Workshop N° 2 & 3 - Data System Architecture and Performance Strategies

Universidad Distrital Francisco José de Caldas
School of Engineering
Computer Engineering Program

David Stiven Muñoz Amaya - 20202020071

Janeth Oliveros Ramírez - 20182020100
Daniel Felipe Paez Mosquera - 20202020070

November 2025

Introduction

This document presents the data architecture design for the Digital University Library, developed across Workshop 2 and Workshop 3. It defines how the system should store, manage and process information in a consistent, scalable and efficient way, considering both functional requirements and non-functional aspects such as performance, concurrency and availability. The first part revisits and refines the scope from Workshop 2, summarising the business context, key requirements and information needs, and presenting the conceptual data model together with representative queries that link user needs to the database design.

The second part focuses on runtime behaviour and implementation concerns. It first analyses critical concurrency scenarios in both the relational metadata layer and the NoSQL activity-log layer, and then proposes specific solutions based on transaction isolation levels, row-level locking, optimistic concurrency control, MVCC-based versioning and retry/backoff strategies. Finally, it introduces a parallel and distributed database design that combines relational, NoSQL, search and storage components into a polyglot architecture, explains how data is distributed and replicated across nodes, and uses high-level diagrams to illustrate the overall architecture, data placement and parallel execution of search, metadata and recommendation queries.

1 Improvements to Workshop 2

This section summarizes the refinements applied to the previous Workshop 2 submission based on the instructor's feedback. The goal of these improvements is to correct conceptual deviations, reorganize the document structure, and align the analysis with a data–centric perspective rather than an application–centric design. Each improvement explicitly addresses a concrete observation from the instructor and contributes to a clearer, more coherent, and academically consistent data architecture.

1.1 Authentication Scope Clarification

The original version of Workshop 2 included the statement *"authentication with institutional email (SSO or local login)"*. This phrasing introduced ambiguity and mixed application–level concerns with data architecture, which is outside the intended scope of the course.

Improvement applied: The authentication mechanism is now described conceptually, assuming institutional Single Sign-On (SSO) as the identity provider. No technical details regarding login flows, password storage, or authentication protocols are modeled, since these belong to application logic rather than data system design.

User accounts, roles, and academic programs remain in the data model because they are essential for tracking downloads, reviews, suggestions, and analytical reporting. However, the description no longer includes implementation details related to authentication, ensuring the focus remains strictly on the data entities and their relationships.

1.2 Adjustment of Project Scope

The original list of functional requirements included the feature *"Citation Export (APA/MLA/BibTeX)"*, which allowed users to generate bibliographic citations for digital resources. After reviewing the instructor's feedback, this requirement was identified as being outside the intended scope of the course.

Improvement applied: The Citation Export functionality has been removed from the system requirements. Although useful in real academic platforms, it represents an application-level feature that does not contribute to the analysis of data architecture, information flow, performance, or hybrid SQL/NoSQL modeling.

For this reason, citation generation is no longer part of the core system design and may be considered a potential future enhancement rather than a functional requirement of the current project.

1.3 Reordering of User Stories

In the previous version of Workshop 2, the User Stories were placed after the architecture and data design sections, which disrupted the methodological sequence of the document. According to the instructor's feedback, User Stories must appear immediately after the Functional and Non-Functional Requirements, as they define the behavioral expectations of the system and should guide all subsequent design decisions.

Improvement applied: The User Stories have been repositioned to follow directly after the requirements section.

1.4 Conceptual Data System Architecture

Figure 1 presents the revised high-level data system architecture for the Digital University Library. This updated version corrects the issues identified in Workshop 2 by removing application-layer components (front-end, back-end services, controllers, authentication flows) and focusing exclusively on the data-oriented view of the system, as required by the course. The diagram highlights the main data stores, processing components, and the flows of information between them.

The architecture is divided into four conceptual layers:

- **Operational Data Sources:** These represent the points where raw data enters the system. User interactions (searches and downloads), administrative ingestion of metadata and digital resources, and system-generated events act as the primary producers of operational data.
- **Core Data Management Layer:** This layer contains the core operational datasets. The *Metadata Database (Relational)* stores structured information such as resource descriptions, authors, users, roles, and licenses. The *Activity Log Store (NoSQL)* captures high-volume, semi-structured events including searches, downloads, and usage traces. The *Search Index* maintains lightweight copies of resource metadata optimized for fast full-text search. The *Digital Content Storage* holds the binary digital resources (PDF/ePub files) uploaded by administrators.
- **Analytics & BI Layer:** Periodic *Batch Processing Jobs* consolidate and transform data from the relational metadata store and the NoSQL activity logs to generate aggregated analytical datasets. These results populate the *Data Warehouse*, which stores metrics such as top downloads, frequent authors, and usage trends, enabling efficient analytical queries.
- **System Outputs:** The system produces multiple outputs derived from the data architecture: search results served from the Search Index, download records derived from activity logs, and analytics dashboards generated from the Data Warehouse. Together, these outputs support both operational and decision-making workflows.

Overall, the revised architecture separates structured, semi-structured, and binary data while distinguishing operational workloads from analytical workloads. This separation improves scalability, supports hybrid SQL/NoSQL modeling, and provides a clear foundation for concurrency analysis, distributed design, and performance strategies in Workshop 3.

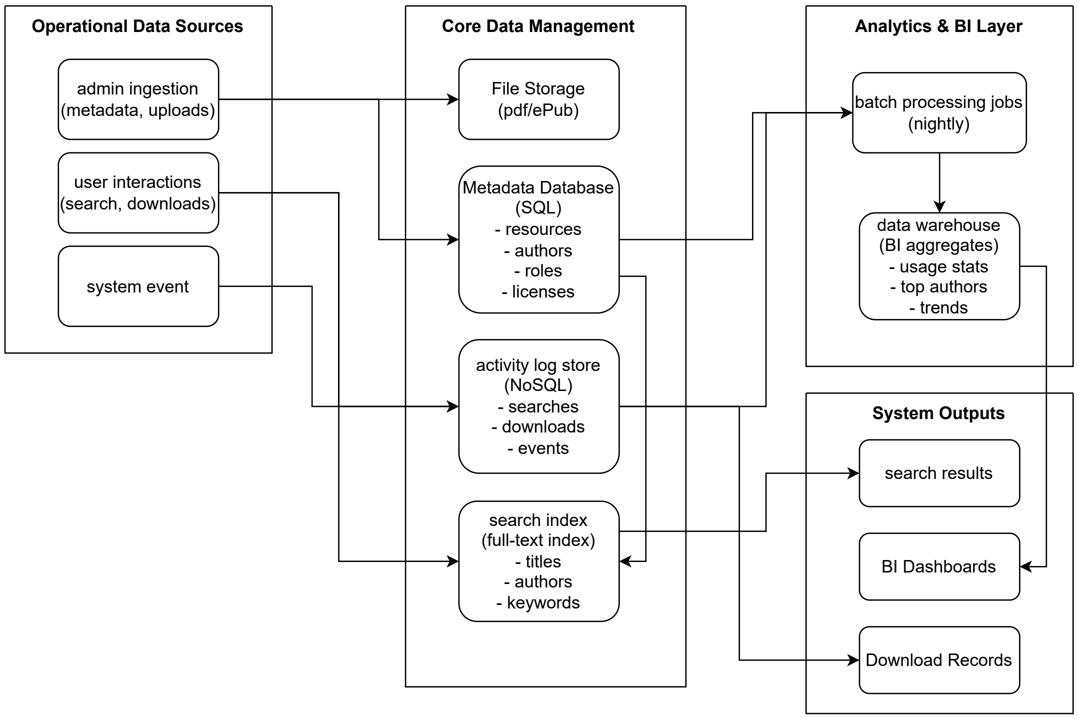

Operational Data Sources
admin ingestion (metadata, uploads)
user interactions (search, downloads)
system event
Core Data Management
File Storage (pdf/ePub)
Metadata Database (SQL) - resources - authors - roles - licenses
activity log store (NoSQL) - searches - downloads - events
search index (full-text index) - titles - authors - keywords
Analytics & BI Layer
batch processing jobs (nightly)
data warehouse (BI aggregates) - usage stats - top authors - trends
System Outputs
search results
BI Dashboards
Download Records

Figure 1: Revised Conceptual Data System Architecture

1.5 Reorganization of Information Requirements

The Information Requirements (IR) defined in the previous version of Workshop 2 mixed functional statements with data needs and did not follow a clear information-oriented structure. Based on the instructor's feedback, the IR have been reorganized to reflect the actual data entities and flows of the system, aligned with the revised architecture.

The new organization separates the information into four conceptual categories according to its structure, purpose, and storage model:

1. **Structured Metadata (Relational).** These requirements cover all information stored in the relational database, including digital resource descriptions (title, authors, publication year, keywords, categories), user and role information, academic programs, and resource licensing. This data supports cataloging, resource management, and operational queries.

2. **Semi-Structured Activity Data (NoSQL).** This category captures high-volume, event-driven data generated by system usage, such as search queries, resource downloads, and user activity traces. These records do not require rigid schemas and are optimized for high write throughput and behavioral analysis.

3. **Binary Digital Content (File Storage).** The system must store PDF and ePub files associated with each resource. These binary files are kept in a dedicated content storage layer, while only their file paths and metadata are stored in the relational database.

4. **Analytical and Aggregated Data (Data Warehouse).** Periodic batch processes generate analytical datasets such as daily download statistics, top authors, usage trends, and program-level activity summaries. These aggregated results are stored in the Data Warehouse to support reporting and decision-making dashboards.

1.6 Information Volume and Frequency Analysis

Following the instructor's feedback, the Information Requirements (IR) have been extended to include volume and frequency estimates, which are essential for characterizing the data needs of the system and for supporting decisions related to SQL/NoSQL storage, indexing, batch processing, and BI workloads. The estimated values reflect typical usage patterns of a university digital library and provide a realistic basis for the architectural decisions made.

1. Structured Metadata (Relational)

- **Volume:** The system is expected to manage between 8,000 and 20,000 digital resources including books, theses, institutional publications, and academic articles. Metadata records typically range from 1–5 KB per resource.
- **Growth Rate:** Approximately 50–150 new resources added per month through administrative ingestion.
- **Frequency of Access:** High read frequency (hundreds of metadata lookups per hour during peak usage), low write frequency (limited to administrative ingestion and corrections).

2. Semi-Structured Activity Data (NoSQL)

- **Volume:** User interactions generate a large number of event records, including searches, downloads, and page views. A medium-size academic system may produce 5,000–20,000 log entries per day, each 0.3–1 KB in size.
- **Growth Rate:** Continuous. Logs accumulate rapidly and can reach millions of entries per semester. Their flexible schema and high write throughput requirements justify the use of NoSQL storage.
- **Frequency:** Extremely high write frequency (several events per second during peak periods) and moderate read frequency (primarily batch reads for analytics).

3. Binary Digital Content (File Storage)

- **Volume:** PDF and ePub files typically range from 2–50 MB depending on resolution and content. Total storage requirements are estimated between 200 GB and 1 TB in a medium-term horizon.
- **Growth Rate:** Follows the rate of new resource ingestion. Approximately 3–10 GB of new content per month.
- **Frequency of Access:** Moderate read frequency (downloads), low write frequency (uploads).

4. Analytical Aggregates (Data Warehouse)

- **Volume:** Aggregated BI tables remain relatively small: hundreds to thousands of rows per reporting period. Each table stores daily, weekly, or monthly summaries such as downloads per program, top authors, or trending keywords.
- **Frequency of Updates:** Updated via nightly batch jobs. Frequency may increase to hourly for metrics that require fresher insights.
- **Frequency of Access:** Read frequently by dashboards and reports, with minimal write load outside the scheduled batch processes.

1.7 Revised Information Flow

Figure 2 presents the revised information flow diagram for the Digital University Library. This new version addresses the instructor's feedback by explicitly modeling the system as a sequence of data processing steps rather than as a static architectural representation. The diagram now distinguishes operational data flows from analytical data flows and incorporates the required transformation steps that were absent in the previous submission.

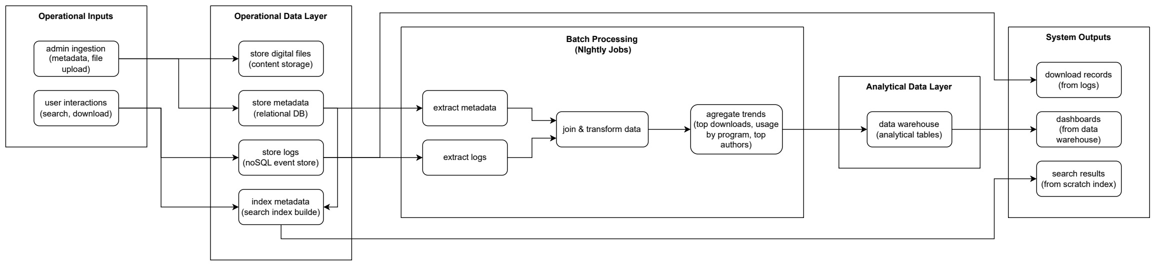

Operational Inputs
admin ingestion (metadata, file upload)
user interactions (search, download)
Operational Data Layer
store digital files (content storage)
store metadata (relational DB)
store logs (noSQL event store)
index metadata (search index builde)
Batch Processing (NIghtly Jobs)
extract metadata
extract logs
join & transform data
agregate trends (top downloads, usage by program, top authors)
Analytical Data Layer
data warehouse (analytical tables)
System Outputs
download records (from logs)
dashboards (from data warehouse)
search results (from scratch index)

Figure 2: Revised Information Flow Diagram for the Digital Library System

1.8 Conceptual Entity–Relationship Model

Figure 3 presents the revised conceptual Entity–Relationship (ER) model for the Digital University Library. This new version corrects the issues from Workshop 2 by removing implementation details such as SQL tables, primary and foreign keys, technical attributes, logs, and analytical tables. The diagram focuses exclusively on the core entities of the domain and the semantic relationships between them, as expected in a proper conceptual model.

1.9 Query Proposals

This section presents conceptual query expressions associated with the main information requirements of the Digital Library System. These queries are not intended as executable SQL or NoSQL statements; instead, they represent abstract operations describing how information would be retrieved from the relational metadata layer or the semi-structured log store.

IR 1: Resource Metadata (Relational Layer)

Query 1: Retrieve resources by category.

```
QUERY getResourcesByCategory(categoryName):
    FROM Resource
    WHERE Resource is classified in Category with name = categoryName
    RETURN Resource.title
```

Purpose: Identifies resources associated with a specific academic category.

Query 2: Retrieve authors of a resource.

```
QUERY getAuthorsOfResource(resourceId):
    FROM Author
    WHERE Author is linked to Resource with id = resourceId
    RETURN Author.name
```

Purpose: Shows the authors who contributed to a given resource.

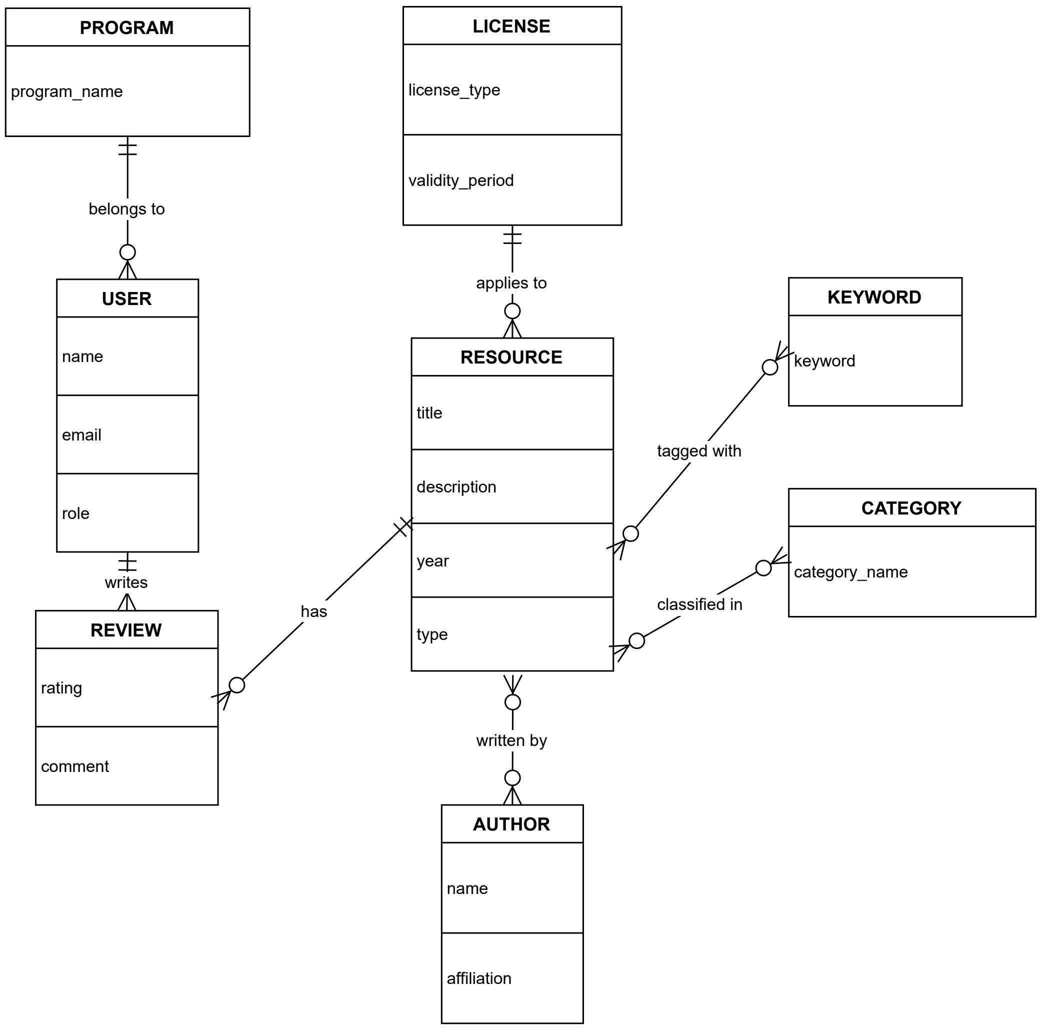


Figure 3: Conceptual Entity–Relationship Model for the Digital Library System

1.10 Synthesis of Improvements

The set of corrections implemented in this revised version of Workshop 2 establish a coherent and academically sound foundation for the development of Workshop 3. The improvements address all structural issues identified in the instructor's feedback, especially the lack of separation between operational, analytical, and domain-level information.

2 Concurrency Analysis

This section identifies the main scenarios in which concurrent access to shared data may occur in the Digital University Library and analyzes the consistency risks associated with each situation.

Scenario 1: Simultaneous Metadata Updates by Administrators

Multiple administrators may edit the metadata of the same resource at the same time. If two updates occur concurrently, one modification may overwrite the other. **Potential anomaly: Lost Update.**

Scenario 2: Multiple Users Submitting Reviews Concurrently

Users may write reviews for the same resource simultaneously. **Potential anomalies: Non-Repeatable Read and Phantom Read.**

Scenario 3: High-Volume Download Events

During peak usage periods (e.g., exam weeks), many users may download the same resource at the same time. **Potential anomaly: Write Skew in Non-Transactional Logs.**

Scenario 4: Concurrent Search Index Updates

When administrators update metadata, the Search Index Builder may run to propagate changes. **Potential anomaly: Non-Repeatable Read.**

Scenario 5: Batch Processing Executed While Logs Are Active

Analytical batch jobs periodically extract metadata and activity logs. **Potential anomaly: Phantom Read.**

Summary

The scenarios above highlight that concurrency issues may arise both in the relational metadata layer and in the semi-structured NoSQL activity log layer. The identification of these anomalies provides the foundation for the concurrency-control strategies.

3 Concurrency Solutions

Based on the scenarios identified in Section 3, the proposed solutions build on the transactional capabilities of PostgreSQL, complemented by application-level mechanisms such as optimistic concurrency control and retry/backoff. For the high-volume, semi-structured activity logs, MongoDB is used with an append-only model and eventual consistency.

Together, these mechanisms aim to mitigate anomalies such as lost updates, non-repeatable reads, phantom reads, and write skew in the five scenarios previously described.

3.1 Overall Concurrency-Control Strategy

At a high level, the platform adopts a *hybrid concurrency-control strategy*:

- **PostgreSQL (relational metadata and users):** Acts as the system of record for critical entities (users, resources, licenses, reviews). It relies on:
  - MVCC (Multi-Version Concurrency Control) to provide consistent snapshots and minimize blocking between readers and writers.
  - Carefully chosen transaction isolation levels depending on the workload.
  - Row-level locking for contended updates on individual resources.
  - Optimistic concurrency control (OCC) with version columns to detect conflicting edits.
- **MongoDB (activity logs and behavioral data):** Stores high-volume, append-only data such as download events and search logs. Its primary goals are write throughput and durability rather than strict transactional isolation. Consistency is managed at the application level, accepting **eventual consistency**.
- **Application layer:** Implements retry and exponential backoff for transient errors, like deadlocks, serialization failures, temporary connectivity issues) to increase robustness under contention.

3.2 Transaction Isolation Levels

PostgreSQL provides several transaction isolation levels. Selecting an appropriate level for each class of operation enables a balance between consistency and performance.

READ COMMITTED. Each statement within a transaction sees only data committed before that statement begins. This level:

- Prevents dirty reads.
- Provides good performance under concurrent workloads.
- Is suitable for:
  - Most catalog browsing and simple CRUD operations on resource metadata.
  - Insertion of reviews by multiple users (Scenario 2), where small timing differences between reads are acceptable.
  - Logging-related updates that touch relational tables.

REPEATABLE READ. A transaction sees a stable snapshot of the database taken at the start of the transaction, thus avoiding many non-repeatable reads and phantoms.

- Recommended for back-office operations that:
  - Read the same set of rows multiple times.
  - Require a consistent view during complex computations.
- For example, analytical or reporting tasks over metadata (Scenario 5) can run under REPEATABLE READ to ensure that calculations are based on a stable snapshot.

SERIALIZABLE. Provides full serializable isolation, guaranteeing that the concurrent execution of transactions is equivalent to some serial order.

- Reserved for rare, highly critical operations, such as complex administrative updates on licenses or access rules where write skew is unacceptable.
- Increases the likelihood of serialization failures; therefore, it is combined with retry/backoff strategies (Section 3.6).

3.3 Locking: Row-Level vs. Table-Level

Isolation levels are complemented by explicit locking in cases where concurrent updates to the same records must be strictly serialized.

Row-level locking (preferred). The system uses row-level locks to coordinate updates on individual resources or aggregates. For example, in Scenario 1 (Simultaneous Metadata Updates by Administrators), an administrator transaction may execute:

```
SELECT *
FROM resource
WHERE resource_id = $1
FOR UPDATE;
```

Followed by one or more `UPDATE` statements on that row. This ensures that:

- Only one transaction at a time can modify the selected resource.
- Other transactions attempting to update the same resource must wait or fail, avoiding lost updates.

Row-level locking can also be applied when updating aggregated counters. For example, number of reviews or downloads, so that increments remain consistent even under concurrent activity.

Table-level locking. Table-level locks are restricted to maintenance or migration operations, like schema changes, one-off data cleaning tasks, where temporarily blocking concurrent writes on the entire table is acceptable. Table-level locks are not used in regular application flows, as they would unnecessarily reduce concurrency and degrade user experience. By favoring short transactions with row-level locks, the platform prevents critical conflicts in Scenario 1 without blocking access to unrelated data.

3.4 Optimistic Concurrency Control and Application-Level Versioning

To prevent silent overwriting of changes in administrative workflows, the platform employsoptimistic concurrency control (OCC) on top of PostgreSQL's MVCC.

In the resource metadata table, each row contains a versioning attribute (e.g., an integer `version` or a timestamp `updated_at`). When an administrator opens an edit form, the current version is included in the form data. The update logic then:

- Issues an `UPDATE` that checks both the primary key and the expected version:

```
UPDATE resource
SET title    = $1,
    abstract = $2,
    keywords = $3,
    version  = version + 1
WHERE resource_id = $4
  AND version     = $5;
```

- Verifies the number of rows affected:
  - If exactly one row is updated, the change is committed as usual.
  - If zero rows are updated, this indicates that another transaction modified the same record first (the version no longer matches). The application:
    - Detects the concurrency conflict.
    - Notifies the administrator that the resource has been updated by someone else.
    - Requests a reload or a manual merge of changes.

This strategy is particularly effective for Scenario 1, where multiple administrators may attempt to update the same metadata concurrently. OCC is well-suited to these workloads because conflicts are relatively infrequent, but correctness is essential when they do occur.

3.5 MVCC-Based Versioning in PostgreSQL

PostgreSQL internally implements MVCC, which creates multiple versions of a row as updates occur. Each transaction sees a consistent snapshot of the database based on its start time.

The system leverages MVCC in two main ways:

- **Non-blocking reads under concurrent updates:** Regular users browsing the catalog or reading resource details can do so without blocking ongoing administrative updates (Scenarios 1 and 2). SELECT queries read a snapshot that excludes uncommitted changes, avoiding dirty reads.

- **Consistent snapshots for analytical workloads:** For batch or analytical jobs that need to work with a stable view of metadata (Scenario 5), transactions may run under REPEATABLE READ, which uses MVCC snapshots to avoid non-repeatable reads and many phantom reads.

Additionally, the explicit version columns used for OCC complement MVCC by:

- Providing a logical version that can be propagated to external components such as the search index (Scenario 4).

- Allowing consumers to ignore stale updates if they carry an older version than the one currently stored.

In this way, MVCC and application-level versioning jointly support both internal transactional consistency and coherent propagation of changes to external services.

3.6 Retry and Backoff Strategies

Despite careful selection of isolation levels and the use of locking and OCC, some conflicts and transient failures are unavoidable, especially during periods of high contention. To handle such situations gracefully, the application includes retry mechanisms with exponential backoff:

Conditions for retry.

1. Serialization failures under `SERIALIZABLE` or `REPEATABLE READ`.
2. Deadlock detection errors raised by PostgreSQL.
3. Transient connectivity issues, timeouts, or temporary unavailability of database nodes.

Retry procedure.

1. The failed transaction is rolled back.
2. The application waits for a random delay that increases with each attempt (e.g., 50–100 ms for the first retry, 100–200 ms for the second, etc.).
3. The transaction is retried up to a configurable maximum number of attempts (for example, three tries).
4. If all attempts fail, the user receives a controlled error message and is invited to retry the operation later.

Relation to scenarios. Retry and backoff help mitigate transient conflicts in Scenario 1 (concurrent metadata updates), Scenario 4 (index updates overlapping with metadata changes), and Scenario 5 (batch processing interacting with ongoing operations).

By combining preventive mechanisms (appropriate isolation, locking, OCC, MVCC) with reactive mechanisms (retry and backoff), the system achieves a robust concurrency-control model that accommodates both transactional correctness and scalability.

4 Parallel and Distributed Database Design

The Digital University Library must support a growing number of users, increasing volumes of digital resources, and intensive access patterns, especially during exam periods and peak academic deadlines. In this context, a purely centralized database architecture would be insufficient to guarantee performance, scalability, and availability. This section therefore proposes a parallel and distributed database design that integrates relational, NoSQL, search, and storage technologies in a coherent way. The design is intentionally polyglot: each component is responsible for a specific class of workload, while the application layer provides a unified access interface.

At the core of the system, a relational database management system (PostgreSQL) acts as the primary store for critical, structured information. This includes user accounts and roles, resource metadata (titles, authors, categories, licenses), and user-generated content such as reviews. To increase read scalability and availability, PostgreSQL is deployed following a primary–replica architecture: one primary node handles all write operations and consistency-critical reads, while one or more read replicas serve read-only traffic. This arrangement allows the system to offload read-intensive operations—such as catalog browsing and resource detail retrieval—from the primary node, without compromising transactional guarantees for updates.

Complementing the relational core, a sharded MongoDB cluster is used to store high-volume, semi-structured data, primarily activity logs and behavioral events. Collections such as download events and search logs exhibit write-intensive, append-only access patterns and may grow to millions of documents over time. For these workloads, horizontal partitioning (sharding) across multiple nodes is preferred over vertical scaling of a single server. Shard keys are chosen so as to distribute load by both academic dimension (for example, faculty or program) and time, which helps balance write traffic and enables efficient, shard-local aggregations. Each shard is configured as a replica set, providing resilience to node failures and the possibility of scaling read-heavy analytical queries over logs.

Search functionality, which is central to the user experience of a digital library, is delegated to a distributed search engine such as Elasticsearch. Instead of performing complex full-text queries directly in the relational database, the system maintains one or more inverted indices that are partitioned into multiple shards and replicated across several search nodes. A coordinating node receives each search request, distributes it to all relevant shards in parallel, and merges the partial results into a single ranked list. This parallel query processing significantly reduces latency for full-text searches and allows the search sub-system to scale independently as the collection of resources grows.

The storage of the actual digital content (PDF, EPUB, and other file formats) is separated from the databases and delegated to a distributed object storage system. In this layer, files are stored as objects in buckets and are automatically replicated across storage nodes according to predefined policies (for instance, a replication factor or erasure coding scheme). The relational database maintains only references to these objects—such as URIs, checksums, and basic file metadata—rather than the files themselves. This design choice reduces the load on the database, simplifies backup procedures, and enables the content layer to scale in capacity and throughput without impacting transactional performance.

Analytical workloads, such as long-term usage analysis, collection evaluation, and reporting for library management, are offloaded to a dedicated data warehouse or analytical database. Periodic ETL processes extract data from PostgreSQL and MongoDB, transform it into a schema optimized for analytics, and load it into the warehouse. These ETL processes can themselves exploit parallelism by reading from multiple PostgreSQL replicas and MongoDB shards simultaneously. By separating operational and analytical concerns, the architecture prevents expensive analytical queries from interfering with day-to-day transactional operations.

Parallelism and distribution are exploited in several key paths of the system. When a user submits a search query, the request first reaches the web application or API gateway, which forwards the query to the search cluster. The search engine then evaluates the query in parallel across all index shards, each performing local ranking and returning its top results. The coordinator merges these results and produces a global ranked list of resource identifiers. In a second step, the application retrieves the corresponding metadata from PostgreSQL, often using batched queries that are balanced across read replicas. Frequently accessed items may be served directly from a cache layer, further reducing pressure on the database. In some cases, additional information—such as recommendations based on co-download patterns—is obtained by executing aggregations in parallel across MongoDB shards. The final response presented to the user is thus the result of several parallel operations across distinct components.

The proposed design is justified by the project's requirements in several dimensions. First, with respect to data volume ("big data"), the combination of sharded MongoDB for logs and a data warehouse for analytics ensures that the system can absorb and analyze large streams of behavioral data without overloading the relational core. Second, multi-location and campus-wide access are supported by the use of replicas and distributed storage: read replicas of PostgreSQL, search nodes, and object storage nodes can be deployed closer to different campuses, reducing latency and providing local failover options. Third, high availability is addressed through replication at multiple layers: PostgreSQL streaming replication, MongoDB replica sets, replicated search shards, and redundant object storage all contribute to resilience in the face of node or network failures.

Finally, the separation of responsibilities between components reflects a deliberate architectural trade-off. Strong consistency is maintained where it is most critical—namely, in PostgreSQL for user accounts, access control, and authoritative metadata—while eventual consistency is accepted for activity logs, search indices, and some recommendation features. This balance allows the Digital University Library to scale horizontally, maintain acceptable response times under peak loads, and continue operating even when individual nodes become unavailable, all while preserving the integrity of its core data.

High-level Distributed Data Architecture As a synthesis of the design discussed above, Figure 4 presents a high-level view of the distributed data architecture of the Digital University Library. The diagram shows how Node A (PostgreSQL primary), the MongoDB shards (Nodes B–D), the parallel search engine, the distributed object storage, and the data warehouse interact to support both transactional and analytical workloads.

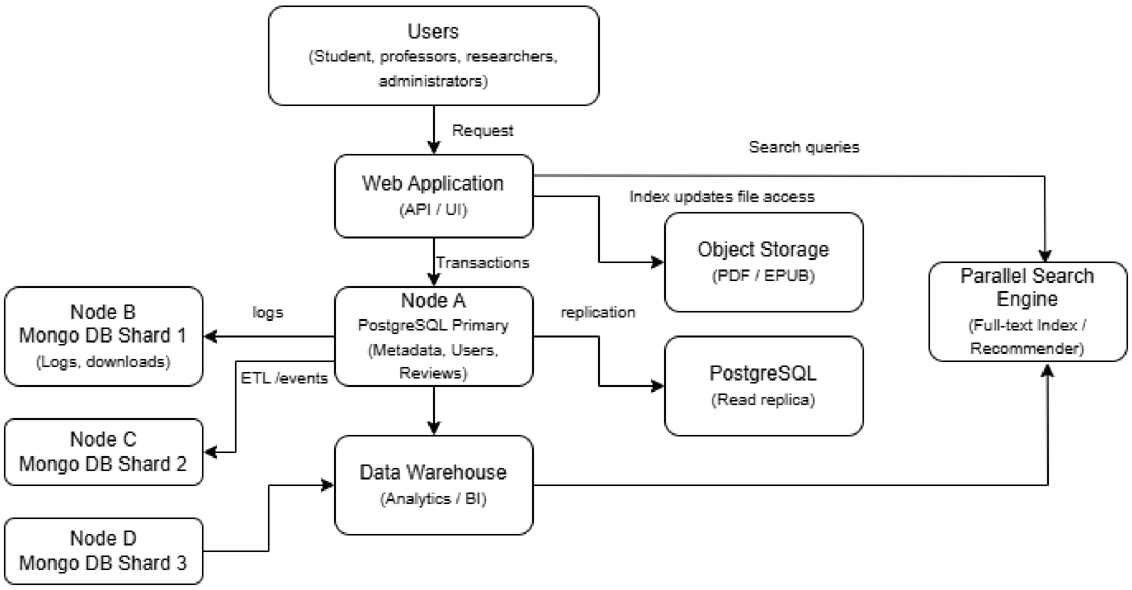
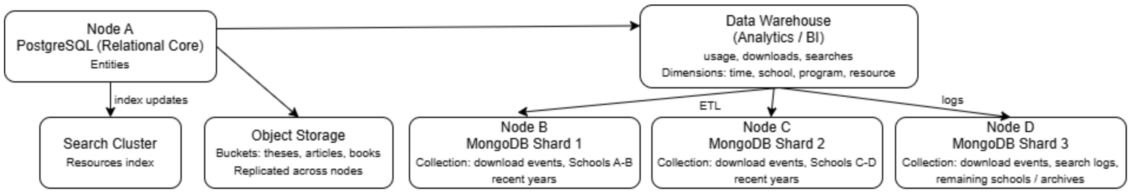


Figure 4: High-level distributed data architecture for the Digital University Library.

Data Distribution Across Nodes Beyond the high-level architecture, it is important to clarify how data is physically and logically distributed across the different nodes. Figure 5 summarizes the assignment of responsibilities for each component: Node A (PostgreSQL) stores the authoritative relational metadata, Nodes B–D (MongoDB shards) manage high-volume activity logs, the search cluster maintains the full-text index, the object storage layer holds the digital files, and the data warehouse integrates these sources for analytical purposes.



Node A
PostgreSQL (Relational Core)
Entities
Data Warehouse
(Analytics / BI)
usage, downloads, searches
Dimensions: time, school, program, resource
index updates
ETL
logs
Search Cluster
Resources index
Object Storage
Buckets: theses, articles, books
Replicated across nodes
Node B
MongoDB Shard 1
Collection: download events, Schools A-B recent years
Node C
MongoDB Shard 2
Collection: download events, Schools C-D recent years
Node D
MongoDB Shard 3
Collection: download events, search logs, remaining schools / archives

Figure 5: Data distribution across nodes in the distributed Digital University Library.

5 Performance Improvement Strategies

To ensure the system meets the requirements of low latency and high availability, particularly given the hybrid architecture (PostgreSQL + MongoDB + Redis), four key performance strategies have been defined. These strategies address specific bottlenecks identified in the volume analysis.

5.1 Caching Strategy (Redis)

Mechanism: Implementation of a high-speed caching layer using Redis (Key-Value store) to intercept frequent read operations before they reach the primary database.

- **Target Data:** Frequently accessed metadata (e.g., "Top 10 most borrowed books", "New Arrivals"), user session tokens, and pre-computed search suggestions.
- **Advantage:** Drastically reduces the read load on the PostgreSQL database and improves response times for the end-user to sub-millisecond latency.
- **Trade-off:** Risk of data staleness. A strict Time-To-Live (TTL) policy (e.g., 10 minutes for lists, 30 minutes for sessions) is required to balance freshness with performance.

5.2 Data Partitioning (Sharding in MongoDB)

Mechanism: Horizontal scaling (Sharding) applied to the MongoDB collections to handle the high volume of activity logs.

- **Shard Key:** `Faculty_ID`.
- **Target Data:** Activity Logs (Searches, Downloads) and Semi-structured Resource Metadata.
- **Advantage:** Distributes the storage and read/write workload across multiple servers. As the "Engineering" faculty generates significantly more logs than smaller faculties, sharding prevents a single server from becoming a bottleneck.
- **Trade-off:** Increases infrastructure complexity. Queries that do not include the shard key (Scatter-Gather queries) become more expensive.

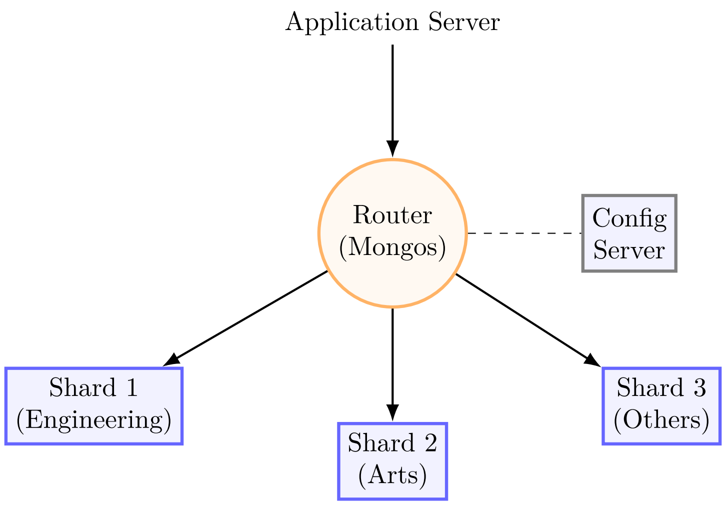


Figure 6: Conceptual Sharding Strategy for Activity Logs

5.3 Geographic Replication

Mechanism: Deployment of Read Replicas in different campus locations (Multi-site availability).

- **Priority:** Availability (AP in CAP Theorem) for read operations.
- **Advantage:** Ensures that if the main campus server goes down or experiences network latency, satellite campuses can still access catalog data for reading purposes.
- **Trade-off:** Eventual Consistency. A student might return a book at one campus, and the status might take a few seconds to reflect in another campus node.

5.4 Parallel Query Execution (PostgreSQL)

Mechanism: Enabling parallel workers for the BI/Analytics module within the relational database.

- **Target Data:** Structured historical data for generating complex reports (e.g., "Books borrowed per semester per career").
- **Advantage:** Utilizes multiple CPU cores to scan large tables simultaneously, significantly reducing the execution time for complex aggregate queries used in the administrative dashboard.
- **Trade-off:** High CPU consumption during report generation. These heavy analytical queries should be scheduled during off-peak hours or directed to a specific Read Replica to avoid slowing down the transactional system.

6 Final Reflection: The Challenge of Hybrid Consistency

Designing the Digital Library platform under a Polyglot Persistence architecture presented significant challenges regarding data consistency. While separating structured data (PostgreSQL) from semi-structured logs (MongoDB) allows for specialized optimization, it introduces the "dual-write" problem—where a transaction might succeed in one database but fail in the other.

We concluded that maintaining strict ACID properties across the entire system is neither feasible nor necessary for all modules. For instance, while user authentication and inventory management require strong consistency (handled by PostgreSQL), activity logging and search indexing favor availability and partition tolerance (handled by MongoDB and Redis).

The primary trade-off accepted in this architecture is *operational complexity*. Managing three different database engines requires a robust DevOps strategy and careful application-level handling of eventual consistency. However, this complexity is justified by the flexibility gained: we can scale the logging system independently from the transactional core, ensuring the platform remains responsive even during high-traffic periods like exam weeks.

7 Conclusion

The evolution of the Digital University Library project from Workshop 2 to Workshop 3 demonstrates the necessity of a data-centric approach in modern software architecture. By refining the scope to exclude application-level concerns and focusing on the underlying data models, we established a robust hybrid architecture capable of handling diverse information types.

The concurrency analysis highlighted the inherent risks of multi-user environments, such as lost updates and phantom reads, which were mitigated through the proposed performance strategies. The integration of Caching, Sharding, and Parallelism ensures that the system can scale horizontally to accommodate growing volumes of logs and users while maintaining low latency for critical operations. Ultimately, the proposed Polyglot Persistence model successfully balances the trade-offs between consistency and availability, providing a solid foundation for a high-performance academic platform.