Database Indexes Explained

A detailed guide to understanding how database indexes work.

🔍 What Are Indexes?

At their core, indexes exist to speed up reads on a database table. The trade-off is that writes (inserts, updates, deletes) become slower. This is because every write must update not only the table's data but also the index structure itself.

This extra work involves maintaining a sorted data structure on disk, which has a logarithmic time complexity cost. The most common structures used are **B-Trees** (great for general-purpose reads and writes) and **LSM-Trees** (optimized for high write throughput).

PRO **Faster Reads**

Queries that filter on indexed columns (e.g., `SELECT * FROM Users WHERE Username = 'alex'`) are significantly faster by avoiding a full table scan.

CON **Slower Writes**

When you `INSERT`, `UPDATE`, or `DELETE` data, the database must update the index as well as the table, adding overhead.

Practical Example: Finding a User

Imagine a `Users` table with 10 million rows. You want to find one user by their email address with the query: `SELECT * FROM Users WHERE Email = 'user@example.com';`

UserID	Username	Email
1	alex	alex@email.com
...	...	...
5M	jordan	user@example.com
...	...	...

- **Without an index on `Email`:** The database must perform a full table scan, checking all 10 million rows one by one.
- **With an index on `Email`:** The database uses the index to find the exact location of the user's data in milliseconds.

🗂️ Index Abstraction

Conceptually, think of an index as a way to keep your table sorted by a key. For example, if you index a username column in a social media posts table, you're organizing posts alphabetically by username.

Now, when you search for a specific username, instead of scanning the whole table (O(n)), you can use a binary search (O(log n)). This is a massive performance boost for large datasets and is effective for both individual lookups and range queries.

Why This Matters

This mental model of a sorted list is crucial because it explains the performance gain. Searching through unsorted data requires checking every single item (linear time, O(n)). Searching through sorted data allows the database to jump to the middle, discard half the data, and repeat, leading to incredibly fast lookups (logarithmic time, O(log n)).

📚 Clustered vs. Non-Clustered Indexes

The two main types of indexes are clustered and non-clustered. They differ primarily in how they store data, which has significant performance implications.

Clustered Index

A clustered index determines the physical order of data in a table. The leaf nodes of the clustered index contain the actual data rows.

Example: Range Queries on Orders

An e-commerce dashboard needs to show all orders from yesterday: `WHERE OrderDate BETWEEN ...`. With a clustered index on `OrderDate`, all the order rows are physically stored together on disk in chronological order. The database can read them sequentially, which is extremely fast.

OrderDate	OrderID	CustomerID	Total
2025-06-26	1137	45	$50.00
2025-06-27	1138	82	$120.00
2025-06-27	1139	31	$85.50
2025-06-28	1140	76	$205.00

Non-Clustered Index (Secondary Index)

A non-clustered index has a structure separate from the data rows. It contains the keys in sorted order, with each key having a pointer to the data row.

Example: Filtering by Category

A blog needs to show all articles with the tag "Databases": `WHERE Tag = 'Databases'`. A non-clustered index on `Tag` can quickly find the pointers to all matching rows.

ArticleID	Title	Tag
1	Intro to SQL	SQL
2	Database Indexes	Databases
3	Learning React	WebDev
4	LSM-Trees Explained	Databases

Key Differences at a Glance

Attribute	Clustered Index	Non-Clustered Index
Number Per Table	Only one	Many
Data Storage	Stores data rows directly	Stores pointers to data
Performance	Faster for reads (no extra lookup)	Slightly slower (requires extra lookup)
Primary Use Case	Table's primary key; range scans	Frequently searched secondary columns

📖 Covered Index: The Middle Ground

A covered index is a special type of non-clustered index that offers a performance middle ground. In addition to the indexed key, it also includes a few extra columns from the table directly within the index itself.

Reasoning: Avoiding the Second Lookup

The goal of a covered index is to "cover" a query. If a query can get all the information it needs directly from the index (the `WHERE` clause key and all the columns in the `SELECT` list), it doesn't need to do the second step of looking up the full row in the main table. This makes it as fast as a clustered index for that specific query, without storing the *entire* row.

Example: Displaying a User List

Imagine a query to populate a user list in an admin panel: `SELECT UserID, Username, LastLogin FROM Users ORDER BY LastLogin DESC;`

A **covered index** on `(LastLogin, Username, UserID)` contains all the necessary data. The database can satisfy the entire query just by scanning the index, making it extremely fast.

📐 Composite vs. Multiple Single-Column Indexes

For queries that filter on multiple columns, you can create a composite index or multiple single-column indexes. Their behavior is very different.

Composite Index on (`Username`, `PostDate`)

This index sorts first by `Username`, and then by `PostDate` within each user's posts. It creates a single, unified, multi-level sorted structure.

Key Difference: Composite vs. Multiple Single Indexes

Imagine you have a query: `WHERE username = 'alex' AND postDate > '2023-02-01'`

- **With a Composite Index on `(username, postDate)`:** The database uses the single index to instantly locate the "alex" block and then efficiently scans the sorted dates within that block. This is highly efficient.
- **With two separate indexes on `username` and `postDate`:** The database has a harder choice. It might use the `username` index to find all "alex" rows and then manually filter them by date, or attempt a complex and often slow merge of both indexes.

Conclusion: For queries that consistently filter on the same set of columns, a single composite index is almost always more performant.

✅ Takeaway

Indexes are a classic systems design trade-off:

- **Pro:** Much faster reads.
- **Con:** Slower writes and extra storage overhead.

Good database design means picking the right indexes for your workload — often balancing between clustered, non-clustered, covered, and composite indexes depending on your specific query patterns.