Indexes and Partitioned Indexes

Indexes are optional structures associated with tables that allow SQL statements to execute more quickly against a table. Even though table scans are very common in many data warehouses, indexes can often speed up queries. Partitioning can enhance data access and improve overall data warehouse performance that access tables and indexes with millions of rows and many gigabytes of data.
Partitioned tables and indexes facilitate administrative operations by enabling these operations to work on subsets of data. For example, you can add a new partition, organize an existing partition, or drop a partition and cause less than a second of interruption to a read-only application.

Types of partitioned indexes:

- Local Partitioned Indexes
- Non-partitioned Indexes
- Global Partitioned Indexes

1- Local Partitioned Indexes

In a local index, all keys in a particular index partition refer only to rows stored in a single underlying table partition. A local index is equipartitioned with the underlying table. Database partitions the index on the same columns as the underlying table, creates the same number of partitions or sub-partitions, and gives them the same partition boundaries as corresponding partitions of the underlying table.

For data warehouse applications, local non-prefixed indexes can improve performance because many index partitions can be scanned in parallel by range queries on the index key. Bitmap indexes use a very efficient storage mechanism for low cardinality columns.

Bitmap indexes are used in data warehouses, and especially common in data warehouses that implement *star schemas*. Foreign key columns in the fact table are ideal candidates for bitmap indexes, because generally there are few distinct values relative to the total number of rows. Fact tables are often range or range-partitioned, in which case you must create local bitmap indexes. Bitmap indexing provides:

- Bitmap indexes are most effective for queries that contain multiple conditions in the WHERE clause
- Reduced response time for large classes of ad hoc queries.
- Reduced storage requirements compared to other indexing techniques.
- Dramatic performance gains even on hardware with a relatively small number of CPUs.
- Efficient maintenance during parallel DML and loads.

A B-tree index is organized like an upside-down tree. The bottom level of the index holds the actual data values and pointers to the corresponding rows, much as the index in a book has a page number associated with each index entry. In general, use B-tree indexes when you know that your typical query refers to the indexed column and retrieves a few rows. In these queries, it is faster to find the rows by looking at the index.

B-tree indexes are most commonly used in a data warehouse to index unique or near-unique keys. In many cases, it may not be necessary to index these columns in a data warehouse, because unique constraints can be maintained without an index, and because typical data warehouse queries may not work better with such indexes. B-tree indexes are more common in environments using third normal form schemas. In general, bitmap indexes should be more common than B-tree indexes in most data warehouse environments.

2- Non-partitioned Indexes

You can create non-partitioned indexes on non-partitioned and partitioned tables. Non-partitioned indexes are primarily used on non-partitioned tables in data warehouse environments and in general to enforce uniqueness if the status of a unique constraint is required to be enforced in a data warehousing environment.

You can use a non-partitioned global index on a partitioned table to enforce a primary or unique key. A non-partitioned (global) index can be useful for queries that commonly retrieve very few rows based on equality predicates or IN-list on a column or set of columns that is not included in the partitioning key. In those cases, it can be faster to scan a single index than to scan many index partitions to find all matching rows.

3- Global Partitioned Indexes

You can create global partitioned indexes on non-partitioned and partitioned tables. In a global partitioned index, the keys in a particular index partition may refer to rows stored in multiple underlying table partitions or sub-partitions. A global index can be range or hash partitioned, though it can be defined on any type of partitioned table. A global index is created by specifying the GLOBAL attribute.

Global index can be useful if there is a class of queries that uses an access path to the table to retrieve a few rows through an index, and by partitioning the index you can eliminate large portions of the index for the majority of its queries.

Partitioning

Partitioning can improve performance in data warehouses. Partitioning can enhance data access and improve overall application performance. This is especially true for applications that access tables and indexes with millions of rows and many gigabytes of data like data warehouse. Partitioned tables and indexes facilitate administrative operations by enabling these operations to work on subsets of data.

Granularity can be easily added or removed to the partitioning scheme by splitting partitions. Thus, if a table's data is skewed to fill some partitions more than others, the ones that contain more data can be split to achieve a more even distribution. Partitioning also allows one to swap partitions with a table. By being able to easily add, remove, or swap a large amount of data quickly, swapping can be used to keep a large amount of data that is being loaded inaccessible until loading is completed, or can be used as a way to stage data between different phases of use.

Partitioning Methods: 1- Range 2- Hash 3- List 4- Composite

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. For example, you might want to partition sales data into monthly partitions. Range partitioning is defined by the partitioning specification for a table or index in PARTITION BY RANGE(column_list) and by the partitioning specifications for each individual partition in VALUES LESS THAN(value_list), where column_list is an ordered list of columns that determines the partition to which a row or an index entry belongs. These columns are called the partitioning columns. The values in the partitioning columns of a particular row constitute that row's partitioning key.

Hash partitioning maps data to partitions based on a hashing algorithm that is applied to a partitioning key that you identify. The hashing algorithm evenly distributes rows among partitions, giving partitions approximately the same size. Hash partitioning is the ideal method for distributing data evenly across devices. Hash partitioning is a good and easy-to-use alternative to range partitioning when data is not historical and there is no obvious column or column list where logical range partition pruning can be advantageous.

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 column in the description for each partition. This is different from range partitioning, where a range of values is associated with a partition and with hash partitioning, where you have no control of the row-to-partition mapping. The advantage of list partitioning is that you can group and organize unordered and unrelated sets of data in a natural way.

Composite partitioning combines range and hash or list partitioning. Database first is distributed data into partitions according to boundaries established by the partition ranges. Then, for range-hash partitioning, u a hashing algorithm is used to further divide the data into sub-partitions within each range partition. For range-list partitioning, the data is divided into sub-partitions within each range partition based on the explicit list you chose.

References:-

1- **Oracle Database Data Warehousing Guide 10*g***