WORKSHOP III

SOFIA LOZANO MARTINEZ
20211020088

JOSÉ JESÚS CÉSPEDES RIVERA
20211020118

CARLOS ANDRÉS SIERRA VIRGUEZ

UNIVERSIDAD DISTRITAL
FRANCISCO JOSÉ DE CALDAS

FRANCISCO JOSÉ DE CALDAS DISTRICT UNIVERSITY

FACULTY OF ENGINEERING
CURRICULAR PROJECT: SYSTEMS ENGINEERING
DATABASE II

BOGOTÁ, D.C., JULY 04
2025

INTRODUCTION

This third workshop in the Database II course represents an important step forward in the development of the Kine system, a digital financial platform inspired by the architecture and functionalities of Nequi. The focus of this installment is on analyzing critical concurrency issues, integrating a distributed database solution, and formulating strategies to improve the overall performance of the system.

First, it provides a detailed analysis of the scenarios in which concurrent access to data may occur within Kine, identifying potential risks such as race conditions and deadlocks, and proposing specific solutions such as the use of multi-version concurrency control (MVCC), isolation level locks, and optimistic control mechanisms.

Subsequently, a distributed architecture design is proposed that responds to both the operational needs of high availability and the regulatory requirements of the Colombian financial sector, especially in relation to the retention and consultation of transaction histories. This architecture includes an active hot data system and a cold storage component in the cloud for historical archiving, coordinated through automated extraction and migration processes.

Finally, strategies for improving system performance are presented (section developed by a teammate), exploring techniques such as data fragmentation, geographic replication, and parallel query execution to reduce latency and increase scalability. These proposals are aimed at ensuring that the system can sustain a growing load of users and transactions in real time, without compromising the integrity or availability of information.

Overall, this workshop consolidates the evolution of the project through technical decisions that strengthen its robustness and responsiveness to the operational and regulatory demands of a modern financial platform.

CONCURRENCY ANALYSIS

Concurrency Scenario	Problem Description	Proposed Solution	Technical Explanation
Simultaneous transfers and payments from the same account	Two operations (a transfer and a payment) are executed almost at the same time, which could lead to a negative balance if both read the same initial balance.	Pessimistic locking on the account record	By locking the account row while the balance is being updated, it ensures that only one operation can modify it at a time. The second operation waits for the first to finish.
Money transfers from multiple users to the same account	Multiple users might attempt to transfer money simultaneously to the same recipient, as could be the case for a business. This is a potential case of dirty reads.	MVCC (Multiversion Concurrency Control)	Implemented natively in PostgreSQL. Allows reads not to block writes and vice versa. Useful for balance queries, transaction histories, without interfering with write operations.
Processing of automatic payments or scheduled debits	These processes can run in the background while the user performs other operations, simultaneously accessing the same account record.	Transaction isolation levels	It is recommended to use at least Read Committed or Repeatable Read for most operations. For critical operations like transfers or debits: Serializable, although with higher cost.
Simultaneous use of a Pocket (Savings Pocket)	The user has an automatic daily saving of $5,000 into a Pocket, and simultaneously performs a manual withdrawal of $10,000. If both operations execute at the same time, they may cause inconsistencies in the balance or duplicate records.	Optimistic concurrency control (versioning)	A version or timestamp field is used in the Pocket. Before saving changes, the system checks whether another transaction has modified the same record. If differences are detected, it rejects the operation and forces a retry with updated data, avoiding write collisions.

Duplicate transfer due to retries	A user sends a transfer. After a network error or slowness, they press “Send” again. The backend receives two identical requests. Without control, both could be processed.	Use of UUIDs and idempotency	A unique transaction_id (UUID) is generated on the frontend and sent with the operation. Before processing, the backend checks if this transaction_id already exists in the database. If it exists, the transaction is not executed again, and the original response is returned. This ensures that even if the same request is sent multiple times, the effect on the balance occurs only once.

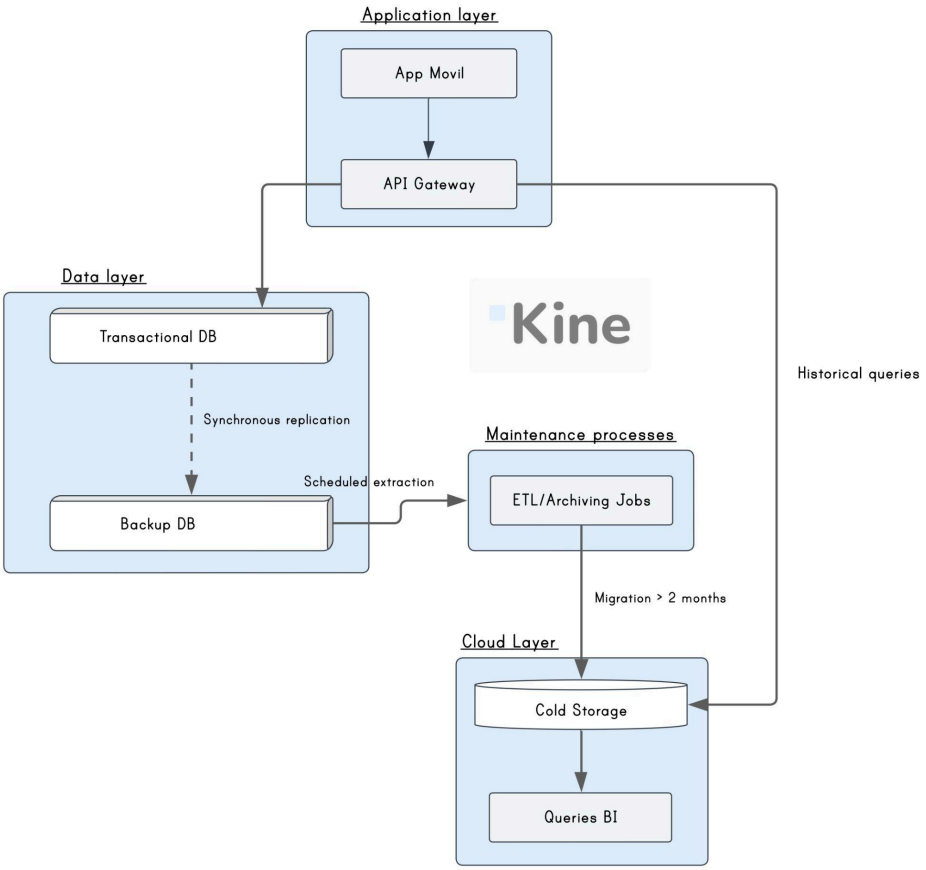
PARALLEL AND DISTRIBUTED DATABASE DESIGN

The Kine system, inspired by digital banking platforms such as Nequi, operates under a transactional model (OLTP), with a constant flow of small, critical operations such as transfers, payments, and balance inquiries. In this type of system, the priority is to ensure the integrity, consistency, and immediate availability of data, aspects that are best aligned with a distributed database architecture. Unlike parallel databases—designed for massive batch processing and complex analytical analysis (OLAP)—Kine does not execute intensive exploratory queries or batch processing that justify fragmentation and computing parallelism.

In addition, the distributed model allows data to be replicated across regions, handles fault tolerance, and scales horizontally without compromising ACID guarantees, which is more suitable for a platform that serves millions of users simultaneously. From a regulatory perspective, historical data storage for legal reasons is also handled efficiently through a hybrid distributed architecture, which combines hot databases for recent data and cold storage for historical backups in the cloud. Thus, the use of a parallel database would not only be unnecessary, but would add technical complexity and cost without real benefits for Kine's functional needs.

This is why the need to maintain transaction histories and comply with regulatory requirements in Colombia (e.g., keeping data online for at least two months and then archiving it) better justifies a distributed architecture rather than a parallel one. A distributed database allows hot operational data to be separated from backups or cold storage, complying with regulations, reducing costs, and improving performance.

HIGH-LEVEL ARCHITECTURAL DESIGN - DISTRIBUTED DATABASE



User interaction

The flow begins when a user accesses the Kine platform from their mobile app or web interface. From there, requests such as transfers, balance inquiries, payments, etc. are sent.

Receipt by the API Gateway

All incoming requests are channeled through an API Gateway, which:

- Centralizes access to services.
- Validates authentications and tokens.
- Decides whether to query recent data (active transactions) or historical data (archived).

Query or write to the transactional database

For everyday operations (latest movements, balance, top-ups, etc.), the Gateway interacts directly with the transactional database cluster, which:

- Is optimized for performance and concurrency.
- Has synchronous replication with one or more regional replicas, ensuring high availability.

Execution of ETL processes

A set of scheduled processes periodically accesses the transactional cluster and:

- Extracts data that exceeds the age threshold (e.g., transactions older than 2 months).
- Transforms and compresses it if necessary.
- Moves it to the cold storage system.

Migration to cold storage (archiving)

Historical transactions are stored in a cloud archiving system. This space:

- Is economical.
- Does not impact the performance of the main system.
- Can be consulted when old information is required.

Historical queries from the Gateway or BI

When a user wants to view old transactions (e.g., more than 2 months old), the API Gateway directs that request to the historical archive. Also, business intelligence (BI) tools or auditors can query the archive directly using engines, without affecting the active database.

Continuous monitoring and coordination

The entire system can be monitored with monitoring tools to ensure that:

- Replication and archiving processes are working correctly.
- The integrity of the migrated data is maintained.
- Errors or bottlenecks are detected.

PERFORMANCE IMPROVEMENT STRATEGIES

1. **Distributed Replication**

Justification	● Kine must guarantee 24/7 availability for financial services. ● Users are distributed across different regions in Colombia. ● Replication keeps services running even during regional failures. ● Kine performs a high number of read operations (like balance queries), which can be offloaded to replicas to avoid overloading the primary database.
Advantages	● High availability: if one node fails, another can take over. ● Faster reads: queries are directed to the nearest replica geographically. ● Reduced latency for users in different regions. Allows maintenance or updates without stopping the service.
Disadvantages	● More complex to configure and monitor. ● Risk of temporary inconsistencies in asynchronous replicas. ● Higher storage costs due to data duplication. ● May require additional hardware or instances.
Potential Challenge	● Replication latency between distant regions. ● Deciding which data to replicate and which to keep local. ● Managing conflicts if multi-master writes are allowed. ● Monitoring replication lag to avoid stale reads.

2. **Offloading to Distributed Cloud Storage**

Justification	● Nequi (now Kine) only shows two months of transactions in the app. ● By law, historical data must be retained but doesn't need to remain online constantly. ● Moving it to cold storage allows: ○ Cost savings. ○ Freeing up space and improving performance in active databases. ○ Compliance with financial regulations. ● This way, Kine can quickly respond to recent queries without being affected by old data.
Advantages	● Cost savings: cloud storage is cheaper for cold data. ● Reduces size and weight of active databases. ● Complies with Colombian financial regulations on historical data retention. ● Allows analytics on large volumes without impacting production systems.

Disadvantages	● Queries on cold data are usually slower. ● ETLs are required to migrate and keep data consistent. ● More complex management of permissions and cloud security. ● Higher latency if integrating cold data into real-time processes.
Potential Challenge	● Clearly defining what data to migrate and when. ● Ensuring traceability for audits. ● Integrating hybrid queries (hot + cold data) without impacting user experience.

3. Horizontal Scaling in Distributed Systems

Justification	● Kine experiences huge traffic spikes during events like:
	  ○ Massive subsidy payments.
	  ○ Key dates like payday or month-end.
	  ○ Banking promotions or campaigns.
	● Horizontal scaling helps manage these peaks without degrading service quality.
	● It's more efficient and cost-effective than vertical scaling (buying bigger servers).
	● Both MongoDB and PostgreSQL offer mature partitioning and horizontal scaling features, ideal for supporting millions of users.

Advantages	● Supports massive user growth without relying on bigger single servers. ● Allows paying only for resources needed during traffic spikes. ● Avoids single points of failure. ● Flexibility to grow with business demands.
Disadvantages	● More complex coordination among nodes. ● Requires careful data design to support partitioning. ● Load balancing becomes critical. ● Increased administrative and monitoring efforts.
Potential Challenge	● Choosing the right partitioning criteria (by user, region, date, etc.). ● Managing distributed transactions across nodes. ● Setting up efficient load balancers. ● Controlling costs when scaling many nodes simultaneously.

IMPROVEMENTS TO WORKSHOP 2

Attributes were added to the Pocket entity in order to add a new feature to our fintech: it allows you to create a savings goal and automatically deduct a balance from the account to achieve it.

i. Pocket		
pocket_id (PK)	serial	PRIMARY KEY
account_id(FK)	int	FOREIGN KEY
name	varchar(20)	UNIQUE KEY
balance	numeric(10,2)	NOT NULL
creation_date	date	NOT NULL
isGoal	boolean	NOT NULL
goal	numeric(10,2)	NULL
regularity	varchar(15)	NULL
quota	numeric(10,2)	NULL