Let's Start with INDEXES.

WHAT is an Index:

In a database, an index is a data structure that provides quick access to rows in a table based on the values in one or more columns.

WHY do we need indexes:

Indexes can greatly improve the performance of database queries, especially for large tables with many rows. They allow the database to quickly narrow down the rows that match a query condition, which can greatly reduce the amount of time and resources required to execute the query.

Let's know a little something about each index and when to use it:

1. **B-tree index:** A B-tree index is a balanced tree structure that allows for quick access to data based on a single column or a combination of columns. It is the most common type of index used in databases and is well-suited for range queries, such as queries that use greater than or less than operators.
2. **Hash index:** A hash index is a data structure that uses a hash function to map keys to values. It is typically used for equality queries, such as queries that use the equals operator. Hash indexes are very fast for lookups, but are not well-suited for range queries or sorting.
3. **Bitmap index:** A bitmap index is a data structure that uses a bitmap to represent the presence or absence of values in a column. Bitmap indexes are well-suited for columns with a low number of distinct values and are particularly useful for data warehousing and decision support systems.
4. **Clustered index:** A clustered index is an index that determines the physical order of the data in a table. This means that the rows in the table are physically sorted based on the indexed columns. Each table can have only one clustered index, and it is typically created on the primary key column(s) of the table. Because the data is physically sorted, clustered indexes are very efficient for range queries and queries that retrieve a large number of rows.
5. **Non-clustered index:** A non-clustered index is an index that does not determine the physical order of the data in a table. Instead, it creates a separate data structure that contains the indexed columns and a pointer to the corresponding rows in the table. Non-clustered indexes are typically created on columns that are frequently searched, but not used for sorting or grouping. Each table can have multiple non-clustered indexes.
6. **Unique index:** A unique index is an index that enforces a constraint that ensures that the values in the indexed column(s) are unique. This means that no two rows in the table can have the same values in the indexed column(s). Unique indexes can be clustered or non-clustered, and they are useful for enforcing data integrity and optimizing queries that search for unique values.

Is it all good?

However, indexes also have some drawbacks. They require additional storage space and can slow down insert, update, and delete operations, as the database must update the index as well as the main table. Additionally, creating too many indexes or using them improperly can actually decrease query performance, as the database must spend more time maintaining and updating the indexes.

As a result, it's important to carefully consider the use of indexes in a database and to create them only for columns that are frequently queried and have a high selectivity (i.e., a large number of unique values). It's also important to regularly monitor and optimize the use of indexes to ensure that they are providing a net benefit to the database's performance.

Finally

In DWH we use a large volume of data, so many rows, so many tables, and so many joins.

In addition to this we do not manipulate our data so often so indexes will be a great help for DWH.

Let's go for PARTITIONING.

Table partitioning is a database design technique that involves dividing a large table into smaller, more manageable pieces called partitions. Each partition is essentially a separate table that contains a subset of the data in the original table, based on a partitioning scheme that is defined by the database administrator.

There are several ways to partition a table, including:

1. **Range partitioning:** This involves dividing the table based on a range of values in a column, such as dates or numeric values.
2. **Hash partitioning:** This involves dividing the table based on a hash function that is applied to a column.
3. **List partitioning:** This involves dividing the table based on a specific list of values in a column.
4. **Composite partitioning:** This involves using a combination of partitioning methods, such as range and hash partitioning.

Table partitioning is commonly used in data warehouses for several reasons:

1. **Improved query performance:** Data warehouses typically contain very large tables, and partitioning these tables can greatly improve query performance by reducing the amount of data that needs to be scanned for a given query. By partitioning the data based on commonly used query criteria, such as time period or geographic region, queries can be executed more quickly and efficiently.
2. **Easier data management:** Data warehouses often require frequent updates and maintenance, and partitioning the data can make it easier to manage and maintain. By dividing the data into smaller, more manageable partitions, it is easier to load and unload data, perform backups and restores, and perform maintenance tasks such as index rebuilding.
3. **Increased availability:** Partitioning the data across multiple disks or servers can increase availability and reduce the risk of data loss. By storing each partition on a separate disk or server, it is possible to isolate failures and ensure that the data remains available even in the event of a hardware failure.
4. **Cost-effective storage:** By partitioning the data and storing it on different types of storage media, it is possible to optimize storage costs. For example, frequently accessed data can be stored on high-performance disks, while less frequently accessed data can be stored on less expensive storage media.

Each partitioning scheme has its own advantages and drawbacks, and the choice of scheme depends on the specific requirements of the database and the queries being executed. It's important to carefully consider the use of table partitioning and to regularly monitor and optimize it for best performance.