Oracle Partitioning – Issues and Handy Tips and Tricks
Sally Banyan, Server Technologies Team Cybern Consulting (UK) Ltd
Guildford, Surrey UK
Feb 2002

Definition

Oracle partitioning is a method of breaking up a very large table and/or its associated indexes into smaller pieces. Each piece, in essence, is either a table or an index although they are referred to as ‘partitions’ since together, they make up a larger object. Although indexes belonging to a given table are generally partitioned along with the table, Oracle does support the ability to partition tables and indexes independently such that you could have a regular, non-partitioned table but its associated indexes are partitioned. Each partition will be in its own segment and potentially, and for greatest flexibility, in its own tablespace (will allow independent backup and recovery).

Purpose and Benefits

The primary purpose of partitioning is faster query access. This is accomplished via partition pruning (elimination), a method where Oracle can query the data dictionary and determine the content or definition of a given partition without having to query that partition’s data, as it otherwise would in a non-partitioned table. In this way, Oracle can very quickly exclude large portions of data before the query search begins and not have to search through certain partitions at all in order to resolve a query. Rather, very focused subsets of data can be quickly isolated to be further refined.

The secondary purpose of partitioning allows easy archiving of data. In addition, if each partition is in its own tablespace, this would permit the easy exchange of an entire tablespace into a different, data warehouse database via Oracle’s transportable tablespace mechanism. Alternatively, partitions can also be individually exported to a file via the Oracle export utility.

Furthermore, partitioning gives the ability to quickly remove large amounts of data without fragmenting a table. This is done by using the truncate command to remove all rows in a given partition, as opposed to the regular method of simply deleting rows from a non-partitioned table. With deletion of rows, Oracle must first identify those rows matching the criteria as opposed to the truncation method where the rows are already pre-identified by virtue of the fact that they are in the very partition being truncated.

Other benefits would be the ability of Oracle to use parallelism to resolve queries since multiple partitions can be queried simultaneously. Both partition and non-partition tables can be in a single query. In addition, along with queries, delete and update operations should also benefit from partitioning as these too can use queries to identify the appropriate rows to either delete or update. Finally, when new data is loaded, Oracle can put this new data in its
own partition, if above the range of the last partition. In this way, the direct load method can be used to load data much more quickly and free block lists do not have to be scanned.

In addition to loading new data, partitions can also be added, dropped, renamed, altered, merged and split (some restrictions depending upon partition type used). Moreover, existing non-partition tables can be swapped to become partitioned tables and partitioned tables can be converted to regular tables.

**Affect on DML (Select, Insert, Update, Delete)**

Partitioned tables can be used just like any other table. When records are queried, inserted, updated or deleted, all the underlying mechanisms for finding the correct partition are handled transparently by Oracle. However, if needed, specific partitions can be queried and manipulated with extended syntax. With selects, after the table name, include the key word ‘PARTITION’ and then the name of the partition you want to specifically query. With inserts, updates and deletes, the process is almost exactly the same except that parentheses are needed around the partition name.

**Object Candidate Consideration**

The main criteria for determining which tables will be partitioned, will be the size of the table in bytes. Generally, tables above 1G should be considered and attempted first, as these should give the greatest gains. (I am making the assumption all associated indexes will also be partitioned unless there are many null values making the index of insufficient size to justify partitioning.) Of course, tables often queried should be focussed upon first.

In addition, the queries, or more specifically, the ‘where’ and join conditions commonly associated with such tables must also be identified. This will, in most cases, become the ‘partition key’. More than one column may make up the partition key. Special consideration must be given to tables that are often joined and that are also both candidates for partitioning. Best performance will be obtained if the partition key is the same for both tables.

**Partition Options (Methodology)**

**Range Partitioning**

Oracle 8 has expanded partition methods to two additional variations.

Range partitioning is often used when queries use dates in their ‘where’ clauses as a major criteria to eliminate records. Both a specific date as well as date ranges can be used such as ‘where utc_timestamp = ‘01-JAN-2000’ or utc_timestamp between ‘01-JAN-2000’ and ‘31-JAN-2000’. However, any
column often used in a where clause may be a candidate for the partition key although as the name implies, range partitioning is most useful where queries contain a bounded range such that columnA is between value1 and value2. The sizes of the individual partitions implementing range partitioning does not necessarily have to be the same. Range partitioning makes it easy to archive or delete large amounts of data and will probably be used with EMAN tables.

Hash Partitioning

This method appears to be better when values are not necessarily ordered such as in a abbreviation code or large numbers of single elements are used such as with an ‘in’ statement. Hash partitions allow Oracle to create evenly sized partitions so that the work can be better distributed. One drawback to hash partitions is that partition pruning can only be done if the criteria is limited to exact values, e.g. ‘=’ or ‘in’. Range queries will not be appropriate for hash partitioning. Additionally, the following operations are not permitted: split, drop, and merge. This does limit the partition's ability to be manipulated and would indicate it would be best where the table to be partitioned does not undergo much change or where it can be easily rebuilt without causing major application outage time.

Composite Partitioning

As the name implies, composite partitioning combines the previous two methods. This method seems best where there is first a primary range used and then a secondary value used, that would be more specific and non-range oriented. Composite should give even better performance than straight range partitioning if appropriate. Composite partitioning uses sub-partitioning. A sub-partition is another segment and allows further levels of parallelism as well as separate tablespace placement and storage clauses.

Indexes

Index partitioning has additional options of being local or global although local indexes are preferred, as they require less maintenance than global indexes when there are partition merges, drops, splits or even an insert using direct load. The main difference between the two is that the local index is only per partition while the global index spans partitions.

Additional Syntax Needed to Implement

In addition to the create table/index syntax, additional keywords needed to create new partitions are:

For range:
PARTITION BY RANGE(partition_key)
  (PARTITION partition_name VALUES LESS THAN (partition_key), ...)
  partition_name VALUES LESS THAN (MAXVALUE);

Note: with date ranges, the to_date function must also be used. Also, the last partition (with maxvalue) will be a ‘catch-all-else’ type bucket for any values that do not fit into the other partitions.

For hash:

PARTITION BY HASH(partition_key)
PARTITIONS partition_number

Note: with hash partitions, the number of partitions is best if it is in a power of two.

For composite – additional subpartition clause:

PARTITION BY RANGE(partition_key)
SUBPARTITION BY HASH (2nd_partition_key) SUBPARTITIONS partition_number
  (PARTITION partition_name VALUES LESS THAN (partition_key), ...)
  partition_name VALUES LESS THAN (MAXVALUE);

How to convert a table to a partitioned table

There are three methods of conversion: a) export/import, b) create/insert, c) alter table exchange partition command. Both methods b and c temporarily require double the amount of disk space per partition while option a requires disk space external to the database to hold the export dump file (although can be compressed). Option b seems like it would be the fastest and require the least steps. In addition, when these tables are created or before they are populated, I will turn nologging on per table to reduce the amount of redo generated and to speed the process. Once the population of data is complete, I will turn logging back on. For options b and c, Alter session enable parallel DML; should be issued so that the operations can be done in parallel as opposed to serial.

Import/Export method

1. Create an empty partitioned table (must use a new name but will rename at the end)
2. Export the table to be partitioned
3. Drop the original non-partitioned table (just exported)
4. Rename partitioned table to original table (just dropped)
5. Import the table. Data will automatically go to the correct partition
6. Build appropriate indexes – need to verify if indexes imported if they will be built local or global

Advantages: No additional database space is needed to hold the table while it is being partitioned. Export file can be compressed while it is being created so that less disk space is needed. Indexes possibly automatically created on the import but storage space would probably be incorrect because it would be based on the entire table, not the individual partition. Furthermore, index would probably be global and non-partitioned.
Disadvantages: Time spent to do an export. Additional time to do an import as opposed to the insert method of the other two options. Additional disk space outside the database is needed. The insert statements cannot be done in parallel unless the table is altered.

Create/Insert method

1. Create an empty partitioned table (must use a new name but will rename at the end)
2. Populate partitioned table via insert /*+ parallel(partitioned_table_name, degree_of_parallelism) / into partitioned_table as select /*+ parallel (orig_table_name,degree_of_parallelism) */ from original_table; Records will automatically go to the correct partition. Both inserts and selects will be parallelized and records will go direct load above high water mark. Most likely, only subsets of data will be able to be inserted or a pl/sql block created so that commit points can be issued to avoid rollback segment errors.
3. Drop the original non-partitioned table
4. Rename partitioned table to original table
5. Build appropriate indexes

Advantages: Can do inserts both in parallel as well as direct load (default with parallel) to speed processing. It may even be possible to do a CPTAS (Create partitioned table as select) in one step and to parallelize this as well. This needs to be further investigated.
Disadvantages: Additional database space may be needed to hold both copies of the table, partitioned and non-partitioned but this needs to be confirmed. Indexes will have to be manually created afterward.

Alter Table Exchange Partition

1. Create an empty partitioned table (must use a new name but will rename at the end)
2. Create a separate non-partitioned table that will correspond to the new partitions. This can be done as Create table as select * from ... where ... and will need to be done for each partition to be populated but can be done one at a time and operation can be parallelized.
3. Issue the command: `Alter table NEW_PARTITIONED_TABLE exchange partition FIRST_PARTITION_NAME with NEW_NON-
PARTITIONED_TABLE; This will NOT create a new table; this is only a
data dictionary update.
4. Optionally delete the data from the original non-partitioned table for the
   partition just created
5. Repeat steps 2-4 for each partition
6. Drop the original non-partitioned table
7. Rename partitioned table to original table
8. Build appropriate indexes

Advantages: Can be easily done in pieces
Disadvantages: Requires additional database space to hold both copies of the
table, partitioned and various non-partitioned as well as the original table.
Uses the most amount of database space of the three. After deletes in step 4
(which need to be controlled to avoid rollback segment issues), original table
can possibly be resized to recover some space.

Data Dictionary Views

After the partitions are created, additional views can be used, namely:
user_tab_partitions, user_part_tables,
user_ind_partitions, user_part_indexes. There are a number of
others such as if composite partitions are used that will include information on
sub-partitions as well.

Additional Terms

Regarding indexes, there are several new terms. The main one refers to
whether an index is prefixed or non-prefixixed. This simply indicates whether
the index in question includes at least the same first column (and preferably
all, in the same order) as the first column in the table partition key.

Restrictions

No LONG or similar type column data types are permitted.

Miscellaneous

Issue the command: ‘Alter table enable row movement;’ if you plan
to update a partition key that would cause it to migrate to another partition.
Creating partitioned indexes is very similar to creating partitioned tables
except that an additional keyword of ‘LOCAL’ is required; the default is global.
Indexes should preferably be local because if partitions are modified,
generally the global indexes need to be rebuilt.