# Partitioning Tables and Indexing Them

*Hemant K Chitale*
Standard Chartered Bank

## Introduction

Twenty years ago, databases were not very large; they did not support very many users. Backup (and Recovery) requirements were simple. However, Oracle recognized the advent of large datasets and introduced the VLDB ("Very Large Database") terminology in the Oracle8 documentation. Oracle Partitioning was made available as an optional feature to facilitate handling of large tables.

This paper is an *Introduction* to Partitioning. Complex Partitioning types and advanced commands in Partition maintenance are not in the scope.

Examples of much of the material in this paper are published in the Appendix pages.

## Pre – Oracle8 (and for those without the Partitioning Option installed)

Prior to Version 8, "home-grown" partitioning implementations by DBAs and Consultants were based on UNION-ALL Views on physically separate tables. Even now, if you do not purchase the Partitioning Option with the Enterprise Edition, you have to build multiple tables and "join" them together with views. Such partitioning is not transparent --- DML like INSERT, UPDATE and DELETE must explicitly specify the base table being manipulated, although queries can be executed across all the tables using the view.
In V7.3 you could also define Partition Views with explicit Check Constraints to identify the base table. [See the References in the Appendix]

## Elements of Table Partitioning

A Table is Partitioned into separate "pieces" on the basis of the values stored in one or more columns, known as the "Partition Key". The Partition Key unambiguously specifies which "piece" or Partition a row will be stored in (you can specify MAXVALUE and DEFAULT Partitions for "undefined" values in Range and List partitioning). Each Partition within the table is a separate Object and a separate Segment. (Similarly, if you extend Partitions into SubPartitions in Composite Partitioning, each SubPartition also is distinct from the "parent" Partition and Table).
Each Partition (and SubPartition) of a Table has the same logical attributes (Column Names, DataTypes, Scale and Precision). Constraints defined for the Table apply to all Partitions. However, Partitions can have distinct physical attributes (Tablespace for Storage, Extent Sizes, Compression). Note: If you create SubPartitions, then no Partition Segments are created because the physical storage is now in the SubPartitions.
[See the References in the Appendix]

## Common Types of Partitioning

Although 11g has introduced some more "complex" partitioning definitions, the most common ones still are:

- Range Partitioning (introduce in V8 and still the most popular)
- Hash Partitioning (introduced in 8i but much less used than Range and List)
- List Partitioning (introduced in 9i)
- Composite (Range-Hash) (Range-List) (List-Range) (List-Hash) etc Partitioning

11g has expanded the list of combinations for Composite Partitioning and added these complex definitions:

- Virtual Column Partitioning
- Reference Partitioning
- Interval Partitioning (as an extension to Range Partitioning)

Here, I shall be touching on only the simpler Partitioning methods as an introduction to Partitioning.

## Range Partitioning

In Range Partitioning, each Partition is defined by an upper bound for the values in the Partition Key column(s). Thus, data is mapped to a Partition based on the values in the Partition Key columns(s) vis-à-vis the boundary values for the Partition Keys. Each Partition is thus allowed a range of values from the value after the upper bound of the previous partition up to the upper bound of the same partition. Typically Range Partitioning is used against DATE columns (and Partitioning by Date has been the most popular method). However, when you need to Partition on multiple columns as a Partition Key (e.g. just as a Primary Key can be a composite of multiple columns), you may need to consider Range Partitioning as well.
Multiple columns can be concatenated together to act as the Partition Key. In such a case, Oracle matches the value of each incoming row in the order of the columns. If an incoming row matches the first Key column, Oracle checks the second Key column and so on.
[See the example in the Appendix]

## Hash Partitioning

In Hash Partitioning, you are only allowed to specify the Partition Key and the number of Partitions (i.e. you do not define the values to be used for partitioning). Hash Partitioning is used when you need to partition the table into smaller "pieces" but cannot identify how the rows are to be distributed across the Partitions. Oracle applies a Hashing algorithm to the values in the Partition Key in an attempt to distribute rows across the defined Partitions. This means that the Partition Key should have a large distribution of values. Ideally, you would define the number of Partitions to be a power of 2 – i.e. 2^N.
[See the examples in the Appendix]

## List Partitioning

List Partitioning is useful where you have a well-defined list of possible values for the Partition Key. Every row in a Partition will then have the same value for the Partition Key. Note that List Partitioning, unlike Range Partitioning, is restricted to a single column. 11g now allows composite List-List Partitioning where the SubPartitions can be based on a list of secondary column values.
[See the example in the Appendix]

## Choosing the Partitioning Method

Why would you need to / want to partition a table?  What is the nature of the data in the table?  The answers to these questions should drive your selection.  Here are some pointers:

1. Range Partitioning by a DATE column allows you to archive out older records simply by exporting the oldest partition and then truncating (or dropping) it.

2. Similarly, Date Ranged Partitions containing "old" data can be set to Read Only and backed up less frequently (by creating them in separate Tablespaces and, on expiry of the desired period – e.g. 3 years – setting the Tablespaces to Read Only).

3. If you periodically load "new" data (by date) and purge "old" data, Date Range Partitioning makes sense.

4. If you simply need to randomly distribute data across multiple "pieces"  (each Partition could be in a separate Tablespace on a separate set of disks as well), without relation to a business or logical view, you may consider using Hash Partitioning. Note that Range-Scan queries on the Partition Key will not perform – all queries must be based on Equality Predicates.

5. List Partitioning works for a small, discrete and "static" list of possible values for the Partitioning Key.  There may be a few changes where you might occasionally add a new Partition because a new value for the Partition Key is expected to be inserted into the table.

<u>Examples</u>
Let's look at some examples:

A. Sales data is loaded into a SALES_FACT table and needs to be retained for only 3 years.  Data is loaded daily or weekly.  The table can be Date Range Partitioned by Month with a new Partition created each month (a new Partitions at the upper-end is "added" by executing a SPLIT PARTITION against the MAXVALUE Partition).

B. Historical data of transactions needs to be preserved for 7 years.  However, data once inserted will not be updated. Data older than 1 year will be very rarely referenced. The data can be stored in a Date Range Partitioned table whereby data is maintained such :
    a. Data of years below 2004 has been purged by truncating their partitions
    b. Data of years 2004 to 2007 are in Partitions that have 1-year ranges (thus 4 Partitions) which have been moved one or more Tablespaces that are on Tier-3 storage.  Optionally, the COMPRESS attribute has also been set for these Partitions.  These Partitions (i.e. Tablespaces) are then set to READ ONLY after a Backup.  Thus, all subsequent BACKUP DATABASE runs do not need to backup this data repeatedly.
    c. Data of years 2008 to 2009 are in Partitions that are 1-year or Quarter-year ranges in Tablespaces on Tier-2 storage.  Optionally, the COMPRESS attribute has been set for these Partitions.
    d. Data of years 2010 and 2011 are in Partitions that are Quarter-year ranges in on Tier-1 storage

C. Employee information is to be stored by State.  The Table can be List Partitioned as the State names are well defined and unlikely to change frequently.

D. Statistical information from manufacturing equipment needs to be stored. The range of possible values is very high and the values cannot be predicted. Data will be queried by equality predicates (e.g. "DATA_VALUE IN (1,3,5,7)"). Hash Partitioning may distribute the data equally across multiple Partitions.

## Adding Data in Partitioned Tables

Single Row Inserts

When you insert data into a Partitioned Table, you do not need to specify the name of the target Partition. Oracle dynamically determines the target Partition as it views the value of the incoming Partition Key column(s). Thus, a simple `INSERT INTO tablename ….` will suffice. However, you can choose to optionally specify the Partition but Oracle will, nevertheless, check the value being inserted – if you name a Partition where the defined Partition Key values do not match the row being inserted, Oracle will raise an error. The examples on Range Partitioning and Hash Partitioning in the Appendix show how Oracle automatically maps the target Partition name and inserts the row into the "correct" Partition.
[See the example in the Appendix]

Bulk Inserts

Parallel DML and Direct Path INSERT (the "APPEND" Hint in SQL) can both be used. Oracle can use separate Parallel Slaves to insert into distinct target Partitions. When attempting such Bulk operations against a specific (named) Partition, only that target Partition is locked, the rest of the table is not locked. DML against rows other Partitions is permitted. This is a significant advantage of Partitioning. Direct Path operations against a non-partitioned table would lock the entire table.
[See the example in the Appendix]

"Switching" data from a non-partitioned Table to a Partitioned Table

The `ALTER TABLE tablename EXCHANGE PARTITION partitionname` syntax allows you to "switch" a non-partitioned Table with a Partition of an existing Partitioned Table. Obviously, the logical structure of the Tables must match. Similarly, DBMS_REDEFINITION can be used to "convert" a non-partitioned Table into a Partitioned Table by creating the Partitioned Table definition as the "interim" Table and copying rows from the existing (non-partitioned) Table to the new Partitioned Table.

## Maintaining Partitioned Tables

Oracle provides a number of maintenance commands for Partitions: ADD, DROP, COALESCE, TRUNCATE, SPLIT, MERGE, EXCHANGE and MOVE are commonly used.

ADD

The `ALTER TABLE tablename ADD PARTITION partitionname` can be used in Range, List and Hash Partitioning. Thus, you can "expand" the list of Partition Keys to allow for new data.

If the table is Range Partitioned, you can ADD a new Partition only at the "high" end after the last existing Partition. Otherwise, you SPLIT to "create" a new Partition out of an existing Partition.

If the List Partitioned Table already has a DEFAULT Partition, you need to SPLIT the DEFAULT to create a Partition for the new Partition Key value.
Note that for a Hash Partitioned table, Oracle has to re compute Hash values and reallocate the data from an existing Partition. So adding a new Partition would take significant time and resources and, importantly, result in "un-balanced" Partitions.
[See the example in the Appendix]

DROP or TRUNCATE
When you DROP a Table Partition, you physically "remove" the Partition. Data that was mapped to the Partition Key boundaries is no longer accessible. In a Range Partitioned Table, if you re-INSERT the same Partition Key, you will find the rows going into the "next" Partition. In a List Partitioned Table, data would fail to re-insert unless a DEFAULT Partition has been created.

COALESCE, SPLIT and MERGE
These commands can be used against adjacent Partitions or to create adjacent Partitions. (In Hash Partitioning, the COALESCE is used to reduce the number of Partitions). MAXVALUE and DEFAULT Partitions in Range and List Partitioning can be SPLITted to "create" new Partitions.

EXCHANGE
The EXCHANGE Partition command is useful to "merge" a non-Partitioned Table into an existing Partitioned Table. Thus, an empty Partition is exchanged with the populated table. Oracle simply updates the data dictionary information causing the Partition segment to be become an independent Table segment and the non-Partitioned Table to become a Partition. Similarly, EXCHANGE can be used to "move" data out of an existing Partitioned Table so that it can be copied or transported to another database.
[See the example in the Appendix]

DBMS_REDEFINITION and MOVE
DBMS_REDEFINITION allows you to "move" a Partition -- .e.g from one Tablespace to another as an "online" operation. Alternatively, the simple `ALTER TABLE tablename MOVE PARTITION partitionname` command can be used to make an "offline" move. Note that if the table is SubPartitioned, then the physical segments are the SubPartitioned – so it is the SubPartitions that are to be moved, not the Partitions.

**Maintaining Indexes**
Indexes against a Partitioned Table may be

- Global
- Global Partitioned
- Local

Global Indexes
Global Indexes are the "regular" indexes that you would create on a non-partitioned table. Typically Global Indexes are useful in an OLTP environment where queries against the Partitioned Table are not restricted to specific Partitions and/or do not specify the Partition Key that Oracle can use for automatic Partition Pruning.
Any Partition Maintenance operation (SPLIT, TRUNCATE, MERGE) can cause a Global Index to be made Invalid. Since 10g, Oracle has added the keywords "UPDATE GLOBAL

INDEXES" to Partition Maintenance commands so as to update the Global Indexes as well. If you do not include this clause, you may find Global Indexes to be marked UNUSABLE. This is because while Partition Maintenance operation is executed as a DDL, updates to a Global Index must be executed as normal DML.
[See the example in the Appendix]

Global Partitioned Indexes
Global Partitioned Indexes are indexes that, while partitioned, are not partitioned along the same key as the table. Thus, with respect to the Table Partitions, they are still "Global". A Partition of a Global Partitioned Index may have keys referencing none, one or multiple partitions of the table --- there is no one-to-one correspondence between Index Partitions and Table Partitions.

Local Index
A Local Index (what I call "Locally Partitioned Index") is Equi-Partitioned with the Table. For every Table Partition, there is a corresponding Index Partition and vice-versa. Entries in an Index Partition will refer only to rows in the corresponding Table Partition. Thus, a scan of a particular Index Partition will never have to read rowids for rows not in the matching Table Partition.
A Local Index is ideally usable in DWH/DSS environments.
Note that if the Index is to be a Unique Index, it *must* include the Partition Key column(s) because the Index does not reference any rows outside the corresponding Table Partition so it has to use the Partition Key columns to help maintain Uniqueness across the Table.
Partition Maintenance operations against the Table may automatically applied to the corresponding Index Partitions – the DBA does not have to explicitly specify the Index Partitions. Thus an `ALTER TABLE tablename DROP PARTITION partitionname` automatically drops the corresponding Index Partitions as well without making the Index UNUSABLE. However, depending on the nature of the Partitioning and the presence/absence of rows, actions like SPLIT and MERGE may cause Index Partitions to be left UNUSABLE unless the UPDATE INDEXES clause is used. The number of possible combinations is too numerous to enumerate here (ADD, DROP, TRUNCATE, SPLIT, MERGE against Range, List, Hash and Composite Partitioned tables, against regular Partitions, the DEFAULT Partition or MAXVALUE partition, with or without rows).
[See the example in the Appendix]

**Performance Strategies**
Here are some performance strategies:

1. Ideally, you want to be able to use Partition Pruning when querying subsets of the data. Thus, when querying for only a month's data in a table containing 5 years data, the DATE column would be the Partition Key and the query would specify the DATE column as a query predicate.

2. When doing a Bulk Insert into a Partitioned Table, use Parallel and Direct Path Inserts. Consider NoLogging but be aware of the pitfalls.

3. If a query submitted by the application cannot supply the Partition Key as a predicate, a Local Index is not usable – consider creating a Global Index (but note the maintenance overheads when doing Partition Maintenance).

4. If two tables are to be joined frequently, consider defining Equi-Partitioning on both tables – i.e. have the same Partition Key and boundaries. Oracle can then execute Partition-Wise Joins.

5. If attempting to EXCHANGE PARTITION with a Table where a Unique Constraint and Index is to be maintained, create the Unique Index as a Local index on the Partitioned Table (as well as a "normal" Unique Index on the Exchange Table). Then execute `ALTER TABLE tablename DISABLE CONSTRAINT constraintname KEEP INDEX` on the exchange table *before* attempting the EXCHANGE with INCLUDING INDEXES NOVALIDATE. (Remember to ensure that the rows in the table *are* Unique with the Constraint Enabled upto the point you attempt the EXCHANGE PARTITION!!)

If you have data that is Partitioned but your queries are not doing Partition Pruning (filtering by the Partition Key as a predicate), you must seriously consider your Partitioning design. Also remember that range-based queries cannot be used against a Hash Partitioned table.

**Archiving Data**
As has been pointed out, Partitioning by a DATE is a right fit for Archival requirements. "Old" data is automatically stored in Partitions that identify the age of the data. These Partitions can then be exported from the database and preserved as backups outside of the database. Alternatively, they can be moved to Tier-2 or Tier-3 storage on slower/cheaper disks (by placing them in Tablespaces with Datafiles on such storage). The oldest partitions can be TRUNCATEd when the data is no longer needed.

Remember that you can also use SPLIT and MERGE commands to manage Range Partitions. Thus, as data gets "older" you can "merge" the Month Partitions containing data that is 36-40months old into a Quarter Partitions. Over time, you can merge older Quarter Partitions into Year Partitions. The oldest Year partitions can then be archived or moved to Tier-2 or Tier-3 storage.
[See the example in the Appendix]

**Common Mistakes**
Some common mistakes that I see are:

1. Using the wrong Partitioning type – particularly Range where List should have been used.
   An example I came across recently was:
   Two columns from a table
   ```
   YEAR NUMBER(4) NOT NULL,
   PERIOD NUMBER(2) NOT NULL,
   ```
   Partition Key definitions
   ```
   PARTITION BY RANGE (YEAR)
   SUBPARTITION BY HASH (PERIOD)
   ```
   Ideally, this would be Range Partitioning on (YEAR,PERIOD) OR List-List Composite Partitioning in 11gR2. (In fact, we might even ask why these two

columns are NUMBER columns and take a more detailed look at the Schema design).

2. Not defining all the proper Ranges causing “undefined” values to be entered into an unexpected Range Partition – this can be particularly troublesome if you were to TRUNCATE partitions without verifying the data. In the Range Partitioning example, I showed how rows for ‘JP’ were saved in the ‘SG’ partition. If the DBA were to TRUNCATE the ‘SG’ partitions, he would lose the ‘JP’ data as well! This is very dangerous!

3. Not defining the right boundaries causing Partitions to be unequally sized – Hash Partitioning might be preferable if the Partition boundaries cannot be explicitly identified.

4. Attempting to move data from one Partition to another. By default, ROW MOVEMENT is not enabled – Oracle does not expect rows to move between partitions. If rows have to move between partitions frequently, it would indicate an incorrectly defined Partitioning schema (somewhat akin to frequent updates to a Primary Key indicating an incorrectly defined Primary Key).

5. Incorrectly creating LOCAL indexes when GLOBAL indexes are required – for queries that do not do Partition Pruning. Such queries end up scanning each Index Partition separately or doing a Full Table Scan, completely bypassing the Index.

6. Defining a GLOBAL index on a DWH/DSS table where all DML and queries are against targeted partitions. DML requires updates to the GLOBAL index and Partition Maintenance operations can result in the GLOBAL index being UNUSABLE.

**Conclusion**

Oracle Partitioning has varied uses in Performance, Data Management, Data Warehousing (quick loading of data). It can be used concurrently with Parallel Query, Direct Path operations and, when necessary, NoLogging commands. However, care must be taken to define the *correct* Partitioning method.

# Appendix:  References and Examples

**References : Partition Views**

1. There's a document "Handling large datasets - Partition Views (Nov 1996)" by Jonathan Lewis which explains the concept.
2. Also see Oracle Support articles
   a. "Partition Views in 7.3: Examples and Tests [ID 43194.1]"
   b. "How to convert 7.3.X partition view tables to 8.X partition tables [ID 1055257.6]").

**Elements of Table Partitions**

This demonstration shows how Partitions and SubPartitions (implementing Range-List Composite Partitioning) appear as Objects and Segments that are distinct from the Table.

```
SQL> connect PART_DEMO/PART_DEMO
Connected.
SQL>
SQL> drop table SALES_TABLE;

Table dropped.

SQL> purge recyclebin;

Recyclebin purged.

SQL>
SQL> create table SALES_TABLE(sale_date date not null, region varchar2(8), sale_qty number)
  2  partition by range (sale_date) subpartition by list (region)
  3  (
  4  partition p_2010 values less than (to_date('01-JAN-2011','DD-MON-YYYY'))
  5    (subpartition p_2010_s_east values ('EAST'),
  6     subpartition p_2010_s_north values ('NORTH'),
  7     subpartition p_2010_s_south values ('SOUTH'),
  8     subpartition p_2010_s_west values ('WEST')
  9    )
 10  ,
 11  partition p_2011 values less than (to_date('01-JAN-2012','DD-MON-YYYY'))
 12    (subpartition p_2011_s_east values ('EAST'),
 13     subpartition p_2011_s_north values ('NORTH'),
 14     subpartition p_2011_s_south values ('SOUTH'),
 15     subpartition p_2011_s_west values ('WEST')
 16    )
 17  )
 18  /

Table created.

SQL>
SQL> select object_id, object_name, subobject_name, object_type
  2  from user_objects
  3  order by object_type, object_name, subobject_name
  4  /

 OBJECT_ID OBJECT_NAME            SUBOBJECT_NAME            OBJECT_TYPE
---------- ---------------------- ------------------------- -------------------
     54889 SALES_TABLE                                      TABLE
     54890 SALES_TABLE            P_2010                    TABLE PARTITION
     54891 SALES_TABLE            P_2011                    TABLE PARTITION
     54892 SALES_TABLE            P_2010_S_EAST             TABLE SUBPARTITION
     54893 SALES_TABLE            P_2010_S_NORTH            TABLE SUBPARTITION
     54894 SALES_TABLE            P_2010_S_SOUTH            TABLE SUBPARTITION
     54895 SALES_TABLE            P_2010_S_WEST             TABLE SUBPARTITION
     54896 SALES_TABLE            P_2011_S_EAST             TABLE SUBPARTITION
     54897 SALES_TABLE            P_2011_S_NORTH            TABLE SUBPARTITION
     54898 SALES_TABLE            P_2011_S_SOUTH            TABLE SUBPARTITION
     54899 SALES_TABLE            P_2011_S_WEST             TABLE SUBPARTITION
```

```
11 rows selected.

SQL>
SQL> select segment_name, partition_name, segment_type, tablespace_name
  2  from user_segments
  3  order by segment_name, partition_name
  4  /

SEGMENT_NAME    PARTITION_NAME       SEGMENT_TYPE       TABLESPACE_NAME
--------------- -------------------- ------------------ ----------------
SALES_TABLE     P_2010_S_EAST        TABLE SUBPARTITION USERS
SALES_TABLE     P_2010_S_NORTH       TABLE SUBPARTITION USERS
SALES_TABLE     P_2010_S_SOUTH       TABLE SUBPARTITION USERS
SALES_TABLE     P_2010_S_WEST        TABLE SUBPARTITION USERS
SALES_TABLE     P_2011_S_EAST        TABLE SUBPARTITION USERS
SALES_TABLE     P_2011_S_NORTH       TABLE SUBPARTITION USERS
SALES_TABLE     P_2011_S_SOUTH       TABLE SUBPARTITION USERS
SALES_TABLE     P_2011_S_WEST        TABLE SUBPARTITION USERS

8 rows selected.

SQL>
SQL>
SQL> select partition_name, partition_position, high_value
  2  from user_tab_partitions
  3  where table_name = 'SALES_TABLE'
  4  order by partition_position
  5  /

PARTITION_NAME       PARTITION_POSITION
-------------------- ------------------
HIGH_VALUE
--------------------------------------------------------------------------------
P_2010                                1
TO_DATE(' 2011-01-01 00:00:00', 'SYYYY-MM-DD HH24:MI:SS', 'NLS_CALENDAR=GREGORIA

P_2011                                2
TO_DATE(' 2012-01-01 00:00:00', 'SYYYY-MM-DD HH24:MI:SS', 'NLS_CALENDAR=GREGORIA


SQL>
SQL> select partition_name,subpartition_name, subpartition_position, high_value
  2  from user_tab_subpartitions
  3  where table_name = 'SALES_TABLE'
  4  order by partition_name, subpartition_position
  5  /

PARTITION_NAME       SUBPARTITION_NAME    SUBPARTITION_POSITION
-------------------- -------------------- ---------------------
HIGH_VALUE
--------------------------------------------------------------------------------
P_2010               P_2010_S_EAST                            1
'EAST'

P_2010               P_2010_S_NORTH                           2
'NORTH'

P_2010               P_2010_S_SOUTH                           3
'SOUTH'

P_2010               P_2010_S_WEST                            4
'WEST'

P_2011               P_2011_S_EAST                            1
'EAST'

P_2011               P_2011_S_NORTH                           2
'NORTH'

P_2011               P_2011_S_SOUTH                           3
'SOUTH'

P_2011               P_2011_S_WEST                            4
'WEST'


8 rows selected.
```

```
SQL>
```

Note :  In 11g with “deferred_segment_creation” set to TRUE, segments are not created until at least one row is inserted into each “target” segment.

## Range Partitioning

This demonstration shows how Oracle matches the incoming values against the Partition Key boundaries.  Notice how the records for ‘JP’ and ‘US’ go into the ‘SG’ and ‘MAX’ partitions ignoring the acctg_year value after the first column of the partition key is “matched”.

```
SQL> drop table ACCOUNTING ;

Table dropped.

SQL> purge recyclebin;

Recyclebin purged.

SQL>
SQL> create table ACCOUNTING
  2  (biz_country varchar2(10) not null, acctg_year number not null, data_1 varchar2(20))
  3  partition by range (biz_country, acctg_year)
  4  (
  5  partition p_in_2006 values less than ('IN',2007),
  6  partition p_in_2007 values less than ('IN',2008),
  7  partition p_in_2008 values less than ('IN',2009),
  8  partition p_sg_2006 values less than ('SG',2007),
  9  partition p_sg_2007 values less than ('SG',2008),
 10  partition p_sg_2008 values less than ('SG',2009),
 11  partition p_max values less than (MAXVALUE, MAXVALUE)
 12  )
 13  /

Table created.

SQL>
SQL> insert into ACCOUNTING values ('IN',2007,'Row 1');

1 row created.

SQL> insert into ACCOUNTING values ('IN',2008,'Row 2');

1 row created.

SQL> insert into ACCOUNTING values ('JP',2007,'Row 3');

1 row created.

SQL> insert into ACCOUNTING values ('JP',2015,'Row 4');

1 row created.

SQL> insert into ACCOUNTING values ('US',2006,'Row 5');

1 row created.

SQL> insert into ACCOUNTING values ('US',2009,'Row 6');

1 row created.

SQL>
SQL> select * from ACCOUNTING partition (p_in_2006);

no rows selected

SQL> select * from ACCOUNTING partition (p_in_2007);

BIZ_COUNTR ACCTG_YEAR DATA_1
```

```
---------- ---------- --------------------
IN               2007 Row 1

SQL> select * from ACCOUNTING partition (p_in_2008);

BIZ_COUNTR ACCTG_YEAR DATA_1
---------- ---------- --------------------
IN               2008 Row 2

SQL> select * from ACCOUNTING partition (p_sg_2006);

BIZ_COUNTR ACCTG_YEAR DATA_1
---------- ---------- --------------------
JP               2007 Row 3
JP               2015 Row 4

SQL> select * from ACCOUNTING partition (p_max);

BIZ_COUNTR ACCTG_YEAR DATA_1
---------- ---------- --------------------
US               2006 Row 5
US               2009 Row 6

SQL>
```

Had I used List Partitioning, I would have avoided 'JP' and 'US' going into the 'SG' Partition (they could have been sent to a DEFAULT Partition).  However, when I want to define a multi-column Partition Key, I cannot use List Partitioning --- List Partitioning restricted to a single column.  Therefore, I use Range Partitioning for a 2-column key.  But as the example shows, I must be very careful with my Partition bounds and know what data is getting inserted !
Note : 11g now supports composite List-List Partitioning.  So you could address this issue in 11g – but remember it is still only 2 levels, what if you wanted a third column in your Partition Key ?

## Hash Partitioning

This example shows how it is important to match the number of Partitions to the range of values.  A mismatch causes unbalanced Partitions.

```
SQL> drop table MACHINE_DATA ;

Table dropped.

SQL> purge recyclebin;

Recyclebin purged.

SQL>
SQL> create table MACHINE_DATA
  2  (machine_id  number not null, data_value  number not null, read_date date)
  3  partition by hash (data_value)
  4  (partition P_1, partition P_2, partition P_3, partition P_4, partition P_5)
  5  /

Table created.

SQL>
SQL>
SQL> -- first example of poor distribution
SQL> -- 8 distinct data_values are created
SQL> insert into MACHINE_DATA
  2  select mod(rownum,10), mod(rownum,8), sysdate+rownum/1000000
  3  from dual connect by level < 1000000
  4  /

999999 rows created.
```

```
SQL>
SQL> exec
dbms_stats.gather_table_stats('','MACHINE_DATA',estimate_percent=>100,granularity=>'ALL');

PL/SQL procedure successfully completed.

SQL>
SQL> select partition_name, num_rows
  2  from user_tab_partitions
  3  where table_name = 'MACHINE_DATA'
  4  order by partition_position
  5  /

PARTITION_NAME                   NUM_ROWS
------------------------------ ----------
P_1                                125000
P_2                                124999
P_3                                250000
P_4                                500000
P_5                                     0

SQL>
SQL> select data_value, count(*)
  2  from MACHINE_DATA
  3  group by data_value
  4  order by data_value
  5  /

DATA_VALUE   COUNT(*)
---------- ----------
         0     124999
         1     125000
         2     125000
         3     125000
         4     125000
         5     125000
         6     125000
         7     125000

8 rows selected.

SQL>
SQL>
SQL> -- second example of poor distribution
SQL> -- 10500 distinct data_values are created
SQL> drop table MACHINE_DATA ;

Table dropped.

SQL> purge recyclebin;

Recyclebin purged.

SQL>
SQL> create table MACHINE_DATA
  2  (machine_id  number not null, data_value  number not null, read_date date)
  3  partition by hash (data_value)
  4  (partition P_1, partition P_2, partition P_3, partition P_4, partition P_5)
  5  /

Table created.

SQL>
SQL> insert into MACHINE_DATA
  2  select mod(rownum,10), mod(rownum,10500), sysdate+rownum/1000000
  3  from dual connect by level < 1000000
  4  /

999999 rows created.

SQL>
SQL> exec
dbms_stats.gather_table_stats('','MACHINE_DATA',estimate_percent=>100,granularity=>'ALL');

PL/SQL procedure successfully completed.

SQL>
SQL> select partition_name, num_rows
```

```
  2  from user_tab_partitions
  3  where table_name = 'MACHINE_DATA'
  4  order by partition_position
  5  /

PARTITION_NAME                   NUM_ROWS
------------------------------ ----------
P_1                                128669
P_2                                252465
P_3                                251151
P_4                                248675
P_5                                119039

SQL>
SQL>
SQL> -- third example with 2^N partitions
SQL> -- 10500 distinct data_values are created
SQL> drop table MACHINE_DATA ;

Table dropped.

SQL> purge recyclebin;

Recyclebin purged.

SQL>
SQL> create table MACHINE_DATA
  2  (machine_id  number not null, data_value  number not null, read_date date)
  3  partition by hash (data_value)
  4  (partition P_1, partition P_2, partition P_3, partition P_4,
  5   partition P_5, partition P_6, partition P_7, partition P_8)
  6  /

Table created.

SQL>
SQL> insert into MACHINE_DATA
  2  select mod(rownum,10), mod(rownum,10500), sysdate+rownum/1000000
  3  from dual connect by level < 1000000
  4  /

999999 rows created.

SQL>
SQL> exec
dbms_stats.gather_table_stats('','MACHINE_DATA',estimate_percent=>100,granularity=>'ALL');

PL/SQL procedure successfully completed.

SQL>
SQL> select partition_name, num_rows
  2  from user_tab_partitions
  3  where table_name = 'MACHINE_DATA'
  4  order by partition_position
  5  /

PARTITION_NAME                   NUM_ROWS
------------------------------ ----------
P_1                                128669
P_2                                123503
P_3                                126000
P_4                                120182
P_5                                119039
P_6                                128962
P_7                                125151
P_8                                128493

8 rows selected.

SQL>
SQL> select count(distinct(ora_hash(data_value))) from machine_data;

COUNT(DISTINCT(ORA_HASH(DATA_VALUE)))
-------------------------------------
                                10500

SQL>
```

## List Partitioning

This example shows a table incorrectly defined as Range Partitioned when it should have been List Partitioned :

```
SQL> -- badly defined as Range Partitioned
SQL> drop table MONTH_END_BALANCES;

Table dropped.

SQL> purge recyclebin;

Recyclebin purged.

SQL> create table MONTH_END_BALANCES
  2  (Partition_Key     varchar2(8) not null, account_number number, balance number)
  3  partition by Range (Partition_Key)
  4  (partition P_2011_JAN values less than ('20110132'),
  5  partition P_2011_FEB values less than ('20110229'),
  6  partition P_2011_MAR values less than ('20110332'),
  7  partition P_2011_APR values less than ('20110431'),
  8  partition P_2011_MAY values less than ('20110532'),
  9  partition P_2011_JUN values less than ('20110631'),
 10  partition P_2011_JUL values less than ('20110732'),
 11  partition P_2011_AUG values less than ('20110832'),
 12  partition P_2011_SEP values less than ('20110931'),
 13  partition P_2011_OCT values less than ('20111032'),
 14  partition P_2011_NOV values less than ('20111131'),
 15  partition P_2011_DEC values less than ('20111232')
 16  )
 17  /

Table created.

SQL>
SQL> -- correctly defined as List Partitioned
SQL> drop table MONTH_END_BALANCES;

Table dropped.

SQL> purge recyclebin;

Recyclebin purged.

SQL> create table MONTH_END_BALANCES
  2  (Partition_Key     varchar2(6) not null, account_number number, balance number)
  3  partition by List (Partition_Key)
  4  (partition P_2011_JAN values ('201101'),
  5  partition P_2011_FEB values ('201102'),
  6  partition P_2011_MAR values ('201103'),
  7  partition P_2011_APR values ('201104'),
  8  partition P_2011_MAY values ('201105'),
  9  partition P_2011_JUN values ('201106'),
 10  partition P_2011_JUL values ('201107'),
 11  partition P_2011_AUG values ('201108'),
 12  partition P_2011_SEP values ('201109'),
 13  partition P_2011_OCT values ('201110'),
 14  partition P_2011_NOV values ('201111'),
 15  partition P_2011_DEC values ('201112')
 16  )
 17  /

Table created.

SQL>
```

What is the significant difference between the two definitions ?  We know that even in the first definition, all the rows in a particular Partition have the same value for the PARTITION_KEY column.  However, when it is defined as Range Partitioned, the

Optimizer *cannot* be sure that this is so.  For example, with a Range Partitioned definition, the P_2011_JAN partition might have values '20110129', '20110130' in two rows out of a million rows.  We know this won't be the case --- but the Optimizer cannot know so.  On the other hand,  the List Partition definition acts like a constraint – every row in the P_2011_JAN partition will only have the value '201101' and no other value.

## Adding Data in Partitioned Tables

Single Row Inserts

This demonstration shows how Oracle automatically maps the target Partition name when it is not specified but, nevertheless, verifies the target Partition when it is explicitly specified. Note also how the specification can be Partition name or even a SubPartition name.

```
SQL> desc sales_table
 Name                                      Null?    Type
 ----------------------------------------- -------- ----------------------------
 SALE_DATE                                 NOT NULL DATE
 REGION                                             VARCHAR2(8)
 SALE_QTY                                           NUMBER

SQL> select partition_name, subpartition_name from user_tab_subpartitions
  2  where table_name = 'SALES_TABLE'
  3  order by 1,2;

PARTITION_NAME                 SUBPARTITION_NAME
------------------------------ ------------------------------
P_2010                         P_2010_S_EAST
P_2010                         P_2010_S_NORTH
P_2010                         P_2010_S_SOUTH
P_2010                         P_2010_S_WEST
P_2011                         P_2011_S_EAST
P_2011                         P_2011_S_NORTH
P_2011                         P_2011_S_SOUTH
P_2011                         P_2011_S_WEST

8 rows selected.

SQL>
SQL> insert into sales_table values (to_date('05-NOV-11','DD-MON-RR'),'EAST', 1);

1 row created.

SQL> insert into sales_table partition (P_2010)
  2  values (to_date('05-AUG-11','DD-MON-RR'),'WEST',1);
insert into sales_table partition (P_2010)
            *
ERROR at line 1:
ORA-14401: inserted partition key is outside specified partition


SQL>
SQL> insert into sales_table partition (P_2011)
  2  values (to_date('05-AUG-11','DD-MON-RR'),'WEST',1);

1 row created.

SQL>
SQL>  insert into sales_table subpartition (P_2011_S_WEST)
  2  values (to_date('05-SEP-11','DD-MON-RR'),'EAST',1);
 insert into sales_table subpartition (P_2011_S_WEST)
             *
ERROR at line 1:
ORA-14401: inserted partition key is outside specified partition


SQL> insert into sales_table subpartition (P_2011_S_EAST)
  2  values (to_date('05-SEP-11','DD-MON-RR'),'EAST',1);

1 row created.
```

```
SQL>
```

## Bulk Insert

This demonstration shows a Bulk Insert from a source table (I use DUAL as a simulated source table) with Parallel and Direct Path operations.

```
SQL> desc month_end_balances;
 Name                                      Null?    Type
 ----------------------------------------- -------- ----------------------------
 PARTITION_KEY                             NOT NULL VARCHAR2(6)
 ACCOUNT_NUMBER                                     NUMBER
 BALANCE                                            NUMBER

SQL> alter session enable parallel dml;

Session altered.

SQL> select count(*) from month_end_balances;

  COUNT(*)
----------
         0

SQL>
SQL> insert /*+ APPEND PARALLEL (meb 4) */
  2  into month_end_balances meb
  3  select /*+ PARALLEL (s 4) */
  4  decode(mod(rownum,4),0,'201101',1,'201102',2,'201103',3,'201104'),
  5  mod(rownum,125),
  6  rownum
  7  from dual s
  8  connect by level < 1000001
  9  /

1000000 rows created.

SQL>
SQL> pause Press ENTER to commit
Press ENTER to commit

SQL> commit;

Commit complete.

SQL>
SQL> select /*+ PARALLEL (meb 4) */
  2  partition_key, count(*)
  3  from month_end_balances meb
  4  group by partition_key
  5  order by 1
  6  /

PARTIT   COUNT(*)
------ ----------
201101     250000
201102     250000
201103     250000
201104     250000

SQL>
```

## Maintaining Partitioned Tables

## ADD Partition

This is an example of adding a Partition to a Hash Partitioned Table.  Note how data for one existing Partition is split and reassigned.  The Partitions are now “unbalanced”.

```
SQL> exec
dbms_stats.gather_table_stats('','MACHINE_DATA',estimate_percent=>100,granularity=>'ALL');
```

```
PL/SQL procedure successfully completed.

SQL>
SQL> select partition_name, num_rows
  2  from user_tab_partitions
  3  where table_name = 'MACHINE_DATA'
  4  order by partition_position
  5  /

PARTITION_NAME                   NUM_ROWS
------------------------------ ----------
P_1                                128669
P_2                                123503
P_3                                126000
P_4                                120182
P_5                                119039
P_6                                128962
P_7                                125151
P_8                                128493

8 rows selected.

SQL>
SQL> set autotrace on statistics
SQL> alter table MACHINE_DATA
  2  add partition P_9
  3  /

Table altered.

SQL>
SQL>
SQL> exec
dbms_stats.gather_table_stats('','MACHINE_DATA',estimate_percent=>100,granularity=>'ALL');

PL/SQL procedure successfully completed.

SQL>
SQL> select partition_name, num_rows
  2  from user_tab_partitions
  3  where table_name = 'MACHINE_DATA'
  4  order by partition_position
  5  /

PARTITION_NAME                   NUM_ROWS
------------------------------ ----------
P_1                                 63431
P_2                                123503
P_3                                126000
P_4                                120182
P_5                                119039
P_6                                128962
P_7                                125151
P_8                                128493
P_9                                 65238

9 rows selected.

SQL>
```

## <u>EXCHANGE PARTITION</u>

This demonstration shows how a populated non-Partitioned Table (a Staging Table) is exchanged with an empty partition very quickly

```
SQL> select /*+ PARALLEL (meb 4) */
  2  partition_key, count(*)
  3  from MONTH_END_BALANCES
  4  group by partition_key
  5  order by 1
  6  /

PARTIT   COUNT(*)
------ ----------
201101     250000
201102     250000
201103     250000
```

```
201104     250000


Statistics
----------------------------------------------------------
          1  recursive calls
          1  db block gets
       2837  consistent gets
          0  physical reads
         96  redo size
        548  bytes sent via SQL*Net to client
        385  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
         12  sorts (memory)
          0  sorts (disk)
          4  rows processed

SQL>
SQL> -- supposing that we get a staging table from the ODS/Source
SQL> -- the staging table has the same structure
SQL>
SQL> create table STAGING_BALANCES_TABLE
  2  (Partition_Key     varchar2(6) not null, account_number number, balance number)
  3  /

Table created.

SQL>
SQL> -- the Staging Table has been populated by the source
SQL> insert into STAGING_BALANCES_TABLE
  2  select
  3  '201105',
  4  mod(rownum,125),
  5  rownum
  6  from dual s
  7  connect by level < 250001
  8  /

250000 rows created.


Statistics
----------------------------------------------------------
        695  recursive calls
       7624  db block gets
       1565  consistent gets
          0  physical reads
    6659388  redo size
        683  bytes sent via SQL*Net to client
        652  bytes received via SQL*Net from client
          4  SQL*Net roundtrips to/from client
          3  sorts (memory)
          0  sorts (disk)
     250000  rows processed

SQL> select count(*) from STAGING_BALANCES_TABLE;

  COUNT(*)
----------
    250000


Statistics
----------------------------------------------------------
         29  recursive calls
          1  db block gets
        820  consistent gets
          0  physical reads
        132  redo size
        411  bytes sent via SQL*Net to client
        385  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
          1  rows processed

SQL>
SQL> -- we do an EXCHANGE
SQL> set timing on
SQL> alter table  MONTH_END_BALANCES exchange partition P_2011_MAY with table
STAGING_BALANCES_TABLE;

Table altered.

Elapsed: 00:00:00.16
SQL>
SQL> -- verify
SQL> select count(*) from MONTH_END_BALANCES partition (P_2011_MAY);

  COUNT(*)
----------
    250000

Elapsed: 00:00:00.01

Statistics
----------------------------------------------------------
          1  recursive calls
          0  db block gets
        740  consistent gets
          0  physical reads
          0  redo size
        411  bytes sent via SQL*Net to client
        385  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
          1  rows processed

SQL> select count(*) from STAGING_BALANCES_TABLE;

  COUNT(*)
----------
         0

Elapsed: 00:00:00.00

Statistics
----------------------------------------------------------
          1  recursive calls
          0  db block gets
          4  consistent gets
          0  physical reads
          0  redo size
        410  bytes sent via SQL*Net to client
        385  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
          1  rows processed

SQL>
SQL> select /*+ PARALLEL (meb 4) */
  2  partition_key, count(*)
  3  from MONTH_END_BALANCES
  4  group by partition_key
  5  order by 1
  6  /

PARTIT   COUNT(*)
------ ----------
201101     250000
201102     250000
201103     250000
201104     250000
201105     250000

Elapsed: 00:00:00.35

Statistics
----------------------------------------------------------
          1  recursive calls
          0  db block gets
       3572  consistent gets
          0  physical reads
          0  redo size
        560  bytes sent via SQL*Net to client
        385  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
         12  sorts (memory)
          0  sorts (disk)
          5  rows processed

SQL>
```

**Maintaining Indexes**

Global Indexes

This demonstration shows how a Global Index is automatically marked UNUSABLE when Partition Maintenance is performed.

```
SQL> desc month_end_balances;
 Name                                      Null?    Type
 ----------------------------------------- -------- ----------------------------
 PARTITION_KEY                             NOT NULL VARCHAR2(6)
 ACCOUNT_NUMBER                                     NUMBER
 BALANCE                                            NUMBER

SQL> create index meb_a_no_ndx on month_end_balances(account_number) parallel 4 nologging;

Index created.

SQL> select status from user_indexes where index_name = 'MEB_A_NO_NDX';

STATUS
--------
VALID

SQL> alter table month_end_balances truncate partition P_2011_JAN;

Table truncated.

SQL> select status from user_indexes where index_name = 'MEB_A_NO_NDX';

STATUS
--------
UNUSABLE

SQL>
SQL> alter index meb_a_no_ndx rebuild;

Index altered.

SQL> select status from user_indexes where index_name = 'MEB_A_NO_NDX';

STATUS
--------
VALID

SQL> alter table month_end_balances truncate partition P_2011_FEB update global indexes;

Table truncated.

SQL> select status from user_indexes where index_name = 'MEB_A_NO_NDX';

STATUS
--------
VALID

SQL>
```

Local Index

This demonstration shows a locally partitioned index where Index Partitions are automatically created to match the Table Partitions :

```
SQL> desc accounting
 Name                                      Null?    Type
 ----------------------------------------- -------- ----------------------------
 BIZ_COUNTRY                               NOT NULL VARCHAR2(10)
 ACCTG_YEAR                                NOT NULL NUMBER
 DATA_1                                             VARCHAR2(20)

SQL> create index accntng_data_ndx on accounting(data_1) LOCAL;

Index created.

SQL> select partition_name from user_tab_partitions where table_name = 'ACCOUNTING';

PARTITION_NAME
------------------------------
P_IN_2006
P_IN_2007
P_IN_2008
P_MAX
P_SG_2006
P_SG_2007
P_SG_2008

7 rows selected.

SQL> select index_name, status from user_indexes where table_name = 'ACCOUNTING';

INDEX_NAME                     STATUS
------------------------------ --------
ACCNTNG_DATA_NDX               N/A

SQL> select partition_name, status from user_ind_partitions where index_name =
'ACCNTNG_DATA_NDX';

PARTITION_NAME                 STATUS
------------------------------ --------
P_IN_2006                      USABLE
P_IN_2007                      USABLE
P_IN_2008                      USABLE
P_MAX                          USABLE
P_SG_2006                      USABLE
P_SG_2007                      USABLE
P_SG_2008                      USABLE

7 rows selected.

SQL>
SQL> alter table accounting drop partition P_SG_2006;

Table altered.

SQL> select partition_name from user_tab_partitions where table_name = 'ACCOUNTING';

PARTITION_NAME
------------------------------
P_IN_2006
P_IN_2007
P_IN_2008
P_MAX
P_SG_2007
P_SG_2008

6 rows selected.

SQL> select partition_name, status from user_ind_partitions where index_name =
'ACCNTNG_DATA_NDX';

PARTITION_NAME                 STATUS
------------------------------ --------
P_IN_2006                      USABLE
P_IN_2007                      USABLE
P_IN_2008                      USABLE
P_MAX                          USABLE
P_SG_2007                      USABLE
P_SG_2008                      USABLE

6 rows selected.

SQL>
```

**Archiving Data**

The demo “SH” schema (that can be installed as part of the installation of EXAMPLES) contains a table “SALES” that shows how Date-Ranged Partitions can be created and managed.  Note how 1995 and 1996 data is stored in Year Partitions, 1997 data is stored in Half-Year Partitions and 1998 to 2003 data is stored in Quarter-Year Partitions.

```
SQL> connect sh/sh
Connected.
SQL> desc sales
 Name                                      Null?    Type
 ----------------------------------------- -------- ----------------------------
 PROD_ID                                   NOT NULL NUMBER
 CUST_ID                                   NOT NULL NUMBER
 TIME_ID                                   NOT NULL DATE
 CHANNEL_ID                                NOT NULL NUMBER
 PROMO_ID                                  NOT NULL NUMBER
 QUANTITY_SOLD                             NOT NULL NUMBER(10,2)
 AMOUNT_SOLD                               NOT NULL NUMBER(10,2)

SQL> l
  1  select partition_name, high_value
  2  from user_tab_partitions
  3  where table_name = 'SALES'
  4* order by partition_position
SQL> /

PARTITION_NAME
------------------------------
HIGH_VALUE
--------------------------------------------------------------------------------
SALES_1995
TO_DATE(' 1996-01-01 00:00:00', 'SYYYY-MM-DD HH24:MI:SS', 'NLS_CALENDAR=GREGORIA

SALES_1996
TO_DATE(' 1997-01-01 00:00:00', 'SYYYY-MM-DD HH24:MI:SS', 'NLS_CALENDAR=GREGORIA

SALES_H1_1997
TO_DATE(' 1997-07-01 00:00:00', 'SYYYY-MM-DD HH24:MI:SS', 'NLS_CALENDAR=GREGORIA

SALES_H2_1997
TO_DATE(' 1998-01-01 00:00:00', 'SYYYY-MM-DD HH24:MI:SS', 'NLS_CALENDAR=GREGORIA

SALES_Q1_1998
TO_DATE(' 1998-04-01 00:00:00', 'SYYYY-MM-DD HH24:MI:SS', 'NLS_CALENDAR=GREGORIA

SALES_Q2_1998
TO_DATE(' 1998-07-01 00:00:00', 'SYYYY-MM-DD HH24:MI:SS', 'NLS_CALENDAR=GREGORIA

SALES_Q3_1998
TO_DATE(' 1998-10-01 00:00:00', 'SYYYY-MM-DD HH24:MI:SS', 'NLS_CALENDAR=GREGORIA

SALES_Q4_1998
TO_DATE(' 1999-01-01 00:00:00', 'SYYYY-MM-DD HH24:MI:SS', 'NLS_CALENDAR=GREGORIA

SALES_Q1_1999
TO_DATE(' 1999-04-01 00:00:00', 'SYYYY-MM-DD HH24:MI:SS', 'NLS_CALENDAR=GREGORIA

SALES_Q2_1999
TO_DATE(' 1999-07-01 00:00:00', 'SYYYY-MM-DD HH24:MI:SS', 'NLS_CALENDAR=GREGORIA

SALES_Q3_1999
TO_DATE(' 1999-10-01 00:00:00', 'SYYYY-MM-DD HH24:MI:SS', 'NLS_CALENDAR=GREGORIA

SALES_Q4_1999
TO_DATE(' 2000-01-01 00:00:00', 'SYYYY-MM-DD HH24:MI:SS', 'NLS_CALENDAR=GREGORIA

SALES_Q1_2000
TO_DATE(' 2000-04-01 00:00:00', 'SYYYY-MM-DD HH24:MI:SS', 'NLS_CALENDAR=GREGORIA

SALES_Q2_2000
TO_DATE(' 2000-07-01 00:00:00', 'SYYYY-MM-DD HH24:MI:SS', 'NLS_CALENDAR=GREGORIA

SALES_Q3_2000
TO_DATE(' 2000-10-01 00:00:00', 'SYYYY-MM-DD HH24:MI:SS', 'NLS_CALENDAR=GREGORIA
SALES_Q4_2000
TO_DATE(' 2001-01-01 00:00:00', 'SYYYY-MM-DD HH24:MI:SS', 'NLS_CALENDAR=GREGORIA

SALES_Q1_2001
TO_DATE(' 2001-04-01 00:00:00', 'SYYYY-MM-DD HH24:MI:SS', 'NLS_CALENDAR=GREGORIA

SALES_Q2_2001
TO_DATE(' 2001-07-01 00:00:00', 'SYYYY-MM-DD HH24:MI:SS', 'NLS_CALENDAR=GREGORIA

SALES_Q3_2001
TO_DATE(' 2001-10-01 00:00:00', 'SYYYY-MM-DD HH24:MI:SS', 'NLS_CALENDAR=GREGORIA

SALES_Q4_2001
TO_DATE(' 2002-01-01 00:00:00', 'SYYYY-MM-DD HH24:MI:SS', 'NLS_CALENDAR=GREGORIA

SALES_Q1_2002
TO_DATE(' 2002-04-01 00:00:00', 'SYYYY-MM-DD HH24:MI:SS', 'NLS_CALENDAR=GREGORIA

SALES_Q2_2002
TO_DATE(' 2002-07-01 00:00:00', 'SYYYY-MM-DD HH24:MI:SS', 'NLS_CALENDAR=GREGORIA

SALES_Q3_2002
TO_DATE(' 2002-10-01 00:00:00', 'SYYYY-MM-DD HH24:MI:SS', 'NLS_CALENDAR=GREGORIA

SALES_Q4_2002
TO_DATE(' 2003-01-01 00:00:00', 'SYYYY-MM-DD HH24:MI:SS', 'NLS_CALENDAR=GREGORIA

SALES_Q1_2003
TO_DATE(' 2003-04-01 00:00:00', 'SYYYY-MM-DD HH24:MI:SS', 'NLS_CALENDAR=GREGORIA

SALES_Q2_2003
TO_DATE(' 2003-07-01 00:00:00', 'SYYYY-MM-DD HH24:MI:SS', 'NLS_CALENDAR=GREGORIA

SALES_Q3_2003
TO_DATE(' 2003-10-01 00:00:00', 'SYYYY-MM-DD HH24:MI:SS', 'NLS_CALENDAR=GREGORIA

SALES_Q4_2003
TO_DATE(' 2004-01-01 00:00:00', 'SYYYY-MM-DD HH24:MI:SS', 'NLS_CALENDAR=GREGORIA


28 rows selected.

SQL>
```