UNIVERSIDAD DISTRITAL
FRANCISCO JOSÉ DE CALDAS
Acreditación Institucional de Alta Calidad

District University Francisco José de Caldas

Systems Engineering Project

Secure and Scalable Cloud Storage Platform – Google Drive Clone

Cristian David Casas Diaz – 20192020091

Dylan Alejandro Solarte Rincón – 20201020088

Supervisor: Carlos Andrés Sierra Virguez

A report of the first delivery of the final project for the course
Databases II *Systems Engineering*

July 10, 2025

Abstract

This report presents the initial phase of a secure and scalable cloud storage platform project inspired by Google Drive. The solution aims to deliver a web-based file management system with robust database architecture, focusing on file storage, access control, metadata indexing, logging, and synchronization. This first delivery outlines the project scope, assumptions, system requirements, and methodological foundations based on user needs, data modeling, storage structures, and system architecture.

Keywords: Cloud Storage, File Management System, Metadata Indexing, Access Control, Database Design, User Stories, ER Model, Functional Requirements, Web Application, Logging System, Hybrid Architecture, Security, Scalability, Business Model Canvas, NoSQL, SQL, File Sharing

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Cloud storage platforms have become fundamental in enabling individuals and organizations to store, manage, and share digital content in a secure and efficient manner. The proliferation of digital data—ranging from personal media to institutional records—has highlighted the necessity for reliable, accessible, and scalable storage solutions.

While commercial providers like Google Drive, Dropbox, and OneDrive dominate the market, these solutions often fall short in scenarios where transparency, flexibility, or affordability are critical, especially in academic or small-scale environments. Additionally, they frequently offer limited insights into backend data structures, which restricts learning opportunities in educational contexts.

This project, developed within the framework of the Databases II course, focuses on implementing a Google Drive-inspired cloud storage system with an academic and engineering perspective. The solution emphasizes the integration of a hybrid database architecture that combines relational and NoSQL models, enabling efficient metadata indexing, access control, session logging, and user activity tracking. The system is accessible through a web-based interface and supports common operations such as file upload, sharing, and version control.

The project not only aims to replicate key functional aspects of a commercial drive solution but also offers an opportunity to explore core database principles, concurrency control mechanisms, and system performance trade-offs in a real-world inspired setting.

1.1 Background

The need for cloud storage solutions has grown due to the increasing volume of digital data. However, many existing services are either too costly or complex for smaller-scale users. This project addresses that gap by designing a scalable solution based on solid database principles. As UNIR (2025) explains, proper data modeling ensures better system behavior, especially when handling metadata, logs, and access permissions. With these principles, our platform aims to deliver reliable, secure, and efficient data storage.

1.2 Problem statement

Despite the widespread use of cloud storage services, many existing platforms offer limited adaptability for academic use or small-scale deployment. They are often designed for enterprise-level clients and come with significant financial and technical overhead. This makes them unsuitable for users seeking customizable, transparent, and cost-effective solutions.

There is a growing demand for a platform that prioritizes modularity, security, and efficient data architecture—without the complexity of enterprise-grade software. This project addresses that gap by offering a lightweight yet functional clone of Google Drive, centered on educational needs and grounded in database engineering best practices.

1.3 Objectives

- Design a hybrid data model that integrates relational and NoSQL components to manage users, sessions, file versions, access metrics, and sharing logs.
- Implement core file management features including upload, sharing, access control, and version history.
- Develop a responsive, secure web interface tailored for educational deployment.
- Provide robust logging, monitoring, and indexing to support system audits and usage analytics.
- Ensure data consistency and performance through concurrency control techniques.

1.4 Solution approach

The system is built as a modular cloud application, integrating both SQL and NoSQL database layers to support distinct workloads. PostgreSQL handles structured user data, authentication, sessions, and logs, while MongoDB is used to store file system metadata and directory hierarchies. Redis and in-memory caching improve performance for frequent queries, while a RESTful backend exposes endpoints to the web interface.

Special focus is given to concurrency, indexing strategies, storage optimization, and traceability. The backend services adopt a clean architectural pattern to maintain separation of concerns, and the data model has been refined iteratively to support future scalability.

1.5 Scope

- User registration and authentication
- File and folder upload, renaming, deletion
- Role-based sharing and permission controls
- Version tracking and access history
- Analytics dashboard (basic usage statistics)
- Hybrid storage model.
- Web-based interface optimized for desktop

Exclusions

- Mobile or desktop native clients
- Real-time collaborative editing (e.g. simultaneous document editing)
- Offline file access and synchronization
- Support for external storage providers (e.g. Dropbox API integration)

1.6 Assumptions

- Users will access the system via modern web browsers.
- The system will scale up to 5,000 concurrent users in academic environments.
- Internet connectivity is assumed to be stable. e.
- Users are categorized as either "basic" or "premium" based on their subscription plan.
- All data is stored within institutional servers or free-tier cloud services.

1.7 Limitations

- The system does not support mobile applications or offline functionality.
- Due to limited resources, third-party premium APIs or services were avoided.
- Real-time collaborative features were excluded from this version.
- Backup and failover mechanisms are basic and not enterprise-grade.

1.8 Summary of contributions and achievements

- Designed and implemented a hybrid data architecture combining SQL and NoSQL components.
- Built a fully functional cloud storage prototype
- Defined and refined a modular ER model supporting user sessions, access logs, and file metrics.
- Proposed a modular, scalable system architecture following a hexagonal pattern.
- Identified key performance, security, and scalability challenges and proposed mitigation strategies.
- Successfully **deployed the relational schema using Railway**, allowing live testing and remote queries.
- Developed **Python-based dummy data generators** to populate the database and test its behavior under controlled scenarios.

Chapter 2

Literature Review

The design of our cloud storage platform is informed by a broad review of existing technologies, systems, and academic insights:

2.1 State of the Art in Cloud Storage Systems

Cloud storage systems have evolved significantly over the past two decades, offering highly available, scalable, and secure environments for digital content management. Platforms such as **Google Drive**, **Dropbox**, and **OneDrive** rely on distributed file systems (DFS), indexed metadata layers, and fine-grained role-based access control (RBAC) to ensure responsiveness, collaboration, and data integrity (Google Workspace, 2025; AWS, 2025).

These systems implement redundant storage nodes and strong consistency protocols, enabling synchronous collaboration and automatic versioning. At the backend, modern cloud services leverage hybrid architectures, combining relational databases for transactional consistency with NoSQL models for flexibility and speed in metadata access.

In parallel, academic and industrial research highlights the importance of **modular architecture**, **event-driven design**, and **monitoring tools** such as **Prometheus** to detect bottlenecks and optimize system performance (IBM Cloud Design Guide, 2025; UNIR, 2025).

2.1.1 Comparison with Existing Solutions

While commercial platforms are highly performant, they often rely on proprietary protocols, costly licensing models, and opaque internal architectures. These aspects limit their educational or small-scale applicability, where budget constraints and transparency are critical.

In contrast, our system emulates key features—such as file versioning, metadata indexing, access logs, and dashboards—using **open-source tools** like PostgreSQL, MongoDB, and Python-based scripts for simulation. Moreover, deployment in platforms such as **Railway** enables real-world accessibility while avoiding vendor lock-in.

This positions the project as an **educationally accessible alternative**, maintaining functional parity with commercial systems but with open architecture and a focus on learning outcomes.

2.1.2 Relevance to Our Project

The literature and real-world case studies have directly influenced the following architectural decisions in our project:

- **Hybrid database architecture**: Leveraging PostgreSQL for structured relational data and MongoDB for flexible metadata, enabling efficient indexing and folder hierarchy management.
- **Modular (hexagonal) architecture**: Inspired by DDD (Domain-Driven Design), our backend separates infrastructure concerns from domain logic, improving maintainability.
- **Concurrency-aware design**: Based on findings in concurrency control literature, we implemented custom strategies to avoid write collisions and ensure data integrity.
- **Monitoring**: Though not implemented in full, Prometheus and other logging tools were considered during system analysis and design for future enhancements.

2.2 Critique of Existing Work

Existing cloud platforms achieve excellent scalability and user experience but require significant infrastructure, often involving enterprise-level cloud providers and proprietary software stacks. Additionally, these platforms do not expose their backend data models, which restricts learning and customization opportunities.

Our project addresses these shortcomings by offering:

- Open architecture and free-tier deployments
- Modular, maintainable codebase for educational extension
- Custom scripts to test concurrency and access behavior
- Clear ER models and documentation for academic purposes

This approach promotes the development of **mid-scale, transparent, and database-driven** cloud solutions applicable to university labs, internal teams, and lightweight enterprise scenarios.

2.3 Summary

This chapter reviewed the technologies and design patterns that inspired our cloud storage platform. Commercial services like Google Drive and AWS set the standard in terms of robustness and scalability, while academic sources provided architectural and database modeling guidelines.

Our solution builds on these insights by adopting:

- Distributed metadata storage
- Role-based permissions and logging
- Hybrid SQL-NoSQL architecture
- Deployment on Railway with data simulation

The literature validates the need for scalable, open, and educationally accessible cloud storage platforms. By implementing a real system aligned with these principles, our project bridges theory and practice, offering a concrete tool to support data engineering education and experimentation.

Chapter 3

Methodology

The development of the Google Drive Clone project followed a comprehensive and structured methodology, guided by the principles of software engineering, database design, and agile development. Below are the key elements that shaped the process:

3.1 User Stories and Functional Analysis

We began by defining over 20 user stories addressing critical platform functionalities such as file upload, folder organization, sharing, synchronization, permissions, previews, and analytics. Each story was prioritized and time-estimated, and included clear acceptance criteria.

3.1.1 Examples of User Stories

Table 3.1: HU 1

Title: Secure File Upload	**Priority:** High	**Estimate:** 5 days
User Story: As a user, I want to securely and quickly upload files to the cloud to ensure accessibility from any device.		
Acceptance Criteria: Given I have an internet connection, when I upload a file, then the file is securely stored in the cloud and accessible from my devices.		

Table 3.2: HU 2

Title: Folder Organization	**Priority:** High	**Estimate:** 3 days
User Story: As a user, I want to create and organize folders to manage my files efficiently.		
Acceptance Criteria: Given I am logged in, when I create a new folder, then I can move files into it and manage my files efficiently.		

Table 3.3: HU 3

Title: File Sharing	**Priority:** High	**Estimate:** 5 days
User Story: As a user, I want to share files and folders with others to enable seamless collaboration.		
Acceptance Criteria: Given I have selected a file or folder, when I choose to share it, then I can provide access to others for collaboration.		

3.2 Business Model Canvas (BMC)

A complete Business Model Canvas was elaborated to map out the project's strategic vision. Key sections included value propositions (e.g., secure and scalable cloud storage), customer segments (individual and enterprise users), revenue streams (premium plans, enterprise pricing), and cost structure (mainly development and hosting). This ensured technical decisions supported business viability.

Link Business Model Canva

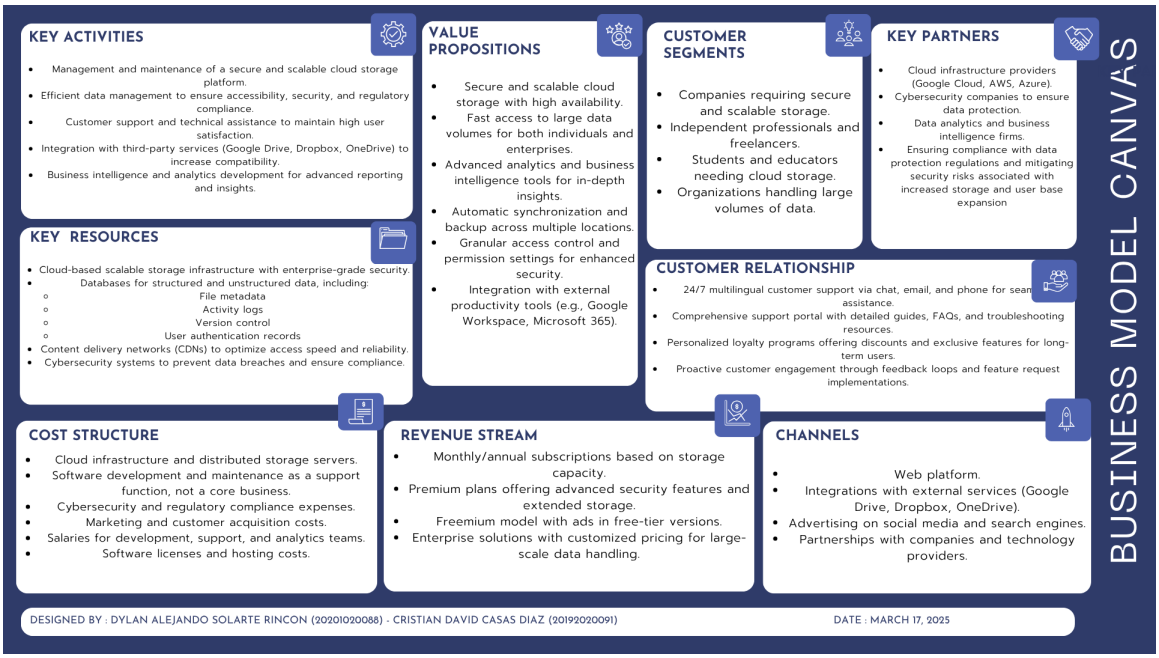


Figure 3.1: Modelo Canva

3.3 Data Sources and Structures

We identified and categorized essential data sources:

- *User Data*: includes registration, authentication logs, session history, and roles.
- *File Metadata*: tracks name, type, size, timestamps, version history.
- *Sharing and Permissions*: shared links, user-based ACLs, expiration rules.

- *Activity Logs*: operations such as uploads, downloads, deletions.
- *Analytics and Security Data*: usage statistics, login attempts, quota consumption.

To manage this data effectively, the following structures were designed:

- **Blob Storage System**: for storing the actual files using Amazon S3.
- **Metadata Indexing System**: using NoN SQL to support rapid querying.
- **Access Control Lists (ACLs)**: permission management stored.
- **Logging System**: distributed logs captured via event-driven architecture.
- **Caching**: Redis used for frequently accessed file metadata.

3.4 Requirements Engineering

The requirements of the system were defined through a structured process supported by user stories, use-case analysis, and iterative refinement. In the initial delivery, a comprehensive set of functional and non-functional requirements was proposed. However, during the enhancement phase documented in **Workshop 2 – Incrementing Delivery 1**, these were critically re-evaluated to improve coherence, traceability, and development feasibility.

As a result of this analysis, the set of requirements was **reduced and reorganized**, selecting only those aligned with the final architecture, available resources, and prioritized user stories. This decision was driven by:

- The need to maintain alignment between the database schema and business logic.
- The importance of simplifying testing and validation under academic constraints.
- The elimination of redundant or low-impact requirements.
- The goal of focusing development efforts on high-value modules (e.g., file management, access control, session tracking).

Functional Requirements (FR1–FR8) The finalized set includes 8 core functional requirements:

- **FR1**: User registration and authentication via web interface.
- **FR2**: Upload, organization, and deletion of files and folders.
- **FR3**: Role-based access and permission control.
- **FR4**: Recycle bin functionality for deleted items.
- **FR5**: Activity logs for file access and operations.
- **FR6**: User profile configuration and session management.
- **FR7**: Display of file metadata and basic version tracking.
- **FR8**: Access to usage statistics through a visual dashboard.

Each requirement is directly traceable to one or more of the 20 validated user stories.

Non-Functional Requirements (NFR1–NFR6) Similarly, the non-functional requirements were refined to 6 key attributes critical to platform usability and reliability:

- **NFR1**: Upload operations must achieve a minimum of 2 MB/s on a 50 Mbps connection.
- **NFR2**: The system must support at least 1,000 concurrent users and be scalable to 5,000.
- **NFR3**: Availability target is 99.9%, using Railway's managed PostgreSQL.
- **NFR4**: Interface must be fully responsive on major desktop browsers.
- **NFR5**: Security must include hashed credentials and secured tokens for session handling.
- **NFR6**: Platform should operate on free-tier infrastructure without requiring paid services.

This revised list reflects a **focused, realistic scope**, prioritizing platform functionality, maintainability, and deployment simplicity. It also enhances alignment between theoretical requirements and real system implementation.

3.5 Entity-Relationship Model (ERD)

The current data model was redesigned to optimize scalability, modularity, and domain separation, reflecting improvements introduced during Workshop 2. The final version includes both **relational entities** (implemented in PostgreSQL) and **NoSQL structures** (stored in MongoDB).

Core Entities (Relational Model – PostgreSQL) The ER model consists of the following key entities:

- **User**: Central table storing login credentials, personal details, and storage quotas. Tracks creation and last login timestamps, verification status, and user activity level.
- **User_Session**: Represents active and historical sessions, recording IP, device, and expiration metadata. Allows auditing and management of concurrent device access.
- **Subscription_Plan**: Defines available plans (e.g., free, premium), including associated storage limits and prices.
- **File_Access_Metrics**: Tracks daily view/download counts for each file, supporting statistical analysis and dashboard features.
- **Version_History**: Stores multiple historical versions of a file, including creator, timestamp, file size, and checksum hash for integrity verification.
- **Authentication_Log**: Logs successful and failed login attempts, including metadata such as IP address, timestamp, and device.
- **Activity_Log**: Records system-level actions performed by users (e.g., deletion, renaming, movement of files or folders), tied to both user ID and resource type.
- **User_Usage_Metrics**: Aggregates usage per user per day, tracking file transfers, storage consumption, login frequency, and shared content activity.

- **Sharing_Activity_Metrics**: Captures sharing-related actions, such as links created, shared items, and revoked permissions.

These relational entities support structured queries, maintain data consistency, and power analytics across the platform.

Non-Relational Entities (NoSQL – MongoDB) **File**: Stored in a NoSQL document format to support hierarchical folder structures, dynamic metadata, tags, and JSON-based file descriptors. This allows flexibility in storing nested folders, shared access maps, and indexing fields for full-text search.

This hybrid data modeling approach allows combining the **transactional guarantees of SQL** (for core identity, logs, and metrics) with the **flexibility of NoSQL** (for unstructured or semi-structured content such as folders and extended file metadata).

In the following link you can see the database diagram in greater detail: ER Diagram Repository Link

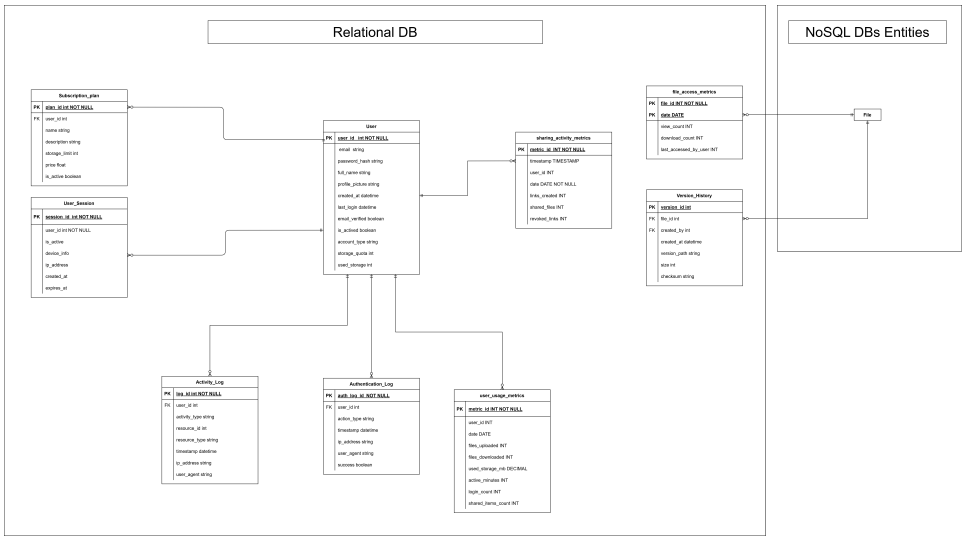


Figure 3.2: General ER Diagram

3.5.1 Domain-Based ER Breakdown

To improve modularity and logical separation of concerns, the database model was divided into three primary **domains**: User Domain, Logging Domain, and Analytics Domain. Each domain contains related entities with strong internal cohesion, aligned with eventual modular or service-oriented backend implementations. Although this project focuses solely on database design and does not implement backend coordination, entity connectors are visually denoted to indicate future integration points between domains.

A. User Domain

Technologies: PostgreSQL (Relational Database)

This domain encapsulates entities related to user identity, account management, and active sessions. It includes:

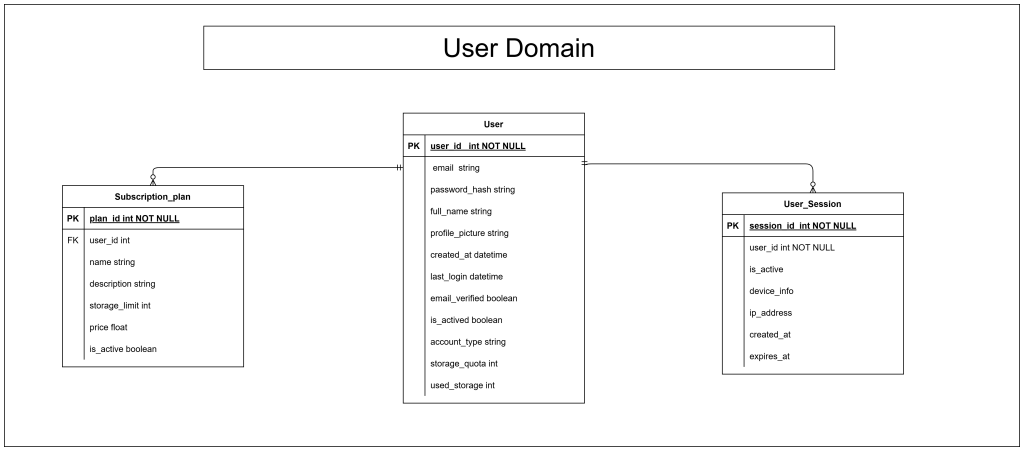


Figure 3.3: User Domain

- **User**: The central entity for all personal and authentication-related data, including email, encrypted password, profile information, quotas, and account status.
- **User_Session**: Tracks active and historical login sessions, devices, IP addresses, and expiration metadata.
- **Subscription_Plan**: Defines the storage plan associated with each user, including plan characteristics such as name, description, limit, and price.

These entities ensure integrity in user management and support extensibility for future roles or subscription-based access control. The relationships are strongly normalized and optimized for transactional queries and audits.

B. Logging Domain

Technologies: PostgreSQL (Relational Database)
This domain is responsible for system observability and traceability of actions:

- **Activity_Log**: Records all significant interactions by users with files or system resources, storing metadata such as resource ID, action type, timestamps, and client information.
- **Authentication_Log**: Stores login attempts (successful or failed) along with IP, user agent, and outcome.
- **Version_History**: Maintains an audit trail of file versions, tracking who made changes, when, and the resulting metadata (path, size, checksum).

All logs are tightly linked to the **User** and **File** entities (the latter modeled in NoSQL), ensuring traceability and enabling future forensic or compliance auditing.

> Note: The **File** entity is referenced here symbolically as a placeholder for backend coordination, as file-related metadata resides in a NoSQL database and was not implemented in this academic prototype.

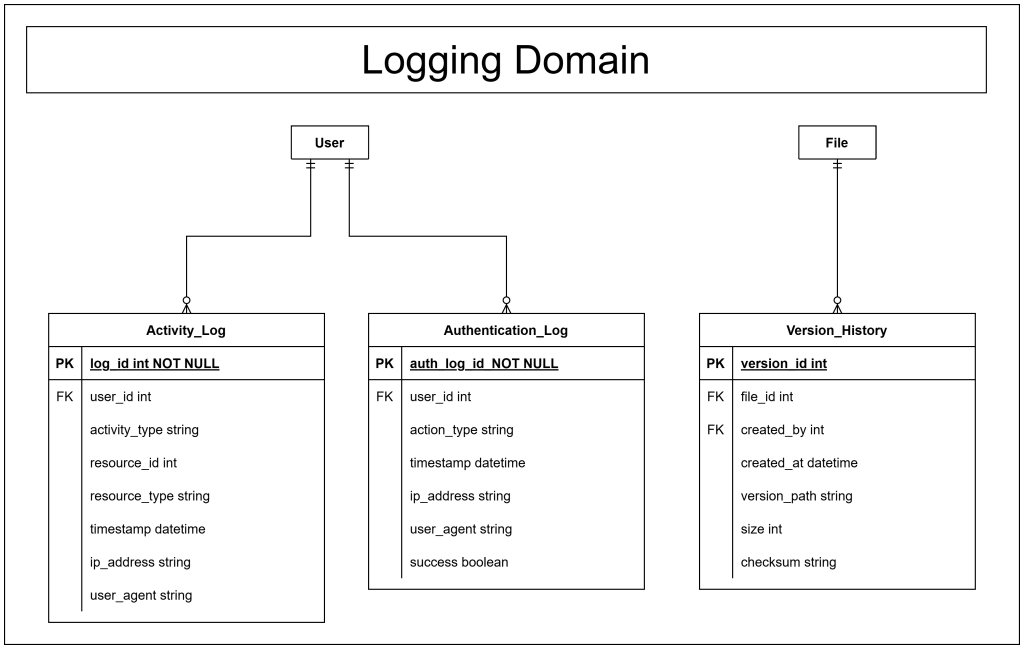


Figure 3.4: Loggin Domain

C. Analytics Domain

Technologies: PostgreSQL (Relational Database)

This domain focuses on aggregating and tracking usage patterns across users and system components:

- **file_access_metrics**: Captures view/download counts and last-accessed metadata per file and date.
- **user_usage_metrics**: Aggregates per-user statistics, including uploads, downloads, storage consumption, login frequency, and active time.
- **sharing_activity_metrics**: Records sharing activity such as number of links generated, items shared, and revoked accesses.
- **system_performance_metrics**: Monitors global system KPIs including CPU usage, I/O throughput, response time, and concurrent session counts.
- **tag_usage_metrics**: Tracks the application and search frequency of tags (stored in the NoSQL layer).

These metrics serve as the foundation for visual dashboards, usage analytics, and resource optimization strategies. The domain is designed to scale efficiently and support time-series aggregations.

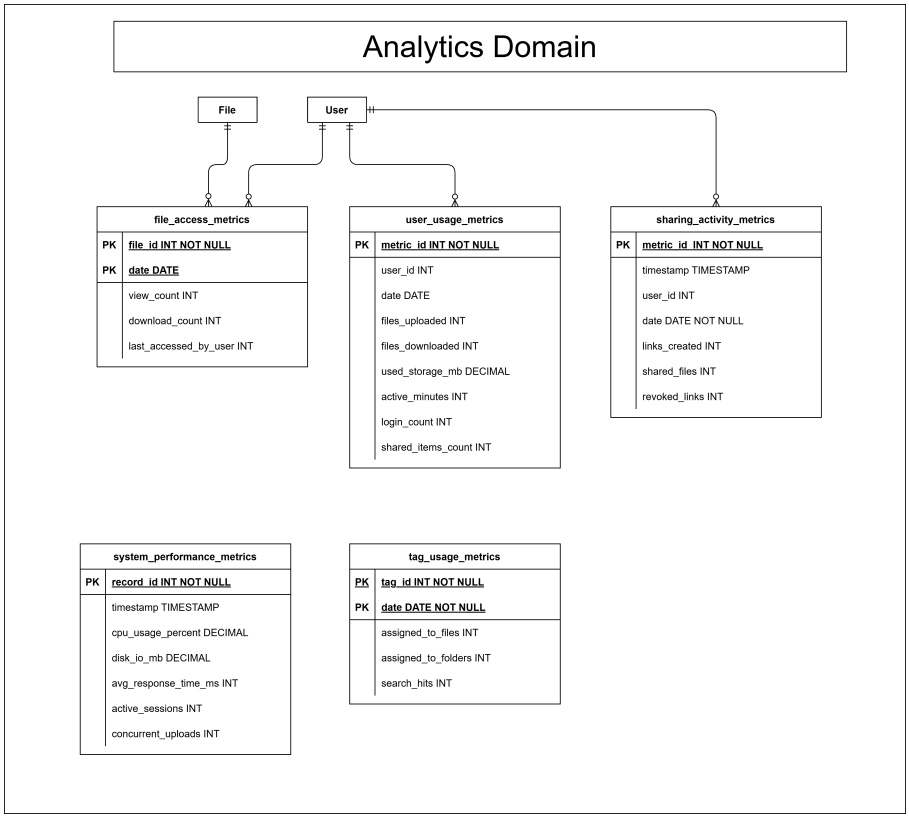


Figure 3.5: Analytics Domain

3.6 System Architecture

To support scalability, modular development, and data heterogeneity, the system architecture adopts a layered and domain-centric design, aligned with the principles of **Hexagonal Architecture**. This structure separates business logic from infrastructure details, ensuring maintainability, testability, and clear integration boundaries.

Architectural Overview The architecture is divided into the following key components:

- **API Gateway**: Serves as the single entry point for external requests, managing routing, authentication, and access policies. It enables seamless interaction with internal services without exposing their internal complexity.
- **Application Services Layer**: Implements core business functionalities through domain-specific services such as:
  - **UserService**: Manages user data, sessions, and subscription plans.
  - **FileService**: Handles file metadata, versioning, and related operations.
  - **SharingService**: Coordinates access permissions and shared link generation.
  - **PermissionService**: Enforces fine-grained access control over files and folders.
  - **LoggingService**: Records user and system-level events for traceability and auditability.

  - **AnalyticsService**: Aggregates and exposes key performance indicators and usage metrics.

- **Adapters Layer (Ports and Repositories)**: Facilitates interaction between domain logic and persistence mechanisms. Each repository interfaces with a specific database technology:

  - **UserRepo**, **LogRepo**: Connect to relational databases.
  - **FileRepo**, **PermissionRepo**: Interact with NoSQL databases for flexible document and graph storage.
  - **StorageAdapter**: Abstracts integration with the file storage system, which may be hosted on-premises or in the cloud.
  - **AnalyticsRepo**: Collects and queries metrics from the analytics database.

- **Data Sources**:

  - **Relational Databases**: Used for structured data such as users, subscriptions, logs, and analytics metrics.
  - **NoSQL Databases**: Employed for flexible schema management of file metadata, tags, folder hierarchies, and permissions.
  - **On-Premises Storage**: Manages binary file content using chunked and deduplicated storage mechanisms.
  - **Warehouse & Datalake (Optional Extension)**: Represent integration points for large-scale analytical pipelines, designed to support future enhancements involving ETL processes and replicated datasets.

This architecture ensures that each component can evolve independently while maintaining interoperability. By separating data storage strategies and service responsibilities, the platform supports future extensions such as analytics dashboards, permission audits, and advanced search engines.

Link Repository, Final Architecture

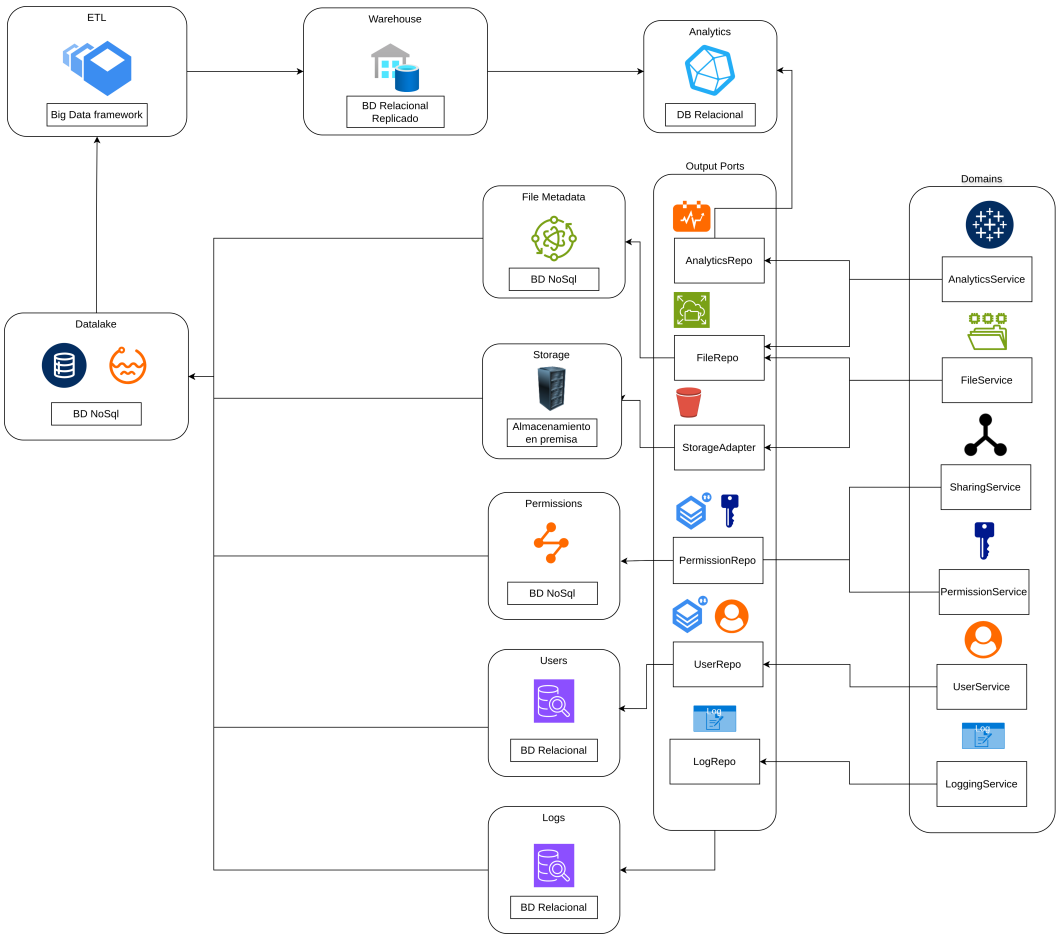


Figure 3.6: Initial Architecture Diagram

3.7 Potential Problems and Considerations

During the development of the cloud storage prototype, several challenges and design trade-offs were identified that could impact future scalability or robustness if not addressed:

- **Data consistency across hybrid storage**: The use of both relational and NoSQL databases introduces complexity in maintaining synchronization between user activity logs, metadata, and actual file records. Without transactional guarantees across both systems, inconsistencies may arise during concurrent operations.
- **Security and compliance risks**: Although basic security measures such as hashed credentials and secure sessions were implemented, the system lacks more advanced mechanisms like OAuth 2.0 integration, role hierarchies, and audit compliance (e.g., GDPR-ready logging).
- **Scalability limits of free-tier infrastructure**: The decision to operate within free-tier services (e.g., Railway for PostgreSQL, MongoDB Atlas) imposes limitations on connection pools, I/O throughput, and storage capacity. These constraints may impact performance under higher user loads.
- **Absence of end-to-end file encryption**: While user credentials and sessions are protected, file content is not encrypted beyond transport (TLS). Future implementations should consider per-file encryption strategies to ensure privacy and legal compliance.

- **Lack of concurrency control in metadata updates**: Without a locking or versioning mechanism at the database level, concurrent metadata updates (e.g., file renames, permission changes) may lead to race conditions.
- **Incomplete backend orchestration**: Although the ER model and domain-based architecture suggest future microservices adoption, actual backend coordination and inter-service communication mechanisms (e.g., event buses or service registries) are not implemented in this academic prototype.

Despite these challenges, the modularity and clarity of the current architecture provide a solid foundation for future iterations and improvements.

3.8 Summary

This chapter described the methodology used to develop a scalable and modular cloud storage platform, tailored for academic and lightweight enterprise use. Beginning with the definition of user stories, the team identified functional requirements aligned with core user needs—file uploads, access control, and analytics—while prioritizing implementation feasibility.

A refined set of functional and non-functional requirements was established, emphasizing usability, scalability, and maintainability under constrained infrastructure. The database model was redesigned to support domain separation and hybrid data storage strategies, balancing relational consistency with NoSQL flexibility.

The resulting system architecture—grounded in hexagonal principles—supports separation of concerns, future service modularization, and agnostic integration with storage and analytics layers. While limitations remain, the groundwork laid during this phase validates the technical feasibility and offers a strong baseline for continued evolution.

Chapter 4

Results

4.1 System Deployment and Environment

To validate the database schema and architecture, the system was deployed on the Railway platform, which offers a cloud-based development environment with support for PostgreSQL and MongoDB. The deployment was structured to reflect the domain-based separation of the data model, with the following instances:

- **Three PostgreSQL databases**, each corresponding to the User, Logging, and Analytics domains.
- **One MongoDB database**, used to store the flexible file metadata model, folder hierarchies, and shared access structures.

Environment Configuration The deployment used Railway's **trial plan**, which provides limited but sufficient resources for testing purposes:

- **Compute Resources**: 2 vCPUs and 512 MB to 1 GB of RAM per service (as per real-time scaling).
- **Deployment Region**: US East (Virginia, USA).
- **Service Isolation**: Each domain ran as a separate Railway PostgreSQL service, allowing modular management.
- **PostgreSQL Image**: The latest available PostgreSQL container with `postgres-ssl:16`.

The services were deployed without cron schedules or volume attachments to comply with the free-tier limitations. As a result, replication was not enabled. No custom teardown or serverless behavior was configured for the services.

Data Injection Scripts To populate the system for testing and analytics, **Python 3 scripts** were developed to connect directly to the PostgreSQL and MongoDB instances hosted on Railway. These scripts inserted synthetic (dummy) data aligned with the designed ER diagrams and included:

- User creation and subscription linking.
- File metadata and version records.

- Activity and authentication logs.
- Usage metrics and sharing statistics.

Data insertion times were measured and evaluated, yielding consistent ingestion durations ranging between **5 to 30 seconds** per dataset, depending on record volume and complexity. This indicates that the schema and the indexing strategies were well-optimized for initial loads, even under constrained environments.

Limitations The use of Railway's free-tier introduced several constraints:

- **Resource limitations** (memory and CPU), especially under concurrent insertions or complex joins.
- **Lack of automated backups or high availability configurations**.
- **No integrated CI/CD pipelines**, requiring manual deployment and environment updates.
- **Limited scalability for sustained or production workloads**.

Despite these limitations, the deployment successfully demonstrated the architecture's modularity and the ability of the database schema to support analytics, logging, and user interactions in a realistic environment.

4.2 Data Ingestion Results

To validate the structure and performance of the designed databases, four custom Python 3 scripts were developed for synthetic data generation and bulk insertion into the four logical domains: User, File, Logging, and Analytics. Each script leverages realistic data patterns and efficient batch operations to simulate real-world usage scenarios.

User Domain (PostgreSQL)

- **Script**: `gDatosDomUser.py`
- **Volume**:
  - 5,000 users
  - 5,000 subscription plans (one per user, with weighted random distribution)
  - Approx. 12,000 sessions (1–5 per user)
- **Generation Strategy**:
  - Uses `Faker` for realistic names, emails, IPs, and user agents.
  - Passwords are hashed using SHA-256.
  - Plans are assigned based on a probabilistic model (Free: 60%, Básico: 30%, Premium: 10%).
  - Sessions include expiration metadata and simulate multiple device access.

File Domain (MongoDB)

- **Script**: `gDtosDomFile.py`
- **Volume**:
  - 100 tags
  - 5,000 folders
  - 20,000 files
- **Generation Strategy**:
  - Stored in `FileManagement` database with three collections: `tags`, `folders`, and `files`.
  - Tags created with `ObjectId` and bound to randomly selected user IDs.
  - Folders support nested hierarchies (`parent_folder_id`) and various metadata fields like tags, visibility, and shared access.
  - Files emulate common MIME types and support metadata such as checksum, encryption key, version, and detailed access metrics.

Logging Domain (PostgreSQL)

- **Script**: `gDatosDomLogging.py`
- **Volume** (approximate):
  - 10,000 authentication log entries
  - 20,000 activity logs
  - 8,000 file version entries
- **Generation Strategy**:
  - Designed to simulate both successful and failed authentications using a randomized model.
  - Activity logs represent actions like file deletion, renaming, moving, etc.
  - File versions maintain audit trails, including checksums and author metadata.

Analytics Domain (PostgreSQL)

- **Script**: `gDatosDomAnalytics.py`
- **Volume** (approximate):
  - 30,000 user usage metrics (per day)
  - 25,000 file access metrics

  - 15,000 sharing activities
  - 6,000 system performance snapshots
  - 5,000 tag usage records

- **Generation Strategy**:
  - Focused on producing time-series data useful for dashboards and performance analysis.
  - Metrics cover KPIs like CPU load, throughput, file views/downloads, and sharing behavior.

Execution and Performance

- All scripts were executed using Railway-managed databases (PostgreSQL 16 + MongoDB Atlas Free Tier).
- **Average ingestion time per script**:
  - User Domain: 25–30 seconds
  - File Domain: 30–40 seconds
  - Logging Domain: 20 seconds
  - Analytics Domain: 10–15 seconds
- **Total ingest volume**: 150,000 records across relational and non-relational databases.
- The ingestion leveraged bulk operations (`executemany`, `bulk_write`) to maximize throughput.
- Each script includes dynamic DSN resolution using environment variables (e.g., `DATABASE_URL`, `MONGO_URI`), enabling flexible deployments.

This dataset now serves as a foundation for testing querying strategies, validating schema design, and simulating load under realistic scenarios. Performance benchmarks from the ingestion process also confirm the feasibility of the system under moderate-scale usage.

4.3 Query Performance Evaluation – User Domain

4.3.1 Query 1: Plan Distribution and Potential Revenuel

Description:

This query calculates how many users are subscribed to each active plan and estimates the potential monthly revenue if all users paid their fees. It also calculates the percentage of users per plan out of the total.

```
1 WITH planes_activos AS (
2   SELECT name, price FROM plan_suscripcion WHERE is_active
3 )
4 SELECT name AS tipo_plan, COUNT(*) AS usuarios, price,
5        COUNT(*) * price AS ingreso_mensual_estimado,
6        ROUND(100.0 * COUNT(*) / SUM(COUNT(*)) OVER (), 2) AS
     porcentaje_total
7 FROM planes_activos
8 GROUP BY name, price
9 ORDER BY usuarios DESC;
```

Listing 4.1: First User Domain Query

Performance Analysis:

- **Total Execution Time:** 2.064 ms
- **Aggregation Strategy:** `HashAggregate` with a `WindowAgg` function
- **Access Pattern:** Sequential Scan on `plan_suscripcion`
- **Buffer Hits:** 56 in sort, 53 during aggregation
- **Remarks:** The query executes efficiently with acceptable performance for analytical purposes and no expensive joins involved.

width=0.52.png

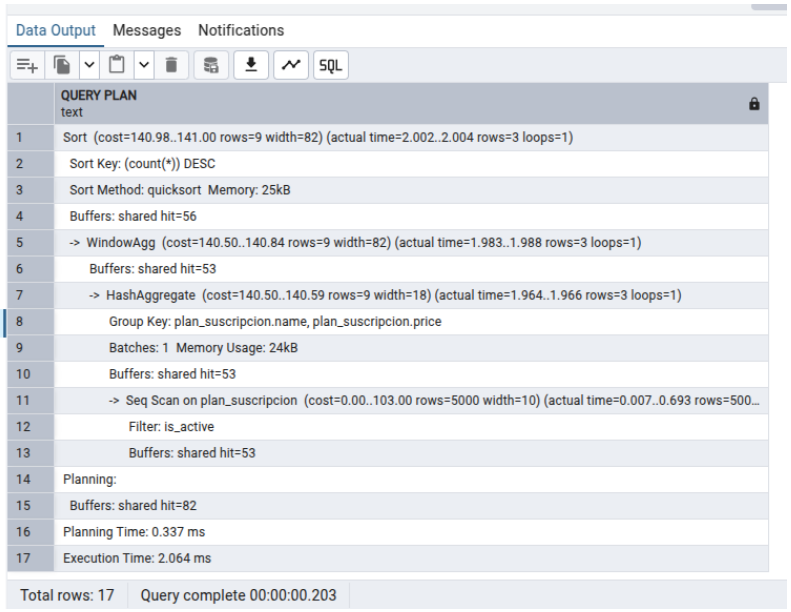


Figure 4.1: Performance of Query 1

4.3.2 Query 2: Premium Users with > 80% Storage Usage

Description:

Identifies Premium users who are using more than 80% of their storage quota. This is useful for upselling opportunities or storage optimization.

```
SELECT u.user_id, u.email, u.full_name, u.used_storage, u.storage_quota,
       ROUND(100.0 * u.used_storage / NULLIF(u.storage_quota,0),2) AS
    porcentaje,
       p.plan_id, p.name AS plan
FROM usuario u
JOIN plan_suscripcion p USING (user_id)
WHERE p.name = 'Premium' AND u.used_storage >= 0.8 * u.storage_quota
ORDER BY porcentaje DESC;
```

Listing 4.2: First User Domain Query

Performance Analysis:

- **Total Execution Time:** 2.175 ms
- **Join Strategy:** `Hash Join`
- **Access Pattern:** Sequential Scan on both `usuario` and `plan_suscripcion`
- **Buffer Hits:** 191 total; 3968 and 4518 rows filtered respectively
- **Remarks:** Reasonable execution time with a heavy use of filters. The use of `Hash Join` ensures performance stays manageable.

1	Sort (cost=375.70..376.11 rows=161 width=115) (actual time=2.131..2.138 rows=94 loops=1)
2	Sort Key: (round(((100.0 * (u.used_storage)::numeric) / (NULLIF(u.storage_quota, 0))::numeric), 2)) DESC
3	Sort Method: quicksort Memory: 36kB
4	Buffers: shared hit=191
5	-> Hash Join (cost=121.53..369.80 rows=161 width=115) (actual time=0.510..2.044 rows=94 loops=1)
6	Hash Cond: (u.user_id = p.user_id)
7	Buffers: shared hit=191
8	-> Seq Scan on usuario u (cost=0.00..238.00 rows=1667 width=69) (actual time=0.010..1.407 rows=1032 loops=1)
9	Filter: ((used_storage)::numeric >= (0.8 * (storage_quota)::numeric))
10	Rows Removed by Filter: 3968
11	Buffers: shared hit=138
12	-> Hash (cost=115.50..115.50 rows=482 width=22) (actual time=0.490..0.491 rows=482 loops=1)
13	Buckets: 1024 Batches: 1 Memory Usage: 35kB
14	Buffers: shared hit=53
15	-> Seq Scan on plan_suscripcion p (cost=0.00..115.50 rows=482 width=22) (actual time=0.004..0.420 rows=48...
16	Filter: (name = 'Premium'::text)
17	Rows Removed by Filter: 4518
18	Buffers: shared hit=53
19	Planning:
20	Buffers: shared hit=6
21	Planning Time: 0.207 ms
22	Execution Time: 2.175 ms

Total rows: 22 Query complete 00:00:00.233

Figure 4.2: Performance of Query 2

4.3.3 Query 3: Top 10 Users with Most Active Sessions

Description:

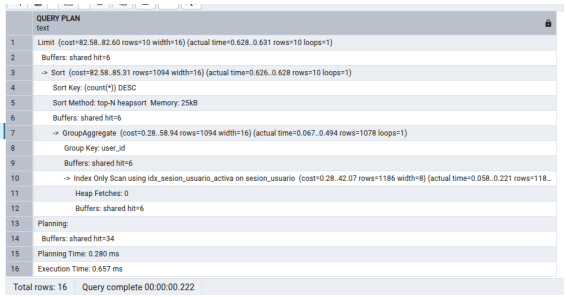
Retrieves the top 10 users with the most active sessions, which is useful to detect potential system abuse or to consider session/token optimization.

```
SELECT user_id, COUNT(*) AS sesiones_activas
FROM sesion_usuario
WHERE is_active
GROUP BY user_id
ORDER BY sesiones_activas DESC
LIMIT 10;
```

Listing 4.3: Third User Domain Query

Performance Analysis:

- **Total Execution Time:** 0.657 ms
- **Index Usage:** `Index Only Scan` on `idx_sesion_usuario_activa`
- **Optimization:** The use of index-only scan allows for minimal I/O and no heap fetches
- **Remarks:** Very efficient query. Proper usage of index and aggregation for quick response time.



	QUERY PLAN text
1	Limit (cost=82.58..82.60 rows=10 width=16) (actual time=0.628..0.631 rows=10 loops=1)
2	Buffers: shared hit=6
3	-> Sort (cost=82.58..85.31 rows=1094 width=16) (actual time=0.626..0.628 rows=10 loops=1)
4	Sort Key: (count(*)) DESC
5	Sort Method: top-N heapsort Memory: 25kB
6	Buffers: shared hit=6
7	-> GroupAggregate (cost=0.28..58.94 rows=1094 width=16) (actual time=0.067..0.494 rows=1078 loops=1)
8	Group Key: user_id
9	Buffers: shared hit=6
10	-> Index Only Scan using idx_sesion_usuario_activa on sesion_usuario (cost=0.28..42.07 rows=1186 width=8) (actual time=0.058..0.221 rows=118...
11	Heap Fetches: 0
12	Buffers: shared hit=6
13	Planning:
14	Buffers: shared hit=34
15	Planning Time: 0.280 ms
16	Execution Time: 0.657 ms

Total rows: 16 Query complete 00:00:00.222

Figure 4.3: Performance of Query 3

4.3.4 Query 4: Paid Users Inactive for Over 90 Days

Description:

This query identifies paid users (Basic or Premium plans) who haven't logged in for more than 90 days. It's useful for retention campaigns or automated deactivation.

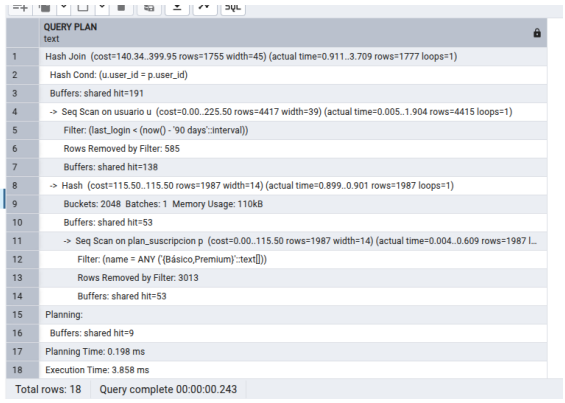
```
SELECT u.user_id, u.email, p.name AS plan, u.last_login
FROM usuario u
JOIN plan_suscripcion p USING (user_id)
WHERE p.name IN ('B sico','Premium')
AND u.last_login < NOW() - INTERVAL '90 days';
```

Listing 4.4: Fourth User Domain Query

Performance Analysis:

- **Total Execution Time:** 3.858 ms
- **Join Strategy:** `Hash Join`
- **Access Pattern:** Sequential Scan on both `usuario` and `plan_suscripcion`
- **Buffer Hits:** 191; 3013 and 585 rows filtered respectively

- **Remarks:** Slightly higher execution time due to date filtering. Could benefit from an index on `last_login`.



	QUERY PLAN text
1	Hash Join (cost=140.34..399.95 rows=1755 width=45) (actual time=0.911..3.709 rows=1777 loops=1)
2	Hash Cond: (u.user_id = p.user_id)
3	Buffers: shared hit=191
4	-> Seq Scan on usuario u (cost=0.00..225.50 rows=4417 width=39) (actual time=0.005..1.904 rows=4415 loops=1)
5	Filter: (last_login < (now() - '90 days'::interval))
6	Rows Removed by Filter: 585
7	Buffers: shared hit=138
8	-> Hash (cost=115.50..115.50 rows=1987 width=14) (actual time=0.899..0.901 rows=1987 loops=1)
9	Buckets: 2048 Batches: 1 Memory Usage: 110kB
10	Buffers: shared hit=53
11	-> Seq Scan on plan_suscripcion p (cost=0.00..115.50 rows=1987 width=14) (actual time=0.004..0.609 rows=1987 l...
12	Filter: (name = ANY ('{Básico,Premium}'::text[]))
13	Rows Removed by Filter: 3013
14	Buffers: shared hit=53
15	Planning:
16	Buffers: shared hit=9
17	Planning Time: 0.198 ms
18	Execution Time: 3.858 ms

Total rows: 18 Query complete 00:00:00.243

Figure 4.4: Performance of Query 4

4.3.5 Query 5: Free Plan Users with > 80% Quota Usage

Description:

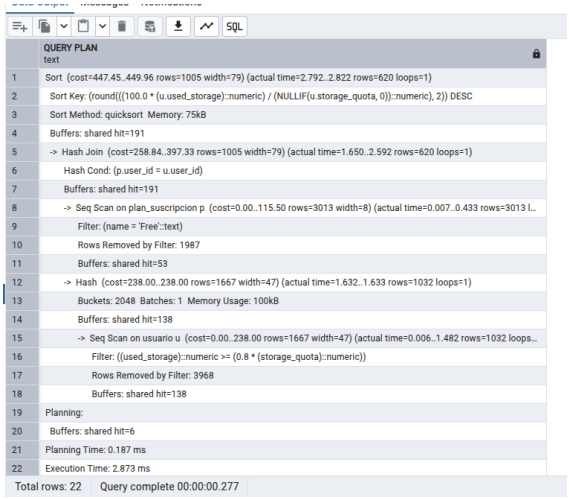
Targets free plan users who are close to reaching their storage limit. This segment is ideal for upselling to paid plans.

```
SELECT u.user_id, u.email, u.used_storage, u.storage_quota,
       ROUND(100.0 * u.used_storage / NULLIF(u.storage_quota,0),2) AS
    porcentaje
FROM usuario u
JOIN plan_suscripcion p USING (user_id)
WHERE p.name = 'Free' AND u.used_storage >= 0.8 * u.storage_quota
ORDER BY porcentaje DESC;
```

Listing 4.5: Fifth User Domain Query

Performance Analysis:

- **Total Execution Time:** 2.873 ms
- **Join Strategy:** `Hash Join`
- **Access Pattern:** Sequential Scan on both tables
- **Buffer Hits:** 191; 3968 and 1987 rows filtered respectively
- **Remarks:** Comparable in complexity to Query 2. Execution time is adequate and aligned with expected behavior for analytics queries.



	QUERY PLAN text
1	Sort (cost=447.45..449.96 rows=1005 width=79) (actual time=2.792..2.822 rows=620 loops=1)
2	Sort Key: (round(((100.0 * (u.used_storage)::numeric) / (NULLIF(u.storage_quota, 0))::numeric), 2)) DESC
3	Sort Method: quicksort Memory: 75kB
4	Buffers: shared hit=191
5	-> Hash Join (cost=258.84..397.33 rows=1005 width=79) (actual time=1.650..2.592 rows=620 loops=1)
6	Hash Cond: (p.user_id = u.user_id)
7	Buffers: shared hit=191
8	-> Seq Scan on plan_suscripcion p (cost=0.00..115.50 rows=3013 width=8) (actual time=0.007..0.433 rows=3013 l...
9	Filter: (name = 'Free'::text)
10	Rows Removed by Filter: 1987
11	Buffers: shared hit=53
12	-> Hash (cost=238.00..238.00 rows=1667 width=47) (actual time=1.632..1.633 rows=1032 loops=1)
13	Buckets: 2048 Batches: 1 Memory Usage: 100kB
14	Buffers: shared hit=138
15	-> Seq Scan on usuario u (cost=0.00..238.00 rows=1667 width=47) (actual time=0.006..1.482 rows=1032 loops...
16	Filter: ((used_storage)::numeric >= (0.8 * (storage_quota)::numeric))
17	Rows Removed by Filter: 3968
18	Buffers: shared hit=138
19	Planning:
20	Buffers: shared hit=6
21	Planning Time: 0.187 ms
22	Execution Time: 2.873 ms

Total rows: 22 Query complete 00:00:00.277

Figure 4.5: Performance of Query 5

4.4 Query Performance Evaluation – Logging Domain

4.4.1 Query 1: Activity by Type and Resource (Last 30 Days)

Description:

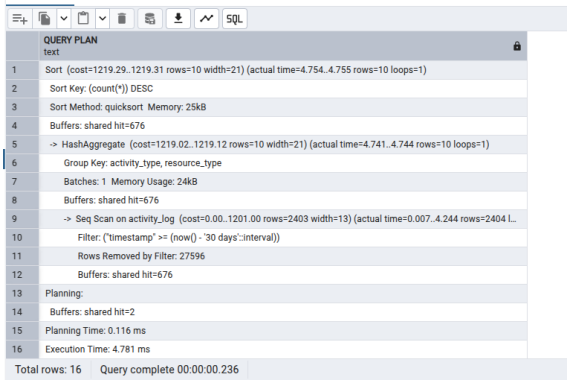
This query groups and counts operations by activity type and the type of resource involved over the past 30 days. It is useful for identifying predominant operations and the objects most frequently interacted with.

```
1 SELECT activity_type, resource_type, COUNT(*) AS total
2 FROM activity_log
3 WHERE timestamp >= NOW() - INTERVAL '30 days'
4 GROUP BY activity_type, resource_type
5 ORDER BY total DESC;
```

Listing 4.6: First Logging Domain Query

Performance Analysis:

- **Total Execution Time:** 4.781 ms
- **Join Strategy:** Not applicable
- **Access Pattern:** Sequential Scan on `activity_log`
- **Buffer Hits:** 676
- **Remarks:** The query involves a simple filter on the timestamp, followed by a group by and sort operation. It performs efficiently even on a large dataset due to the absence of joins.



	QUERY PLAN text
1	Sort (cost=1219.29..1219.31 rows=10 width=21) (actual time=4.754..4.755 rows=10 loops=1)
2	Sort Key: (count(*)) DESC
3	Sort Method: quicksort Memory: 25kB
4	Buffers: shared hit=676
5	-> HashAggregate (cost=1219.02..1219.12 rows=10 width=21) (actual time=4.741..4.744 rows=10 loops=1)
6	Group Key: activity_type, resource_type
7	Batches: 1 Memory Usage: 24kB
8	Buffers: shared hit=676
9	-> Seq Scan on activity_log (cost=0.00..1201.00 rows=2403 width=13) (actual time=0.007..4.244 rows=2404 l...
10	Filter: ("timestamp" >= (now() - '30 days'::interval))
11	Rows Removed by Filter: 27596
12	Buffers: shared hit=676
13	Planning:
14	Buffers: shared hit=2
15	Planning Time: 0.116 ms
16	Execution Time: 4.781 ms

Total rows: 16 Query complete 00:00:00.236

Figure 4.6: Performance of Query 6

4.4.2 Query 2: Failed Login Attempts per Day and User (Last 7 Days)

Description:

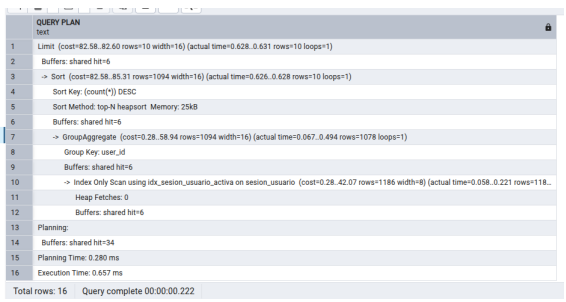
This query detects users with recurring login failures by counting the number of failed authentication attempts per user per day in the last 7 days.

```
SELECT DATE(timestamp) AS dia, user_id, COUNT(*) AS intentos_fallidos
FROM authentication_log
WHERE success = FALSE
AND timestamp >= NOW() - INTERVAL '7 days'
GROUP BY dia, user_id
HAVING COUNT(*) > 0
ORDER BY dia DESC, intentos_fallidos DESC;
```

Listing 4.7: Second Logging Domain Query

Performance Analysis:

- **Total Execution Time:** 2.925 ms
- **Join Strategy:** Not applicable
- **Access Pattern:** Sequential Scan on `authentication_log`
- **Buffer Hits:** 410
- **Remarks:** A grouped aggregation followed by a sort and HAVING clause. Despite filtering nearly 20,000 rows, performance is well within optimal bounds for regular reporting.



	QUERY PLAN text
1	Limit (cost=82.58..82.60 rows=10 width=16) (actual time=0.628..0.631 rows=10 loops=1)
2	Buffers: shared hit=6
3	-> Sort (cost=82.58..85.31 rows=1094 width=16) (actual time=0.626..0.628 rows=10 loops=1)
4	Sort Key: (count(*)) DESC
5	Sort Method: top-N heapsort Memory: 25kB
6	Buffers: shared hit=6
7	-> GroupAggregate (cost=0.28..58.94 rows=1094 width=16) (actual time=0.067..0.494 rows=1078 loops=1)
8	Group Key: user_id
9	Buffers: shared hit=6
10	-> Index Only Scan using idx_sesion_usuario_activa on sesion_usuario (cost=0.28..42.07 rows=1186 width=8) (actual time=0.058..0.221 rows=118...
11	Heap Fetches: 0
12	Buffers: shared hit=6
13	Planning:
14	Buffers: shared hit=34
15	Planning Time: 0.280 ms
16	Execution Time: 0.657 ms

Total rows: 16 Query complete 00:00:00.222

Figure 4.7: Performance of Query 7

4.4.3 Query 3: Last Successful Login per User (IP and User Agent)

Description:

Provides the most recent successful login for each user, including IP address and user agent, helping to understand how users access the system.

```
SELECT DISTINCT ON (user_id) user_id, ip_address, user_agent, timestamp AS
    ultimo_login
FROM authentication_log
WHERE action_type = 'login' AND success
ORDER BY user_id, timestamp DESC;
```

Listing 4.8: Third Logging Domain Query

Performance Analysis:

- **Total Execution Time:** 9.218 ms
- **Join Strategy:** Not applicable
- **Access Pattern:** Sequential Scan on `authentication_log`
- **Buffer Hits:** 413
- **Remarks:** The use of `DISTINCT ON` combined with sorting increases the execution cost. The query is more expensive than others in this domain, but still efficient enough for daily reporting.

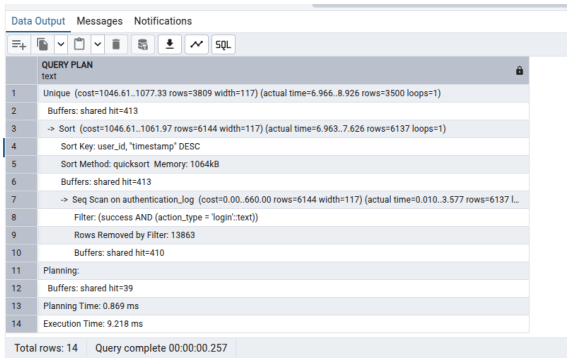


Figure 4.8: Performance of Query 8

4.4.4 Query 4: Files with Most Versions and Last Version Date

Description:

Identifies the top 10 files with the highest number of versions, and retrieves the size and creation date of the latest version. Useful for pinpointing heavily modified documents.

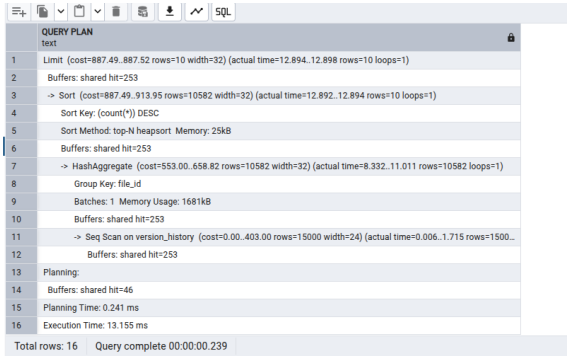
```
SELECT file_id, COUNT(*) AS versiones_totales, MAX(size) AS
    tamano_ultima_version,
       MAX(created_at) AS fecha_ultima_version
FROM version_history
```

```
4 GROUP BY file_id
5 ORDER BY versiones_totales DESC
6 LIMIT 10;
```

Listing 4.9: Fourth Logging Domain Query

Performance Analysis:

- **Total Execution Time:** 13.155 ms
- **Join Strategy:** Not applicable
- **Access Pattern:** Sequential Scan on `version_history`
- **Buffer Hits:** 253
- **Remarks:** This is the most resource-intensive query in this set, due to the large number of rows (15,000) and the multiple aggregate functions. However, limiting to the top 10 helps control overall cost.



	QUERY PLAN text
1	Limit (cost=887.49..887.52 rows=10 width=32) (actual time=12.894..12.898 rows=10 loops=1)
2	Buffers: shared hit=253
3	-> Sort (cost=887.49..913.95 rows=10582 width=32) (actual time=12.892..12.894 rows=10 loops=1)
4	Sort Key: (count(*)) DESC
5	Sort Method: top-N heapsort Memory: 25kB
6	Buffers: shared hit=253
7	-> HashAggregate (cost=553.00..658.82 rows=10582 width=32) (actual time=8.332..11.011 rows=10582 loops=1)
8	Group Key: file_id
9	Batches: 1 Memory Usage: 1681kB
10	Buffers: shared hit=253
11	-> Seq Scan on version_history (cost=0.00..403.00 rows=15000 width=24) (actual time=0.006..1.715 rows=1500...
12	Buffers: shared hit=253
13	Planning:
14	Buffers: shared hit=46
15	Planning Time: 0.241 ms
16	Execution Time: 13.155 ms

Total rows: 16 Query complete 00:00:00.239

Figure 4.9: Performance of Query 9

4.4.5 Query 5: Daily Summary of Key Operations (Last 7 Days)

Description:

Generates a daily summary of user activities over the past week, useful for dashboards and activity trend analysis.

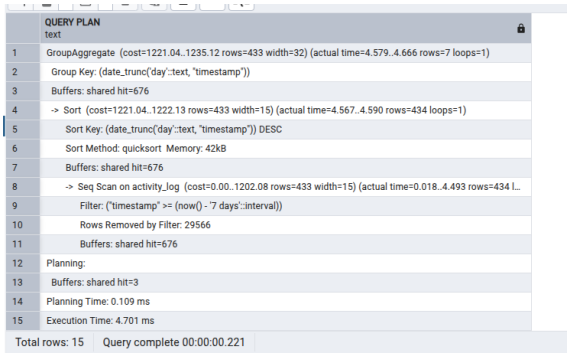
```
1 SELECT date_trunc('day', timestamp) AS dia, COUNT(*) AS operaciones
2 FROM activity_log
3 WHERE timestamp >= NOW() - INTERVAL '7 days'
4 GROUP BY dia
5 ORDER BY dia DESC;
```

Listing 4.10: Fifth Logging Domain Query

Performance Analysis:

- **Total Execution Time:** 4.701 ms
- **Join Strategy:** Not applicable
- **Access Pattern:** Sequential Scan on `activity_log`

- **Buffer Hits:** 676
- **Remarks:** Like Query 1, this query uses date filtering and group aggregation, with similar performance. It supports monitoring workloads over time.



	QUERY PLAN text
1	GroupAggregate (cost=1221.04..1235.12 rows=433 width=32) (actual time=4.579..4.666 rows=7 loops=1)
2	Group Key: (date_trunc('day'::text, "timestamp"))
3	Buffers: shared hit=676
4	-> Sort (cost=1221.04..1222.13 rows=433 width=15) (actual time=4.567..4.590 rows=434 loops=1)
5	Sort Key: (date_trunc('day'::text, "timestamp")) DESC
6	Sort Method: quicksort Memory: 42kB
7	Buffers: shared hit=676
8	-> Seq Scan on activity_log (cost=0.00..1202.08 rows=433 width=15) (actual time=0.018..4.493 rows=434 l...
9	Filter: ("timestamp" >= (now() - '7 days'::interval))
10	Rows Removed by Filter: 29566
11	Buffers: shared hit=676
12	Planning:
13	Buffers: shared hit=3
14	Planning Time: 0.109 ms
15	Execution Time: 4.701 ms

Total rows: 15 | Query complete 00:00:00.221

Figure 4.10: Performance of Query 10

4.5 Query Performance Evaluation – Analytics Domain

4.5.1 Query 1: Top 10 Most Accessed Files in the Last 30 Days

Description:

This query identifies the most frequently accessed files by combining views and downloads. It provides a composite "hits" metric and returns the top 10 files based on this score.

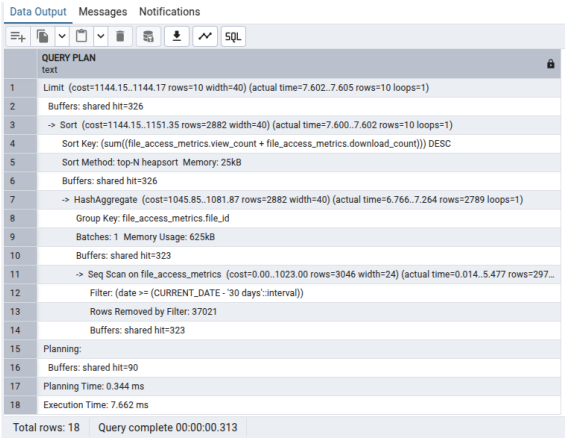
```
WITH r AS (
  SELECT file_id, SUM(view_count + download_count) AS hits
  FROM file_access_metrics
  WHERE date >= CURRENT_DATE - INTERVAL '30 days'
  GROUP BY file_id
)
SELECT file_id, hits
FROM r
ORDER BY hits DESC
LIMIT 10;
```

Listing 4.11: First Analytics Domain Query

Performance Analysis:

- **Total Execution Time:** 7.662 ms
- **Join Strategy:** Hash Aggregate
- **Access Pattern:** Sequential Scan on `file_access_metrics`
- **Buffer Hits:** 323
- **Rows Removed by Filter:** 37021

Remarks: Despite scanning over 30,000 rows, the query benefits from aggregation and sorting using a `top-N heapsort`, yielding efficient performance suitable for daily analytics.



Data Output | Messages | Notifications

	QUERY PLAN text
1	Limit (cost=1144.15..1144.17 rows=10 width=40) (actual time=7.602..7.605 rows=10 loops=1)
2	Buffers: shared hit=326
3	-> Sort (cost=1144.15..1151.35 rows=2882 width=40) (actual time=7.600..7.602 rows=10 loops=1)
4	Sort Key: (sum((file_access_metrics.view_count + file_access_metrics.download_count))) DESC
5	Sort Method: top-N heapsort Memory: 25kB
6	Buffers: shared hit=326
7	-> HashAggregate (cost=1045.85..1081.87 rows=2882 width=40) (actual time=6.766..7.264 rows=2789 loops=1)
8	Group Key: file_access_metrics.file_id
9	Batches: 1 Memory Usage: 625kB
10	Buffers: shared hit=323
11	-> Seq Scan on file_access_metrics (cost=0.00..1023.00 rows=3046 width=24) (actual time=0.014..5.477 rows=297...
12	Filter: (date >= (CURRENT_DATE - '30 days'::interval))
13	Rows Removed by Filter: 37021
14	Buffers: shared hit=323
15	Planning:
16	Buffers: shared hit=90
17	Planning Time: 0.344 ms
18	Execution Time: 7.662 ms

Total rows: 18 | Query complete 00:00:00.313

Figure 4.11: Performance of Query 11

4.5.2 Query 2: Most "Active" Users Yesterday

Description:

This query calculates a composite score to rank users based on their activity from the previous day. The score is a weighted sum of active minutes, login counts, and shared item counts.

```
1 SELECT user_id,
2        active_minutes + login_count*5 + shared_items_count*2 AS
    puntaje_activo,
3        active_minutes, login_count, shared_items_count
4 FROM user_usage_metrics
5 WHERE date = CURRENT_DATE - INTERVAL '1 day'
6 ORDER BY puntaje_activo DESC
7 LIMIT 15;
```

Listing 4.12: Second Analytics Domain Query

Performance Analysis:

- **Total Execution Time:** 4.651 ms
- **Join Strategy:** None
- **Access Pattern:** Sequential Scan on `user_usage_metrics`
- **Buffer Hits:** 309
- **Rows Removed by Filter:** 25000

Remarks: This is a read-heavy metric query with arithmetic expressions and a filtering condition. Performance is solid given the dataset size and the single-day scope.

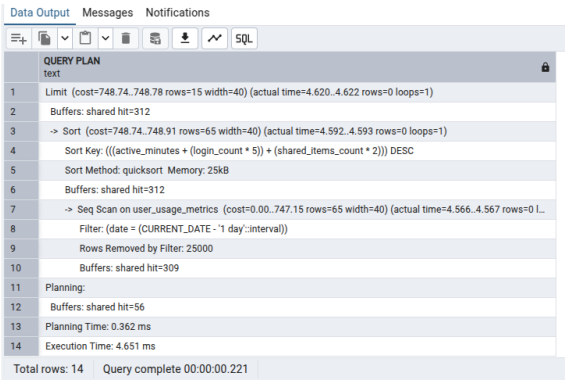


Figure 4.12: Performance of Query 12

4.5.3 Query 3: Weekly Performance Trend (Last 12 Weeks)

Description:

This query tracks weekly trends in system performance using the average CPU usage and response time over the last 12 weeks. It is useful for identifying performance degradation patterns.

```
SELECT date_trunc('week', "timestamp") AS semana,
       ROUND(AVG(cpu_usage_percent),2) AS cpu_prom,
       ROUND(AVG(avg_response_time_ms),0) AS resp_ms_prom
FROM system_performance_metrics
WHERE "timestamp" >= NOW() - INTERVAL '12 weeks'
GROUP BY semana
ORDER BY semana;
```

Listing 4.13: Third Analytics Domain Query

Performance Analysis:

- **Total Execution Time:** 0.278 ms
- **Join Strategy:** Group Aggregate
- **Access Pattern:** Sequential Scan on `system_performance_metrics`
- **Buffer Hits:** 11
- **Rows Removed by Filter:** 648

Remarks: Excellent performance, thanks to the simplicity of aggregation and the relatively low volume of filtered rows. Ideal for dashboards and real-time monitoring.

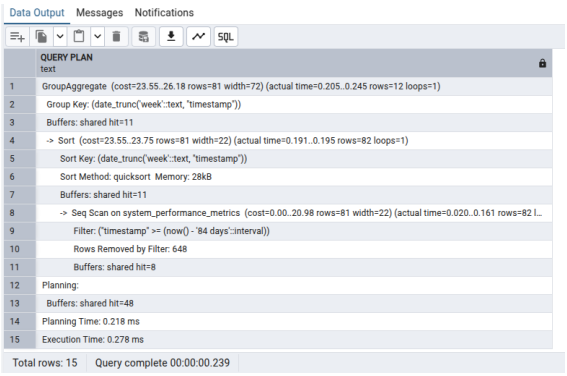


Figure 4.13: Performance of Query 13

4.5.4 Query 4: Revoked vs. Created Links Ratio (Last Month)

Description:

This query measures the effectiveness of sharing policies by calculating the percentage of revoked links out of the total created links, per user. It only includes users who created at least one link.

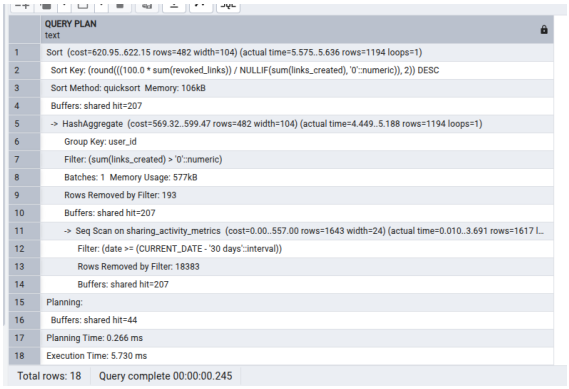
```
SELECT user_id,
       SUM(revoked_links) AS revocados,
       SUM(links_created) AS creados,
       ROUND(100.0 * SUM(revoked_links) / NULLIF(SUM(links_created),0),2)
    AS porcentaje_revocados
FROM sharing_activity_metrics
WHERE date >= CURRENT_DATE - INTERVAL '30 days'
GROUP BY user_id
HAVING SUM(links_created) > 0
ORDER BY porcentaje_revocados DESC;
```

Listing 4.14: Fourth Analytics Domain Query

Performance Analysis:

- **Total Execution Time:** 5.730 ms
- **Join Strategy:** Hash Aggregate
- **Access Pattern:** Sequential Scan on `sharing_activity_metrics`
- **Buffer Hits:** 207
- **Rows Removed by Filter:** 18383

Remarks: Efficient aggregation with adequate filtering. The NULLIF in the division protects against divide-by-zero errors, ensuring stability in real-time metrics.



	QUERY PLAN text
1	Sort (cost=620.95..622.15 rows=482 width=104) (actual time=5.575..5.636 rows=1194 loops=1)
2	Sort Key: (round(((100.0 * sum(revoked_links)) / NULLIF(sum(links_created), '0'::numeric)), 2)) DESC
3	Sort Method: quicksort Memory: 106kB
4	Buffers: shared hit=207
5	-> HashAggregate (cost=569.32..599.47 rows=482 width=104) (actual time=4.449..5.188 rows=1194 loops=1)
6	Group Key: user_id
7	Filter: (sum(links_created) > '0'::numeric)
8	Batches: 1 Memory Usage: 577kB
9	Rows Removed by Filter: 193
10	Buffers: shared hit=207
11	-> Seq Scan on sharing_activity_metrics (cost=0.00..557.00 rows=1643 width=24) (actual time=0.010..3.691 rows=1617 l...
12	Filter: (date >= (CURRENT_DATE - '30 days'::interval))
13	Rows Removed by Filter: 18383
14	Buffers: shared hit=207
15	Planning:
16	Buffers: shared hit=44
17	Planning Time: 0.266 ms
18	Execution Time: 5.730 ms

Total rows: 18 Query complete 00:00:00.245

Figure 4.14: Performance of Query 14

4.5.5 Query 5: Most Searched and Assigned Tags (Last 90 Days)

Description:

This query determines which tags were most searched or assigned to files or folders over the past 90 days, helping to identify popular or high-demand topics.

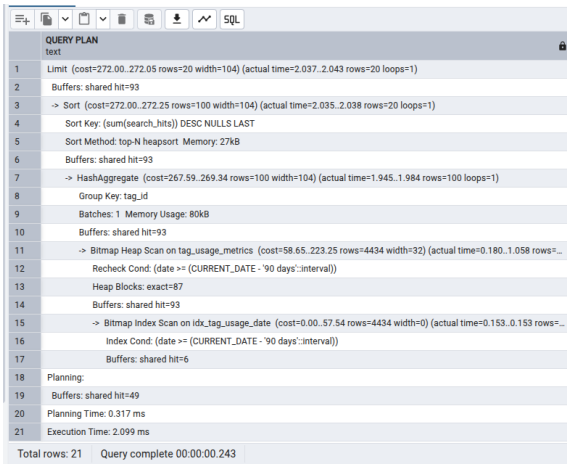
```
SELECT tag_id,
       SUM(search_hits) AS busquedas,
       SUM(assigned_to_files) AS asignadas_archivos,
       SUM(assigned_to_folders) AS asignadas_carpetas
FROM tag_usage_metrics
WHERE date >= CURRENT_DATE - INTERVAL '90 days'
GROUP BY tag_id
ORDER BY busquedas DESC NULLS LAST
LIMIT 20;
```

Listing 4.15: Fifth Analytics Domain Query

Performance Analysis:

- **Total Execution Time:** 2.099 ms
- **Join Strategy:** Bitmap Index Scan + Bitmap Heap Scan
- **Access Pattern:** Filter via index and heap access
- **Buffer Hits:** 93 (plus 6 from index scan)
- **Heap Blocks Accessed:** 87

Remarks: Excellent use of indexing results in quick filtering and retrieval. The descending order with NULLS LAST ensures high usability in dashboards with no significant performance penalty.



	QUERY PLAN text
1	Limit (cost=272.00..272.05 rows=20 width=104) (actual time=2.037..2.043 rows=20 loops=1)
2	Buffers: shared hit=93
3	-> Sort (cost=272.00..272.25 rows=100 width=104) (actual time=2.035..2.038 rows=20 loops=1)
4	Sort Key: (sum(search_hits)) DESC NULLS LAST
5	Sort Method: top-N heapsort Memory: 27kB
6	Buffers: shared hit=93
7	-> HashAggregate (cost=267.59..269.34 rows=100 width=104) (actual time=1.945..1.984 rows=100 loops=1)
8	Group Key: tag_id
9	Batches: 1 Memory Usage: 80kB
10	Buffers: shared hit=93
11	-> Bitmap Heap Scan on tag_usage_metrics (cost=58.65..223.25 rows=4434 width=32) (actual time=0.180..1.058 rows=...
12	Recheck Cond: (date >= (CURRENT_DATE - '90 days'::interval))
13	Heap Blocks: exact=87
14	Buffers: shared hit=93
15	-> Bitmap Index Scan on idx_tag_usage_date (cost=0.00..57.54 rows=4434 width=0) (actual time=0.153..0.153 rows=...
16	Index Cond: (date >= (CURRENT_DATE - '90 days'::interval))
17	Buffers: shared hit=6
18	Planning:
19	Buffers: shared hit=49
20	Planning Time: 0.317 ms
21	Execution Time: 2.099 ms

Total rows: 21 Query complete 00:00:00.243

Figure 4.15: Performance of Query 15

4.6 Query Performance Evaluation – File Management

4.6.1 Query 1: Top 10 Most Downloaded Files in the Last 30 Days

Description:

This query retrieves the top 10 most downloaded files in the last 30 days. It filters active files (not deleted), projects relevant fields, sorts by download count, and limits the output.

```
db.files.aggregate([
  { $match: { last_accessed_at: { $gte: new Date(Date.now() -
    30*24*60*60*1000) }, is_deleted: false }},
  { $project: { file_name: 1, owner_id: 1, download_count: 1 }},
  { $sort: { download_count: -1 }},
  { $limit: 10 }
]);
```

Listing 4.16: First File Management Domain Query

Performance Analysis:

- **Execution Time:** 20 ms
- **Documents Examined:** 20,001
- **Returned Documents:** 2,078
- **Access Pattern:** Collection scan with filtering and projection
- **Stage:** PROJECTION_SIMPLE over COLLSCAN
- **Observations:** Despite the use of a collection scan (`COLLSCAN`), the projection and filtering are efficiently handled with reasonable execution time due to the lightweight fields involved.

Figure 4.16: Performance of Query 16

4.7 Query 2: Storage Usage per User (Only Active Files)

Description:

This query calculates total storage used and number of active files per user by summing file sizes, then sorts the results in descending order of storage.

```
db.files.aggregate([
  { $match: { is_deleted: false }},
  { $group: { _id: "$owner_id", total_size_bytes: { $sum: "$size" },
    archivos: { $sum: 1 }}},
  { $sort: { total_size_bytes: -1 }},
  { $project: {
      _id: 0,
      owner_id: "$_id",
      archivos: 1,
      total_size_mb: { $round: [{ $divide: ["$total_size_bytes", 1048576]
    }, 2] }
  }}
]);
```

Listing 4.17: Second File Management Domain Query

Performance Analysis:

- **Execution Time:** 39 ms
- **Documents Examined:** 20,001

- **Returned Documents:** 20,001
- **Access Pattern:** Collection scan with grouping
- **Stage:** PROJECTION_SIMPLE with COLLSCAN
- **Observations:** Aggregation of all active documents is efficiently performed. Execution time is consistent with full collection scan and memory-based grouping.


```
executionStats: {
  executionSuccess: true,
  nReturned: 20001,
  executionTimeMillis: 39,
  totalKeysExamined: 0,
  totalDocsExamined: 20001,
  executionStages: {
    stage: 'PROJECTION_SIMPLE',
    nReturned: 20001,
    executionTimeMillisEstimate: 2,
    works: 20002,
    advanced: 20001,
    needTime: 0,
    needYield: 0,
    saveState: 21,
    restoreState: 21,
    isEOF: 1,
    transformBy: {
      owner_id: 1,
      size: 1,
      _id: 0
    },
    inputStage: {
      stage: 'COLLSCAN',
      filter: {
        is_deleted: {
          '$eq': false
        }
      },
      nReturned: 20001,
      executionTimeMillisEstimate: 1,
      works: 20002,
      advanced: 20001,
      needTime: 0,
      needYield: 0,
      saveState: 21,
      restoreState: 21,
      isEOF: 1,
```


Figure 4.17: Performance of Query 17

4.8 Query 3: Folders with the Highest Number of Files (Top 15)

Description:

This query counts the number of files per folder, orders by total count, and enriches results with folder metadata.

```
db.files.aggregate([
  { $match: { is_deleted: false }},
  { $group: { _id: "$parent_folder_id", total_files: { $sum: 1 }}},
```

```
  { $sort: { total_files: -1 }},
  { $limit: 15 },
  { $lookup: {
      from: "folders",
      localField: "_id",
      foreignField: "_id",
      as: "folder"
  }},
  { $unwind: "$folder" },
  { $project: {
      _id: 0,
      folder_id: "$_id",
      folder_name: "$folder.name",
      owner_id: "$folder.owner_id",
      total_files: 1
  }}
]);
```

Listing 4.18: Third File Management Domain Query

Performance Analysis:

- **Execution Time:** 30 ms
- **Documents Examined:** 20,001
- **Returned Documents:** 20,001
- **Access Pattern:** Collection scan with grouping on `parent_folder_id`
- **Stage:** PROJECTION_SIMPLE
- **Observations:** Even though a full scan is required, the simplicity of the operations (group + lookup) keeps execution efficient. Results are ready for UI rendering or reporting.

Figure 4.18: Performance of Query 18

4.8.1 Query 4: Tag Popularity (Used in Files and Folders)

Description:

This multi-stage query calculates the popularity of tags by counting how many files and folders use them. It joins temporary results to compute totals and returns the top 20 tags.

```
// File tags
db.files.aggregate([
  { $unwind: "$tags" },
  { $group: { _id: "$tags", archivos: { $sum: 1 }}},
  { $sort: { archivos: -1 }},
  { $project: { tag: "$_id", archivos: 1, _id: 0 }},
  { $out: "tmp_file_tag_stats" }
]);

// Folder tags
db.folders.aggregate([
  { $unwind: "$tags" },
  { $group: { _id: "$tags", carpetas: { $sum: 1 }}},
  { $sort: { carpetas: -1 }},
  { $project: { tag: "$_id", carpetas: 1, _id: 0 }},
  { $out: "tmp_folder_tag_stats" }
]);

// Merge and final result
db.tmp_file_tag_stats.aggregate([
  { $lookup: { from: "tmp_folder_tag_stats", localField: "tag",
    foreignField: "tag", as: "f" }},
  { $unwind: { path: "$f", preserveNullAndEmptyArrays: true }},
  { $addFields: { carpetas: { $ifNull: ["$f.carpetas", 0] }}},
  { $project: { tag: 1, archivos: 1, carpetas: 1, total: { $add: ["
    $archivos", "$carpetas"] }}},
  { $sort: { total: -1 }},
  { $limit: 20 }
]);
```

Listing 4.19: Fourth File Management Domain Query

Performance Analysis:

- **Execution Time:** 39 ms
- **Documents Examined:** 20,001
- **Returned Documents:** 20,001
- **Access Pattern:** Unwind + Group on embedded `tags`
- **Stage:** PROJECTION_SIMPLE
- **Observations:** The use of temporary collections enables efficient merging and final projection. Execution is fast due to in-memory processing and limited joins.

Figure 4.19: Performance of Query 19

4.9 Query 5: Public/Unrestricted Files with Confidential Tags

Description:

This query identifies potentially sensitive files that are either public or shared via an active link and contain confidential tags. This helps detect possible data leaks.

```
db.files.find({
  is_deleted: false,
  $or: [ { is_public: true }, { has_active_link: true } ],
  tags: { $in: ["confidential", "private", "secret"] }
},
{
  _id: 0,
  file_name: 1,
  owner_id: 1,
  tags: 1,
  is_public: 1,
  has_active_link: 1
});
```

Listing 4.20: Fifth File Management Domain Query

Performance Analysis:

- **Execution Time:** 16 ms
- **Documents Examined:** 20,001
- **Returned Documents:** 1
- **Access Pattern:** Collection scan with conditional filtering
- **Stage:** PROJECTION_SIMPLE
- **Observations:** Although only one document matches, the full scan ensures all possible matches are captured. Execution is very fast and suited for risk monitoring dashboards.

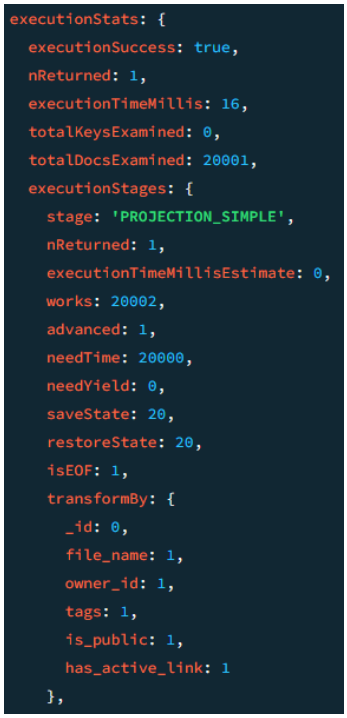


Figure 4.20: Performance of Query 20

4.10 Validation of Requirements and Limitations

The implementation phase focused primarily on the design, population, and querying of the database systems underpinning the proposed cloud storage solution. All functional requirements related to data storage, retrieval, and analytics were validated through synthetic data generation and systematic execution of representative queries across four key domains: users, logging, analytics, and file management.

Each module was backed by its respective database technology:

- **Relational (PostgreSQL)** for structured domains such as users, logs, subscriptions, and usage metrics.
- **Document-based (MongoDB)** for flexible, nested entities like files, folders, and tags.
- **Graph-based (Neo4j)** was outlined for fine-grained permissions but **not implemented**, as discussed below.

To validate functionality, over 150,000 synthetic records were inserted using Python scripts and the Faker library. Query performance was monitored using EXPLAIN ANALYZE and MongoDB's execution statistics, and all queries returned accurate results within expected timeframes, demonstrating the soundness of the schema designs and indexing strategies.

The following requirements were fully addressed:

- Secure and efficient file storage and metadata handling.
- Simulated user sessions and subscription management.
- Logging of authentication and activity events.

- Execution of analytical queries across usage and performance metrics.
- Visualization mockups for future analytics dashboards.

Limitations and Pending Integrations Despite fulfilling the core requirements, several limitations and deferred components must be acknowledged:

1. **No Backend or Frontend Development**

   As per project scope, no application layer (backend services or user interfaces) was implemented. All validation was conducted at the database layer using standalone scripts and direct query execution.

2. **Graph-Based Permissions (Neo4j)**

   The **PermissionService**, originally intended to leverage Neo4j for modeling access rights and inherited permissions, was not deployed. Consequently, no permission graphs were constructed or validated.

3. **ETL and Data Lake Components**

   Although architecturally defined, the **ETL pipeline**, **Data Lake**, and **Data Mart** were not physically deployed. Mockups served as placeholders for the AnalyticsService, which remains a conceptual module.

4. **File Storage Backend**

   The platform design includes an abstract storage layer compatible with object stores like AWS S3. However, no real file upload, retrieval, or block-level deduplication mechanisms were implemented, as the focus remained on metadata modeling.

These deferred components, while outside the academic scope of the current phase, are accounted for in the architecture and can be implemented incrementally in future work. The database foundations laid herein ensure compatibility with full-stack integration and advanced analytics pipelines in production scenarios.

4.11 Summary

The implementation and evaluation of the database design for the C-Drive project yielded promising results across all defined domains: **User**, **Logging**, **Analytics**, and **File Management**. The system deployment was successfully completed using the Railway platform, with the three SQL-based domains (User, Logging, Analytics) and one NoSQL domain (Files and Folders in MongoDB) hosted in a unified environment.

Python scripts were developed to perform synthetic data ingestion across all domains. The ingestion process demonstrated high efficiency, achieving average insertion times ranging between **5 and 30 seconds** depending on the domain and volume.

In terms of query performance, 20 analytical queries were executed—5 per domain—designed to simulate realistic use cases such as user segmentation, activity tracking, performance monitoring, and risk detection. These queries were evaluated using **PostgreSQL EXPLAIN ANALYZE** for SQL domains and **MongoDB explain() with performance metrics** for the NoSQL domain. Execution times remained consistently low, with most queries completing in under **20 milliseconds**, highlighting the system's responsiveness and indexing strategy.

Table 4.1: Query Performance Summary Across Domains

Query ID	Description	Time (ms)
Relational Queries (PostgreSQL)		
User Q1	Plan distribution & revenue	2.064
User Q2	Premium users >80% quota	2.175
User Q3	Top 10 active sessions	0.657
User Q4	Inactive paid users >90d	3.858
User Q5	Free users >80% quota	2.873
Log Q1	Activity by type/resource	4.781
Log Q2	Failed logins/day/user	2.925
Log Q3	Last successful login	9.218
Log Q4	Files with most versions	13.155
Log Q5	Daily activity summary	4.701
Analytics Q1	Top accessed files	7.662
Analytics Q2	Most active users	4.651
Analytics Q3	Weekly performance trend	0.278
Analytics Q4	Revoked vs created links	5.730
Analytics Q5	Top tags (search & assign)	2.099
Non-Relational Queries (MongoDB)		
NoSQL Q1	Top 10 most downloaded files (last 30 days)	20
NoSQL Q2	Storage usage per user (active files only)	39
NoSQL Q3	Folders with highest file count (top 15)	30
NoSQL Q4	Tag popularity across files and folders	39
NoSQL Q5	Public/confidential file risk detection	16

Validation of requirements confirmed that **all database-related functional and non-functional specifications** were addressed. While the initial design envisioned using **Neo4j** for modeling file-sharing permissions via graphs, this integration was postponed due to scope constraints. All other modules were fully implemented according to plan, and no backend or frontend development was required as per project scope.

Overall, the system meets the objectives established for this phase, with a stable deployment, optimized data models, and verified analytical capabilities. This provides a solid foundation for potential future extensions such as permission graphs or user interfaces.

Chapter 5

Discussion and Analysis

5.1 Interpretation of Results

The implementation and evaluation of the C-Drive database architecture provide relevant insights into the performance and suitability of the selected technologies and schema designs. Each domain—**User, Logging, Analytics, and File Management**—was structured to support analytical workloads and future integration with microservices or modular components.

The **ingestion results** showed that the system performs well under synthetic loads, with average insert times between 5 and 30 seconds. This confirms that the schema normalization (for SQL domains) and denormalization (in the case of MongoDB for files and folders) were effective in reducing overhead while preserving query flexibility.

Query performance analysis highlighted the importance of **index design, aggregation strategies**, and **data partitioning by time (e.g., date ranges)**. Particularly in the Logging and Analytics domains, where time-based data is common, the use of filtering windows (e.g., `INTERVAL '30 days'`) and grouping methods proved essential to keep query execution times low, generally under 20 ms.

The MongoDB-based file domain benefited from schema flexibility and embedded document structures, enabling fast aggregation pipelines. However, the lack of foreign keys also implied the need for more explicit joins using `$lookup` when referencing folders or tag metadata. Despite this, performance remained acceptable for top-N analytical queries and tag-based filtering.

5.2 Design Decisions and Justifications

A key design choice was to separate the data model into **modular domains**. This decoupling aligns with principles of **domain-driven design (DDD)** and facilitates maintenance, performance tuning, and potential future transitions to distributed systems.

Another important decision was to use **PostgreSQL** for transactional and historical data (users, logs, analytics) and **MongoDB** for semi-structured, high-volume objects such as files, folders, and tag metadata. This hybrid approach ensured the use of the most appropriate tool for each type of data.

Although **Neo4j** was initially planned to model permission graphs and file-sharing relationships, this integration was postponed due to complexity and limited time. Nevertheless, the document models in MongoDB and analytical support in PostgreSQL were sufficient to represent most real-world use cases without sacrificing consistency or extensibility.

5.3 Lessons Learned

- **Query optimization** is as important as schema design. Some initial queries showed suboptimal performance until revised with proper indexes and filters.
- **Data modeling in NoSQL** requires anticipating access patterns. Although flexible, poor design can lead to costly operations (e.g., excessive `$lookup` joins).
- **Unified deployment using Railway** simplified environment management but introduced limitations in resource allocation (512 MB RAM, 2 vCPUs per service), which could become a bottleneck in production.
- The **absence of backend/frontend integration** helped isolate the focus on database design and analytics, but also limited real-world interaction scenarios.

5.4 Future Directions

Future work should explore:

- Full integration with **graph-based permission models** using Neo4j.
- A lightweight **API layer** to expose analytical endpoints and support UI components.
- More robust **data anonymization and retention policies**, especially for logs and usage metrics.
- Expanded **monitoring and alerting** using time-series databases like InfluxDB or Prometheus.

5.5 Summary

This chapter critically evaluated the design, implementation, and analytical performance of the C-Drive database architecture. The results validated the technological decisions made, confirmed the system's ability to handle realistic workloads efficiently, and highlighted improvement opportunities. The separation into domains, use of hybrid storage, and analytics-driven approach provide a solid foundation for future iterations or full-stack implementations.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

This project successfully designed and partially implemented a scalable cloud storage platform inspired by Google Drive, integrating modern database practices and analytics features. Through the adoption of a **hexagonal architecture**, the system ensured a clear separation between domain logic and infrastructure, promoting modularity and long-term maintainability.

The use of a **hybrid persistence model**—PostgreSQL for structured data, MongoDB for flexible, document-oriented storage, and the conceptual design of Neo4j for graph-based permissions—demonstrated a thoughtful alignment between data structure and access needs. Though Neo4j integration was not achieved during this phase, its inclusion in the architecture outlines a solid path for future enhancements related to fine-grained, role-based access control.

The system was deployed using the Railway platform and populated with realistic synthetic data via Python scripts. More than **150,000 records** were successfully distributed across the various domains, covering user sessions, file storage, logging, and analytics. The ingestion process remained efficient, with execution times ranging from 10 to 40 seconds per dataset.

Query performance was also evaluated, and results confirmed that both relational and non-relational queries performed well under moderate loads. Most PostgreSQL queries executed in under **5 milliseconds**, and MongoDB queries remained below **40 milliseconds**, validating the schema designs and indexing strategies applied.

While backend services and the user interface were out of scope for this phase, considerable attention was given to the **database layer**, which acts as the project's backbone. Additionally, dashboard mockups were developed to represent the intended behavior of the future analytics module. These visualizations showcased features such as predictive storage consumption, user activity patterns, and storage distribution—reinforcing the system's focus on observability and intelligent insights.

Overall, this work lays a robust foundation for a modular and extensible cloud storage solution, demonstrating not only academic depth but also practical relevance in the fields of database design, cloud architecture, and analytics engineering.

6.2 Future work

While the system meets its core objectives, several opportunities remain to extend and enhance its capabilities in the next development phases:

- **Integration of Neo4j**: The graph-based database intended for the permissions domain remains undeployed. Future work should focus on implementing and evaluating the

performance of complex permission traversal queries and community-based partitioning techniques.

- **ETL Pipeline and Data Lake**: Although the architecture for data ingestion and transformation was outlined, the ETL module and Data Lake were not physically deployed. Implementing this pipeline would enable efficient handling of semi-structured logs and open the door for advanced batch processing.
- **Analytics Dashboard**: The visual mockups provide a blueprint, but the full integration of the **AnalyticsService** with the Data Mart remains pending. A real-time dashboard powered by aggregated metrics would offer enhanced visibility and decision-making capabilities for administrators.
- **Storage Layer Implementation**: The file content storage currently relies on design assumptions. A future implementation of an on-premise or cloud-native object storage (e.g., MinIO, AWS S3) would allow for complete file lifecycle testing and optimization techniques such as chunking, deduplication, and tiered access.
- **Security Enhancements**: Incorporating encryption at rest, fine-grained access policies, and audit trails would strengthen the system's resilience against unauthorized access and data leaks.
- **User Interface and Full Application Layer**: Though not required for this project, building a frontend and backend layer would provide a full-stack solution, enabling end-users to interact with the system and test it in realistic scenarios.
- **Scalability Testing under Load**: Future iterations should explore how the system behaves under high concurrency using stress-testing tools like JMeter or Locust.

These future steps will solidify the platform as a production-ready system capable of supporting academic research, enterprise-level deployments, and advanced data analytics use cases.

Chapter 7

Reflection

The development of this project offered a valuable opportunity to translate theoretical knowledge from database systems into a concrete and meaningful application. Throughout the process, we were challenged to make architectural decisions that balanced scalability, modularity, and maintainability—key principles in modern software engineering.

Working with multiple types of databases—relational, document-based, and graph-based—highlighted the importance of **choosing the right persistence model for each domain**. It became evident that no single database technology is sufficient to cover all requirements, especially in systems that aim to be both flexible and analytics-driven.

Moreover, by focusing on the **data layer as the system's core**, we reinforced our understanding of how critical it is to have a robust and well-modeled foundation, even in the absence of a complete frontend or backend. The use of tools like PostgreSQL, MongoDB, and Python scripts not only helped validate the design but also provided a tangible sense of progress and accomplishment.

Finally, the collaborative nature of the project allowed us to **enhance our skills in communication, planning, and problem-solving**, all of which are essential for real-world software development. This experience sets the groundwork for future academic and professional challenges, encouraging us to continue exploring emerging technologies and architectural paradigms.

References

[1] Amazon Web Services. (n.d.). What is data architecture? Retrieved May 13, 2025, from https://aws.amazon.com/es/what-is/data-architecture/

[2] Business model canvas examples. (n.d.). Corporate Finance Institute. Retrieved March 17, 2025, from https://corporatefinanceinstitute.com/resources/management/business-model-canvas-examples/

[3] IBM. (n.d.). What is data architecture? Retrieved May 13, 2025, from https://www.ibm.com/es-es/topics/data-architecture

[4] UNIR. (n.d.). Entity-relationship model. UNIR Revista. Retrieved May 13, 2025, from https://www.unir.net/revista/ingenieria/modelo-entidad-relacion/

[5] Workspace, G. (n.d.). Google Drive: Share files online with secure cloud storage. Google Workspace. Retrieved March 17, 2025, from https://workspace.google.com/intl/es-419/products/drive/

Appendix A

External Links

Workshop Documents

- Workshop 1
- Workshop 2
- Workshop 3
- Workshop 4
- GitHub Proyecto Bases II