**Indexes**

Indexes in SQL are specialized lookup tables that are used by the database search engine to accelerate data retrieval.

In simple terms, an index in SQL is a tool used to quickly identify rows with specific column values. If there were no indexes, the SQL server would have to start with the first row and then go through the entire table until it discovers the relevant rows. This method is known as a full-table scan and can be highly inefficient for large tables

**Why Indexes in SQL are Used?**

**Improved Query Performance**: The primary reason for using indexes is to accelerate query processing. Indexes can drastically reduce the amount of data the server needs to examine.

**Efficient Data Access**: Indexes provide a quick way to access row data for SELECT statements. This is particularly beneficial for tables with a large number of rows.

**Sorting and Grouping Speed**: Indexes improve the speed of data retrieval operations by providing a sorted version of the data, which is faster to process for ORDER BY and GROUP BY operations.

**Unique Constraints**: Indexes can be used to enforce uniqueness for columns to ensure that no two rows of a table have duplicate values in a particular column or a combination of columns.

**Disadvantages:**

**Overuse of Indexes:** While indexes speed up data retrieval, they can slow down data input, through INSERT, UPDATE, and DELETE statements. Each index needs to be updated when data is modified.

**Storage Space:** Indexes consume additional disk space.

**Maintenance Overhead:** Indexes need to be maintained and rebuilt over time, which can add overhead to database maintenance routines.

## ****Clustered Index****

A clustered index sorts and stores the rows of a table based on the values in one or more specified columns. Each table can have only one clustered index, and the choice of the clustering column(s) significantly impacts how data is stored and retrieved.

**Importance of Clustered Index**

**Physical Data Organization:** The primary purpose of a clustered index is to physically order the data rows in the table based on the values in the indexed column(s). This arrangement allows for efficient data retrieval when queries request data in the same order as the clustered index.

**Optimized Data Retrieval:** Clustered indexes are particularly useful for improving query performance when selecting, sorting, or filtering data based on the columns included in the clustered index. They eliminate the need for a separate data lookup process, as the data rows are already stored in the desired order

**Sequential Access:** When queries involve range scans or retrieving a range of data values, a clustered index is highly efficient. It allows for sequential access, reducing disk I/O operations and enhancing query speed.

## ****Non-Clustered Index****

A non-clustered index is a type of index used in relational databases to improve the efficiency of data retrieval operations. Unlike clustered indexes, which affect the physical order of data rows within a table, non-clustered indexes create separate data structures to allow fast access to specific data subsets. This means that non-clustered indexes do not rearrange the physical organization of data, but rather create a separate structure to facilitate quicker access to the data.

**Importance of Non-Clustered Index**

**Faster Data Retrieval**: Non-clustered indexes significantly improve query performance by allowing the database management system (DBMS) to quickly locate and retrieve specific data rows based on the indexed column(s).

**Reduced Disk I/O**: Non-clustered indexes reduce the need for full table scans when querying data. This leads to fewer disk input/output (I/O) operations, resulting in faster query execution.

**Support for Multiple Indexes:** Unlike clustered indexes, which limit a table to one, non-clustered indexes can be created on multiple columns, enabling efficient retrieval for various query patterns.

|  |  |  |
| --- | --- | --- |
| **Index Type** | **Non-Clustered Index** | **Clustered Index** |
| **Physical Order** | Does not determine the physical order of data rows. | Determines the physical order of data rows based on indexed column(s). |
| **Data Structure** | Creates a separate data structure containing index entries with pointers to the corresponding data rows. | Organizes the actual data rows in the table in the specified order. |
| **Multiple Indexes** | Allows for multiple non-clustered indexes on a single table. | Restricts a table to having only one clustered index. |
| **Query Optimization** | Suitable for optimizing query performance when the query does not align with the physical data order. | Ideal for queries that frequently retrieve data in the same order as the clustering column(s). |

**Performance Optimizations in SQL**

[**Avoid SELECT \* and retrieve only necessary columns**](https://www.thoughtspot.com/data-trends/data-modeling/optimizing-sql-queries#sql2)

[**Optimize JOIN operations**](https://www.thoughtspot.com/data-trends/data-modeling/optimizing-sql-queries#sql3)

[**Minimize the use of subqueries**](https://www.thoughtspot.com/data-trends/data-modeling/optimizing-sql-queries#sql4)

Subqueries are very hard for anyone to read and understand. Instead of using subqueries, especially in complex models or reporting, opt for CTEs instead. CTE stands for common table expression and separates your code into a few smaller queries rather than one big query.

Eg: **SELECT** MAX(customer\_signup) **AS** most\_recent\_signup **FROM** (**SELECT** customer\_name, customer\_phone, customer\_signup **FROM** customer\_details **WHERE** **YEAR**(customer\_signup)=2023)

→

**WITH**

2023\_signups **AS** (

**SELECT**

    customer\_name,

customer\_phone,

customer\_signup

**FROM** customer\_details

**WHERE** **YEAR**(customer\_signup)=2023

),

Most\_recent\_signup **AS** (

**SELECT**

MAX(customer\_signup) **AS** most\_recent\_signup

**FROM** 2023\_signups

)

**SELECT** most\_recent\_signup **FROM** Most\_recent\_signup

## Avoid redundant or unnecessary data retrieval

However, it is also important to limit the number of rows you are returning, not just columns.

You can use LIMIT to reduce the number of rows returned

**SELECT** customer\_name **FROM** customer\_details **ORDER** **BY** customer\_signup **DESC** LIMIT 100;

You can also add an OFFSET clause to your LIMIT functions if you don’t want to return the first 100 rows, but want to skip some first. If you wanted to skip the first 20 rows and select the 100 customers after that, you would write:

**SELECT** customer\_name **FROM** customer\_details **ORDER** **BY** customer\_signup **DESC** LIMIT 100 **OFFSET** 20;

**Normalize database tables**

The premise of normalization is to make sure the values in your database tables are easy to locate and query. Normalization at the layer closest to your raw data is important so that you can easily query the values downstream

**Monitor query performance**

One tool to optimize performance is query profiling. This allows you to pinpoint the source of performance issues by looking at statistics such as runtime and rows returned. Query profiling also includes query execution plans which give you insight into what code is running in what order before it runs. To optimize query performance you can also look at database logs, the server itself, and any external applications connected to your cloud database.

**Use UNION ALL instead of UNION**

UNION is an operator used to join the outputs of two SQL queries. It comes in handy when you need to combine two datasets that have the same column names. However, it is important that you understand the difference between the two UNION operators- UNION and UNION ALL.

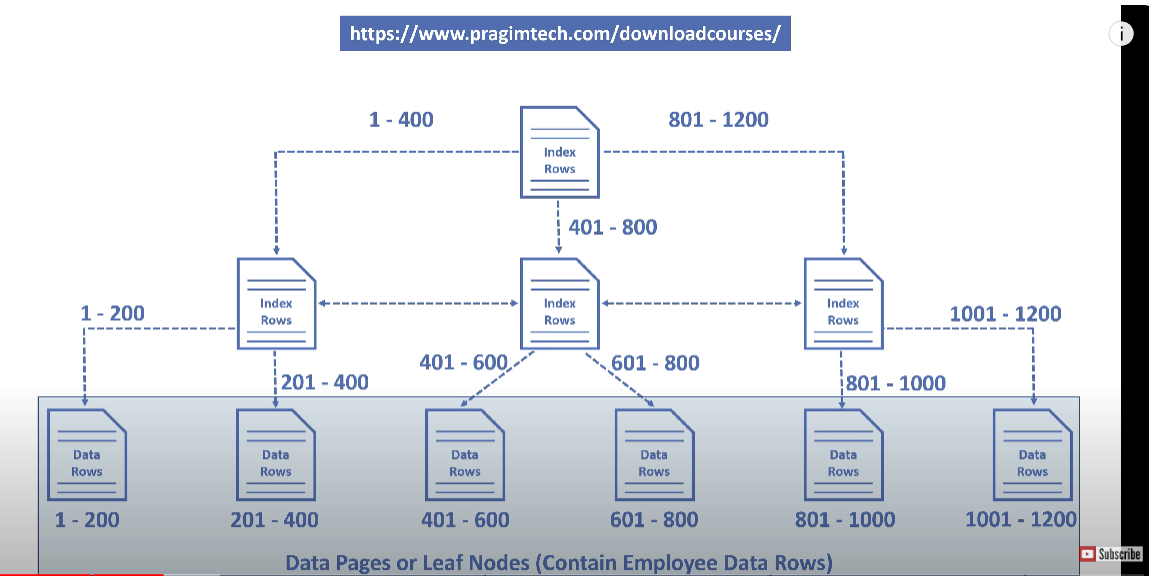
UNION joins all of the rows from Table A with all of the rows from Table B. No deduplication occurs. However, UNION ALL joins all of the rows from Table A with all of the rows from Table B and then deduplicates rows that contain the same values. If you don’t care about duplicates, UNION is going to save you a lot of processing time compared to UNION ALL. I typically always opt for UNION because, even if there are duplicates, I would want to know about them and take the time to understand why that is happening.

### **Use WHERE instead of HAVING to define filters**

A successful query will only retrieve the necessary records from the database. HAVING statements are computed after ***WHERE***statements in accordance with the SQL Order of Operations. A WHERE statement is more effective if the goal is to filter a query based on conditions.

**How indexing works demo :**

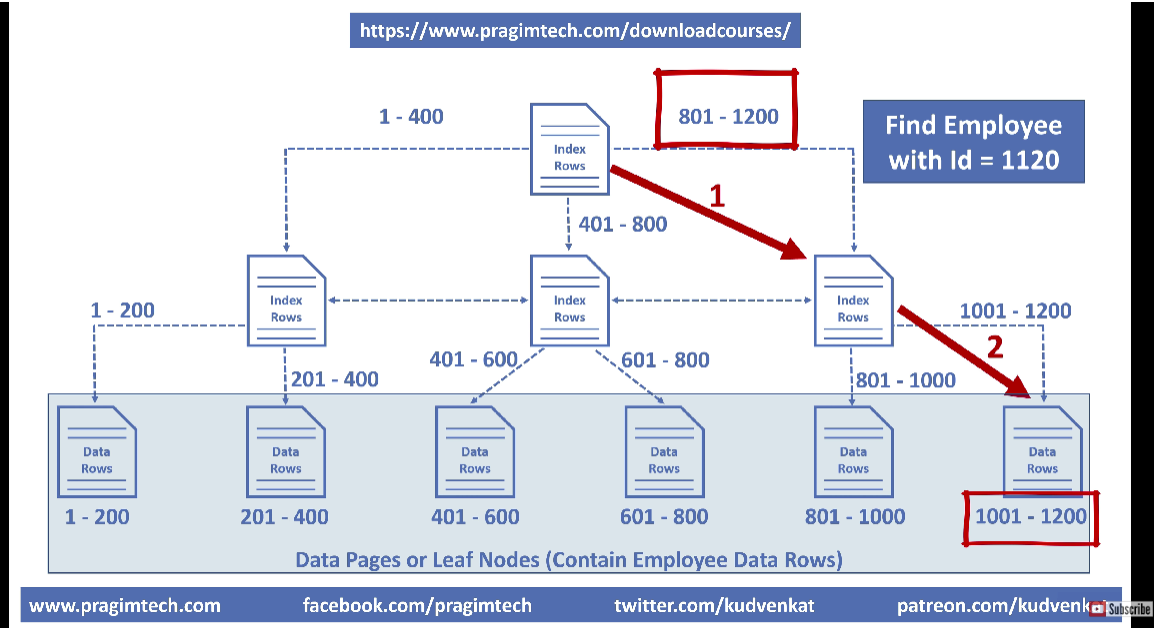
Clustured index:



Leaf nodes contains all data rows which contains ID primary key column.

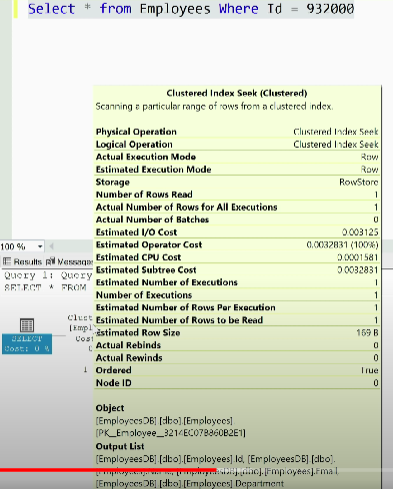
Here ID is a pointer to a intermediate level or leaf nodes. So this tree like structure helps query engine to find data easier.

Eg: select \* form table where id = 1120



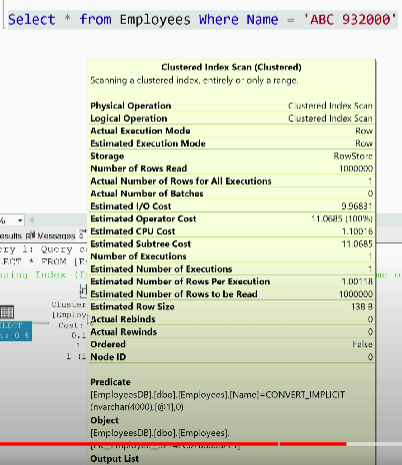
**Execution plan:**

No:of rows read: 1



**Without index:**

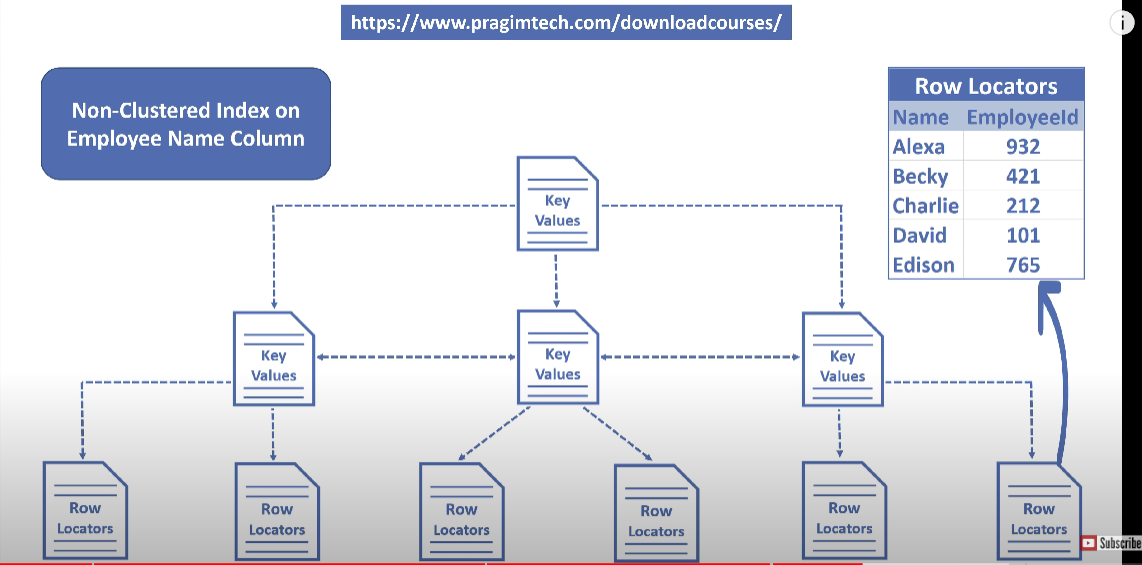
No:of rows read : 100000



**Non clustered index on Name column**

It created row locators with key values in alphabetical orders.

Row locators contains names with cluster key of row (id).



Steps involved in Non-clustured index execution:

