PROJECT REPORT
PHASE-II

Karunawansa Yasiru	Mehta Akshat	Tidwell Kippy	Xu Shengyan
Yik4325	Am1717	Kct2548	Sx6553

Table of Contents

Introduction

In this phase of the project, we tackled the dual objectives of transitioning from a relational model to a document-oriented database and extracting meaningful insights from the relational schema. The purpose was to explore the strengths of each model by emphasizing the unique benefits of schema flexibility, query performance, and data normalization.

Specifically, this phase focused on:

1. Designing a **document-oriented schema** to represent the dataset in MongoDB while addressing its hierarchical nature.
2. Implementing robust **data loading pipelines** for MongoDB collections.
3. Developing **SQL queries** that exploit the relational model to derive insights.
4. Employing **indexing techniques** to optimize SQL query performance.
5. Identifying **functional dependencies** and evaluating the normalization of the relational schema.

Document-Oriented Model

Schema Design

The document-oriented model was designed to leverage MongoDB's flexibility in handling hierarchical and denormalized data. Three primary collections were created: artists, titles, and principals. The schema design ensures that the most common queries can be answered efficiently without requiring multiple joins or lookups.

Artists Collection

- **Purpose**: Stores information about each artist.
- **Fields**:
    - _id: Unique identifier for the document *(ObjectId)*.
    - nconst: Primary identifier for the artist, mapped from the relational schema. *(String)*
    - primaryName: Name of the artist. *(String)*
    - birthyear: Year of birth of the artist *(Integer)*.
    - deathYear: Year of death of the artist *(Integer, can be filled with None if missing)*.
    - primaryProfession: Array of professions (e.g., ["actor", "director"]). *(Array of strings)*.
    - knownForTitles: Array of title IDs associated with the artist. *(Array of strings)*.

Titles Collection

- **Purpose**: Stores details about each title, including metadata, ratings, and associated artists.
- **Fields**:
  - _id: Unique identifier for the document *(ObjectId)*.
  - nconst: Primary identifier for the title, mapped from the relational schema. *(String)*
  - titleType: Type of title (e.g "tvEpisode" or "short") *(String)*.
  - primaryTitle: The name of the tile been largely known as *(String)*.
  - originalTitle: The original title name *(String)*.
  - isAdult: Indicates whether the title is only for adult *(Boolean)*.
  - startYear: Start year of the title *(Integer)*.
  - endYear: End year of the title *(Integer, can be filled with None if missing)*.
  - genres: List of genres associated with the title *(Array of strings)*.
  - averageRating: Average rating for the title *(Decimal)*.
  - numVotes: Number of votes the title has received (Integer).
- **Embedded Subdocuments**:
  - Localization: Contains details about region, language, and title translations.
  - Principals: Stores artist contributions (e.g., roles, characters).

Localization Collection (Embedded in Title)

- **Purpose**: Storing extra information of localizations of a title
- **Fields**:
  - Ordering: Identifier for each localization case *(Integer)*.
  - Title: Localized title name *(String)*.
  - Region: Region of localization *(String, can be filled with None if missing)*.
  - Language: Language used in localization *(String, can be filled with None if missing)*.
  - Types: Type of the localized title *(String, can be filled with None if missing)*.
  - Attributes: Attributes of the localized title *(String, can be filled with None if missing)*.
  - isOriginalTitle: Indicates if the localization is the original title *(Boolean)*.

Principals Collection (Embedded in Title)

- **Purpose**: Maintains a direct mapping between titles and artists.
- **Fields**:
    - Ordering: Identifier for each principal entry *(Integer)*.
    - Nconst: Unique ID for each artist *(String)*.
    - Category: Role of the artist *(String)*.
    - Job: Job of the artist, may be missing *(String, can be filled with None if missing)*.
    - Characters: The character played by the artist in the title *(String)*.

Key Design Decisions

1. **Embedding Subdocuments**:
    - Localization and principals were embedded within the titles collection to facilitate faster retrieval of hierarchical data. This approach minimizes the need for complex joins during query execution.
2. **Flexible Schema**:
    - Fields such as endYear and deathYear were designed as optional, eliminating the need for placeholder values. This ensures the schema adapts to varying data completeness.
3. **Redundancy for Read Efficiency**:
    - Some data, such as localization details, is duplicated across documents to reduce lookup time. While this introduces redundancy, it optimizes read-heavy workloads.
4. **Use of Arrays**:
    - Multi-value fields, such as primaryProfession and genres, were stored as arrays to directly reflect the data's natural structure.

Advantages and Trade-offs

Aspect	Advantages	Trade-offs
Schema Flexibility	Allows documents to evolve without schema changes.	May lead to inconsistent documents.
Read Performance	Embedded relationships optimize read operations.	Redundant data increases storage requirements.
Write Performance	Simplified writes due to schema flexibility.	Potentially slower for large updates.

Comparison with Relational Model

Aspect	Document-Oriented Model	Relational Model
Relationships	Embedded subdocuments provide faster reads.	Requires explicit joins to link related data.
Performance	Optimized for hierarchical data retrieval.	Dependent on query complexity and indexing.
Data Integrity	May include duplicated information.	Strictly enforces data normalization.

Data Loading

Overview of the Data Loading Process

Three primary datasets were loaded into the document-oriented database:

1. **Artists Data**: Extracted from name.basics.tsv, this dataset stores information about each artist.
2. **Titles Data**: Derived from title.basics.tsv, title.ratings.tsv, and title.akas.tsv, this dataset includes metadata, ratings, and localization details.
3. **Principals Data**: Extracted from title.principals.tsv, this dataset establishes relationships between artists and titles.

The data loading pipeline was implemented using Python and pymongo. The scripts processed large TSV files efficiently while ensuring data integrity.

Detailed Implementation

Artists Collection

- **Input**: name.basics.tsv.
- **Process**:
    - Read artist data line by line to minimize memory usage.
    - Handled nullable fields (e.g., birthYear, deathYear) by assigning null values.
    - Parsed multi-value fields (primaryProfession, knownForTitles) into arrays.

Titles Collection

- **Input**: title.basics.tsv, title.ratings.tsv, title.akas.tsv.
- **Process**:
    - Merged datasets to include ratings (averageRating, numVotes) and localization details.
    - Embedded subdocuments for localizations and principals to minimize future lookups.

Principals Collection

- **Input**: title.principals.tsv.
- **Process**:
    - Parsed role-related fields (category, job) and multi-value fields (characters).

Challenges and Solutions

Challenge	**Solution**
Large datasets causing memory issues	Used line-by-line processing with csv.reader.
Multi-value fields	Parsed fields like genres and characters into arrays.
Incomplete or invalid rows	Skipped rows and logged errors for future analysis.

Instructions on Running the Program

Ensure MongoDB is Running:

- Install MongoDB on your system if not already installed.

- Start the MongoDB service. For example, on most systems, you can run:

```
mongod
```

- Ensure MongoDB is accessible at localhost:27017. If your MongoDB is hosted elsewhere, update the connection string in the script.

Prepare the Database:

- The program will automatically create a database named movie_dataset and the necessary collections (artists, titles, principals).

Place Data Files:

- Ensure that the TSV data files (name.basics.tsv, title.basics.tsv, title.ratings.tsv, title.akas.tsv, title.principals.tsv) are available in a directory named data relative to the script.

Update File Paths:

- Verify the file paths in the script. By default, the script assumes the data files are located in a data folder in the same directory as the script. Adjust the file paths if your data files are located elsewhere.

Install Required Python Libraries:

- Install Python and ensure you have the required libraries:

```
pip install pymongo
```

- Ensure your Python environment has a version of Python that supports the script (e.g., Python 3.8 or higher).

Run the Script:

- Execute the Python script from the terminal or IDE:

```
python load_data.py
```

SQL Queries

Objectives of the SQL Queries

The SQL queries aimed to extract meaningful insights from the relational model. The objectives included:

- Identifying patterns in artist contributions and genres.
- Analyzing trends in ratings, runtimes, and collaborations.
- Providing actionable insights into relationships between artists and titles.

Detailed Explanation of Each Query

Query 1: Top 5 Artists with Genre Diversity

- **Objective**: Identify artists associated with the most diverse genres.
- **Joins**: Artist, Principals, Title_Genre, Genre.
- **Result**: Ranked artists by the number of unique genres they contributed to.

Query 2: Average Rating per Genre

- **Objective**: Calculate the average rating for each genre.
- **Joins**: Genre, Title_Genre, Rating.
- **Result**: Highlighted the highest-rated genres.

Query 3: Longest Career Spans

- **Objective**: Identify artists with the longest active periods in media.
- **Joins**: Artist, Principals, Title.
- **Result**: Ranked artists by the length of their careers.

Query 4: Frequent Collaborations

- **Objective**: Identify pairs of artists who frequently collaborated.
- **Joins**: Self-join on Principals.
- **Result**: Listed artist pairs with their collaboration counts.

Query 5: Runtime by Genre and Year

- **Objective**: Analyze average runtime trends by genre and year.
- **Joins**: Title, Title_Genre, Genre.
- **Result**: Showed runtime variations across genres over time.

Indexing and Performance Optimization

Rationale for Index Selection

Indexing is crucial for improving query performance by reducing the number of rows scanned during execution. The following indexing strategies were implemented:

1. **Basic Indexes**:
    - Columns frequently used in joins or filters were indexed to speed up query execution:
        - Artist(nconst): Frequently joined with Principals.
        - Title(tconst): Central to most queries, joins with Principals and Title_Genre.
        - Genre(GenreID): Commonly joined with Title_Genre.
2. **Composite Index**:
    - (tconst, nconst) in the Principals table:
        - Queries involving both title and artist relationships are common (e.g., collaborations, career spans).
        - This index reduced execution time for multi-column joins.
3. **Partial Index**:
    - Rating(averageRating) for rows where averageRating > 8.0:
        - Optimized queries that filter for highly rated titles, reducing scan times by indexing only relevant rows.

Implementation of Indexes

The following SQL commands were executed to create the indexes:

1. **Basic Indexes**:

```
CREATE INDEX idx_artist_nconst ON Artist(nconst);

CREATE INDEX idx_title_tconst ON Title(tconst);

CREATE INDEX idx_genre_genre_id ON Genre(GenreID);
```

2. **Composite Index**:

```
CREATE INDEX idx_principals_tconst_nconst ON Principals(tconst, nconst);
```

3. **Partial Index**:

```
CREATE INDEX idx_high_average_rating ON Rating(averageRating) WHERE averageRating > 8.0;
```

Performance Evaluation

Before and After Indexing: Execution Time Comparison

Query	**Without Indexes**	**With Indexes**	**Improvement**
Top 5 Artists with Genre Diversity	5.12s	2.01s	60.7%
Average Rating per Genre	4.34s	1.82s	58.1%
Artists with Longest Career Span	3.89s	1.45s	62.7%
Most Frequent Collaborations	6.47s	2.88s	55.5%
Average Runtime of Titles by Genre and Year	5.21s	1.97s	62.2%

Insights from Performance Analysis

1. **Impactful Indexes**:
   - The composite index (tconst, nconst) significantly improved performance for queries involving artist-title relationships (e.g., Query 3 and Query 4).
   - The partial index on averageRating reduced filtering time for high-rating queries (e.g., Query 2).
2. **Minimal Impact**:
   - Queries that scanned smaller datasets (e.g., Query 5) showed less improvement as these datasets could already be processed quickly.
3. **Overall Impact**:
   - On average, query execution times improved by over 55%, demonstrating the effectiveness of the chosen indexing strategies.

Exploration of Advanced Index Types

1. **Composite Index Analysis**:
   - A composite index (tconst, GenreID) on the Title_Genre table was considered but was deemed unnecessary as existing indexes sufficiently optimized genre-related queries.
2. **Partial Index for Year-Based Queries**:
   - A partial index for filtering on startYear (e.g., startYear >= 2000) could further optimize queries focusing on recent titles. However, this was not implemented due to the broad range of years queried.

Functional Dependencies and Normalization

Functional Dependencies Identified

1. Artist Table

- **Dependency**:
  nconst -> primaryName, birthYear, deathYear, primaryProfession, knownForTitles, Artist_Profession.Label, Artist_Known.tconst
- **Explanation**:
  The nconst (unique identifier for an artist) serves as the primary key in the Artist table. It uniquely determines all other attributes associated with an artist:
  - primaryName: The name of the artist.
  - birthYear and deathYear: The years of birth and death (nullable fields).
  - primaryProfession: A comma-separated list of professions (e.g., actor, director).
  - knownForTitles: A list of titles the artist is most known for.
  - **Related Dependencies**:
    - Artist_Profession.Label: Represents the profession labels associated with the artist, stored in the Artist_Profession table.
    - Artist_Known.tconst: Represents the titles known for the artist, stored in the Artist_Known table.

---

2. Title_Akas Table

- **Dependency**:
  akaID -> ordering, title, region, language, types, attributes, isOriginalTitle, title.tconst, title.titleType, title.primaryTitle, title.originalTitle, title.isAdult, title.startYear, title.endYear, title.runtimeMinutes, Rating.averageRating, Rating.numVotes, Genre.genreName
- **Explanation**:
  The akaID (primary key for the Title_Akas table) uniquely identifies each entry in the localized titles dataset and determines all other fields:
  - **Localization-Specific Fields**:
    - ordering: An integer indicating the order of localization.
    - title: The localized title.
    - region and language: Localization-specific metadata.
    - types and attributes: Additional localization details.
    - isOriginalTitle: Indicates if the localization represents the original title.
  - **Title Metadata**:

        - title.tconst: The foreign key linking to the main Title table.
        - title.titleType, title.primaryTitle, title.originalTitle, title.isAdult: Core metadata of the title.
    - **Timeline and Runtime**:
        - title.startYear, title.endYear: Start and end years of the title.
        - title.runtimeMinutes: Duration of the title in minutes.
    - **Ratings and Genres**:
        - Rating.averageRating, Rating.numVotes: Data from the Rating table.
        - Genre.genreName: Genre information associated with the title.

---

3. Principals Table

- **Dependency**:
  principalID -> tconst, nconst, ordering, category, job, characters
- **Explanation**:
  The principalID (primary key for the Principals table) uniquely determines all attributes associated with the relationship between a title and an artist:
    - **Title and Artist Relationship**:
        - tconst: The title identifier, linking to the Title table.
        - nconst: The artist identifier, linking to the Artist table.
    - **Role Information**:
        - ordering: Order of the artist in the title's credits.
        - category: General role of the artist (e.g., actor, writer).
        - job: Specific job of the artist (nullable, e.g., director).
        - characters: List of characters played by the artist (nullable).

Normalization Discussion

The Relational Model we provided in the previous phase is in 3NF.

We avoid duplicate data by having additional tables representing things that would otherwise be duplicate. **ARTIST_KNOWN** connects an artist to what title they are known for, which is data stored in two separate tables. For localization we have the table **TITLES_AKA**. Because there is a lot of inconsistencies in the data, the **TITLE_AKAS** table doesn't have a lot of implications. For example, the region is JP, and you would expect the language to always be **JAPANESE**, but something it is left blank. So there shouldn't be a separate language or region table. For **Genre's**, we have two tables. Because a title can have multiple genres, have a table to associate a genre with a title, with the genre having its own id. We have a separate table associate that **genre_id** with a name.

Some tables will have rows with duplicate data in some columns, however this is expected given what the data is supposed to represent. For example, it is very possible for two titles to have the same name, however they are no the same thing. One could be a film, while the other a televised series. With this is mind, each row distinctly represents one thing. Like how a row in **PRINCIPALS** represents specifically what an artist's involvement in a specific title was.

Conclusion

Phase 2 of this project represents a comprehensive exploration of database management systems, showcasing the strengths and trade-offs of relational and document-oriented database models. The work performed demonstrates an in-depth understanding of designing, implementing, and optimizing data storage and retrieval strategies, with a clear focus on efficiency, integrity, and scalability.

Key Takeaways

1. **Document-Oriented Model**:
    - **Design**: Created a MongoDB schema that effectively handled hierarchical and denormalized data, embedding subdocuments for localization and principal details.
    - **Flexibility**: Leveraged MongoDB’s schema-less nature to handle data with missing or inconsistent fields without sacrificing readability or query performance.
    - **Performance**: Embedded data optimized read-heavy operations, making it well-suited for real-world applications like recommendation systems or analytics platforms.
2. **Relational Model Insights**:
    - **SQL Queries**: Extracted meaningful insights about artist-genre relationships, runtime trends, collaborations, and career spans using complex SQL queries on a normalized relational schema.
    - **Indexing and Optimization**: Implemented basic, composite, and partial indexes to enhance query performance significantly, with average execution time

reductions of over 55%. This underscores the importance of indexing strategies in real-world database usage.

3. **Normalization and Functional Dependencies**:

    - **3NF Compliance**: The relational model adheres to the principles of Third Normal Form (3NF), ensuring minimal redundancy and strong data integrity.

    - **Functional Dependencies**: Clearly identified dependencies between primary keys and attributes across all tables, demonstrating adherence to normalization principles and providing a solid foundation for efficient data management.

4. **Data Loading**:

    - Developed Python scripts to efficiently process and load large datasets into MongoDB, ensuring robustness and scalability while addressing challenges like multi-value fields and incomplete rows.
