System Design Day 2: Database Design

SQL (Structured Query Language) vs NoSQL (Not Only SQL) with Sharding

---

🔷 1. Data Structure

- **SQL (e.g., MySQL):**
  - Structured, relational data organized into **tables** with rows and columns.
  - Relationships are enforced using **foreign keys**.
- **NoSQL (e.g., DynamoDB):**
  - Flexible data models like **key-value**, **document**, or **wide-column**.
  - Best for loosely structured or rapidly changing data.

🔷 2. Schema

- **SQL:**
  - Enforces a **fixed schema**; changes need explicit migration.
  - Promotes **data integrity and validation**.
- **NoSQL:**
  - **Schema-less** or flexible schema — fields can vary across records.
  - Supports rapid iteration and deployment.

🔷 3. Scalability

- **SQL:**
  - Scales **vertically** (adding more CPU/RAM to a single server).
  - Challenging to scale horizontally without complex partitioning or sharding.
- **NoSQL:**

- Designed for **horizontal scaling** — data is spread across many nodes automatically.

🔷 4. Consistency

- **SQL:**
  - Ensures strong consistency with **ACID** properties:

    (Atomicity, Consistency, Isolation, Durability)
- **NoSQL:**
  - Uses **BASE** properties:

    (Basically Available, Soft state, Eventual consistency)
  - Consistency can be relaxed to improve availability and performance.

🔷 5. Query Capabilities

- **SQL:**
  - Rich **query language (SQL)**: supports joins, aggregates, filtering, subqueries.
  - Ideal for complex reporting and data relationships.
- **NoSQL:**
  - Limited to **simple, key-based** access.
  - Optimized for performance over flexibility.

🔷 6. Use Cases

- **SQL:**
  - **Banking systems**, ERP, E-commerce order management — where consistency and relations matter.
- **NoSQL:**
  - **User activity tracking**, IoT telemetry, session stores — where speed and scale are critical.

🔷 7. Integration with Java

- **SQL:**
  - Typically used with **Spring Data JPA (Java Persistence API)**.
  - Enables ORM (**Object-Relational Mapping**) and declarative transaction handling.
- **NoSQL:**
  - Integrated using **Spring Data DynamoDB** or native **AWS SDK**.
  - Works well with **microservices** for loosely coupled, independent modules.

🔷 8. Sharding

📘 **Sharding Definition:**

> Sharding is a database partitioning technique where large datasets are split into smaller, more manageable pieces called "shards", based on a shard key (e.g., user ID, region). Each shard is stored on a different server or node to distribute load and improve scalability.

- **SQL:**
  - Manual sharding using application logic or middleware.
  - Adds complexity in **query routing and joins across shards**.
- **NoSQL:**
  - **Native support** for automatic sharding.
  - Example: DynamoDB uses **partition key + sort key** for transparent and automatic data distribution.

🔷 9. Performance at Scale

- **SQL:**
  - Performance can degrade under **high concurrent load** or large dataset size.
  - Needs caching layers or read replicas.

- **NoSQL:**
  - Designed for **low-latency** and **high-throughput** workloads.
  - Optimized for write-heavy applications.

✅ Real-Life Application Examples

📌 SQL Example: E-Commerce Order Management System (MySQL)

- **Why SQL?**
  - Requires **strong consistency** and **complex relationships** (orders → users → payments).
  - Relational queries needed for inventory, reporting, and analytics.
- **Tech Stack:**
  - MySQL + Spring Boot + Spring Data JPA
  - Scaled using **Read Replicas**, **Database Connection Pooling**, and **Caching** with Redis.

🔁 Scalable Design Patterns Used in SQL Systems

- **Read Replicas:** Distribute read load from the primary DB.
- **Database Connection Pooling:** Reduces overhead from opening/closing connections.
- **CQRS (Command Query Responsibility Segregation):** Split read and write models for performance.
- **Caching Layer (Redis, Ehcache):** Reduce DB load on frequently accessed data.

📌 NoSQL Example: Social Media User Activity Tracking (DynamoDB)

- **Why NoSQL?**
  - Handles massive, fast writes (likes, posts, follows).
  - Schema-less design supports evolving data.
  - Scales horizontally for millions of users.

- **Tech Stack:**
  - DynamoDB + AWS Lambda + Spring Boot (for APIs)
  - Uses **partition key** (user ID) + **sort key** (timestamp).

🔁 Scalable Design Patterns Used in NoSQL Systems

- **Auto-Sharding:** Managed by DynamoDB via partition keys.
- **Event Sourcing:** Store each user action as an event for audit or replay.
- **Time-Series Data Modeling:** Store data with timestamps for range queries.
- **Microservices:** Decompose features by bounded context; each owns its own NoSQL data store.

How Sharding Affects Query Performance

🔷 1. Positive Impact on Query Performance (When Done Right)

Benefit	Explanation
Parallel Query Execution	Queries can be executed in **parallel across shards**, reducing response time.
Reduced Data Size per Node	Each shard holds **less data**, so **index scans and lookups are faster**.
Less Contention	By distributing load, **read/write contention is minimized** on any single node.
Improved Scalability	As the dataset grows, you can **add more shards**, keeping query performance stable.

🔷 2. Negative Impact on Query Performance (When Done Poorly)

Risk	Impact
Cross-Shard Queries	If the query doesn't include the **shard key**, the system has to **query all shards**, increasing latency.
Hotspotting	Uneven data distribution (e.g., all writes go to one shard) leads to **overloaded shards**, slowing performance.

Risk	Impact
Complex Joins/Aggregations	SQL-style **joins and aggregates across shards** are difficult or slow (often not supported in NoSQL).
Rebalancing Overhead	Adding/removing shards can require **data migration**, temporarily affecting performance.

📘 Example:

- Suppose you're sharding a **user activity log** by `user_id`.
- Query like `SELECT * FROM activity WHERE user_id = 'abc123'` will hit **only one shard** → **fast**.
- But a query like `SELECT * FROM activity WHERE action = 'login'` has no shard key → needs **scatter-gather across all shards** → **slow**.

🔁 Best Practices to Optimize Query Performance in Sharded Systems

1. **Design with Shard Key in Mind**

   Always **include shard key in queries** wherever possible.

2. **Avoid Cross-Shard Joins**

   Denormalize data if needed; joins across shards are expensive.

3. **Uniform Data Distribution**

   Choose a shard key that results in **balanced load and storage**.

4. **Use Routing Layers**

   Implement smart query routers or service layers that know where to send a query.

5. **Monitor and Re-shard Carefully**

   Periodically evaluate shard balance; re-shard during low traffic windows.