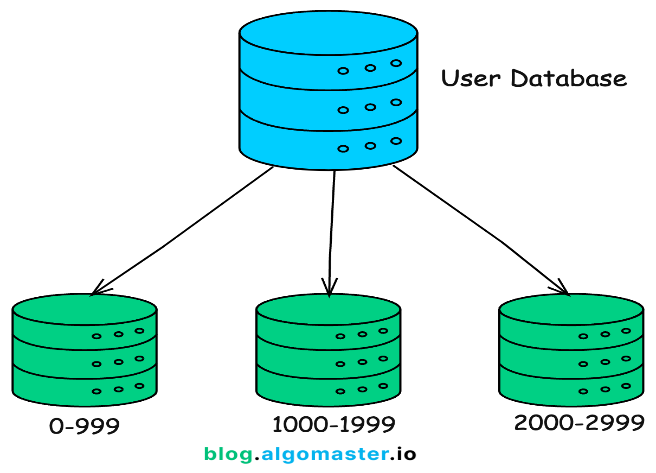
**What is Database Sharding** (ভাগ করা)**?**

Imagine a social media site like Instagram, which has over **1 billion** active users.

Think about what would happen if it tried to keep all the **user profile** data on a single server.

Due to limited scalability of a single machine, it would quickly run out of storage space and slow down leading to performance issues.

But what if we divided the user base into smaller groups based on a key like **userId** and stored each group on separate servers?

[[](https://substackcdn.com/image/fetch/$s_!7xen!,f_auto,q_auto:good,fl_progressive:steep/https%3A%2F%2Fsubstack-post-media.s3.amazonaws.com%2Fpublic%2Fimages%2Fc3ab4e45-65c7-4f81-aeda-19f28dfe3443_1210x1068.png)](https://substackcdn.com/image/fetch/$s_!7xen!,f_auto,q_auto:good,fl_progressive:steep/https%3A%2F%2Fsubstack-post-media.s3.amazonaws.com%2Fpublic%2Fimages%2Fc3ab4e45-65c7-4f81-aeda-19f28dfe3443_1210x1068.png" \t "_blank)

**Example:**

* Group 1: Users with IDs 0-999
* Group 2: Users with IDs 1000-1999
* Group 3: Users with IDs 2000-2999

Now, a single server only deals with subset of data.

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

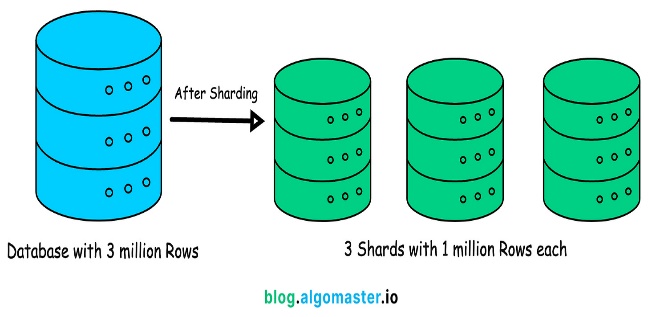
This is the idea behind **Database Sharding**.

In this article, we will explore what database sharding is, it’s benefits, how it works, challenges that come with it and the best practices for implementing it.

**1. What is Database Sharding?**

**Database sharding** is a horizontal scaling technique used to split a large database into smaller, independent pieces called **shards**.

These shards are then distributed across multiple servers or nodes, each responsible for handling a specific subset of the data.

[[](https://substackcdn.com/image/fetch/$s_!17Jc!,f_auto,q_auto:good,fl_progressive:steep/https%3A%2F%2Fsubstack-post-media.s3.amazonaws.com%2Fpublic%2Fimages%2F6e67c3b7-92cd-4f84-854e-e71f45f7efa2_1902x668.png)](https://substackcdn.com/image/fetch/$s_!17Jc!,f_auto,q_auto:good,fl_progressive:steep/https%3A%2F%2Fsubstack-post-media.s3.amazonaws.com%2Fpublic%2Fimages%2F6e67c3b7-92cd-4f84-854e-e71f45f7efa2_1902x668.png" \t "_blank)

Sharding is widely used by large-scale applications and services that handle massive amounts of data, such as:

* **Social Networks:** Instagram uses sharding to manage billions of user profiles and interactions.
* **E-commerce Platforms:** Amazon employs sharding to handle massive product catalogs and customer data.
* **Search Engines:** Google relies on sharding to index and retrieve billions of web pages.
* **Gaming**: Online gaming platforms use sharding to handle millions of players and vast amounts of game data.

**2. Benefits of Sharding**

1. **Improved Performance**: By distributing the data across multiple nodes, sharding can significantly reduce the load on any single server, resulting in faster query execution and improved overall system performance.
2. **Scalability**: Sharding allows databases to grow horizontally. As data volume increases, new shards can be added to spread the load evenly across the cluster.
3. **High Availability**: With data spread across multiple shards, the failure of a single shard doesn't bring down the entire system. Other shards can continue serving requests, ensuring high availability.
4. **Geographical Distribution:** Sharding allows you to strategically place shards closer to your users, reducing latency and improving the user experience.
5. **Reduced Cost**: Instead of scaling vertically by investing in more powerful and expensive hardware, sharding allows for cost-effective scaling by utilizing commodity hardware.

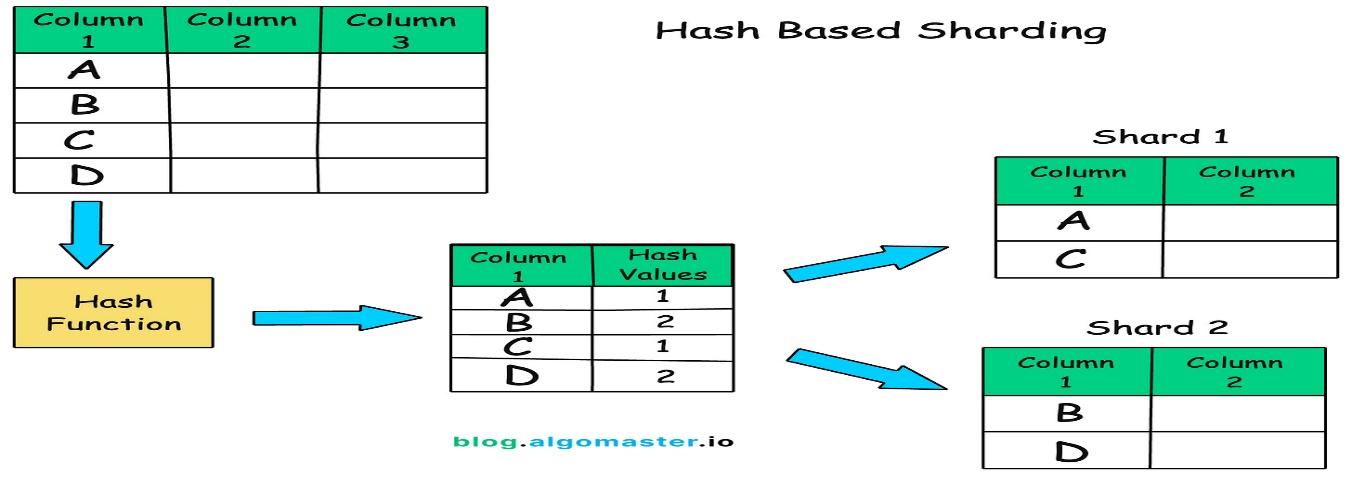
**3. How Does Sharding Work?**

The sharding process involves several key components including:

1. **Sharding Key**: The shard key is a unique identifier used to determine which shard a particular piece of data belongs to. It can be a single column or a combination of columns.
2. **Data Partitioning**: Partitioning involves splitting the data into shards based on the shard key. Each shard represents a portion of the total data set. Common strategies to partition database are **range-based, hash-based, and directory-based sharding.**
3. **Shard Mapping**: Creating a mapping of shard keys to shard locations.
4. **Shard Management:** The shard manager oversees the distribution of data across shards, ensuring data consistency and integrity.
5. **Query Routing**: The query router intercepts incoming queries and directs them to the appropriate shard(s) based on the shard key.

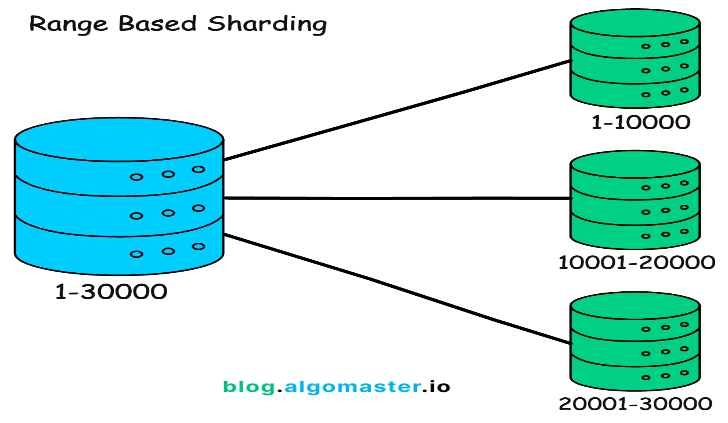
**4. Sharding Strategies**

* **Hash-Based Sharding**: Data is distributed using a hash function, which maps data to a specific shard.
  + **Example:** Hash(user\_id) % 2 determines the shard number for a user, distributing users evenly across 2 shards.

[[](https://substackcdn.com/image/fetch/$s_!Q2iq!,f_auto,q_auto:good,fl_progressive:steep/https%3A%2F%2Fsubstack-post-media.s3.amazonaws.com%2Fpublic%2Fimages%2F29dfc20e-7ac6-4193-9b79-eeb12b970e9b_1886x1404.png)](https://substackcdn.com/image/fetch/$s_!Q2iq!,f_auto,q_auto:good,fl_progressive:steep/https%3A%2F%2Fsubstack-post-media.s3.amazonaws.com%2Fpublic%2Fimages%2F29dfc20e-7ac6-4193-9b79-eeb12b970e9b_1886x1404.png" \t "_blank)

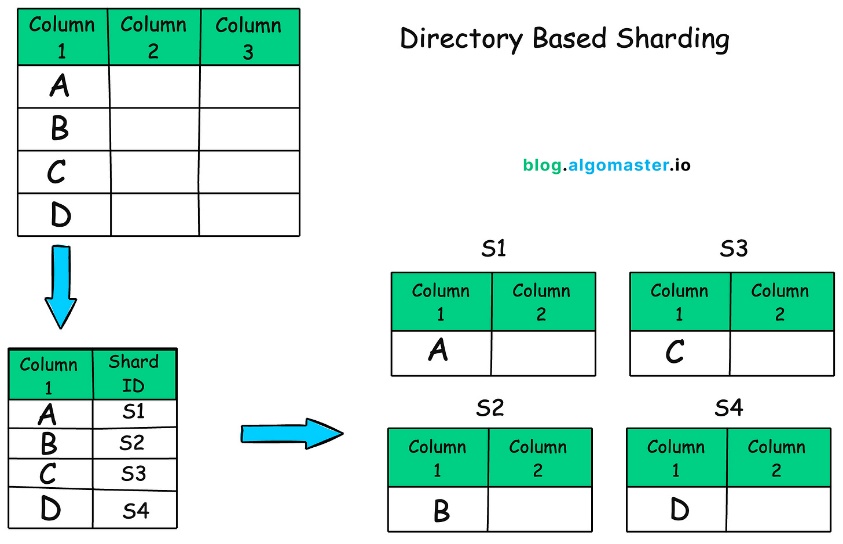
**Range-Based Sharding**: Data is distributed based on a range of values, such as dates or numbers.

* + **Example:** Shard 1 contains records with IDs from 1 to 10000, Shard 2 contains records with IDs from 10001 to 20000, and so on.

[[](https://substackcdn.com/image/fetch/$s_!CUpP!,f_auto,q_auto:good,fl_progressive:steep/https%3A%2F%2Fsubstack-post-media.s3.amazonaws.com%2Fpublic%2Fimages%2F51416dc8-3adb-493d-a7c5-bd1cd08e20ac_1254x1070.png)](https://substackcdn.com/image/fetch/$s_!CUpP!,f_auto,q_auto:good,fl_progressive:steep/https%3A%2F%2Fsubstack-post-media.s3.amazonaws.com%2Fpublic%2Fimages%2F51416dc8-3adb-493d-a7c5-bd1cd08e20ac_1254x1070.png" \t "_blank)

**Geo-Based Sharding**: Data is distributed based on geographic location.

* + **Example:** Shard 1 serves users in North America, Shard 2 serves users in Europe, Shard 3 serves users in Asia.
* **Directory-Based Sharding**: Maintains a lookup table that directly maps specific keys to specific shards.

[[](https://substackcdn.com/image/fetch/$s_!X5I9!,f_auto,q_auto:good,fl_progressive:steep/https%3A%2F%2Fsubstack-post-media.s3.amazonaws.com%2Fpublic%2Fimages%2Fb74afae7-0029-441d-ae39-629c2b27224a_1972x1310.png)](https://substackcdn.com/image/fetch/$s_!X5I9!,f_auto,q_auto:good,fl_progressive:steep/https%3A%2F%2Fsubstack-post-media.s3.amazonaws.com%2Fpublic%2Fimages%2Fb74afae7-0029-441d-ae39-629c2b27224a_1972x1310.png" \t "_blank)

**5. Challenges with Sharding**

1. **Complexity**: Sharding introduces additional complexity, requiring careful planning and management.
2. **Data Consistency**: Maintaining data consistency across shards can be challenging, especially when data needs to be joined or aggregated from multiple shards.
3. **Cross-shard Joins:** Performing joins across multiple shards can be complex and computationally expensive.
4. **Data Rebalancing:** As data volumes grow, shards may become unevenly distributed, requiring rebalancing to maintain optimal performance.

**6. Best Practices for Sharding**

* **Choose the Right Sharding Key**: Select a sharding key that ensures an even distribution of data across shards and aligns with the application's access patterns.
* **Use Consistent Hashing**: Implement a consistent hashing algorithm to minimize data movement when adding or removing shards.
* **Monitor and Rebalance Shards**: Regularly monitor shard performance and data distribution. Rebalance shards as needed to ensure optimal performance and data distribution.
* **Handle Cross-Shard Queries Efficiently**: Optimize queries that require data from multiple shards by using techniques like query federation or data denormalization.
* **Plan for Scalability**: Design your sharding strategy with future growth in mind. Consider how the system will scale as data volume and traffic increase.

**7. Conclusion**

To summarize, database sharding is a powerful technique for scaling databases horizontally and handling large amounts of data, but it does come with additional complexity and requires thoughtful planning and understanding of application’s data access patterns.

Before implementing sharding, think about whether the benefits outweigh the costs or if there is a simpler solution.

Thank you for reading!

**1. What is Database Sharding and why do we need it?**

**Answer:**

* **Definition:** Sharding is a technique to split a large dataset across multiple databases (shards) so each shard handles a portion of the load.
* **Need:**
  + Single DB can’t scale beyond certain IOPS/size.
  + High read/write throughput.
  + Reduce query latency by isolating subsets of data.
* **Trade-off:** Increases complexity (cross-shard queries, rebalancing).

**2. What are the common sharding strategies?**

**Answer:**

* **Hash-based sharding:** Evenly distributes load → prevents hotspots but hard for range queries.
* **Range-based sharding:** Easy range queries but risk of hotspots.
* **Directory/Lookup sharding:** Flexible but adds lookup latency and dependency.
* **Geo/functional sharding:** Based on tenant/region/business logic.

**3. Scenario: You need to shard a user database with 500M users. Which key would you choose?**

**Answer:**

* **Option 1:** user\_id % N (hash-based). Good for load balance.
* **Option 2:** Range by user\_id (e.g., 0–100M → shard 1). Easy for range queries but hotspot risk if most new users join latest range.
* **Option 3:** Geographic shard (e.g., US users vs EU users). Good for compliance + latency.  
  👉 **Final:** Hash-based sharding on user\_id for balance, plus region-based sharding for latency compliance.

**4. What happens if one shard becomes “hot” (too many writes)?**

**Answer:**

* Causes: Poor shard key choice (low-cardinality key, e.g., country).
* Solutions:
  1. **Resharding with better distribution** (consistent hashing).
  2. **Shard splitting** (split large shard into multiple smaller shards).
  3. **Add caching layer** to reduce DB load.

**5. How do you handle cross-shard queries?**

**Answer:**

* **Option 1:** Query all shards (scatter-gather). Expensive.
* **Option 2:** Maintain **global indexes/metadata service**.
* **Option 3:** Use **federated query engines** (e.g., Presto/Trino).  
  👉 Best practice: Avoid cross-shard joins; denormalize data when possible.

**6. How do you reshard without downtime?**

**Answer:**

* **Dual-write approach:** Write to both old and new shard during migration.
* **Backfill:** Copy historical data to new shard in background.
* **Cutover:** Switch read traffic when new shard is up-to-date.
* Use tools like **Vitess**, **Citus**, or custom resharding pipelines.

**7. Scenario: Your SaaS app needs per-tenant data isolation. How do you shard?**

**Answer:**

* Best approach: **Tenant-based sharding** (each tenant/customer gets its own shard).
* Pros: Isolation, easy compliance (GDPR).
* Cons: Load imbalance if one tenant is very large.
* Mitigation: Large tenants may need sub-shards or dedicated cluster.

**8. How do you handle transactions across shards?**

**Answer:**

* **Options:**
  1. Two-Phase Commit (2PC) → guarantees consistency but high latency.
  2. Sagas → eventual consistency using compensating actions.
  3. Application-level orchestration → avoid cross-shard transactions if possible.
* **Best practice:** Keep transactions within a shard whenever possible.

**9. What’s the difference between sharding and partitioning?**

**Answer:**

* **Partitioning:** Within the same DB instance (logical division).
* **Sharding:** Across multiple DB instances/servers.  
  👉 Sharding adds **horizontal scaling** beyond single-machine limits.

**10. Scenario: You use hash-based sharding, but now need range queries (e.g., last 30 days of orders). How do you optimize?**

**Answer:**

* Hash-based prevents efficient range scans.
* Options:
  + Add **secondary index service** for time-based queries.
  + Maintain **hybrid scheme**: shard by hash but partition by date inside each shard.
  + Use **time-series DB** for recent data.

**11. How do you ensure uniqueness of IDs across shards?**

**Answer:**

* **Options:**
  1. Use **UUIDs** (globally unique, but larger).
  2. Use **snowflake IDs** (Twitter’s approach, timestamp + shard ID + sequence).
  3. Assign **ID ranges per shard** (shard 1 → 1–1B, shard 2 → 1B–2B).

**12. How does replication interact with sharding?**

**Answer:**

* Each shard can have **primary + replicas** for HA and read scaling.
* Replication happens *within a shard cluster*, not across shards.
* Load balancing must consider **both sharding + replication topology**.

**13. Scenario: One shard is growing much faster than others (skew). What do you do?**

**Answer:**

* **Diagnosis:** Shard key distribution issue.
* **Fix:**
  + Split that shard into multiple smaller shards.
  + Use **consistent hashing** to redistribute load.
  + Add **virtual shards** (map multiple logical shards to one physical node).

**14. How do you monitor and detect shard imbalance?**

**Answer:**

* Metrics:
  + Shard size distribution.
  + Query latency per shard.
  + Hotspot detection (# of writes/reads per shard).
* Tools:
  + Shard-aware monitoring dashboards.
  + Query tracing with shard annotations.

**15. Scenario: You’re designing sharding for a ride-sharing app. How would you shard the data?**

**Answer:**

* Entities: Users, Rides, Drivers, Locations.
* **Option 1:** Shard rides by ride\_id % N.
* **Option 2:** Shard by city/region to optimize local queries.
* **Option 3:** Users + drivers hashed separately, with metadata service to resolve.  
  👉 **Final Design:**
* **Regional shard** for rides (because queries are mostly local).
* **User/driver profile hashed** for balance.
* **Global service directory** to route requests.

**5 Scenario-Based Sharding Interview Questions & Answers**

**1. Login requests are slow with 2B users in one table. How to shard?**

✅ **Answer:**  
Shard by user\_id using **hash-based sharding** so logins are evenly distributed. Use a **global index or directory service** to route login requests to the right shard.  
👉 Trade-off: cross-shard queries (e.g., all users created last week) become harder.

**2. One country shard is overloaded with 100K writes/sec. What now?**

✅ **Answer:**  
Instead of sharding by country, shard by **user\_id hash** (or user+timestamp). This avoids skew when one country is more active. You can still maintain a **secondary index on country** for queries.

**3. Business needs “get all users created last 7 days” but you sharded by user\_id hash. How to fix slow scans?**

✅ **Answer:**  
Keep writes sharded by user\_id but create a **time-partitioned table** (or “hot table”) for new users. Queries for “last 7 days” go to this small table, avoiding full cross-shard scans.

**4. One tenant has 100x more data than others. Their shard is overloaded.**

✅ **Answer:**  
Split that tenant across multiple sub-shards (a technique called **“tenant sub-sharding”**). Example: shard on (tenant\_id, hash(user\_id)). Big tenants get multiple shards, small ones stay on one.

**5. Range-based sharding by order date causes latest shard to be a write hotspot (Black Friday). Fix?**

✅ **Answer:**  
Switch to **hybrid sharding**: partition by time + hash of order\_id. This spreads hot writes across multiple shards while preserving some time locality.

**6. Ride-sharing app needs region-based queries (e.g., rides in New York). How to shard?**

✅ **Answer:**  
Shard by **geohash / region id**. This makes local queries fast. For global queries, use an **aggregator service** that queries multiple region shards in parallel.

**7. Joins across shards (users + orders) are needed. How to solve?**

✅ **Answer:**  
Options:

* Co-locate related entities (user + orders in same shard).
* Or use **application-level joins** (fetch from both shards, merge in app).
* For frequent queries, maintain a **denormalized read model**.

**8. Global unique order IDs across shards?**

✅ **Answer:**  
Use **Snowflake-style ID generator** (timestamp + shard\_id + sequence). Or use **UUIDv7** for ordered unique IDs. Avoid auto-increment in each shard.

**9. How to reshard without downtime?**

✅ **Answer:**

* Use **dual writes** (to old + new shards).
* Sync historical data in background.
* Switch traffic gradually with a **routing layer** (like Vitess, ProxySQL).  
  👉 Key: ensure **idempotent writes** during migration.

**10. Read replicas return stale data, but users need strong consistency. Fix?**

✅ **Answer:**

* For strong consistency: route reads after writes to the **primary shard** (read-after-write consistency).
* Use **“read-your-own-write” session stickiness**.  
  👉 Trade-off: increased load on primary.

**11. Social network needs friends-of-friends queries, but user\_id sharding causes cross-shard lookups. Fix?**

✅ **Answer:**

* Shard by **graph partitioning** (users in same community likely end up on same shard).
* Or use a **graph database** (Neo4j) for relationships while keeping user profiles sharded by ID.

**12. One celebrity has millions of followers; their shard grows to 3TB. How to handle hotspot?**

✅ **Answer:**

* Split celebrity’s data into multiple sub-shards (celebrity\_id + hash(follower\_id)).
* Cache follower lists heavily in **Redis/CDN** to reduce DB load.

**13. US users fast, Asian users slow (400ms write latency). Fix global writes?**

✅ **Answer:**

* Deploy **multi-region sharding**: users in Asia write to Asia shard, US users write to US shard.
* Use **eventual consistency** for global aggregation.  
  👉 Trade-off: strong consistency across regions is expensive (needs consensus like Spanner).

**14. Game logs 10M events/hour, queries like “last 1,000 events per player” are slow. Fix?**

✅ **Answer:**

* Shard by player\_id.
* Use **time-series DB** (Cassandra, ClickHouse, TimescaleDB).
* Maintain per-player **rolling buffer index** for quick retrieval.

**15. Hotel booking system: availability checks are slow across shards. How to optimize?**

✅ **Answer:**

* Shard by hotel\_id. Availability queries only hit that shard.
* Pre-compute **availability calendars** in cache (Redis, ElasticSearch) for faster lookups.  
  👉 DB is used for truth, cache is used for real-time queries.