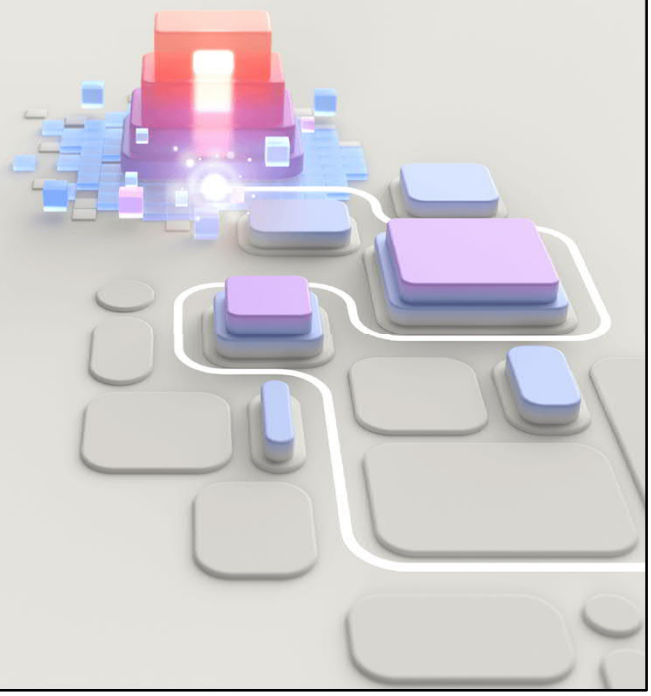
Microsoft



Non-Clustered Indexes

Recap - Clustered & Non-Clustered Index

- With a clustered index the rows are stored physically on the disk in the same order as the index. Therefore, **there can be only one clustered index.**
- With a non-clustered index there is a **second list** that has pointers to the physical rows.
  - You can have many non clustered indices, although each new index will increase the time it takes to write new records.
- It is generally faster to read from a clustered index if you want to get back all the columns. You do not have to go first to the index and then to the table.
- Writing to a table with a clustered index can be slower, if there is a need to rearrange the data.

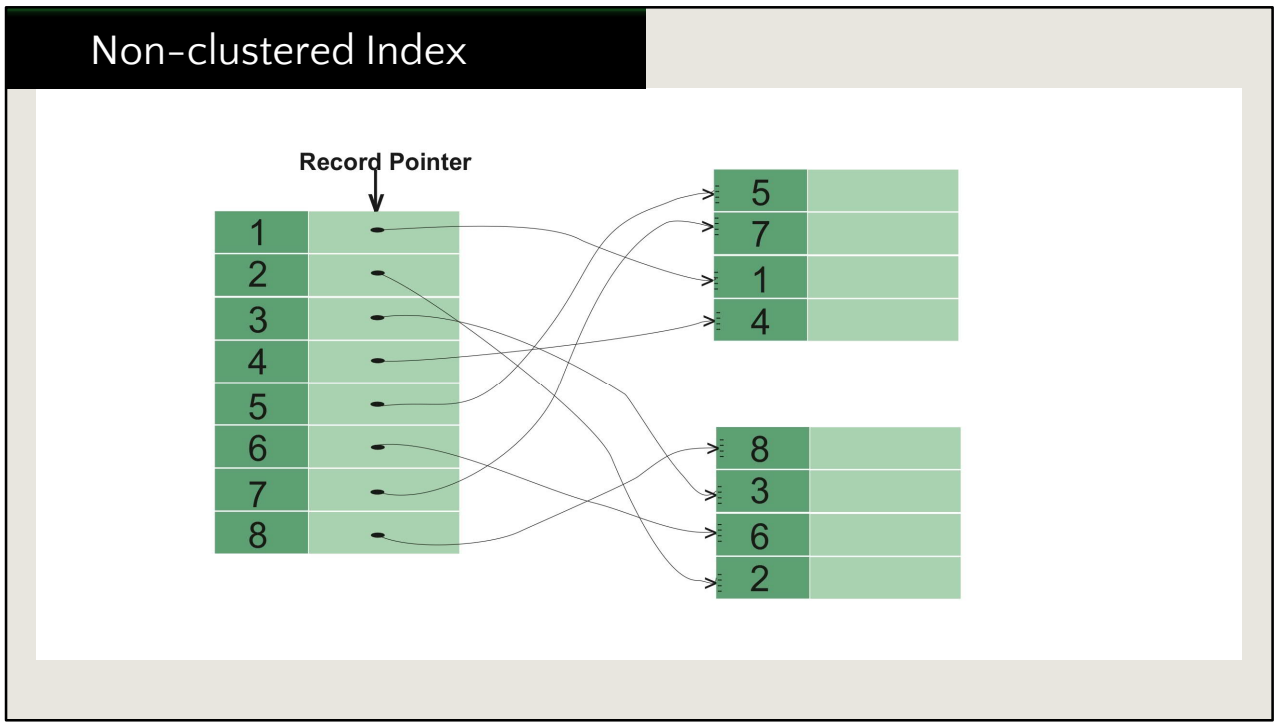
https://stackoverflow.com/questions/91688/what-are-the-differences-between-a-clustered-and-a-non-clustered-index
https://stackoverflow.com/questions/1251636/what-do-clustered-and-non-clustered-index-actually-mean

Non-Clustered Index (Heap Tables)

A **table** without a clustered index is called a heap.

- Data is stored without any defined order, and rows are located using a **Row Identifier (RID)**, which includes the file number, data page number, and slot on the page.
- To guarantee the order of rows returned from a heap in response to a SELECT query, use the ORDER BY clause.
  - To specify a permanent logical order for storing the rows, create a clustered index on the table, so that the table is not a heap.

https://learn.microsoft.com/en-us/sql/relational-databases/indexes/heaps-tables-without-clustered-indexes

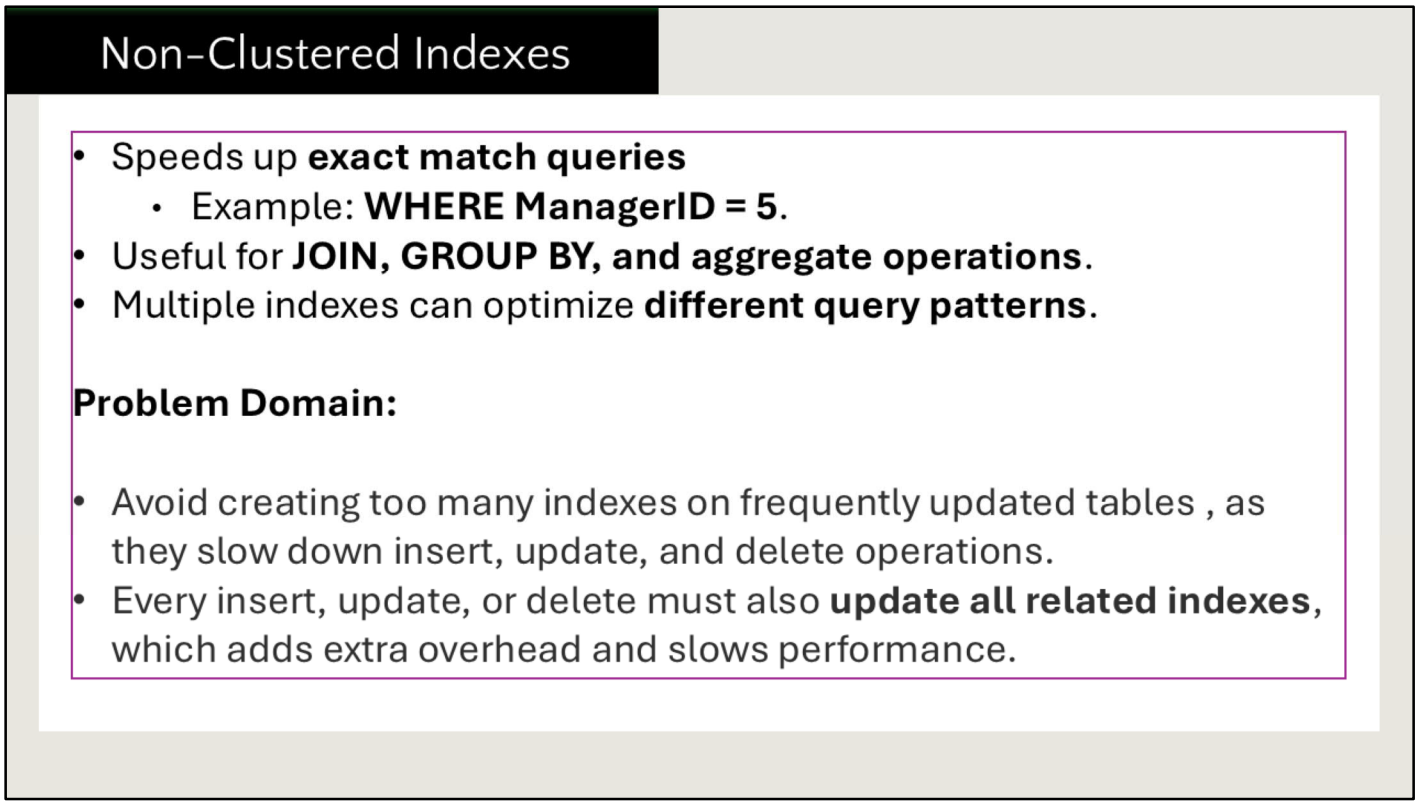


Nonclustered indexes have a structure separate from the data rows. A nonclustered index contains the nonclustered index key values and each key value entry has a pointer to the data row that contains the key value.

The pointer from an index row in a nonclustered index to a data row is called a row locator. The structure of the row locator depends on whether the data pages are stored in a heap or a clustered table. For a heap, a row locator is a pointer to the row. For a clustered table, the row locator is the clustered index key.

Both clustered and nonclustered indexes can be unique. With a unique index, no two rows can have the same value for the index key. Otherwise, the index isn't unique and multiple rows can share the same key value.

Indexes are automatically maintained for a table or view whenever the table data is modified.



Clustered and nonclustered indexes - SQL Server | Microsoft Learn

Non-clustered Indexes in SQL Server

A nonclustered index is a separate data structure that contains index key values and row locators pointing to the actual table data.
Unlike clustered indexes, the table rows are not physically sorted by the nonclustered key.
A table can have multiple nonclustered indexes to optimize different query patterns.

Supports exact match queries very efficiently (e.g., WHERE ManagerID = 5).
Leaf nodes of the B+ tree contain index pages (not actual data rows) along with key and included columns.

Non-Clustered Indexes – Implementation

The implementation of the nonclustered index depends on whether the data pages of a table are managed as a Heap or as a clustered index.

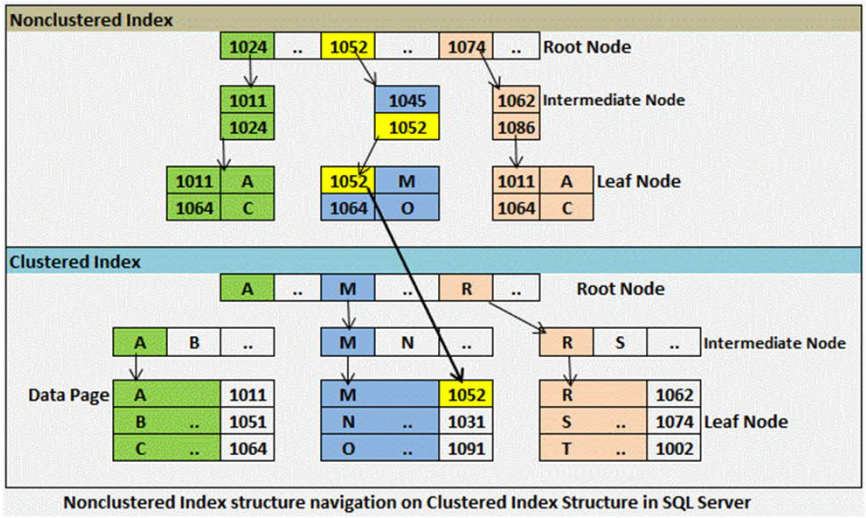
Heap based Non-clustered Index structure in SQL Server:

- If non-clustered index is built in a heap then SQL Server uses pointers in the leaf node index pages that point to a row in the data pages.

Non-clustered Index based on Clustered Index Structure:

- If we have a table with clustered index, SQL Server builds a non-clustered index on the top of the clustered index structure and SQL Server uses clustering keys in the leaf node index page of the non-clustered index to point the cluster index.

Non-Clustered Indexes - Implementation

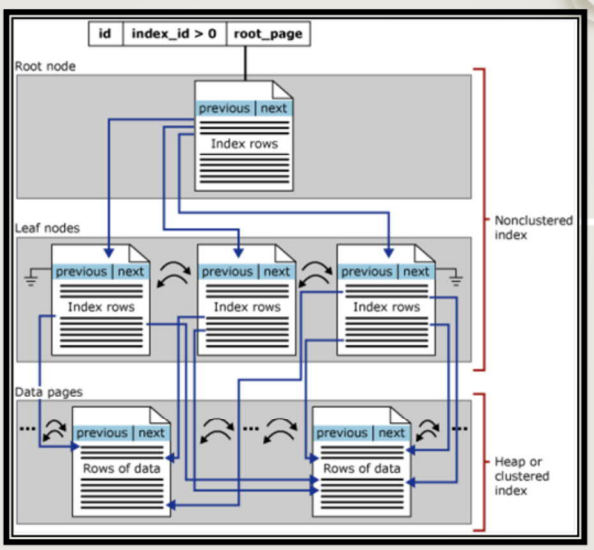


Nonclustered Index structure navigation on Clustered Index Structure in SQL Server

https://www.sql-datatools.com/2017/02/nonclustered-index-on-clustered-index-sql-server-2016.html

Architecture

- Organized as a **B+ tree** for fast searching.
- **Root node:** entry point for searches.
- **Intermediate nodes:** guide the search down the tree.
- **Leaf nodes:** contain **key values, included columns, and row locators** (pointing to actual table rows).
- **Row locator:**
  - Heap → Row ID (RID) - Direct physical location of the row
  - Clustered table → Clustered index key - Use key to locate row in the clustered index
- **Inserts:** follow key order; leaf splits maintain balance.



- Leaf nodes of the non-clustered index do not store actual table rows.
- Each leaf entry contains:
  - Key column value(s) (the indexed column)
  - Optional included columns
  - A row locator (Row locator points to actual data)
- Heap table: points to the row in the table using a Row ID (RID).
- Clustered table: points to the row via the clustered index key.

- So when you query using a non-clustered index:

1. SQL Server traverses the B+ tree to find the key in the leaf node.
2. Uses the row locator to fetch the actual data from the table (or clustered index).

1. Heap Table (No Clustered Index)

We don't use a stack or queue structure because tables need random access to any row, not just the "top" or "last" row.
A heap is better since it allows SQL Server to place rows anywhere with free space and access them directly via a Row ID (RID).

A heap is a table with no clustered index.
Rows are stored in no particular order.
Row Locator: Row ID (RID)

The RID contains:

- File ID → which file the row is in.
- Page number → which page in that file.
- Row number → which row on that page.

SQL Server uses the RID to go directly to the row.
Analogy: Like a library shelf number + row number to find a book in a disorganized library.

2. Table with Clustered Index

The table's rows are physically sorted by the clustered index key.
Row Locator: the clustered index key itself

Instead of storing the physical address, SQL Server uses the clustered key to find the row in the clustered index.

This ensures non-clustered index entries remain valid even if rows move within the clustered index due to inserts/updates.
Analogy: Like a library catalog using ISBN numbers to find the book on a shelf that's sorted by ISBN.

Example of use:

EmployeeID	Name	Dept	ManagerID
101	Ali	IT	2
102	Sarah	CS	7
103	Omer	Physics	2
104	Ben	Math	5

- **Index on ManagerID** column lets the database quickly find employees reporting to a manager.
- **Leaf nodes** store **ManagerID + row locator**, not full data.
- **Row locator** points to the actual row in database

ManagerID	Row Locator
7	RID -> points to row 102
5	RID -> points to row 104

Query:

```
SELECT Name, Department
FROM Employees
WHERE ManagerID = 5;
```

- Query WHERE ManagerID = 5:
  1. Searches index leaf nodes.
  2. Finds all matching rows.
  3. Follows row locators to fetch Name and Department.
- Multiple rows with the same ManagerID are **sorted in leaf nodes by key**.

If you create an index on ManagerID, the database builds a separate structure (like a quick lookup list) that makes it easy to find all employees under a certain manager. Instead of checking every row in the table, it can just search this index.

The leaf nodes of this index don't hold the entire employee record. They only keep the ManagerID value and a small pointer called a row locator.

The row locator works differently depending on how the table is stored:

In a heap (no clustered index), it points directly to the row's physical location on disk (called a Row ID or RID).

In a clustered table, it points to the row's clustered index key (e.g., EmployeeID), and SQL Server uses that key to find the row.

In short: the index helps you get to the right rows quickly, and the row locator is the "map" that tells SQL where to fetch the full data.

Example of how Index is formed in Microsoft SQL

```sql
CREATE TABLE Products_table (
    ProductID INT CONSTRAINT PK_Products PRIMARY KEY NONCLUSTERED,
    -- PK but nonclustered
    ProductName NVARCHAR(50),
    Price DECIMAL(10,2)
);

-- Now add clustered index manually
CREATE CLUSTERED INDEX IX_Prod_ID
ON Products_table(ProductID);

-- Add nonclustered index
CREATE NONCLUSTERED INDEX IX_Prod_Price
ON Products_table(Price);
```

This block **creates a table with a nonclustered primary key, then adds a clustered index on ProductID and a nonclustered index on Price.**


```sql
SELECT name, type_desc
FROM sys.indexes
WHERE object_id = OBJECT_ID('Products_table');
```

This query **retrieves the names and types (clustered/nonclustered) of all indexes that exist on the table Products_table.**

Results | Messages

	name	type_desc
1	IX_Prod_ID	CLUSTERED
2	PK_Products	NONCLUSTERED
3	IX_Prod_Price	NONCLUSTERED

Without records inserted

Results | Messages

	IndexName	IndexType	index_id	TotalPages	TotalRecords	SizeKB	SizeMB
1	IX_Prod_ID	CLUSTERED	1	0	0	0	0.000000
2	PK_Products	NONCLUSTERED	2	0	0	0	0.000000
3	IX_Prod_Price	NONCLUSTERED	3	0	0	0	0.000000

- Index structures are created when you define **PRIMARY KEY / CLUSTERED / NONCLUSTERED**.
- But until rows exist in the table → **no records are stored in the indexes**.
- A record = one **row** of your table

In SQL Server B+ Tree Index
- **Root Page** → the topmost page (only one).
- **Intermediate Pages** → the middle levels (if index is large enough).
- **Leaf Pages** → the bottom level:
  - For a **clustered index** → leaf pages = **actual data rows**.
  - For a **nonclustered index** → leaf pages = **index key + pointer to data**.

What a Page Stores
A page is 8 KB block of storage inside the database file.
It stores multiple rows (records) from a table or index.
Types of pages:
Data page → actual table rows.
Index page → index key values + pointers (like a directory).

What a Record Stores
A record = one row of your table (e.g., ProductID = 1, ProductName = 'Mouse', Price = 25.00).
Stored inside a page along with other rows.
SQL Server packs as many records into an 8 KB page as possible.

Putting it Together
Pages = “boxes” of 8 KB each.
Records = “rows” stored inside those boxes.
Size = number of boxes × 8 KB.
Example: if 500 rows fit into 100 pages → size = 100 × 8 KB = 800 KB.

With records inserted

```
INSERT INTO Products_table (ProductID, ProductName, Price)
VALUES (1, 'Mouse', 25.00),
       (2, 'Keyboard', 45.00),
       (3, 'Monitor', 250.00),
       (4, 'Laptop', 1200.00);
```

Results | Messages

	IndexName	IndexType	index_id	TotalPages	TotalRecords	SizeKB	SizeMB
1	IX_Prod_ID	CLUSTERED	1	1	4	8	0.007812
2	PK_Products	NONCLUSTERED	2	1	4	8	0.007812
3	IX_Prod_Price	NONCLUSTERED	3	1	4	8	0.007812

IX_Prod_ID (Clustered Index)
- Stores the **actual table rows**.
- 4 rows fit into **1 page (8 KB)**.

PK_Products (Nonclustered on ProductID)
- Stores **ProductID + pointer** to clustered index row.
- 4 entries, fits in **1 page (8 KB)**.

IX_Prod_Price (Nonclustered on Price)
- Stores **Price + pointer** to clustered index row.
- 4 entries, also **1 page (8 KB)**.

IX_Prod_ID (Clustered Index)
The clustered index leaf pages hold the actual table rows.
Since we inserted 4 rows, all of them fit inside a single 8 KB page.
That's why TotalPages = 1, TotalRecords = 4, Size = 8 KB.

PK_Products (Nonclustered on ProductID)
A nonclustered index doesn't store the full row, only the key column (ProductID) plus a pointer to where the full row exists in the clustered index.
One entry is created for each row → 4 entries total.
They all fit in a single 8 KB page as well.

IX_Prod_Price (Nonclustered on Price)
Similar to PK_Products, but the key column is Price instead of ProductID.
It also stores a pointer to the clustered index to retrieve the full row.
With only 4 rows, all entries fit into a single page (8 KB).

Non-Clustered Index is like a index in the back of a book …

- It lists topics and page numbers (pointers).
- You still need to flip to the page to read the actual content.