**types of indexes and partitions used in Data warehousing and their usage.**

**1) Indexing usage:**

**-: Indexing the data warehouse can reduce the amount of time it takes to see query results. When indexing dimensions, you want to index on the dimension key. When indexing the fact table, you want to index on the date key or the combined data plus time.**

**-: Indexing a data warehouse is tricky. If you have too few indexes, the data loads quickly but the query response is slow. If you have too many indexes, the data loads slowly but the query response is good. Indexing in any database, transactional or warehouse, most often reduces the length of time it takes to see query results. This is especially true with large tables and complex queries that involve table joins.**

**1:1) Indexes types:**

**1:1:1) B-Tree Index:**

**The B-Tree Index is popular in data warehouse applications for high cardinality columns such as names since the space usage of the index is independent of the column cardinality. However, the B-Tree Index has characteristics that make them a poor choice for DW’s queries. First of all, a B-Tree index is of no value for low cardinality data such as the gender column since it reduces very few numbers of I/Os and may use more space than the raw indexed column. Secondly, each B-Tree Index is independent and thus cannot operate with each other on an index level before going to the primary source.**

**1:1:2) Projection Index:**

**Normally, the queries against a data warehouse retrieve only a few of the table’s columns; so having the Projection Index on these columns reduces tremendously the cost of querying because a single I/O operation may bring more values into memory. Sybase builds a Projection Index under the name of FastProjection Index on every column of a table.**

**1:1:3) Bitmap Index:**

**The bitmap representation is an alternate method of the row ids representation. It is simple to represent, and uses less space- and CPU-efficient than row ids when the number of distinct values of the indexed column is low. The indexes improve complex query performance by applying low-cost Boolean operations such as OR, AND, and NOT in the selection predicate on multiple indexes at one time to reduce search space before going to the primary source data. Many variations of the Bitmap Index (Pure Bitmap Index, Encoded Bitmap, etc.) have been introduced, aiming to reduce space requirement as well as improve query performance.**

**2) partitioning usage:**

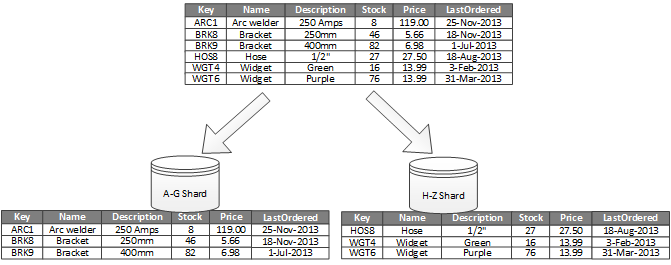
**:- Partitioning offers support for very large tables and indexes by letting you decompose them into smaller and more manageable pieces called partitions. This support is especially important for applications that access tables and indexes with millions of rows and many gigabytes of data. Partitioned tables and indexes facilitate administrative operations by enabling these operations to work on subsets of data.**

**:- Partitioning can help you tune SQL statements to avoid unnecessary index and table scans. It also enables you to improve the performance of massive join operations when large amounts of data are joined together by using partition-wise joins. Finally, partitioning data greatly improves manageability of very large databases and dramatically reduces the time required for administrative tasks such as backup and restore.**

**2:1) partitions types:**

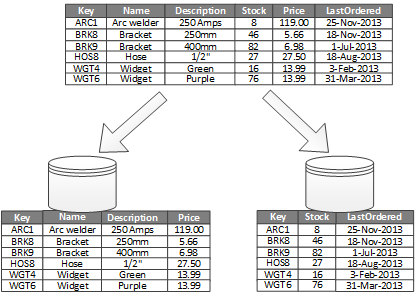
**2:1:1) Horizontal partitioning (sharding):**

**In this strategy, each partition is a separate data store, but all partitions have the same schema. Each partition is known as a shard and holds a specific subset of the data, such as all the orders for a specific set of customers.**

****

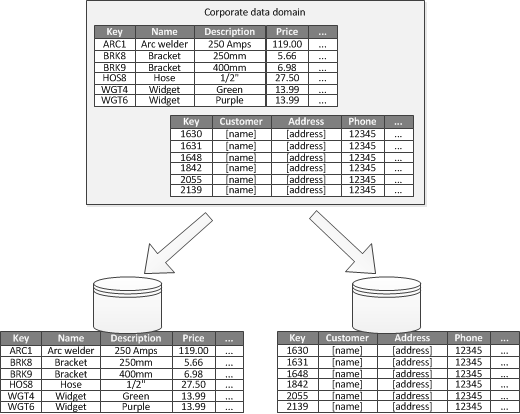
**2:1:2) Vertical partitioning:**

**In this strategy, each partition holds a subset of the fields for items in the data store. The fields are divided according to their pattern of use. For example, frequently accessed fields might be placed in one vertical partition and less frequently accessed fields in another.**

****

**2:1:3) Functional partitioning:**

**In this strategy, data is aggregated according to how it is used by each bounded context in the system. For example, an e-commerce system might store invoice data in one partition and product inventory data in another.**

****