Instructor notes and guidance.

This space has been left deliberately blank

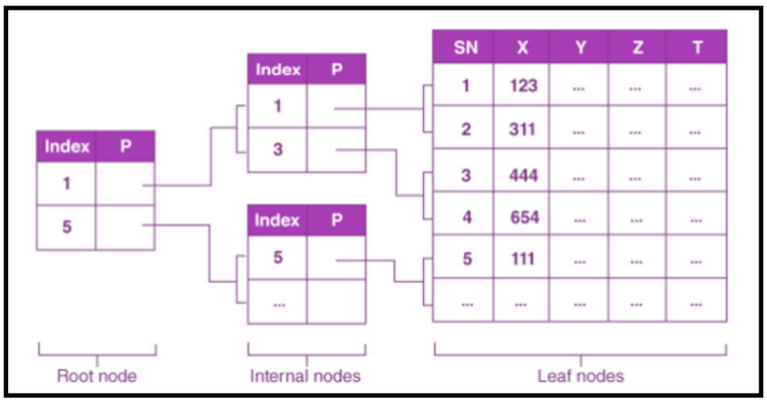
What is an Index?

Think about a regular book:

- At the end of the book, there's an index that helps to quickly locate information within the book.
- The index is a sorted list of **keywords**
- Next to each **keyword** is a set of page numbers pointing to the **pages** where each keyword can be found.

What are SQL Indexes?

- An **index** is a special data structure that improves data retrieval speed.
- Works like a **book index** → find data without scanning the whole table.
- Stored separately from the main table but linked to it.
- Does **not change data**, only speeds up queries.



Without an index, SQL must do a full table scan → checks every row one by one.
With an index, SQL jumps directly to the matching rows.

Index = Shortcut for the database: It helps find data quickly without looking at every row.
Like a book index: Instead of reading the whole book, you check the index to jump straight to the page you need.

Possible structures of indexes:

B-tree / B+ tree: Good when you want things in order or want ranges (like "all students with marks 70–90").
Hash index: Super fast if you know exactly what you're looking for (like "find student with ID 102").

Stored separately: Index doesn't live in the main table but points to it, so it doesn't take over your table data.
Safe to use: It doesn't change the actual data, it just makes queries faster.

Why is indexing important?

- Speeds up data retrieval and query execution.
- Supports **efficient access types**:
  - Find specific values
  - Find values in a range
- Reduces **access time** for searches.
- Makes **inserts and deletes** faster, but requires extra work to update the index after changes
- Uses **extra space** to improve performance - stores additional structures (like pointers or trees) to achieve that speed.

SQL without indexes = linear search → slow for large datasets.
Indexes allow logarithmic or near-constant time access.

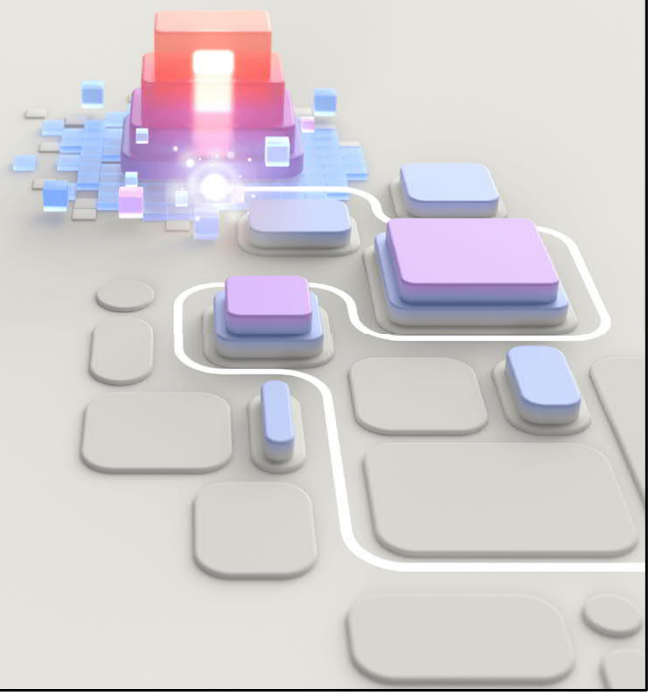
Trade-offs:
Faster reads, but inserts/updates/deletes may slow down since the indexes will have to be updated every time data is deleted or added.
Storage overhead is a small price for big performance gains.

Equality search: Hash indexes are excellent for this because they jump directly to the value.
Range search: B-tree or B+ tree indexes are better here because they maintain a sorted order.
Indexes make both types of queries much faster than scanning every row in the table.
Choice of index type depends on the type of query you run most often.

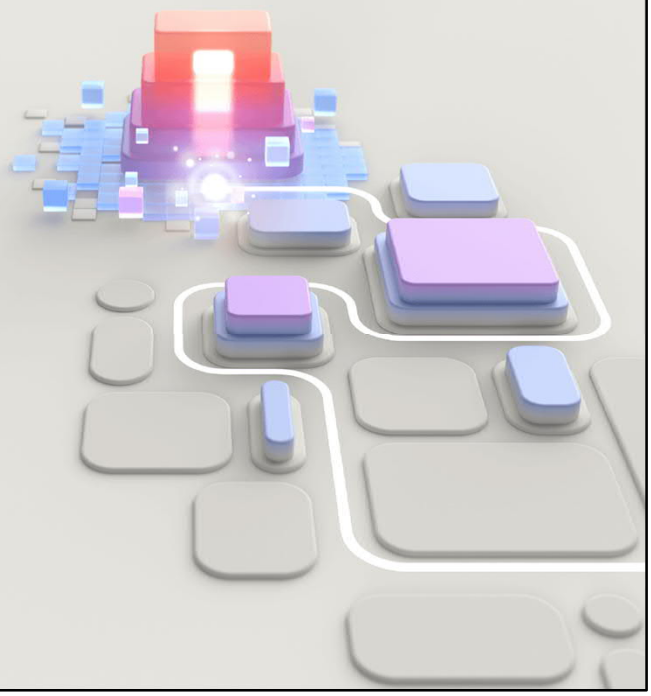
Microsoft

Index Types

Clustered Index
Non-Clustered Index

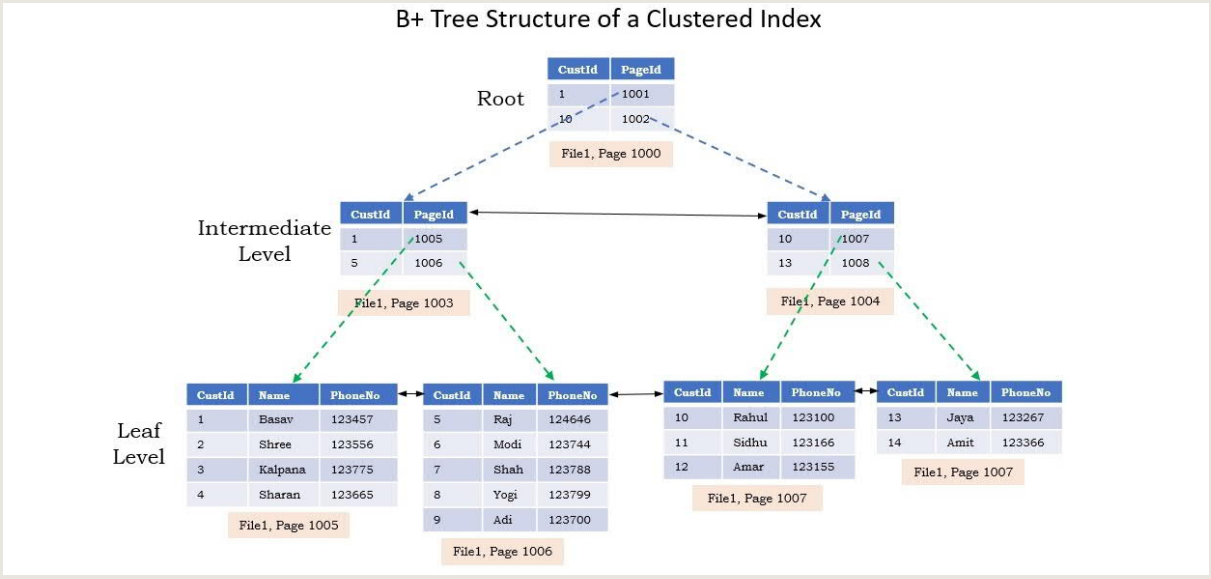


Microsoft



Clustered Index

Clustered Index



Clustered Index (cont...)

- Clustered indexes, implemented as a B+ Trees, sort and store the data rows in the table or view based on their key values.
  - These key values are the columns included in the index definition.
  - There can be only one clustered index per table, because the data rows themselves can be stored in only one order.
- The only time the data rows in a table are stored in sorted order is when the table contains a clustered index.
  - When a table has a clustered index, the table is called a clustered table.
  - If a table has no clustered index, its data rows are stored in an unordered structure called a heap.

https://learn.microsoft.com/en-us/sql/relational-databases/indexes/clustered-and-nonclustered-indexes-described?view=sql-server-ver17

Clustered Index

- Speeds up queries that return **ranges of data**
  - **Example:** `WHERE EmployeeID BETWEEN 100 AND 200;`
- Ideal for columns **frequently sorted or searched**.
  - Example: **EmployeeID**
- Good for **range queries, ORDER BY, and grouping** since rows are stored in index order

Problem Domain:

- Works best for **or mostly unique columns**.
- Not ideal for columns **frequently updated or inserted randomly** (reordering is costly).
  - SQL Server may need to **physically move rows around to maintain the sorted order** of the clustered index.
- Usually applied to **primary key columns**, as each table can have only **one clustered index**.

Clustered:
Clustered indexes sort and store the data rows in the table or view based on their key values. These key values are the columns included in the index definition. There can be only one clustered index per table, because the data rows themselves can be stored in only one order.
The only time the data rows in a table are stored in sorted order is when the table contains a clustered index. When a table has a clustered index, the table is called a clustered table

Clustered indexes are implemented in the following ways:
PRIMARY KEY and UNIQUE constraints

When you create a PRIMARY KEY constraint, a unique clustered index on the column or columns is automatically created if a clustered index on the table doesn't already exist and you don't specify a unique nonclustered index. The primary key column can't allow NULL values.

When you create a UNIQUE constraint, a unique nonclustered index is created to enforce a UNIQUE constraint by default. You can specify a unique clustered index if a clustered index on the table doesn't already exist.

When you create a clustered index, SQL Server has to reorganize the table data on disk according to the index order.
The old table (source) still exists while SQL Server builds the new ordered structure (target).
That means SQL needs extra disk space to hold both versions temporarily (the old data and the new indexed copy) until the process finishes.
Once the clustered index is complete, the old structure is dropped and only the new one remains.

In short: SQL Server needs double storage during creation because it must keep the original table and the new clustered index at the same time.

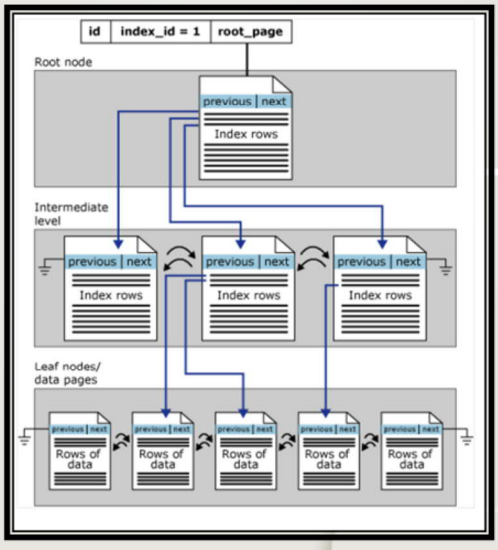
Indexes and constraints

- SQL Server automatically creates indexes when PRIMARY KEY and UNIQUE constraints are defined on table columns.
  - For example, when you create a table with a UNIQUE constraint, Database Engine automatically creates a nonclustered index.
  - If you configure a PRIMARY KEY, Database Engine automatically creates a clustered index, unless a clustered index already exists.
  - When you try to enforce a PRIMARY KEY constraint on an existing table and a clustered index already exists on that table, SQL Server enforces the primary key using a nonclustered index.

https://learn.microsoft.com/en-us/sql/relational-databases/indexes/clustered-and-nonclustered-indexes-described?view=sql-server-ver17

Architecture:

- Organized as a **B+ tree**
- **Root node**: top level for searching
- **Intermediate nodes**: guide the search
- **Leaf nodes**: contain actual table rows (physically sorted)
- **Inserts** follow key order: New rows are placed where they fit to keep the table sorted.
- **Partitions**: Large tables can be split into sections, each with its own B+ tree for faster queries.
- Supports **variable-length columns**: Large or flexible-size data is managed without breaking the index structure.



1. Organized as a B+ tree
Clustered indexes in SQL Server use a B+ tree structure, which is a type of balanced tree.
Balanced tree means all paths from root to leaf have the same length, so searches are predictable and fast.
The tree ensures that data can be quickly searched, inserted, or deleted while keeping rows sorted.

2. Root node: top level for searching
The root node is the entry point of the B+ tree.
When a query is executed, the database starts at the root to find the path to the leaf node containing the data.
The root contains key values and pointers to intermediate nodes.
There is only one root per B+ tree.

3. Intermediate nodes: guide the search
Intermediate nodes sit between the root and leaf nodes.
They contain key values and pointers to other nodes below them.
Their job is to guide the search efficiently down the tree, reducing the number of pages that must be read.

4. Leaf nodes: contain actual table rows (physically sorted)
Leaf nodes are at the bottom of the tree.
In a clustered index, leaf nodes contain the actual table data, not just pointers.
Rows in the leaf nodes are physically stored in order of the clustered key.
This ordering allows range queries and sequential reads to be extremely fast.

5. Inserts follow key order
When a new row is added, SQL Server finds the correct leaf node based on the key.
The row is inserted in the right position to maintain the sort order.
If the leaf node is full, SQL Server splits the node and updates parent/intermediate nodes to keep the tree balanced.

6. Partitions
Very large tables can be divided into partitions (sections of the table).
Each partition has its own B+ tree, making queries faster because the engine only searches the relevant partition.
Partitioning also helps manage storage and maintenance efficiently.

7. Supports LOBs & variable-length columns
LOBs (Large Objects): e.g., images, documents, large text fields.

Variable-length columns: e.g., VARCHAR, NVARCHAR, which can change size per row.
SQL Server manages these in special allocation units separate from the normal B+ tree pages.
This ensures large or flexible data doesn't break the tree structure or slow down searches.

Image credits: SQL Server and Azure SQL Index Architecture and Design Guide - SQL Server | Microsoft Learn

Usage

In this table, the rows are physically stored on disk in the order of **EmployeeID**.

EmployeeID	Name	Dept	Salary
101	Ali	IT	5000
102	Sarah	CS	3400
103	Omer	Physics	4500
104	Ben	Math	6000

- Table rows are **physically sorted by EmployeeID**.
- **Leaf nodes** of the index contain the actual table data.
- Only **one clustered index per table**; usually on the **primary key**.

Query:

```
SELECT *
FROM Employees
WHERE EmployeeID BETWEEN
101 AND 103;
```

- Database uses the **clustered index B+ tree** to find **EmployeeID 101** quickly.
- Reads **consecutive rows** in order until EmployeeID 103.
- **No need to scan the entire table**, so the query is faster.

Clustered Index is like a Dictionary

- The words (data rows) are stored in alphabetical order (index key order).
- You can quickly find a word because you know where it should be.
- The dictionary itself is the data—no need to look elsewhere.