UNIVERSITÀ DI PISA

Computer Engineering

Artificial Intelligence and Data Engineering

Large Scale and Multi-structured Databases

Project Documentation of *LargeB&B*

Authors:
Grillo Alessio - Luca Rotelli - Omar Tomàs Sfar

Academic Year: 2025-2026

Contents

Chapter 1

Introduction

LargeB&B is a web application that acts as an intermediary platform for booking accommodations. It enables travelers to search, compare, and book hotels, bed & breakfasts, vacation rentals, apartments, and other types of lodging.

At the same time, the platform provides property owners and managers with the opportunity to promote their accommodations to a global audience, monitor bookings, and access detailed analytics on the performance of their properties.

The main features of LargeB&B include:

- Advanced search and filtering of available rooms based on location and review ratings.
- The ability for users to write and read reviews to help other travelers make informed decisions.
- Real-time chat communication between customers and property managers, both before and after booking.
- Access to analytics and performance insights for property owners regarding the accommodations they have published.

Chapter 2

Design

2.1 Requirements

2.1.1 Functional requirements

2.1.2 Guest

- The system must allow guests to register on the platform.

2.1.3 Customer

- The system must allow customers to log in.
- The system must allow customers to log out.
- The system must allow customers to delete their account.
- The system must allow customers to book one or more rooms.
- The system must provide a messaging system to communicate with the property manager.
- The system must allow customers to publish their own reviews.
- The system must allow customers to modify their own reviews.
- The system must allow customers to delete their own reviews.
- The system must allow customers to add their payment method
- The system must allow customers to remove their payment method
- The system must allow customers to see their payment method
- The system must allow customers to cancel one or more existing bookings.
- The system must allow customers to modify one or more existing bookings.

- The system must allow customers to mark specific properties as favorites for future reference.
- The system must allow customers to remove specific properties as favorites.
- The system must allow customers to view their booking history.
- The system must notify the manager when a customer/manager sends a message through the messaging system.
- The system must allow customers to create a pending booking while awaiting payment confirmation.
- The system must allow customers to delete a pending booking.
- The system must notify the customer in case of:
  - booking confirmation or modification;
  - booking cancellation.

2.1.4 Customer and Guest

- The system must allow guests to search and view properties managed by hosts.
- The system must allow guests to view a map showing the geographical distribution of properties.
- The system must allow users to view reviews written by other users.
- The system must allow users to filter properties by category.
- The system must allow users to order properties based on star rating.
- The system must allow users to filter properties and rooms by price range.
- The system must allow users to filter rooms by available services.
- The system must provide a view showing rooms offering a specific service.
- The system must provide a view showing properties offering a specific service.
- The system must allow users to view photos related to properties.
- The system must allow users to view detailed room information.
- The system must display room availability in real time.
- The system must display a list of Points of Interest (POIs) located in the proximity of the property on the property detail page.

- The system must provide personalized recommendations by displaying properties booked by other users who also reserved the property currently being viewed.
- The system must display the number of concurrent users currently viewing the same room to the user.
- The system must display the top 10 most visited properties within the last hour.
- The system must display a list of recently viewed rooms to the user.
- The system must recommend alternative properties that share the highest number of amenities with the property currently being viewed.

2.1.5 Manager

- The system must allow the manager to add a property.
- The system must allow the manager to delete a property.
- The system must allow the manager to modify property information.
- The system must allow the manager to add a room.
- The system must allow the manager to delete a room.
- The system must allow the manager to modify room information.
- The system must allow the manager to view analytics by period and by property.
- The system must allow the manager to view properties information.
- The system must allow the manager to view analytics starting from the moment a property is added.
- The system must notify the manager with a notification in case of a new message from a customer
- The system must notify the manager with a notification in case of a new reservation from a customer
- The system must notify the manager with a notification if a reservation is removed by a customer.
- The system must notify the manager with a notification if a reservation is modified by a customer
- The system must allow the manager to view the payment status of rooms.
- The system must allow the manager to reply to user reviews.

- The system must allow the Manager to see all bookings for his properties
- The system must display the room occupancy rate within the analytics dashboard for a specific time period.
- The system must display the overall review rating aggregated over a selected time period.
- The system must show the total number of bookings made during a specific time period.
- The system must display the total count of reviews received over a selected time period.
- The system must identify and display the most frequently booked room type for a given time period.
- The system must calculate the average number of guests per room per booking for a specific time period.
- The system must display the total revenue generated over a selected time period.
- The system must list the reviews received during a specific time period.

Admin

- The system must allow the admin to delete any review.
- The system must allow the admin to delete any property.
- The system must allow the admin to delete any room.
- The system must allow the admin to ban any user.
- The system must allow the admin to view analytics of any property.
- The system must allow the admin to filter analytics by time period for any property.
- The system must allow the admin to view analytics starting from the moment a property is added.

2.1.6 Non-Functional Requirements

- Passwords of registered users must be securely encrypted.
- The system should be able to scale to accommodate an increase in user traffic.
- The system must provide notifications via application.

- The results produced by the system must be consistent.
- The system must ensure high data availability.
- The system must manage data storage using Document DB, Graph DB, and Key-Value DB.
- The DBMS technologies used must be Neo4j, MongoDB, and Redis.
- The business logic and data access layers must be implemented using the Spring framework in Java.
- The codebase must be easily maintainable, readable, and updatable.
- The system must use a document database to store information permanently.
- The system must use a key-value database to temporarily store information, manage cache, locks, and user sessions.
- The system must implement a stateless authentication mechanism using JSON Web Tokens (JWT). All protected API resources must require a valid Bearer token in the Authorization header to grant access.

2.1.7 Use Cases Diagram

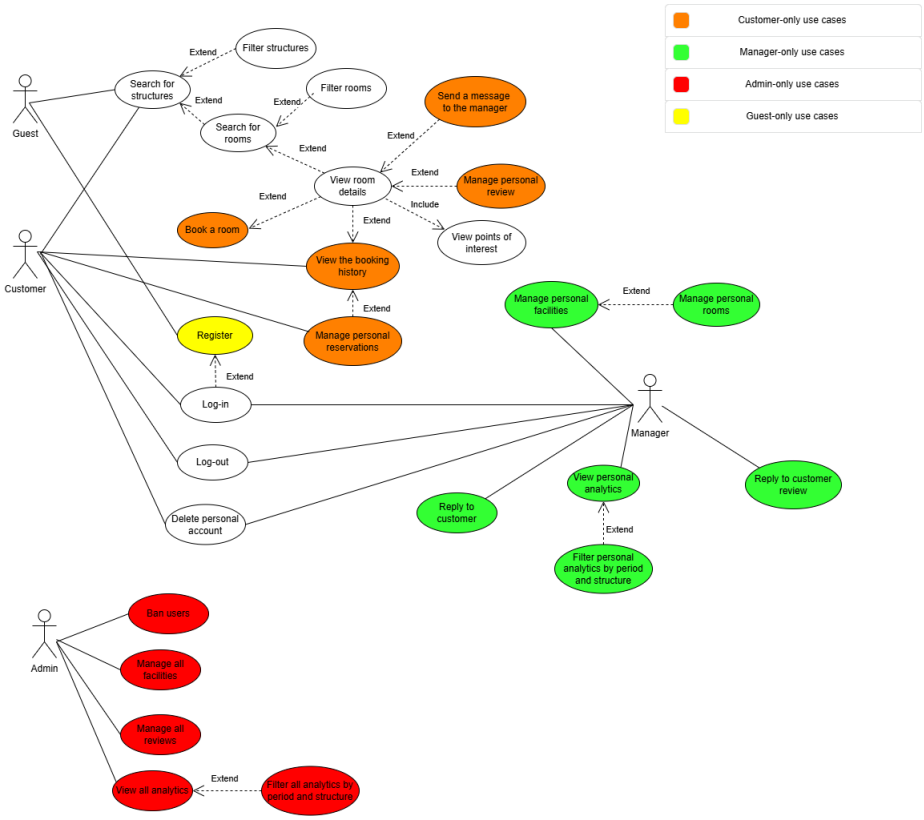

Figure 2.1: Use cases diagram

2.1.8 Analysis Class Diagram

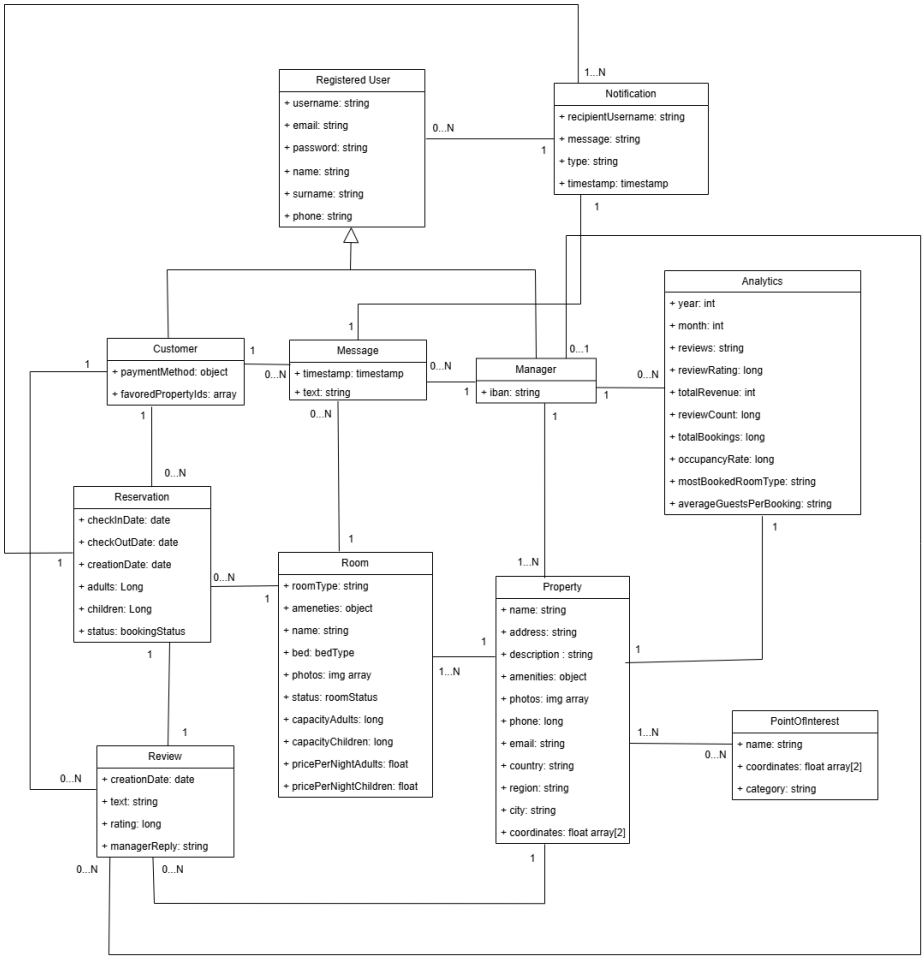

Figure 2.2: Analysis Class diagram

2.1.9 Mock-ups

2.1.10 Guest and customer view

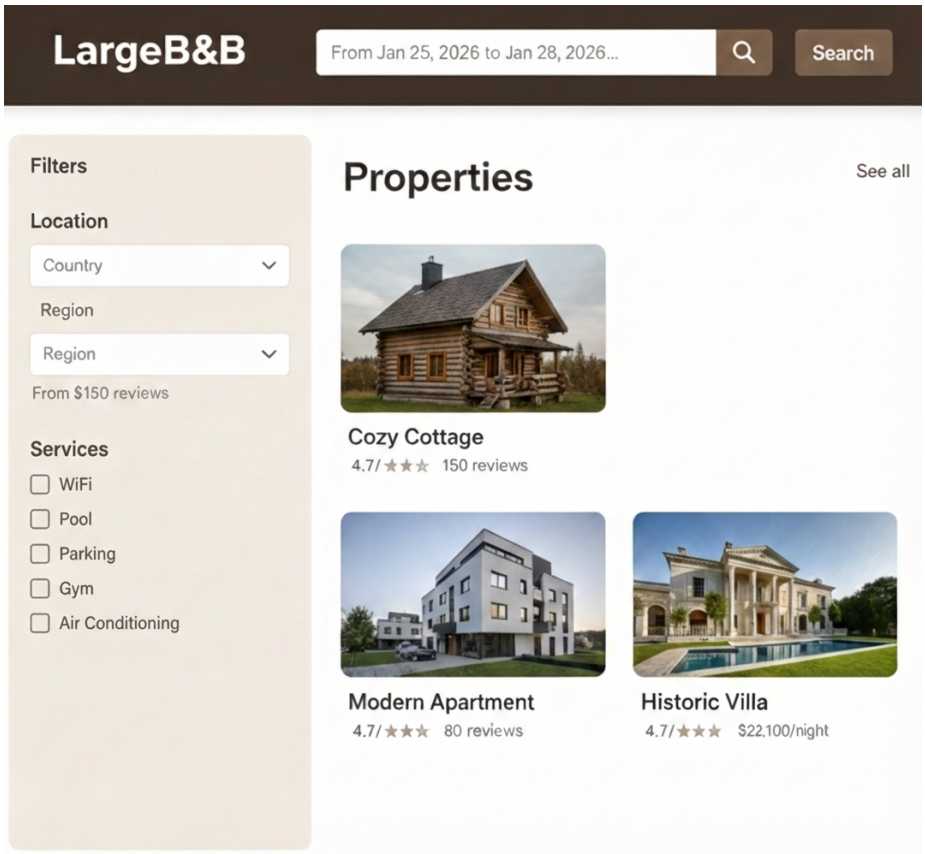

Figure 2.3: Showing properties view

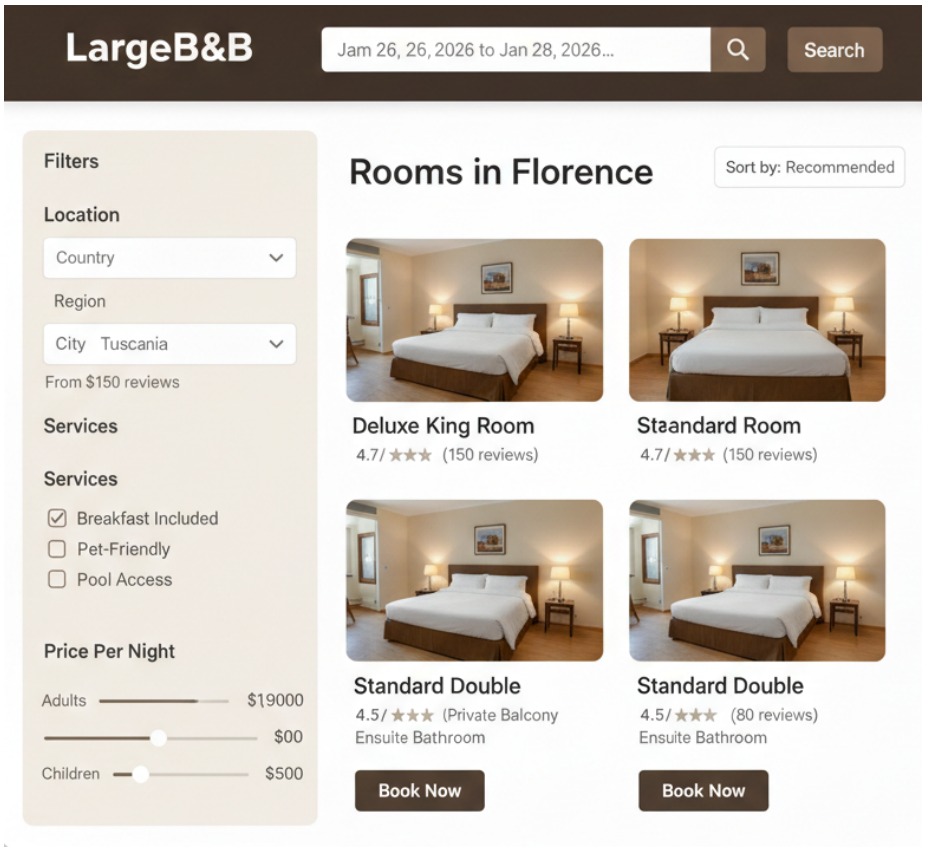

Figure 2.4: Showing rooms view

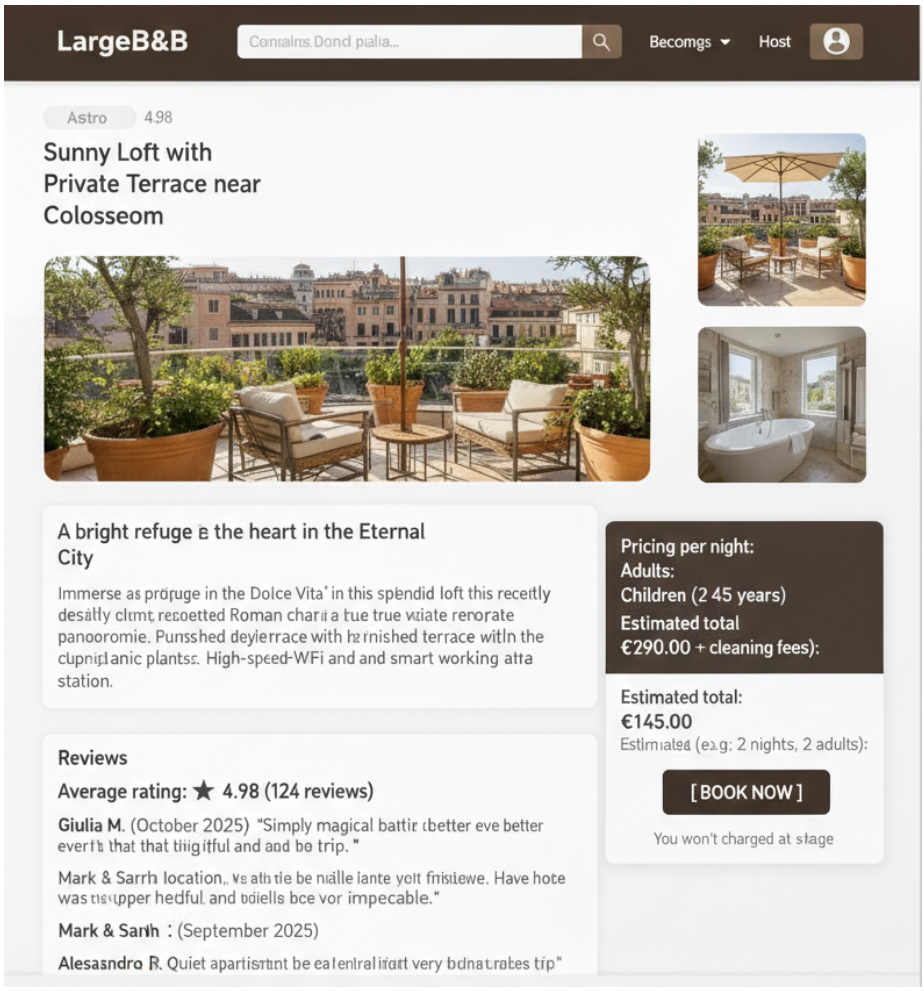

Figure 2.5: Room details view

Customer

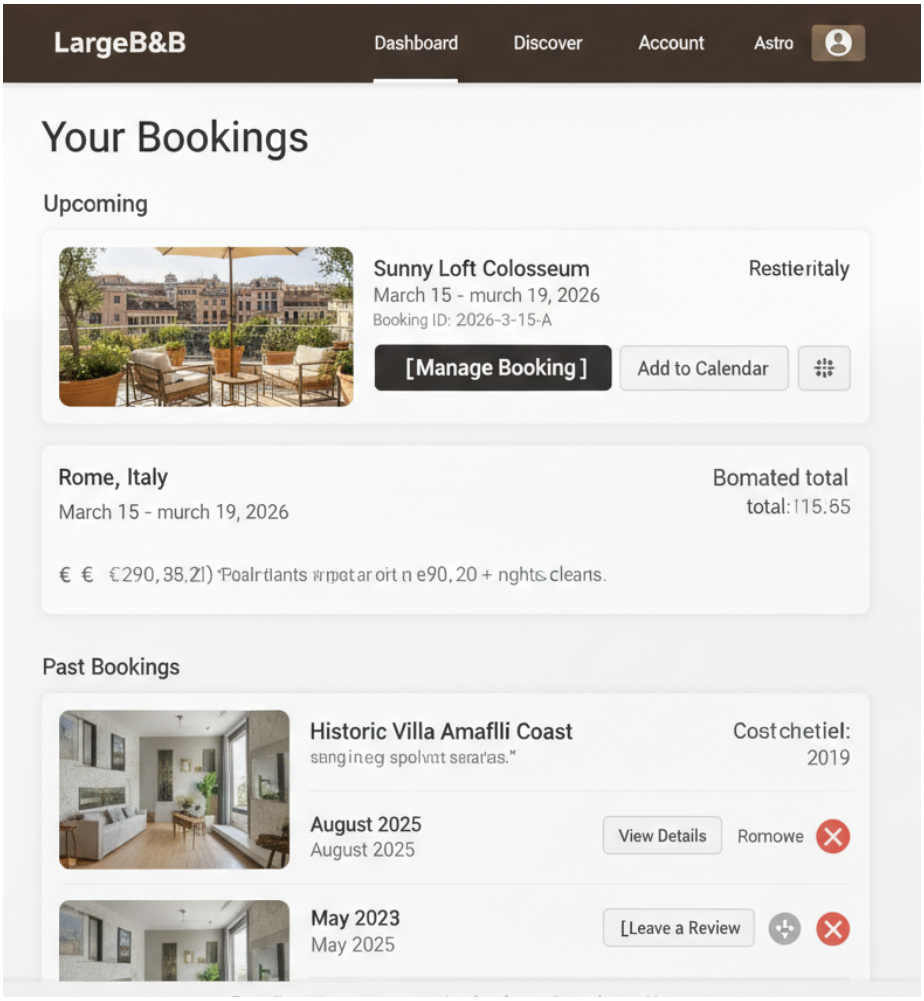

Figure 2.6: Customer bookings view

Manager view

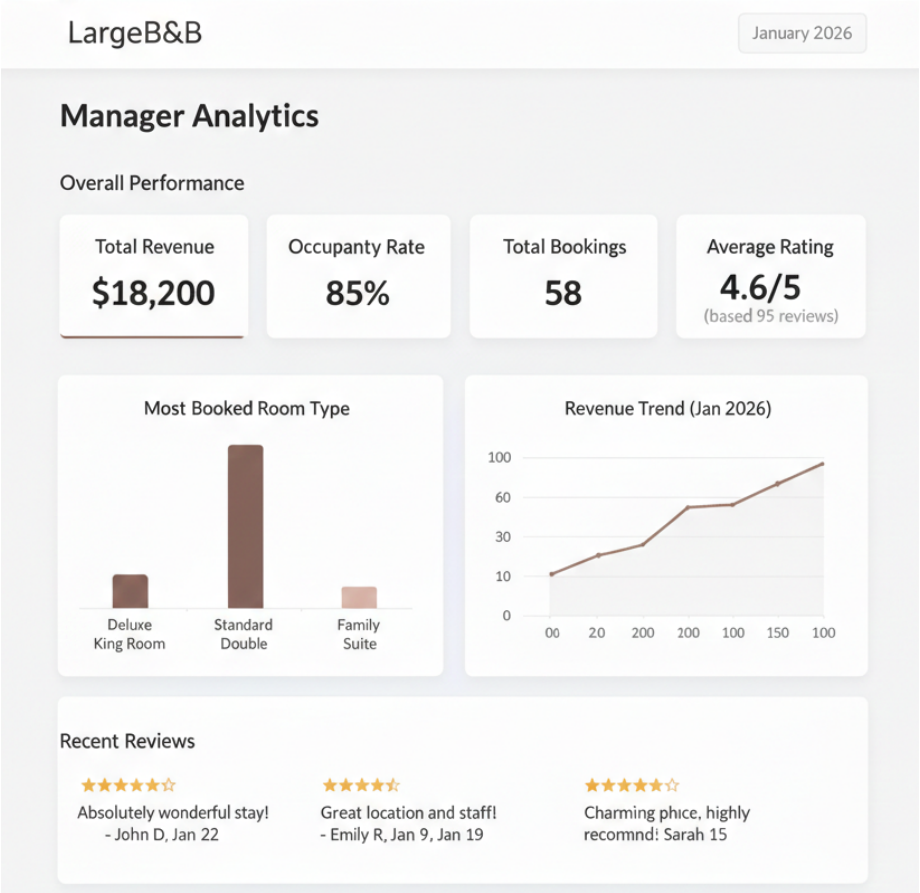

Figure 2.7: Analytics properties view

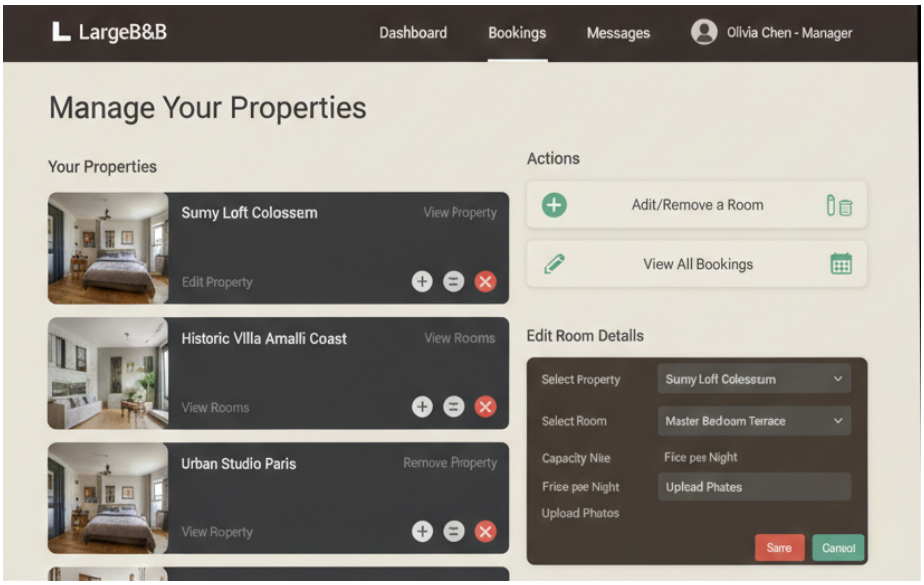

Figure 2.8: Manager properties view

2.2 CAP Theorem and Consistency Management

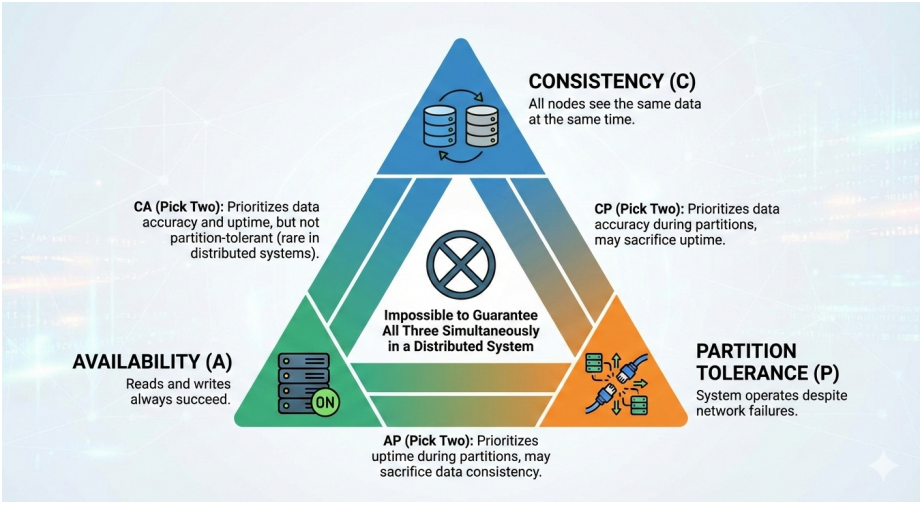

Figure 2.9: Cap Theorem

In the design of the distributed architecture for the B&B management platform, we addressed the trade-offs imposed by the **CAP Theorem** (Consistency, Availability, and Partition Tolerance). Given that network partitions (P) are unavoidable in a distributed environment involving microservices and cloud databases, the system adopts a hybrid strategy depending on the criticality of the specific business domain.

2.2.1 Reservation Service: Prioritizing Consistency (CP)

For the critical path of booking and payments, the system prioritizes **Consistency** over Availability. The business logic dictates that avoiding "double bookings" (two customers booking the same room for the same dates) is strictly more important than accepting a request during a partial system failure.

To achieve strong consistency:

- **Distributed Locking with Redis:** Before a transaction is finalized in the persistent database, a temporary lock (Time-To-Live of 15 minutes) is acquired on Redis. This acts as a pessimistic locking mechanism, ensuring that once a user selects a room for payment, no other nodes can write to that specific resource.
- **Atomic Writes in MongoDB:** The final confirmation is performed using MongoDB's atomic document updates. If the database primary node is unreachable (partition), the system rejects the write request rather than allowing a potential state conflict, effectively sacrificing Availability to preserve Data Integrity.

2.2.2 Search and Discovery: Prioritizing Availability (AP)

For the browsing, searching, and property listing modules, the system prioritizes **Availability**. It is acceptable for a user to see a room that was booked seconds ago as "available" in the search results, provided the error is caught at the checkout stage. It is not acceptable, however, for the homepage or search filters to be unresponsive.

To achieve high availability:

- **Caching Strategy:** Frequently accessed data (property details, amenities) are cached in Redis. If the primary database experiences high latency or downtime, the system serves data from the cache.
- **Read Replicas:** The system allows reading from secondary database replicas, accepting that there may be a slight replication lag (stale data) in exchange for system responsiveness.

2.2.3 Polyglot Persistence and Eventual Consistency

The system utilizes a polyglot persistence architecture, syncing data between the document store (MongoDB) and the graph database (Neo4j). This synchronization follows an **Eventual Consistency** model.

When a reservation is confirmed in MongoDB, the relationship is propagated to Neo4j asynchronously. While there is a minimal delay between the transaction confirmation and the update of the social graph, this does not impact the transactional integrity of the booking. This approach ensures that the Graph Service remains decoupled and does not block the critical path of the user experience.

¡¡¡¡¡¡¡ HEAD In conclusion, the system is **CP** regarding write operations for financial transactions and inventory management, while remaining **AP** for read-heavy operations such as search and recommendation generation.

2.3 Data Distribution and Replication Strategy

According to the non-functional requirements outlined in Section 2.1.6, our system must ensure **High Availability** and **Partition Tolerance** to guarantee low-latency responses to user requests, particularly for search and browsing operations.

In the context of the **CAP theorem**, LargeB&B is designed primarily as an **AP system** (Availability and Partition Tolerance), accepting *Eventual Consistency* for read operations to maximize throughput. However, to prevent critical issues such as double bookings, we enforce stronger consistency constraints on write operations through specific Write Concerns.

The fundamental aspect of our design relies on a hybrid approach combining **Replica Sets** for high availability and **Sharding** for horizontal scalability.

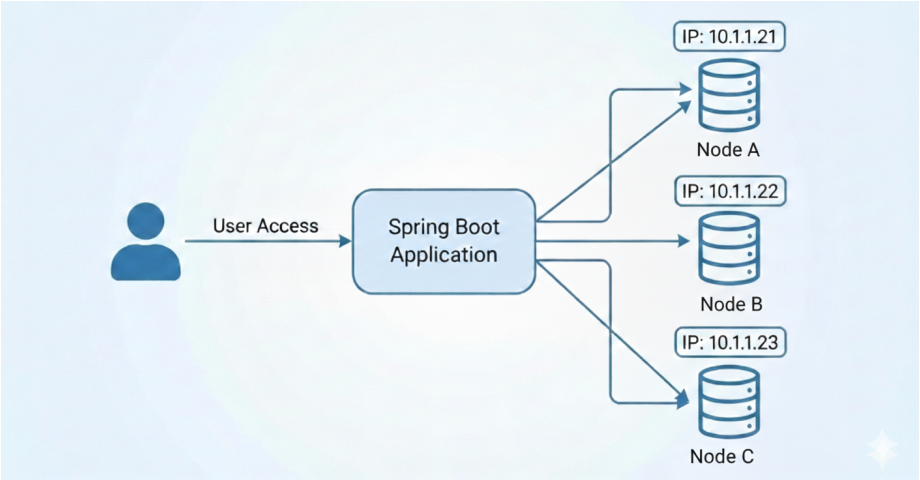

Figure 2.10: Database Architecture: Replica Set with 3 Nodes

2.3.1 MongoDB Replica Set Configuration

To ensure redundancy and automatic failover, the MongoDB architecture is deployed as a **Replica Set** consisting of three nodes. This configuration protects the system against single-node failures and network partitions.

The cluster is composed of one **Primary** node and two **Secondary** nodes. To optimize the election process and resource utilization, we have configured specific **priority levels** for each member:

- **Node A (IP 10.1.1.21 - Priority 5):** This is the preferred Primary node. It resides on the highest-performance hardware and handles all write operations under normal conditions.
- **Node B (IP 10.1.1.22 - Priority 2):** This node acts as the immediate hot-standby. It serves read operations and is the first candidate to become Primary if Node A fails, it also hosts the backend application.
- **Node C (IP 10.1.1.23 - Priority 1):** This node has the lowest priority. It is primarily used for read offloading and acts as a disaster recovery option. Assigning a lower priority ensures it is elected as Primary only as a last resort.

2.3.2 Shard Key and Partitioning Algorithm Selection

Since the _id field is mandatory in every document and is the primary filter for the majority of queries, it was selected as the shard key. This choice enables direct query routing, avoiding the overhead of scatter-gather operations. Consequently, we adopted Hashed Sharding as the partitioning algorithm to ensure uniform distribution.

2.3.3 Read and Write Policies

To balance the trade-off between latency and data integrity, we have configured specific policies for reading and writing data.

Read Operations (Read Preference: `nearest`)

Given that the read-to-write ratio in LargeB&B is estimated to be approximately 10:1 (browsing vs. booking), minimizing read latency is crucial. We utilize the `nearest` read preference. The driver routes the read request to the replica set member with the lowest network latency, regardless of whether it is Primary or Secondary.

This choice provides two main benefits:

1. **Load Balancing:** It distributes the heavy read traffic across all three nodes, preventing the Primary from being overwhelmed.
2. **Geo-latency Reduction:** If the infrastructure is geo-distributed in the future, users will automatically read from the data center closest to them.

Write Operations (Write Concern: `majority`)

Although we prioritize availability for reads, write operations—specifically *User Registration*, *Property Creation*, and critical *Reservation* flows—require strict durability. We have configured the Write Concern to `majority`.

- The system acknowledges a write only after it has been committed to the Primary and propagated to at least one Secondary (2 out of 3 nodes).
- This ensures that even if the Primary node crashes immediately after a write (e.g., after a confirmed payment), the data is safe on a Secondary node that will become the new Primary, preventing data rollback and financial inconsistencies.

2.3.4 Redis Configuration

Redis is utilized for caching, session management, and the temporary storage of reservation drafts.

Architecture and High Availability

Redis is deployed using a **Master-Slave** architecture:

- **Replication:** The Master (Node A) handles all writes. These are asynchronously propagated to the Slaves (Node B and C), which can serve read traffic.
- **Sentinel:** To ensure automatic failover, we deployed three **Redis Sentinel** instances. They monitor the cluster health and, if the Master fails, automatically promote one of the Slaves to be the new Master (in descending order of priority).

Sharding Strategy

Although deployed in a replicated setup, we utilize an algorithmic sharding approach for key distribution:

- **Shard Key:** The database key itself acts as the shard key.
- **Partitioning Algorithm:** Redis uses the **CRC16** checksum of the key modulo 16,384 to determine the hash slot. This ensures that keys are distributed deterministically.

Eviction and Persistence

To manage memory efficiently and ensure data safety:

- **Eviction Policy (`allkeys-lfu`):** We set a `maxmemory` limit of 512MB. When this limit is reached, the Least Frequently Used keys are evicted first. This protects the system from crashing during traffic spikes.
- **Persistence (RDB):** We enabled RDB snapshots (e.g., save every 5 minutes if 10 keys change) to allow disaster recovery in case of a total cluster reboot.

2.3.5 Neo4j Configuration

Deployment Strategy

Neo4j is deployed as a **Single Instance** on Node C (10.1.1.23).

- **Rationale:** Graph data is inherently highly connected, making it difficult to shard effectively without incurring massive performance penalties from cross-shard traversals. Additionally, the Community Edition of Neo4j does not support clustering.
- **Placement:** By isolating Neo4j on Node C and lowering that node's MongoDB priority, we ensure the graph database has dedicated computational resources for complex traversal algorithms (e.g., recommendation engines).

In conclusion, the system is **CP** regarding write operations for financial transactions and inventory management, while remaining **AP** for read-heavy operations such as search and recommendation generation.

Chapter 3

Dataset and Data Modelling

3.1 Dataset

The dataset was created using **Airbnb** to obtain public data (properties, rooms, etc.), and **Faker** to generate private data (user and manager information, reservations, etc.), its size is about 172MB. Some public data that were not available were also generated using Faker, while points of interest (POIs) were added manually.

3.1.1 Airbnb

Airbnb is an online platform that connects people looking for short-term (or long-term) accommodations with homeowners or private room owners who want to rent out extra space. To obtain public data from this platform, the provided files in *InsideAirbnb* were used: 'listings.csv', which contains information about the properties, and 'reviews.csv', which contains information about the reviews. Specifically, data from 150 properties were collected for several European capitals: Rome, Paris, London, Berlin, Madrid, Lisbon, Vienna, Athens, Budapest, Prague, Oslo, Copenhagen, and Stockholm, with 3 to 5 rooms and 50 reviews collected for each property.

3.1.2 Faker

Faker is a Python library that generates large amounts of fake but realistic data, such as names, addresses, emails, phone numbers, and more, to populate databases, test applications, and create realistic development environments without using real sensitive data. Using Faker, 150 managers were generated (one per property), along with between 10 and 20 customers per property and one reservation for each customer, and finally the notifications were generated.

3.1.3 Points of interest

The points of interest are 5 per property and were manually added, with their respective coordinates, and include museums, public parks, monuments, historic districts, castles, and so on.

3.1.4 Obtained Datasets

The datasets obtained using the previous script are:

- **customers.json 20.5MB** – contains all generated customers
- **managers.json 1MB** – contains all generated managers
- **pois.json 2.7MB** – contains famous places near the property
- **properties.json 19.8MB** – contains all properties with their information (address, city, coordinates, amenities, manager_id, etc.)
- **reservations.json 10MB** – contains all reservations linked to rooms and customers
- **reviews.json 75.3MB** – contains all reviews linked to properties
- **rooms.json 11.6MB** – contains all rooms linked to properties via `property_id`
- **messages.json 9.6MB** – contains the messages exchanged between customers and managers
- **notifications.json 22.3MB** – contains the notification about reservations and received messages

3.2 Data Modelling Strategy

Based on the Analysis Class Diagram and the non-functional requirements of scalability and read-performance, we adopted a **Polyglot Persistence** architecture. The data layer is designed following a *Query-Driven Approach*, characteristic of NoSQL systems, where data models are optimized for specific access patterns rather than purely for storage efficiency.

3.2.1 MongoDB Document Design

MongoDB serves as the primary operational database. To maximize performance for the most frequent query—retrieving property details for the booking page—we utilized the **Embedded Data Model**.

The "Properties" Collection and Embedding Strategy

Contrary to the normalized Relational Model (SQL), where "Rooms" and "Properties" would be stored in separate tables requiring costly JOIN operations, we decided to **embed** the room information directly within the Property document.

- **Why Embedding?** A property has a bounded number of rooms (1-to-Few relationship). Since room availability and prices are always accessed together with the property description, embedding them eliminates the need for disk I/O on a second collection, drastically reducing latency.
- **Subset Pattern for Reviews:** To optimize the initial load time, we applied the *Subset Pattern*. The 5 most recent reviews are embedded in the `Property` document for immediate display. The full history of reviews is stored in a separate `reviews` collection, linked via reference.

Below is the JSON structure of the final `properties` document:

```
{
  "_id": "bdd640fb-...",
  "name": "Cozy Flat in Rome",
  "location": { "city": "Rome", "geo": [12.49, 41.90] },
  "amenities": ["Wifi", "AC", "Kitchen"],

  // EMBEDDED ROOMS (1:Few Relationship)
  "rooms": [
      {
          "roomId": "r1",
          "name": "Master Bedroom",
          "price": 120.00,
          "capacity": { "adults": 2 }
      },
      {
          "roomId": "r2",
          "name": "Single Room",
          "price": 80.00
      }
  ],

  // SUBSET PATTERN (Top 5 Reviews)
  "latestReviews": [
      { "user": "Mario", "rating": 5, "text": "Great!" }
  ]
}
```

Other Collections

- **Users:** Stores both Customers and Managers. A unique index on the `email` field ensures identity integrity.
- **Reservations:** Uses the *Referencing Pattern*. Since the history of bookings grows indefinitely (Unbounded growth), they are stored in a separate collection referencing `userId` and `propertyId`.

3.2.2 ETL Process and Data Ingestion

The raw dataset obtained from scraping and generation (Section 3.1) was originally structured in a normalized format (JSON files for rooms separate from properties) and contained inconsistencies. To migrate this data into our document-oriented structure, we developed a custom **ETL (Extract, Transform, Load) script in Python**.

The Python script performs the following critical operations before insertion:

1. **Data Cleaning:** The script parses the raw JSON files, correcting malformed fields (e.g., the `amenities` array which contained nested escaped strings) and normalizing date formats to MongoDB `ISODate`.
2. **Denormalization (Merging):** It reads the `rooms.json` file into memory and injects each room object into the corresponding parent object in `properties.json`, effectively transforming the 1:N relationship into an embedded array.
3. **Deduplication:** During the ingestion of `customers.json` and `managers.json`, the script enforces uniqueness by checking against a hash set of email addresses, discarding duplicate entries to prevent key collisions in the database.
4. **Subset Calculation:** For each property, the script sorts the associated reviews by date and slices the top 5 to populate the `latestReviews` field, implementing the Subset Pattern at ingestion time.

This automated process ensures that the MongoDB database is populated with clean, structured, and query-ready documents, as opposed to a raw import of CSV/JSON tables.

3.2.3 Graph Data Modelling (Neo4j)

While MongoDB provides excellent performance for document retrieval and geospatial queries (via `2dsphere` indexes), it lacks the efficiency required to traverse complex, highly connected relationships in real-time. To implement the **Recommendation Engine** (Collaborative Filtering), we integrated Neo4j as a supplementary Graph Database.

Nodes and Relationships Strategy

Unlike the document model, we do not replicate the full dataset in Neo4j. We adopted a **Projection Strategy**, storing only the topological structure of the data required for graph algorithms:

- **Nodes:**
  - `(:User)` Contains only `userId` and minimal metadata (role).
  - `(:Property)` Contains only `propertyId` and `city`.
- **Relationships:**
  - `(:User)-[:BOOKED {date: ...}]->(:Property)` represents a confirmed reservation.

Note on Granularity: In MongoDB, reservations are linked to specific *Rooms*. However, for the recommendation logic ("Users who liked this Property also liked..."), the granularity of the "Room" is too fine. Therefore, during the graph ingestion, we aggregated room-level bookings up to the **Property** level to create meaningful connections between users and properties.

Graph Ingestion Script

We developed a specific Python script (`import_graph.py`) using the Neo4j Bolt driver to populate the graph. The script performs a critical transformation: it maps 'room_id' from the reservation files back to 'property_id' (using a hash map built from 'rooms.json') to correctly establish the `[:BOOKED]` relationship between Users and Properties.

3.2.4 Key-Value Data Modelling (Redis)

Redis is employed as a high-performance, in-memory data store to handle transient data and enforce consistency in high-concurrency scenarios. It complements MongoDB by offloading operations that require atomic locking or high-frequency updates.

Concurrency Control (The Booking Lock)

The most critical functional requirement is preventing **Overbooking** (two users booking the same room simultaneously). Since MongoDB's document locking can be performance-heavy under high load, we implemented a **Pessimistic Locking** mechanism on Redis:

- **Key Pattern:** `lock:room:{roomId}:{date}`
- **Mechanism:** When a user attempts to book, the backend executes a `SET ... NX EX 600` command (Set if Not Exists, with 10 minutes expiration).

- **Result:** If Redis returns `OK`, the room is temporarily reserved, and the transaction proceeds to MongoDB. If `NULL`, the user receives a "Room busy" error immediately, without impacting the primary database.

Real-Time Analytics features

Redis is also utilized for features that require high-write throughput, avoiding write-amplification on MongoDB:

- **Trending Properties:** A `Sorted Set` (`trending_properties`) is used to track property views in real-time. The `ZINCRBY` command increments the score of a property efficiently.
- **Session & Scarcity:** Transient counters (e.g., "5 people are viewing this page") are stored as simple keys with short TTL (Time-To-Live), ensuring the data is self-cleaning and does not permanently occupy storage.

Chapter 4

Implementation

The **LargeB&B** system has been implemented following all the requirements and design specifications listed in the previous chapters, utilizing modern and scalable technologies. The application is built using the Spring Boot framework, following the Controller-Service-Repository architectural pattern to organize the code structure and ensure separation of concerns.

4.1 Spring Boot Framework

The project has been developed using the Spring Boot framework for application initialization and to establish connections between the server and the databases. Spring Boot is a powerful tool that simplifies and accelerates the development of web applications and microservices through three main features: automatic configuration, adoption of a declarative approach to configuration, and the ability to create standalone applications.

The backend is built with **Spring Boot 3.2.0**, providing:

- **Dependency Injection**: Automatic management of component dependencies through the IoC (Inversion of Control) container
- **Auto-configuration**: Simplified configuration of database connections and services with minimal XML or annotation configuration
- **RESTful API**: Native support for building REST APIs with embedded Tomcat server
- **Security**: Integrated Spring Security for authentication and authorization management

4.1.1 Spring Security and JWT Authentication

A particularly useful feature leveraged from the Spring framework is the Spring Security token management system, which enables a more fluid and persistent

user login experience. The system handles registration and authentication via JWT (JSON Web Token), which must be included in the header of any subsequent request after successful login.

This security management is implemented using a filter chain, allowing certain requests to proceed without authorization checks when not needed (such as property browsing, registration, and login), while requiring the JWT token in the request header for protected operations. This implementation fulfills the security check and authorization management requirements specified in the non-functional requirements.

The JWT tokens are configured with a 1-hour expiration time and are signed using the HS512 algorithm, ensuring both security and stateless session management across the distributed system.

4.1.2 Polyglot Persistence Architecture

LargeB&B implements a **polyglot persistence** approach, utilizing three different database systems, each optimized for specific use cases:

- **MongoDB** (Port 27017): Document-oriented NoSQL database for operational data storage including users, properties, rooms, reservations, reviews, messages, and notifications
- **Neo4j** (Port 7687): Graph database specifically designed for collaborative filtering and recommendation system algorithms
- **Redis** (Port 6379): In-memory key-value store used for caching trending properties and high-performance data retrieval

This architectural choice allows each database to handle the workload it is best suited for, maximizing performance and scalability while maintaining data consistency across the platform.

4.2 Model Layer

The models for this application are divided into three main categories: MongoDB models, Neo4j models, and utility classes. The model layer defines the domain entities and their mapping to the respective databases, representing the core business logic following the MVC (Model-View-Controller) pattern.

4.2.1 MongoDB Models

MongoDB models use the `@Document` annotation to map Java classes to MongoDB collections. All models leverage Lombok annotations (`@Data`, `@NoArgsConstructor`, `@AllArgsConstructor`, `@Builder`) to reduce boilerplate code and improve maintainability.

User Models

The user hierarchy is implemented using object-oriented inheritance, with `RegisteredUser` serving as an abstract base class containing common attributes shared by all user types.

RegisteredUser Model (Abstract Base Class)

```
@Data
@NoArgsConstructor
@AllArgsConstructor
@Document(collection = "users")
public abstract class RegisteredUser {
    @Id
    private String id;

    private String username;
    private String email;
    private String password;
    private LocalDate birthdate;
    private String name;
    private String surname;
    private String phoneNumber;
    private String role; // "CUSTOMER" or "MANAGER"
}
```

This abstract class defines the common structure for all registered users in the system, storing essential information such as credentials, personal data, and role-based access control identifiers.

Customer Model

```
@Data
@EqualsAndHashCode(callSuper = true)
public class Customer extends RegisteredUser {
    private PaymentMethod paymentMethod;
    private Set<String> favoredPropertyIds;
}
```

The Customer class extends RegisteredUser with customer-specific attributes including payment method details and a set of favorite property IDs for quick access to preferred accommodations.

Manager Model

```
@Data
@EqualsAndHashCode(callSuper = true)
public class Manager extends RegisteredUser {
    private String iban;
    private String vatNumber;
    private BillingAddress billingAddress;
}
```

The Manager class is similarly structured but includes business-specific attributes such as IBAN for receiving payments, VAT number for tax purposes, and billing address information.

Property Model

The Property model demonstrates advanced MongoDB features including embedded documents, compound indexes for query optimization, text indexes for full-text search capabilities, and geospatial indexes for location-based queries.

```
@Data
@NoArgsConstructor
@AllArgsConstructor
@Document(collection = "properties")
@CompoundIndexes({
    @CompoundIndex(name = "location_idx",
                   def = "{'city': 1, 'region': 1, 'country': 1}"),
    @CompoundIndex(name = "room_capacity_idx",
                   def = "{'rooms.capacityAdults': 1,
                       'rooms.capacityChildren': 1}"),
    @CompoundIndex(name = "rating_sort_idx",
                   def = "{'ratingStats.value': -1}")
})
public class Property {
    @Id
    private String id;

    @TextIndexed(weight = 3)
    private String name;

    @TextIndexed
    private String description;

    @TextIndexed
    private String city;

    @Indexed
    private String managerId;

    @GeoSpatialIndexed(type = GeoSpatialIndexType.GEO_2DSPHERE)
    private List<Double> coordinates; // [longitude, latitude]

    private List<String> amenities;
    private List<String> photos;
    private String country;
    private String region;

    // Embedded documents
    private List<Room> rooms;
```

```
    private List<PointOfInterest> pois;
    private RatingStats ratingStats;
}
```

Key design decisions for the Property model:

- **Embedded Rooms**: Rooms are embedded within properties as a list rather than stored as separate documents. This design choice enables atomic updates and faster retrieval of complete property information in a single query.
- **Text Indexes**: Full-text search is enabled on property names (weight=3 for higher relevance), descriptions, and city names, allowing users to search properties using natural language queries.
- **Geospatial Index**: A 2dsphere index on coordinates supports proximity queries for "find properties near me" functionality, calculating distances using spherical geometry.
- **Compound Indexes**: Multiple compound indexes optimize common filter combinations such as location hierarchy (country/region/city), room capacity searches, and rating-based sorting.

Room Model

```
@Data
@NoArgsConstructor
@AllArgsConstructor
public class Room {
    @Field("roomId")
    private String id;

    @Indexed
    private String propertyId; // Reverse lookup to parent property

    @Indexed
    private String roomType; // Type category (e.g., "deluxe",
        "standard")

    @TextIndexed(weight = 2)
    private String name;

    @Indexed
    @Field("beds")
    private short numBeds; // Number of beds

    private List<String> amenities; // Room-specific amenities
    private List<String> photos;

```

```
    private String status; // Availability status

    private Long capacityAdults;
    private Long capacityChildren;

    private Float pricePerNightAdults;
    private Float pricePerNightChildren;
}
```

Rooms are designed as embedded documents within the Property model, storing detailed information about each accommodation unit including capacity constraints and pricing structure.

Reservation Model

The Reservation model includes critical compound indexes specifically designed for efficient availability checking operations.

```
@Data
@Builder
@NoArgsConstructor
@AllArgsConstructor
@Document(collection = "reservations")
@CompoundIndex(name = "room_availability_idx",
              def = "{'roomId': 1, 'dates.checkIn': 1, 'dates.checkOut':
                  1}")
public class Reservation {
    @Id
    private String id;

    private Integer adults;
    private Integer children;
    private String status;

    @Indexed
    private String userId;

    @Indexed
    private String roomId;

    private ReservationDates dates;

    private Double totalPrice; // Total price for the reservation

    @CreatedDate
    private LocalDateTime createdAt;

    @Data
    @Builder
```

```
    @NoArgsConstructor
    @AllArgsConstructor
    public static class ReservationDates {
        @JsonFormat(shape = JsonFormat.Shape.STRING, pattern =
            "yyyy-MM-dd")
        private LocalDate checkIn;

        @JsonFormat(shape = JsonFormat.Shape.STRING, pattern =
            "yyyy-MM-dd")
        private LocalDate checkOut;
    }
}
```

The compound index on `roomId`, `checkIn`, and `checkOut` is critical for efficiently querying room availability with the question: "Is Room X available between Date A and Date B?" This index prevents the need for full collection scans and ensures sub-millisecond response times for availability queries.

Review Model

```
@Data
@NoArgsConstructor
@AllArgsConstructor
@Document(collection = "reviews")
public class Review {
    @Id
    private String id;

    private String reservationId; // Links to specific reservation

    @Indexed
    private String propertyId;

    private String userId;
    private LocalDate creationDate;

    private String text; // Review content

    @Min(1)
    @Max(5)
    private Long rating; // Overall rating (1-5)

    // Detailed ratings
    @Min(1)
    @Max(5)
    private Double cleanliness;
    @Min(1)
    @Max(5)
```

```
    private Double communication;
    @Min(1)
    @Max(5)
    private Double location;
    @Min(1)
    @Max(5)
    private Double value;

    private String managerReply; // Manager's response to review
}
```

Reviews are stored in a separate collection rather than embedded in properties to enable flexible querying, pagination, and independent scaling. The indexed `propertyId` field allows efficient retrieval of all reviews for a specific property.

Message Model

```
@Data
@NoArgsConstructor
@AllArgsConstructor
@Document(collection = "messages")
public class Message {
    @Id
    private String id;

    @Indexed
    private String senderId;

    @Indexed
    private String receiverId;

    private String text;
    private LocalDateTime timestamp;
    private Boolean isRead;
}
```

The Message model supports the chat functionality between customers and property managers, with indexes on both sender and receiver IDs to optimize conversation retrieval.

Notification Model

```
@Data
@NoArgsConstructor
@AllArgsConstructor
@Document(collection = "notifications")
```

```
public class Notification {
    @Id
    private String id;

    @Indexed
    private String userId;

    private String type;
    private String message;
    private LocalDateTime timestamp;
    private Boolean isRead;
}
```

4.2.2 Neo4j Models

Neo4j models use the `@Node` annotation to define graph vertices and `@Relationship` annotations to define edges. The graph database is specifically optimized for social features and collaborative filtering algorithms.

User Node

```
@Node("User")
@Data
@NoArgsConstructor
@AllArgsConstructor
public class UserNode {
    @Id
    @GeneratedValue
    private Long graphId;

    private String userId;
    private String username;
    private String privacyStatus;
}
```

The UserNode represents users in the graph database with a minimal set of attributes necessary for relationship traversal and recommendation algorithms.

Property Node

```
@Node("Property")
@Data
@NoArgsConstructor
@AllArgsConstructor
public class PropertyNode {
    @Id
```

```
    @GeneratedValue
    private Long graphId;

    private String propertyId;
    private String name;
    private String city;
}
```

PropertyNode stores essential property information in the graph for efficient traversal during recommendation queries.

Booking Relationship

```
@RelationshipProperties
@Data
@NoArgsConstructor
@AllArgsConstructor
public class BookedRelationship {
    @Id
    @GeneratedValue
    private Long id;

    private LocalDate date;

    @TargetNode
    private PropertyNode property;
}
```

The BOOKED relationship connects users to properties they have reserved, storing the booking date as a relationship property. This graph structure enables powerful collaborative filtering queries such as: "Users who booked property X also frequently booked properties Y and Z."

4.2.3 Utility Classes

PaymentMethod

```
@Data
@NoArgsConstructor
@AllArgsConstructor
public class PaymentMethod {
    private String id;
    private String gatewayToken;
    private String cardType;
    private String last4Digits;
    private String expiryDate;
    private String cardHolderName;
```

```
}
```

The PaymentMethod class implements secure payment information storage using tokenization. Actual card numbers are never stored; instead, a gateway token is used to represent the payment method, adhering to PCI DSS compliance standards.

RatingStats

```
@Data
@NoArgsConstructor
@AllArgsConstructor
public class RatingStats {
    private Double cleanliness;
    private Double communication;
    private Double location;
    private Double value;
}
```

RatingStats aggregates multiple rating dimensions for properties, allowing users to evaluate accommodations across different quality criteria.

4.3 Repository Layer

The repository modules, extended by Spring Data classes, provide an abstraction layer for database operations, significantly simplifying the implementation of data access logic. In our application, we utilize repositories for MongoDB, Neo4j, and Redis, employing both simple queries for basic CRUD operations and more complex queries for analytics and recommendation functionalities. The advanced queries will be explored in greater depth in subsequent sections.

4.3.1 MongoDB Repositories

Spring Data MongoDB provides automatic query generation based on method naming conventions, eliminating the need for manual query implementation in most cases. The framework parses method names and generates the appropriate MongoDB queries automatically.

Our MongoDB repositories handle operations for users, properties, reservations, reviews, messages, and notifications. For instance, queries like `findByEmail`, `findByManagerId`, or `findByStatus` are automatically implemented by Spring Data based on their method signatures.

For more complex operations that cannot be expressed through method naming, custom queries are defined using the `@Query` annotation with MongoDB query syntax. An example is the `findConflictingReservations` method, which checks for booking conflicts using date range comparisons with `$or` and `$lte`/`$gte` operators.

4.3.2 Neo4j Repositories

Neo4j repositories leverage Cypher queries for graph traversals and pattern matching. These repositories are essential for implementing the recommendation system, allowing complex multi-hop graph queries to discover relationships between users and properties.

The repositories support operations such as finding all properties booked by a specific user, identifying users with similar booking patterns, and calculating collaborative filtering scores based on shared preferences. Custom Cypher queries are defined using the `@Query` annotation to express complex graph traversal logic.

4.3.3 Redis Operations

Redis operations are handled through `RedisTemplate`, providing high-performance caching and sorted set operations. The primary use case is maintaining a sorted set of trending properties based on view counts, enabling real-time ranking and retrieval of popular accommodations.

4.4 Service Layer

The service layer acts as an intermediary between the controller and repository layers, representing the core functionalities and main business logic implementations of the application. It encapsulates complex business rules, orchestrates interactions with multiple repositories, and ensures a clean separation of concerns. Additionally, the service modules enable better transaction management and error handling, making the system more robust and maintainable.

4.4.1 Authentication Services

LoginService

This service implements the main functionalities regarding user authentication. It verifies user credentials against the database, validates passwords using BCrypt hashing, and generates JWT tokens upon successful authentication. The JWT token contains the user's email and role information, which is used throughout the application for authorization decisions.

RegistrationService

The RegistrationService manages user registration processes for both customers and managers. It validates that email addresses are unique, encrypts passwords using BCrypt before storage, and creates user entries in MongoDB. During customer registration, it also initializes empty sets for favored properties. For managers, it additionally validates business information such as VAT numbers and IBAN codes.

4.4.2 PropertyService

This service implements all property-related operations including advanced search, filtering, and geospatial queries. It supports multi-criteria searches combining city filters, price ranges, amenity requirements, and room capacity constraints. The service uses MongoDB's aggregation framework and query builders to construct dynamic queries based on user-provided search parameters.

For geospatial functionality, the service implements proximity-based searches using MongoDB's `$nearSphere` operator, which calculates distances on a spherical surface and returns properties within a specified radius. Each property view is also tracked by incrementing a counter in Redis, contributing to the trending properties ranking.

4.4.3 ReservationService

The ReservationService manages the complete booking lifecycle including availability checking, price calculation, payment processing, and reservation creation. It orchestrates operations across multiple data sources to ensure transactional consistency.

The availability checking mechanism queries existing reservations to identify date conflicts using the optimized compound index on room ID and dates. Price calculations account for the number of adults, children, and night count, applying per-person pricing models. Payment processing integrates with the payment gateway using tokenized payment methods, ensuring PCI compliance. Upon successful payment, the service creates a confirmed reservation record and updates all relevant statistics.

4.4.4 ReviewService

This service manages CRUD operations for property reviews. It allows customers who have completed stays to post reviews with ratings across multiple dimensions (cleanliness, communication, location, value). Upon review submission, the service automatically recalculates and updates the property's aggregate rating statistics, ensuring that displayed ratings always reflect the latest feedback.

4.4.5 MessageService

The MessageService facilitates real-time messaging between customers and property managers. It supports sending messages, retrieving conversation histories, and marking messages as read. The service implements privacy controls to ensure users can only access their own conversations.

4.4.6 NotificationService

This service manages the notification system, creating notifications for important events such as booking confirmations, review responses, and message re-

ceipts. It provides endpoints for retrieving unread notifications and marking them as read, supporting the real-time notification requirements.

4.4.7 AnalyticsService

The AnalyticsService provides comprehensive business intelligence for property managers. It generates detailed analytics including:

- Revenue calculations aggregating all confirmed reservations for a property within a specified time period
- Occupancy rate analysis measuring the percentage of nights booked versus available across all rooms
- Reservation trends showing booking patterns over time
- Rating evolution tracking changes in review scores
- Comparative performance metrics benchmarking against similar properties

The service implements complex aggregation pipelines using MongoDB's aggregation framework to compute these metrics efficiently, even across large datasets.

4.4.8 RecommendationService

This service provides personalized recommendations using a hybrid approach combining collaborative filtering and content-based filtering algorithms.

The collaborative filtering component uses Neo4j graph traversals to identify users with similar booking histories and recommend properties those similar users have booked. It employs Cypher queries to find multi-hop paths in the user-property booking graph, calculating similarity scores based on overlapping bookings.

The content-based filtering component analyzes property attributes, particularly amenities, to recommend similar properties. It calculates similarity scores by counting common amenities and attributes, ranking properties by their similarity to ones the user has previously booked or favorited.

The hybrid approach combines results from both algorithms, providing diverse and relevant recommendations that account for both social proof (what similar users like) and property characteristics (what matches the user's preferences).

4.4.9 FavoredPropertyService

This service manages users' favorite property lists, allowing customers to add properties to their favorites for quick access and receive notifications about price changes or availability. It ensures that favorites are synchronized between MongoDB (for quick access) and potentially used in recommendation algorithms.

4.5 Controller Layer

Controllers expose RESTful API endpoints and handle HTTP request/response mapping, serving as the entry point for client requests. They validate incoming data, delegate business logic to appropriate services, and format responses according to REST conventions.

4.5.1 AuthController

This controller handles user authentication processes, mapping HTTP requests for registration and login endpoints. It validates user input during registration using Spring Validation annotations, delegates authentication logic to the AuthService, and returns JWT tokens upon successful login. The controller implements proper HTTP status codes (200 for successful login, 201 for registration, 401 for authentication failures).

4.5.2 PropertyController

The PropertyController manages all property-related operations accessible to end users. It provides endpoints for:

- Browsing properties with pagination support
- Searching properties with multiple filter criteria (city, price range, amenities, room capacity)
- Retrieving detailed property information including embedded rooms and points of interest
- Geospatial queries for finding nearby properties based on coordinates and radius
- Fetching trending properties based on view counts from Redis
- Text search using full-text indexes on property names and descriptions

All GET endpoints are publicly accessible without authentication, supporting the browse-before-booking user flow.

4.5.3 ManagerController

This controller provides protected endpoints for property management operations, restricted to users with the MANAGER role. It handles:

- Creating new property listings with validation of all required fields
- Updating existing property information including rooms, amenities, and photos
- Managing property availability and pricing

- Accessing analytics dashboards with revenue and occupancy metrics
- Viewing and responding to customer reviews
- Managing booking requests and reservations

All requests require a valid JWT token in the Authorization header, and the controller verifies that managers can only modify their own properties.

4.5.4 ReservationController

This controller maps reservation-related operations for customers, including creating new bookings, viewing reservation history, modifying existing reservations, and canceling bookings. It enforces business rules such as cancellation policies and ensures customers can only access their own reservation data.

4.5.5 ReviewController

The ReviewController handles review submission, retrieval, and deletion. It allows authenticated customers who have completed stays to post reviews, enforces one-review-per-stay policies, and provides endpoints for browsing reviews with pagination and filtering by rating.

4.5.6 MessageController

This controller manages the messaging system, providing endpoints for sending messages between customers and managers, retrieving conversation threads, and marking messages as read. It implements real-time updates using polling or WebSocket connections for instant message delivery.

4.5.7 RecommendationController

This module relies on the RecommendationService to map users' requests concerning recommendation functionalities. It provides personalized property suggestions based on user history, similar user preferences, and property attributes. The controller supports both authenticated recommendations (using user history) and anonymous recommendations (based on current search context).

4.5.8 NotificationController

The NotificationController manages notification delivery, providing endpoints for retrieving unread notifications, marking notifications as read, and configuring notification preferences. It supports real-time notification updates for critical events.

4.6 Exception Handler

The application includes a Global Exception Handler implemented using the `@ControllerAdvice` annotation. By centralizing exception handling logic, the code becomes cleaner and more maintainable while ensuring consistent error responses across the entire application. This approach reduces the need for repetitive try-catch blocks in individual controllers and provides a unified error response format.

The exception handler catches various exception types including:

- `IllegalArgumentException`: For invalid input parameters, returning 400 Bad Request
- `SecurityException`: For unauthorized access attempts, returning 403 Forbidden
- `EntityNotFoundException`: For missing resources, returning 404 Not Found
- `IllegalStateException`: For business rule violations, returning 409 Conflict
- Generic `Exception`: For unexpected errors, returning 500 Internal Server Error with sanitized messages

All error responses follow a consistent structure including timestamp, status code, error message, and request path, facilitating debugging and client-side error handling.

4.7 API Documentation

The application uses SpringDoc OpenAPI (version 2.3.0) for automatic API documentation generation. The interactive documentation is accessible at `http://localhost:8080/swagger-ui.html` when the application is running.

The Swagger UI provides:

- Interactive endpoint testing with request/response examples
- Complete API schema documentation with all DTOs and models
- Authentication flow demonstration including JWT token usage
- Request parameter descriptions and constraints
- Response status code explanations
- Example payloads for all endpoints

This auto-generated documentation remains synchronized with the codebase, eliminating the risk of outdated API documentation and significantly improving developer experience for frontend integration.

4.8 Data Population Scripts

The project includes comprehensive Python scripts for generating realistic datasets and populating all three databases with test data, ensuring the application can be thoroughly tested with production-like data volumes.

4.8.1 Dataset Generation

The `createDataset.py` script generates synthetic data using the Faker library combined with real Airbnb data sourced from InsideAirbnb open datasets. This hybrid approach ensures data realism while maintaining privacy.

The script generates:

- 30,000 customer accounts with realistic names, emails, and payment methods
- 1,000 manager accounts with business information
- 2,000 property listings with actual geographic coordinates and amenities from real Airbnb data
- 100,000 reviews with varying ratings and realistic text comments
- 30,000 reservations spanning past and future dates
- Message conversations and notification records

Data generation maintains referential integrity, ensuring all foreign key relationships are valid and realistic patterns emerge (e.g., popular properties have more reviews, users favor properties they've booked).

4.8.2 MongoDB Population

The `populateMongoDB.py` script imports the generated JSON data into MongoDB with appropriate transformations:

- Converting date strings to proper DateTime objects
- Transforming coordinate arrays into GeoJSON Point geometries for spatial indexing
- Embedding room documents within property documents
- Creating all specified indexes including compound, text, and geospatial indexes
- Validating data integrity before insertion

The script uses batch insertion for performance, processing 1,000 documents at a time to minimize network round trips and maximize throughput.

4.8.3 Neo4j Population

The `populateNeo4j.py` script creates the graph structure in Neo4j:

- Creating User and Property nodes with unique constraints
- Establishing BOOKED relationships between users and properties based on reservation data
- Adding relationship properties such as booking dates
- Building indexes on node properties for query optimization

The script uses Cypher's `MERGE` operation to ensure idempotency, allowing it to be run multiple times without creating duplicate nodes. Batch processing with `UNWIND` statements significantly improves import performance for large datasets.

Chapter 5

Indexes

5.1 Database Indexing Strategy

To meet the non-functional requirements regarding performance and the specific functional requirements for filtering and searching, a targeted indexing strategy was designed for both MongoDB and Neo4j.

5.1.1 MongoDB Indexing Strategy

The indexing strategy is implemented directly via Spring Data MongoDB annotations on the entity classes. The following indexes have been defined to optimize specific query patterns required by the system.

Property Collection

The `properties` collection utilizes a combination of Compound, Text, and Geospatial indexes to handle complex filtering and search requirements.

- **Room Capacity Compound Index**
  - **Type:** Compound Index
  - **Fields:** `{ "rooms.capacityAdults": 1, "rooms.capacityChildren": 1 }`
  - **Justification:** Critical for the primary search functionality. It allows the database to efficiently filter properties that contain at least one room capable of accommodating the requested number of adults and children.

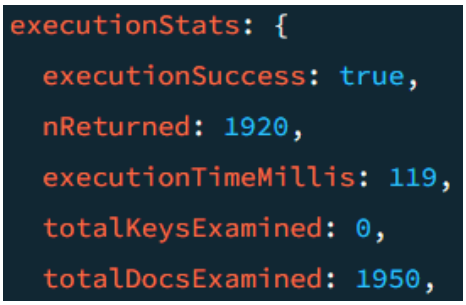

Figure 5.1: Query results without index

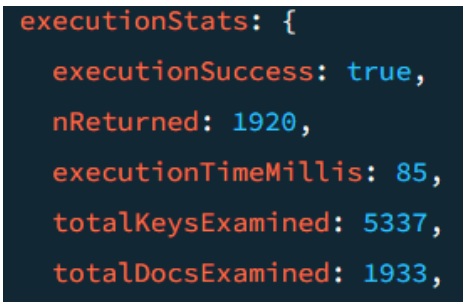

Figure 5.2: Query results with index

- **Rating Sort Index**
  - **Fields:** { `"ratingStats.value": -1` }
  - **Justification:** Optimizes the "Sort by Highest Rated" feature, ensuring that high-rated properties are retrieved efficiently without performing expensive in-memory sorting.

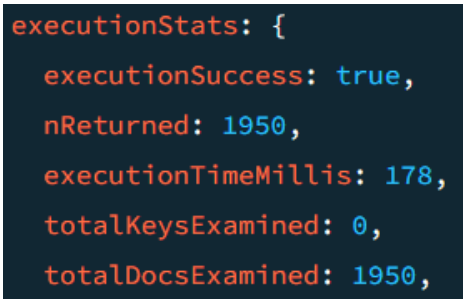

Figure 5.3: Query results without index

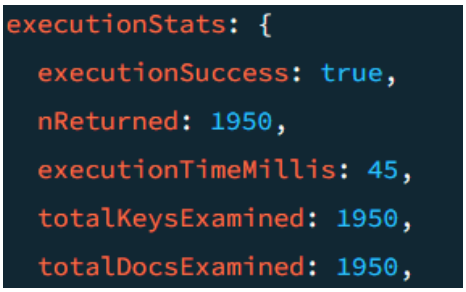

Figure 5.4: Query results with index

- **Text Search Index**
  - **Type:** Text Index
  - **Fields:** `name` (property) (weight: 3), `name` (room) (weight: 2), `description` (weight: 1), `city` (weight: 1)
  - **Justification:** Enables full-text search capabilities with relevance scoring. Matches in the property `name` are prioritized over matches in the `description`, improving search relevance for users.

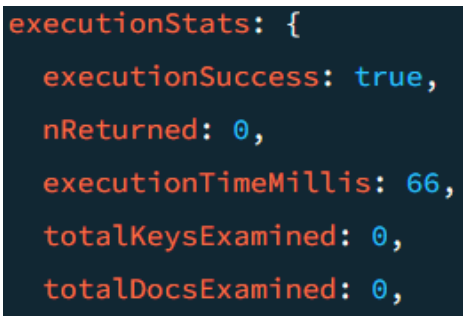

Figure 5.5: Query results with index

- **Geospatial Index**
  - **Type:** 2dsphere
  - **Fields:** `coordinates`
  - **Justification:** Required to support geospatial queries such as `$near` and `$geoWithin`, satisfying the requirement to view properties on a map or find POIs nearby.

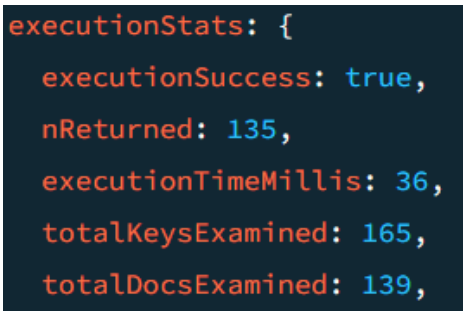

Figure 5.6: Query results with index

- **Manager Ownership Index**
  - **Type:** Single Field Index
  - **Fields:** `managerId`
  - **Justification:** Ensures fast retrieval of the specific subset of properties owned by a logged-in manager for the dashboard view.

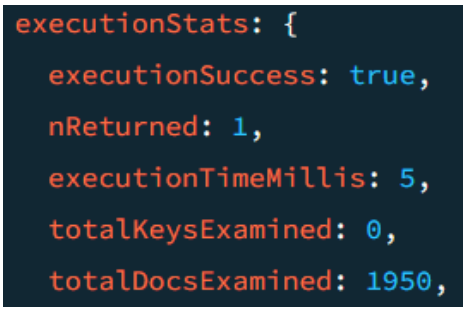

Figure 5.7: Query results without index

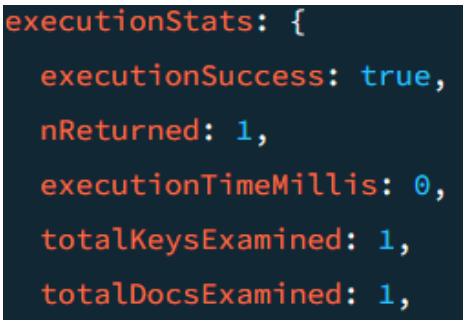

Figure 5.8: Query results with index

Room Embeddings (Indexed on Property)

Although rooms are embedded within the Property document, specific fields are indexed to allow filtering properties based on room attributes.

- **Room Price Compound Index**
  - **Type:** Compound Multikey Index
  - **Fields:** `rooms.pricePerNightAdults` (Asc), `rooms.pricePerNightChildren` (Asc)
  - **Justification:** Optimizes complex budget queries where users filter by both adult and child prices simultaneously. This is more efficient than separate indexes because it allows the database to narrow down candidate rooms satisfying both price constraints in a single index scan.

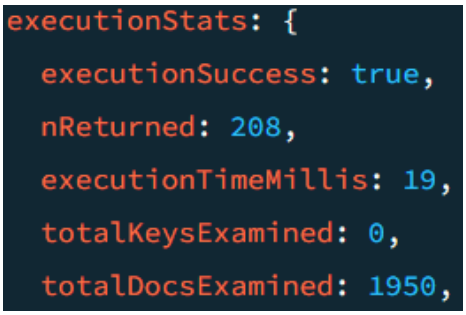

Figure 5.9: Query results without index

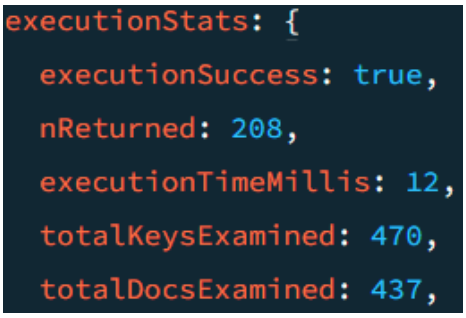

Figure 5.10: Query results with index

- **Room Type Index**
  - **Fields:** `rooms.roomType`

– **Justification:** Optimizes searches where users specify a required room configuration (e.g., filtering for "Single", "Double", or "Suite").

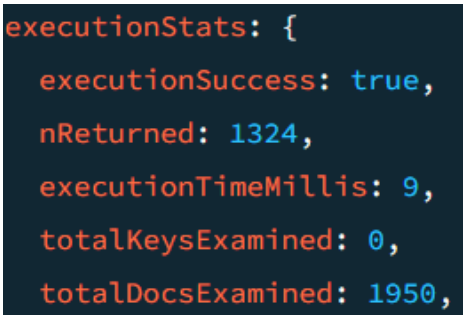

Figure 5.11: Query results without index

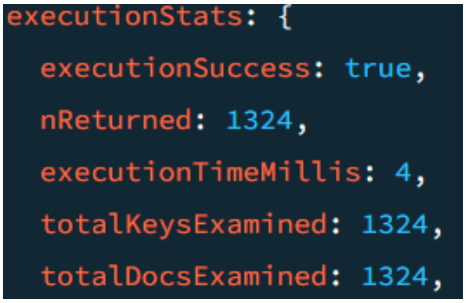

Figure 5.12: Query results with index

- **Room Beds Index**
  - **Fields:** `rooms.beds`
  - **Justification:** Accelerates queries where users filter by the specific number of beds required (e.g., distinguishing between a double bed room and a twin room with 2 separate beds).

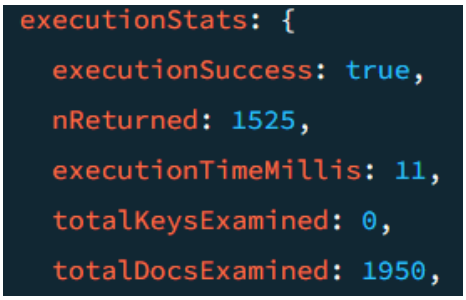

Figure 5.13: Query results without index

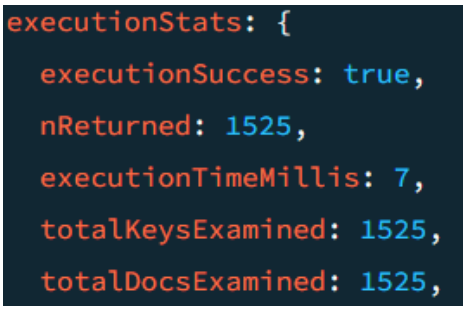

Figure 5.14: Query results with index

Reservation Collection

The `reservations` collection is indexed to support availability checks and user history retrieval.

- **Availability Check Compound Index**
  - **Type:** Compound Index
  - **Fields:** `{ "roomId": 1, "dates.checkIn": 1, "dates.checkOut": 1 }`
  - **Justification:** Critical for data integrity and performance. It allows the system to efficiently check if a specific room is occupied during a given date range before confirming a new booking.

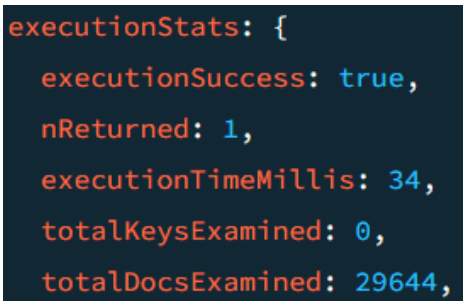

Figure 5.15: Query results without index

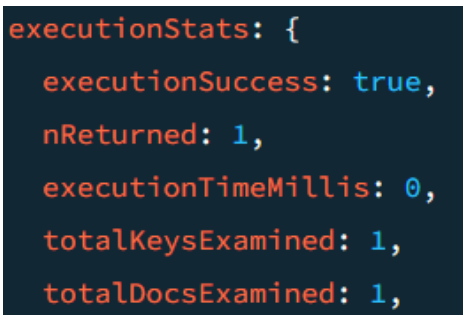

Figure 5.16: Query results with index

- **User History Index**
  - **Type:** Single Field Index
  - **Fields:** `userId`
  - **Justification:** Optimizes the retrieval of booking history for the "My Bookings" page.

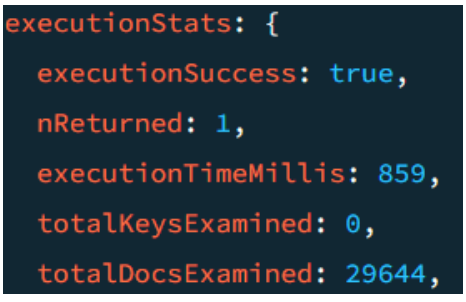

Figure 5.17: Query results without index

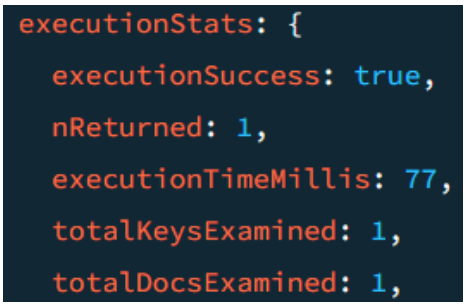

Figure 5.18: Query results with index

- **Room Lookup Index**
  - **Type:** Single Field Index
  - **Fields:** `roomId`
  - **Justification:** Allows the system (and Managers) to quickly retrieve all reservations associated with a specific room.

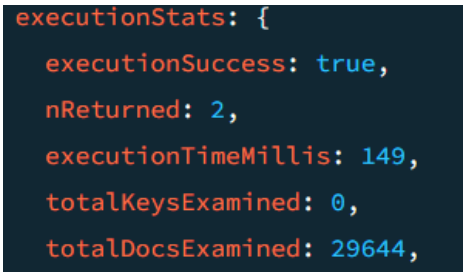

Figure 5.19: Query results without index

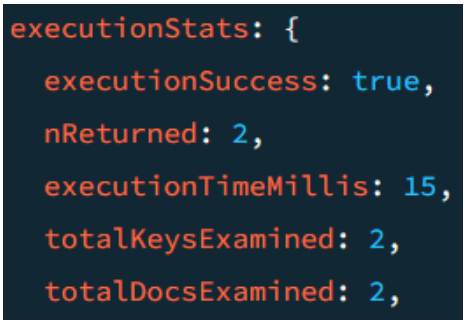

Figure 5.20: Query results with index

5.1.2 Neo4j Indexing Strategy

Neo4j is utilized exclusively for recommendation algorithms (Content-Based and Collaborative Filtering). The indexing strategy focuses on ensuring Data Integrity (Constraints) and optimizing the retrieval of "Anchor Nodes" to start graph traversals.

- **Property Identity Constraint**
  - **Constraint:** `Property(id) IS UNIQUE`.
  - **Requirements Satisfied:**
    - *"The system shall recommend alternative properties that share the highest number of amenities..."*
    - *"The system must provide personalized recommendations by displaying properties booked by other users..."*
  - **Justification:** Both recommendation algorithms begin with a specific property currently being viewed (the "Anchor Node"). This constraint ensures $O(1)$ lookup time to locate this starting point in the graph immediately.

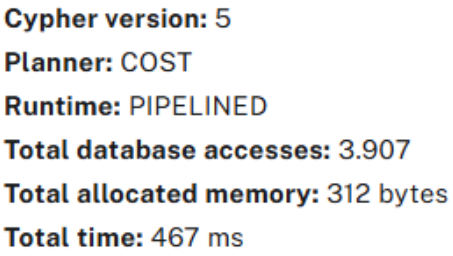

Figure 5.21: Query results without index

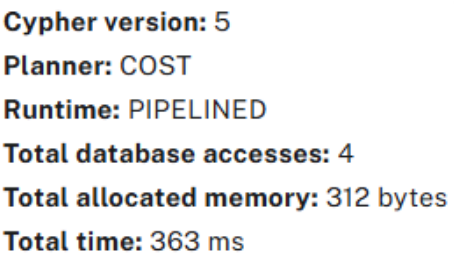

Figure 5.22: Query results with index

- **User Identity Constraint**
  - **Constraint:** `User(id) IS UNIQUE`.

- **Requirements Satisfied:**
  - *"The system must provide personalized recommendations by displaying properties booked by other users..."*
- **Justification:** Essential for the Collaborative Filtering algorithm. It ensures that bookings are linked to unique User nodes, allowing the traversal `(Property A)<-[:BOOKED]-(User)-[:BOOKED]->(Property B)` to function correctly.

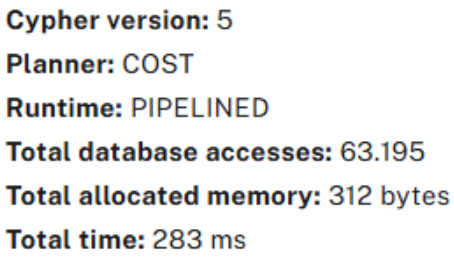

Figure 5.23: Query results without index

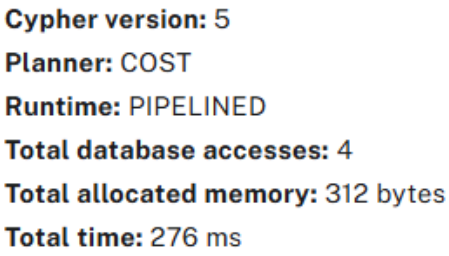

Figure 5.24: Query results with index

- **Amenity Identity Constraint**
  - **Constraint:** `Amenity(name) IS UNIQUE`.
  - **Requirements Satisfied:**
    - *"The system shall recommend alternative properties that share the highest number of amenities..."*
  - **Justification:** Ensures that shared features (e.g., "WiFi", "Pool") are represented as single, unique nodes in the graph. This allows the traversal algorithm to "hop" from the current property to related properties via these shared amenity nodes efficiently.

Cypher version: 5
Planner: COST
Runtime: PIPELINED
Total database accesses: 6
Total allocated memory: 312 bytes
Total time: 286 ms

Figure 5.25: Query results without index

Cypher version: 5
Planner: COST
Runtime: PIPELINED
Total database accesses: 3
Total allocated memory: 312 bytes
Total time: 126 ms

Figure 5.26: Query results with index

Chapter 6

Most Relevant Queries

This chapter presents the most relevant queries implemented in the application, showcasing how each database technology is leveraged to fulfill specific functional requirements. The queries are organized by database system and demonstrate the query-driven design approach adopted throughout the project.

6.1 MongoDB Queries

MongoDB serves as the primary database for the application, handling the majority of data storage and retrieval operations. The queries are implemented using Spring Data MongoDB's `@Query` and `@Aggregation` annotations directly in repository interfaces, providing a clean declarative approach.

6.1.1 Property Search and Filtering

The property search functionality is one of the most critical features of the platform, allowing users to find accommodations based on multiple criteria.

Search by Name with Average Rating Calculation

The following query demonstrates searching properties by name while calculating their average rating on-the-fly using an aggregation pipeline:

```
@Aggregation(pipeline = {
    "{ '$match': { 'name': { $regex: ?0, $options: 'i' } } }",
    "{ '$addFields': { 'averageRating': { '$cond': { 'if': { '$eq':
        ['$ratingStats.count', 0] }, 'then': 0, 'else': { '$divide':
        ['$ratingStats.sum', '$ratingStats.count'] } } } } }",
    "{ '$project': { 'id': '$_id', 'name': 1, 'city': 1,
        'averageRating': 1 } }"
})
Slice<PropertySearchDTO> findByNameContaining(String name, Pageable
    pageable);
```

Key aspects of this aggregation:

- **$match stage**: Filters properties using case-insensitive regex (`$options: 'i'`).
- **$addFields stage**: Computes derived fields, using `$cond` to handle division by zero.
- **$project stage**: Shapes the output to only include relevant fields.
- **Pageable support**: Spring Data handles pagination automatically with `Slice<>`.

Find Property by Room ID

A query to find the parent property when only the room ID is known:

```
@Query("{ 'rooms.roomId': ?0 }")
Optional<Property> findByRoomsId(String roomId);
```

This leverages MongoDB's ability to query nested array elements using dot notation.

Advanced Property Search with Multiple Filters

The advanced search functionality allows optional filtering by city, amenities, and price range:

```
@Aggregation(pipeline = {
    "{ '$match': { $and: [ " +
    "   { $expr: { $or: [ { $eq: [?0, null] }, { $regexMatch: { input:
        '$city', regex: ?0, options: 'i' } } ] } }, " +
    "   { $expr: { $or: [ { $eq: [?3, null] }, { $eq: [?3, []] }, {
        $setIsSubset: [?3, '$amenities'] } ] } }, " +
    "   { $expr: { $or: [ " +
    "     { $and: [ { $eq: [?1, null] }, { $eq: [?2, null] } ] }, " +
    "     { $gt: [{ $size: { $filter: { input: '$rooms', as: 'room',
        cond: { $and: [ " +
    "       { $or: [ { $eq: [?1, null] }, { $gte:
        ['$$room.pricePerNightAdults', ?1] } ] }, " +
    "       { $or: [ { $eq: [?2, null] }, { $lte:
        ['$$room.pricePerNightAdults', ?2] } ] } " +
    "     ] } } } }, 0] } " +
    "   ] } } " +
    "] } }"
})
List<Property> searchPropertiesAdvanced(String city, Double minPrice,
    Double maxPrice, List<String> amenities);
```

This complex aggregation demonstrates:

- **$expr with $or**: Handles null parameters gracefully for optional filters.
- **$regexMatch**: Case-insensitive city search.
- **$setIsSubset**: Verifies all requested amenities are present.
- **$filter on embedded arrays**: Filters rooms by price range without $unwind.

Geospatial Query - Properties Near Location

Finding properties within a specific radius using MongoDB's geospatial capabilities:

```
@Aggregation(pipeline = {
    "{ '$geoNear': { " +
    "   'near': { 'type': 'Point', 'coordinates': [?1, ?0] }, " +
    "   'distanceField': 'distance', " +
    "   'maxDistance': ?2, " +
    "   'spherical': true " +
    "} }"
})
List<Property> findPropertiesNearLocation(double latitude, double
    longitude, double maxDistanceMeters);
```

Key aspects:

- **$geoNear**: Must be the first stage in the pipeline.
- **spherical: true**: Uses Earth's curvature for accurate distance calculation.
- **distanceField**: Automatically adds calculated distance to results.
- Requires a `2dsphere` index on the `coordinates` field.

Top Rated Properties

Retrieving properties sorted by average rating:

```
@Aggregation(pipeline = {
    "{ '$addFields': { 'averageRating': { '$cond': { 'if': { '$eq':
        ['$ratingStats.count', 0] }, 'then': 0, 'else': { '$divide':
        ['$ratingStats.sum', '$ratingStats.count'] } } } } }",
    "{ '$sort': { 'averageRating': -1 } }",
    "{ '$limit': ?0 }"
})
List<Property> findTopRated(int limit);
```

This query:

- Dynamically calculates average rating from stored sum and count.
- Handles division by zero using `$cond`.
- Allows flexible limit parameter for different use cases.

6.1.2 Reservation Management Queries

Availability Checking (Overlap Detection)

One of the most critical queries ensures that double-booking is prevented:

```
@Query("{ 'roomId': ?0, 'status': { $ne: 'cancelled' }, 'dates.checkIn':
    { $lt: ?2 }, 'dates.checkOut': { $gt: ?1 } }")
List<Reservation> findOverlappingReservations(
    String roomId, LocalDate newCheckIn, LocalDate newCheckOut);
```

This query implements the classic interval overlap detection algorithm:

- Two intervals [A, B] and [C, D] overlap if and only if A ¡ D AND C ¡ B.
- The `$ne: 'cancelled'` filter excludes cancelled reservations.
- Proper indexing on `(roomId, dates.checkIn)` ensures O(log n) execution.

Active Bookings Detection

To prevent account deletion while active bookings exist:

```
@Query(value = "{ 'userId': ?0, 'status': 'confirmed', 'dates.checkOut':
    { $gt: ?1 } }", exists = true)
boolean hasActiveBookings(String userId, LocalDate today);
```

The `exists = true` parameter makes this an existence check query, returning a boolean instead of retrieving documents.

6.1.3 Analytics Aggregation Pipelines

The manager analytics dashboard leverages complex aggregation pipelines for business intelligence.

Reservation Statistics by Status

```
@Aggregation(pipeline = {
    "{ '$match': { 'roomId': { $in: ?0 }, 'dates.checkIn': { $gte: ?1,
        $lte: ?2 } } }",
```

```
    "{ '$group': { '_id': null, 'totalReservations': { $sum: 1 },
        'confirmed': { $sum: { $cond: [{ $eq: ['$status', 'CONFIRMED']
        }, 1, 0] } }, 'cancelled': { $sum: { $cond: [{ $eq: ['$status',
        'CANCELLED'] }, 1, 0] } }, 'completed': { $sum: { $cond: [{ $eq:
        ['$status', 'COMPLETED'] }, 1, 0] } }, 'totalAdults': { $sum:
        '$adults' }, 'totalChildren': { $sum: '$children' } } }",
    "{ '$project': { '_id': 0 } }"
})
Map<String, Object> getReservationStatistics(
    List<String> roomIds, LocalDate startDate, LocalDate endDate);
```

This aggregation demonstrates:

- **$match with $in**: Filters reservations for multiple rooms efficiently.
- **$group with $cond**: Conditional counting for status-based metrics.
- **Date range filtering**: Uses `$gte` and `$lte` for inclusive date bounds.

Rating Evolution Over Time

Analyzing how property ratings change over time:

```
@Aggregation(pipeline = {
    "{ '$match': { 'propertyId': ?0, 'creationDate': { $gte: ?1, $lte:
        ?2 } } }",
    "{ '$addFields': { 'yearMonth': { $dateToString: { format: '%Y-%m',
        date: '$creationDate' } } } }",
    "{ '$group': { '_id': '$yearMonth', 'averageRating': { $avg:
        '$rating' }, 'reviewCount': { $sum: 1 } } }",
    "{ '$sort': { '_id': 1 } }",
    "{ '$project': { 'month': '$_id', 'averageRating': 1, 'reviewCount':
        1, '_id': 0 } }"
})
List<Map<String, Object>> getMonthlyRatingEvolution(
    String propertyId, LocalDate startDate, LocalDate endDate);
```

Key techniques:

- **$dateToString**: Formats dates for grouping by month.
- **$avg**: Computes average rating per time period.
- Chronological sorting enables trend visualization.

Trending Properties (Improving Ratings)

Finding properties with improving ratings (recent reviews ¿ historical average):

```
@Aggregation(pipeline = {
    "{ '$addFields': { " +
```

```
    "   'averageScore': { '$cond': { " +
    "      if: { '$gt': ['$ratingStats.count', 0] }, " +
    "      then: { '$divide': ['$ratingStats.sum', '$ratingStats.count']
        }, " +
    "      else: 0 " +
    "   } }, " +
    "   'recentReviews': { '$slice': [ " +
    "      { '$sortArray': { 'input': '$reviews', 'sortBy': {
        'timestamp': -1 } } }, " +
    "      5 ] } " +
    "} }",
    "{ '$addFields': { 'recentAverage': { '$avg': '$recentReviews.score'
        } } }",
    "{ '$match': { '$expr': { '$gt': ['$recentAverage', '$averageScore']
        } } }",
    "{ '$addFields': { 'scoreDifference': { '$subtract':
        ['$recentAverage', '$averageScore'] } } }",
    "{ '$sort': { 'scoreDifference': -1 } }",
    "{ '$limit': 10 }",
    "{ '$project': { '_id': 0, 'id': '$_id', 'name': 1,
        'scoreDifference': 1 } }"
})
List<TrendingPropertyDTO> findTopImprovingProperties();
```

Advanced techniques demonstrated:

- **$sortArray**: Sorts embedded array by timestamp without $unwind.
- **$slice**: Takes only the 5 most recent reviews.
- **$expr in $match**: Enables comparison between computed fields.
- **Multi-stage $addFields**: Builds up computed values incrementally.

Top Properties by Genre/Category (Optional Filter)

Finding top-rated properties with optional category filtering:

```
@Aggregation(pipeline = {
    "{ '$match': { '$expr': { '$or': [ { '$eq': [?0, null] }, { '$in':
        [?0, '$amenities'] } ] } } }",
    "{ '$addFields': { 'averageScore': { '$cond': { 'if': { '$gt':
        ['$ratingStats.count', 0] }, 'then': { '$divide':
        ['$ratingStats.sum', '$ratingStats.count'] }, 'else': 0 } } } }",
    "{ '$sort': { 'averageScore': -1 } }",
    "{ '$limit': 10 }",
    "{ '$project': { 'id': 1, 'name': 1, 'averageScore': 1 } }"
})
List<PropertyAverageDTO> findTop10Properties(String amenity);
```

This query handles optional parameters elegantly using `$expr` with `$or`.

6.1.4 Update Operations with @Query and @Update

Spring Data MongoDB supports atomic update operations using the combination of `@Query` and `@Update` annotations.

Update Username in Reviews

When a user changes their username, all their reviews must be updated:

```
@Query("{ 'reviews.username': ?0 }")
@Update("{ '$set': { 'reviews.$.username': ?1 } }")
void updateReviewsByUsername(String oldUsername, String newUsername);
```

The positional operator `$` identifies the matched array element for the update.

Remove User Reviews on Account Deletion

```
@Query("{ 'reviews.username': ?0 }")
@Update("{ '$pull': { 'reviews': { 'username': ?0 } } }")
void deleteReviewsByUsername(String username);
```

The `$pull` operator removes all matching elements from the embedded array in a single atomic operation.

Increment Rating Statistics

```
@Query("{ '_id': ?0 }")
@Update("{ '$inc': { 'ratingStats.count': 1, 'ratingStats.sum': ?1 } }")
void incrementRatingStats(String propertyId, int rating);
```

The `$inc` operator provides atomic increment operations, essential for concurrent rating submissions.

6.1.5 Most Controversial Properties

Finding properties with highest rating variance (most polarizing opinions):

```
@Aggregation(pipeline = {
    "{ '$addFields': { 'variance': { '$pow': [{ '$stdDevPop':
        '$reviews.rating' }, 2] } } }",
    "{ '$match': { 'variance': { $gt: 0 } } }",
    "{ '$sort': { 'variance': -1 } }",
    "{ '$limit': 10 }",
    "{ '$project': { '_id': 0, 'id': '$_id', 'name': 1, 'variance': 1 }
        }"
})
```

```
List<ControversialPropertyDTO> findMostControversialProperties();
```

This uses MongoDB's `$stdDevPop` (population standard deviation) function to measure rating dispersion.

6.1.6 Message Conversation Retrieval

The messaging system requires bidirectional message retrieval:

```
@Query(value = "{ $or: [ { 'senderId': ?0, 'recipientId': ?1 }, {
    'senderId': ?1, 'recipientId': ?0 } ] }",
      sort = "{ 'timestamp': 1 }")
List<Message> findConversation(String user1, String user2);
```

This query:

- Uses `$or` to match messages in both directions.
- Sorts by timestamp ascending to display conversation chronologically.
- Leverages compound index on `(senderId, recipientId)` for performance.

6.2 Neo4j Queries

Neo4j powers the recommendation engine through graph-based queries using Cypher, Neo4j's declarative query language.

6.2.1 Collaborative Filtering Recommendations

The primary recommendation algorithm identifies properties booked by users with similar preferences:

```
public List<PropertyResponseDTO> getCollaborativeRecommendations(
       String propertyId) {

   String cypherQuery = """
       MATCH (p:Property {propertyId: $propId})<-[:BOOKED]-(u:User)
            -[:BOOKED]->(other:Property)
       RETURN other.propertyId AS recommendedId,
              count(*) AS strength
       ORDER BY strength DESC
       LIMIT 5
   """;

   Collection<String> recommendedIds = neo4jClient.query(cypherQuery)
       .bind(propertyId).to("propId")
       .fetchAs(String.class)
```

```
        .mappedBy((typeSystem, record) ->
            record.get("recommendedId").asString())
        .all();

    return propertyRepository.findAllById(recommendedIds);
}
```

The Cypher query pattern works as follows:

- **Start node**: Finds the property being viewed.
- **First hop**: Traverses `BOOKED` relationships backward to find users who booked this property.
- **Second hop**: Traverses `BOOKED` relationships forward to find other properties these users booked.
- **Aggregation**: Groups by property and counts co-occurrences as "strength".
- **Ranking**: Orders by strength to surface the most relevant recommendations.

This implements the classic "users who booked X also booked Y" collaborative filtering pattern.

6.2.2 Graph Constraints for Data Integrity

The application creates constraints at startup to ensure graph integrity:

```
@EventListener(ApplicationReadyEvent.class)
public void createIndexesAndConstraints() {
    List<String> startupQueries = List.of(
        // Property uniqueness - O(1) lookup for recommendations
        "CREATE CONSTRAINT property_id_unique IF NOT EXISTS " +
        "FOR (p:Property) REQUIRE p.id IS UNIQUE",

        // User uniqueness - essential for collaborative filtering
        "CREATE CONSTRAINT user_id_unique IF NOT EXISTS " +
        "FOR (u:User) REQUIRE u.id IS UNIQUE",

        // Amenity uniqueness - ensures shared nodes for traversal
        "CREATE CONSTRAINT amenity_name_unique IF NOT EXISTS " +
        "FOR (a:Amenity) REQUIRE a.name IS UNIQUE"
    );

    startupQueries.forEach(query -> neo4jClient.query(query).run());
}
```

These constraints serve dual purposes:

- **Data integrity**: Prevents duplicate nodes.
- **Performance**: Creates implicit indexes for O(1) lookups.

6.3 Redis Queries and Operations

Redis provides in-memory data structures for caching, session management, and real-time features.

6.3.1 Token Blacklisting for Secure Logout

JWT tokens are stateless by design, but the application implements secure logout through Redis blacklisting:

```
public void blacklistToken(String token) {
    String cleanedToken = cleanToken(token);
    Claims claims = getClaims(cleanedToken);

    // Calculate remaining TTL
    long tokenExpiration = claims.getExpiration().getTime();
    long currentTime = System.currentTimeMillis();
    long ttl = tokenExpiration - currentTime;

    if (ttl > 0) {
        // Key format: "blacklist:<token>"
        redisTemplate.opsForValue()
            .set("blacklist:" + cleanedToken, "true",
                 ttl, TimeUnit.MILLISECONDS);
    }
}

private boolean isTokenBlacklisted(String token) {
    Boolean exists = redisTemplate.hasKey("blacklist:" + token);
    return exists != null && exists;
}
```

Key design decisions:

- **TTL-based expiration**: Blacklist entries automatically expire when the token would have naturally expired, preventing unbounded memory growth.
- **String data type**: Simple key-value structure for O(1) lookup.
- **Validation integration**: Token validation checks blacklist before accepting requests.

6.3.2 Pessimistic Locking for Reservations

To prevent double-booking during the payment window, Redis implements temporary locks:

```
private static final String REDIS_PREFIX = "temp_res:";

public ReservationResponseDTO initiateReservation(
        String token, ReservationRequestDTO request) {

    // Check MongoDB for confirmed bookings
    List<Reservation> confirmedOverlaps = reservationRepository
        .findOverlappingReservations(room.getId(),
            request.getCheckIn(), request.getCheckOut());

    if (!confirmedOverlaps.isEmpty()) {
        throw new IllegalStateException("Room already booked");
    }

    // Check Redis for temporary locks
    Set<String> keys = redisTemplate.keys(REDIS_PREFIX + "*");
    if (keys != null) {
        for (String key : keys) {
            Reservation pending = objectMapper.convertValue(
                redisTemplate.opsForValue().get(key),
                Reservation.class);

            if (pending != null &&
                pending.getRoomId().equals(room.getId()) &&
                isOverlapping(pending.getDates(),
                    request.getCheckIn(), request.getCheckOut())) {
                throw new IllegalStateException(
                    "Room being paid for. Try again in 15 mins.");
            }
        }
    }

    // Create temporary reservation with 15-minute TTL
    String tempId = UUID.randomUUID().toString();
    Reservation tempReservation = Reservation.builder()
        .id(tempId)
        .roomId(room.getId())
        .dates(new ReservationDates(
            request.getCheckIn(), request.getCheckOut()))
        .status("PENDING_PAYMENT")
        .build();

    redisTemplate.opsForValue()
        .set(REDIS_PREFIX + tempId, tempReservation,
             15, TimeUnit.MINUTES);
```

```
46
47      return mapToDTO(tempReservation);
48  }
```

This implements a pessimistic locking pattern:

- **Temporary reservation**: Stored in Redis with 15-minute TTL during payment.
- **Overlap checking**: Both MongoDB (confirmed) and Redis (pending) are checked.
- **Automatic unlock**: TTL ensures locks are released if payment is abandoned.

6.3.3 Trending Properties with Sorted Sets

Redis sorted sets maintain real-time property popularity rankings:

```
1   // Increment view count when property is viewed
2   public PropertyResponseDTO getPropertyDetails(String propertyId) {
3       Property property = propertyRepository.findById(propertyId)
4           .orElseThrow(() -> new IllegalArgumentException("Not found"));
5
6       // ZINCRBY trending_properties 1 <propertyId>
7       redisTemplate.opsForZSet()
8           .incrementScore("trending_properties", propertyId, 1);
9
10      return mapToDTO(property);
11  }
12
13  // Get top 10 trending properties
14  public List<PropertyResponseDTO> getTrendingProperties() {
15      // ZREVRANGE trending_properties 0 9
16      var topIds = redisTemplate.opsForZSet()
17          .reverseRange("trending_properties", 0, 9);
18
19      if (topIds == null || topIds.isEmpty()) return List.of();
20
21      List<String> ids = topIds.stream()
22          .map(Object::toString)
23          .collect(Collectors.toList());
24
25      return propertyRepository.findAllById(ids).stream()
26          .map(this::mapToDTO)
27          .collect(Collectors.toList());
28  }
```

Redis sorted sets provide:

- **O(log N) insertion**: Efficient score updates.
- **O(log N + M) range queries**: Fast top-K retrieval.
- **Atomic operations**: Thread-safe score increments.

6.3.4 User Browsing History with Lists

Redis lists store recently viewed properties per user:

```
public void addToUserHistory(String userId, String propertyId) {
    String key = "history:" + userId;

    // LPUSH adds to front of list
    redisTemplate.opsForList().leftPush(key, propertyId);

    // LTRIM keeps only last 10 items
    redisTemplate.opsForList().trim(key, 0, 9);
}

public List<PropertyResponseDTO> getUserHistory(String userId) {
    String key = "history:" + userId;

    // LRANGE retrieves all items
    List<Object> historyIds = redisTemplate.opsForList()
        .range(key, 0, -1);

    if (historyIds == null || historyIds.isEmpty())
        return List.of();

    List<String> ids = historyIds.stream()
        .map(Object::toString)
        .collect(Collectors.toList());

    return propertyRepository.findAllById(ids).stream()
        .map(this::mapToDTO)
        .collect(Collectors.toList());
}
```

This pattern:

- Uses `LPUSH` for O(1) insertion at head.
- Uses `LTRIM` to maintain fixed-size history (last 10 properties).
- Provides O(1) retrieval for "recently viewed" feature.

6.4 Summary

The query implementations in this chapter demonstrate how the application leverages each database technology's strengths:

- **MongoDB**: Complex document queries using `@Query` annotations, powerful aggregation pipelines with `@Aggregation` for analytics, and atomic updates with `@Update`.
- **Neo4j**: Graph traversal for collaborative filtering recommendations, relationship management with Cypher's `MERGE` and `MATCH` patterns.
- **Redis**: In-memory caching with TTL, pessimistic locking for reservations, real-time rankings with sorted sets, and browsing history with lists.

The query-driven design approach ensures that data models are optimized for the access patterns they need to support, resulting in efficient and maintainable database operations throughout the application.