UNIVERSITÉ Concordia UNIVERSITY

Course: SOEN363 – Data Systems for Software Engineers

Section: S

DataZenith Relational to NoSQL Database Project Final Group Report

Team Members:

Mohamed Saidi - 40248103

Miskat Mahmud – 40250110

Abdel-Rahman Khalifa - 40253332

Beaudelaire Tsoungui Nzodoumkouo – 40216598

Presented to Professor Ali Jannatpour

Submitted on: Monday, 14 April 2025

Project Overview

DataZenith is a comprehensive database engineering project developed by **Mohamed Saidi** (Team Lead), alongside team members **Miskat Mahmud**, **Abdel-Rahman Khalifa**, and **Beaudelaire Tsoungui Nzodoumkouo**. The project focused on gathering and analyzing data related to **books and publications**, using public APIs such as **OpenLibrary** and **OpenArchive**. We successfully collected over **600MB of data**, encompassing a wide range of bibliographic information.

We then designed a robust **SQL database schema** using PostgreSQL, modeling the gathered data through well-structured relational tables. Custom data ingestion clients were implemented to populate the database, enabling complex queries and relationship handling across entities.

Following the SQL implementation, we migrated the data to **Neo4j**, a NoSQL graph database. This phase involved writing scripts for data transformation and transfer from the relational model to a graph-based structure. We then performed a **comparative analysis** of the two database systems, focusing on query performance, indexing strategies, and overall efficiency.

Throughout the project, we explored a variety of data access patterns and optimization techniques across both SQL and NoSQL paradigms to demonstrate real-world database engineering principles.]

Data Sources

For this project, we utilized two main public APIs as our primary data sources:

1. **OpenLibrary API**
   The OpenLibrary API provides access to an extensive collection of bibliographic data, including book titles, authors, publication years, subjects, ISBNs, and edition details. We chose OpenLibrary because of its rich, well-documented dataset and its ability to return comprehensive book metadata in a structured format.
2. **Internet Archive (OpenArchive) API**
   The Internet Archive API offers additional book metadata along with digital

access links, previews, and scanning information. It served as a complementary source to enrich our dataset with metadata not covered by OpenLibrary.

We selected these APIs to ensure a diverse and substantial dataset of over **600MB**, sufficient for both relational and graph-based modeling. During the data gathering phase, we implemented custom scripts in **Python** to handle pagination, API rate limits, and inconsistent data structures across responses.

After fetching the data, we stored it temporarily in JSON format and performed cleaning and normalization before populating the SQL database.

Data Coverage

To ensure a representative dataset, we gathered books from the following topics:

- General science
- Literature
- World History
- Technology
- Art
- Mathematics
- Philosophy
- Engineering
- Biology
- Music
- Economics
- Psychology
- Sociology
- Health
- Environmental Science
- Political Science

This wide topic range helped gather the required size of data and test the effectiveness of schema design, query optimization, and data modeling in both SQL and NoSQL settings.

Relational Database (Phase I)

In the first phase of the project, we fetched book data from the sources mentioned earlier and stored it in a PostgreSQL database. We designed a normalized schema that accurately captured the relationships between entities such as books, authors, subjects, and editions. We ensured efficiency through appropriate indexing, created views to represent different perspectives of the data, and enforced data integrity using table constraints and triggers to prevent invalid operations.

Data Model & Design

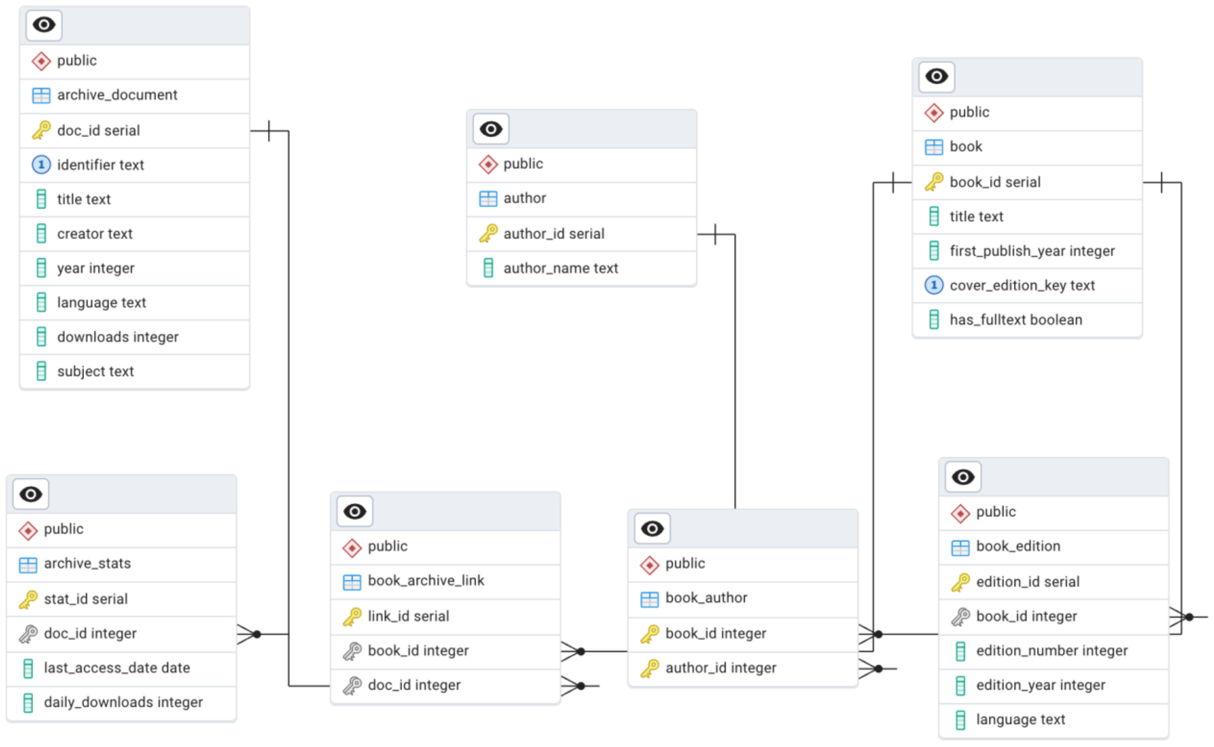

FIGURE 1: SQL POSTGRES PUBLIC SCHEMA ER DIAGRAM

Core Entities and Relationships:

- **Book**: Central entity representing a book title. It includes core metadata such as title, first publish year, and whether full text is available.
- **Author**: Separate entity normalized from books to support many-to-many relationships. Indexed for fast author name lookups.
- **Book_Author**: A join table enabling many-to-many associations between books and authors.

- **Book_Edition** (*IS-A* Relationship): A subtype of Book capturing edition-specific metadata like language, edition year, and edition number. This demonstrates an **inheritance**-style relationship in SQL.
- **Archive_Document**: Stores metadata for digital documents retrieved from Archive.org, including download statistics and subject tags.
- **Archive_Stats** (*Weak Entity*): Dependent on **Archive_Document**, it tracks time-sensitive statistics like daily downloads, showcasing an example of a weak entity relationship.
- **Book_Archive_Link**: Provides a logical bridge between OpenLibrary's book records and Archive.org's digital documents, linking **book_id** and **doc_id** despite differing identifier systems. This was key to unifying both datasets.

Schema Features

Our schema is optimized for representing and querying book metadata across both sources, emphasizing data integrity, normalization, and cross-dataset linkage.

Key Design Highlights

- **Indexing**: Indexes were created on key fields (e.g., title, author_name, edition_year, language) to improve query performance.
- **Views**:
  - Archive_Download_Summary: An aggregation view summarizing downloads per year.
  - Restricted_Books: A filtered view that simulates access control based on publication date.
- **Referential Integrity**:
  - Constraints and foreign keys enforce relational correctness.
  - A trigger (prevent_book_deletion) prevents deletion of books that are still linked to an archive document, ensuring consistency across joined datasets.

SQL Query Examples

We implemented a diverse set of SQL queries to showcase data retrieval strategies, performance optimizations, and advanced relational features.

Selected Queries

```sql
-- (c) LEFT OUTER JOIN (shows all Books even if no matching edition; nulls appear):
SELECT b.book_id, b.title, be.edition_year
FROM Book b
LEFT JOIN Book_Edition be
    ON b.book_id = be.book_id;
-- (a) Find Books that do not have a defined publication year.
SELECT book_id, title
FROM Book
WHERE first_publish_year IS NULL;
-- (c) GROUP BY with HAVING (show years with more than 1 book):
SELECT first_publish_year, COUNT(*) AS num_books
FROM Book
GROUP BY first_publish_year
HAVING COUNT(*) > 1;
-- List book titles and author names for books that have at least one edition published after 2010.
SELECT b.title, a.author_name
FROM Book b
JOIN Book_Author ba ON b.book_id = ba.book_id
JOIN Author a ON a.author_id = ba.author_id
WHERE b.book_id IN (
    SELECT be.book_id
    FROM Book_Edition be
    WHERE be.edition_year > 2010
);

-- (b) List books whose publication year equals the average publication year
--     of all books written by at least one of its authors.
SELECT b.title, b.first_publish_year
FROM Book b
WHERE b.first_publish_year = (
    SELECT AVG(b2.first_publish_year)
    FROM Book b2
    JOIN Book_Author ba2 ON b2.book_id = ba2.book_id
    WHERE ba2.author_id IN (
         SELECT ba3.author_id
         FROM Book_Author ba3
         WHERE ba3.book_id = b.book_id
    )
);
```

For the complete set of SQL queries, please refer to the GitHub repository under the GitHub Repository section.

Implementation Platform

We used the Python programming language to write clients that request and fetch data from public APIs and populate it into the PostgreSQL database. For database connectivity, we used the **pg8000** library, a pure Python driver for PostgreSQL, which allowed seamless execution of SQL queries and transactions from our scripts.

Additionally, we used platforms like **PGAdmin** to visually monitor and manage database tables, and **Postman** to test and adapt public APIs to our specific needs. We also employed **Docker** to ensure consistent and reproducible deployment environments across systems.

NoSQL Database (Phase II)

In the second phase of the project, we developed Python scripts to migrate data from the relational PostgreSQL database into the graph-based NoSQL database, Neo4j. During this transition, we applied several schema design changes to better align with the graph data model and to take advantage of the flexibility provided in the Phase II specifications. These adjustments were aimed at improving data representation, enabling more intuitive relationships, and optimizing query performance within the graph structure.

Design Adjustments

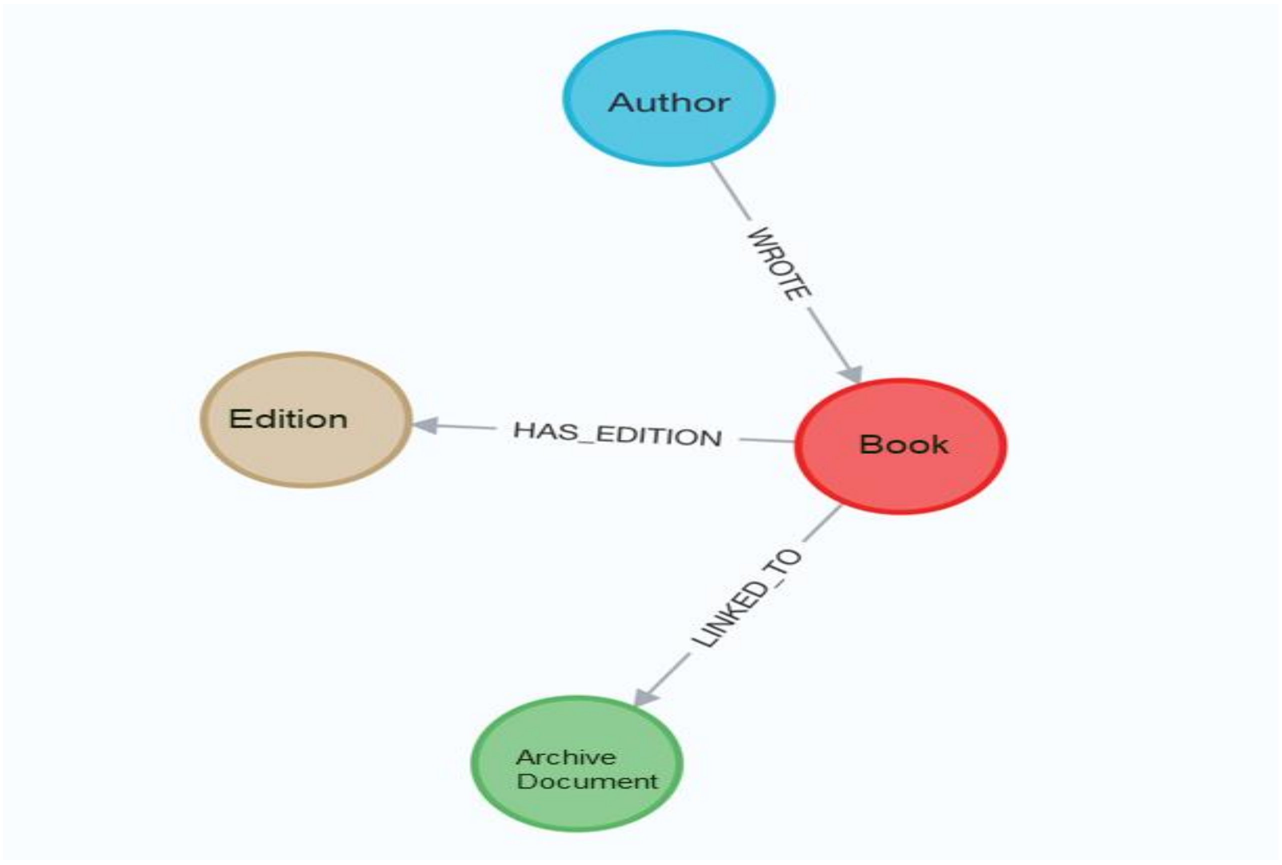

FIGURE 2: NOSQL NEO4J GRAPH DIAGRAM

1. **IS-A Relationship Simplification**
   a. *Before:* Book_Edition was a subtype of Book using foreign keys.
   b. *Now:* BookEdition is its **own node** connected to Book with HAS_EDITION.
   c. *Reason:* Graphs don't support class inheritance — relationships are cleaner.
2. **Join Tables Converted to Relationships**
   a. *Before:* Book_Author and Book_Archive_Link were join tables.
   b. *Now:* Replaced with native **graph relationships**:
      i. (:Book)-[:WRITTEN_BY]->(:Author)
      ii. (:Book)-[:HAS_ARCHIVE]->(:ArchiveDocument)
   c. *Reason:* Graphs support many-to-many relationships natively, reducing complexity.
3. **Weak Entity Simplification (Archive_Stats)**
   a. *Before:* Modeled as a weak entity tied to Archive_Document.
   b. *Now:* ArchiveStats is a separate node, linked via HAS_STATS.

   c. *Reason:* This allows tracking multiple time-series stats per document in the future.

4. **Views and Aggregations**
   a. *Before:* SQL views like Archive_Download_Summary, Restricted_Books.
   b. *Now:* Replaced with **dynamic Cypher queries**
   c. *Reason:* Neo4j doesn't support views but supports powerful aggregations via Cypher.
5. **Referential Integrity**
   a. *Before:* Enforced with foreign keys and triggers.
   b. *Now:* Handled in **Python scripts or APIs** (e.g., only delete nodes if detached).
   c. *Reason:* Neo4j does not enforce referential integrity — application logic is responsible.

Final Graph Model Summary

- **Nodes**:
  - Book, Author, BookEdition, ArchiveDocument, ArchiveStats
- **Relationships**:
  - (:Book)-[:WRITTEN_BY]->(:Author)
  - (:Book)-[:HAS_EDITION]->(:BookEdition)
  - (:Book)-[:HAS_ARCHIVE]->(:ArchiveDocument)
  - (:ArchiveDocument)-[:HAS_STATS]->(:ArchiveStats)
  -

NoSQL Implementation and Data Migration

In Phase II, we migrated and restructured the data from the PostgreSQL database to the Neo4j graph database.

Migration Strategy

The migration was performed directly using Python scripts. We used the `pg8000` library to connect to PostgreSQL and the official Neo4j Python driver to interact with the graph database. Data was fetched from the SQL tables and inserted into Neo4j by creating the appropriate nodes and relationships. Structural differences—such as converting join tables to relationships—were handled dynamically during the transformation process.

NoSQL query Demonstrations

To showcase the querying capabilities of Neo4j and the expressiveness of the Cypher query language, we implemented several types of queries on the graph data, including attribute filtering, aggregation, and full-text search.

Below are two representative examples:

1. Basic Attribute Filter Query

```
// Search for books by a specific title
MATCH (b:Book)
WHERE b.title = "The Lord of the Rings"
RETURN b;
```

2. Aggregation and Grouping

```
// Count how many books have full text
MATCH (b:Book)
WHERE b.has_fulltext = true
RETURN count(b) AS number_of_books_with_fulltext;
```

Additional query examples, including full-text search, indexing, and grouping by publication year, can be found by referring to the GitHub repository under the GitHub Repository section.

Performance Comparison

To evaluate performance, we compared query execution times between PostgreSQL and Neo4j across multiple query types, ranging from simple counts to relational joins and full-text searches. Below are the key findings:

1. **Simple Count Queries**
   Neo4j consistently outperformed PostgreSQL on basic aggregation queries. For example, counting books or archive documents took **4–5 ms** in Neo4j, compared to **110–188 ms** in PostgreSQL. This highlights Neo4j's efficiency in scanning node-based data structures.

2. **Join and Relationship Queries**
   For queries involving relationships (e.g., authors and books), PostgreSQL showed slightly better performance. A join on the book_author table returned in **161 ms** in PostgreSQL, while Neo4j took **214 ms** for a similar (:Author)-[:WROTE]->(:Book) traversal. However, Neo4j's performance was still competitive, especially considering its native graph traversal capabilities.

3. **Grouped Aggregations and Filtering**
   When fetching books along with their editions (with filtering and aggregation), PostgreSQL executed the query in **400 ms**, while Neo4j took **802 ms**. While Neo4j processed a large volume of records, PostgreSQL's optimized JOIN and GROUP BY operations proved more efficient in this case.

4. **Multiple Aggregated Counts**
   Queries combining multiple counts (authors, books, archive documents) were faster in Neo4j (**84 ms**) compared to PostgreSQL (**329 ms**), suggesting that Neo4j handles multiple aggregations in separate entity types more efficiently.
5. **Index Impact on Query Time**
   We tested the impact of indexing in Neo4j using PROFILE.
   a. A query filtered by first_publish_year = 1999 took **194 ms** without an index, and only **112 ms** with an index.
   b. Total DB hits dropped from **27,604** to **541**, clearly demonstrating the performance boost from index creation.

6. **Full-Text Search Optimization**
   A full-text search on ArchiveDocument nodes improved significantly after index creation:
   a. **Before Index**: 643 ms with **50,001** DB hits
   b. **After Index**: 231 ms with only **27** DB hits
      This confirms the importance of full-text indexing in optimizing text-heavy queries in Neo4j.

Please refer to Appendix A for query details.

Resources & Access

GitHub Repository

The complete source code, including schema design files, SQL and Cypher scripts, and data migration tools, is available on GitHub:

https://github.com/codedsami/datazenith-soen-363/tree/main

The repository is organized as follows:

- Phase 1/: Python API fetching code, SQL schema, views, and relational scripts
- Phase 2/: Neo4j Cypher scripts and Python migration code
- Final report/: Final report document
- Presentation Slides/: Presentation slides
- Video/: Contains the demo video

Database Dump File

A SQL dump of the final Postgres database (Phase I) is available for download:
Download SQL Dump

A NoSQL dump of the final Neo4j database (Phase II) is available for download:
Download Neo4j Dump

Challenges & Lessons Learned

Challenges

In **Phase I**, we encountered several challenges, particularly when switching between different APIs to meet the data requirements. Initially, we focused on "science" books, but we had to broaden our scope to include other topics. This led to complications when navigating the varying data structures and API responses.

We also dealt with significant **public API limitations**, such as rate limiting and reliability issues, which disrupted the flow of data collection and delayed progress. To handle these, we had to implement workarounds like managing retries and batch processing.

Another major hurdle was optimizing **queries for large datasets**. We needed to ensure that our queries were efficient and could handle the growing volume of data without sacrificing performance.

Additionally, **maintaining schema consistency and ensuring that all team members had access to a reliable database state** was a constant challenge. Collaborating across multiple team members involved careful planning and coordination to avoid conflicts in data structures or redundant efforts.

Lastly, **normalizing tables** and understanding what should be indexed in the relational schema was another complex issue that required careful attention to performance and future scalability.

Lessons Learned

Throughout the project, we gained valuable insights into **schema design** for both **SQL** and **NoSQL** databases. In **SQL**, we focused on normalizing tables and optimizing relational structures, while in **Neo4j**, we had to rethink our approach, using graph relationships instead of foreign keys or join tables. This taught us the importance of adapting schemas to the strengths of each database model.

Handling **large datasets** required careful query optimization. In PostgreSQL, complex joins can degrade performance, while in Neo4j, we optimized for efficient graph traversal, leveraging relationships for faster queries. We learned to design the graph model for both storage efficiency and retrieval speed.

Managing **data consistency** during migration was another challenge. Ensuring schema alignment between PostgreSQL and Neo4j, and validating the data during transformation, highlighted the importance of data validation and clear communication within the team to avoid inconsistencies.

In summary, the project taught us how to design effective schemas and manage data integrity across different database types, while prioritizing performance and scalability, especially with large datasets.

Presentation Summary

This project involved migrating book data from a PostgreSQL relational database (Phase I) to a Neo4j graph database (Phase II) to improve scalability and flexibility. In Phase I, we designed a normalized schema, optimized queries, and ensured data consistency. Phase II focused on restructuring the data for the graph model, using Python scripts to migrate data and create relationships between entities like books, authors, and archive documents.

Key challenges included handling API limitations, optimizing large datasets, and managing schema consistency across the team. A major lesson was learning how to design and optimize schemas for both SQL and NoSQL databases, particularly for handling complex relationships and ensuring data integrity in large-scale systems.

Appendix A – Query Details

A.1 – Simple Count Queries

Count all Books

Neo4j:

```
MATCH (b:Book)

RETURN count(b);
```

Execution Time: Started in 4 ms, completed in 5 ms.

PostgreSQL

```
SELECT COUNT(*) FROM public.book;
```

Execution Time: 110 ms.

Count all Archive Documents

Neo4j:

```
MATCH (n:ArchiveDocument)
RETURN count(n);
```

Execution Time: Started and completed in 4 ms

PostgreSQL:

```
SELECT COUNT(*) FROM public.archive_document;
```

Execution Time: 188 ms.

A.2 – Join and Relationship Queries

Authors and Books Relationship

Neo4j:

```
MATCH (a:Author)-[:WROTE]->(b:Book) RETURN a.author_id AS
AuthorID, b.book_id AS BookID;
```

Execution Time: Started in 7 ms, completed in 214 ms (115,504 records streamed).

PostgreSQL: Author-Book Join

```
SELECT a.author_id AS AuthorID, a.book_id AS BookID FROM
public.book_author a;
```

Execution Time: 161 ms.

A.3 – Grouped Aggregations and Filtering

Books with Editions (Filtered & Aggregated)

Neo4j:

```
MATCH (b:Book)-[:HAS_EDITION]->(e:BookEdition) WHERE
e.edition_year > 1940 AND e.language = 'eng' RETURN b.title
```

```
AS BookTitle, b.first_publish_year AS FirstPublishYear,
COLLECT({ EditionID: e.edition_id, EditionNumber:
e.edition_number, EditionYear: e.edition_year, Language:
e.language }) AS EditionDetails ORDER BY
b.first_publish_year ASC;
```

Execution Time: Started in 10 ms, completed in 802 ms (62,296 records streamed).

PostgreSQL:

```
SELECT b.title AS BookTitle, b.first_publish_year AS
FirstPublishYear, JSON_AGG(JSON_BUILD_OBJECT( 'EditionID',
be.edition_id, 'EditionNumber', be.edition_number,
'EditionYear', be.edition_year, 'Language', be.language ))
AS EditionDetails FROM Book b JOIN Book_Edition be ON
b.book_id = be.book_id WHERE be.edition_year > 1940 AND
be.language = 'eng' GROUP BY b.book_id, b.title,
b.first_publish_year ORDER BY b.first_publish_year ASC;
```

Execution Time: 400 ms (62,816 rows affected).

A.4 – Multiple Aggregated Counts

Count Authors, Books, and Archive Documents

Neo4j:

```
MATCH (a:Author)

WITH COUNT(a) AS AuthorCount

MATCH (b:Book)

WITH AuthorCount, COUNT(b) AS BookCount

MATCH (ad:ArchiveDocument)

RETURN AuthorCount, BookCount, COUNT(ad) AS
ArchiveDocumentCount;
```

Execution Time: Started in 8 ms, completed in 84 ms.

--

PostgreSQL:

```
SELECT
    (SELECT COUNT(*) FROM public.Author) AS AuthorCount,
    (SELECT COUNT(*) FROM public.Book) AS BookCount,
    (SELECT COUNT(*) FROM public.Archive_Document) AS
ArchiveDocumentCount;
```

Execution Time: 329 ms.

A.5 – Index Impact on Query Time (Neo4j)

Query

```
PROFILE MATCH (b:Book) WHERE b.first_publish_year = 1999
RETURN b.title;
```

Performance Before Index: 27,604 total DB hits, 194 ms.

--

Index Creation

```
CREATE INDEX book_year_index IF NOT EXISTS FOR (b:Book) ON
(b.first_publish_year);
```

--

Performance After Index: 541 total DB hits, 112 ms.

A.6 – Full-Text Search Optimization (Neo4j)

Before Full-Text Index

```
PROFILE
```

```
MATCH (d:ArchiveDocument)

WHERE d.title CONTAINS "history" OR d.subject CONTAINS
"history"

RETURN d.title AS title, d.subject AS subject

ORDER BY d.title

LIMIT 10;
```

Performance: 50,001 DB hits, 643 ms.

Create Full-Text Index

```
CREATE FULLTEXT INDEX archive_fulltext_fts FOR
(d:ArchiveDocument) ON EACH [d.title, d.subject];
```

After Full-Text Index

```
PROFILE CALL
db.index.fulltext.queryNodes("archive_fulltext_fts",
"history") YIELD node, score RETURN node.title AS title,
node.subject AS subject, score ORDER BY score DESC LIMIT
10;
```

Performance: 27 DB hits, 231 ms.