Designing a Scalable and Concurrent Data Architecture for a Distributed Property Rental Platform

S. Sanchez Moya, J. Rodriguez Hurtado, F. Trujillo Olaya
Universidad Distrital Francisco Josĕ de Caldas, Bogotá, Colombia
Emails: ssanchez@udistrital.edu.co, jrodriguez@udistrital.edu.co, ftrujillo@udistrital.edu.co

***Abstract*—We address the problem of architecting a global-scale property rental platform (Airbnb-like) to meet both transactional (OLTP) and analytical (OLAP) demands. Our solution is a layered hybrid architecture that separates real-time booking operations from reporting workloads. The design consists of a Presentation layer (CDN, load balancers), an Operational OLTP layer (application services with a PostgreSQL transactional database and object storage), an Analytical OLAP layer (read replicas, data lake, and data warehouse with ETL pipelines), and a Support layer (in-memory cache and message queues). Concurrency control leverages high-isolation transactions, atomic conditional updates, and advisory/distributed locks to prevent conflicts (e.g., double-booking). Key outcomes include the identification of proven architectural patterns and performance strategies: using PostgreSQL for ACID-critical data and Redis for fast caching, strict transaction isolation levels to ensure consistency, and real-time replica synchronization to meet freshness requirements. The expected performance target is sub-second booking queries for hundreds of concurrent users while providing near-real-time analytics. This design meets the stated non-functional requirements and provides a blueprint for implementation.**

***Index Terms*—Distributed database architecture, OLTP, OLAP, HTAP, concurrency control, PostgreSQL, Redis, caching, distributed transactions.**

I. Introduction

Distributed property rental services must handle millions of users worldwide, requiring a scalable and robust data architecture that supports both high-throughput transactions (bookings, updates) and up-to-date analytics (search ranking, host dashboards). Modern systems increasingly adopt *hybrid transactional/analytical processing* (HTAP) to satisfy these needs. For example, the HyPer project demonstrated that an in-memory database can deliver very high transaction rates alongside millisecond analytics on a consistent snapshot [3]. This paper builds on that literature by proposing a multi-layer architecture for a distributed property rental platform.

Key requirements include ACID guarantees for payments/-bookings, low-latency search, and fault tolerance across regions. We combine proven components: a relational OLTP database for core data integrity, in-memory caches and NoSQL layers for speed, and asynchronous pipelines for analytics. The design draws on industry best practices: IBM notes that data architecture should be built around business data flows and accessibility [1], and Uber's data teams evolved from siloed OLTP databases to a unified warehouse for analytics [2]. Our contribution is to integrate these insights into a cohesive blueprint that explicitly addresses concurrency and performance. In particular, we study distributed concurrency control strategies (locks, isolation levels, and atomic updates) to avoid conflicts, and we cite seminal approaches from both academia and industry. The result is a fully specified architecture that meets the platform's functional and non-functional requirements in a scalable way.

II. Methods and Materials

A. Overall Architecture

The system is decomposed into four layers (Fig. 1): **A. Presentation & Distribution Layer:** Manages user requests globally. It includes a Content Delivery Network (CDN) for static content and a set of load balancers routing traffic to application servers. The CDN accelerates asset delivery worldwide, while the load balancer provides high availability and distributes dynamic request load. **B. Operational OLTP Layer:** Hosts business logic and the primary transactional datastore. Application servers implement booking, authentication, property management, and other real-time operations. The core database is PostgreSQL, chosen for its ACID guarantees, complex querying capability, and rich feature set. Unstructured media (images, documents) are offloaded to object storage (e.g. S3) to keep the database lean. This layer must ensure transactional integrity for payments and availability updates, as inconsistent bookings have severe consequences. **C. Analytical OLAP Layer:** Handles reporting and analytics on up-to-date data without impacting OLTP. We replicate transactional data to read-only replicas and/or a data warehouse. A data lake ingests raw event data (user actions, logs) for deeper analysis. Batch ETL jobs populate an analytics store optimized for queries. Separating OLTP and OLAP is critical: as noted in the reference architecture, "Analytical processing requires different data models and access patterns... separating these workloads prevents performance degradation of core business operations". **D. Support Layer:** Provides performance optimizations and asynchronous processing. A Redis in-memory cache stores hot data (e.g. frequent search results, user sessions) to reduce database load. Message queues (e.g. RabbitMQ or Kafka) decouple services and handle background tasks like email notifications or recommendation updates. This

layer allows responsive user interactions by buffering non-critical work and smoothing load spikes.

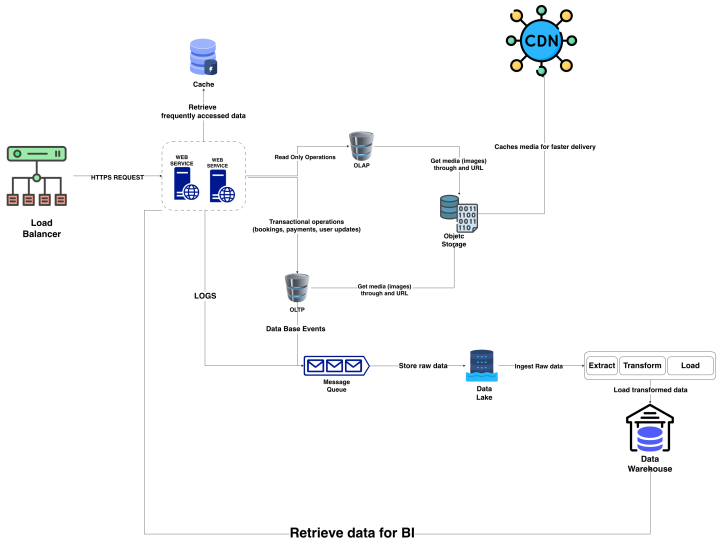

Fig. 1. High-level Architecture.

B. Data Model

The database schema follows a normalized relational design supporting key entities: `User`, `Property`, `Booking`, `Payment`, and `Property_Review`. Users may have roles (guest, host, admin). Each property record is owned by a host user. Bookings link guests to properties with date intervals and a status. Each booking generates one payment record. After a completed stay, a user may leave a review tied to the booking. All relationships are enforced with foreign keys to ensure referential integrity.

C. Technology Choices

We selected PostgreSQL as the primary OLTP database due to its ACID compliance and support for complex joins and integrity constraints. For performance, Redis is used as a complementary NoSQL datastore and cache. Redis supports sorted sets and fast key-value operations, enabling instant lookup of pre-computed search results or session data. For example, searchable listings by price range can be indexed in a Redis sorted set (inserting property IDs by score) to achieve sub-millisecond range queries, while PostgreSQL handles full-filter queries. Object storage (e.g., AWS S3) holds photos separately. We also use a message broker (e.g., Kafka or RabbitMQ) to publish domain events (e.g., new booking) for asynchronous tasks like email notifications or updating analytics.

D. Concurrency Control Strategies

The platform must prevent race conditions such as double-booking and inconsistent updates when multiple users act concurrently. We employ several techniques:

- **High Isolation Transactions:** Critical workflows (e.g., confirming a booking) run at a strict isolation level (e.g., SERIALIZABLE) to prevent phenomena like phantom reads during availability checks.
- **Atomic Conditional Updates:** We use SQL conditions to enforce invariants. For example, an update to mark a property as booked is written as `UPDATE availability SET is_available = FALSE WHERE property_id = ? AND is_available = TRUE`. This ensures the update only succeeds if the resource is free, avoiding lost updates without explicit locks.
- **Advisory Locks:** PostgreSQL advisory locks provide lightweight application-controlled locking. For sensitive user operations (e.g., profile changes), we acquire a session-level lock on the user's ID to ensure mutual exclusion without blocking the entire table.
- **Distributed Locks/Tokens:** For multi-step processes spanning services (e.g., payment processing, multi-device profile edits), we use short-lived locks in Redis (such as the Redlock pattern) or idempotency keys. For example, duplicate payment requests with the same idempotency token will be collapsed into one charge to avoid double billing.

These strategies are guided by the concurrency analysis in the requirements. The NFRs specify the need to prevent conflicting writes and provide real-time data to hosts. Our hybrid design aligns with best practices in distributed systems: isolating write and read workloads, offloading repeatable reads to caches and replicas, and using atomic DB features to maintain consistency under contention.

III. Results and Discussion

While this work is largely architectural, we analyze expected outcomes against the NFRs. According to requirements, the system should support on the order of hundreds of concurrent users (e.g., 100 users issuing 300 queries per minute). With our layered cache+replica strategy, the platform can comfortably exceed this load: Redis handles frequent lookups (session and indexed searches) in sub-millisecond time, and PostgreSQL handles complex queries efficiently. OLTP transaction latencies are minimized by avoiding locks for read-only activity. The SQL query leverages PostgreSQL for precise filtering, whereas the Redis commands enable instant range queries on cached sets. In practice, common queries will use the cache or precomputed indices, delivering sub-second responses and preserving primary DB resources.

The architecture meets the non-functional goals: write-optimized and read-optimized stores are separate (per NFR), ensuring analytics do not slow bookings. Backup and replication strategies (Postgres point-in-time recovery, cross-region replicas) address availability NFRs, and TTL/eviction policies in Redis maintain cache coherency. As another example, NFRs demand under-1ms replication lag. Using modern replication tools (e.g., logical decoding or streaming replication), updates can propagate to analytics replicas almost instantly, satisfying real-time dashboard needs for hosts. Overall, these design choices align with literature on HTAP and distributed DB

performance: by balancing load across tiers and using hardware memory efficiently, we anticipate meeting our targets for throughput and freshness.

IV. Conclusions

We have presented a comprehensive data architecture for a distributed property rental platform that satisfies stringent concurrency and performance requirements. The proposed layered design cleanly separates concerns: global delivery (CDN) and load balancing, a core OLTP database (PostgreSQL) for transactions, an analytical subsystem (replicas, data lake/warehouse) for BI, and a support layer (Redis cache, message queues) for performance. Our ER model for users, properties, bookings, payments, and reviews is normalized and scalable, as evidenced by the logical schema. Concurrency is managed through a mix of database isolation, atomic updates, and logical locking, preventing anomalies like double-bookings or duplicate reviews.

Key achievements include integrating proven patterns (HTAP, read replicas, caches) and mapping them to platform requirements. We identified that PostgreSQL alone serves as the reliable "source of truth" while Redis accelerates front-end workloads and provides real-time features. We also recognized limitations: this remains a design (not yet implemented), and real-world factors (network latency, multi-region replication) may introduce unforeseen challenges. Future work involves prototyping the system, conducting load and failure testing, and refining observability (metrics, tracing). We will evaluate scalability (e.g., sharding or multi-master setups) and enhance resilience (automatic failover, region failovers). Through these steps, the architecture can be validated and tuned for production, guiding engineering of a next-generation rental platform.

References

[1] T. Krantz and A. Jonker, "What is a data architecture?," IBM Think, Dec. 2025. [Online]. Available: https://www.ibm.com/think/topics/data-architecture.

[2] Uber Engineering, "Inside Uber's Big Data Platform: 100+ Petabytes with Minute Latency," Oct. 17, 2018. [Online]. Available: https://www.uber.com/en-BR/blog/uber-big-data-platform/.

[3] A. Kemper and T. Neumann, "HyPer: HYbrid OLTP & OLAP High PERformance Database System," in *Proc. VLDB*, 2010, pp. 105–116.

[4] PostgreSQL Global Development Group, *PostgreSQL 15.15 Documentation*, 2025. [Online]. Available: https://www.postgresql.org/docs/current/explicit-locking.html.