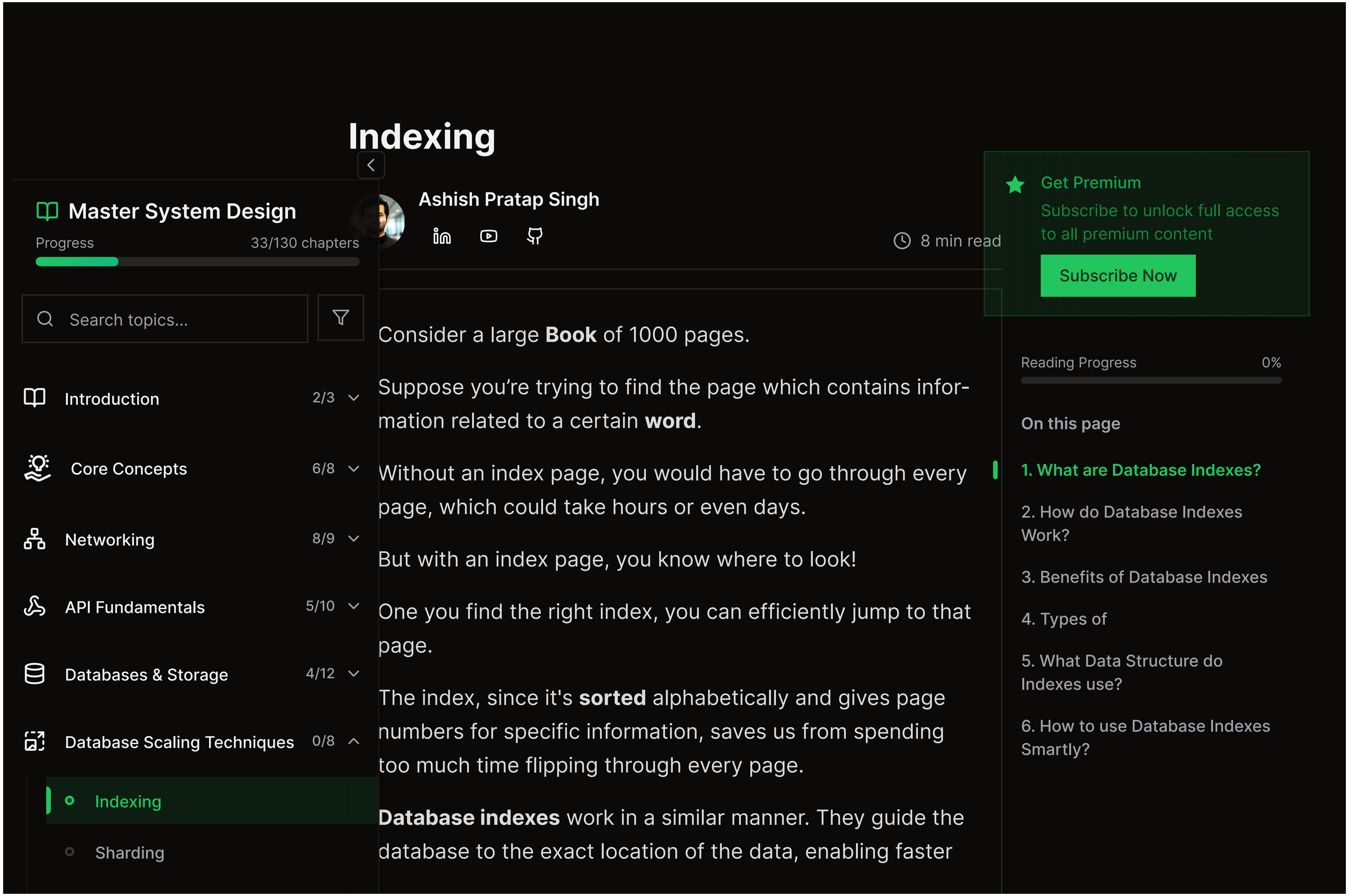


Indexing

Ashish Pratap Singh

8 min read

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.

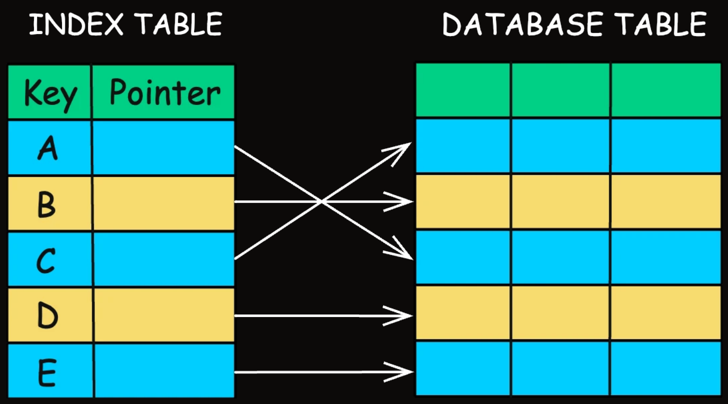
Database indexes work in a similar manner. They guide the database to the exact location of the data, enabling faster and more efficient data retrieval.

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 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:

Sql

```sql
CREATE TABLE employees (
    id INT PRIMARY KEY,
    first_name VARCHAR(50),
    last_name VARCHAR(50),
    email VARCHAR(100),
    department VARCHAR(50),
    salary DECIMAL(10, 2)
);
```

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.

Sql

```sql
-- Create index on last_name for faster lookups
CREATE INDEX idx_last_name ON employees (last_name);
```

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:

Sql

```sql
-- Query to find employees with last name 'Smith'
SELECT * FROM employees WHERE last_name = 'Smith';
```

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:

Sql

```sql
-- Create composite index on first_name and last_nar
CREATE INDEX idx_full_name ON employees (first_name,
```

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 data-base 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

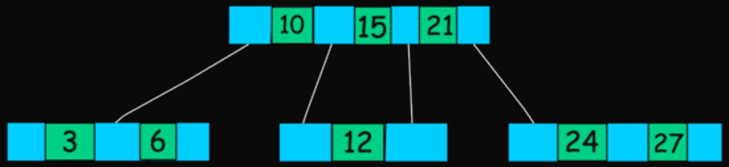
types.

---

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)



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:

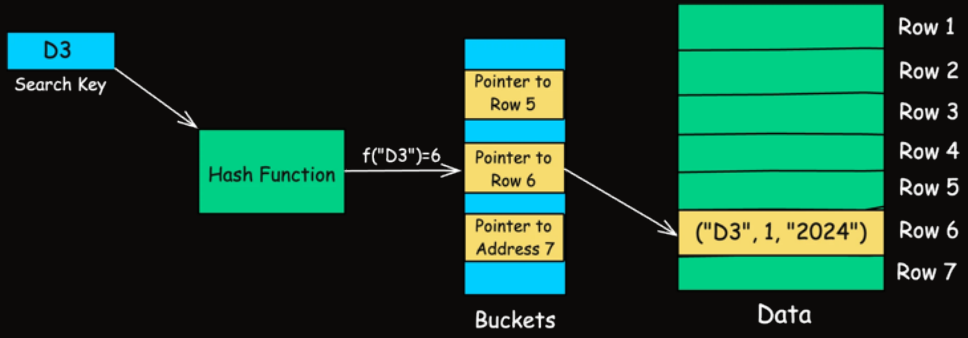
- **Self-Balancing:** B-trees ensure that the 'height' of the tree stays balanced even when inserting or deleting data. This ensures `logarithmic time complexity` for insertion, deletion, and searching.

- **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



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

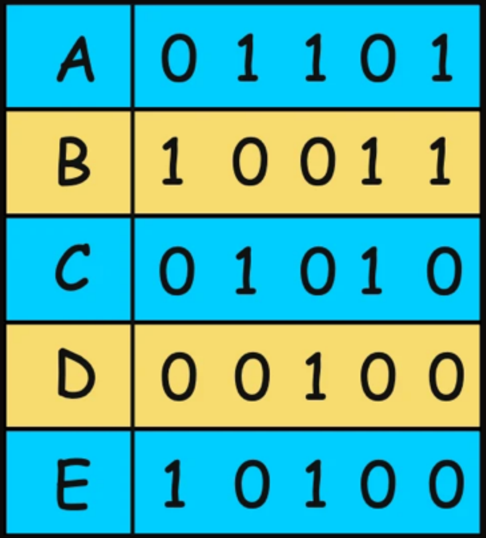
A hash table consists of an array of buckets, with each bucket containing the addresses for rows in the data.

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	0 1 1 0 1
B	1 0 0 1 1
C	0 1 0 1 0
D	0 0 1 0 0
E	1 0 1 0 0

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 col-

umns 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.