Databases II

Semester 2025-III

Project Definition and Database Modeling

Julian Camilo Espinosa Morales
Cod. 20191020073

Pablo Alejandro Montaña Moreno
Cod. 20201020090

Juan Carlos Duarte Sandoval
Cod. 20212020149

Part I
Workshop 1

Project Context

The rapid expansion of digital platforms for property rentals has reshaped how travelers and hosts interact globally. Nevertheless, in developing regions such as Colombia, the adoption of these solutions remains limited due to high operational costs, infrastructure challenges, and the lack of locally adapted analytics. The *ColombianStay* project emerges as an initiative to address these issues by providing an affordable, scalable, and analytically empowered accommodation management platform tailored to regional needs.

The system architecture of *ColombianStay* is grounded in three core architectural models that together ensure flexibility, reliability, and analytical depth. At the database level, a **Client–Server DBMS architecture** with data replication has been adopted to enhance system availability and scalability. This ensures continuous access to data even in distributed or low-connectivity environments. From a software design perspective, *ColombianStay* follows a **Service-Oriented Architecture (SOA)** approach, enabling modular development and easy integration of services such as reservation management, user handling, recommendation engines, and business intelligence modules. For analytical operations, a **Data Warehouse architecture** supports centralized and structured data analysis, empowering decision-making through BI dashboards and performance reports.

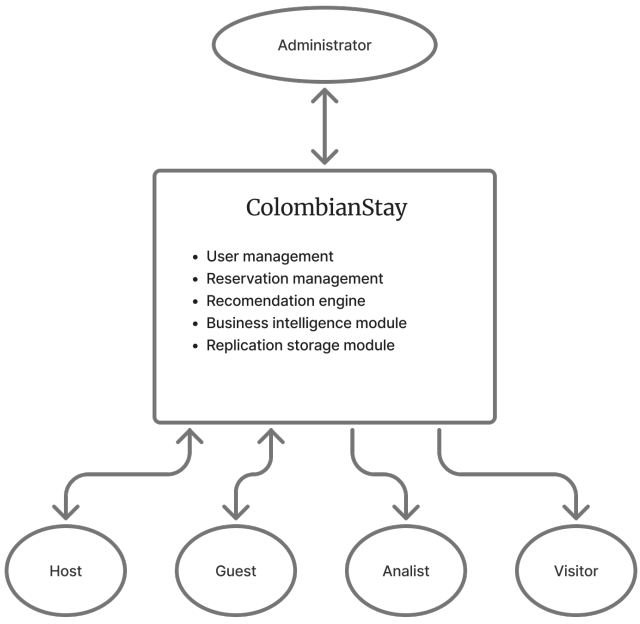


Figure 1: General context diagram

The general context diagram illustrates the main external actors—such as administrators, hosts, guests, and visitors—and their interactions with the *ColombianStay* system. Each user type interacts with the platform through specific modules designed to fulfill their functional needs, from accommodation listings and reservations to system management and data analysis.

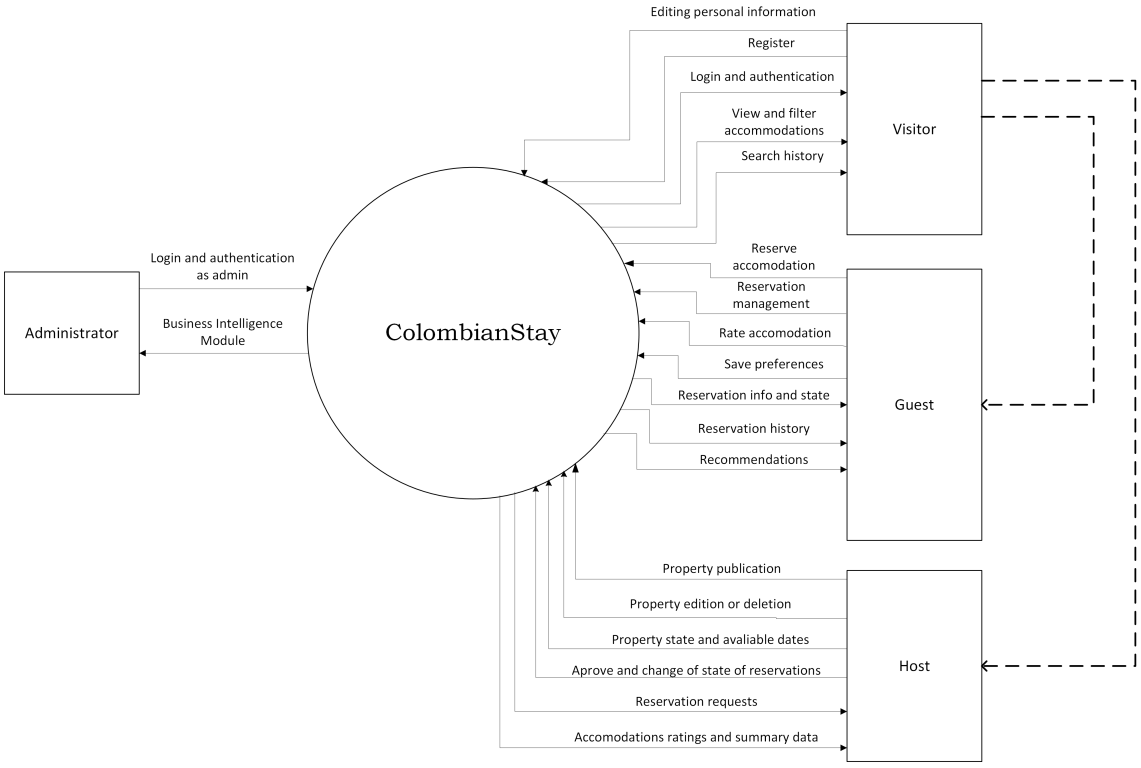


Figure 2: System interaction context diagram

The detailed system interaction context diagram expands this view by showing the specific information flows between these actors and the core *ColombianStay* modules, including user management, reservation handling, and data replication services.

Together, these architectural and interaction models provide a comprehensive overview of *ColombianStay*'s design philosophy: a robust, modular, and data-driven solution that seeks to democratize access to digital tourism technologies in Colombia and similar emerging markets.

Business Model Canvas

Key Partners

Payment processors and gateways (Stripe, PayU, Wompi): handle secure transactions, refunds, and PCI compliance.

Local tourism offices and travel agencies: provide verified local experiences and increase platform credibility.

CDN and object-storage providers (AWS S3, Cloudflare): ensure fast and reliable delivery of photos and videos.

Cleaning, verification, and property-inspection services: maintain quality and trust of listings.

Technology partners (search engines, recommendation algorithms, analytics tools): support innovation and scalability.

Key Activities

Operating the web/mobile platform

Host verification and trust/safety checks

Managing bookings, payments, and refunds

Media upload, transcoding, and delivery

Developing recommendation engine and analytics dashboards

Key Resources

Object storage for photos and videos

Services for functionality

Engineering and support teams

Machine learning models for recommendations

Storages machines for multi-location replication.

Value Propositions

Localized and verified listings covering Colombian cities, rural stays, and unique experiences not found on global platforms.

Secure, transparent booking and payment flows that reduce fraud and build user trust.

Personalized recommendations and smart search based on traveler preferences, location, and seasonality.

Modular oriented service architecture enabling rapid feature development by independent teams and easier scalability.

High-quality media delivery so guests can confidently view listings with detailed photos and videos.

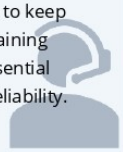
Customer Relationships

Self-service portal for travelers and hosts

24/7 support chat

Trust & safety verification and reviews system

Automated processes to keep costs low while maintaining reliabilitylow while essential verification ensures reliability.



Channels

Web application

Mobile web access

Partner integrations with travel agencies

Social media presence

Collaboration with local tourism boards

Customer Segments

Travelers (domestic and international) looking for trustworthy, localized stays and experiences.

Hosts (property owners) seeking to monetize their spaces with secure and easy-to-use tools.

Local experience/activity providers offering tours, workshops, or services complementary to stays.

Administrators and managers who oversee operations, compliance, and analytics of the platform.

Cost Structure

With 1000 to 3000 active users and 150 to 400 accommodations:

Infrastructure costs: hosting, databases, storage, security, and monitoring. Aprox: $6000 to $10000

Engineering and development: salaries of developers, designers, DevOps, and data scientists. Aprox: $90000 to $120000

Customer support operations: 24/7 chat, verification team, dispute handling. Aprox: $10000 to $ 20000

Payment gateway fees and chargebacks for processing transactions. Aprox: $7000 to $8000

Marketing and partnerships: campaigns, affiliate payouts, and tourism board collaborations. Aprox: $10000 to $20000

Revenue Streams

Booking fees: percentage-based commission on each reservation from travelers or hosts. Aprox: $24 000 per year

Host subscriptions or premium listings: monthly or annual plans offering higher visibility and extra features. Aprox: $4 800 per year

Featured placement and advertising: paid options for hosts or local businesses to appear in top search results. Aprox:$30 000 per year

Ancillary services: partnerships with travel insurance, transport providers, or experiences generating referral income. Aprox: $7 000 per year

Requirements Documentation

Functional Requirements (selected & prioritized)

- User registration & login (email/social auth, roles: guest/host/admin).
- Host and guest accounts self management.
- Search & filters by location, date, guests, price, amenities.
- Booking/reservation flow with atomic availability checks and payment integration.
- Payments & refunds of reservations.
- Reviews & ratings for hosts and guests.
- Media management: upload, transcode, store, share photos/videos.
- Recommendation engine: personalized suggested listings.
- Admin dashboard & BI: occupancy, revenue, fraud alerts, KPIs.

Non-Functional Requirements

- **Performance:** The system must respond to basic operations (login, search, booking, viewing accommodations) in less than 3 seconds under average user load, and queries to the data warehouse (BI or recommendations) must be executed in less than 5 seconds for medium-sized data sets.
- **Scalability:** The system must support a 150% increase in the number of accommodation listings without noticeable performance loss.
- **Availability:** The system must have 95% availability during testing, ensuring that core services are accessible.
- **Consistency:** Each booking transaction must be correctly reflected in the operational warehouse and subsequently in the data warehouse. ETL data must ensure that there are no duplicates in the data warehouse fact tables.
- **Security:** OAuth2/JWT, password hashing, input sanitization, PCI-DSS compliance for payments, read and write operations must be performed using authenticated services (tokens or sessions).
- **Maintainability:** Each module must have technical documentation and comments in the code. Connection configurations and credentials must be stored in external files.

User History

Title: Guest - Search & Book	**Priority:** High	**Estimate:** 10 h
User Story: As a guest, I want to search for listings by city and date so that I can book a stay.		
Acceptance Criteria: Given an guest and an existing accommodation of his search when the guest booked it for a certain period then the accommodation is reserved by the guest and is occupied on that period		

Title: Host - Create Listing	**Priority:** High	**Estimate:** 12 h
User Story: As a host, I want create a listing with photos and availability so that receive bookings.		
Acceptance Criteria: Given a host with images and information about an accommodation when the host upload it those to the platform then listings created.		

Title: User - Authentication	**Priority:** Medium	**Estimate:** 6 h
User Story: As a user, I want log in using email/password so that I don't need to remember another account		
Acceptance Criteria: Given an user account when the user sign up with the account credentials then the user access to the platform with his respective role and its available options.		

Title: Admin - Analytics Dashboard	**Priority:** Medium	**Estimate:** 35 h
User Story: As a administrator, I want to see monthly revenue and occupancy so that I can evaluate business health.		
Acceptance Criteria: Given data marts of essential information when the administrator select the BI option then the administrator get dashboards of essential information.		

Title: Guest - Review Accommodation	**Priority:** Low	**Estimate:** 16 h
User Story: As a guest, I want leave a review after my stay so that others can benefit and award the accommodation.		
Acceptance Criteria: Given an guest after had finished his stay when the guest review the accommodation then the review his shared to the host and post on his accommodation		

Initial Database Architecture

High-level Database Architecture

The project is based on a client-server architecture, with a service-oriented architecture (SOA) and a data warehouse as the analytical core for data management and exploitation. This architecture seeks to ensure a modular, scalable platform capable of offering both operational and business analytics services.

Justification for Architectural Decisions

- **Client-server:** Ensures the database is connected through an API, thus contributing to the loose physical and logical coupling of the database with the program's logical system.
- **Service-oriented architecture (SOA):** Facilitates the system's modularity; each feature (users, reservations, BI, recommendations) can be developed and deployed independently, allowing the development team to work in parallel.

- **Data Warehouse:** Provides an efficient solution for consolidating and analyzing large volumes of data, ensuring rapid responses to business analytics queries without overloading the transactional system.
- **Scalability and sustainability:** The architecture is lightweight enough to run in limited environments, yet scalable to distributed infrastructures in later phases.

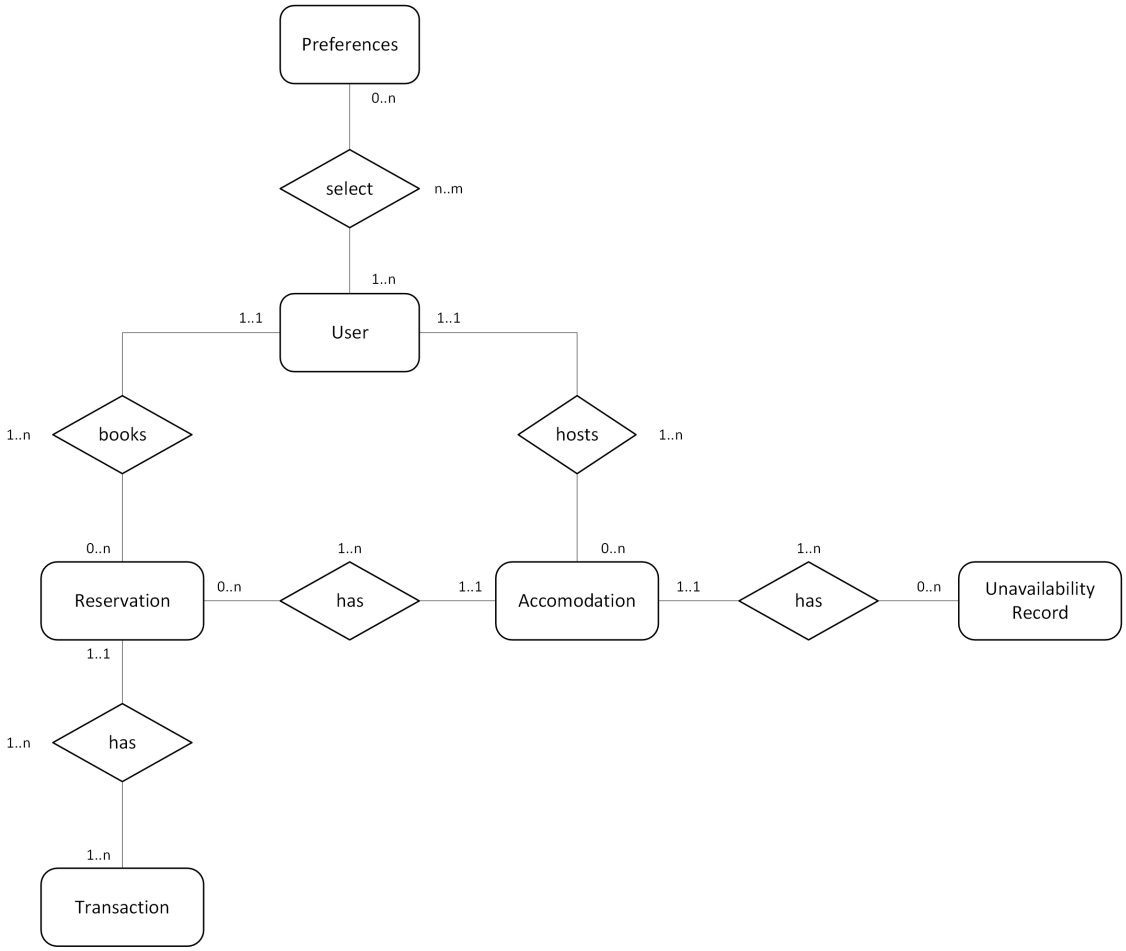
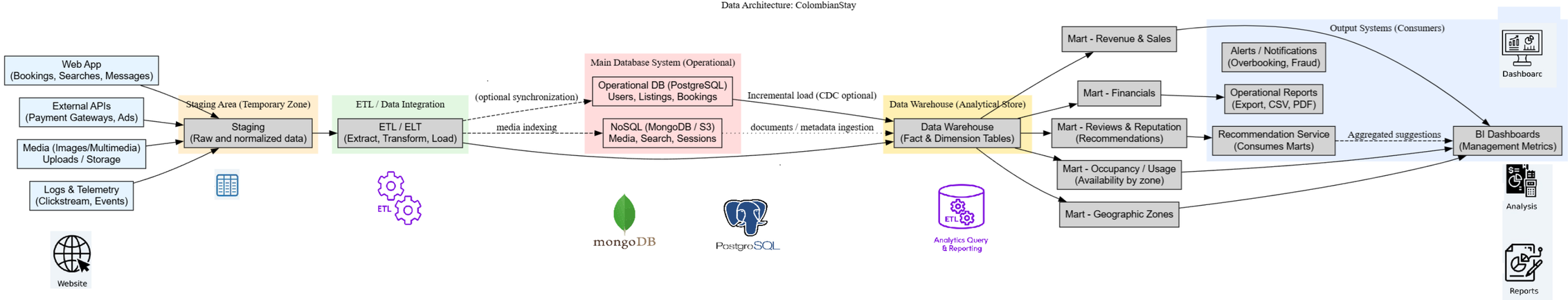


Figure 3: Entity Relation model

Part II

Work Shop 2

Data System Architecture



Data System Architecture

The data architecture of ColombianStay follows a simplified and well-justified structure focused on operational consistency, analytical support, and scalability through replication. Unlike the earlier draft, this version clarifies the data flow and explains why each component exists without relying on unnecessary distributed technologies.

Data Sources

Data enters the platform through the web application (bookings, listing updates, searches, authentication), system logs, external services such as payment gateways, and media uploads. These heterogeneous sources generate operational and analytical information that must be validated and prepared before persistence.

Staging Layer

The staging layer temporarily hosts raw information coming from Data Sources. This separation avoids introducing inconsistencies into the operational database and supports preliminary validation, cleaning, and format alignment before transformation.

ETL Processes

ETL processes handle the extraction of staging data, apply business rules, normalize structures, and load curated information into both the operational database and the data warehouse. ETL execution is designed to minimize the load on the primary database by using read replicas for extraction whenever possible.

Operational Database (PostgreSQL)

The system uses a primary–replica configuration:

- The **primary node** executes all write operations such as bookings, payments, and profile updates.
- **Replica nodes** serve frequent read requests such as listing searches, availability lookups, and analytics queries.

This approach is cost-effective, easy to implement, and directly aligned with the system's non-functional requirements of performance, scalability, and availability.

A complementary NoSQL store (MongoDB) contains semi-structured data including media metadata, search-optimized documents, and rapidly changing attributes. This improves page rendering performance and avoids complex relational joins.

Data Warehouse

The data warehouse stores historical and aggregated information using star or snowflake schemas. It is populated periodically through ETL jobs that read from replicas to reduce load on the primary system. The warehouse supports BI dashboards, reporting, and time-series analysis.

Data Marts

Data marts simplify access to specialized analytical domains such as occupancy, financial KPIs, and review insights. They reduce execution time for dashboards and allow each area to access only the information it needs.

Output Systems

Dashboards, reports, alerts, and the recommendation engine consume processed information from the data warehouse and data marts. Output requests also leverage replica servers to reduce latency and maintain system responsiveness.

Justification of Technologies

- **PostgreSQL with replication** ensures ACID compliance for bookings and payments while offering scalable reads through replicas.
- **MongoDB** supports flexible queries for heterogeneous document structures and improves rendering performance for listing pages.

This hybrid design balances consistency, performance, and schema flexibility, matching the system's functional and non-functional requirements.

Information Requirements

The platform depends on different categories of information to support its operational workflows, analytical needs, and user experience. These requirements also align with the non-functional requirements related to performance, scalability, and data availability.

Booking and Availability Information

The system must retrieve up-to-date availability for each listing, filtered by location, date, and guest capacity. Searches and general browsing run on read replicas for performance, while booking confirmation queries are executed on the primary node to guarantee consistency.

Host and Listing Information

Host dashboards require listing data including title, description, price, amenities, and occupancy patterns. Editing operations go to the primary node, while viewing operations are served from replicas to reduce load.

User Authentication and Profiles

User profiles include authentication credentials, roles, contact information, and activity history. Authentication writes occur in the primary server, while profile reads rely on replica servers to meet system response-time requirements.

Reviews and Ratings

The system retrieves reviews and ratings associated with listings and hosts. Read-heavy workloads route to replicas, while new review submissions write to the primary node.

Payments and Financial Reports

Payment information includes transaction status, amounts, timestamps, and refund tracking. Aggregated monthly revenue, commission metrics, and payout summaries are derived using the data warehouse and financial data marts.

Media Metadata and Analytics

Media metadata includes URLs, upload timestamps, resolutions, and associated listing identifiers. Analytics require user activity logs, page impressions, and search patterns to support recommendations and BI dashboards. These operations rely heavily on replica reads to avoid performance degradation.

Query Proposals

The proposed platform integrates both relational and non-relational databases to support different use cases:

- **Relational (PostgreSQL)**: transactional data consistency (users, listings, bookings, payments, reviews).
- **NoSQL (MongoDB)**: fast document retrieval and denormalized views for search, recommendations, and caching.

Relational Database Queries (PostgreSQL)

Query 1 — Retrieve the most popular listings by city
Purpose: Identify the top listings in each city based on the number of bookings.
Business goal: Helps the platform recommend top-rated or most-booked stays to users.

```
SELECT
    l.city,
    l.title AS listing_name,
    COUNT(b.booking_id) AS total_bookings
FROM listings l
JOIN bookings b ON l.listing_id = b.listing_id
-- Maybe implementing another states list
WHEre b.status = 'confirmed'
GROUP BY l.city, l.title
ORDER BY l.city, total_bookings DESC
LIMIT 10;
```

Insight: Provides marketing and analytics teams with data on trending destinations in order gather information and improving the service.

Query 2 — Calculate total earnings per host
Purpose: Determine total revenue generated by each host through completed bookings.
Business goal: Enables hosts to visualize their performance and supports payout operations.

```
SELECT
    u.name AS host_name,
    SUM(p.amount) AS total_earnings
FROM users u
JOIN listings l ON u.user_id = l.host_id
JOIN bookings b ON l.listing_id = b.listing_id
JOIN payments p ON b.booking_id = p.booking_id
WHERE p.status = 'completed'
GROUP BY u.name
ORDER BY total_earnings DESC;
```

Insight: Shows which hosts contribute the most revenue and supports their financial reporting, something that might help hosts a lot with finantial info.

Query 3 — Get guest booking history
Purpose: Retrieve all past and current bookings for a specific user.
Business goal: Supports usser dashboards and customer service operations.

```
SELECT
    b.booking_id,
    l.title AS listing,
    b.start_date,
    b.end_date,
    b.status,
    p.amount AS payment_amount
FROM bookings b
JOIN listings l ON b.listing_id = l.listing_id
LEFT JOIN payments p ON b.booking_id = p.booking_id
WHERE b.guest_id = 'exampleUserUid'
ORDER BY b.start_date DESC;
```

Insight: Provides personalized views of guest activity.

Non-Relational Database Queries (MongoDB)

Query 1 — Retrieve denormalized listing document
Purpose: Fetch a full listing document with host, location, and reviews embedded.
Business goal: Enables fast rendering of listing pages with minimal joins.

```
db.listings.find(
  { "listing_id": "eexamplelist:)" },
  {
    title: 1,
    location: 1,
    price_per_night: 1,
    "host.name": 1,
    "reviews.comment": 1,
    "reviews.rating": 1,
    "media.file_url": 1
  }
);
```

Insight: Supports high-performance reads in the web frontend.

Query 2 — Find top-rated listings
Purpose: Retrieve listings with average rating above 4.5 stars.
Business goal: Drives recommendation and discovery modules.

```
db.listings.aggregate([
  { $unwind: "$reviews" },
  { $group: {
      _id: "$listing_id",
      title: { $first: "$title" },
      avg_rating: { $avg: "$reviews.rating" }
  }},
  { $match: { avg_rating: { $gte: 4.5 } }},
  { $sort: { avg_rating: -1 }},
  { $limit: 10 }
]);
```

Insight: Quickly identifies top-quality listings without heavy relational joins.

Query 3 — Track user activity logs
Purpose: Retrieve recent actions from the event stream stored in MongoDB (e.g., viewed listings, searches).
Business goal: Enhances personalization and activity tracking for analytics.

```
db.user_activity.find(
  { "user_id": "exampleuseruid" },
  { action: 1, listing_id: 1, timestamp: 1 }
).sort({ timestamp: -1 }).limit(20);
```

Insight: Provides behavioral data for analytics and recommendation systems, something that can be scalable in the future to a better algorithm.

Summary Table

Query Type	Database	Purpose	Output Example
Top listings by city	PostgreSQL	Identify popular destinations	City + Listing + Count
Host earnings report	PostgreSQL	Financial summary	Host + Total revenue
Guest booking history	PostgreSQL	Personalized dashboard	Booking list
Listing document	MongoDB	Quick read access	JSON listing
Top-rated listings	MongoDB	Recommendations	JSON top 10
User activity log	MongoDB	Behavioral insights	Recent actions

Integration Insight

Relational (PostgreSQL) supports transactions, consistency, and reliable data for payments and bookings.
NoSQL (MongoDB) enables fast document retrieval for liistings, search, and recommendations.
Together, they can help us to fulfill both real-time and analytical needs of the platform.

Part III
Work Shop 3

Concurrency analysis

This section identifies where concurrent access occurs in the ColombianStay platform, describes potential problems, and proposes concrete mitigations from both the application and database architecture perspectives. Recommendations emphasize correctness for critical operations like bookings or payments while preserving throughput for read-heavy features.

Situations where simultaneous access occurs

- **Booking a stay:** Many guests may try to reserve the same property or the same dates at the same moment.
- **Payment attempts:** Guests may retry payments, or the payment service may automatically resend the request.
- **Hosts editing listings:** A host might update prices or availability while guests are viewing the listing.
- **Posting reviews:** Many guests may leave reviews for the same listing after staying there.
- **Uploading media:** Several uploads may happen at once for the same listing, such as photos or videos.
- **Analytics and system events:** Large numbers of events (views, bookings, clicks) are recorded at the same time and processed for reporting.

Possible problems when this happens

- **Overlapping reservations:** Two guests may both think the dates are available and complete a booking at the same time.
- **Lost changes:** If two people edit the same listing at once, one person's update may erase the other's.
- **Conflicting information:** Data may appear inconsistent if different parts of the system read it while it is being updated.
- **System freezes or waiting loops:** Two processes may block each other while waiting for access to the same data.
- **Out-of-date information:** Pages such as analytics or recommendations may temporarily show older information if updates happen faster than the system can refresh.

Solutions and design decisions

- **Keep critical operations short:** Booking-related operations should only lock the exact dates needed and finish quickly.
- **Prevent duplicate actions:** Payment requests should use a unique identifier so repeated attempts do not charge the guest twice.
- **Use the right level of protection:** Reservation and payment operations need strong protection so only one can succeed at a time, while search and browsing can allow more flexibility.
- **Check for changes before saving:** When hosts edit listings, the system should verify whether someone else has modified the listing first.
- **Retry when necessary:** If a conflict happens, the system can simply retry the operation after a brief pause.
- **Record updates in order:** Events used for analytics or recommendations should include unique IDs so duplicates are ignored.
- **Monitor system behavior:** Track delays, waiting times, and conflicts to improve system performance.

Concrete proposals and examples

Booking / reservation (PostgreSQL)

For the booking flow we recommend a short, authoritative transaction that:

1. Reads candidate availability using a query that locks the affected calendar rows (row-level lock).
2. Inserts the booking (or marks a pending booking), then processes payment.
3. Commits only after payment confirmation (or uses a two-phase approach with compensation if payment is external).

Example using `SELECT ... FOR UPDATE`:

```
BEGIN;

-- lock availability row(s) for the listing and date range
SELECT * FROM listing_availability
 WHERE listing_id = :listing_id
   AND date >= :start_date AND date < :end_date
 FOR UPDATE;

-- confirm none of the rows are already reserved, then create booking
INSERT INTO bookings (booking_id, listing_id, guest_id, start_date, end_date, status)
 VALUES (:booking_id, :listing_id, :guest_id, :start_date, :end_date, 'pending');
```

Notes / trade-offs:

- The `FOR UPDATE` locks prevent concurrent transactions from reading and booking the same slots.
- Keep the transaction short: ideally perform payment (external call) outside the locked transaction, but use a short-lived "pending" state and a separate confirmation transaction; or use a payment gateway with synchronous confirmation and very short lock windows.
- If strict serializability is required, use `SET TRANSACTION ISOLATION LEVEL SERIALIZABLE` for the critical section, but be prepared to retry transactions on serialization failures.

Optimistic concurrency for listing edits

For host edits to listing metadata or availability calendar managed via GUI, use optimistic concurrency control with a `version` column.

```
-- Table: listings (listing_id PK, ..., version integer)
-- Update with optimistic check
UPDATE listings
 SET title = :title, price = :price, version = version + 1
 WHERE listing_id = :listing_id AND version = :expected_version;
-- If affected_rows == 0 => conflict detected; notify user to refresh
```

Notes: optimistic control minimizes locking and provides a user-friendly conflict resolution: if the update fails, the UI can show the latest data and ask the host to reapply changes.

Atomic availability change in MongoDB (NoSQL)

If part of availability or session data is kept in MongoDB, use an atomic update (e.g., `findOneAndUpdate`) to reserve a slot.

```
db.availability.findOneAndUpdate(
  { listing_id: "L1", date: "2025-11-20", reserved: false },
  { $set: { reserved: true, booking_id: "B123" } }
);
-- if result is null => someone else reserved the slot
```

Notes: MongoDB single-document updates are atomic; when reservations span multiple documents, use multi-document transactions (replica-set required) or perform application-level coordination.

Payment concurrency and idempotency

- Require an **idempotency key** for each payment attempt. Store the key and payment status so repeated requests do not create duplicate charges.
- Use database-backed status transitions: `payment.status` with controlled state machine (pending → completed/refunded/failed).

- Example: insert payment row only if idempotency key not present:

```
INSERT INTO payments (payment_id, booking_id, amount, status, idempotency_key)
 SELECT :payment_id, :booking_id, :amount, 'pending', :key
 WHERE NOT EXISTS (
   SELECT 1 FROM payments WHERE idempotency_key = :key
 );
```

Deadlock avoidance and resolution

- **Acquire locks in a consistent order:** design code paths such that rows or resources are locked in the same sequence (e.g., always lock listing row before user row).
- **Keep transactions short:** reduce contention windows.
- **Use optimistic concurrency where feasible:** reduces the need for long-held locks.
- **Implement retry with exponential backoff:** handle transient deadlocks by catching database-specific deadlock exceptions and retrying safely.
- Monitor deadlock frequency (PostgreSQL `pg_locks`, logs) and tune application logic.

Event-driven / eventual-consistency flows

- Use event streaming (e.g., Kafka) for asynchronous propagation to analytics, caches, and recommendation pipelines. Accept eventual consistency for BI and recommended lists.
- Ensure idempotent consumers and at-least-once processing semantics: store processed offsets and deduplicate by event id.
- For derived aggregates that must be timely (e.g., top listings), consider materialized views or periodic batch updates plus incremental updates via events.

Distributed locks and cross-service coordination

- For critical cross-service coordination (rare), consider a Redis-based distributed lock (e.g., Redlock) with short TTLs. Use this sparingly — prefer database transactions when the authoritative data resides in the database.
- Always design for failure: ensure lock TTLs, health checks, and fallback logic in case a lock owner crashes.

Testing, monitoring and operational suggestions

- **Stress tests:** simulate heavy concurrent booking attempts on hot listings to validate locking strategy and retry behavior.
- **Chaos testing:** induce failures during bookings/payments to verify compensation logic and no-duplicate guarantees.

- **Monitoring:** collect metrics on transaction latency, serialization failures, lock wait times, deadlock counts, and idempotency-key reuse.
- **Alerts:** create alarms for high rates of serialization failures or long-running transactions.

Examples of how these solutions apply

Booking a property

When a guest tries to book a property, the system temporarily locks the specific dates of the stay. While these dates are locked, other guests cannot book them. If two guests try at the same time, the second one is asked to try different dates.

Editing a listing

When a host edits the price or description, the system checks whether someone else already made changes. If so, the host is asked to refresh the page and update the latest version.

Processing payments

Each payment attempt includes a unique identifier. If the same request is sent more than once, the system recognizes it and prevents duplicate charges.

Preventing system freezes

The platform avoids long operations that require holding data for too long. This reduces the chance that two processes block each other. If a conflict cannot be avoided, the system repeats the operation automatically.

Summary

To maintain reliability, the system protects the most important operations (bookings and payments), allows flexibility for browsing and searching, and uses simple conflict-handling strategies such as retries, version checks, and short-lived locks. These decisions help ensure both accuracy and performance even when many users interact with the platform at the same time.

High-Level Parallel Database Design

High-Level Design Overview

The ColombianStay platform incorporates a distributed database architecture based on PostgreSQL's primary–replica model, aligned with the **Client–Server Model**, where clients issue queries and servers store and serve the data. In this design, a single primary server handles all write operations, while multiple additional servers maintain synchronized replicas used exclusively for read operations.

This structure distributes the workload across several machines, enabling the backend services to route data access in a structured way: write requests such as booking operations or host updates

are sent to the primary server, while read requests such as browsing listings or viewing user profiles are sent to any available replica. This improves responsiveness, increases total request-handling capacity, and enhances system reliability without introducing data fragmentation.

The distributed setup integrates through a read/write routing layer at the application server level, which determines whether a given request should be directed to the primary node or any of the replica nodes. This preserves consistency for write operations while improving efficiency for read-heavy usage patterns.

How Data and Queries Are Distributed

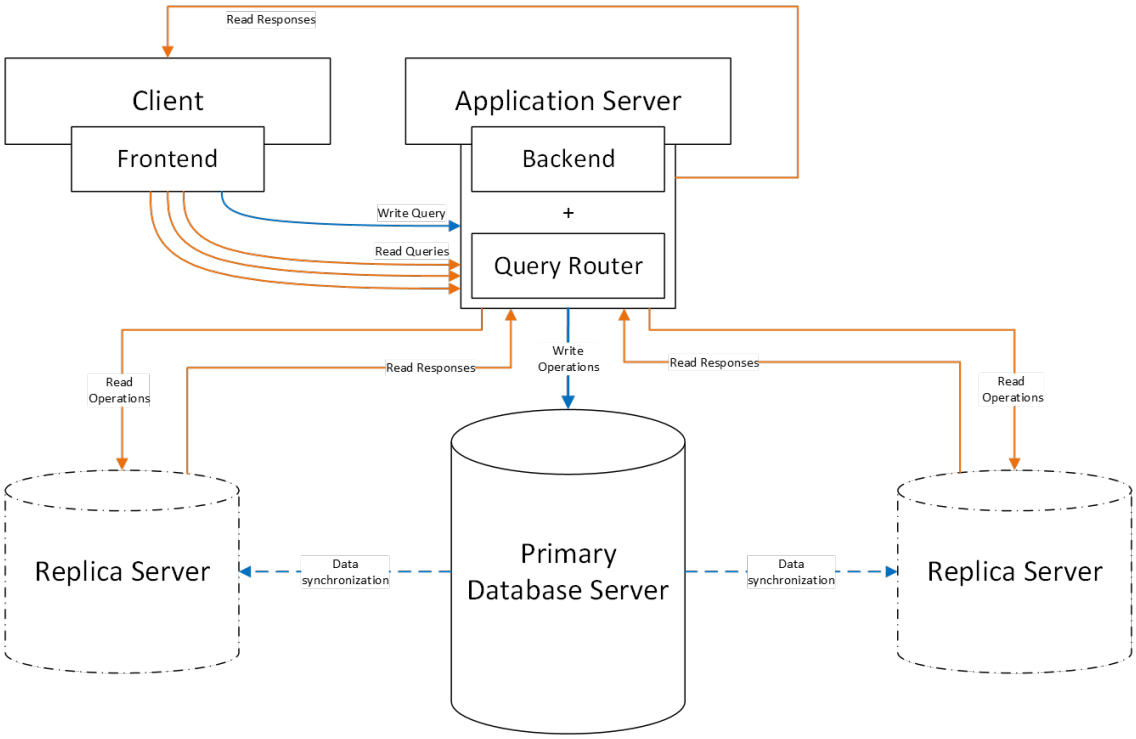


Figure 4: Proposed distributed DB diagram of ColombianStay

Data Distribution Across Servers.

Instead of applying sharding or partition-based distribution, ColombianStay uses full replication of the database across multiple machines. The distribution mechanism works as follows:

- **Primary Server:** Stores the authoritative dataset and processes all update operations. Its Write-Ahead Log (WAL) is the source used to synchronize the replicas.

- **Replica Servers:** Contain synchronized copies of the full dataset. These servers continuously receive and apply WAL segments from the primary server, ensuring that their state remains closely aligned with the main system.

Since all replicas store the complete dataset, any server can independently answer read queries without cross-node communication. This avoids distributed transactions and reduces architectural complexity.

Query Distribution Across Servers.

To make efficient use of the distributed environment, ColombianStay adopts a read/write separation strategy:

- **Write queries** (INSERT, UPDATE, DELETE, transactional operations) are routed exclusively to the primary database server to maintain strict consistency.
- **Read queries** (SELECT operations and browsing-oriented requests) are routed to one of the replica servers, allowing the system to handle increasing user demand by distributing read traffic across available machines.

A routing layer or connection pooler (such as PgBouncer configured with read/write rules) directs each incoming request to the appropriate server, balancing the load while preserving transactional correctness. PostgreSQL's execution capabilities for complex SELECT statements within each server further improve performance in this distributed setup.

Justification of the Architectural Choices

Alignment With a Query-Intensive Platform.

Platforms with frequent browsing activity must support large volumes of read operations. Using multiple replica servers allows the system to handle significant user demand by distributing read traffic, improving responsiveness and reducing congestion on the primary server.

Strong Consistency Requirements for Transactional Operations.

ColombianStay includes reservation-like interaction flows that must avoid inconsistencies, such as double bookings or conflicting updates. A single primary server managing all write operations ensures correct serialization of transactions and preserves ACID guarantees, avoiding the need for complex coordination protocols.

High Availability and Fault Tolerance.

Replica servers act as redundant nodes that can take over if the primary server becomes unavailable. PostgreSQL's failover and promotion mechanisms allow replica nodes to replace the primary server when required, improving system resilience and continuity.

Scalable Read Capacity Without Data Fragmentation.

Because the entire dataset is fully replicated, each replica can answer any read query independently. This avoids the complexity of distributing data across multiple partitions while still providing scalable capacity through the addition of more replica servers.

Summary

The distributed architecture designed for ColombianStay follows the Client–Server Model, where clients submit requests and specialized servers manage all data-related operations. PostgreSQL's primary–replica configuration ensures strong consistency for critical write operations while distributing read traffic across multiple servers to enhance scalability and responsiveness. This approach supports high availability, efficient workload distribution, and reliable data management without requiring dataset partitioning.

Performance Improvement Strategies

1. Horizontal Scaling (Scale-Out of Database Instances)

To support increasing query volumes and maintain low latency during high-demand periods, ColombianStay applies horizontal scaling by deploying multiple parallel database instances. Instead of relying on a single high-performance server, workloads are distributed across several database instances that operate concurrently. This enables the system to process searches, availability checks, and general user interactions in parallel, significantly improving throughput. A routing mechanism or load balancer distributes incoming requests across these database instances to avoid overload on any single node.

Trade-offs and Challenges

- Requires careful request routing to prevent uneven load distribution.
- Increases complexity in maintaining data consistency across distributed database instances.
- Debugging and monitoring are more challenging due to distributed query execution.
- Synchronization overhead grows as the number of instances increases.

2. Data Partitioning / Sharding (Functional Partitioning by Region)

ColombianStay implements functional data sharding based on geographical regions such as the Caribbean, Andes, and Amazon areas. Each shard contains data related to its region, including listings, reservations, host profiles, and user interactions. Regional partitioning reduces the dataset size per shard, minimizes cross-region interference, and allows query operations to be executed in parallel. Search and availability queries are routed directly to the region-specific shard, improving query efficiency and reducing overall database load.

Trade-offs and Challenges

- Cross-shard queries increase latency and require additional coordination.
- Some regions may become hot spots, requiring rebalancing or further partitioning.
- Application routing logic becomes more complex to ensure correct shard selection.
- Fault tolerance mechanisms must handle failures affecting only one region without disrupting others.

3. Replication with Read/Write Separation (Primary + Replicas per Region on Separate Machines)

To optimize read-heavy workloads and enhance fault tolerance, each regional shard is deployed using a Primary–Replica architecture hosted on separate physical machines. The Primary node handles all write operations such as reservations, payments, and listing updates. Read replicas handle queries related to listings, pricing, reviews, and user browsing activity. Each physical machine hosting a replica can also scale internally through multiple compute nodes, further increasing parallelism for regional data access. Hosting primaries and replicas on different machines improves availability and ensures that regional subsystems remain operational even under node failures.

Trade-offs and Challenges

- Eventual consistency limitations may cause replicas to lag behind the primary.
- Failover mechanisms must be robust to handle primary node failures without data loss.
- Physical distribution across machines increases operational complexity and networking overhead.
- Reservation and booking workflows must be designed to avoid race conditions or overbooking.

References

[1] Corporate Finance Institute. (n.d.). Business model canvas examples. Retrieved from https://corporatefinanceinstitute.com/resources/management/business-model-canvas-examples/

[2] PostgreSQL Global Development Group. (n.d.). PostgreSQL: The world's most advanced open source database. Retrieved from https://www.postgresql.org/

[3] MongoDB, Inc. (n.d.). MongoDB — The application data platform. Retrieved from https://www.mongodb.com/

[4] Redis Ltd. (n.d.). Redis — In-memory data store & cache. Retrieved from https://redis.io/

[5] The Apache Software Foundation. (n.d.). Apache Kafka — Distributed event streaming platform. Retrieved from https://kafka.apache.org/

[6] Amazon Web Services. (n.d.). Amazon Simple Storage Service (S3). Retrieved from https://aws.amazon.com/s3/

[7] Elastic N.V. (n.d.). Elasticsearch — The official search & analytics engine. Retrieved from https://www.elastic.co/elasticsearch/

[8] Snowflake Inc. (n.d.). Snowflake — Data cloud. Retrieved from https://www.snowflake.com/

[9] Google Cloud. (n.d.). BigQuery — Serverless, highly scalable data warehouse. Retrieved from https://cloud.google.com/bigquery

[10] w3schools. (n.d.). MongoDB Tutorial. Retrieved from https://www.w3schools.com/mongodb

[11] w3schools. (n.d.). PostgreSQL Tutorial. Retrieved from https://www.w3schools.com/postgresql

[12] CodeWithHarry(Youtube). (n.d.). MongoDB Tutorial in 1 Hour (2024). Retrieved from https://www.youtube.com/watch?v=J6mDkcqU_ZE