**System Design: How to Scale a Database**

Let’s say you are building an application that needs to store **user information**.

When your app has a few hundred users, you can keep all the data on one database server.

But as your app grows and you get more users, you need to store **more data**.

If you don't scale your database to handle the increased load, it can **slow down** your app and cause problems.

In this article we will explore **8 strategies** to scale databases to ensure your application keeps operating at optimal performance without the database becoming a bottleneck.

**1. Vertical Scaling**

[[A diagram of a green cylinder

AI-generated content may be incorrect.](https://substackcdn.com/image/fetch/$s_!aiMX!,f_auto,q_auto:good,fl_progressive:steep/https%3A%2F%2Fsubstack-post-media.s3.amazonaws.com%2Fpublic%2Fimages%2Ffe676127-7997-463f-8300-86cfee413354_1030x524.png)](https://substackcdn.com/image/fetch/$s_!aiMX!,f_auto,q_auto:good,fl_progressive:steep/https%3A%2F%2Fsubstack-post-media.s3.amazonaws.com%2Fpublic%2Fimages%2Ffe676127-7997-463f-8300-86cfee413354_1030x524.png" \t "_blank)

Vertical scaling involves adding more resources (CPU, RAM, storage) to a single database server.

It's a quick and easy solution when you have a **smaller database**, but it has limitations.

It can become **expensive**, and there's a limit on how much you can scale up.

Additionally, vertical scaling introduces a **single point of failure**, as all your eggs are in one basket.

**Example:** A small e-commerce website experiences increased traffic during a holiday sale. They vertically scale their database server by adding more RAM to handle the extra load.

**2. Indexing**

Indexes at the back of a book help you quickly find specific information without having to go through every page.

In the same way, **database indexes** help find data much faster without scanning every single row in a table.

Indexes are usually created on **most frequently queried columns** to make read requests faster, but over-indexing can slow down the write performance due to overhead.

**3. Sharding**

A single machine can only hold so much data.

It will run out of space and slow down as more people start using your application.

To avoid this, you can split the data into **smaller pieces (shards)** and store them on different servers.

This process is called **Database Sharding.**

[[A diagram of a server

AI-generated content may be incorrect.](https://substackcdn.com/image/fetch/$s_!EUKM!,f_auto,q_auto:good,fl_progressive:steep/https%3A%2F%2Fsubstack-post-media.s3.amazonaws.com%2Fpublic%2Fimages%2F95837cf8-add0-41c2-8222-025645c20f74_1902x576.png)](https://substackcdn.com/image/fetch/$s_!EUKM!,f_auto,q_auto:good,fl_progressive:steep/https%3A%2F%2Fsubstack-post-media.s3.amazonaws.com%2Fpublic%2Fimages%2F95837cf8-add0-41c2-8222-025645c20f74_1902x576.png" \t "_blank)

Distributing data in this way makes it easier to scale and handle more users.

To learn more about Database Sharding, you can read my previous article:

**4. Vertical Partitioning**

In situations where some columns are accessed more often than others, it’s a good idea to split the database table into smaller tables, each containing a subset of the columns from the original table.

This helps reduce the amount of data read during queries and can improve performance for specific access patterns.

**Example:** An e-commerce application might split their product table into:

* A "core\_product" table with frequently accessed data (ID, name, price, category)
* A "product\_details" table with less frequently accessed data (description, specifications)
* A "product\_media" table with large binary data (images, videos)

**5. Caching**

In almost every major application, some data is accessed more often than others.

For example:

* In a blogging app, some articles are read more often than others.
* In a social media app, some users use the platform more often than others.
* In a streaming platform, some movies are watched more often than others.

It's smart to store this frequently accessed data in a **faster storage layer** to speed up access and reduce the load on the database.

[[A diagram of a computer network

AI-generated content may be incorrect.](https://substackcdn.com/image/fetch/$s_!dMwl!,f_auto,q_auto:good,fl_progressive:steep/https%3A%2F%2Fsubstack-post-media.s3.amazonaws.com%2Fpublic%2Fimages%2F656e2de5-8755-485f-a6ff-560493336caa_1406x896.png)](https://substackcdn.com/image/fetch/$s_!dMwl!,f_auto,q_auto:good,fl_progressive:steep/https%3A%2F%2Fsubstack-post-media.s3.amazonaws.com%2Fpublic%2Fimages%2F656e2de5-8755-485f-a6ff-560493336caa_1406x896.png" \t "_blank)

This is referred to as **caching** and it’s a popular choice to speed up queries for frequently accessed data.

**6. Replication**

If your database servers are only located in one region, users from other regions may experience a higher latency.

To fix this, we can replicate the primary database to other regions and handle **read requests** locally.

This process is called **Database Replication.**

Simply put, database replication involves creating and maintaining multiple copies (replicas) of a database across different servers or locations to improve read performance, ensure high availability, and disaster recovery.

These replicas are synchronized with the original database (the primary), ensuring data consistency.

[[Diagram of a diagram of a computer

AI-generated content may be incorrect.](https://substackcdn.com/image/fetch/$s_!47zn!,f_auto,q_auto:good,fl_progressive:steep/https%3A%2F%2Fsubstack-post-media.s3.amazonaws.com%2Fpublic%2Fimages%2F637edf35-b087-4848-af7f-bd73f4175d7a_1232x1074.png)](https://substackcdn.com/image/fetch/$s_!47zn!,f_auto,q_auto:good,fl_progressive:steep/https%3A%2F%2Fsubstack-post-media.s3.amazonaws.com%2Fpublic%2Fimages%2F637edf35-b087-4848-af7f-bd73f4175d7a_1232x1074.png" \t "_blank)

**Types of Replication**:

1. **Synchronous Replication:** Changes made to the primary database are immediately replicated to all replicas before the transaction is considered complete. This ensures strong data consistency but can impact performance due to the additional overhead.
2. **Asynchronous Replication:** Changes to the primary database are replicated to replicas with a slight delay. This offers better performance but with the trade-off of potential data inconsistency between the primary and replicas (known as replication lag).

**7. Materialized Views**

Some database queries are complex and can take a long time to run.

This can slow down the performance of the application if these queries are run often.

But, what if we pre-compute and store the results of these complex and frequent queries?

This is the idea behind **Materialized Views.**

Materialized views are **pre-computed, disk-stored** result sets of complex queries.

Unlike regular views, which are virtual and computed on-the-fly, materialized views physically store the results, making them readily available for fast retrieval.

It significantly improves the query performance for **complex** and **resource-intensive** operations.

**Example:**

An e-commerce platform needs to generate daily sales reports that aggregate sales data by **date** and **product**. These reports are accessed frequently by the management team to make business decisions.

**Create Materialized View:**

CREATE MATERIALIZED VIEW **daily\_sales\_summary**

AS

SELECT date, product\_id, SUM(quantity) AS total\_quantity, SUM(amount) AS total\_amount

FROM sales

GROUP BY date, product\_id;

**Schedule Refresh:**

CREATE OR REPLACE SCHEDULE **job\_refresh\_sales\_summary**

ON SCHEDULE EVERY 1 DAY STARTS '2024-07-03 00:00:00'

DO

REFRESH MATERIALIZED VIEW daily\_sales\_summary;

**8. Data Denormalization**

Some database queries may involve **multiple tables** and **complex joins**.

These queries are often slow and can make the application slower for large tables.

To avoid this, we can add redundancy by combining **multiple tables into one** to reduce the need for complex joins.

This is called **Data denormalization.**

It is the process of intentionally introducing redundancy into a database to optimize read performance by combining tables or adding redundant data.

**Example:**

A social media platform has a normalized database schema with separate tables for **users**, **posts**, **comments**, and **likes**.

**Normalized Schema:**

CREATE TABLE **users** (

user\_id INT PRIMARY KEY,

username VARCHAR(100),

email VARCHAR(100)

);

CREATE TABLE **posts** (

post\_id INT PRIMARY KEY,

user\_id INT,

content TEXT,

created\_at TIMESTAMP,

FOREIGN KEY (user\_id) REFERENCES users(user\_id)

);

CREATE TABLE **comments** (

comment\_id INT PRIMARY KEY,

post\_id INT,

user\_id INT,

comment TEXT,

created\_at TIMESTAMP,

FOREIGN KEY (post\_id) REFERENCES posts(post\_id),

FOREIGN KEY (user\_id) REFERENCES users(user\_id)

);

The platform experiences high read traffic for user profiles and their associated posts and comments so it store posts and comments as JSON arrays within the user\_profiles table.

**Denormalized Schema:**

CREATE TABLE user\_profiles (

user\_id INT PRIMARY KEY,

username VARCHAR(100),

email VARCHAR(100),

posts JSON,

comments JSON

);

Each of these strategies has its own trade-offs in terms of complexity, consistency, and performance.

The best approach often involves a combination of these techniques, tailored to the specific needs and constraints of your application.

Thank you for reading!

**Core Concepts**

**1. Difference between vertical and horizontal scaling.**

* **Vertical scaling (scale up):** Add more resources (CPU, RAM, SSD) to a single server.  
  *Pros:* Simple, no code changes. *Cons:* Expensive, limited capacity, single point of failure.
* **Horizontal scaling (scale out):** Add more servers, e.g., sharding or replication.  
  *Pros:* High scalability, fault tolerance. *Cons:* Complex routing, data distribution.

**2. What is a database index, and how does it affect performance?**

* An index is a data structure (often a B-tree or hash) that speeds up searches.
* **Impact:** Faster reads, but slower writes (updates/inserts) due to index maintenance. Also uses extra storage.

**3. What are read replicas, and how do they help?**

* Read replicas are secondary copies of the database that sync from the primary.
* **Helps:** Offloads read queries, improves availability, supports geo-distributed reads.
* **Trade-off:** Replication lag → stale data.

**4. Difference between sharding and partitioning.**

* **Partitioning:** Splitting data within a single database node (e.g., by range or hash).
* **Sharding:** Distributing data across multiple database nodes/servers.
* All sharding is partitioning, but not all partitioning is sharding.

**5. Normalization vs. denormalization trade-offs.**

* **Normalization:** Reduces redundancy, improves consistency, but requires joins (slower reads).
* **Denormalization:** Speeds up reads by duplicating data, but increases write complexity and risk of inconsistency.

**Performance & Architecture**

**6. Identify and fix a slow query.**

* Steps:
  1. Enable slow query log / use EXPLAIN.
  2. Check for missing indexes.
  3. Optimize joins, avoid SELECT \*, limit rows.
  4. Consider caching or precomputing results.

**7. How caching helps and invalidation strategies.**

* **Helps:** Reduces DB load by serving hot data from memory (e.g., Redis).
* **Invalidation strategies:**
  + Time-to-live (TTL)
  + Write-through (update cache on write)
  + Cache-aside (lazy loading + manual invalidation)
  + Write-behind (async cache update)

**8. What is a materialized view, and when to use it?**

* A materialized view stores the result of a query physically.
* **Use:** For expensive aggregations or joins that don’t change often (e.g., daily sales totals).
* **Trade-off:** Must refresh periodically; may show stale data.

**9. Avoid hotspots when sharding.**

* Choose high-cardinality shard keys.
* Add randomness to keys (e.g., hash order IDs).
* Use time bucketing for time-series but avoid putting all recent writes in one shard.
* Distribute write load evenly.

**10. Eventual consistency and where it’s acceptable.**

* Means data may not be immediately up-to-date across replicas.
* Acceptable in:
  + Social media feeds
  + Analytics dashboards
  + Caching layers
* Not acceptable for:
  + Banking transactions
  + Inventory tracking in e-commerce

**Scenario-based**

**11. 90% reads, 10% writes, reads are slow.**

* Add read replicas.
* Introduce caching for hot queries.
* Create proper indexes.
* Use materialized views for heavy aggregates.

**12. Primary DB at 80% CPU & storage limits, no downtime allowed.**

* Move reporting/analytics queries to replicas.
* Add read replicas immediately.
* Partition large tables.
* Offload static data to cheaper storage (e.g., S3).
* Plan for sharding if growth continues.

**13. Global e-commerce writes from multiple regions.**

* Options:
  + **Multi-leader replication** (conflict resolution needed).
  + **Regional leader + async replication** for non-critical writes.
  + Use distributed DB (e.g., Spanner, Cosmos DB) for strong global consistency.

**14. Analytics queries slowing OLTP.**

* Move analytics to a separate OLAP database (Snowflake, BigQuery, Redshift).
* Use CDC (Change Data Capture) to sync from OLTP to OLAP in near real-time.

**15. Billions of rows, slow queries.**

* Add proper indexes and partitioning (range/hash).
* Archive old/historical data.
* Precompute common aggregates into summary tables.
* Use columnar storage for analytical queries.

**1. Sudden Traffic Spike During Flash Sale**

**Scenario:**  
Your e-commerce site’s Orders table is slowing down checkout during a flash sale. CPU is at 90%, writes queue up, and payment processing is delayed. Profiling shows lock contention and long-running read queries.

**Answer:**

* **Diagnosis:**
  + High write volume on Orders + read queries scanning large ranges.
  + Secondary indexes are slowing inserts.
  + Payment workflow holds DB transactions open too long.
* **Options:**
  + Move non-critical reads (order history) to **read replicas** or cache.
  + Use **append-only writes** for orders, then process in background.
  + Remove or delay maintenance of non-essential indexes during flash sales.
  + Implement **write queue/outbox pattern** for payment and fulfillment events.
* **Trade-offs:**
  + Removing indexes improves write throughput but may slow some reads.
  + Eventual consistency acceptable for email notifications but not for payment confirmation.
* **Recommendation:**
  + Immediately **add Redis cache** for hot reads.
  + Shift analytics/reporting to replicas.
  + Schedule index rebuilds for off-peak hours.
  + In long term, **shard Orders table by customer\_id** to distribute write load.

**2. Large Chat Application with Latency Complaints**

**Scenario:**  
Your chat app stores all messages in a single PostgreSQL table with 2 billion rows. Users report delays when loading the last 50 messages in large group chats.

**Answer:**

* **Diagnosis:**
  + Large table → index pages don’t fit in memory.
  + Queries scan too many rows before finding latest messages.
* **Options:**
  + **Partition table by conversation\_id** (or hash of conversation\_id).
  + Maintain **per-conversation “inbox” table** storing only latest N messages.
  + Use **Redis sorted sets** keyed by conversation for latest messages.
* **Trade-offs:**
  + Caching is fast but needs invalidation.
  + Partitioning reduces index size per partition, improving locality.
* **Recommendation:**
  + Short term: Redis cache for last 50 messages with async write to DB.
  + Medium term: Partition + covering index (conversation\_id, created\_at DESC).

**3. OLTP Performance Impacted by Analytics**

**Scenario:**  
The marketing team runs daily reports directly on your production MySQL DB. CPU spikes and API response times degrade.

**Answer:**

* **Diagnosis:**
  + OLTP and analytics workloads are mixed.
  + Analytical queries (full table scans, group by) block transactional queries.
* **Options:**
  + Create **read replicas** dedicated to analytics.
  + Implement **Change Data Capture (CDC)** to stream data to a warehouse (Snowflake/BigQuery).
  + Build **materialized views** for repetitive aggregates.
* **Trade-offs:**
  + Replicas give near-real-time analytics but can lag.
  + Data warehouses decouple completely but add ETL complexity.
* **Recommendation:**
  + Long term: CDC → OLAP warehouse.
  + Short term: Dedicated replica + query governor to stop long scans on OLTP.

**4. Global User Latency Issue**

**Scenario:**  
Your SaaS product serves global customers from a US-hosted primary DB. Asian users see ~500ms latency per write.

**Answer:**

* **Diagnosis:**
  + Network round-trip delay due to single write leader in US.
* **Options:**
  + Deploy **multi-region leaders** with conflict resolution (e.g., CRDTs, last-write-wins).
  + Use **regional write partitions** (tenant pinned to region).
  + Use globally distributed DB (Spanner/CosmosDB strong consistency).
* **Trade-offs:**
  + Multi-leader increases complexity due to conflicts.
  + Regional partitioning works if tenants don’t need cross-region collaboration.
* **Recommendation:**
  + Partition tenants by region, use async replication to US.
  + For global collaborative features, isolate into CP-consistent DB.

**5. Hotspot on Inventory Table**

**Scenario:**  
Inventory table has one row per SKU. Black Friday causes heavy updates to popular SKUs, leading to row-level lock contention.

**Answer:**

* **Diagnosis:**
  + Single row per SKU → all updates block each other.
* **Options:**
  + Use **sharded inventory buckets** (sku\_id, bucket\_id).
  + Use **optimistic locking** with version checks + retries.
  + Process decrements in **in-memory counters (Redis)** and batch to DB.
* **Trade-offs:**
  + Buckets reduce contention but require sum aggregation.
  + Redis is fast but risks losing data without persistence.
* **Recommendation:**
  + Short term: UPDATE ... WHERE sku\_id=? AND version=? to avoid blocking.
  + Long term: bucketed counters + async persistence.

**6. Large Hotel Booking Search Slow**

**Scenario:**  
Checking availability for a date range is slow because Reservations table has 1B+ rows.

**Answer:**

* **Diagnosis:**
  + Overlap queries on large range dataset are costly.
* **Options:**
  + Precompute **availability table** per room per date.
  + Use **range indexes** (Postgres GiST).
* **Trade-offs:**
  + Availability table is faster but increases write operations.
  + GiST indexing is elegant but can bloat storage.
* **Recommendation:**
  + Availability table + batch update on booking/cancel.

**7. Stale Data on Read Replicas**

**Scenario:**  
You added read replicas to handle load, but users see stale data right after updates.

**Answer:**

* **Diagnosis:**
  + Async replication delay (replica lag).
* **Options:**
  + Route post-write reads to primary for that session.
  + Use **synchronous replication** for critical tables.
  + Implement **read-your-writes consistency** with GTID-based wait.
* **Recommendation:**
  + Session stickiness to primary for few seconds after write.

**8. Big Table Performance Drop**

**Scenario:**  
Product catalog table is 10TB, but only 5% of items are “hot.” Queries are slow because indexes don’t fit in memory.

**Answer:**

* **Diagnosis:**
  + Cold data polluting cache, causing disk I/O for hot queries.
* **Options:**
  + Partition by last\_updated or popularity flag.
  + Use **partial indexes** on hot subset.
  + Cache hot products in Redis.
* **Recommendation:**
  + Partition + Redis for hot set; move cold set to cheaper tiered storage.

**9. Billions of Rows in Analytics**

**Scenario:**  
You need to count “active rides” every second for a dashboard. Querying the rides table is too slow.

**Answer:**

* **Diagnosis:**
  + Full table scans on high-volume table.
* **Options:**
  + Maintain real-time counters in Redis.
  + Stream ride start/stop events to Kafka, aggregate in Flink, persist to small table.
* **Recommendation:**
  + Stream-based aggregation + cache results for dashboard.

**10. Shard Rebalancing Needed**

**Scenario:**  
Your sharded DB has uneven load — some shards are huge, some are small.

**Answer:**

* **Diagnosis:**
  + Poor shard key choice, maybe low-cardinality.
* **Options:**
  + Migrate to **consistent hashing**.
  + Split largest shards (requires app routing logic).
* **Recommendation:**
  + Migrate with dual-write + validation phase, then cut over.