Systems Design Fundamentals

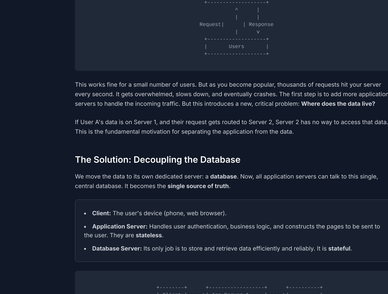
From One User to Millions

Introduction to Systems Design

At its core, **Systems Design** is the art and science of architecting the components of a large-scale application. The primary challenge arises from **scale**. A simple application that works perfectly for a hundred users will crash and burn under the load of a million users. System design is about planning for that success.

The Core Problem: The Limits of a Single Server

Imagine you build a social media site. Initially, you run it on a single computer (a server). This server does everything: it runs the application logic, communicates with users, and stores all the data (profiles, posts, photos) on its hard drive.



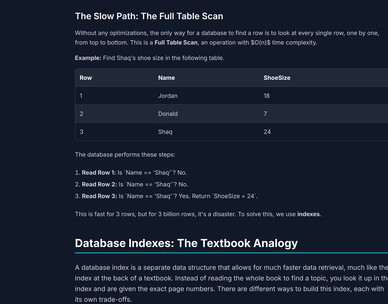
This works fine for a small number of users. But as you become popular, thousands of requests hit your server every second. It gets overwhelmed, slows down, and eventually crashes. The first step is to add more application servers to handle the incoming traffic. But this introduces a new, critical problem: **Where does the data live?**

If User A's data is on Server 1, and their request gets routed to Server 2, Server 2 has no way to access that data. This is the fundamental motivation for separating the application from the data.

The Solution: Decoupling the Database

We move the data to its own dedicated server: a **database**. Now, all application servers can talk to this single, central database. It becomes the **single source of truth.**

- **Client:** The user's device (phone, web browser).
- **Application Server:** Handles user authentication, business logic, and constructs the pages to be sent to the user. They are **stateless**.
- **Database Server:** Its only job is to store and retrieve data efficiently and reliably. It is **stateful**.



Now, no matter which App Server a user's request hits, it can get the correct data from the central database. This architecture is scalable and consistent. But this creates a new bottleneck: the database itself. The rest of this guide focuses on making the database fast and reliable.

Database Internals: The Quest for Fast Data Access

Simply storing data in a file isn't good enough. As the table grows to millions or billions of rows, finding the specific data you need becomes incredibly slow.

The Slow Path: The Full Table Scan

Without any optimizations, the only way for a database to find a row is to look at every single row, one by one, from top to bottom. This is a **Full Table Scan**, an operation with $O(n)$ time complexity.

Example: Find Shaq's shoe size in the following table.

Row	Name	ShoeSize
1	Jordan	18
2	Donald	7
3	Shaq	24

The database performs these steps:

1. **Read Row 1:** Is `Name == 'Shaq'`? No.
2. **Read Row 2:** Is `Name == 'Shaq'`? No.
3. **Read Row 3:** Is `Name == 'Shaq'`? Yes. Return `ShoeSize = 24`.

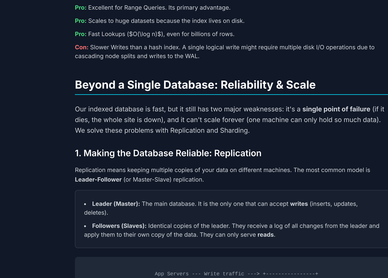
This is fast for 3 rows, but for 3 billion rows, it's a disaster. To solve this, we use **indexes**.

Database Indexes: The Textbook Analogy

A database index is a separate data structure that allows for much faster data retrieval, much like the index at the back of a textbook. Instead of reading the whole book to find a topic, you look it up in the index and are given the exact page numbers. There are different ways to build this index, each with its own trade-offs.

1. The Hash Index

A Hash index uses an in-memory **hash map** to map an indexed key directly to its location on disk. Its strength is incredibly fast direct lookups ($O(1)$).



Trade-offs of Hash Indexes:

Pro: Extremely Fast Reads & Writes ($O(1)$) for direct key lookups.

Con: The entire index must fit in memory (RAM), which is expensive and limited.

Con: No support for range queries. Finding names "between A and B" is impossible to do efficiently.

2. The B-Tree Index

The B-Tree (Balanced Tree) is the most common database index. It's a self-balancing tree structure stored entirely on disk. It's composed of fixed-size blocks called **pages** (or nodes), which contain sorted key ranges and references to other pages on disk. This structure is excellent for both direct lookups ($O(\log n)$) and range queries.

The Write Process: Node Splitting

The B-Tree maintains its balance through a clever write process. When adding a new key (e.g., "Betty"):

1. The tree is traversed to find the correct leaf page where "Betty" should reside.
2. **If the page has space:** The key is added, the page is re-sorted, and written back to disk. This is a relatively simple operation.
3. **If the page is full:** This is where it gets complex. The full page is **split** into two separate pages, and the data is divided between them. Then, the parent page must be updated with new key ranges and pointers to these two new child pages.

This splitting can cascade. If the parent page is also full, it must also split, and so on, potentially all the way up to the root. If the root splits, a new root is created, and the tree's height increases by one.

Durability and the Write-Ahead Log (WAL)

What happens if the computer crashes mid-update, after splitting the leaf but before updating the parent's pointer? The index is now corrupted. To prevent this, B-Trees use a **Write-Ahead Log (WAL)**.

Before any page on disk is modified, the intended change is first written to a simple, sequential log file (the WAL). If a crash occurs, the database can replay this log upon restart to bring the B-Tree back to a consistent and valid state.

Trade-offs of B-Trees:

Pro: Excellent for Range Queries. Its primary advantage.

Pro: Scales to huge datasets because the index lives on disk.

Pro: Fast Lookups ($O(\log n)$), even for billions of rows.

Con: Slower Writes than a hash index. A single logical write might require multiple disk I/O operations due to cascading node splits and writes to the WAL.

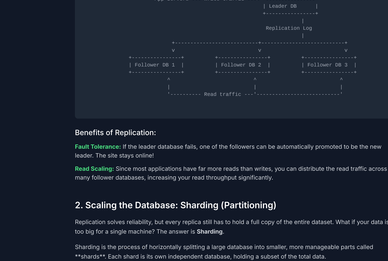
Beyond a Single Database: Reliability & Scale

Our indexed database is fast, but it still has two major weaknesses: it's a **single point of failure** (if it dies, the whole site is down), and it can't scale forever (one machine can only hold so much data). We solve these problems with Replication and Sharding.

1. Making the Database Reliable: Replication

Replication means keeping multiple copies of your data on different machines. The most common model is **Leader-Follower** (or Master-Slave) replication.

- **Leader (Master):** The main database. It is the only one that can accept **writes** (inserts, updates, deletes).
- **Followers (Slaves):** Identical copies of the leader. They receive a log of all changes from the leader and apply them to their own copy of the data. They can only serve **reads**.



Benefits of Replication:

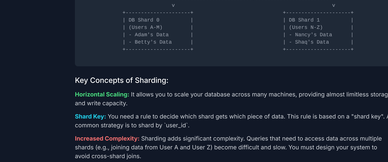
Fault Tolerance: If the leader database fails, one of the followers can be automatically promoted to be the new leader. The site stays online!

Read Scaling: Since most applications have far more reads than writes, you can distribute the read traffic across many follower databases, increasing your read throughput significantly.

2. Scaling the Database: Sharding (Partitioning)

Replication solves reliability, but every replica still has to hold a full copy of the entire dataset. What if your data is too big for a single machine? The answer is **Sharding**.

Sharding is the process of horizontally splitting a large database into smaller, more manageable parts called **shards**. Each shard is its own independent database, holding a subset of the total data.



Key Concepts of Sharding:

Horizontal Scaling: It allows you to scale your database across many machines, providing almost limitless storage and write capacity.

Shard Key: You need a rule to decide which shard gets which piece of data. This rule is based on a "shard key": A common strategy is to shard by `user_id`.

Increased Complexity: Sharding adds significant complexity. Queries that need to access data across multiple shards (e.g., joining data from User A and User Z) become difficult and slow. You must design your system to avoid cross-shard joins.