Database Project - Final Report

SOEN 363 Section S

Data Systems for Software Engineers

Presented to

Ali Jannatpour

Prepared by

Valeria Dolgaliova (40212218)

Vanessa DiPietrantonio (40189938)

Clara Gagnon (40208598)

Ahmad Elmahallawy (40193418)

Department of Computer Science & Software Engineering
Concordia University

Montreal, Canada

December 6th, 2023

Phase I

Overview of the System

Our team has chosen to implement a relational database on the topic of written works. A written work is considered any piece of writing, which can be either books or journal articles. These are modeled as entities and have an IS-A relationship, such that a Book is a Written Work and a Journal article is a Written Work. These relationships are modeled in our system by using single table inheritance. This means that both the Book entity as well as the Journal article entity are stored in the parent entity: Written Works. The works table contains some fields that are only relevant for each of its child entities. For example, only books have ISBN's while only journal articles contain DOI. Furthermore, this type of inheritance model requires the table Works to contain a field for the type of the entity of each row. In our system, the Works table has a field called work_type which either contains a 'b' or a 'j' representing book and journal article respectively.

The weak entity in our system is the Character entity. The Book entity, its owner entity set, and Character entity participate in a one-to-many relationship, that is a book may have many characters but a character only belongs to one book. Character entity also has a total participation constraint and is uniquely identified by its partial key character_id. Thus, a Character entity can be identified uniquely only by considering the primary key of the Book entity. There is a whole-part relationship, in which if a record of a book is deleted, then the characters belonging to that book must also be deleted.

A publisher, who can be uniquely identified by their publisher_id, can publish many works and a work can have many publishers (many-to-many relationship). In addition to this, every work must be published by a publisher (total participation). Furthermore, similar to the publisher, an Author, who can be identified by their author_id, can write many Works and a Work can be written by many authors (many-to-many relationship). Every work must have an author. Finally, a work can touch on many Subjects, which can be identified by the subject_id, while a Subject can also be in many different Works (many-to-many relationship). Each Work has to have a subject, and thus Work has a total participation constraint in this relationship. We have made relationship tables for all the many-to-many relationships, which include work_publishers, work_authors, and work_subjects. All of these tables contain a foreign key work_id, referencing the id in the Work table, and the foreign key subject_id, author_id, subject_id referencing the corresponding primary key 'id' in the table of Publishers, Authors, and Subjects respectively. Thus, the association tables in our system will hold two foreign keys.

Regarding the technical overview of our system, we used PostgreSQL as our SQL database. We also used Ruby to write a script that helped us alter the data into a usable and consistent format throughout the system along with populating it into our PostgreSQL database. Furthermore, we wrote a shell script for each public API we used that would simply call the API and store the contents of the call in a JSON file. For the books API each page/file contained 100 entries while for the journal article API each contained 1000 entries. We ended up using thousands of pages of data from each API, resulting in hundreds of thousands of records in our database. The DDL's for creating tables and triggers can be found in the SQL file called DDLPhase1.sql. The DDL for inserting into tables can be found in the Ruby script called works.rb. Since all of the data we used amounted to almost 1 GB, we were unable to include all of the files on our moodle submission. Instead we included a sample of about 100 files for both journal articles and books to give you an idea of the functionality of our system.

Data Model

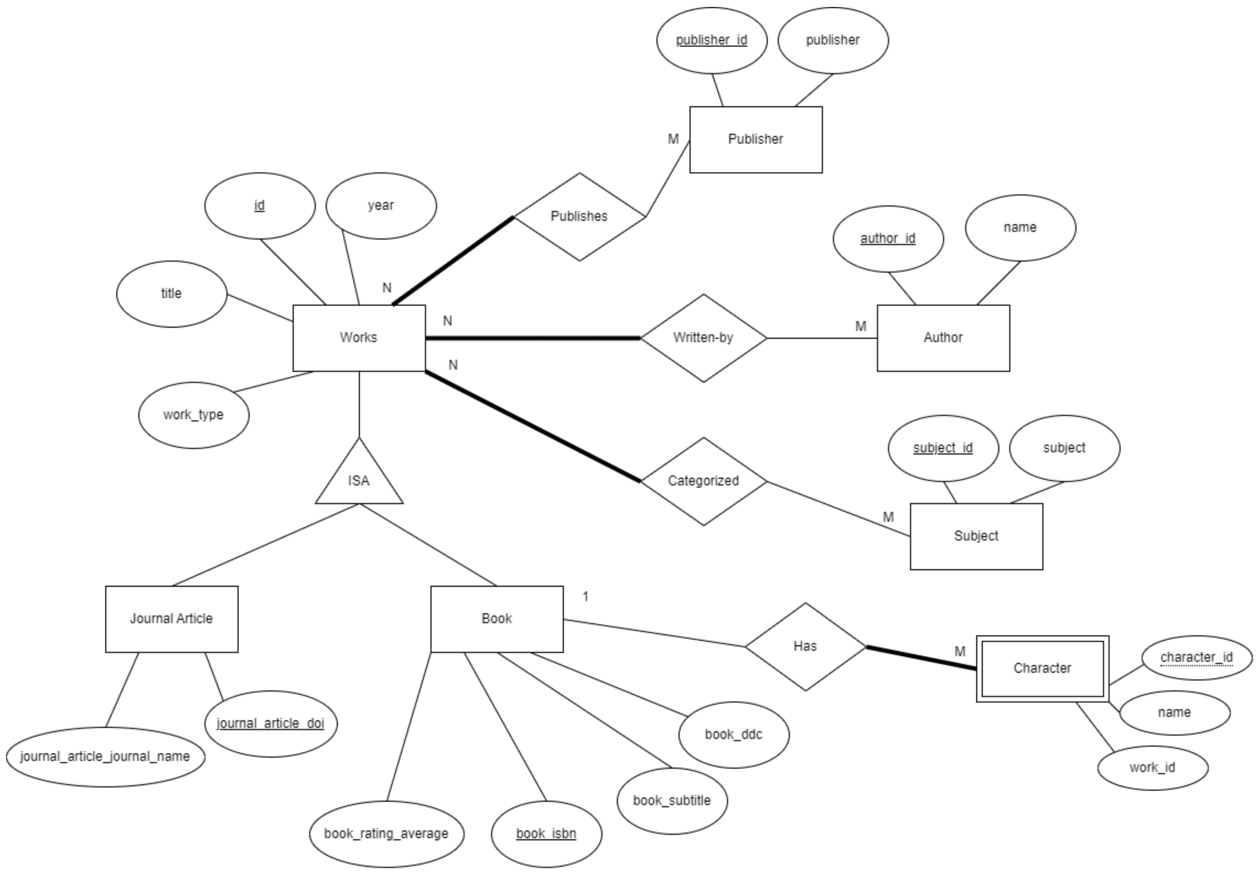


Figure 1: Entity Relationship Model for Works Database

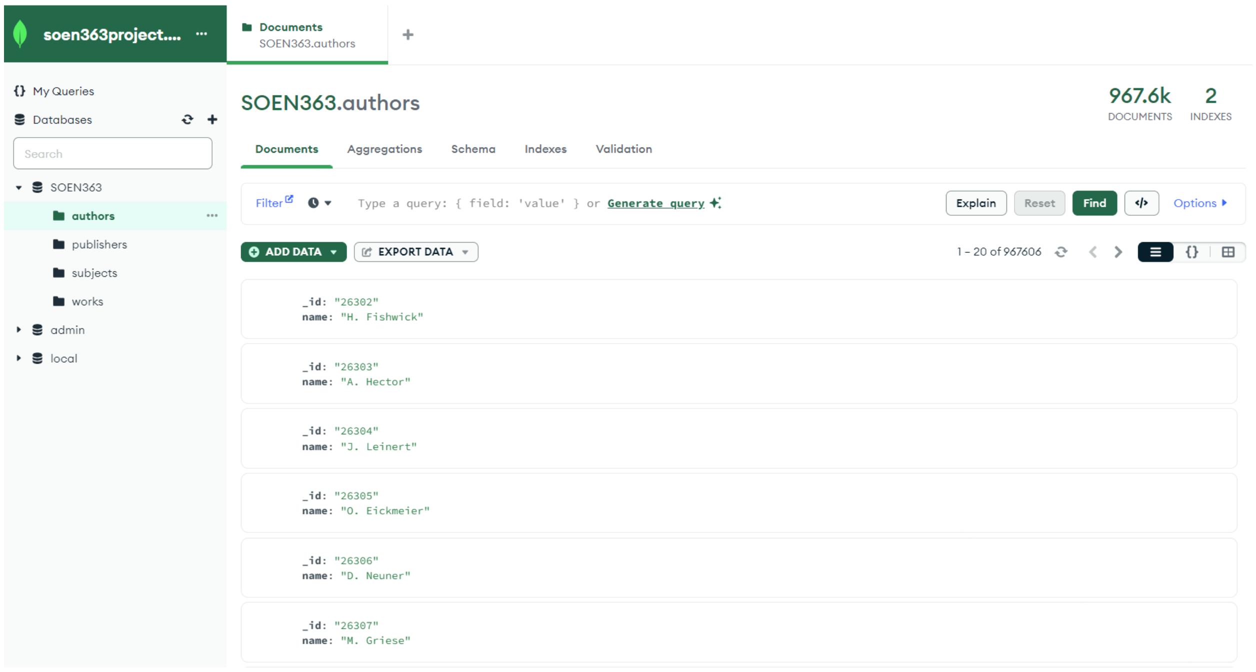
Challenges Faced in Populating the Data
While working on getting data for our works tables, we faced many challenges. First off, finding APIs that let us retrieve significant quantities of data without any limits or monetary fees was tough. Many platforms set rules on how much data we could get. After much research, we found two sources to get our data. Since in this project the database had to use at least two different sources, we also faced some challenges retrieving and altering the data from the API's so that the formats are consistent throughout the database. For example, the date attribute was given in a various number of ways in each API, we had to take that into consideration when populating the database. We wrote a ruby script to help us along that. Next, we realized that calling API's the number of times we needed was very taxing on the API, causing it to crash, go offline and glitch. We decided that instead of calling the API everytime we needed to repopulate (due to a change in our system model etc..), it was more efficient to simply call the API's once and store the responses in JSON files. In order to do so, we wrote two shell scripts that would retrieve a page of data from the respective API to store in files for each page. We then read from those JSON files from the script that we created as needed. However, calling from the API nonetheless was a very slow and time consuming process. Thus it took a long time to create the JSON files as well as run the script to populate the database from the files. These challenges made it clear that working with APIs and getting data into our database isn't always a straightforward task.

Sources for API's used:
https://openlibrary.org/swagger/docs#/search/read_search_json_search_json_get
https://api.crossref.org/swagger-ui/index.html#/Works/get_works

Phase 2:

Overview of the System

In Phase 2 of our project, we chose to use MongoDB as our NoSQL database. MongoDB is a document based NoSQL database that has collections that hold documents. In our specific system our database has four collections: works, publishers, authors, subjects. The works collection document contains a field for all of the attributes corresponding to the works table in our PostgreSQL database (year, book_isbn, journal_articles_doi, journal_articles_journal_name, title, book_subtitle, book_rating_average, book_ddc, work_type). On top of that, we have also implemented the weak entity "character" in the work collection. Works has an array attribute called characters that contains the names of all the characters associated to that specific work. This ensures that Character is treated as a weak entity as intended since it will be deleted if the parent entity (work) is deleted. Furthermore, the works collection also contains array attributes for authors, subjects, and publishers. Each of those arrays contain the id and name of the respective entity. The three other collections of our database are authors, subjects and publishers. Each of these collections contains an id and name for the respective entity, as seen on Figure 1. We have chosen to have separate collections of authors, subjects and publishers alongside as well as having them as array attributes within the works collection to implement a concept called denormalization. Denormalization involves combining data from multiple collections into one and duplicating data across collections to optimize read performance at the expense of increased storage space. Another benefit of using denormalization is reduced query complexity. Since one of the main objectives of this project is to query efficiently, that is why we chose to use denormalization even at the cost of increased storage space.



```
_id: "26302"
name: "H. Fishwick"

_id: "26303"
name: "A. Hector"

_id: "26304"
name: "J. Leinert"

_id: "26305"
name: "O. Eickmeier"

_id: "26306"
name: "D. Neuner"

_id: "26307"
name: "M. Griese"
```

Figure 2: Collections on MongoDB

Graph Model

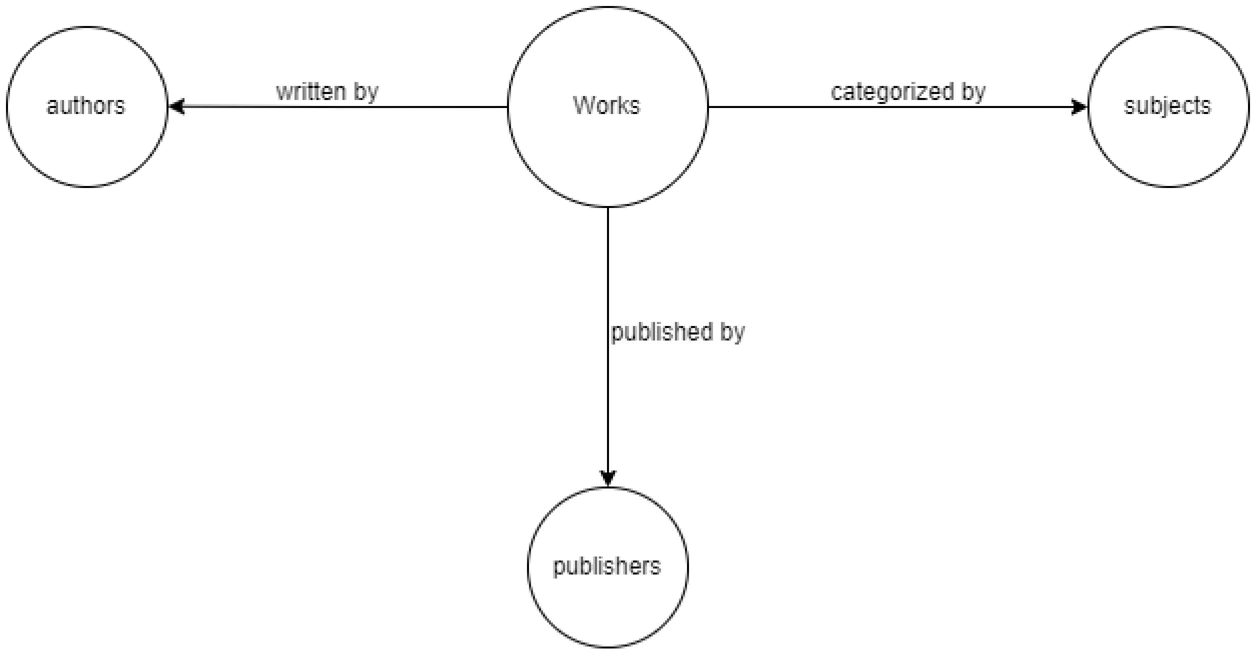


Figure 3: Graph Model for Works Database

Challenges faced when migrating the data:
In order to migrate our PostgreSQL database into a MongoDB we first had to alter our data model in order for it to be compatible with NoSQL format. We mapped out how to remodel our data as collections and documents instead of tables. One key difference between the data models from SQL and NoSQL is that the Character weak entity in SQL was kept in its own table and was referenced to its parent entity with foreign keys. However in the NoSQL model, we chose to include it as an array attribute of the parent entity instead. After having mapped out our model, we wrote a ruby script that took care of the migration. This script created the collections in Mongo, it then selected from our PostgreSQL tables and created documents for our collections from the data extracted. The biggest challenge we faced when migrating the data was the sheer time it took for our script to execute. Since our PostgreSQL database was large (around 500MB), the script took over two days to run to completion and migrate all of the data into MongoDB. Time was the biggest challenge we faced in Phase 2.

Conclusion

In conclusion, our project successfully transitioned through two distinct phases, each with its own set of challenges and accomplishments. Phase I of our project involved the implementation of a relational database for managing written works, including books and journal articles. We adopted a single table inheritance model to represent these entities efficiently, and various relationships such as one-to-many, many-to-many, and whole-part relationships were established between entities like Books, Characters, Authors, Publishers, and Subjects. PostgreSQL was selected as our SQL database, and we encountered challenges while populating the data, particularly in dealing with API limitations and data format consistency.

In Phase II, we transitioned to MongoDB as our NoSQL database, introducing collections and documents to store our data. We denormalized certain attributes by duplicating data across collections to optimize read performance and reduce query complexity. The migration of data from PostgreSQL to MongoDB presented its own set of challenges, primarily related to the time it took to complete the migration process due to the large size of our initial database.

Throughout both phases, we aimed to design and implement a robust database system that could efficiently handle large volumes of data, all while accommodating various relationships between entities. These challenges and practical applications have highlighted the complexity of working with databases and data migration, and emphasized the importance of careful planning and consideration of database models and technologies.