
       COL2B VARCHAR2(200),
       COL3B VARCHAR2(200)
)
PARTITION BY RANGE (COL1B)
(
PARTITION P1 VALUES LESS THAN (1001),
PARTITION P2 VALUES LESS THAN (2001),
```
and so on…
```
PARTITION P9 VALUES LESS THAN (9001),
PARTITION PM VALUES LESS THAN (MAXVALUE)
);
```

We will insert data into both tables as follows.

```
INSERT INTO PTEST3A
SELECT ROWNUM, OBJECT_TYPE, OBJECT_NAME
FROM ALL_OBJECTS
WHERE ROWNUM < 10001;

INSERT INTO PTEST3B
SELECT ROWNUM, OBJECT_TYPE, OBJECT_NAME
FROM ALL_OBJECTS
WHERE ROWNUM < 10001;
```

If a user queries the tables in this manner

```
EXPLAIN PLAN
SET STATEMENT_ID = 'PTEST3' FOR
SELECT COUNT(*)
FROM PTEST3A , PTEST3B
WHERE PTEST3B.COL1B = PTEST3A.COL1A
AND PTEST3A.COL1A BETWEEN 1500 AND 1700;
```

and then selects from the plan_table, she gets

```
  ID OPERATION                        PB PE  PI
---- ------------------------------ -- -- ---
FILTER_PREDICATES
--------------------
   0 SELECT STATEMENT  ON
   1  SORT AGGREGATE ON
   2   NESTED LOOPS  ON
   3    TABLE ACCESS FULL ON PTEST3A  2 2   3
 "PTEST3A"."COL1A">=1500 AND
"PTEST3A"."COL1A"<=1700
   4    TABLE ACCESS FULL ON PTEST3B 2  2   4
"PTEST3B"."COL1B"="PTEST3A"."COL1A" AND
"PTEST3B"."COL1B">=1500 AND
"PTEST3B"."COL1B"<=1700
```

Note how only partitions 2 from each table were subjected to Full Table Scans, not the entire table. The optimizer determined these steps from the filter predicates, easily explained in the output. This explains how partition pruning has occurred.

Warning: A caveat has to be introduced for the hash-partitioned tables. Consider the following two tables.

```
CREATE TABLE PTEST3HA
(
       COL1A NUMBER,
       COL2A VARCHAR2(200),
       COL3A VARCHAR2(200)
)
PARTITION BY HASH (COL1A)
PARTITIONS 4;

CREATE TABLE PTEST3HB
(
       COL1B NUMBER,
       COL2B VARCHAR2(200),
       COL3B VARCHAR2(200)
)
PARTITION BY HASH (COL1B)
PARTITIONS 4;
```

Insert the data in the same way as before and analyze. If we explain the same query as we did before, and select from the plan table we get

```
  ID OPERATION                        PB PE PI
---- ------------------------------- -- -- --
FILTER_PREDICATES
--------------------
   0 SELECT STATEMENT  on
   1  SORT AGGREGATE on
   2   PARTITION HASH ALL on           1  4  2
   3    NESTED LOOPS  on
   4     TABLE ACCESS FULL on PTEST3HA 1  4  2
"PTEST3HA"."COL1A">=1500 AND
"PTEST3HA"."COL1A"<=1700
   5     TABLE ACCESS FULL on PTEST3HB 1  4  2
"PTEST3HB"."COL1B"="PTEST3HA"."COL1A" AND
"PTEST3HB"."COL1B" >=1500 AND
"PTEST3HB"."COL1B"<=1700
```

Note the partition start and stop values, which are for *all the* partitions. This query does *not* perform a partition wise join; it simply scans the entire table, even though it could have eliminated certain partitions. The filter predicates indicate that the optimizer knew about the rows to look for. So why it didn't do a partition wise join?

The problem is the way hash partitioned table handles joins. If the filter predicates are based on *equality operator* only, then the optimizer can assign a specific partition to the predicate by using the hash function. But if the predicate is a range, the optimizer cannot decide whether a particular partition may be a candidate. If the same query explained earlier was written using *COL1 = some value*, rather than *COL1 BETWEEN two values*, then the partition-wise joins would have kicked in. Therefore, be particularly careful in designing hash-partitioned tables when there is a chance of joining with range filtering.

Character Value in Range Partitioning

Almost all documents, articles, books and other documentation talks about range partitioning using either dates (the most common) or numbers. However, the partitioning scheme could be extended to character strings too. Consider the example of the employee table where the last name column has been made a partitioning key. Here is the proper syntax for designing such a table.

```
CREATE TABLE EMP (………..)
PARTITION BY RANGE (LAST_NAME)
(
PARTITION P1 VALUES LESS THAN ('D%'),
PARTITION P2 VALUES LESS THAN ('G%'),
PARTITION PM VALUES LESS THAN (MAXVALUE)
);
```

Note the percentage character after the names. This ensures that the ranges are well demarked by the boundaries. This is as per a Note in MetaLink. Consider this example of the table EMP described above.

```
SELECT * FROM EMP;

LAST_NAME  FIRST_NAME
---------- ----------
CHAPLIN    CHARLIE
D          HARLEY
DAVIDSON   HARLEY
EINSTEIN   ALBERT

SELECT * FROM EMP PARTITION (P1);

LAST_NAME  FIRST_NAME
---------- ----------
CHAPLIN    CHARLIE
D          HARLEY

SELECT * FROM EMP PARTITION (P2);

LAST_NAME  FIRST_NAME
---------- ----------
DAVIDSON   HARLEY
EINSTEIN   ALBERT
```

Note the placement of two rows with last names starting with `D`. The last name `DAVIDSON` is correctly placed in P2, but the last name `D` is placed in P1. This is the expected behavior, even though it does not seem like. In the character set comparison, 'D' is less than 'D%', satisfying the boundary of the partition P1. You might have expected the last name D to go into the same partition as DAVIDSON. Therefore, be careful while using the scheme for partitioning.

Consider the same table in a slightly different way.

```
CREATE TABLE EMP (………..)
PARTITION BY RANGE (LAST_NAME)
(
PARTITION P1 VALUES LESS THAN ('D'),
PARTITION P2 VALUES LESS THAN ('G'),
PARTITION PM VALUES LESS THAN (MAXVALUE)
);
```

Note, there is no percentage sign after the character values. Inserting the same data into it and selecting from different partitions, we get

```
SELECT * FROM EMP2 PARTITION (P2);

LAST_NAME  FIRST_NAME
---------- ----------
DAVIDSON   HARLEY
EINSTEIN   ALBERT
D          HARLEY
```

Note how the partition P2 now has both DAVIDSON and D, perhaps that will avoid potential problems in the future. If you design character based range partitioning, you must consider the use of the percentage character in your boundary to eliminate confusion. If you use Oracle 9i, you can probably change most of your character based partitioning schemes to LIST, a new option in that version.

Multi-Column Partition Keys

Many people are under the impression that by specifying more than one column as partitioning key creates a multi-dimensional partitioned table. For example if you have a table employee range partitioned on (DEPTNO, ZIPCODE), does that make a two dimensional partitioning scheme? Unfortunately, multi-column partitioning keys are not intended for that objective at all.

The second column is for decision along the sequential path only. Both values do not need to be satisfied for an insert to go to a specific partition. The first column is evaluated first; if it satisfies the condition, then the second column is *not* even evaluated. Only if first column

value is right on the boundary, the next column is considered.

This is perhaps better explained using an example. Consider the following example.

```
CREATE TABLE PTAB1
(
       COL1 NUMBER(10),
       COL2 NUMBER(10),
       COL3 VARCHAR2(20)
)
PARTITION BY RANGE (COL1, COL2)
(
PARTITION P1 VALUES LESS THAN (101, 101),
PARTITION P2 VALUES LESS THAN (201, 201)
)

SELECT * FROM PTAB1;

      COL1       COL2 COL3
---------- ---------- --------------------
        50         50 REC1
        50        150 REC3
       150        150 REC2
       150         50 REC4
       201         50 REC5
       201        101 REC6
```

Which partitions do you think the records will be?

```
SELECT * FROM PTAB1 PARTITION (P1);

      COL1       COL2 COL3
---------- ---------- --------------------
        50         50 REC1
        50        150 REC3
```

Record REC1 is in partition P1 as expected. But should REC3 be on partition P1? The column COL1 is satisfied, but COL2 is not. How does it end up in the partition P1? The reason being, P1 is the first partition, it's evaluated for the first column, col1, the value satisfies it and so col2 is *not even evaluated*. The record goes to P1, *even though* the second column col2 is not satisfied.

So, if the second column, COL2 is not even considered at all in some cases, where does it come into play and why would you define it? Consider the following example.

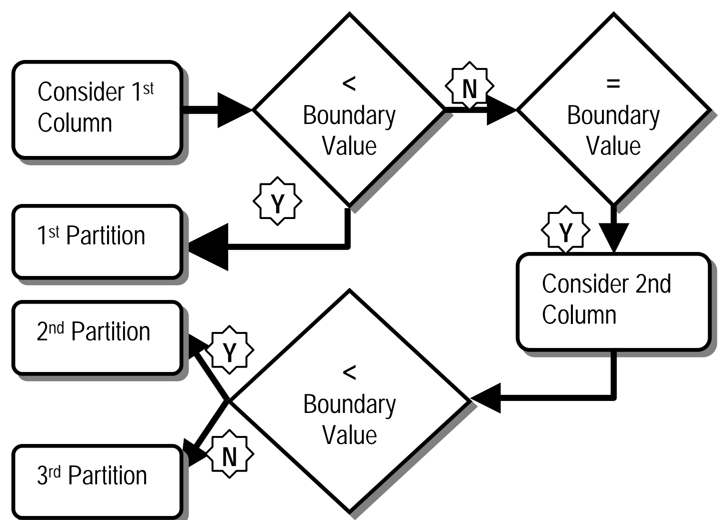
```
SELECT * FROM PTAB1 PARTITION (P2);

      COL1       COL2 COL3
---------- ---------- --------------------
       150        150 REC2
       150         50 REC4
       201         50 REC5
       201        101 REC6
```

The record REC2 satisfies both columns and is as expected. In the record REC4, COL1 value is 150, so falls under partition P2 as explained above. However, for REC5 and REC6 - the COL1 value is 201, which is the boundary value for first column of the partitioning key. Only in that case, i.e. where the value of the first column of partitioning keys is equal to the boundary value, the second column comes into picture. The column COL2 values are less than the maximum value of column COL2 of partition P2. Therefore, the rows went there.

What happens when you insert a row with COL1 = 201 and COL2 = 201?

That insert will fail, since both columns cannot be outside the bounds. Schematically, the decision to insert into a partition can be explained as in Figure below.



With this information now, you can intelligently decide on the partitioning columns. Consider another table.

```
CREATE TABLE PTAB2
(
       COL1 NUMBER(10),
       COL2 NUMBER(10),
       COL3 VARCHAR2(20)
)
PARTITION BY RANGE (COL1, COL2)
(
PARTITION P1 VALUES LESS THAN (101, 51),
PARTITION P2 VALUES LESS THAN (201, 51),
PARTITION P3 VALUES LESS THAN (201, 101),
PARTITION P4 VALUES LESS THAN (201, 201),
PARTITION P5 VALUES LESS THAN (201, 301)
)

SELECT * FROM PTAB2;

      COL1       COL2 COL3
---------- ---------- --------------------
        50         50 REC1
```

```
 50        150 REC3
150        150 REC2
150         50 REC4
201         50 REC5
201        150 REC6
201        250 REC7
```

Guess the partitions the rows go into.

```
SELECT * FROM PTAB2 PARTITION (P1);

      COL1       COL2 COL3
---------- ---------- --------------------
        50         50 REC1
        50        150 REC3
```

This is as expected as explained above.

```
SELECT * FROM PTAB2 PARTITION (P2);

      COL1       COL2 COL3
---------- ---------- --------------------
       150        150 REC2
       150         50 REC4
       201         50 REC5
```

Records, REC2 and REC4 are expected as per the explanation above. But, what about REC5? By now you must have figured out the rationale of record rec5 going into partition P2.

```
SELECT * FROM PTAB2 PARTITION (P4);

      COL1       COL2 COL3
---------- ---------- --------------------
       201        150 REC6

SELECT * FROM PTAB2 PARTITION (P5);

      COL1       COL2 COL3
---------- ---------- --------------------
       201        250 REC7
```

The logical follow up to this discussion is what happens in case list partitioning where there is no concept of a range, so there is no boundary value. Fortunately, list partitioning does not allow multiple columns; so this situation does not arise.

Conversion to Partitioned Table

As you explore more and more into partitioning and understand its advantages, you are more likely than not to introduce partitioning in your existing database. That comes with the inevitable question - what is the best way to convert non-partitioned tables to partitioned ones. Oracle does document such a process in MetaLink Note Id 1070693.6. In summary, the note specifies creating an empty partitioned table similar to the source table and then loading data into the target using direct insert; or just creating the partitioned table as selecting from the non-partitioned table using the NOLOGGING option. These methods work. However, the biggest problem in these is the space requirement. You must have exactly the same amount of free space as your biggest table you are going to partition. Since these methods have been discussed in detail in the Note, they are not being reproduced here.

In Oracle 9i, the online table redefinition feature using DBMS_REDEFINITION package can be used to redefine the table as partitioned without any downtime. While it provides online access to the table while being converted, the time, resource and space consumption is extremely high.

The alternative approach - called *split-split method* greatly alleviates the space problem. Essentially, in this method, you would create the partitioned table with only one partition first, split it at the lowest boundary point and repeat the process until all the partitions are created. This is best described using an example. Consider the table NOPART as follows.

```
COL1     NUMBER
COL2     VARCHAR2(10)
COL3     CHAR(2)
```

This table has one index IN_NOPART on column COL2 and one constraint CK_PART, a check constraint stating that COL3 is not null. It needs to be partitioned into 5 partitions like this.

```
PARTITION BY RANGE (COL1)
(
       PARTITION P1 VALUES LESS THAN (101),
       PARTITION P2 VALUES LESS THAN (201),
       PARTITION P3 VALUES LESS THAN (301),
       PARTITION P4 VALUES LESS THAN (401),
       PARTITION PM VALUES LESS THAN
(MAXVALUE)
)
```

Let's convert this table using the s*plit-split method*. First, create a table PART as follows

```
CREATE TABLE PART
(
        COL1     NUMBER,
        COL2     VARCHAR2(10),
        COL3     CHAR(2)
)
NOLOGGING
PARTITION BY RANGE (COL1)
(
       PARTITION PM VALUES LESS THAN
(MAXVALUE)
);
```

Note, only the maximum value partition has been defined, not the entire set. We will also define the indexes and constraints as seen in in the table NOPART.

```
CREATE INDEX IN_PART ON PART (COL2) LOCAL
NOLOGGING;
```

```
ALTER TABLE PART ADD CONSTRAINT CK_PART_01
CHECK (COL3 IS NOT NULL);
```

Next, we exchange the table NOPART with this partition

```
ALTER TABLE PART EXCHANGE PARTITION PM
WITH TABLE NOPART INCLUDING INDEXES;
```

This statement swaps the table's partition PM with the table NOPART. The contents of NOPART are now in the PM partition and the NOPART table is empty. Since this operation merely changes the data dictionary and doesn't physically move data, it doesn't generate redo and is extremely quick. The clause INCLUDING INDEXES swaps the indexes too.

Next, we will split this single partition, starting with the lowest boundary, i.e. partition P1.

```
ALTER TABLE PART SPLIT PARTITION PM AT (101)
INTO (PARTITION P1, PARTITION PM);
```

This creates a new partition called P1 and moves the rows with COL1 value less than 101 into this from PM. Since the table is defined as NOLOGGING, this doesn't generate much redo. After this operation, the partition PM contains data for the partitions other than P1. Repeat this splitting process, with P2 in mind this time.

```
ALTER TABLE PART SPLIT PARTITION PM AT (201)
INTO (PARTITION P2, PARTITION PM);
```

This process is repeated until the partition PM is split up to P4, the last but one partition. Since the index is defined as LOCAL, it will have been split, too along with the table partition splitting command.

At the end of the process, we will have a table called PART with all the data from NOPART and with the same indexes and constraints. Now drop the table NOPART and rename the table PART to NOPART so that applications will be able to access this table. Also restore the privileges associated with NOPART to PART.

However, renaming table does not rename the constraints or indexes. Although applications may not be affected by the new name of the index and constraints, it may be necessary to change the names to avoid confusion. The names are changed by the statements

```
ALTER INDEX IN_PART RENAME TO IN_NOPART;
```

```
ALTER TABLE NOPART RENAME CONSTRAINT
CK_PART_01 TO CK_NOPART_01;
```

The latter command is available in 9i only. If you are in 8i and cannot do this, you could always drop the constraint from the NOPART table first and then create the constraint with NOVALIDATE option.

The advantage of this method is clear - minimum space requirement. However, the partition split operation is still time and resource consuming as compared to direct load insert method. At your site, you should evaluate the pros and cons of each and decide on the best method fit for your purpose. This paper simply presents another alternative.

Partition Exchange

The concept of exchanging a partition of a table with a non-partitioned table is quite simple and fundamental in partitioning. However, is it possible to exchange a partition of a table with another *partitioned* table?

If the main table is composite partitioned and the sub partitioning scheme is exactly same as the partitioning scheme of the source table, then it is possible. For instance, consider the following partitioning scheme for the table MAIN

```
PARTITION BY RANGE (COL1)
SUBPARTITION BY HASH (COL2)
SUBPARTITIONS 2
(
PARTITION P1 VALUES LESS THAN (101)
(
       SUBPARTITION SP1,
       SUBPARTITION SP2
),
….
```

Now consider a table named CHILD partitioned as

```
PARTITION BY HASH (COL1)
PARTITIONS 2
(
       PARTITION P1,
       PARTITION P2
)
```

You can exchange the partition P1 of the table MAIN with the table CHILD since the partitioning schemes of the partition within the table and the table CHILD are identical.

Subpartition Statistics

This is one tricky part of subpartitioning not very well documented and clear in the manuals. You must be using DBMS_STATS package for quite some time now to collect statistics. To collect statistics for the tables and the sub-objects under them, e.g. partitions and subpartitions, the function under the package named GATHER_TABLE_STATS is used. The function has two little known parameters that needs to be set for proper statistics collection.

PARTNAME
This parameter is set to collect statistics for only the named partition within the table, not for the entire table. However, this is a misnomer. This can be used to collect the stats for a specific subpartition, too, if named here.

GRANULARITY
This parameter instructs the package to collect statistics at different levels and cascade down to other sub-objects. It accepts several values. The default, named DEFAULT, instructs the package to collect global statistics and on the partitions only. The value PARTITION instructs the package to collect stats at the partition level. However, setting these values will not collect stats at the *subpartition* level, which can be collected by setting the parameter to ALL or SUBPARTITON.

For instance, consider the table created as follows.

```
CREATE TABLE SPART1
(
   COL1    NUMBER,
   COL2    NUMBER,
   COL3    VARCHAR2(20)
)
PARTITION BY RANGE (COL1)
SUBPARTITION BY HASH (COL2)
SUBPARTITIONS 4
(
   PARTITION P1 VALUES LESS THAN (101),
   PARTITION P2 VALUES LESS THAN (201),
   PARTITION P3 VALUES LESS THAN (301),
   PARTITION P4 VALUES LESS THAN (401),
   PARTITION PM VALUES LESS THAN (MAXVALUE)
)
```

Analyze the table using the default value of granularity as follows.

```
EXEC DBMS_STATS.GATHER_TABLE_STATS
(TABNAME=>'SPART1')
```

Note, we have not provided the granularity at all; since the default value is to collect stats for the partitions only and not for any of the subpartitions, the stats will not be collected for the subpartitions. This can be verified by issuing

```
SELECT PARTITION_NAME
FROM USER_TAB_SUBPARTITIONS
WHERE LAST_ANALYZED IS NOT NULL;
```

This will not return any rows. While we are on the subject, let's analyze the other options here. A table can have statistics at the table level only, called GLOBAL statistics. If the partitions of the table are analyzed and the optimizer can derive the global statstics from the individual partitions, then the stats for the table are supposed to be derived global. Let's examine each option in detail.

```
EXEC DBMS_STATS.GATHER_TABLE_STATS
(TABNAME=>'SPART1', GRANULARITY=>'GLOBAL')
```

This collects stats at the global level only. The following query confirms this.

```
SELECT LAST_ANALYZED, GLOBAL_STATS
FROM USER_TABLES WHERE TABLE_NAME = 'SPART1';
```

This returns

```
GLO LAST_ANAL
--- ---------
YES 10-MAR-03
```

The presence of global stats indicates that the table has been analyzed as a whole but the optimizer will not know the stats of individual partitions. This can be gathered by

```
EXEC DBMS_STATS.GATHER_TABLE_STATS
(TABNAME=>'SPART1', GRANULARITY=>'PARTITION')
```

This sets the stats at the partition level only. In this case the global stats are not collected on the table and the above query will say NO under GLOBAL_STATS. However, the query

```
SELECT PARTITION_NAME, LAST_ANALYZED
FROM USER_TAB_PARTITIONS
WHERE LAST_ANALYZED IS NOT NULL;
```

will retrieve all the partitions. Another variation of the package is shown below.

```
EXEC DBMS_STATS.GATHER_TABLE_STATS
(TABNAME=>'SPART1',
GRANULARITY=>'SUBPARTITION')
```

This collects stats on the subpartition level only and rolls it up to infer the stats on the partition level; but it does not collect global stats on the partitions itself.

The last value of the option, ALL, does all of these - collects partition level, subpartition level stats as well as the global stats on the subpartition, partition and table.

Therefore, the default value of GRANULARITY parameter in the stats gathering function does not collect stats on subpartitions; you must set it to either SUBPARTITION or ALL to gather stats.

In summary, here are the details on the setting of granularity and the collection of statistics.

GRANULARITY	Table Global	Partition Global	Partition Statistics	Subpartition Statistics
GLOBAL	YES	NO	NO	NO
PARTITION	NO	YES	YES	NO
DEFAULT	YES	YES	YES	NO
SUBPARTITION	NO	NO	YES	YES
ALL	YES	YES	YES	YES

Another interesting concept but not documented clearly is the option to analyze subpartitions only. This can be done by

```
EXEC DBMS_STATS.GATHER_TABLE_STATS
(TABNAME=>'SPART1', PART_NAME=>'P1_SYS123')
```

This will collect subpartition level stats on subpartition P1_SYS123 only.

Parallel Index Rebuilding

When an index needs to be reorganized, or rebuilt, it can be done by issuing a ALTER INDEX … REBUILD. But if an index is partitioned, this statement cannot be used. Rather the commands must be issued for each partition (or subpartition) of the index. For instance, you would issue

```
ALTER INDEX IN_PART REBUILD PARTITION P1;
```

One of the advantages of partitioning is the use of parallel DML on the partitioned table. In this case, a single query on the table actually spins off several parallel query slaves and each of the slaves acts on a separate partition, increasing the throughput of the process. In several cases, more than one slave process acts on each partition, thereby increasing the throughput even more. However, while rebuilding a partitioned index, parallel slaves cannot work with more than one slave per partition. Even though the host could support it, only one parallel query slave act on each partition of the index to rebuild it, effectively limiting the throughput. To circumvent this limitation, Oracle provides a package called DBMS_PCLXUTIL that contains a procedure called BUILD_PART_INDEX(). This procedure, when called, spins off several dbms jobs and each job can work on the partitions in parallel.

For instance, to rebuild the index IN_PART on partitioned table PART with 4 partitions, we can call

```
EXEC DBMS_PCLXUTIL.BUILD_PART_INDEX
(TAB_NAME=>'PART', IDX_NAME=>'IN_PART',
JOBS_PER_BATCH=>2, PROCS_PER_JOB=>4)
```

This will kick off 2 jobs. Be sure to call SET SERVEROUTPUT ON SIZE 99999 before this execute call. It will display all the messages on the screen as follows.

```
INFO: JOB #1 CREATED FOR PARTITION P1 WITH 4
SLAVES
INFO: JOB #2 CREATED FOR PARTITION P3 WITH 4
SLAVES
```

Each job will kick off 4 slaves as per the PROC_PER_JOB parameter, effectively employing 8 slaves. Normally, a parallel DDL would have employed only 4 slaves (for 4 partitions).

However, this will not work for the index partitions marked USABLE. To force the rebuilding of those partitions, use the parameter FORCE_OPT and set it to TRUE.

The biggest problem with this approach is the job interface. Since the processes are actually jobs, the package simply kicks off the jobs and returns the control to the user. If the jobs fail for some reason, the user may not even know about it. Check for the status of the jobs in USER_JOBS and see if they failed and produced any trace files.

Rule Based Optimizer

Can you use partitioning with RBO? The answer is, of course you can. However, when partitioning was introduced, RBO was considered legacy and Oracle decided to gradually desupport it. This led to a general stop in development of RBO and thus today it is blissfully unaware of the several exciting developments, partitioning included. Therefore, to exploit the full advantage of partitioning, like partition pruning, partition-wise joins, etc., you must use the Cost Based Optimizer (CBO). If you use the RBO, and a table in the query is partitioned, Oracle kicks in the CBO while optimizing it. But the statistics are not present; so the CBO *makes up* the statistics and this could lead to severely expensive optimization plans and extremely poor performance.

So, although you can, you shouldn't use partitioning when using the RBO.

Coalesce –vs- Merge

These two potentially confusing statements serve the same purpose - reducing the number of partitions - are applicable in different schemes. In a range or list partitioned table, the partition boundaries are clearly defined and the rows in a partition satisfy some condition

dependent on the boundary values. `ALTER TABLE … MERGE PARTITION` joins the two adjacent partitions and sets the boundary values appropriately.

Consider the example of a table `PART` that is partitioned by range to four different partitions named P1, P2, P3 and P4. To merge partitions P3 and P4 to make a partition called P34, issue the statement

```
ALTER TABLE PART MERGE PARTITIONS P3, P4 INTO
PARTITION P34;
```

However, in hash-partitioned tables, there are no boundary values and the rows are not decided as candidates for the partitions based on some kind of defined range. So, a merge will not be able to identify and set specific boundaries. Rather, a new clause called COALESCE is used to achieve the objective.

```
ALTER TABLE PART COALESCE;
```

In COALESCE, a specific partition, usually the last one, is identified for elimination. All the rows in that partition are supposed to be equally distributed over the remaining partitions and the partition is dropped. In practice, however, the rows are merged with the adjacent partition.

Since this reduces the number of partition by 1, the total number is not a power of 2 any more, making the distribution of rows in all partitions unequal. To avoid this problem issue the coalesce one more time to make the partitions evenly loaded.

In summary, `MERGE` is for range and list partitioning where the values are clearly identified for boundary values and `COALESCE` is for hash partitions, to reduce the number of partitions.

Rebuild Partition and Global Indexes

Oracle9iR2 now offers fast split partitioning. Typically during a split operation, Oracle creates two new partitions and then redistributes the rows from the source partition to the new partitions. This is a very expensive operation from the resource consumption point. In addition, local index partitions become unusable.

With the fast split partitioning, if all the rows will exist in the same partition after the partition split, Oracle simply reuses the old partition and creates an empty partition. Thus, a split operation becomes more like an operation that just creates a new partition.

Global indexes become unusable when a partition is rebuilt. However, in 9i, a new clause updates the global indexes as well.

```
ALTER TABLE PTAB DROP PARTITION P2 UPDATE
GLOBAL INDEXES;
```

Other Questions

While using partitioning, should you use bind variables?

This is an interesting question. As we all know use of bind variables eliminates the necessity of parsing of the cursors and facilitates the reuse of the cursors.

In case of partitions, however, this poses a problematic situation. Partition elimination and joins can occur only if the optimizer knows the filtering predicate in advance. The value of bind variables are not known until the execution time, making the process of partition elimination or joins impossible. Therefore, in order to take advantage of these options, you should not use bind variables.

In Oracle 9i, the first parse of the statement, called hard parse, peeks into the value of the bind variable and therefore can effect these optimization options. But, this occurs only for the hard parse; subsequent parses still go around the bind variable values.

How many partitions can be defined on a table?

Oracle uses a 2-byte field to store the number of segments (partitions or subpartitions), which enables 2^16 or 65536 spaces. The oracle code therefore allows one less than this number, which comes out to 65535. However, this is a limit placed by Oracle software code, a practical limit may be lower.

Remember, every time a query is parsed on a partitioned object, the metadata (i.e. how many partitions, etc.) is loaded into the cursor cache in SGA, meaning the SGA should be large enough to handle a table with several partitions.

About the Author

Arup Nanda has been Oracle DBA for last 10 years, experiencing all aspects of database administration along the way - from modeling to performance tuning to disaster recovery planning. He is the founder of Proligence, Inc. (www.proligence.com), a Norwalk, Connecticut based company that provides specialized Oracle database services like replication, disaster recovery, parallel server, among others. Arup is an editor of SELECT, the journal of International Oracle User Group. He has written Oracle technical articles in several national and international journals and presented in several conferences including IOUG Live! He can be reached at arup@proligence.com.