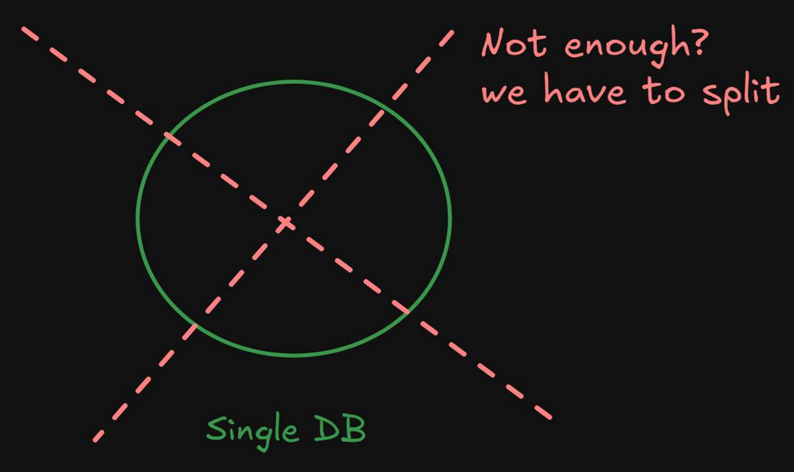
Introduction — Why Sharding Even Exists

Your app starts small.
One database is more than enough.
Life is good.

As users grow, you scale up the DB:
bigger CPU
more RAM
faster SSD
larger instance type

And this works... up to a point.

Then one day: Queries slow down
Writes pile up
Storage hits its upper limits
Replication lag becomes insane

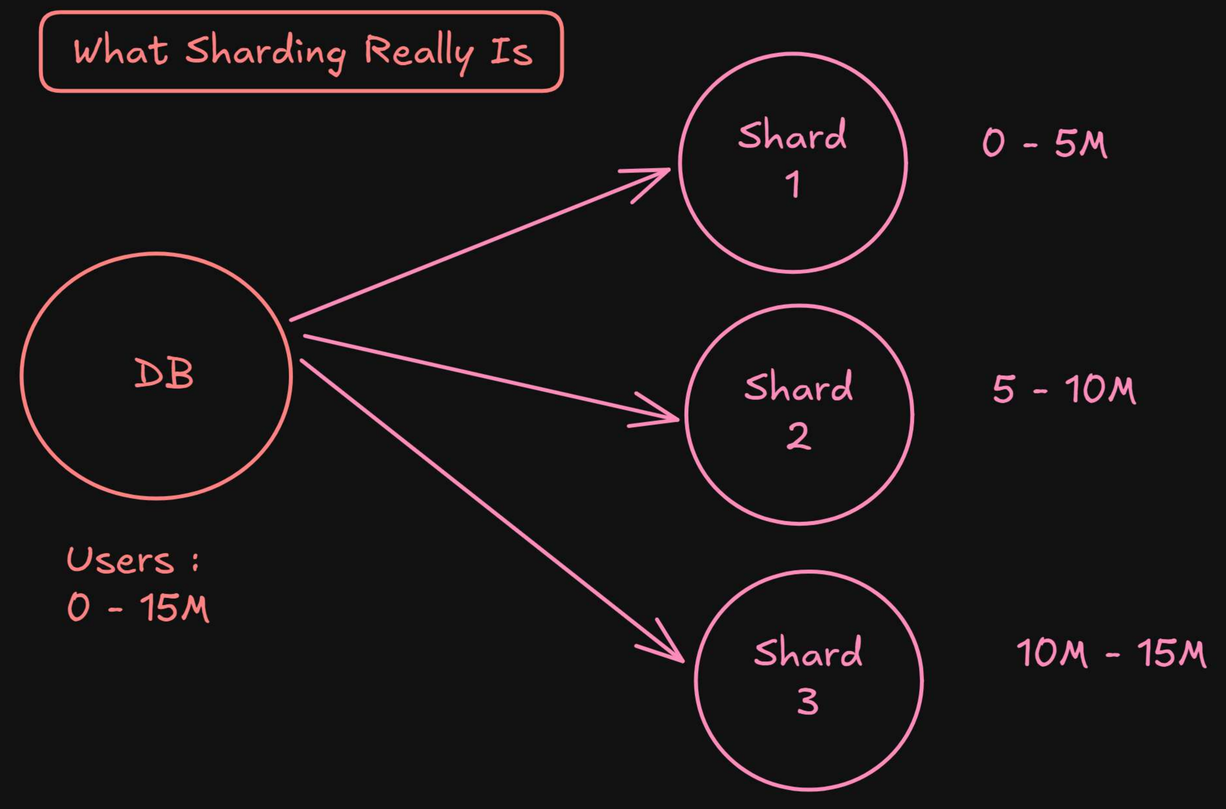


Even giant cloud machines eventually hit a hard ceiling.
At some point you face the ultimate truth: One machine cannot handle everything.

So you split your data across multiple machines.

And that is sharding.

What Sharding Really Is



Sharding = horizontal slicing of your data
into multiple independent databases.
Each shard is:
-> a separate DB instance
-> with its own CPU/RAM
-> its own storage
-> its own query capacity

All shards together = your full dataset.

How it's different from:

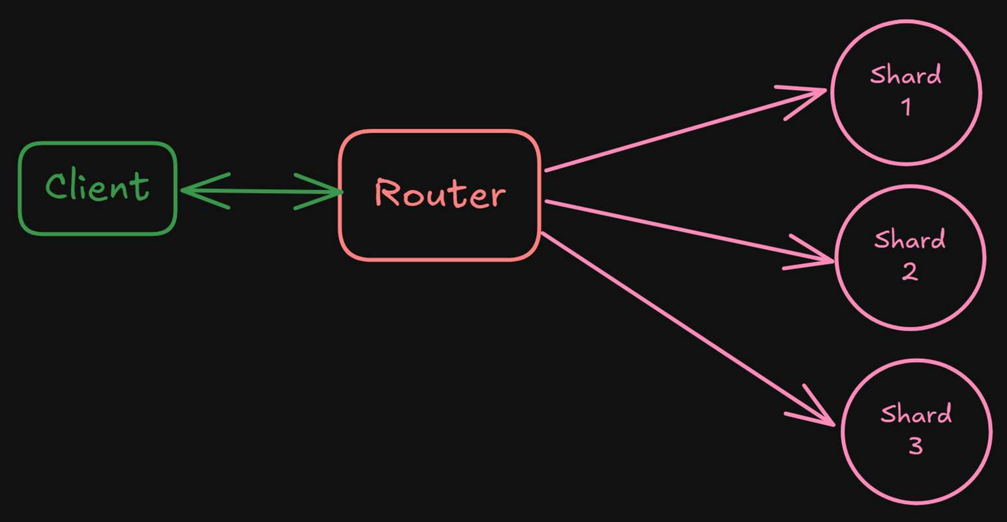
Simple partitioning:

Many SQL databases let you partition a
big table into chunks (by range, hash, list),
but all partitions still live inside one database instance.
Storage and CPU are still limited by that single box.

Replication:

Replication copies the same data to multiple
machines for read scaling and fault tolerance.
Sharding splits data so that each record exists
on only one primary shard (ignoring replication for now).

Basic Terminology & Components



Shard: One physical database containing a subset of data.

Shard key: Field(s) used to decide which shard a
row goes to. (e.g. user_id, tenant_id).

Shard map / metadata: Mapping from shard key → shard location.

Router / Coordinator:
Component that uses the shard key and routes queries to the correct shard.

Could be:

In the app code

A separate service

Built into database (e.g., MongoDB config servers + mongos).

Two core decisions when you shard

When you say "we'll shard", you're really answering two questions:

1. Shard key - — What decides where a row goes?
   (Which column/field: user_id, account_id, city_id, etc.)

2. Distribution strategy — How do we map values to shards?

   -> Range rules? (0–5M → shard 1)
   -> Hash rule? (hash(user_id) % N)
   -> Directory / lookup table?

If either of these is wrong, you get:

-> Unbalanced shards (one is huge, others tiny)
-> "Hot" shards serving almost all traffic
-> Queries that must hit every shard

Choosing a shard key

Think of shard key choice as the most important sharding decision

1. High cardinality (Must have LOTS of unique values)

The shard key should have many distinct values,
so the database can spread data across many shards.

-> A boolean field (is_active, is_premium) has only 2 values → BAD.
-> User IDs, order IDs, device IDs → millions of values → GOOD.

2. Uniform distribution

The shard key should spread rows roughly evenly across shards.

Values are not super skewed (e.g., not 90% of users in one country).

Each shard ends up with roughly similar data + traffic.

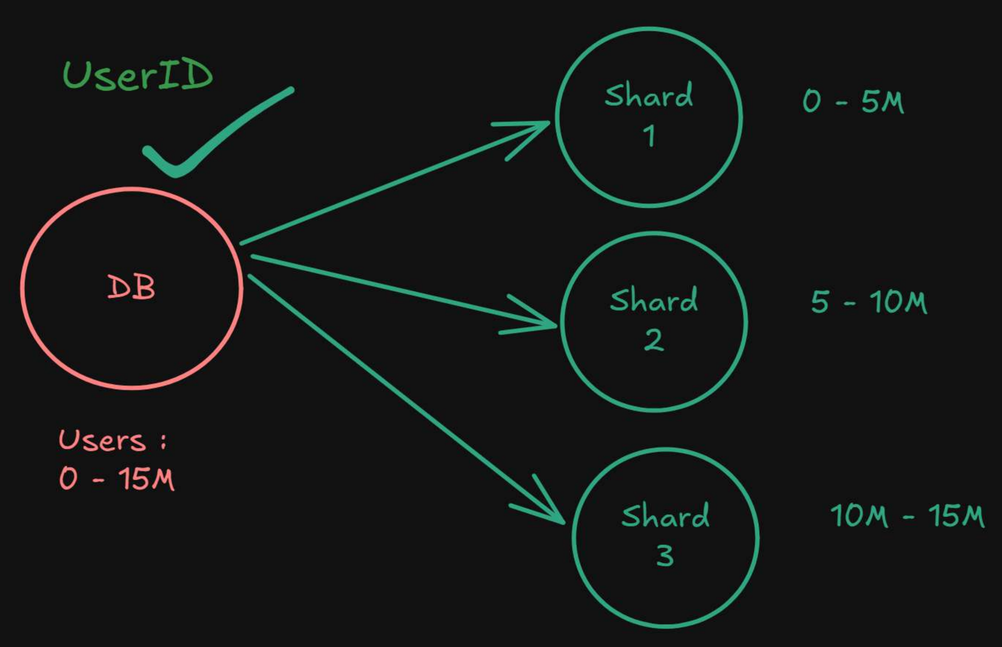
3. Stability
Key shouldn't change often.

If email is shard key and user changes email,
you must move row to another shard.

4. Matches query patterns

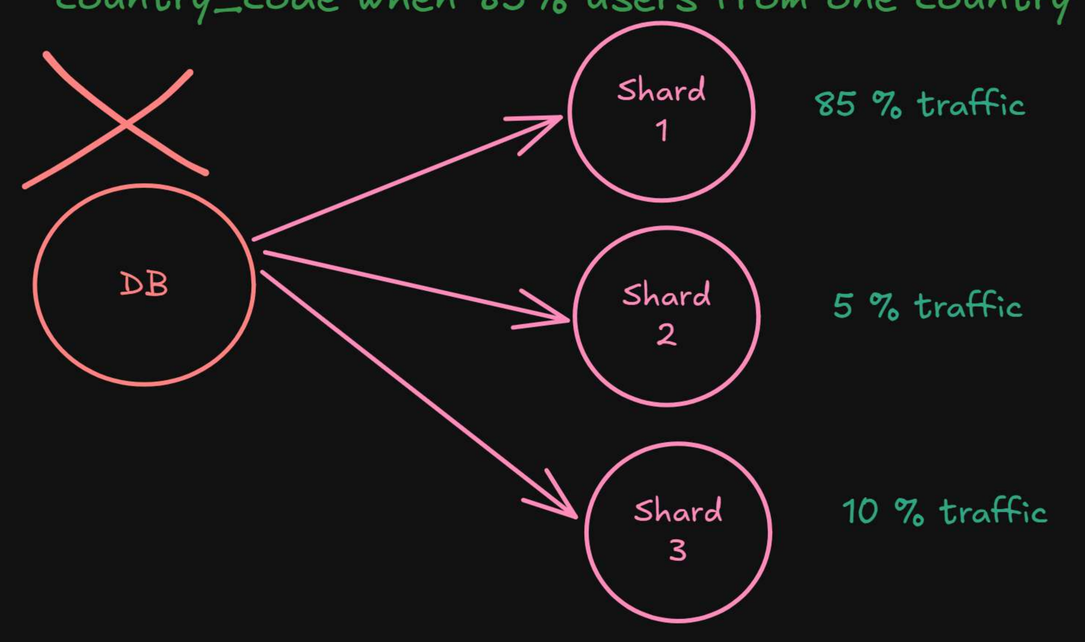
Most frequent queries can be answered from one shard.
Example: "get all orders for this user" → shard by user_id.
Queries that goes to all shards become expensive.

Example of good shard key

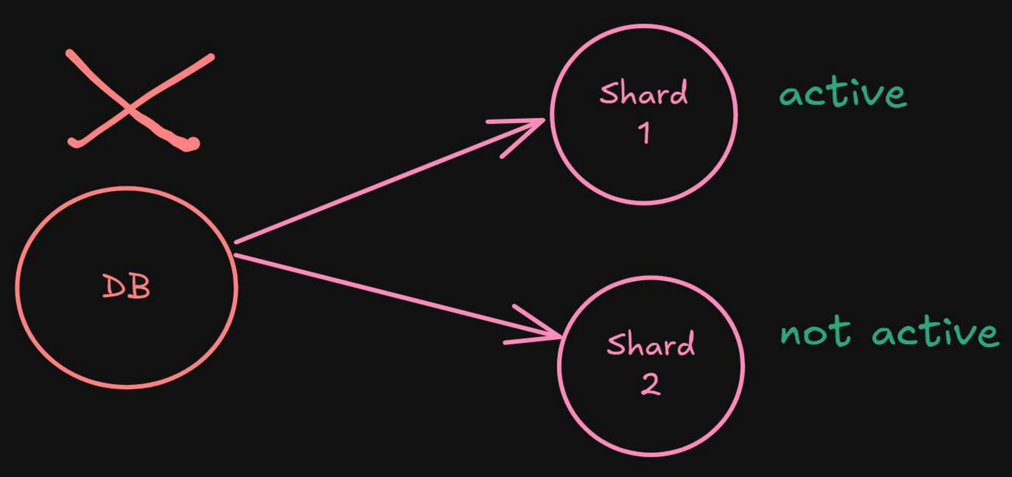


Example of bad shard key

country_code when 85% users from one country



Example of bad shard key
boolean is_active:
Only 2 possible values → effectively 2 shards → pointless



Sharding strategies

Now, assume we picked a shard key (say user_id).
Next, we decide how to map values to shards.

We'll cover three main patterns:

1. Range-based sharding
2. Hash sharding (usually default)
3. Directory / lookup sharding
4. Geo-Sharding

1. Range-based sharding

Group rows based on contiguous value ranges.

Pros

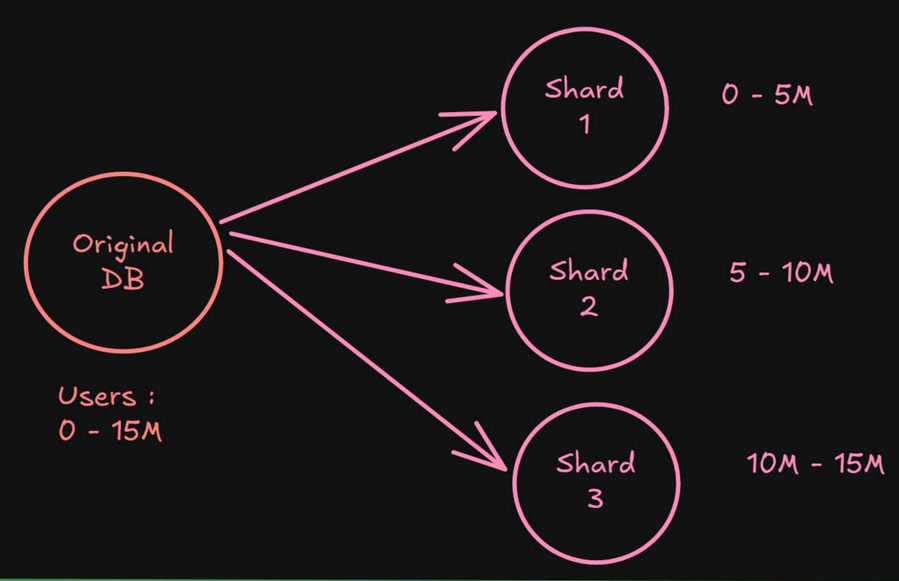
-> Simple to reason about.
-> Great for range queries:
"all orders between order_id 3M and 4M"
→ maybe a single shard.
"logs from 1st–3rd April" if you range by time.

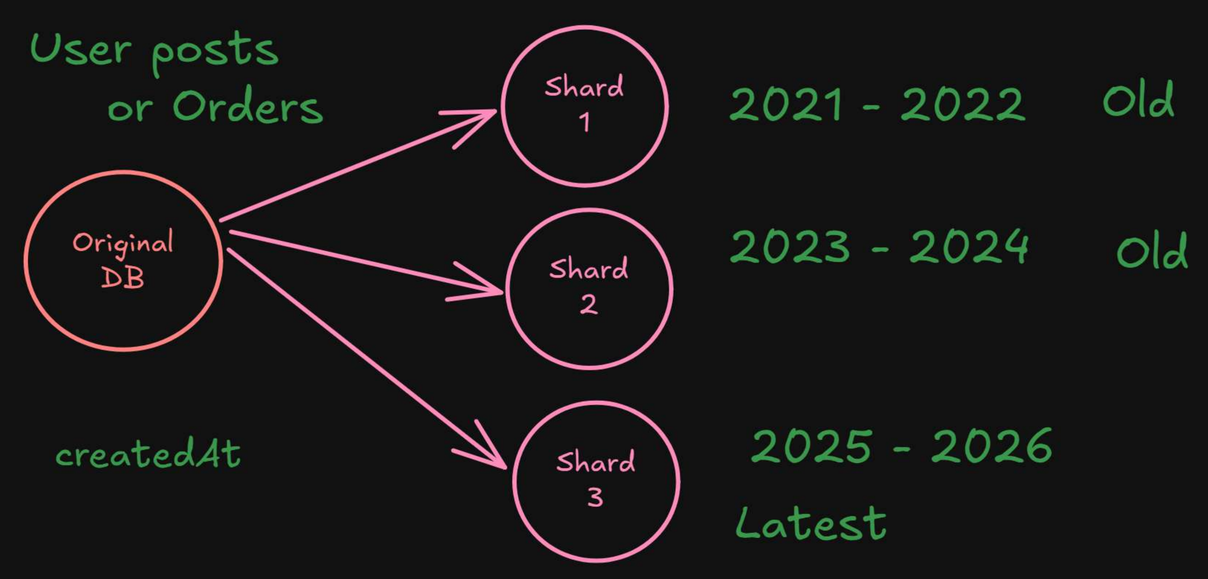
Cons

-> Real traffic is rarely uniform by range.
-> Time-based ranges create a hot "latest" shard": all new inserts target it.

Where it works well

Multi-tenant SaaS where you manually assign ranges per customer:
Tenant A → IDs 1–1M,
Tenant B → 1M–2M ...
Each tenant mostly lives in its own range → better balance.





2. Hash-based sharding (most common)

Apply a hash function on the shard key, then map hash to shard.

shard_id = hash(key) % NUM_SHARDS

Pros

-> Very good at spreading load evenly.
-> New users are naturally scattered across all shards.

Cons (classic modulo hashing)

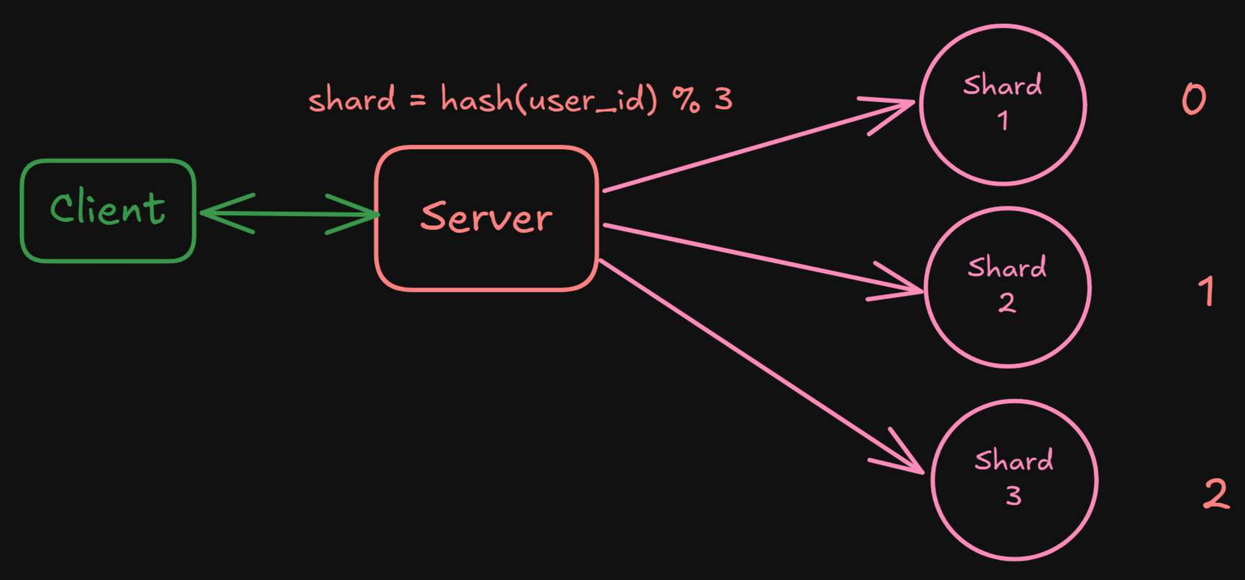
If you go from 3 → 4 shards, hash % 3 becomes hash % 4, almost every key maps to a different shard → need huge data reshuffle.

Enter consistent hashing (high level)

Instead of simple % N, consistent hashing maps both shard IDs and keys to points on a logical ring. When you add/remove a shard, only keys falling into its section move.

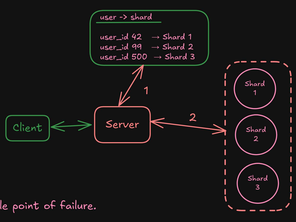
For interviews you can say:

"We'll use hash-based sharding with consistent hashing so that when we add shards, only a fraction of keys need to move."



3. Directory / lookup-table sharding

Here, you keep a mapping table
that tells you where each key lives.



Pros

Super flexible:

-> You can move a heavy user / tenant to a
dedicated shard just by updating the directory.
-> You can manually rebalance specific keys or ranges.

Cons

-> Every request needs an extra lookup → added latency.
-> Directory becomes a critical dependency and potential single point of failure.
-> Needs its own replication, caching, HA.

Usage

Good for "special cases" (e.g., very high-volume tenants)
or when you need fine-grained control.

In interviews, it's rarely your first answer because of complexity.

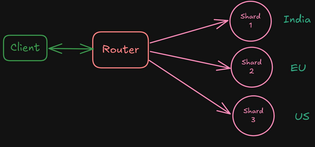
4. Geo-Sharding

Split by region:

India → Shard 1 (Mumbai)

Europe → Shard 2 (Frankfurt)

US → Shard 3 (Ohio)



Pros

Users hit nearest data center

Fewer cross-region writes

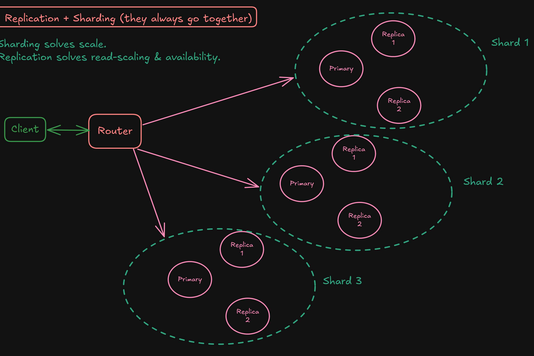
Cons

Complicated global queries

User movement between regions tricky

Replication + Sharding (they always go together)

Sharding solves scale.
Replication solves read-scaling & availability.

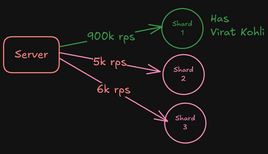


Sharding Challenges

Hotspots

Even with decent sharding, some shards may get hammered.

Example: You shard by creator_id in a short-video app.
One creator becomes a megastar → all their follows,
likes, comments, video views hit the same shard.



Causes of hotspots

-> "Celebrity / whale" keys
A few keys (creators, popular games, mega-tenants) generate massive traffic.

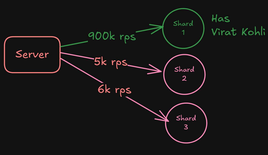
-> Time-based shards with all writes going to latest shard
Shard for the current day/month gets all writes; older shards only occasional reads.

-> Skewed business usage
One tenant uses your app heavily, others mildly.

How to detect hotspot

Monitor metrics like QPS, CPU Usage, Req. Volume:
If one shard constantly shows much higher load → hotspot.

Ways to handle hotspots



1. Move hot keys to their own shard(s)
   -> For example, create a "VIP creators" shard
   and move the top 100 creators there.

   -> Directory-based mapping is handy here.

2. Compound shard keys
   -> Example: hash(creator_id, day) instead of just creator_id
   -> A single creator's data is split by time across multiple shards.
   -> Write load spreads out instead of piling on one machine.
   -> Trade-off: reading "all history" for that creator may now be cross-shard.

3. Automatic shard splitting / rebalancing

   Some systems support splitting hot ranges or
   moving partitions between nodes in the background.

   Conceptually: detect "Shard X too large / too hot"
   → split its range or move some partitions to other machines.

Cross-shard queries

If a query's filter doesn't match the shard key,
it may need to hit many or all shards.

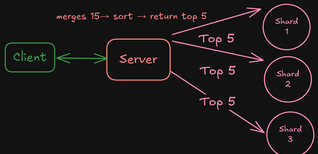
Example:
You shard posts by author_id.

get posts by author 123 → one shard (great).

top 5 posts across the whole platform →
data is scattered across all shards.

This is often called scatter–gather:

Scatter request to N shards.
Wait for all responses.
Merge, sort, aggregate, return.

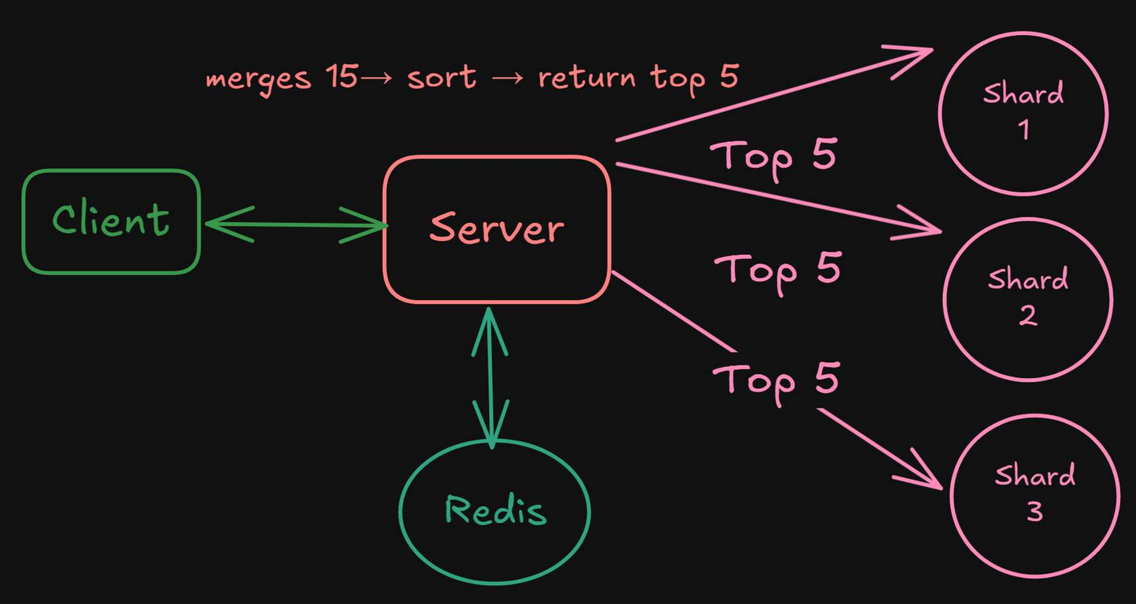


Problems: Latency

More chances of partial failures

Complex error handling and timeouts

How to deal with cross-shard queries



1. Cache heavy global queries

-> e.g., "trending posts", "top creators", "global stats"
-> Store in Redis / in-memory cache
for a short TTL (e.g., 30s, 1 min, 5 min).

2. Precompute with background workers

-> Periodic jobs read from shards, compute metrics, push results to a central summary table or cache.

3. Denormalize / duplicate data

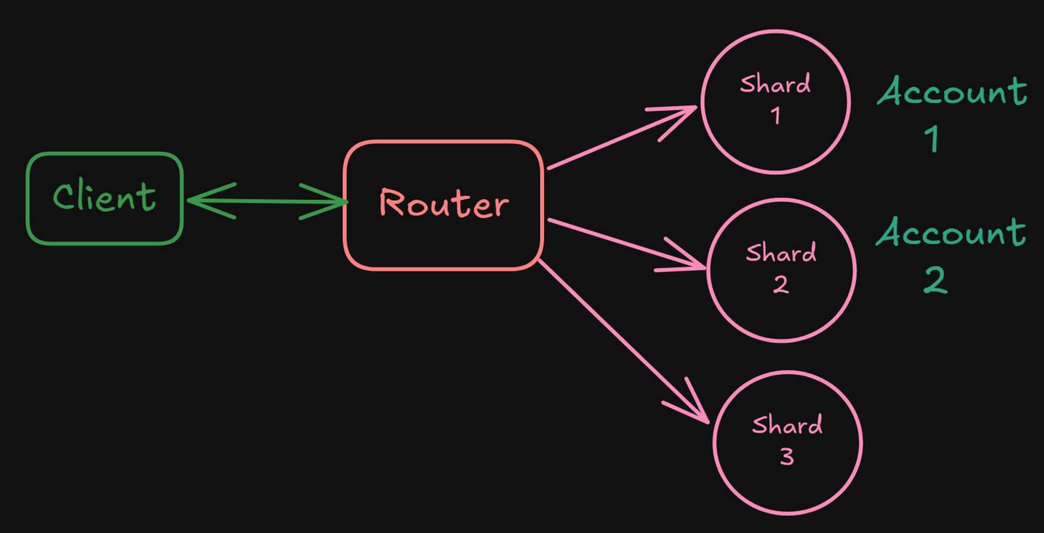
denormalize frequently accessed fields

4. Accept slow rare queries

Interview tip:
If you find yourself saying "we'll query all shards and aggregate" for very common requests, that's a red flag. Try to:

Change data model, or
Change shard key, or
Use caching / precompute / denormalization.

Consistency & transactions in a sharded world



On a single database, multi-row transactions are easy:

```
BEGIN;
UPDATE accounts SET balance = balance - 100 WHERE id = 1;
UPDATE accounts SET balance = balance + 100 WHERE id = 2;
COMMIT;
```

The DB guarantees: either both succeed or both fail.

When the two accounts are on different shards, life gets tricky.

Two-phase commit (2PC)

High-level steps:

Prepare phase — coordinator asks each shard:
"Can you commit this? Lock resources and say yes/no."

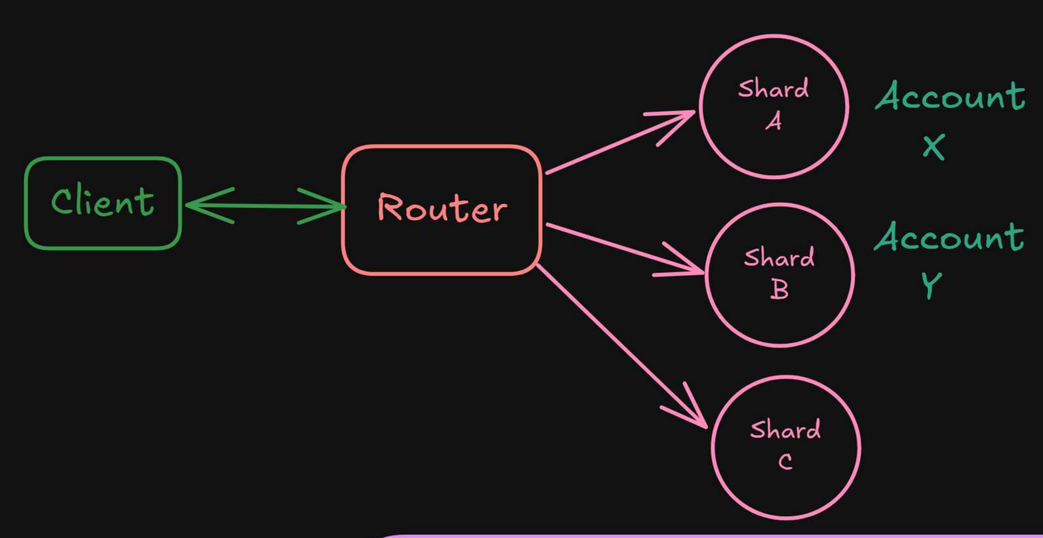
Commit phase — if everyone says yes, coordinator sends COMMIT; else ABORT.

Problems:

-> Slow (extra round trip, locking).
-> Fragile: if coordinator or shard crashes at bad time, you get stuck half-way.
-> Hard to operate in real prod systems at huge scale.

Most large systems avoid 2PC except for niche cases.

So what can we do ?



1. Avoid cross-shard transactions

Put all strongly-related data for a user/entity on the same shard.
-> Example: user profile, user balance, user settings, user's own documents → same shard keyed by user_id.

-> Then most transactions are single-shard and can use normal DB transactions.

2. Eventual consistency is often fine

For secondary views (like counters or feed copies), it's okay if different shards show slightly different values for a few seconds.

Rule of thumb: use strong consistency where money or critical invariants are involved and try to keep those flows "local" to a shard. Use eventual consistency elsewhere.

If a design needs frequent cross-shard transactions, that's often a sign you picked a bad shard key or split your data along the wrong boundary.

3. Use Sagas for multi-shard workflows

Saga = a series of local transactions, each with a compensating action.
Example: transfer wallet balance between two users on different shards:

Shard A: deduct ₹100 from User X.
Shard B: add ₹100 to User Y.
If step 2 fails → run compensating step on shard A: refund ₹100 to User X.

No global lock, but system is eventually consistent.
You might temporarily see "money removed here but not yet added there".

Routing: How does the app know which shard to hit?

1. Routing in the app layer

App receives request with user_id.

Calculates shard_id = hash(user_id) % N (or queries directory).

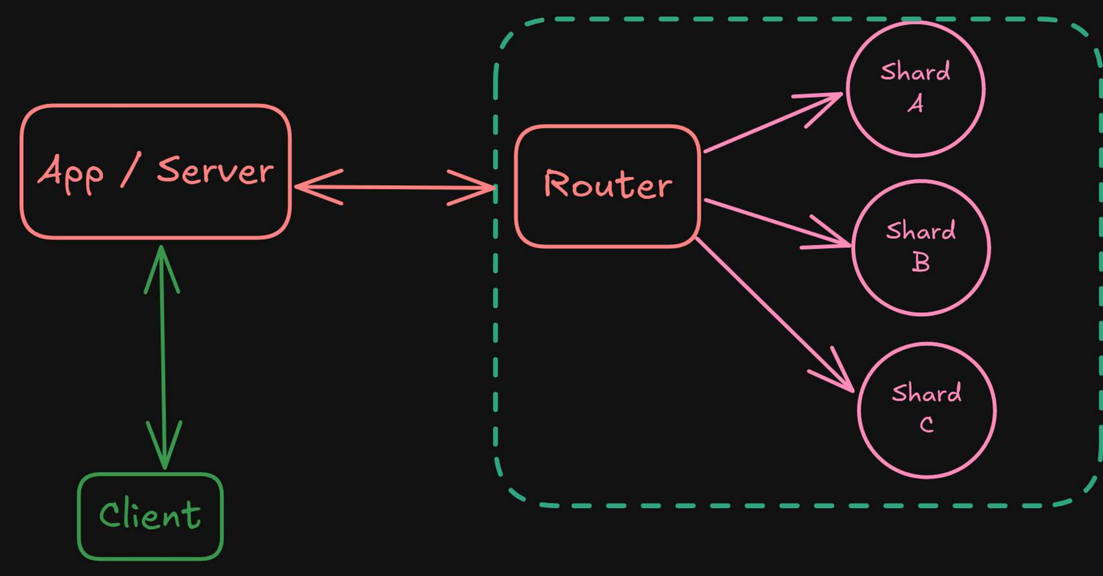
Opens / uses a pooled connection to that shard.

2. Routing via a proxy layer

A middleware (like custom router)
sits between app and DB.

App thinks there is "one big database".

Router parses queries, figures out shard, forwards them.



Operational concerns

Schema changes:
-> Same migration must be applied across all shards.
-> Use migration tooling that runs per-shard.

Backups & restores:
-> Need backup policy per shard.
-> Restoring a single shard must still be possible.

Monitoring & alerting:
-> Per-shard metrics: QPS, errors, latencies, resource usage.
-> Detect hotspots and unbalanced shards early.

Failure handling:
-> Combine sharding with replication: each shard has replicas for HA.
-> On primary failure, promote replica; routing layer must know the new primary.

Sharding in modern databases (real-world mapping)

You almost never implement the low-level plumbing from scratch.

Cassandra-like systems

-> Use a partitioner + token ranges; conceptually similar to consistent hashing.

DynamoDB-like systems

-> Hash partition key → internal partitions; system splits/merges partitions as data and traffic grow.

MongoDB

-> You pick a shard key; Mongo chunks data into ranges of that key (or its hash) and a balancer moves chunks across shards.

Relational:

- >Vitess, Citus sit in front of MySQL/Postgres and handle:
  Query routing
  Cross-shard joins/aggregations
  Live resharding
-> Cloud providers like AWS Aurora also provides distributed SQL
  which has built-in sharding.

Sharding in a system design interview

Don't start with sharding immediately.
First show you understand single DB + vertical scaling + replication.

Introduce sharding when numbers justify it:

Storage limit
"We expect 2 TB today, but 20 TB in 2 years;
that's pushing single-instance comfort zone → sharding."

Write throughput
"Peak 50k–100k writes/sec is too high for one primary;
we should split writes across multiple primaries using sharding."

Read throughput
"Even with read replicas, 100M DAUs may overload a
single primary; shards distribute both read and write pressure."

Shard key

Distribution strategy

Trade-offs / cross-shard

Growth plan

Simple narrative formula:

Estimate data volume & QPS.
Explain why one DB (plus replicas) is not enough.
Propose sharding as the next step.

Also explicitly say: "We won't shard on day 1.
We'll start with one DB, then add sharding
once metrics show we're close to limits."

Migrating an existing system to sharding (extra but useful)

If you already have a single big DB and need to shard it:

-> Decide shard key & strategy
e.g., user_id + hash.

-> Create new shard cluster
Shards + routing layer.

-> Backfill data
Background job reads from old DB, writes to correct shards.

-> Dual-write phase
New writes go to both old DB and shards; reads still from old DB.
Validate data correctness.

-> Switch reads
Switch app to read from shards.
If stable, eventually decommission old monolithic DB (or keep as archival).

Mentioning this shows you understand operational reality, not just theory.

Do we need to Shard Most of the times?

NO

REAL AWS LIMITS (Postgres RDS & Aurora)

Aurora PostgreSQL storage can scale up to 256 TB

This is HUGE.
Most startups will NEVER reach 256 TB of
actual structured transactional data.

RDS PostgreSQL Storage Limit

For non-Aurora RDS PostgreSQL:
Max storage = 64 TiB
Still massive.

Compute Limits (Instance Sizes)

RDS & Aurora let you scale up to:
128 vCPUs + 768 GB RAM

This is a monster machine.

For reference:
A typical Indian startup often runs on 8 vCPUs, 32GB RAM

Moving from 8GB → 768GB RAM means 96× more in-memory cache & index capacity

768GB RAM means your entire working set,
indexes, hot rows stay in memory → insanely fast reads.

IOPS Limits (Disk Speed)

RDS/Aurora supports:
Up to ~256,000 IOPS
and
4,000 MB/sec throughput
(~4GB/s disk read/write)

This is datacenter-grade hardware,
not a normal server.

So What Do These Real Numbers Actually Mean?

Let's evaluate using real numbers.

CASE 1: Storage

If you have:
100 GB
1 TB
2 TB
even 10 TB

You are WAY below the 64TB/256TB limit.
Sharding is NOT needed for storage.

CASE 2: Reads

With:
768GB RAM
4GB/s disk throughput
15 read replicas
ElastiCache (Redis) in front

You can easily handle millions of reads/sec without sharding.
That's why even Instagram & Airbnb didn't shard in early days.

CASE 3: Writes

A single RDS primary can do around:
10k–20k writes/sec (depending on workload)
That's already enough for:

mid-size social network
large e-commerce
multi-tenant SaaS
large fintech backends
streaming platforms
gaming apps

Again: no sharding needed unless you have insane write traffic.

WHEN you actually start thinking about sharding

Even with real RDS limits, you shard only when:

1. Your write throughput exceeds ~20k writes/sec

(Primary becomes bottleneck)

2. Your storage approaches tens of TBs AND keeps growing fast
3. Your indexes get so large that even 768GB RAM can't hold hot data
4. Replication lag becomes unacceptable
5. Maintenance operations (backups, VACUUM, ANALYZE) become too heavy

These limits are VERY high.

Most apps will never reach this stage.

Conclusion: Real AWS Limits Prove We Don't Always Need Sharding

With actual AWS limits like 64–256 TB storage, 768GB RAM, 256k IOPS, and 15 read replicas — most systems never need sharding. You only shard when writes, storage, or latency exceed what even the biggest RDS instance can handle. Until then, sharding adds unnecessary complexity.

Sharding & SPOF

Sharding does NOT automatically eliminate SPOF.

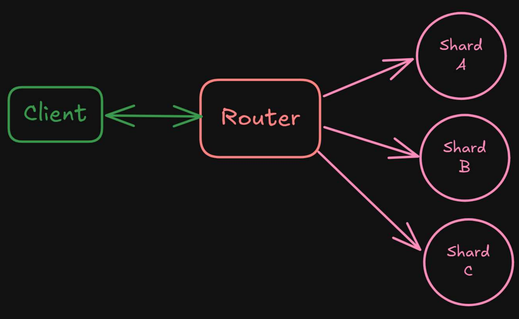
-> If a shard has only 1 DB instance, it IS a SPOF —
but only for that shard's users, not the entire system.

-> When you use RDS for each shard, AWS gives you:
automatic replication
automatic failover
automatic backups
Multi-AZ
read replicas

So each shard becomes a high-availability mini-cluster,
NOT a SPOF.

-> Replication lag causes stale reads, not downtime.

-> If one shard fails entirely:

only that shard's users are impacted
rest of the system stays healthy
this is intentional failure isolation

-> This combination (Sharding + RDS replication)
is exactly how real companies scale relational databases.