**Database Indexes: A detailed guide**

Consider a large **Book** of 1000 pages.

Suppose you’re trying to find the page which contains information related to a certain **word**.

Without an index page, you would have to go through every page, which could take hours or even days.

But with an index page, you know where to look!

One you find the right index, you can efficiently jump to that page.

The index, since it's **sorted** alphabetically and gives page numbers for specific information, saves us from spending too much time flipping through every page.

**Database indexes** work in a similar manner. They guide the database to the exact location of the data, enabling faster and more efficient data retrieval.

In this article, we'll explore:

* What are database indexes?
* How do they work?
* Benefits of using them.
* Different types of indexes.
* Which data structure they use?
* How to use them smartly?

# **1. What are Database Indexes?**

[[A diagram of a table and a pointer

AI-generated content may be incorrect.](https://substackcdn.com/image/fetch/$s_!yl9Q!,f_auto,q_auto:good,fl_progressive:steep/https%3A%2F%2Fsubstack-post-media.s3.amazonaws.com%2Fpublic%2Fimages%2F4b31f2da-a2be-4ceb-aff8-7ce9d0477a2d_1120x738.png)](https://substackcdn.com/image/fetch/$s_!yl9Q!,f_auto,q_auto:good,fl_progressive:steep/https%3A%2F%2Fsubstack-post-media.s3.amazonaws.com%2Fpublic%2Fimages%2F4b31f2da-a2be-4ceb-aff8-7ce9d0477a2d_1120x738.png" \t "_blank)

A database index is a super-efficient lookup table that allows a database to find data much faster.

It holds the indexed column values along with pointers to the corresponding rows in the table.

Without an index, the database might have to scan every single row in a massive table to find what you want – a painfully slow process.

But, with an index, the database can zero in on the exact location of the desired data using the index’s pointers.

### How to create Indexes?

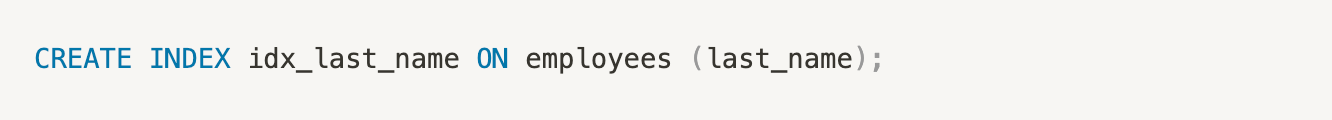
Here's an example of creating an index in a MySQL database.

Let's say we have a table named employees with the following structure:

[[A white background with red numbers

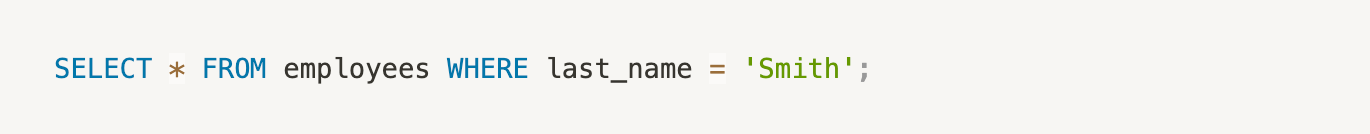
AI-generated content may be incorrect.](https://substackcdn.com/image/fetch/$s_!KdCH!,f_auto,q_auto:good,fl_progressive:steep/https%3A%2F%2Fsubstack-post-media.s3.amazonaws.com%2Fpublic%2Fimages%2Fdd1f2fcd-22d6-4b99-abd6-c56bbf56585f_1216x396.png)](https://substackcdn.com/image/fetch/$s_!KdCH!,f_auto,q_auto:good,fl_progressive:steep/https%3A%2F%2Fsubstack-post-media.s3.amazonaws.com%2Fpublic%2Fimages%2Fdd1f2fcd-22d6-4b99-abd6-c56bbf56585f_1216x396.png" \t "_blank)

Now, let's create an index on the last\_name column to improve the performance of queries that frequently search or sort based on the last name.

[[](https://substackcdn.com/image/fetch/$s_!ygnK!,f_auto,q_auto:good,fl_progressive:steep/https%3A%2F%2Fsubstack-post-media.s3.amazonaws.com%2Fpublic%2Fimages%2Fe6e9c448-4de4-4bbe-9304-e1b8d9a84bfd_1332x120.png)](https://substackcdn.com/image/fetch/$s_!ygnK!,f_auto,q_auto:good,fl_progressive:steep/https%3A%2F%2Fsubstack-post-media.s3.amazonaws.com%2Fpublic%2Fimages%2Fe6e9c448-4de4-4bbe-9304-e1b8d9a84bfd_1332x120.png" \t "_blank)

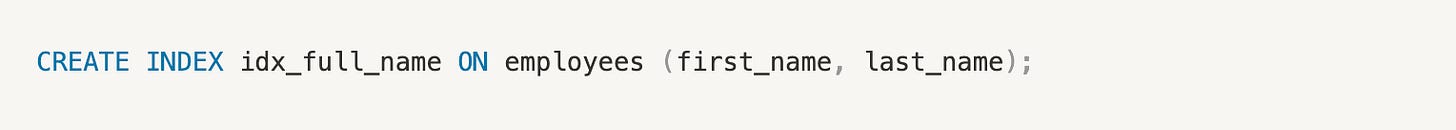
In this example, we use the CREATE INDEX statement to create an index named idx\_last\_name on the employees table. The index is created on the last\_name column.

After creating the index, queries that involve conditions or sorting on the last\_name column will be optimized. For example:

[[](https://substackcdn.com/image/fetch/$s_!afja!,f_auto,q_auto:good,fl_progressive:steep/https%3A%2F%2Fsubstack-post-media.s3.amazonaws.com%2Fpublic%2Fimages%2Fbeb07c6f-61c9-4e5d-9555-aa5451ac56a8_1370x134.png)](https://substackcdn.com/image/fetch/$s_!afja!,f_auto,q_auto:good,fl_progressive:steep/https%3A%2F%2Fsubstack-post-media.s3.amazonaws.com%2Fpublic%2Fimages%2Fbeb07c6f-61c9-4e5d-9555-aa5451ac56a8_1370x134.png" \t "_blank)

This query will use the idx\_last\_name index to quickly locate the rows where the last\_name is 'Smith', avoiding a full table scan.

You can also create indexes on multiple columns (composite indexes) if your queries frequently involve conditions on multiple columns together. For example:

[[](https://substackcdn.com/image/fetch/$s_!lmOl!,f_auto,q_auto:good,fl_progressive:steep/https%3A%2F%2Fsubstack-post-media.s3.amazonaws.com%2Fpublic%2Fimages%2Fd042d7a7-e77c-4d48-b91c-b396474c8f6d_1528x136.png)](https://substackcdn.com/image/fetch/$s_!lmOl!,f_auto,q_auto:good,fl_progressive:steep/https%3A%2F%2Fsubstack-post-media.s3.amazonaws.com%2Fpublic%2Fimages%2Fd042d7a7-e77c-4d48-b91c-b396474c8f6d_1528x136.png" \t "_blank)

This creates a composite index on the first\_name and last\_name columns, which can be useful for queries that search or sort based on both columns.

**2. How do Database Indexes Work?**

Here's a step-by-step explanation of how database indexes work:

1. **Index Creation**: The database administrator creates an index on a specific column or set of columns.
2. **Index Building**: The database management system builds the index by scanning the table and storing the values of the indexed column(s) along with a pointer to the corresponding data.
3. **Query Execution**: When a query is executed, the database engine checks if an index exists for the requested column(s).
4. **Index Search**: If an index exists, the database searches the index for the requested data, using the pointers to quickly locate the data.
5. **Data Retrieval**: The database retrieves the requested data, using the pointers from the index.

**3. Benefits of Database Indexes**

Database indexes offer several benefits, including:

* **Faster Query Performance**: Indexes can significantly improve query performance especially for large datasets by reducing the amount of data that needs to be scanned.
* **Reduced CPU Usage**: By reducing the number of rows that need to be scanned, indexes can decrease CPU usage and optimize resource utilization.
* **Rapid Data Retrieval**: Indexes enable quick data retrieval for queries that involve equality or range conditions on the indexed columns.
* **Efficient Sorting**: Indexes can also be used to efficiently sort data based on the indexed columns, eliminating the need for expensive sorting operations.
* **Better Data Organization**: Indexes can help maintain data organization and structure, making it easier to manage and maintain the database.

**4. Types of Database Indexes**

**Indexes based on Structure and Key Attributes:**

* **Primary Index:** Automatically created when a primary key constraint is defined on a table. Ensures uniqueness and helps with super-fast lookups using the primary key.
* **Clustered Index:** Determines the order in which data is physically stored in the table. A clustered index is most useful when we’re searching in a range. Only one clustered index can exist per table.
* **Non-clustered or Secondary Index:** This index does not store data in the order of the index. Instead, it provides a list of virtual pointers or references to the location where the data is actually stored.

**Indexes based on Data Coverage:**

* **Dense index:** Has an entry for every search key value in the table. Suitable for situations where the data has a small number of distinct search key values or when fast access to individual records is required.
* **Sparse index:** Has entries only for some of the search key values. Suitable for situations where the data has a large number of distinct search key values.

**Specialized Index Types:**

* **Bitmap Index:** Excellent for columns with low cardinality (few distinct values). Common in data warehousing.
* **Hash Index:** A index that uses a hash function to map values to specific locations. Great for exact match queries.
* **Filtered Index:** Indexes a subset of rows based on a specific filter condition. Useful to improve query speed on commonly filtered columns.
* **Covering Index:** Includes all the columns required by a query in the index itself, eliminating the need to access the underlying table data.
* **Function-based index:** Indexes that are created based on the result of a function or expression applied to one or more columns of a table.
* **Full-Text Index**: A index designed for full-text search, allowing for efficient searching of text data.
* **Spatial Index:** Used for indexing geographical data types.

**5. What Data Structure do Indexes use?**

Most commonly used data structures that power indexes are B-Trees, Hash Tables and Bitmaps.

**B-Tree (Balanced Tree)**

[[A diagram of numbers and squares

AI-generated content may be incorrect.](https://substackcdn.com/image/fetch/$s_!JuEy!,f_auto,q_auto:good,fl_progressive:steep/https%3A%2F%2Fsubstack-post-media.s3.amazonaws.com%2Fpublic%2Fimages%2Fbd557e1a-d752-4653-9ae7-1ae2483fa252_2284x632.png)](https://substackcdn.com/image/fetch/$s_!JuEy!,f_auto,q_auto:good,fl_progressive:steep/https%3A%2F%2Fsubstack-post-media.s3.amazonaws.com%2Fpublic%2Fimages%2Fbd557e1a-d752-4653-9ae7-1ae2483fa252_2284x632.png" \t "_blank)

Most database engines use either a B-Tree or a variation of B-Trees like B+ Trees.

B-Trees have a hierarchical structure with a root node, internal nodes (index nodes), and leaf nodes.

Each node in a B-Tree contains a sorted array of keys and pointers to child nodes.

Here's why they are so well-suited:

* **Self-Balancing:** B-trees ensure that the 'height' of the tree stays balanced even when inserting or deleting data. This ensures logarithmic time complexity for insertion, deletion, and searching.
* **Ordered:** B-trees keep the data sorted, making range queries ("find all orders between date X and Y") and inequality comparisons very fast.
* **Disk-Friendly:** B-trees are designed to work well with disk-based storage. A single node of a B-tree often corresponds to a disk block, minimizing disk access operations.

Many databases use a slightly modified B-tree variant called the B+ tree.

In a B+ tree, all data values are stored only in the leaf nodes, which can further improve performance for certain use cases like range queries.

**Hash Tables**

[[A diagram of a computer program

AI-generated content may be incorrect.](https://substackcdn.com/image/fetch/$s_!kh9E!,f_auto,q_auto:good,fl_progressive:steep/https%3A%2F%2Fsubstack-post-media.s3.amazonaws.com%2Fpublic%2Fimages%2F0e98b8f5-8d7b-4c4a-b692-c4d547260ee8_2120x760.png)](https://substackcdn.com/image/fetch/$s_!kh9E!,f_auto,q_auto:good,fl_progressive:steep/https%3A%2F%2Fsubstack-post-media.s3.amazonaws.com%2Fpublic%2Fimages%2F0e98b8f5-8d7b-4c4a-b692-c4d547260ee8_2120x760.png" \t "_blank)

Hash tables are used for hash indexes, which are based on a hash function.

A hash table consists of an array of buckets, with each bucket containing the addresses for rows in the data.

Hash indexes employ a hash function to map keys to their corresponding bucket in the hash table, enabling constant-time lookup operations.

Hash indexes provide fast equality lookups, as the hash function determines the exact location of the data based on the key.

However, hash indexes do not support range queries or sorting efficiently.

**Bitmaps**

[[A blue and yellow squares with black letters

AI-generated content may be incorrect.](https://substackcdn.com/image/fetch/$s_!-Yoj!,f_auto,q_auto:good,fl_progressive:steep/https%3A%2F%2Fsubstack-post-media.s3.amazonaws.com%2Fpublic%2Fimages%2Fc2bcdd47-be2e-48aa-90cb-671b28209d48_270x297.png)](https://substackcdn.com/image/fetch/$s_!-Yoj!,f_auto,q_auto:good,fl_progressive:steep/https%3A%2F%2Fsubstack-post-media.s3.amazonaws.com%2Fpublic%2Fimages%2Fc2bcdd47-be2e-48aa-90cb-671b28209d48_270x297.png" \t "_blank)

Each bit in the bitmap corresponds to a row, and the value of the bit indicates whether the key value exists in that row.

Bitmap indexes use a bitmap (a binary array) to represent the presence or absence of a specific key value in each row of a table.

Bitmap indexes are well-suited for columns with low cardinality (a small number of distinct values) and for performing complex queries involving multiple conditions.

Bitmap operations like AND, OR, and NOT are performed efficiently using bitwise operations, making bitmap indexes suitable for analytical queries involving multiple columns.

**6. How to use Database Indexes Smartly?**

To get the most out of database indexes, consider these best practices:

* **Identify Query Patterns**: Analyze the most frequent and critical queries executed against your database to determine which columns to index and which type of index to use.
* **Index Frequently Used Columns**: Consider indexing columns that are frequently used in WHERE, JOIN, and ORDER BY clauses.
* **Index Selective Columns:** Indexes are most effective on columns with a good spread of data values (high cardinality). Indexing a gender column might be less beneficial than one with a unique customer\_id.
* **Use Appropriate Index Types**: Choose the right index type for your data and queries.
* **Consider Composite Indexes**: For queries involving multiple columns, consider creating composite indexes that encompass all relevant columns. This reduces the need for multiple single-column indexes and improves query performance.
* **Monitor Index Performance**: Regularly monitor index performance, remove unused indexes and adjust your indexing strategy as the database workload evolves.
* **Avoid Over-Indexing**: Avoid creating too many indexes, as this can lead to increased storage requirements and slower write performance.
  + Indexes take up extra disk space since they're additional data structures that need to be stored alongside your tables.
  + Every time you insert, update, or delete data in a table with an index, the index needs to update too. This can slightly slow down write operations.

To summarize, indexes are a powerful tool to optimize database query performance.

But remember to choose the right column and index type, monitor performance, and avoid over-indexing to get the most out of them.

**Basic Questions**

**1. What is a database index?**  
👉 An index is a data structure (commonly a B-Tree or Hash) that improves the speed of data retrieval operations on a database table at the cost of additional writes and storage for maintaining the index.

**2. Why do we use indexes in databases?**  
👉 Indexes are used to:

* Speed up query performance (SELECT, WHERE, ORDER BY, JOIN).
* Reduce I/O by allowing the database to locate rows quickly.
* Support constraints like **UNIQUE**.

**3. What are the types of database indexes?**  
👉 Common types:

* **Clustered Index**: Defines the physical order of rows in the table (one per table).
* **Non-Clustered Index**: A separate structure pointing to table rows.
* **Unique Index**: Ensures uniqueness of values.
* **Composite Index**: Index on multiple columns.
* **Full-text Index**: Optimized for searching large text.
* **Hash Index**: Uses a hash table (common in NoSQL and memory-optimized databases).

**4. What is the difference between a clustered and non-clustered index?**  
👉 **Clustered Index**:

* Defines physical order of rows.
* One per table.
* Faster for range queries.

👉 **Non-Clustered Index**:

* Separate from table storage.
* Can be many per table.
* Needs lookups to fetch actual data.

**5. Can a table have multiple clustered indexes?**  
👉 **No.** A table can only have **one clustered index** because rows can only be stored in one order. However, it can have many **non-clustered indexes**.

**Intermediate Questions**

**6. What is a composite index?**  
👉 An index on multiple columns. Example:

CREATE INDEX idx\_customer\_name\_dob ON Customers(LastName, FirstName, DOB);

* Useful when queries use those columns together in **WHERE** or **ORDER BY** clauses.

**7. What is index selectivity? Why is it important?**  
👉 **Selectivity** = ratio of distinct values to total rows.

* High selectivity (close to 1) = good candidate for indexing.
* Low selectivity (e.g., Gender column with "M" or "F") = poor candidate.

**8. What is a covering index?**  
👉 A non-clustered index that contains all the columns required by a query.

* Query can be answered **entirely from the index** without accessing the base table (avoids key lookups).

**9. What are included columns in an index?**  
👉 Extra columns stored at the leaf level of a non-clustered index to avoid lookups.  
Example:

CREATE INDEX idx\_orders ON Orders(CustomerId) INCLUDE (OrderDate, TotalAmount);

**10. What are the disadvantages of indexes?**  
👉

* Extra storage overhead.
* Slower **INSERT, UPDATE, DELETE** (need to update indexes).
* Can cause fragmentation.
* Too many indexes may confuse the optimizer.

**Advanced Questions**

**11. What is index fragmentation? How do you fix it?**  
👉 **Fragmentation** occurs when data pages are not stored sequentially.

* Fix by using:

ALTER INDEX idx\_name REORGANIZE; -- defragmentation

ALTER INDEX idx\_name REBUILD; -- complete rebuild

**12. What is an index scan vs. index seek?**  
👉 **Index Seek**: Efficient, uses the index tree to find rows directly.  
👉 **Index Scan**: Reads all rows in the index, less efficient, like a table scan.

**13. What is a filtered index?**  
👉 An index created on a subset of rows using a **WHERE clause**.  
Example:

CREATE INDEX idx\_active\_users ON Users(Status) WHERE Status = 'Active';

**14. What are bitmap indexes?**  
👉 Used in data warehouses, a **bitmap index** uses bitmaps for values of a column.

* Very efficient for **low-cardinality columns** (e.g., Gender, Status).

**15. How do you decide which columns to index?**  
👉 Best practices:

* Index **frequently queried columns** in WHERE, JOIN, ORDER BY, GROUP BY.
* Use indexes on columns with **high selectivity**.
* Consider composite indexes for multi-column queries.
* Use covering indexes for performance-critical queries.
* Avoid indexing small lookup tables or columns with few distinct values.

**Scenario-Based Index Questions**

**1. Scenario:**  
You have a Users table with 5 million records. Queries often search by Email. Currently, queries are very slow.  
**Question:** What index would you create?  
**Answer:**  
Create a **unique non-clustered index** on Email since it is highly selective and usually unique.

CREATE UNIQUE NONCLUSTERED INDEX idx\_users\_email ON Users(Email);

**2. Scenario:**  
A query runs frequently:

SELECT FirstName, LastName, DOB

FROM Customers

WHERE LastName = 'Smith' AND FirstName = 'John';

**Question:** What index would you create?  
**Answer:**  
A **composite index** on (LastName, FirstName) because the query filters on both.

CREATE INDEX idx\_customer\_name ON Customers(LastName, FirstName);

**3. Scenario:**  
Your Orders table has 100 million rows. A report queries:

SELECT CustomerId, COUNT(\*)

FROM Orders

WHERE OrderDate BETWEEN '2025-01-01' AND '2025-01-31'

GROUP BY CustomerId;

**Question:** How would you optimize it?  
**Answer:**

* Create an index on OrderDate to speed up filtering.
* If queries also aggregate by CustomerId, a **composite index** (OrderDate, CustomerId) is better.

**4. Scenario:**  
A query uses WHERE Status = 'Active'. The Users table has 90% inactive users and only 10% active users.  
**Question:** What index would help?  
**Answer:**  
A **filtered index** for active users only.

CREATE INDEX idx\_active\_users ON Users(Status) WHERE Status = 'Active';

**5. Scenario:**  
You have a Products table with a column Category (only 5 possible values). Should you index it?  
**Answer:**  
No, because Category has **low selectivity**. Indexing won’t help much. Instead, index columns like ProductName or Price that are more selective.

**6. Scenario:**  
A query is:

SELECT \* FROM Employees ORDER BY HireDate DESC;

**Question:** What index helps?  
**Answer:**  
A **non-clustered index** on HireDate (descending) will help with sorting. If HireDate is queried often, consider making it the **clustered index**.

CREATE INDEX idx\_hiredate\_desc ON Employees(HireDate DESC);

**7. Scenario:**  
You have a table with frequent inserts (like Logs). Adding indexes slowed down inserts drastically.  
**Question:** How do you handle it?  
**Answer:**

* Minimize indexes (only essential ones).
* Use **clustered index on an auto-increment ID**.
* Partition the table (if supported).
* Consider archive strategy for old logs.

**8. Scenario:**  
A query is running slow:

SELECT \*

FROM Orders

WHERE CustomerId = 123 AND TotalAmount > 1000;

**Question:** Which index helps?  
**Answer:**  
A **composite index** (CustomerId, TotalAmount) since query filters by both.

**9. Scenario:**  
You have a Transactions table. Analysts frequently run queries like:

SELECT AVG(Amount)

FROM Transactions

WHERE PaymentType = 'CreditCard';

**Question:** Which index helps?  
**Answer:**

* A **filtered index** on PaymentType = 'CreditCard'.
* Or, if PaymentType has high selectivity, a normal non-clustered index on it.

**10. Scenario:**  
Your query plan shows **Index Scan** instead of **Index Seek** on a column you indexed.  
**Question:** Why, and how do you fix it?  
**Answer:**

* The query is not using the index efficiently (maybe using LIKE '%abc' or functions like UPPER()).
* Fix by rewriting query to use **sargable predicates** (e.g., LIKE 'abc%', no function wrapping).