|  |
| --- |
| EPAM Systems, RD Dep. |
| MTN.BI.07 Partitioning |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| REVISION HISTORY | | | | | |
| Ver. | Description of Change | Author | Date | Approved | |
| Name | Effective Date |
| 1.0 | Initial status | [Kiryl Bucha](mailto:Kiryl_Bucha@epam.com) | 12-JAN-2012 |  |  |
| 2.0 | Updated in accordance with renewed content | [Elias Nema](mailto:Elias_Nema@epam.com) | 20-JAN-2014 |  |  |

Contents

[1. Partitioning Overview 3](#_Toc383292211)

[2. Concept of Partitioning 3](#_Toc383292212)

[3. Table Partitioning Schemes 5](#_Toc383292213)

[3.1. Range Partitioning 5](#_Toc383292214)

[3.2. Hash Partitioning 6](#_Toc383292215)

[3.3. List Partitioning 7](#_Toc383292216)

[3.4. Interval Partitioning 7](#_Toc383292217)

[3.5. Reference Partitioning 8](#_Toc383292218)

[3.6. Composite Partitioning 9](#_Toc383292219)

[3.7. Row Movement 10](#_Toc383292220)

[4. Partitioning and Performance, Revisited 10](#_Toc383292221)

[5. Source Books and Articles 11](#_Toc383292222)

# Partitioning Overview

Oracle Partitioning, first introduced in Oracle 8.0 in 1997, is one of the most important and successful functionalities of the Oracle database that enables large tables and indexes to be subdivided in smaller pieces, improving the performance, manageability, and availability for tens of thousands of applications. Queries and maintenance operations are sped up by an order of magnitude for mission critical systems of any shape – OLTP, data warehousing, or mixed workloads – and any size – from hundreds of Gigabytes to Petabytes. Partitioning enables database designers and administrators to tackle some of the toughest problems posed by cutting-edge applications. Partitioning is a key tool for building multi-terabyte systems or systems with extreme high availability requirements. Moreover, partitioning can greatly reduce the total cost of data ownership, using a “tiered archiving” approach of keeping older relevant information still online on low cost storage devices in the most optimal compressed format. When used together with Automatic Data Optimization and Heat Map, new functionality introduced in Oracle Database 12c, Partitioning provides a simple and automated way to implement an Information Lifecycle Management (ILM) strategy.

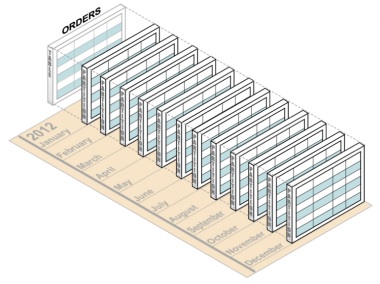
Partitioning facilitates the management of very large tables and indexes using divide and conquer logic. Partitioning introduces the concept of a partition key that is used to segregate data based on a certain range value, a list of specific values, or the value of a hash function. If I were to put the benefits of partitioning in some sort of order, it would be:

1. **Increases availability of data**: This attribute is applicable to all system types, be they OLTP or warehouse systems by nature.
2. **Eases administration** of large segments by removing them from the database: Performing administrative operations on a 100GB table, such as are organization to remove migrated rows or to reclaim “whitespace” left in the table after a purge of old information, would be much more onerous than performing the same operation ten times on individual 10GB table partitions. Additionally, using partitions, we might be able to conduct a purge routine without leaving whitespace behind at all, removing the need for are organization entirely!
3. **Improves the performance** of certain queries: This is mainly beneficial in a large warehouse environment where we can use partitioning to eliminate large ranges of data from consideration, avoiding accessing this data at all. This will not be as applicable in a transactional system, since we are accessing small volumes of data in that system already.
4. **May reduce contention** on high-volume OLTP systems by spreading out modifications across many separate partitions: If you have a segment experiencing high contention, turning it into many segments could have the side effect of reducing that contention proportionally.

# Concept of Partitioning

Partitioning enables tables and indexes to be subdivided into individual smaller pieces. Each piece of the database object is called a partition. A partition has its own name, and may optionally have its own storage characteristics. From the perspective of a database administrator, a partitioned object has multiple pieces that can be managed either collectively or individually. This gives the administrator considerable flexibility in managing a partitioned object. However, from the perspective of the application, a partitioned table is identical to a non-partitioned table; no modifications are necessary when accessing a partitioned table using SQL DML commands. Logically, it is still only one table.

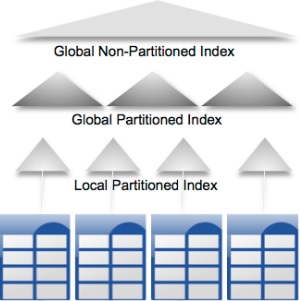
Database objects – tables and indexes - are partitioned using a partitioning **key**, a set of columns that determine in which partition a given row will reside (in case of a composite partitioned table, a partition is further subdivided into subpartitions, using a second set of columns for further subdivision; the data placement of a given row is then determined by both partitioning key criteria and placed in the appropriate subpartition). For example the Orders table shown in Figure 1 is range-partitioned on order date, using a monthly partitioning strategy; the table appears to any application as a single, 'normal' table. However, the database administrator can manage and store each monthly partition individually, potentially using different storage tiers, applying table compression to the older data, or store complete ranges of older data in read only tablespaces. Application developers generally do not have to worry about whether or not a table is partitioned, but they also can leverage partitioning to their advantage: for example a resource intensive DML operation to purge data from a table can be implemented using partition maintenance operations, improving the runtime dramatically while reducing the resource consumption significantly.



**Figure 1 Application and DBA View of Partitioned Table**

Irrespective of the chosen table partitioning strategy, any index of a partitioned table is either coupled or uncoupled with the underlying partitioning strategy of its table. Oracle Database differentiates between three types of indexes.

* A **local** index is an index on a partitioned table that is coupled with the underlying partitioned table; the index 'inherits' the partitioning strategy from the table. Consequently, each partition of a local index corresponds to one - and only one - partition of the underlying table. The coupling enables optimized partition maintenance; for example, when a table partition is dropped, Oracle simply has to drop the corresponding index partition as well. No costly index maintenance is required since an index partition is by definition only tied to its table partition; a local index segment will never contain data of other partitions. Local indexes are most common in data warehousing environments.
* A **global partitioned** index is an index on a partitioned or non-partitioned table that is partitioned using a different partitioning-key or partitioning strategy than the table. Global-partitioned indexes can be partitioned using range or hash partitioning and are uncoupled from the underlying table. For example, a table could be range-partitioned by month and have twelve partitions, while an index on that table could be hash-partitioned using a different partitioning key and have a different number of partitions. Decoupling an index from its table automatically means that any partition maintenance operation on the table can potentially cause index maintenance operations. Global partitioned indexes are more common for OLTP than for data warehousing environments.
* A **global non-partitioned** index is essentially identical to an index on a non-partitioned table. The index structure is not partitioned and uncoupled from the underlying table. In data warehousing environments, the most common usage of global non-partitioned indexes is to enforce primary key constraints. OLTP environments on the other hand mostly rely on global non-partitioned indexes.



**Figure 2 Indexing on Partitioned Tables**

# Table Partitioning Schemes

* **Range partitioning:** You may specify ranges of data that should be stored together. For example, everything that has a timestamp within the month of Jan-2010 will be stored in partition 1, everything with a timestamp within Feb-2010 in partition 2, and so on. This is probably the most commonly used partitioning mechanism in Oracle.
* **Hash partitioning:** You saw this in the first example in this chapter. A column (or columns) has a hash function applied to it, and the row will be placed into a partition according to the value of this hash.
* **List partitioning:** You specify a discrete set of values, which determines the data that should be stored together. For example, you could specify that rows with a STATUS column value in ( 'A', 'M', 'Z' ) go into partition 1, those with a STATUS value in ( 'D', 'P', 'Q' ) go into partition 2, and so on.
* **Interval partitioning:** This is very similar to range partitioning with the exception that the database itself can create new partitions as data arrives. With traditional range partitioning, the DBA was tasked with pre-creating partitions to hold every possible data value, for now and into the future. This typically meant that a DBA was tasked with creating partitions on a schedule—to hold next months’ or next weeks’ data. With interval partitioning, the database itself will create partitions as new data arrives that doesn’t fit into any existing partition based on a rule specified by the DBA.
* **Reference partitioning:** This allows a child table in a parent/child relationship enforced by a foreign key to inherit the partitioning scheme of the parent table. This makes it possible to equipartition a child table with its parent table without having to de-normalize the data model. In the past, a table could only be partitioned based on attributes it physically stored; reference partitioning in effect allows you to partition a table based on attributes from its parent table.
* **Composite partitioning:** This is a combination of range, hash, and list partitioning. It allows you to first apply one partitioning scheme to some data, and then within each resulting partition have that partition subdivided into subpartitions using some partitioning scheme.

## Range Partitioning

The first type we will look at is a range partitioned table. The following CREATE TABLE statement creates a range partitioned table using the column RANGE\_KEY\_COLUMN.

# CREATE TABLE range\_example

( range\_key\_column date NOT NULL,

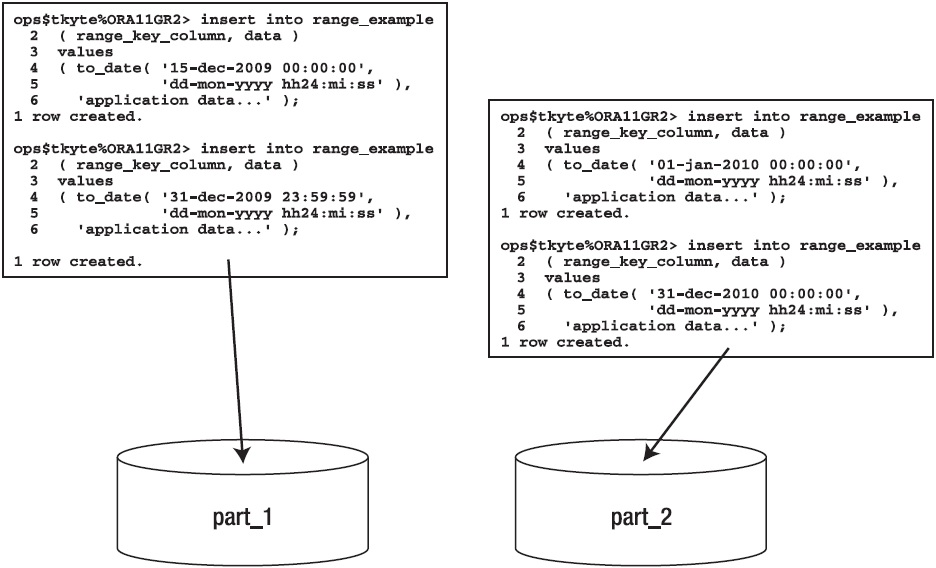
data varchar2(20))

PARTITION BY RANGE (range\_key\_column)

( PARTITION part\_1 VALUES LESS THAN(to\_date('01/01/2010','dd/mm/yyyy')),

PARTITION part\_2 VALUES LESS THAN(to\_date('01/01/2011','dd/mm/yyyy')));

All data with a RANGE\_KEY\_COLUMN strictly less than 01-JAN-2010 will be placed into the partition PART\_1, and all data with a value strictly less than 01-JAN-2011 will go into partition PART\_2. Any data not satisfying either of those conditions (e.g., a row with a RANGE\_KEY\_COLUMN value of 01-JAN-2012) will fail upon insertion, as it cannot be mapped to a partition.



**Figure 3 Range Partition Insert Example**

## Hash Partitioning

When hash partitioning a table, Oracle will apply a hash function to the partition key to determine in which of the N partitions the data should be placed. Oracle recommends that N be a number that is a power of 2 (2, 4, 8, 16, and so on) to achieve the best overall distribution, and we’ll see shortly that this is absolutely good advice.

Hash partitioning is designed to achieve a good spread of data across many different devices (disks), or just to segregate data out into more manageable chunks. The hash key chosen for a table should be a column or set of columns that are unique, or at least have as many distinct values as possible to provide for a good spread of the rows across partitions. If you choose a column that has only four values, and you use two partitions, then all the rows could quite easily end up hashing to the same partition, obviating the goal of partitioning in the first place!

We will create a hash table with two partitions in this case. We will use a column named HASH\_KEY\_COLUMN as our partition key. Oracle will take the value in this column and determine the partition this row will be stored in by hashing that value:

#CREATE TABLE hash\_example

( hash\_key\_column date,

data varchar2(20)

)

PARTITION BY HASH (hash\_key\_column)

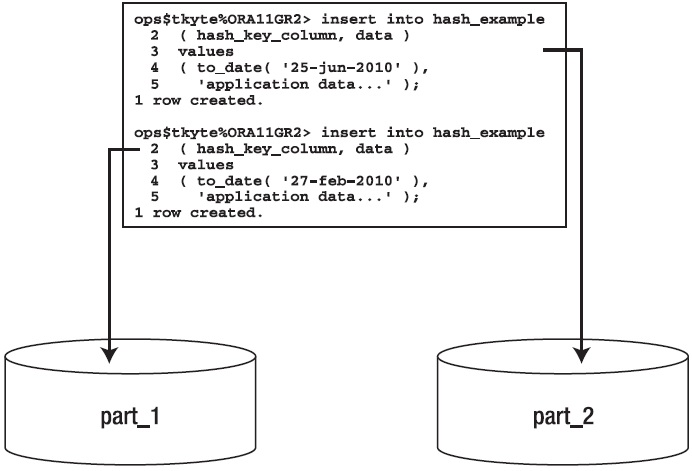
(

partition part\_1 tablespace p1,

partition part\_2 tablespace p2

);

As noted earlier, hash partitioning gives you no control over which partition a row ends up in. Oracle applies the hash function and the outcome of that hash determines where the row goes. If you want a specific row to go into partition PART\_1 for whatever reason, you should not—in fact, you cannot—use hash partitioning. The row will go into whatever partition the hash function says to put it in. If you change the number of hash partitions, the data will be redistributed over all of the partitions(adding or removing a partition to a hash partitioned table will cause all of the data to be rewritten, as every row may now belong in a different partition).Hash partitioning is most useful when you have a large table and you would like to divide and conquer it. Rather than manage one large table, you would like to have 8 or 16 smaller tables to manage. Hash partitioning is also useful to increase availability to some degree, the temporary loss of a single hash partition permits access to all of the remaining partitions. Some users may be affected, but there is a good chance that many will not be. Additionally, the unit of recovery is much smaller now. You do not have a single large table to restore and recover; you have a fraction of that table to recover. Lastly, hash partitioning is useful in high update contention environments. Instead of having a single hot segment, we can hash partition a segment into 16 pieces, each of which is now receiving modifications.



**Figure 4 Hash Partition Insert Example**

## List Partitioning

List partitioning was a new feature of Oracle9i Release 1. It provides the ability to specify in which partition a row will reside, based on discrete lists of values. It is often useful to be able to partition by some code, such as a state or region code. For example, we might want to pull together in a single partition all records for people in the states of Maine (ME), New Hampshire (NH), Vermont (VT), and Massachusetts (MA), since those states are located next to or near each other and our application queries data by geographic region. Similarly, we might want to group together Connecticut (CT), Rhode Island (RI), and New York (NY).

We can’t use a range partition, since the range for the first partition would be ME through VT, and the second range would be CT through RI. Those ranges overlap. We can’t use hash partitioning since we can’t control which partition any given row goes into; the built-in hash function provided by Oracle does that.

With list partitioning, we can accomplish this custom partitioning scheme easily:

# create table list\_example

( state\_cd varchar2(2),

data varchar2(20)

)

partition by list(state\_cd)

( partition part\_1 values ( 'ME', 'NH', 'VT', 'MA' ),

partition part\_2 values ( 'CT', 'RI', 'NY' )

);



**Figure 5 List Partition Insert Example**

## Interval Partitioning

Interval partitioning is a new feature available in Oracle Database 11g Release 1 and above. It is very similar to range partitioning described previously—in fact, it starts with a range partitioned table but adds a rule (the interval) to the definition so the database knows how to add partitions in the future.

The goal of interval partitioning is to create new partitions for data—if, and only if, data exists for a given partition and only when that data arrives in the database. In other words, to remove the need to pre-create partitions for data, to allow the data it to create the partition as it is inserted.

To use interval partitioning, you start with a range partitioned table without a MAXVALUE partition and specify an interval to add to the upper bound, the highest value of that partitioned table to create a new range. You need to have a table that is range partitioned on a single column that permits adding a NUMBER or INTERVAL type to it (e.g. a table partitioned by a VARCHAR2 field cannot be interval partitioned; there is partitioned table; that is, you can ALTER an existing range table to be interval partitioned, or you can create one with the CREATE TABLE command.

For example, suppose you had a range partitioned table that said “anything strictly less than 01-JAN-2010 (data in the year 2009 and before) goes into partition P1—and that was it. So it had one partition for all data in the year 2009 and before. If you attempted to insert data for the year 2010 into the table, the insert would fail as demonstrated previously in the section on range partitioning. With interval partitioning you can create a table and specify both a range (strictly less than 01-JAN-2010) and an interval—say 1 month in duration—and the database would create monthly partitions (a partition capable of holding exactly one month’s worth of data) as the data arrived. The database would not precreate all possible partitions because that would not be practical. But, as each row arrived the database would see whether the partition for the month in question existed. The database would create the partition if needed.

Here is an example of the syntax:

# create table audit\_trail

( ts timestamp,

data varchar2(30)

)

partition by range(ts)

interval (numtoyminterval(1,'month'))

store in (users, example )

(

partition p0 values less than(to\_date('01-01-1900','dd-mm-yyyy'))

);

## Reference Partitioning

Reference partitioning is a new feature of Oracle Database 11g Release 1 and above. It addresses the issue of parent/child equi-partitioning; that is, when you need the child table to be partitioned in such a manner that each child table partition has a one-to-one relationship with a parent table partition. This is important in situations such as a data warehouse where you want to keep a specific amount of data online (say the last five years’ worth of ORDER information) and need to ensure the related child data (the ORDER\_LINE\_ITEMS data) is online as well. In this classic example, the ORDERS table would typically have a column ORDER\_DATE, making it easy to partition by month and thus facilitate keeping the last five years of data online easily. As time advances, you would just have next month’s partition available for loading and you would drop the oldest partition. However, when you consider the ORDER\_LINE\_ITEMS table, you can see you would have a problem. It does not have the ORDER\_DATE column, there is nothing in the ORDER\_LINE\_ITEMS table to partition it by; therefore, and it’s not facilitating the purging of old information or loading of new information.

In the past, prior to reference partitioning, developers would have to denormalize the data, in effect copying the ORDER\_DATE attribute from the parent table ORDERS into the child ORDER\_LINE\_ITEMS table. This presented the typical problems of data redundancy, that of increased storage overhead, increased data loading resources, cascading update issues (if you modify the parent, you have to ensure you update all copies of the parent data) and so on. Additionally, if you enabled foreign key constraints in the database (as you should!), you would discover that you lost the ability to truncate or drop old partitions in the parent table. For example, let’s set up the conventional ORDERS and ORDER\_LINE\_ITEMS tables starting with the ORDERS table:

# create table orders

(

order# number primary key,

order\_date date,

data varchar2(30)

)

enable row movement

PARTITION BY RANGE (order\_date)

(

PARTITION part\_2009 VALUES LESS THAN (to\_date('01-01-2010','dd-mm-yyyy')) ,

PARTITION part\_2010 VALUES LESS THAN (to\_date('01-01-2011','dd-mm-yyyy'))

);

And now we’ll create the ORDER\_LINE\_ITEMS table – with a bit of data pointing to the ORDERS table:

# create table order\_line\_items

(

order# number,

line# number,

order\_date date, -- manually copied from ORDERS!

data varchar2(30),

constraint c1\_pk primary key(order#,line#),

constraint c1\_fk\_p foreign key(order#) references orders

)

enable row movement

PARTITION BY RANGE (order\_date)

(

PARTITION part\_2009 VALUES LESS THAN (to\_date('01-01-2010','dd-mm-yyyy')) ,

PARTITION part\_2010 VALUES LESS THAN (to\_date('01-01-2011','dd-mm-yyyy'))

);

Now, if we were to drop the ORDER\_LINE\_ITEMS partition containing 2009 data, you know and all knows that the corresponding ORDERS partition for 2009 could be dropped as well, without violating the referential integrity constraint. You and I know it, but the database is not aware of that fact:

# ERROR at line x:

ORA-02266: unique/primary keys in table referenced by enabled foreign keys

The simple syntax to re-implement this example from above could be as follows; we’ll reuse the existing parent table ORDERS and just truncate that table and create a new child table:

# create table order\_line\_items

(

order# number,

line# number,

data varchar2(30),

constraint c1\_pk primary key(order#,line#),

constraint c1\_fk\_p foreign key(order#) references orders

)

enable row movement

partition by reference(c1\_fk\_p);

## Composite Partitioning

Lastly, we’ll look at some examples of composite partitioning, which is a mixture of range, hash, and/or list. The methods by which you can composite partition, that is the types of partitioning schemes you can mix and match, varies by release. Table lists what is available in each of the major releases. The partitioning scheme listed down the table is the top level partitioning scheme permitted, whereas as the partitioning scheme listed across the table is the subpartition—the partition within the partition—scheme.

|  |  |  |  |
| --- | --- | --- | --- |
|  | Range | List | Hash |
| Range | **11g Release 1** | **9i Release 2** | **9i Release 1** |
| List | **11g Release 1** | **11g Release 1** | **11g Release 1** |
| Hash | **11g Release 2** | **11g Release 2** | **11g Release 2** |

So, for example, in Oracle 9i Release 2 and later you can partition a table by RANGE and then within each range partition, by LIST or HASH. Starting in Oracle Database 11g Release 1 and above, you go from two composite schemes to six. And in Oracle Database 11g Release 2 and later, you have nine to choose from.

# CREATE TABLE composite\_example

( range\_key\_column date,

hash\_key\_column int,

data varchar2(20)

)

PARTITION BY RANGE (range\_key\_column)

subpartition by hash(hash\_key\_column) subpartitions 2

(

PARTITION part\_1 VALUES LESS THAN(to\_date('01/01/2008','dd/mm/yyyy'))

(

subpartition part\_1\_sub\_1,

subpartition part\_1\_sub\_2

),

PARTITION part\_2VALUES LESS THAN(to\_date('01/01/2011','dd/mm/yyyy'))

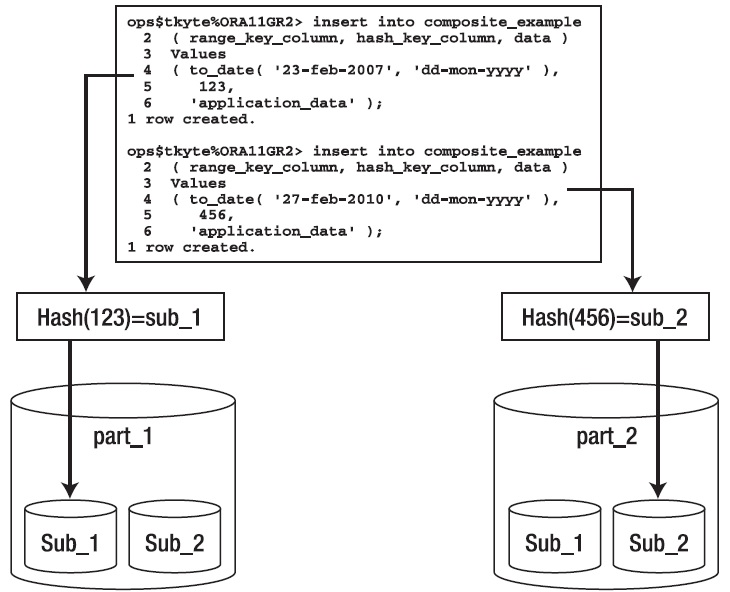
(

subpartition part\_2\_sub\_1,

subpartition part\_2\_sub\_2

)

);



**Figure 6 Composite Partitioning Insert**

## Row Movement

You might wonder what would happen if the column used to determine the partition is modified in any of the preceding partitioning schemes. There are two cases to consider:

* The modification would not cause a different partition to be used; the row would still belong in this partition. This is supported in all cases.
* The modification would cause the row to move across partitions. This is supported if row movement is enabled for the table; otherwise, an error will be raised.

Update immediately raises an error since we did not explicitly enable row movement. In Oracle8iand later releases, we can enable row movement on this table to allow the row to move from partition to partition.

ERROR at line 1:

ORA-14402: updating partition key column would cause a partition change

**Note**: The row movement functionality is not available on Oracle 8.0; you must delete the row and reinsert it in that release.

# Partitioning and Performance, Revisited

People said, “I’m very disappointed in partitioning. We partitioned our largest table and it went much slower. So much for partitioning being a performance increasing feature!” Partitioning can do one of the following three things to overall query performance:

* Make your queries go faster
* Not impact the performance of your queries at all
* Make your queries go much slower and use many time the resources as the no partitioned implementation

In a data warehouse, with an understanding of the questions being asked of the data, the first bullet point is very much achievable. Partitioning can positively impact queries that frequently full scan large database tables by eliminating large sections of data from consideration. Suppose you have a table with1 billion rows in it. There is a timestamp attribute. Your query is going to retrieve one years’ worth of data from this table (and it has 10 years of data). Your query uses a full table scan to retrieve this data. Had it been partitioned by this timestamp entry—say a partition per month—then you could have full scanned one-tenth the data (assuming a uniform distribution of data over the years). Partition elimination would have removed the other 90 percent of the data from consideration. Your query would likely run faster.

The placement of a given row is determined by its value of the partitioning key. How the data of a table is subdivided across the partitions is stored as partitioning metadata of a table or index. This metadata is used to determine for every SQL operation – queries, DML, and partition maintenance operations - what partitions of a table are relevant, and the database automatically only touches relevant partitions. By limiting the amount of data to be examined or operated on, partitioning provides a number of performance benefits.

**Partitioning pruning** (a.k.a. partition elimination) is the simplest and also the most effective means to improve performance. It can often improve query performance by several orders of magnitude by leveraging the partitioning metadata to only touch the data of relevance for a SQL operation. For example, suppose an application contains an Orders table containing an historical record of orders, and that this table has been partitioned by day. A query requesting orders for a single week would only access seven partitions of the Orders table. If the table had 2 years of historical data, this query would access seven partitions instead of 730 partitions. This query could potentially execute 100x faster simply because of partition pruning. Partition pruning works with all of Oracle's other performance features. Oracle will utilize partition pruning in conjunction with any indexing technique, join technique, or parallel access method.

Partitioning can also improve the performance of multi-table joins, by using a technique known as **partition-wise joins**. Partition-wise joins can be applied when two tables are being joined together, and at least one of these tables is partitioned on the join key. Partition-wise joins break a large join into smaller joins of 'identical' data sets for the joined tables. 'Identical' here is defined as covering exactly the same set of partitioning key values on both sides of the join, thus ensuring that only a join of these 'identical' data sets will produce a result and that other data sets do not have to be considered. Oracle is using either the fact of already (physical) equi-partitioned tables for the join or is transparently redistributing (= “repartitioning”) one table – the smaller one - at runtime to create equi-partitioned data sets matching the partitioning of the other table, completing the overall join in less time, using less resources. This offers significant performance benefits both for serial and parallel execution.

**Note:** You can read more about partition-wise joins [here](http://docs.oracle.com/cd/E11882_01/server.112/e25523/part_avail.htm#CIHCDBIF).

# Source Books and Articles

1. Kyte T. Expert Oracle Database Architecture: Oracle Database 9i, 10g, and 11g Programming Techniques and Solutions, Second Edition. Apress, 2010.
2. Morton K., & Osborne K., & Sands R., & Shamsudeen R., & Still J. Pro Oracle SQL. Apress, 2013.
3. Powell G. Oracle Data Warehouse Tuning for 10g. Oxford: Elsevier Digital Press, 2005.
4. Lane, P. Oracle Database Data Warehousing Guide, 11g Release 2 (11.2). Redwood City: Oracle, 2013.
5. Kimball R., & Ross M. The Data Warehouse Toolkit: The Definitive Guide to Dimensional Modeling, Third Edition. Indianapolis: John Wiley & Sons, 2013.