Workshop No. 3 Concurrency, Parallelism, and Distributed Databases

Cuellar B. Paola A.
Ebratt S. Johan D.
Universidad Distrital Francisco José de Caldas

Semester 2025-I

1 Concurrency Analysis

Scenarios of Concurrent Access

- **Lost Updates during Posting:** When multiple users post at the same time, the application may miss updates or display inconsistent timelines because concurrent operations are not properly synchronized.
  Solution: Use pessimistic locking or versioning to prevent overwrites and ensure timeline consistency.

- **Race Conditions on Reactions:** When many users react to a post at the same time, the application may display an incorrect reaction count because a race condition occurs between the processes updating the counter.
  Solution: Apply atomic operations, increment locks, or optimistic concurrency control with conflict detection.

- **Concurrent Profile Edits:** When a user edits his or her profile while another process is also modifying it, the application may overwrite information or lose updates because data locks or versions are not handled correctly.
  Solution: Implement version control, or use SELECT FOR UPDATE.

- **Insert Contention on Comments:** When many users comment on the same publication at the same time, the application may experience locks or congestion in the insertions because there is contention in the concurrent access to the database.
  Solution: Use row-level locks and batch inserts, or scale horizontally with partitioned tables.

- **Dirty Reads during Moderation:** When a user reports a post while it is being moderated, the application may show dirty reads because the unconfirmed information is accessed before the end of the moderation transaction.

Solution: Set the isolation level to Read Committed or higher Repeatable Read, Serializable.

- **Phantom Reads on Concurrent Search:** When search filters are applied while inserting new data, the application may show incomplete or inconsistent results because the snapshot of the data is not controlled under an appropriate level of isolation.
  Solution: Use the Serializable isolation level or predicate locking to avoid phantom reads.

2 Distributed Database Design

To ensure scalability, high availability and low latency in various geographical regions, the system adopts a hybrid distributed architecture combining SQL and NoSQL technologies.

Architecture Overview

A hybrid approach is adopted using PostgreSQL for structured data and MongoDB/Redis for flexible, high-performance access.

Distributed Model

UDshare takes a hybrid approach using PostgreSQL for structured and relational data, and MongoDB/Redis for flexible, high-performance access to semi-structured data such as trending hashtags, user activity and timelines. The architecture will be distributed in three major geographic regions:

- **Region A (Colombia):** Main region for Latin America.
- **Region B (USA):** Central hub for data routing and replication.
- **Region C (Europe):**Ensures data localization compliance and low latency access for European users.

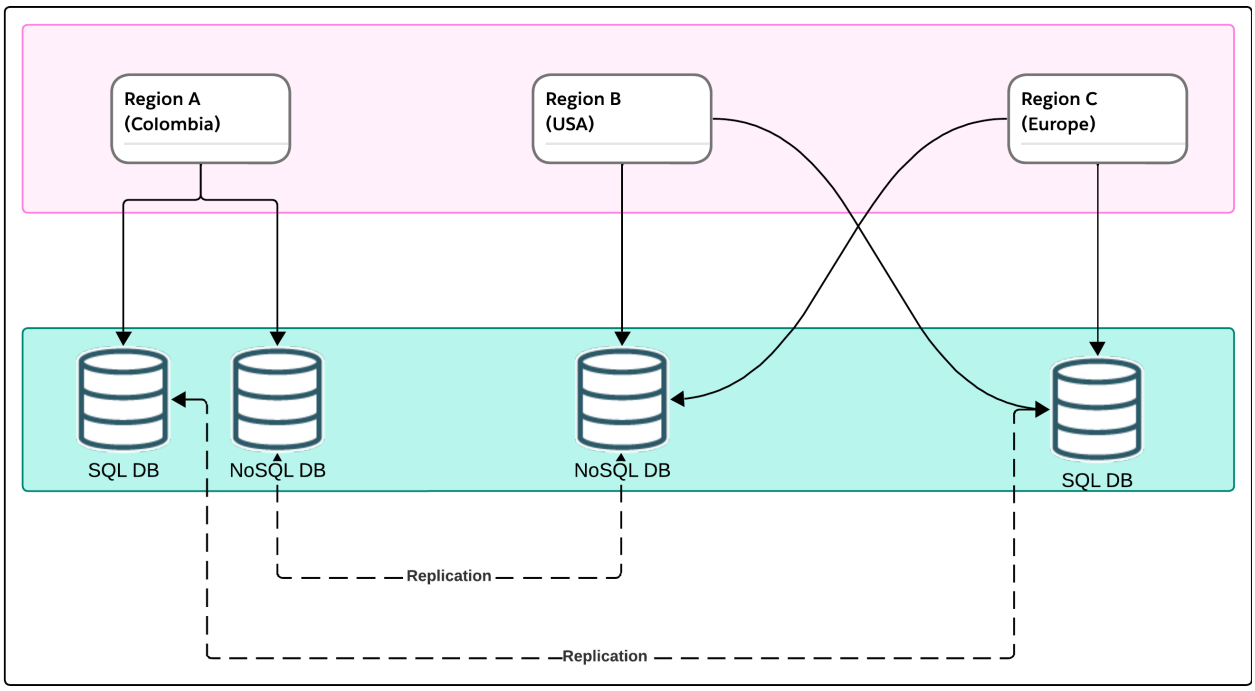

Figure 1: Distributed Diagram DB

2.1 Mechanisms

- **PostgreSQL Sharding by Region:** Each region stores structured user-related data in PostgreSQL. Region sharding reduces write contention and localizes data access.
- **MongoDB Replication Sets:** MongoDB is used for semi-structured documents (entries, comments). Replica sets guarantee high availability and support read scaling on a per-region basis.
- **Redis Hash Slots:** Redis manages high-speed in-memory data, such as trending hashtags and session tokens, distributed by partitioning hash slots.
- **Replication between regions:** Bi-directional replication synchronizes critical data sets across SQL and NoSQL nodes to ensure end-to-end consistency and disaster recovery readiness.

This hybrid architecture, segmented by region, is justified by the key needs of the project: the handling of large volumes of data such as millions of posts, reactions and interactions in real time; the global reach with users located in Latin America, North America and Europe; the delivery of content in real time, where Redis allows to quickly update the most relevant content; and the integrity of the data, guaranteed by PostgreSQL through strong consistency mechanisms for the management of user accounts and relationships.

3 Performance Improvement Strategies

To improve system performance and minimize latency in distributed regions, the following strategies will be applied:

Strategy 1: Horizontal Partitioning (Sharding)

Messages and other large-scale content are horizontally partitioned (sharding) by region or by user ID range. This means that data is spread across multiple database instances, which reduces the load on a single node.

Advantages:

- Increases performance by allowing parallel writes and reads on independent partitions.
- Reduces contention and bottlenecks in high traffic regions.

Challenges / Trade-offs:

- Queries between shards become more complex and may require aggregation mechanisms.
- Difficult to maintain referential integrity between fragments.

Strategy 2: Replication for High Availability

A combination of PostgreSQL master-slave replication and MongoDB replica sets ensures that data is continuously synchronized and redundantly stored.

Advantages:

- Enables failover in case of node failure.
- Distributes read queries to replicas, improving read performance.

Challenges / Trade-offs:

- Replication lag may cause slightly stale reads in asynchronous replicas.
- Increases storage requirements and operational complexity.

Strategy 3: Parallel Query Execution

Enable parallel sequential scans and joins in PostgreSQL for analytical queries, and use Redis Streams to process real-time events (e.g., trending hashtags, live notifications) concurrently.

Advantages:

- Maximizes CPU utilization and reduces response time for large queries.
- Real-time analytics and dashboards are supported with minimal delay.

Challenges / Trade-offs:

- Requires tuning PostgreSQL parameters and understanding workload patterns.
- Real-time stream processing may require back-pressure management and scaling policies.

4 Improvements to Workshop 2

References

1. MongoDB Inc., "Database Sharding: Concepts and Examples". MongoDB Developer Resources, 2020. [Online]. Available: https://www.mongodb.com/resources/products/capabilities/database-sharding-explained#:~:text=Sharding%20is%20a%20method%20for,See%20more%20on%20the%20basics

2. R. Ragul and A. P. Rajan, "Efficient Horizontal Scaling of Databases using Data Sharding Technique". International Journal of Innovative Technology and Exploring Engineering (IJITEE), vol. 9, no. 5, pp. 590–593, 2020. [Online]. Available: https://www.ijitee.org/wp-content/uploads/papers/v9i5/E2418039520.pdf#:~:text=1,unique%20from%20other%20partition%20tables

3. Redis Project, "Redis Cluster Specification". Redis Documentation, sec. Hash Slot, 2019. [Online]. Available: https://redis.io/docs/latest/operate/oss_and_stack/reference/cluster-spec/#:~:text=Each%20master%20node%20in%20a,however%20the%20serving

4. PostgreSQL Global Development Group, "Parallel Query in PostgreSQL – Documentation v17". PostgreSQL Manual, ch. 15, 2023. [Online]. Available: https://www.postgresql.org/docs/current/parallel-query.html#:~:text=PostgreSQL%20can%20devise%20query%20plans,This%20chapter%20explains%20some