# Database Indexing

Learn about how database indexing works and how to optimize your queries

Database performance can make or break modern applications. Think about what it takes to search for a user's profile by email in a table with millions of records. Without any optimizations, the database would have to check each row sequentially, scanning through every single record until it finds a match. For a table with millions of rows, this becomes painfully slow - like searching through every book in a library one by one to find a specific novel.

This is where indexes come in handy. By maintaining separate data structures optimized for searching, indexes allow databases to quickly locate the exact records we need without examining every row. From finding products in an e-commerce catalog to loading user profiles in a social network, indexes are what make fast lookups possible.

Knowing when to add an index, to what columns, and what type of index is a critical part of system design. Choosing the right indexes is often a key focus in interviews. For mid-level engineers, understanding basic indexing strategies is expected. For staff-level engineers, mastery of different index types and their trade-offs is essential.

This deep dive covers how database indexes work under the hood and the different types you'll encounter. We'll start with the core concepts of how indexes are stored and accessed, then examine specific index types like B-trees, hash indexes, geospatial indexes, and more. For each type, we'll cover their strengths, limitations, and when to use them in your system design interviews.

First, let's understand exactly how databases store and use indexes to make our queries efficient.

> Indexing and data organization tends to be a stronger focus in infrastructure style interviews. For full-stack and product-focused roles, you'll likely only need a basic understanding of when and why to use indexes. The depth we cover here goes beyond what's typically asked in full-stack interviews, but understanding the fundamentals will help you make better decisions when designing and optimizing your applications.

## How Database Indexes Work

When we store data in a database, it's ultimately written to disk as a collection of files. The main table data is typically stored as a heap file - essentially a collection of rows in no particular order. Think of this like a notebook where you write entries as they come, one after another.

### Physical Storage and Access Patterns

> Unless interviewing for a database internals role, the details here are not going to be asked in an interview. That said, they are an important foundation to understand the problem of why we need indexes.

Modern databases face an interesting challenge: they need to store and quickly access vast amounts of data. While the data lives on disk (typically SSDs nowadays), we can only process it when it's in memory. This means every query requires loading data from disk into RAM.

When querying without an index, we need to scan through every page of data one by one, loading each into memory to check if it contains what we're looking for. With millions of pages, this means millions of (relatively)slow disk reads just to find a single record. It's like having to flip through every page of a massive book to find one specific word.

> Modern databases have optimizations like prefetching and caching to make random access faster, but the point here still stands. It's too slow to scan through every page of data sequentially.

But with indexes, we transform our access patterns. Instead of reading through every page of data sequentially, indexes provide a structured path to follow directly to the data we need. They help us minimize the number of pages we need to read from storage by telling us exactly which pages contain our target data. It's the difference between checking every page in a book versus using the table of contents to jump straight to the relevant pages.

> While SSDs are the norm today, it's important to note that random access is still significantly slower than sequential access, even on SSDs. This is a common misconception - while the performance gap is smaller than with HDDs, it's still very real. And for systems still using HDDs, especially for large datasets, this performance difference becomes even more pronounced, making proper indexing absolutely critical.

### Cost

But indexes aren't free - they come with their own set of trade-offs. Every index we create requires additional disk space, sometimes nearly as much as the original data.

Write performance takes a hit too. When we insert a new row or update an existing one, the database must update not just the main table, but every index on it. With multiple indexes, a single write operation can trigger several disk writes.

So when might indexes actually hurt more than help? The classic case is a table with frequent writes but infrequent reads. Think of a logging table where we're constantly inserting new records but rarely querying old ones. Here, the overhead of maintaining indexes might not justify their benefit. Similarly, for small tables with just a few hundred rows, the cost of maintaining an index and traversing its structure might exceed the cost of a simple sequential scan.

> In reality, the impact of indexes on memory is often overblown. Modern databases have smart buffer pool management that reduces the performance hit of having multiple indexes. However, it's still a good idea to closely monitor index usage and avoid creating unnecessary indexes that don't provide significant benefits.

## Types of indexes

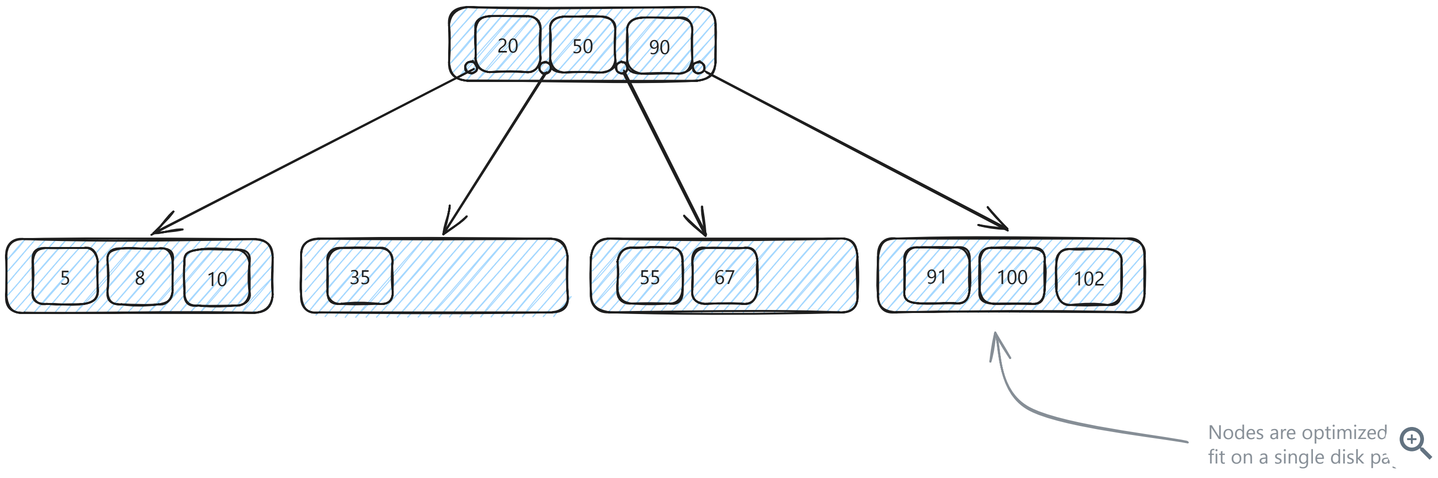
There are lots of indexes, many of which fall into the tail and are rarely used but for specialized use cases. Rather than enumerating every type of index you may see in the wild, we're going to focus in on the most common ones that show up in system design interviews.

### B-Tree Indexes

B-tree indexes are the most common type of database index, providing an efficient way to organize data for fast searches and updates. They achieve this by maintaining a balanced tree structure that minimizes the number of disk reads needed to find any piece of data.

#### The Structure Of B-Trees

A B-tree is a self-balancing tree that maintains sorted data and allows for efficient insertions, deletions, and searches. Unlike binary trees where each node has at most two children, B-tree nodes can have multiple children - typically hundreds in practice. Each node contains an ordered array of keys and pointers, structured to minimize disk reads.



b-tree

Every node in a B-tree follows strict rules:

- All leaf nodes must be at the same depth
- Each node can contain between m/2 and m keys (where m is the order of the tree)
- A node with k keys must have exactly k+1 children
- Keys within a node are kept in sorted order

This structure is particularly clever because it maps perfectly to how databases store data on disk. Each node is sized to fit in a single disk page (typically 8KB), maximizing our I/O efficiency. When PostgreSQL needs to find a record with id=350, it might only need to read 2-3 pages from disk: the root node, maybe an internal node, and finally a leaf node.

### Real-World Examples

B-trees are everywhere in modern databases. PostgreSQL uses them for almost everything - primary keys, unique constraints, and most regular indexes are all B-trees.

When you create a table like this in PostgreSQL:

```sql
CREATE TABLE users (
    id SERIAL PRIMARY KEY,
    email VARCHAR(255) UNIQUE
);
```

PostgreSQL automatically creates two B-tree indexes: one for the primary key and one for the unique email constraint. These B-trees maintain sorted order, which is crucial for both uniqueness checks and range queries.

DynamoDB organizes items within a partition in sort-key order, enabling efficient range queries within that partition. Its storage internals aren't publicly documented in detail, but it's widely understood to use an LSM-style storage architecture rather than a B-tree for its underlying engine.

Even MongoDB, with its document model, uses B-trees (specifically B+ trees, a variant where all data is stored in leaf nodes) for its indexes.

When you create an index in MongoDB like this:

```javascript
db.users.createIndex({ "email": 1 });
```

You're creating a B-tree that maps email values to document locations.

### Why B-Trees Are The Default Choice

B-trees have become the default choice for most database indexes because they excel at everything databases need:

1. They maintain sorted order, making range queries and ORDER BY operations efficient
2. They're self-balancing, ensuring predictable performance even as data grows
3. They minimize disk I/O by matching their structure to how databases store data
4. They handle both equality searches (email = 'x') and range searches (age > 25) equally well
5. They remain balanced even with random inserts and deletes, avoiding the performance cliffs you might see with simpler tree structures

If you find yourself in an interview and you need to decide which index to use, B-trees are a safe bet.

## LSM Trees (Log-Structured Merge Trees)

B-trees are great for balanced workloads, but what happens when you need to handle tons of writes? Think about building a system like DataDog that's ingesting millions of metrics per second from thousands of servers. Every CPU reading, memory stat, and error count needs to be stored immediately.