Workshop 3: Concurrency, Parallelism and Distributed Databases AgroClima Prediction System

César Andrés Torres Bernal (20191020147)
Juan David Duarte Ruiz (20191020159)

Universidad Distrital Francisco José de Caldas

Concurrency Analysis

In data-intensive and distributed systems such as AgroClima, concurrency control is a fundamental aspect of system integrity, availability, and performance. Given that multiple components operate in parallel, including real-time data ingestion, predictive modeling, alert generation, and user interaction, there are several critical scenarios where concurrent access to shared data resources occurs. If not properly managed, such concurrency can lead to inconsistencies, data corruption, race conditions, or system bottlenecks.

Concurrency Scenarios in AgroClima Prediction System

a. **Real-Time Data Ingestion and User Querying**
While the ingestion layer pulls real-time weather data from external APIs or synthetic generators and writes to *Weather_Data* in PostgreSQL or MongoDB, users and services may simultaneously read from these collections to view live conditions, generate alerts, or run predictive models. This read-write concurrency requires careful isolation to avoid reading uncommitted or inconsistent data.

b. **Simultaneous User Activity Logging**
Multiple users interacting with the platform generate high-frequency concurrent writes to collections such as *UserActivityLogs* or *UserPreferences*. As these are time-sensitive and may vary in structure, conflicts or race conditions may emerge if operations are not atomic.

c. **Prediction Generation vs. Alert Dispatch**
Machine learning models running in batch mode (via Apache Spark or internal scripts) periodically generate new *ClimatePredictions*. In parallel, the alerting service consumes these predictions to trigger *RiskAlerts* to end-users. If both services access or update overlapping records without transactional guarantees, there is a risk of duplicate alerts, missed triggers, or inconsistent alert content.

d. **Dashboard Analytics During Data Updates**
Business Intelligence modules may execute heavy analytical queries on datasets like *HistoricalWeatherData* or *ClimatePredictions* for visualization or reporting. At the same time, backend processes might update these same records, leading to inconsistent visualizations or slow query performance if locking is not managed.

Potential Concurrency Problems and Solutions

AgroClima operates in a distributed and partially parallel data environment, where data is continuously ingested, transformed, and queried across multiple layers. These operations occur in relational (PostgreSQL) and non-relational (MongoDB) databases, some of which may be replicated or partitioned across nodes. While this enables performance and scalability, it also introduces several complex concurrency challenges that must be addressed at both the architectural and transactional levels.

Race Conditions

Race conditions occur when two or more processes access shared data concurrently and at least one process modifies the data, causing unpredictable results depending on the execution sequence.
A Spark job is updating the *ClimatePredictions* table while the recommendation engine is querying the same record to generate an advisory. If the update partially completes before the read, the user may receive incorrect recommendations based on stale data.

Mitigation Strategies

- Use row-level locking in PostgreSQL when predictions are being updated and joined with recommendations.
- Apply synchronization logic in application services to restrict simultaneous write access when needed.

Lost Updates

Occurs when two concurrent updates to the same record overwrite each other without being aware of the other's changes, resulting in data loss.
Two backend processes update the same *UserPreferences* document within milliseconds of each other, causing one update to be unintentionally discarded.

Mitigation Strategies

- Implement Optimistic Concurrency Control (OCC) by using version numbers or timestamps in both PostgreSQL and MongoDB.

Deadlocks

A deadlock arises when multiple transactions each hold a lock on a resource and attempt to acquire a lock on the other's resource, leading to a cycle of waiting that can never resolve naturally.
A recommendation service locks the *Prediction* table and waits for *Alert*, while the alerting engine does the reverse — causing both to halt.

Mitigation Strategies

- Keep transactions short and narrow in scope, locking only the required rows or documents.
- Rely on database-level deadlock detection: PostgreSQL, for instance, can automatically abort one transaction to break the cycle.

Stale Reads

A stale read occurs when a query retrieves outdated or incomplete data due to asynchronous replication delays or transaction visibility rules.
A BI tool queries *HistoricalWeatherData* from a PostgreSQL read replica while the primary node is still ingesting and committing recent updates. The report reflects incomplete data, potentially misleading decisions.

Mitigation Strategies

- In MongoDB, apply read concern "majority" to ensure that only fully replicated and acknowledged writes are visible to the reader.
- BI dashboards should query from synchronized nodes or use materialized views that update periodically but remain internally consistent.

Replica Inconsistency and Latency

In replicated database architectures, write operations typically go to the primary node, while reads can be served by secondaries. If replication is asynchronous, delays can cause inconsistencies between nodes. An administrator views alert history from a MongoDB secondary that hasn't yet received recent writes from the primary, resulting in missing or outdated records.

Mitigation Strategies

- Use Redis caching only for data with tolerable staleness and implement time-to-live (TTL) mechanisms.
- For PostgreSQL, query the primary node when absolute consistency is required.

Partitioning and Sharding Conflicts

In systems that partition data horizontally across nodes (e.g., by region or time), queries or writes that span multiple partitions can become bottlenecks or require distributed coordination.
A prediction model processes weather data across regions that are stored in separate shards, causing inter-partition joins or requiring distributed transactions.

Mitigation Strategies

- Use partition-aware queries that minimize cross-shard operations.
- Apply distributed transaction coordination only when required, and favor eventual consistency for non-critical updates.