Partitioning

Introduction to Partitioning

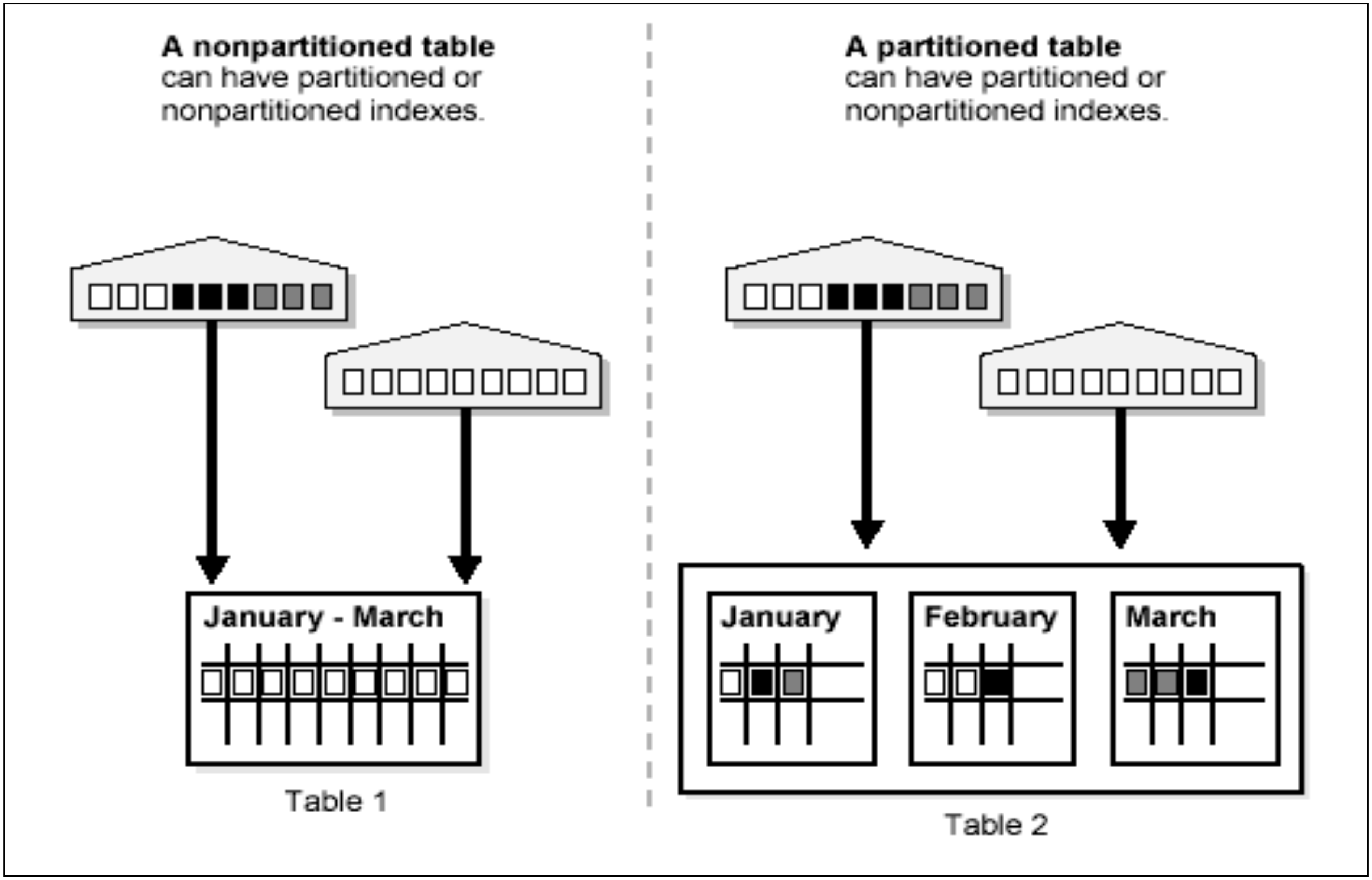
- **Partitioning** lets you decompose large tables and indexes into smaller and more manageable pieces called **partitions**.
- SQL queries and DML statements do not need to be modified in order to access partitioned tables.
- However, once partitions are defined, DDL statements can access and manipulate individuals partitions rather than entire tables or indexes.
- Each partition of a table or index must have the same logical attributes, such as column names, datatypes, and constraints, but each partition can have separate physical attributes such as pctfree, pctused, and tablespaces.
- OLTP systems often benefit from improvements in *manageability* and *availability*, while data warehousing systems benefit from *performance* and *manageability*.

Introduction to Partitioning

Partitioning offers these advantages:

- It *enables data management operations* such data loads, index creation and rebuilding, and backup/recovery at the partition level, rather than on the entire table. This results in significantly *reduced times* for these operations.
- It *improves query performance*. In many cases, the results of a query can be achieved by accessing a subset of partitions, rather than the entire table.
- It can significantly *reduce the impact of scheduled downtime* for maintenance operations. Partition independence for partition maintenance operations lets you perform concurrent maintenance operations on different partitions of the same table or index.
- Partitioning *increases the availability* of mission-critical databases
- Partitioning can be implemented *without requiring any modifications to your applications*.

A View of Partitioned Tables


A nonpartitioned table can have partitioned or nonpartitioned indexes.
A partitioned table can have partitioned or nonpartitioned indexes.
January - March
Table 1
January
February
March
Table 2

Introduction to Partitioning

Partition Key

- The partition key is a set of one or more columns that determines the partition for each row.
- Each row in a partitioned table is unambiguously assigned to a single partition.
- Oracle9*i* automatically directs insert, update, and delete operations to the appropriate partition through the use of the partition key.
- A partition key:
  - consists of an ordered list of 1 to 16 columns
  - cannot contain a `LEVEL, ROWID,` or `MLSLABEL` pseudocolumn or a column of type `ROWID`
  - can contain columns which all the `NULL` value

Introduction to Partitioning

Partitioned Tables

- Tables can be partitioned into any number of separate partitions.
- Any table can be partitioned except those tables containing columns with `LONG` or `LONG RAW` datatypes.

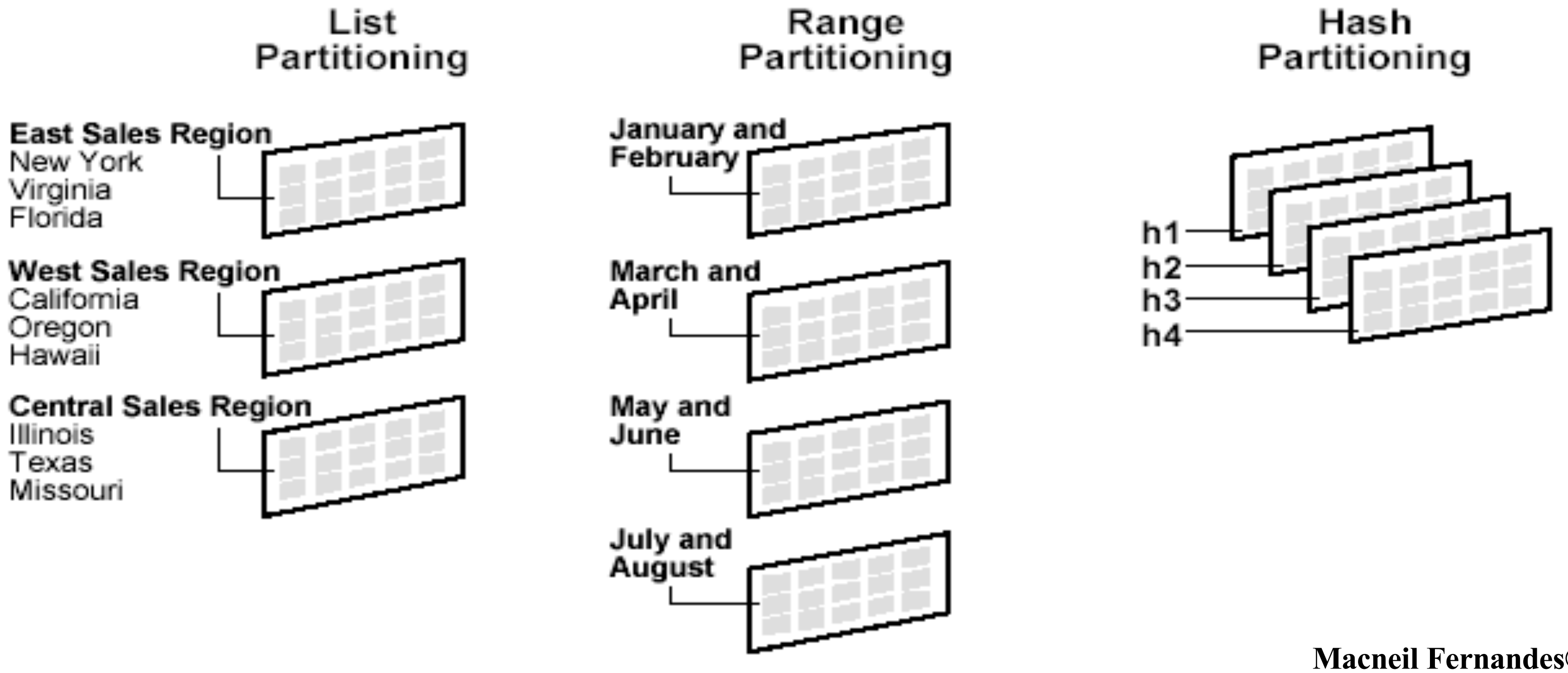
Partitioned Index-Organized Tables

- Only range and hash partitioning are supported
- Partition columns must be a subset of primary key columns
- Secondary indexes can be partitioned — locally and globally
- `OVERFLOW` data segments are always equipartitioned with the table partitions

Partitioning Methods

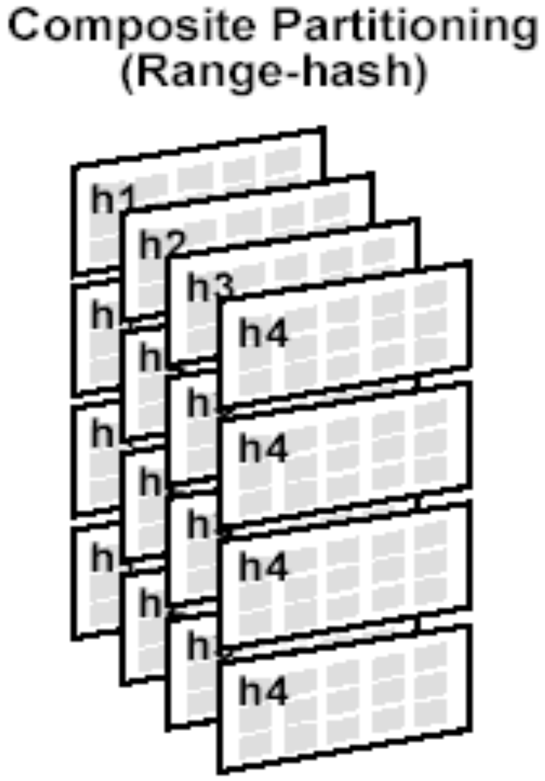
Oracle provides the following partitioning methods:

- Range Partitioning
- List Partitioning
- Hash Partitioning
- Composite Partitioning



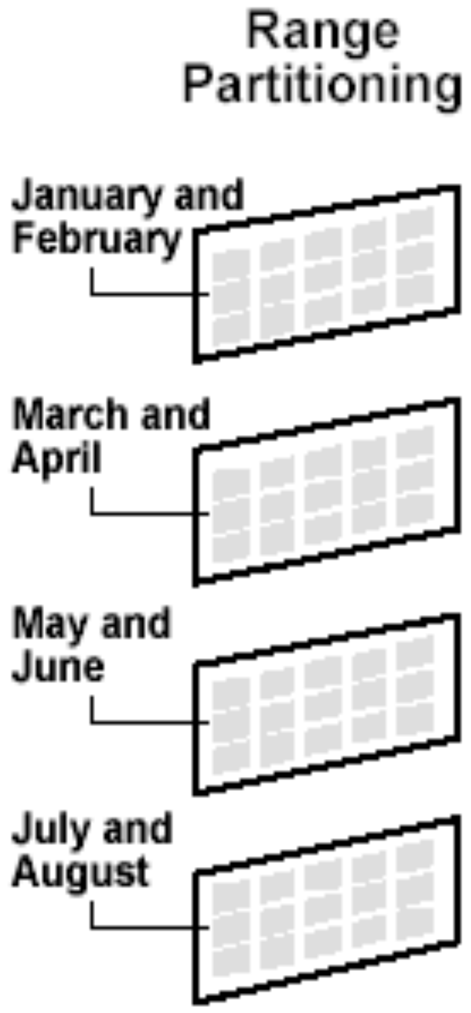
Partitioning Methods

- Composite partitioning is a combination of other partitioning methods. Oracle currently supports range-hash composite partitioning.



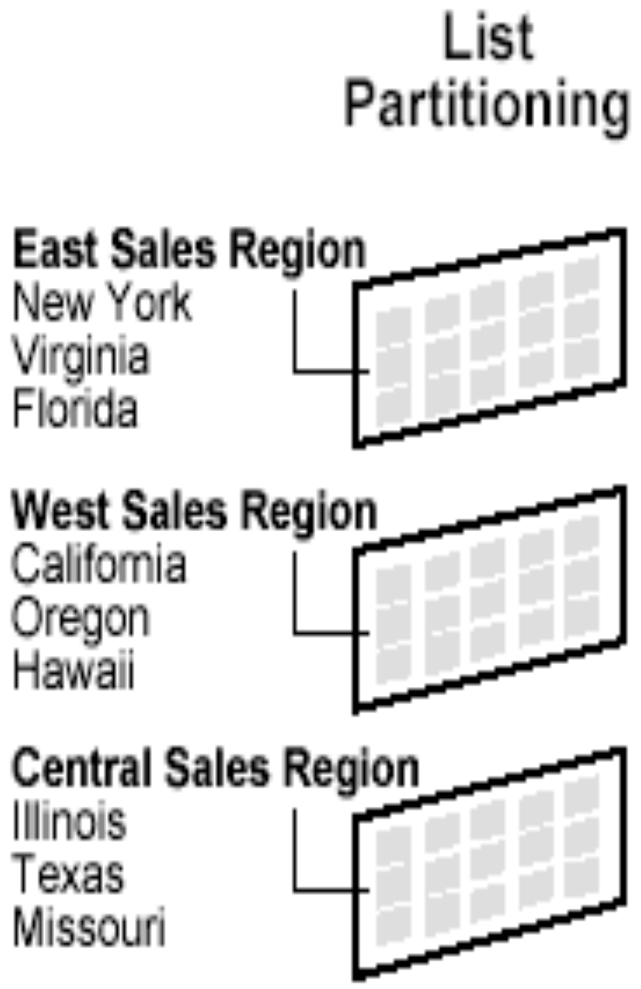
Range Partitioning

- Range partitioning maps data to partitions based on ranges of partition key values that you establish for each partition.
- It is the most common type of partitioning and is often used with dates.
- When using range partitioning, there are a few rules to keep in mind:
  - Each partition has a `VALUES LESS THAN` clause, which specifies a noninclusive upper bound for the partitions. Any binary values of the partition key equal to or higher than this literal are added to the next higher partition.
  - All partitions, except the first, have an implicit lower bound specified by the `VALUES LESS THAN` clause on the previous partition.
  - A `MAXVALUE` literal can be defined for the highest partition.



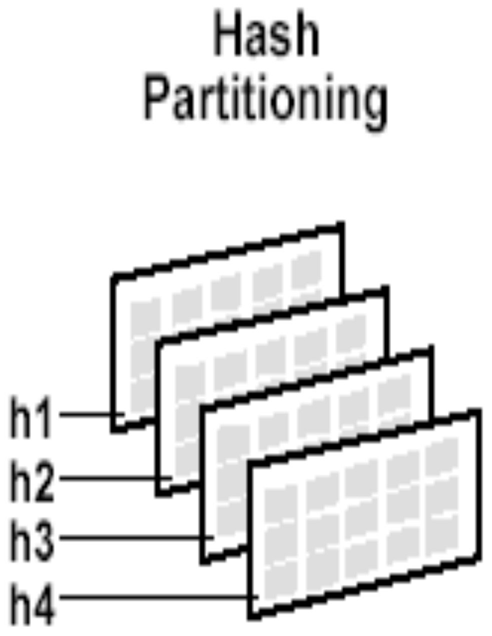
List Partitioning

- List partitioning enables you to explicitly control how rows map to partitions.
- You do this by specifying a list of discrete values for the partitioning key in the description for each partition.
- The advantage of list partitioning is that you can group and organize unordered and unrelated sets of data in a natural way.
- A row is mapped to a partition by checking whether the value of the partitioning column for a row falls within the set of values that describes the partition.



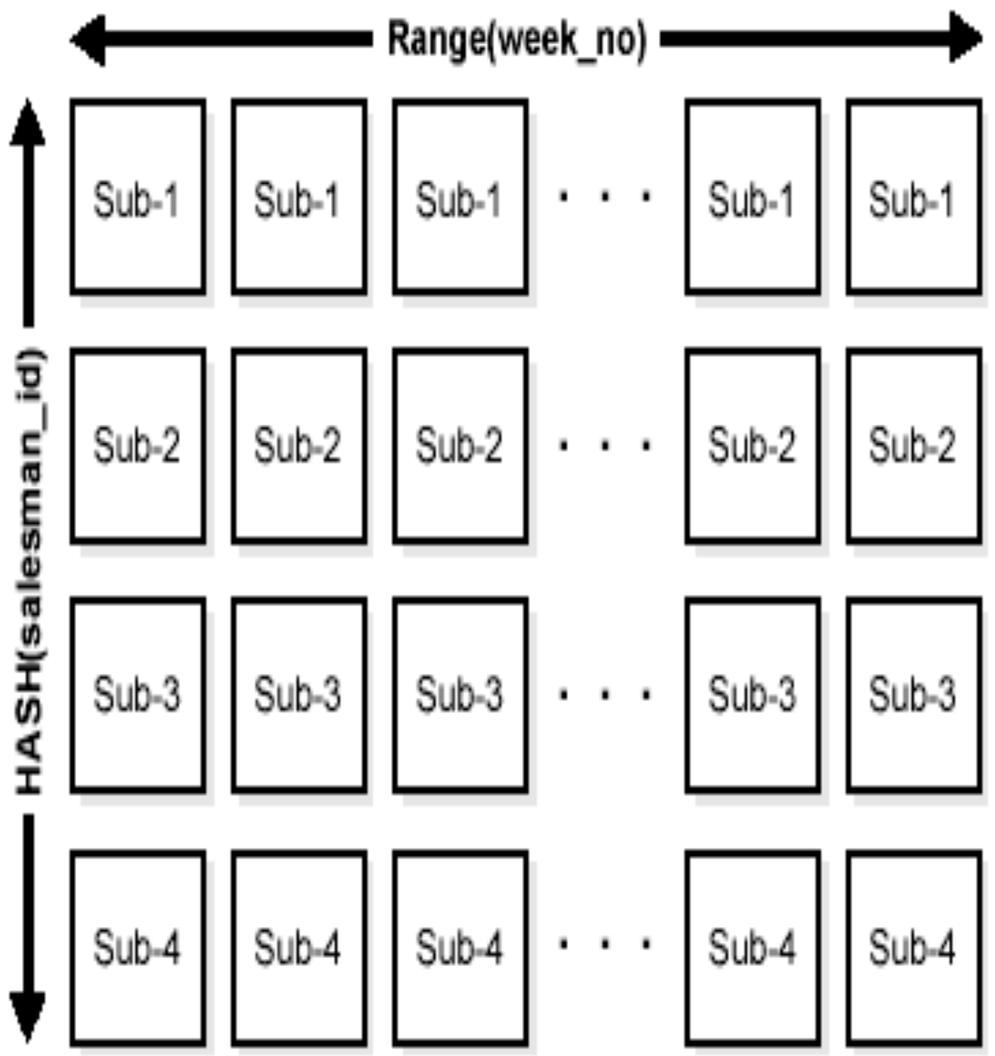
Hash Partitioning

- Hash partitioning enables easy partitioning of data that does not lend itself to range or list partitioning.
- The concepts of splitting, dropping or merging partitions do not apply to hash partitions. Instead, hash partitions can be added and coalesced.
- It is a better choice than range partitioning when:
  - You do not know beforehand how much data will map into a given range
  - The sizes of range partitions would differ quite substantially or would be difficult to balance manually
  - Range partitioning would cause the data to be undesirably clustered
  - Performance features such as parallel DML, partition pruning, and partition-wise joins are important



Composite Partitioning

- Composite partitioning partitions data using the range method, and within each partition, subpartition it using the hash method.
- Composite partitioning provides the improved manageability of range partitioning and the data placement, striping, and parallelism advantages of hash partitioning.



When to Partition a Table

Here are some suggestions for when to partition a table:

- Tables *greater than 2GB* should always be considered for partitioning.
- Tables on which you want to perform *parallel DML operations* must be partitioned.
- Tables *containing historical data*, in which new data is added into the newest partition. A typical example is a historical table where only the current month's data is updatable and the other 11 months are read-only.

Partitioned Indexes

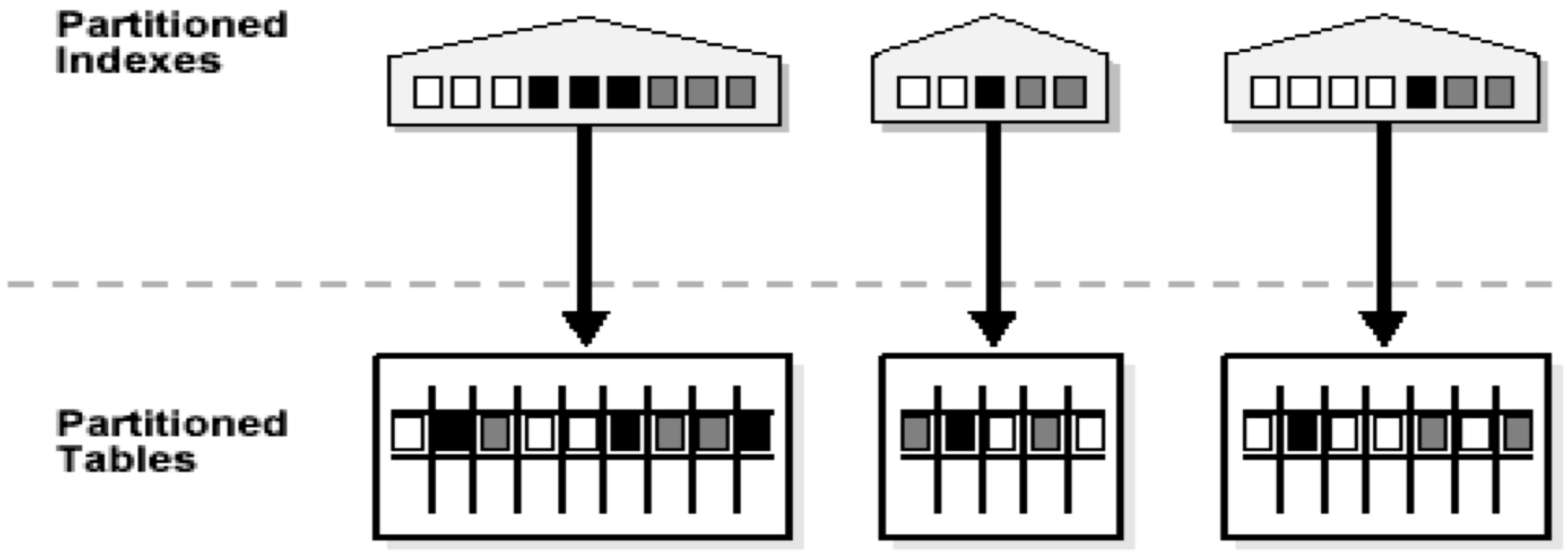
- Just like partitioned tables, partitioned indexes improve manageability, availability, performance, and scalability.
- They can either be partitioned independently (global indexes) or automatically linked to a table's partitioning method (local indexes).

Local Partitioned Indexes

- It follows equipartitioning: each partition of a local index is associated with exactly one partition of the table.
- This enables Oracle to automatically keep the index partitions in sync with the table partitions, and makes each table-index pair independent.
- They are easier to manage, offer greater availability and are common in DSS environments.

Partitioned Indexes

- You cannot explicitly add a partition to a local index. Instead, new partitions are added to local indexes only when you add a partition to the underlying table.
- Likewise, you cannot explicitly drop a partition from a local index. Instead, local index partitions are dropped only when you drop a partition from the underlying table.



Partitioned Indexes

Global Partitioned Indexes

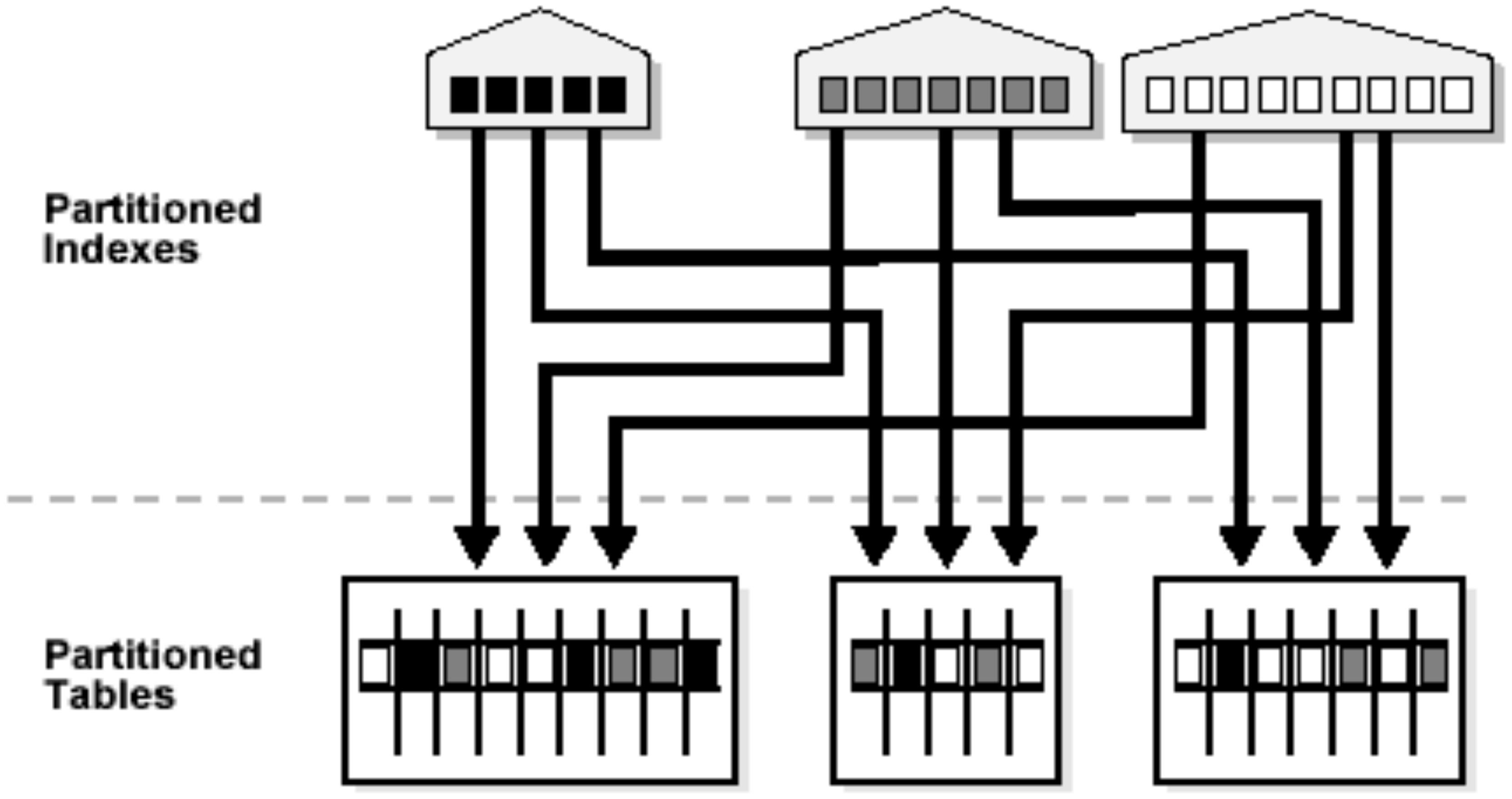
- Global partitioned indexes are flexible in that the degree of partitioning and the partitioning key are independent from the table's partitioning method.
- The highest partition of a global index must have a partition bound, all of whose values are `MAXVALUE`.
- You cannot add a partition to a global index because the highest partition always has a partition bound of `MAXVALUE`. If you wish to add a new highest partition, use the `ALTER INDEX SPLIT PARTITION` statement.
- ***If a global index partition contains data, dropping the partition causes the next highest partition to be marked unusable. You cannot drop the highest partition in a global index.***

Partitioned Indexes

- **Maintenance of Global Partitioned Indexes**
  - By default, the following operations on partitions on a heap-organized table mark all global indexes as unusable:
    - ➢ `ADD (HASH)`
    - ➢ `COALESCE (HASH)`
    - ➢ `DROP`
    - ➢ `EXCHANGE`
    - ➢ `MERGE`
    - ➢ `MOVE`
    - ➢ `SPLIT`
    - ➢ `TRUNCATE`
- These indexes can be maintained by appending the clause `UPDATE GLOBAL INDEXES` to the SQL statements for the operation.
- Advantages to maintaining global indexes: The index remains available and online throughout the operation AND The index doesn’t have to be rebuilt after the operation.

Partitioned Indexes

Figure 12–6 Global Partitioned Index

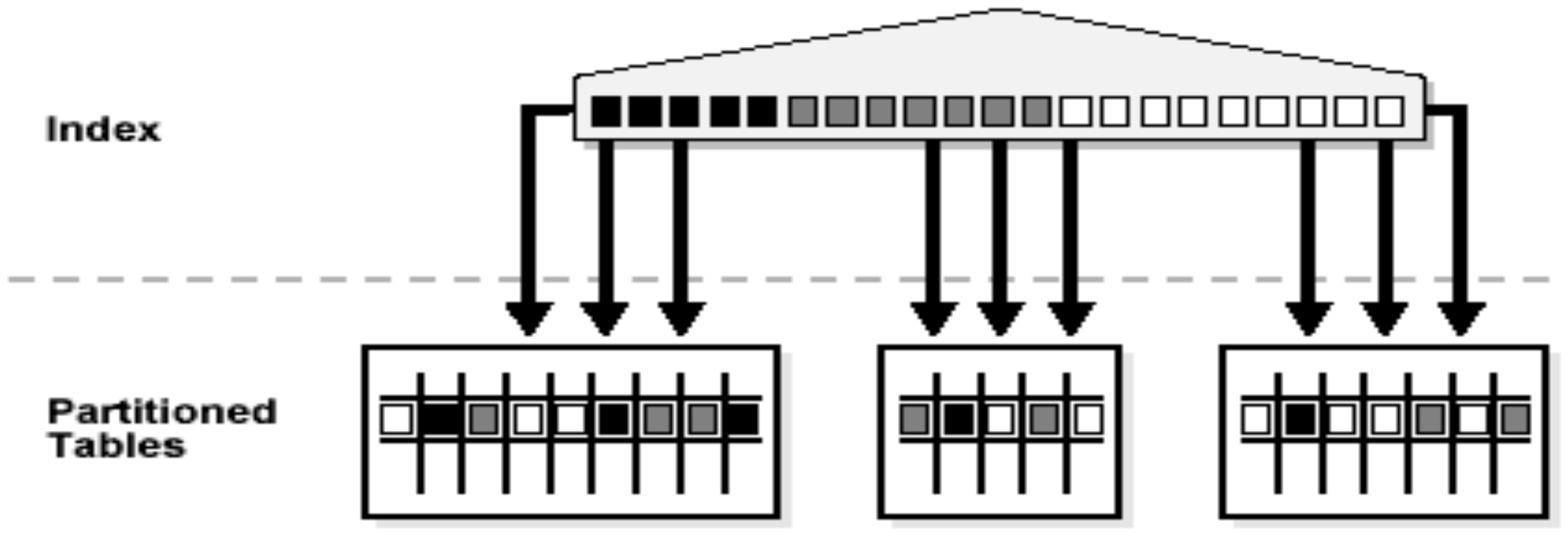


Partitioned Indexes

Global Nonpartitioned Indexes

- Global nonpartitioned indexes behave just like a nonpartitioned index.
- They are commonly used in OLTP environments and offer efficient access to any individual record.

Figure 12–7 Global Nonpartitioned Index



Partitioned Indexes

Miscellaneous Information about Creating Indexes on Partitioned Tables

- You can create bitmap indexes on partitioned tables, with the restriction that the bitmap indexes must be local to the partitioned table. They cannot be global indexes.
- Global indexes can be unique. Local indexes can only be unique if the partitioning key is a part of the index key.

Here are a few guidelines for OLTP applications:

- Global indexes and unique, local indexes provide better performance than nonunique local indexes because they minimize the number of index partition probes.
- Local indexes offer better availability when there are partition or subpartition maintenance operations on the table.

Partitioned Indexes

Here are a few guidelines for DSS applications:

- Local indexes are preferable because they are easier to manage during data loads and during partition-maintenance operations.
- Local indexes can improve performance because many index partitions can be scanned in parallel by range queries on the index key.

Here are a few points to remember when using partitioned indexes on composite partitions:

- Only range partitioned global indexes are supported.
- Subpartitioned indexes are always local and stored with the table subpartition by default.
- Tablespaces can be specified at either index or index subpartition levels.

Partitioning to Improve Performance

Partition Pruning

- The Oracle server explicitly recognizes partitions and subpartitions. It then optimizes SQL statements to mark the partitions or subpartitions that need to be accessed and eliminates (prunes) unnecessary partitions or subpartitions from access by those SQL statements.
- Partition pruning is the skipping of unnecessary index and data partitions or subpartitions in a query.
- Such intelligent pruning can dramatically reduce the data volume, resulting in substantial improvements in query performance.
- If the optimizer determines that the selection criteria used for pruning are satisfied by all the rows in the accessed partition or subpartition, it removes those criteria from the predicate list (`WHERE` clause)

Partitioning to Improve Performance

- However, the optimizer cannot prune partitions if the SQL statement applies a function to the partitioning column (with the exception of the `TO_DATE` function). Similarly, the optimizer cannot use an index if the SQL statement applies a function to the indexed column, unless it is a function-based index.
- Equality, inequality `(<, >, between)` and `IN-list` predicates are considered for partition pruning with range partitioning, and equality and IN-list predicates are considered for partition pruning with hash partitioning.

Partition-wise Joins

- A partition-wise join is a join optimization that you can use when joining two tables that are both partitioned along the join column(s).

Partitioning to Improve Performance

- With partition-wise joins, the join operation is broken into smaller joins that are performed sequentially or in parallel.
- Another way of looking at partition-wise joins is that they minimize the amount of data exchanged among parallel slaves during the execution of parallel joins by taking into account data distribution.

Parallel DML

- Parallel execution dramatically reduces response time.
- You can use parallel query and parallel DML with range- and hash-partitioned tables. By doing so, you can enhance scalability and performance for batch operations.