Workshop 3

Juan Sebastian Colorado Caro, Walter Alejandro Suarez Fonseca

School of Engineering

Computer engineering

Universidad Distrital Francisco José de Caldas

Bogotá, Colombia

Email: jscoloradoc@udistrital.edu.co, wasuarezf@udistrital.edu.co

Click here to Git Repository

Abstract

This document presents the design and implementation of a hybrid data architecture for a scalable short-form video platform inspired by TikTok. The system integrates relational and non-relational databases to address different access patterns: transactional integrity, monetization tracking, and user management are handled via a normalized SQL schema, while high-volume, real-time interactions such as video views, likes, and comments are managed in NoSQL collections. The document outlines the Entity-Relationship model, NoSQL collections, and a set of representative queries aligned with business goals and user stories. Furthermore, it analyzes concurrency challenges in multi-user environments and proposes techniques such as optimistic locking, transaction isolation levels, and atomic operations. Finally, performance optimization strategies are proposed, including sharding, replication, and parallel query execution, to ensure scalability and responsiveness. The resulting system architecture balances consistency, availability, and scalability, providing a strong foundation for future implementation.

Index Terms

Short-form video platform, hybrid database architecture, relational modeling, NoSQL, concurrency control, performance optimization, ER modeling, scalability, distributed systems, transactional integrity.

I. Introduction

Short-form video platforms have surged in popularity due to their dynamic content delivery and high user engagement. Designing a data architecture for such systems involves complex requirements, including supporting millions of concurrent interactions, ensuring transactional integrity, providing monetization mechanisms for creators, and enabling precise ad targeting for advertisers.

This document addresses these challenges by presenting the database design and analysis for a platform inspired by TikTok. The system is built using a hybrid data architecture: a relational model for core business transactions, user profiles, subscriptions, and campaigns, and a NoSQL model to efficiently handle semi-structured, high-volume data such as views, comments, and reactions.

The report is structured into multiple sections:

- The ER diagram is developed using a 10-step methodology and refined from its initial version.
- User stories are expanded to include priorities, effort estimates, and clear acceptance criteria.
- Sample queries (SQL and NoSQL) are provided for each key information requirement.
- A concurrency analysis identifies race conditions and deadlocks and proposes mitigation strategies.
- Performance improvement strategies are discussed, emphasizing horizontal scalability and parallel processing.

By combining rigorous modeling with practical implementation considerations, this document demonstrates how these kind of modern platforms can be designed to scale efficiently while maintaining data integrity and responsiveness.

II. Concurrency Analysis

A. Scenarios with concurrent access

In our short-form video platform, concurrent access to shared data is inevitable, particularly in these high-risk scenarios:

1) **Liking and Viewing Videos**
   Multiple users may like or view the same video simultaneously. If not controlled, this may lead to lost updates in like/view counters.

2) **Comment Posting**

   Many users can write comments on the same video at the same time. If atomicity is not guaranteed, some comments may be lost or written out of order.
3) **Subscription Payments**

   A user may trigger multiple payment operations due to accidental multiple clicks or retries. This may lead to duplicate transactions unless protected by transaction isolation.
4) **Campaign Management by Advertisers**

   Advertisers may attempt to edit, pause, or resume campaigns concurrently. Without proper locking, updates could overwrite previous changes or leave campaigns in inconsistent states.
5) **Content Moderation by Multiple Admins**

   Two admins may attempt to resolve or reject the same content report. This can lead to write-write conflicts or phantom reads without control.

B. Potential Concurrency Problems

Scenario	**Risk Type**	**Possible Outcome**
Likes/views updates	Lost update	Inaccurate counters
Simultaneous commenting	Write-write conflict	Overwritten or lost comments
Duplicate payment submissions	Phantom writes	Overcharged users / inconsistent balance
Concurrent campaign edits	Dirty reads / race	Outdated values being saved
Multiple admins resolving reports	Non-repeatable reads	Inconsistent moderation records

TABLE I

CONCURRENCY ISSUES IN KEY APPLICATION SCENARIOS

C. Proposed Concurrency Control Solutions

Scenario	**Technique**	**Reason**
Likes / views	Optimistic concurrency (NoSQL) + atomic ops	Use MongoDB `$inc` and ensure idempotent writes
Commenting	Document-level locks (Mongo) or append-only	Avoid overwrite by inserting into arrays
Payments	SQL transactions + SERIALIZABLE isolation	Avoid duplicate charges; ensure uniqueness constraints
Campaign editing	Pessimistic locking or version control	Use `last_updated_at` timestamp or a version field
Report resolution	SQL `SELECT ... FOR UPDATE` or advisory locks	Lock specific reports during admin review

TABLE II

CONCURRENCY CONTROL STRATEGIES BY SCENARIO

III. Parallel and Distributed Database Design

To support the scalability, availability, and low-latency requirements of a short-form video platform, we propose a high-level design that integrates parallel and distributed database technologies into our hybrid data architecture.

A. *Proposed Architecture*

- **Hybrid SQL + NoSQL:** We will use PostgreSQL for structured, transactional data (users, subscriptions, transactions) to ensure data integrity, while Firebase Realtime Database will handle high-volume, semi-structured data (views, likes, comments) to support high-concurrency writes and reads in real time.
- **Horizontal Partitioning (Sharding):** Collections such as `Views`, `Comments`, and `Reactions` will be logically partitioned using the `video_id` or `user_id` as the shard key. This reduces contention and enables the system to distribute workload across multiple nodes.
- **Replication for High Availability:** PostgreSQL will be configured with read replicas to separate read-heavy analytics and feed generation workloads from transactional updates, improving fault tolerance and supporting horizontal scalability.
- **Event Streaming Layer:** Redis Streams will serve as a lightweight, distributed event queue for capturing real-time interactions (likes, views, uploads), decoupling ingestion from processing while enabling parallel consumers for analytics and notifications.
- **Data Lake + Data Warehouse:** Raw logs and streams will be stored in a Data Lake (Google Cloud Storage), while processed data for business intelligence and reporting will be managed in Google BigQuery, enabling distributed analytical processing and parallel query execution.

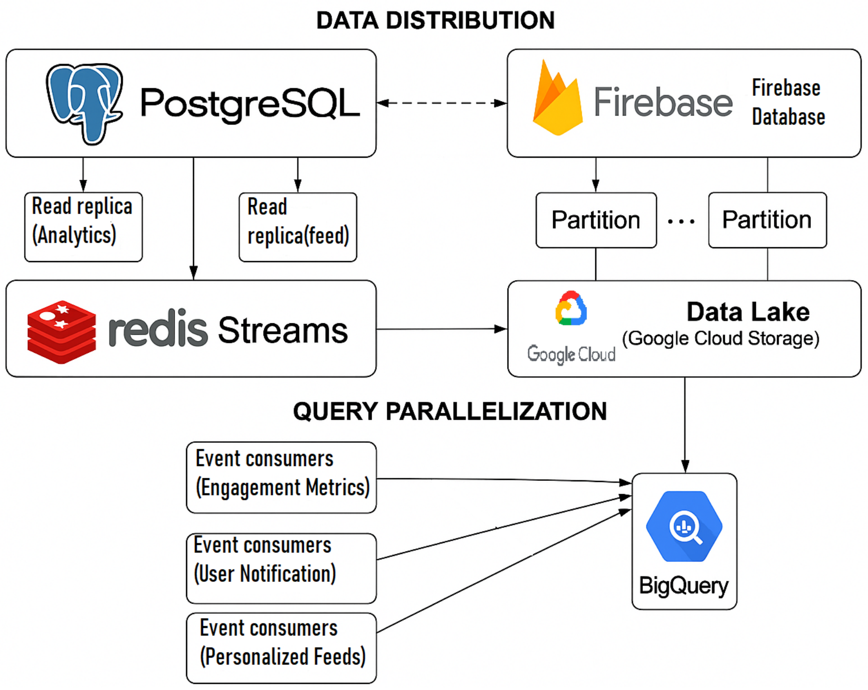

Fig. 1. High-level Distribution and Parallelization Architecture Diagram

The following diagram complements the *Proposed Architecture* by illustrating how data flows through the system, how components interact, and how distribution and parallelization are achieved to support high concurrency and scalability in the short-form video platform.

At the top, the diagram shows the **data distribution layer**:

- The **PostgreSQL** primary node handles transactional writes, while two specialized *Read Replicas* distribute read-heavy workloads: one dedicated to *feed generation queries* and another to *analytics and reporting*, ensuring read scalability and fault tolerance.
- The **Firebase Realtime Database** is logically partitioned into separate collections (`Views`, `Comments`, `Reactions`), each optimized for high-concurrency writes and reads, reducing contention and supporting real-time user interactions.
- A **Data Lake (Google Cloud Storage)** ingests raw event streams for scalable storage, enabling batch processing workflows for downstream analytics.
- **BigQuery** serves as the analytical layer, executing distributed queries over processed data to generate insights for business intelligence, trending analysis, and campaign monitoring.

At the bottom, the diagram highlights the **parallelization strategies**:

- **Redis Streams** acts as the event streaming backbone, capturing high-volume interactions (likes, views, uploads) in real-time and distributing them to consumers for processing.
- Three specialized **Event Consumers** process streams in parallel:
  - *Metrics Consumer* updates engagement metrics in analytical stores, supporting dashboards and trend calculations.
  - *Notifications Consumer* sends real-time notifications to creators and users upon new interactions.
  - *Feed Update Consumer* updates user-specific `FeedCache` entries to maintain up-to-date recommendations with minimal latency.

This diagram operationalizes the proposed architecture, illustrating how the system combines **distribution for availability and scalability** with **parallel processing for low-latency user experience and analytics**, aligning with the project's requirements for a high-concurrency, real-time, and scalable platform.

B. Parallelism Justification

The proposed distributed design incorporates parallelism at multiple levels to improve performance and scalability:

- **Real-time Event Processing:** Using Redis Streams allows multiple consumers to process likes, views, and uploads in parallel, enabling simultaneous updates to engagement metrics, notifications, and dashboards without blocking ingestion.
- **Parallel Analytical Queries:** Google BigQuery enables the execution of distributed analytical queries in parallel over large datasets, allowing the system to generate revenue reports, content rankings, and trend analysis efficiently.
- **Read-Write Load Separation:** By using read replicas in PostgreSQL, the system separates read-heavy operations (feed generation, analytics) from transactional writes, supporting concurrent user interactions without performance degradation.
- **Parallel Batch Processing:** ETL pipelines can process large log datasets in parallel across multiple nodes, accelerating the generation of aggregated metrics for business intelligence dashboards.

Through these strategies, the system achieves effective parallelism, ensuring it can handle high user concurrency, maintain responsiveness, and deliver analytical insights in near real time while remaining aligned with the system's consistency and availability requirements.

IV. Performance Improvement Strategies (via Distribution)

To improve the system's responsiveness and scalability within the practical limits of this project, we propose three distribution-based strategies that can be conceptually designed and partially implemented using lightweight techniques. These strategies focus on distributing access and data logically to improve user experience and performance under increasing load.

A. Logical Partitioning by User or Video ID

- **What it is:** Dividing large NoSQL collections (e.g., `Views`, `Comments`, `Reactions`) based on access keys like `video_id` or `user_id`.
- **Implementation scope:** Collections can be modeled to simulate shard-like separation, without needing real cluster deployment.
- **Benefit:** Enables localized queries and avoids overloading a single collection with unrelated documents.

B. Client-Side Feed Caching

- **What it is:** Storing precomputed video feed data per user (e.g., trending or followed content) in a `FeedCache` collection.
- **Implementation scope:** Feeds are updated periodically (e.g., every 30 minutes) rather than computed on-demand.
- **Benefit:** Reduces query load and improves perceived latency for users.

C. Separation of Read and Write Workloads

- **What it is:** Structuring endpoints and collections such that read-heavy operations (e.g., analytics, trends) are separated from write-heavy ones (e.g., interactions, transactions).
- **Implementation scope:** Modeled at the API and database layer by isolating query paths and collections.
- **Benefit:** Prevents performance bottlenecks and supports future replication.

Trade-offs and System-Level Implications

Strategy	**Benefit**	**Trade-off**
Logical partitioning	Load distribution	Complex queries across partitions are harder to support
Feed caching	Faster feed delivery	Feeds may be slightly outdated between refresh intervals
Read-write workload separation	More efficient query handling	Requires maintaining data consistency across collections

TABLE III

TRADE-OFFS FOR LIGHTWEIGHT DISTRIBUTION STRATEGIES

These design choices reflect a deliberate trade-off between consistency,latency and implementation simplicity. For example:

– Cached feeds sacrifice freshness to gain speed.
– Partitioned data reduces contention but complicates analytics.
– Separate read/write paths simplify scaling but increase data duplication risks.

V. CONCLUSIONS

The design and analysis presented in this Workshop 3 demonstrate that a hybrid architecture combining PostgreSQL for structured transactional data with Firebase Realtime Database for high-volume, semi-structured data enables the system to meet the concurrency, scalability, and responsiveness requirements of a modern short-form video platform.

The proposed distribution strategy, including horizontal partitioning (sharding) for NoSQL collections and read replicas in PostgreSQL, supports high-concurrency read and write operations while maintaining data integrity and availability. The integration of Redis Streams as an event streaming layer effectively decouples ingestion from processing, allowing parallel consumers to handle metrics updates, notifications, and feed caching in real time.

Parallelism is further leveraged through BigQuery, which enables distributed analytical queries for business intelligence and reporting without impacting the system's operational performance. The proposed design aligns with the project's requirements of fast query execution under high loads, constant data ingestion, multi-location access, and real-time business analytics.

While the design provides a strong foundation for scalable and consistent operations, further stages of the project will focus on implementing stress testing, tuning partition strategies, and optimizing parallel pipelines to validate and refine the system under realistic workloads.

These future steps will help transform this architectural plan into a robust, production-ready system capable of supporting millions of concurrent users and providing a seamless user experience.

References

[1] PostgreSQL Documentation, “PostgreSQL 17 Documentation,” 2025. [Online]. Available: https://www.postgresql.org/docs/current/. [

[2] MongoDB, “What is NoSQL?,” 2025. [Online]. Available: https://www.mongodb.com/nosql-explained.

[3] M. Zaharia *et al.*, “Apache Spark: A Unified Engine for Big Data Processing,” *Communications of the ACM*, vol. 59, no. 11, pp. 56–65, 2016.

[4] Quix.io, “Kafka vs Kinesis: A Comparison of Streaming Data Platforms,” 2025. [Online]. Available: https://quix.io/blog/kafka-kinesis-comparison.

[5] Google Cloud, “BigQuery Documentation,” 2025. [Online]. Available: https://cloud.google.com/bigquery/docs.

[6] Redis, “Redis Streams,” 2025. [Online]. Available: https://redis.io/docs/data-types/streams/.