HSLU Hochschule Luzern

CREATING A BOOK RECOMMENDER SYSTEM

Database Project Report

16.06.2024

THE BOOKWORMS

Dominik Bacher - dominik.bachersuarez@stud.hslu.ch
Gwen Junod - gwen.junod@stud.hslu.ch
Vitalia Vedenikova - vitalia.vedenikova@stud.hslu.ch

Lucerne University of Applied Sciences

Master of Science in Applied Information and Data Science

W.DBM02.F24

Database Management for Data Scientists

Table of contents

1. Introduction & Context **1**
1.1 VM and Database Access Information 1
2. Project Idea & Use Case **2**
2.1. Decision support for value generation 2
2.2. Calculation of key figure used for decision support 2
2.3. Data sources 2
2.3.1. Books data 3
2.3.2. Reviews data 4
2.4. Database Technology 5
3. Data Model & Database Schema **6**
3.1. Initial Entity-Relationship Diagram 6
3.2 Normalization 9
3.2.1. First Normal Form (1NF) 9
3.2.2. Second Normal Form (2NF) 14
3.2.3. Third Normal Form (3NF) 17
3.3 Conceptual Model (ER-Diagram, Chen 1976) After Normalization 18
3.4. Final DB Schema After Normalization 20
3.3. Relationship between model and schema 20
3.4. Verbal description of the most important data classes and attributes 21
4. Loading & Transforming the Data **22**
4.1. System components, relationships and data flows incl. access 22
4.2. Data loading 23
4.2.1. Initial Loading 23
4.2.2. Duplicate Removal 27
4.3. Data transformations 27
4.3.3. Data Integrity 27
4.3.2. Vertical Partitioning 28
4.3.3. Adding BookId to the Ratings table 29
5. Analyzing & Evaluating Data **30**
5.1. Verbal description of query 30
5.2. Show connection between query and use case 31
5.3. Query in database syntax 31
6. Efficiency & Query Performance **35**
6.1. Analysis of runtime bottlenecks of your queries 35
6.2. Optimization measures used 35
6.2.1. Materialized Views and Execution plan 35
6.5.2. WHERE clauses 37
6.5.3. Indexes 38
6.5.4. Efficient Joins and Aggregations 38

6.5.5. Temporary Tables 38
6.3. Correlation between measures and runtime optimization 38
7. Visualization & Decision Support 39
7.1. Visualization of query results 39
7.2. Decision recommendations for the persona 42
7.3. Connection between visualization and original use case 42
8. Conclusions & Lessons Learned 43

1. Introduction & Context

In an era where books are available both in physical and digital formats, the task of choosing the right book among millions of options can be daunting. According to the World Intellectual Property Organization, hundreds of thousands of books were newly published in 2022 alone[1], and this does not account for the many self-published authors.

It is critical for platforms like Goodreads, which catalogs books as well as allow their users to search their databases and keep track of which books they are reading, to enhance their readers' experience to make them keep coming back onto their website.

One of their core services is to recommend new books to users once they rate books or mark them as read. To enhance this service, we propose an updated book recommender system constructed with a SQL database. It will make quick and relevant recommendations to readers and simplify their selection process through the use of ratings data.

As avid readers and data enthusiasts, we recognize the value of personalized recommendations in guiding individuals towards books that resonate with their preferences and reading habits. That is why we are excited to present this project which brings together our interests.

1.1 VM and Database Access Information

DELETED

[1]World Intellectual Property Organization (2023). The Global Publishing Industry in 2022. Geneva: WIPO. https://www.wipo.int/edocs/pubdocs/en/wipo-pub-1064-2023-2-en-the-global-publishing-industry-in-2022.pdf

2. Project Idea & Use Case

2.1. Decision support for value generation

Readers in search of a new book to read will only have to enter the title of a book they like, and they will receive suggestions of books rated more than 4 out of 5 by people who also liked the input book. If they are interested in a new genre which they have never read before, they can also indicate it and the book recommender system will output books of this new genre liked by the same group of people. This provides diverse choices to the user who can choose between familiar and entirely new types of books which he knows people similar to him have enjoyed in the past.

2.2. Calculation of key figure used for decision support

Our persona is Lisa, an avid teenage reader who wants to discover new books she has never heard of before, based on the ones she already likes. She likes horror and thriller novels, but recently she has taken an interest in the History genre.

The main figure used for decision support is the weighted score of each book in the results' output. It is calculated according to the following formula: *(number of good ratings * number of good ratings) / total number of ratings*. The weighting of the number of good ratings prevents generally popular books from appearing too often in the results, thus allowing the recommendations to be closely related to the input book.

If the user chooses to indicate a genre, then the formula changes as follows: (*(number of good ratings * number of good ratings) / total number of ratings)*7.* This weighing allows books of the indicated genres to appear in the top recommendations without taking over the whole list.

The suggested books are ordered by weighted score and filtered to display the 10 highest weighted scores.

2.3. Data sources

For this project, we will use two datasets downloaded from Kaggle and contained in csv files[2]. There are two distinct sources for these files :

[2] Bekheet, M. (2022). *Amazon Books Reviews*. Kaggle. https://www.kaggle.com/datasets/mohamedbakhet/amazon-books-reviews/data

1) Amazon review data for book reviews
2) Google Books API for information about the books

They will be used as-is, with no prior adjustments to be able to apply ELT methodology.

2.3.1. Books data

"Books_data.csv" contains metadata of over 200'000 books extracted from Google Books API and has the following attributes:

- **Title**: title of the book
- **description**: content of the book
- **authors**: names of the authors
- **image**: link to the cover image
- **previewLink**: link to Google Books
- **publisher**: name of the publisher
- **publishedDate**: date of publication, sometimes only year or year-month
- **infoLink**: sometimes same link as *previewLink*, sometimes link to download on Google Play
- **categories**: genre(s) of the book
- **ratingsCount**: average rating of the book (unclear where that comes from)

This dataset contains NAs in every column, with up to 36% of missing values in *publisher*. This is not a problem for this project because the recommendations will not be based on those attributes. If the recommender system was modified to take into account the author or the genre for example, it would be important to fill in the missing values. The genres in particular contained the usual "Thriller", "Romance", "Horror" but also got extremely specific with genres such as "Spanish-American War" which resembled themes more than genres. They also contained sub-genres: "Authors, Irish" refers to "Authors" as a genre and "Irish Authors" as a sub-genre.

Since the attributes will be retrieved in the output presenting the recommended books, some books could be presented to users with missing characteristics. As this is a minor inconvenience, it will not be corrected.

We noticed some inconsistencies in the values of *publishedDate*. Indeed, sometimes only the year of publishing of the book is recorded instead of the precise date. This is also only an inconvenience as it will not be used for recommendations.

Furthermore, not all books had reviews and we could have made the choice to keep only those which had at least one review. However, we realized that there might be users in the future who will want to post reviews for books that do not have any at the moment. Therefore, we kept all the books in the data sets, which follows the data requirements of this project and good practices to keep all the data we have; especially for future implementations.

2.3.2. Reviews data

"Books_rating.csv" contains millions of Amazon book reviews extracted from [Amazon review data]() and has the following attributes:

- **Id**: ID of the book
- **Title**: title of the book
- **Price**: price of the book
- **user_id**: ID of the user reviewing the book
- **profileName**: name of the user's profile
- **review/helpfulness**: helpfulness of the review (unclear how this ratio is calculated)
- **review/score**: score given to the book from 0 to 5
- **review/time**: time of the review
- **review/summary**: summary text of the review
- **review/text**: full text of the review

One might immediately notice that these are Amazon book reviews, and not Goodreads reviews. The reason these reviews are suitable for our database is because Amazon owns the company Goodreads, meaning that most of the books on Amazon appear on Goodreads as well.

There are missing values in columns *user_id* (up to 19%), but also in *profileName, Price* and *review/time*.

In terms of limitations, all missing *user_id* will be replaced with the unique identifier "1" since the decision support does not depend on returning specific users. Furthermore, it might actually be faster for SQL to search through less user ids. The other missing values are also not significant for querying.

2.4. Database Technology

MySQL was used as our database management system for this project, using **MySQL**

Workbench for database management, all in a **Windows based VM**. For visualizations of data analysis, we used **Metabase**.

Advantages:

- This system is ideal for handling large datasets due to its robust support for complex SQL queries and efficient data retrieval. It is free, open source and runs on multiple platforms.
- MySQL Workbench provides a user-friendly interface for designing, managing and querying databases with easy to understand visualizations of query results. It can be used to design and model database schemas, as well as monitor and optimize database performance. Lastly, it is very convenient for importing and exporting data and scripts.
- The remote Virtual Machine allows us to access the same dataset from any point and collaborate on the same project easily.
- Metabase makes it easy to implement visualizations in browsers for most user interactions and therefore facilitates decisions by users.

Disadvantages:

- MySQL, MySQL Workbench and Metabase’s learning curves can be steep for new users.
- MySQL Workbench is not very versatile for other database systems than MySQL.
- MySQL Workbench does not have version control like other workbenches.
- Metabase requires that the code developed in the Workbench be adapted, which can be difficult if the provided documentation lacks details.

3. Data Model & Database Schema

3.1. Initial Entity-Relationship Diagram

This is the initial Entity-Relationship Diagram of the datasets as they were initially defined in the SQL Workbench and the types and examples we gave to each attribute:

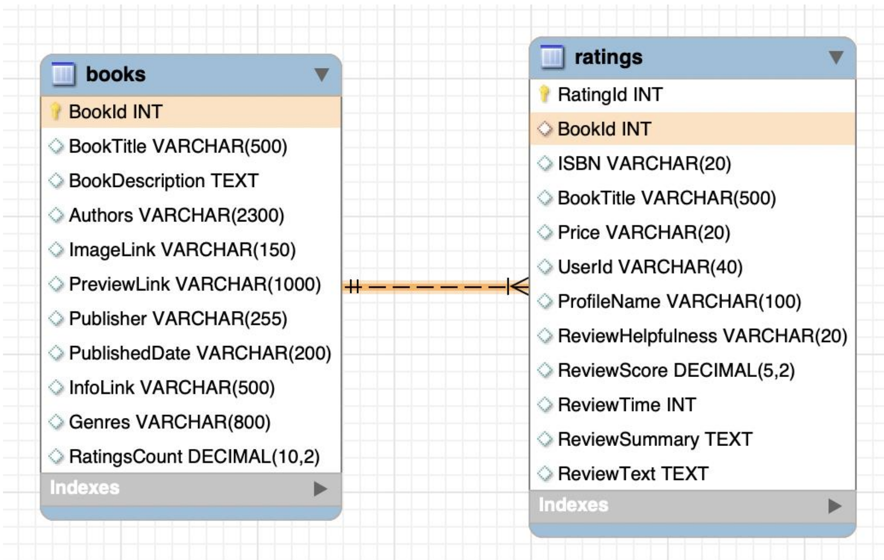

Figure 1: Initial ER diagram

The full query used is copied below. After creating the database “bookworms”, we used the **Data Definition Language (DDL)** specific command CREATE TABLE to create one table for *books* and another for *ratings*.

For the *Books* table, we used the type VARCHAR (character string values of variable length) to store data for *BookTitle, Authors, ImageLink, PreviewLink, Publisher, PublicationDate, InformationLink* and *Genres* where we did not anticipate to go above the limit of 8000 characters. Additionally, the data sets’ contents are entirely in English, so we decided that it was not necessary to use NVARCHAR and opted for VARCHAR which has better performance in terms of storage and retrieval. The *BookDescription* attribute is of type TEXT since book descriptions can be quite long. *RatingsCount* has the type INTEGER (the amount of reviews per book does not exceed 4895 but we also need to consider that the amount of reviews could far exceed this amount in the future). If there are no ratings, the DEFAULT is set to 0. Lastly, the primary key *BookId* is an INT type. We used the AUTO_INCREMENT command to automatically create numeric values for our primary key.

For the *Ratings* table, we defined *RatingId* as our primary key and proceeded the same way as for the primary key of *books*. Again, we used the VARCHAR type for *ISBN, BookTitle, Price, UserID, ProfileName* and *ReviewHelpfulness*. Since *Price* might contain the currency, we decided to use VARCHAR instead of DECIMAL. The *ReviewScore* was defined with the type DECIMAL with precision of 5 and scale of 2. This is to anticipate any averaging that might be done later on to get the average score for each book or weigh review scores as required in recommending specific genres. *ReviewTime* is given the DATETIME data type. It uses Unix time which will need to be converted during the loading phase. *ReviewSummary* and *ReviewText* can be especially long strings, so we give them the LONGTEXT type. Finally, we define a foreign key *BookID* with CONSTRAINT and make sure that it REFERENCES the table *Books*.

Creating the Initial DDL Schema

```sql
-- Database Schema:
CREATE DATABASE bookworms;

USE bookworms;

CREATE TABLE Books (
    BookId INT AUTO_INCREMENT PRIMARY KEY NOT NULL,
    BookTitle VARCHAR(500),
    BookDescription TEXT,
    Authors VARCHAR(2300),
    ImageLink VARCHAR(150),
    PreviewLink VARCHAR(1000),
    Publisher VARCHAR(255),
    PublishedDate VARCHAR(200),
    InfoLink VARCHAR(500),
    Genres VARCHAR(800),
    RatingsCount INT DEFAULT 0
);
CREATE TABLE Ratings (
    RatingId INT AUTO_INCREMENT PRIMARY KEY NOT NULL,
    BookId INT,
    ISBN VARCHAR(20),
    BookTitle VARCHAR(500),
    Price VARCHAR(20),
    UserId VARCHAR(40),
    ProfileName VARCHAR(100),
    ReviewHelpfulness VARCHAR(20),
    ReviewScore DECIMAL(5,2),
    ReviewTime DATETIME,
    ReviewSummary LONGTEXT,
    ReviewText LONGTEXT,
    CONSTRAINT fk_book_id FOREIGN KEY (BookId) REFERENCES Books(BookId)
);
```

3.2 Normalization

To achieve our Conceptual Model and DB Schema, we normalize the data. This is an important step, since this will allow us to study dependencies within tables in order to avoid redundant information and resulting anomalies[3].

3.2.1. First Normal Form (1NF)

"A table is in the first normal form when the domains of the attributes are atomic"[4]. Each cell must have a unique value (i.e. no sets, lists or repetitive groups) and all the cells from each column must have the same type of value.

All entries in the database are already a specific type per column. All rows are uniquely identified either by the column *BookId* or the column *ReviewId*.

The *Books* table contains two columns with non atomic values: *Authors* and *Genres*. It is converted to the first normal form by creating tables *Authors* and *Genres.* This is achieved with relationship sets *wrote* and *has* (respectively the connecting tables *BookHasAuthor* and *BookHasGenre* in the DB Schema) which create a separate tuple for each book in terms of genre and author.

This query is performed using temporary tables with split authors and genres, joins, indices and SQL iterations (the iteration was obtained with the help of ChatGPT).

We copy the code here for the sake of clarity as it is quite long and complex. Our use of **Data Definition Language (DDL)** is commented in detail at each step of the process so that our explanations are clear and easy to follow :

---

[3] Michael Kaufmann, Andreas Meier (2023). SQL and NoSQL Databases. Springer International, p.36
[4] Michael Kaufmann, Andreas Meier (2023). SQL and NoSQL Databases. Springer International, p.38

1NF for Authors

```sql
-- Step 1) Create Tables for Normalization
-- Author is a new table and BookHasAuthor is a new connecting table between the Author and Book tables.
CREATE TABLE Author (
        Author_Id INT AUTO_INCREMENT PRIMARY KEY,
        Name VARCHAR(2300) NOT NULL);
CREATE TABLE BookHasAuthor (
        BookId INT,
        AuthorId INT,
        PRIMARY KEY (BookId, AuthorId),
        FOREIGN KEY (BookId) REFERENCES Books(BookId),
        FOREIGN KEY (AuthorId) REFERENCES Author(AuthorId));

-- Step 2) Create a temporary table for Cleaning Authors
-- A temporary table TempAuthors is created to clean up author names by removing unwanted characters coming from a Python list syntax (brackets, quotes). This table stores the cleaned author names along with their respective book IDs.
CREATE TEMPORARY TABLE TempAuthors (
        BookId INT,
        CleanAuthor VARCHAR(2300));
INSERT INTO TempAuthors (BookId, CleanAuthor)
SELECT BookId,
REPLACE(REPLACE(REPLACE(REPLACE(Authors, '[', ''), ']', ''), '''', ''), '&', '') AS CleanedAuthors
FROM Books;
```

1NF for Genres

```sql
-- Step 1) Create tables for Normalization
-- Genre is a new table and BookHasGenre is a new connecting table between the Genre and Book tables.
CREATE TABLE Genre (
        GenreId INT AUTO_INCREMENT PRIMARY KEY,
        GenreName VARCHAR(800) NOT NULL);
CREATE TABLE BookHasGenre (
        BookId INT,
        GenreId INT,
        PRIMARY KEY (BookId, GenreId),
        FOREIGN KEY (BookId) REFERENCES Books(BookId),
        FOREIGN KEY (GenreId) REFERENCES Genre(GenreId));

-- Step 2) Create a temporary table for Cleaning Genres
-- A temporary table TempGenres is created to clean up genre names by removing unwanted characters coming from a Python list syntax (brackets, quotes). This table stores the cleaned genre names along with their respective book IDs.
CREATE TEMPORARY TABLE TempGenres (
        BookId INT,
        CleanGenre VARCHAR(800));
INSERT INTO TempGenres (BookId, CleanGenre)
SELECT BookId,
REPLACE(REPLACE(REPLACE(REPLACE(Genres, '[', ''), ']', ''), '''', ''),'"', '') AS CleanedGenres
FROM Books;

-- Step 3) Insert Unique Genres into Genre
-- This part extracts unique genre names from TempGenres and inserts them into the Genre table. It uses a cross join with a generated
```

```sql
-- Step 3) Insert Unique Authors into the Author Table
-- This part extracts unique author names from TempAuthors and inserts them into the Author table. It uses a cross join with a generated sequence of numbers to handle cases where multiple authors are listed in a single string. The ON DUPLICATE KEY UPDATE clause ensures that duplicate author names are not inserted.
-- (Query created with the help of ChatGPT, mainly to create the iteration loop)
INSERT INTO Author (AuthorName)
SELECT DISTINCT
TRIM(REPLACE(REPLACE(SUBSTRING_INDEX(SUBSTRING_INDEX(t.CleanAuthor, ',', numbers.n), ',', -1), '[', ''), ']', '')) AS author
FROM TempAuthors t
JOIN (
    SELECT n1.N + n2.N * 10 + 1 AS n
    FROM (SELECT 0 AS N UNION ALL SELECT 1 UNION ALL SELECT 2 UNION ALL SELECT 3 UNION ALL SELECT 4 UNION ALL SELECT 5 UNION ALL SELECT 6 UNION ALL SELECT 7 UNION ALL SELECT 8 UNION ALL SELECT 9) n1,
         (SELECT 0 AS N UNION ALL SELECT 1 UNION ALL SELECT 2 UNION ALL SELECT 3 UNION ALL SELECT 4 UNION ALL SELECT 5 UNION ALL SELECT 6 UNION ALL SELECT 7 UNION ALL SELECT 8 UNION ALL SELECT 9) n2
    ORDER BY n
) numbers
WHERE numbers.n <= 1 + (LENGTH(t.CleanAuthor) - LENGTH(REPLACE(t.CleanAuthor, ',', '')))
ON DUPLICATE KEY UPDATE AuthorName = AuthorName;



-- Step 4) Split Authors and Insert Relationships
```

```sql
sequence of numbers to handle cases where multiple genres are listed in a single string. The ON DUPLICATE KEY UPDATE clause ensures that duplicate genre names are not inserted.
-- (Query created with the help of ChatGPT)

INSERT INTO Genre (GenreName)
SELECT DISTINCT
TRIM(REPLACE(REPLACE(SUBSTRING_INDEX(SUBSTRING_INDEX(t.CleanGenre, ',', numbers.n), ',', -1), '[', ''), ']', '')) AS genre
FROM TempGenres t
JOIN (
    SELECT n1.N + n2.N * 10 + 1 AS n
    FROM (SELECT 0 AS N UNION ALL SELECT 1 UNION ALL SELECT 2 UNION ALL SELECT 3 UNION ALL SELECT 4 UNION ALL SELECT 5 UNION ALL SELECT 6 UNION ALL SELECT 7 UNION ALL SELECT 8 UNION ALL SELECT 9) n1,
         (SELECT 0 AS N UNION ALL SELECT 1 UNION ALL SELECT 2 UNION ALL SELECT 3 UNION ALL SELECT 4 UNION ALL SELECT 5 UNION ALL SELECT 6 UNION ALL SELECT 7 UNION ALL SELECT 8 UNION ALL SELECT 9) n2
    ORDER BY n
) numbers
WHERE numbers.n <= 1 + (LENGTH(t.CleanGenre) - LENGTH(REPLACE(t.CleanGenre, ',', '')))
ON DUPLICATE KEY UPDATE GenreName = GenreName;



-- Step 4) Split Genres and Insert them into
```

```
-- This part creates another temporary table SplitAuthors to split the cleaned author names into individual authors and associate them with their respective book IDs. Indexes are created on the SplitAuthors table to improve join performance and agree with the Normalization. Then it inserts the book-author relationships into the BookHasAuthor table by joining SplitAuthors with the Author table on the author names.
CREATE TEMPORARY TABLE SplitAuthors AS
SELECT
    t.BookId,

TRIM(REPLACE(REPLACE(SUBSTRING_INDEX(SUBSTRING_INDEX(t.CleanAuthor, ',', numbers.n), ',', -1), '[', ''), ']', '')) AS AuthorName
FROM TempAuthors t
JOIN (
    SELECT n1.N + n2.N * 10 + 1 AS n
    FROM (SELECT 0 AS N UNION ALL SELECT 1 UNION ALL SELECT 2 UNION ALL SELECT 3 UNION ALL SELECT 4 UNION ALL SELECT 5 UNION ALL SELECT 6 UNION ALL SELECT 7 UNION ALL SELECT 8 UNION ALL SELECT 9) n1,
         (SELECT 0 AS N UNION ALL SELECT 1 UNION ALL SELECT 2 UNION ALL SELECT 3 UNION ALL SELECT 4 UNION ALL SELECT 5 UNION ALL SELECT 6 UNION ALL SELECT 7 UNION ALL SELECT 8 UNION ALL SELECT 9) n2
    ORDER BY n
) numbers
WHERE numbers.n <= 1 + (LENGTH(t.CleanAuthor) - LENGTH(REPLACE(t.CleanAuthor, ',', '')))
AND
TRIM(REPLACE(REPLACE(SUBSTRING_INDEX(SUBSTRING_INDEX(t.CleanAuthor, ',', numbers.n), ',', -1), '[', ''), ']', '')) <> '';
CREATE INDEX idxSplitAuthorsAuthorName ON SplitAuthors(AuthorName(255));
```

```
Individual Rows
-- This step creates another temporary table SplitGenres to split the cleaned genre names into individual genres and associate them with their respective book IDs. It uses a similar cross join with a generated sequence of numbers to handle multiple genres in a single string. Then it inserts the book-genre relationships into the BookHasGenre table by joining SplitGenres with the Genre table on the genre names.
CREATE TEMPORARY TABLE SplitGenres AS
SELECT
    t.BookId,

TRIM(REPLACE(REPLACE(SUBSTRING_INDEX(SUBSTRING_INDEX(t.CleanGenre, ',', numbers.n), ',', -1), '[', ''), ']', '')) AS GenreName
FROM TempGenres t
JOIN (
    SELECT n1.N + n2.N * 10 + 1 AS n
    FROM (SELECT 0 AS N UNION ALL SELECT 1 UNION ALL SELECT 2 UNION ALL SELECT 3 UNION ALL SELECT 4 UNION ALL SELECT 5 UNION ALL SELECT 6 UNION ALL SELECT 7 UNION ALL SELECT 8 UNION ALL SELECT 9) n1,
         (SELECT 0 AS N UNION ALL SELECT 1 UNION ALL SELECT 2 UNION ALL SELECT 3 UNION ALL SELECT 4 UNION ALL SELECT 5 UNION ALL SELECT 6 UNION ALL SELECT 7 UNION ALL SELECT 8 UNION ALL SELECT 9) n2
    ORDER BY n
) numbers
WHERE numbers.n <= 1 + (LENGTH(t.CleanGenre) - LENGTH(REPLACE(t.CleanGenre, ',', '')))
AND
TRIM(REPLACE(REPLACE(SUBSTRING_INDEX(SUBSTRING_INDEX(t.CleanGenre, ',', numbers.n), ',', -1), '[', ''), ']', '')) <> '';
CREATE INDEX idxSplitGenresGenreName ON
```

```sql
CREATE INDEX idxSplitAuthorsBookId ON
SplitAuthors(BookId);

-- Step 5) Insert book-author relationship
INSERT INTO BookHasAuthor (BookId, AuthorId)
SELECT DISTINCT s.BookId, a.AuthorId
FROM SplitAuthors s
JOIN Author a ON s.AuthorName = a.AuthorName;


-- Step 6) Cleanup
-- The Authors column is removed from the
Books table since it is now normalized.


DROP TEMPORARY TABLE SplitAuthors;
DROP TEMPORARY TABLE TempAuthors;
ALTER TABLE Books DROP COLUMN Authors;

-- Step 7) Verification
-- A verification query is run to check if
the book "Anthology of American Folk Music"
has the correct authors associated with it
(It should return "Ross Hair" and "Thomas
Ruys Smith").
SELECT a.AuthorName
FROM Books b
JOIN BookHasAuthor bha ON b.BookId = bha.BookId
JOIN Author a ON bha.AuthorId = a.AuthorId
WHERE b.BookTitle = 'Anthology of american folk
music';
```

```sql
SplitGenres(GenreName(255));
CREATE INDEX idxSplitGenresBookId ON
SplitGenres(BookId);

-- Step 5) Insert book-genre relationship
INSERT INTO BookHasGenre (BookId, GenreId)
SELECT DISTINCT s.BookId, g.GenreId
FROM SplitGenres s
JOIN Genre g ON s.GenreName = g.GenreName;


-- Step 6) Cleanup
-- The Genres column is removed from the
Books table since it is now normalized.

DROP TEMPORARY TABLE SplitGenres;
DROP TEMPORARY TABLE TempGenres;
ALTER TABLE Books DROP COLUMN Genres;


-- Step 7) Verification
-- A verification query is run to check if
the book "Anthology of American Folk Music"
has the correct genres associated with it
(It should return "Music").
SELECT g.GenreName AS GenreName
FROM Books b
JOIN BookHasGenre bhg ON b.BookId = bhg.BookId
JOIN Genre g ON bhg.GenreId = g.GenreId
WHERE b.BookTitle = 'Anthology of american folk
music';
```

3.2.2. Second Normal Form (2NF)

The second normal form states that all non-key attributes should be fully functionally dependent on specific primary keys[5].

In the case of the *Books* table, the *Publisher* attribute does not depend on each unique *BookId*, which means that it should be moved to its own table and only be related with the *Books* table via a foreign key.

In the case of the *Ratings* table, the attributes *UserId* and *ProfileName* are not dependent on the *RatingId* primary key; this means they will need to be moved to a new table called *Users* which will be linked via a foreign key to *Ratings*.

Also, for the *Ratings* table, both the *ISBN* and *Price* attributes are actually related to the book and not to the ratings given. For this reason these two attributes are to be moved to the *Books* table and removed from the *Reviews* table.

We copy the queries here for the sake of clarity. Our use of Data Definition Language (DDL) is commented in detail at each step of the process so that our explanations are easy to follow :

2NF for Publishers

```
-- ATTRIBUTE 1: Books table and publisher
normalization
-- Step 1) we create the table Publishers
CREATE TABLE Publishers (
    PublisherId INT AUTO_INCREMENT PRIMARY KEY,
    PublisherName VARCHAR(255) NOT NULL
);

-- Step 2) Extracting Unique Publishers
-- A temporary table TempPublishers is
created to hold unique publisher names
extracted from the Books table. Then inserts
the unique publisher names from
TempPublishers into the Publishers table.
Furthermore, the new column PublisherId is
added to the Books table to hold the foreign
key reference to the Publishers table.
```

2NF for Ratings

```
-- ATTRIBUTE 1 : Users Table normalization
-- Step 1) we create the table Users
CREATE TABLE Users (
    UserId INT AUTO_INCREMENT PRIMARY KEY,
    UserCode VARCHAR(40) NOT NULL,
    ProfileName VARCHAR(100),
    UNIQUE(UserCode)
);

-- Step 2) Extracting Unique Users
-- A temporary table TempUsers is created to
hold unique user IDs and their profile names
extracted from the Ratings table.
CREATE TEMPORARY TABLE TempUsers AS
SELECT DISTINCT UserId AS UserCode, ProfileName
FROM Ratings;
```

[5] Michael Kaufmann, Andreas Meier (2023). SQL and NoSQL Databases. Springer International, p.38

```sql
CREATE TEMPORARY TABLE TempPublishers AS
SELECT DISTINCT Publisher AS PublisherName
FROM Books;
INSERT INTO Publishers (PublisherName)
SELECT PublisherName FROM TempPublishers;
DROP TEMPORARY TABLE TempPublishers;
ALTER TABLE Books ADD PublisherId INT;

-- Step 3) Creating Temporary Table for Books
Update
CREATE TEMPORARY TABLE TempBooksUpdate AS
SELECT b.BookId, p.PublisherId
FROM Books b
JOIN Publishers p ON b.Publisher =
p.PublisherName;
CREATE INDEX idx_temp_books_update ON
TempBooksUpdate(BookId);

-- Step 4) Update Books
-- This step updates the Books table by
setting the PublisherId column to the
corresponding PublisherId from the
TempBooksUpdate table.
UPDATE Books b
JOIN TempBooksUpdate t ON b.BookId = t.BookId
SET b.PublisherId = t.PublisherId;
DROP TEMPORARY TABLE TempBooksUpdate;

-- Step 5) Remove the Original Publisher
Column and Add The Foreign Key Constraint
ALTER TABLE Books DROP COLUMN Publisher;
ALTER TABLE Books ADD CONSTRAINT
fk_books_publisher_id FOREIGN KEY (PublisherId)
REFERENCES Publishers(PublisherId);
```

```sql
-- Step 3) Ensure uniqueness in TempUsers
(Gotten with the help of ChatGPT)
-- Another temporary table TempUniqueUsers
is created to ensure uniqueness. It groups
by UserCode and selects the minimum
ProfileName for each UserCode, ensuring that
each user is represented only once.
CREATE TEMPORARY TABLE TempUniqueUsers AS
SELECT UserCode, MIN(ProfileName) AS ProfileName
FROM TempUsers
GROUP BY UserCode;

-- Step 4) Insert distinct users into the
Users table
-- The following inserts the distinct users
from TempUniqueUsers into the Users table.
The IGNORE keyword ensures that any
duplicate entries are ignored.
INSERT IGNORE INTO Users (UserCode, ProfileName)
SELECT UserCode, ProfileName FROM
TempUniqueUsers;
DROP TEMPORARY TABLE TempUsers;
DROP TEMPORARY TABLE TempUniqueUsers;

-- Step 5) Adding NewUserId Column to
Ratings Table
-- A new column NewUserId is added to the
Ratings table to store the foreign key
reference to the Users table.
ALTER TABLE Ratings ADD NewUserId INT;

-- Step 6) Creating Temporary Table for
Ratings Update
-- A temporary table TempRatingsUpdate is
created to map each rating to the
corresponding UserId from the Users table.
CREATE TEMPORARY TABLE TempRatingsUpdate AS
SELECT r.RatingId, u.UserId AS NewUserId
```

```sql
FROM Ratings r
JOIN Users u ON r.UserId = u.UserCode;
CREATE INDEX idx_temp_ratings_update ON
TempRatingsUpdate(RatingId);

-- Step 7) Update the Ratings table with the
new UserId
-- This step updates the Ratings table by
setting the NewUserId column to the
corresponding UserId from the
TempRatingsUpdate table.
UPDATE Ratings r
JOIN TempRatingsUpdate t ON r.RatingId =
t.RatingId
SET r.NewUserId = t.NewUserId;

DROP TEMPORARY TABLE TempRatingsUpdate;
ALTER TABLE Ratings DROP COLUMN UserId;
ALTER TABLE Ratings DROP COLUMN ProfileName;

-- Step 8) Rename column back to UserId
-- The NewUserId column is renamed back to
UserId. And foreign key constraint is added
to the UserId column to reference the UserId
in the Users table.
ALTER TABLE Ratings CHANGE NewUserId UserId INT;
ALTER TABLE Ratings ADD CONSTRAINT
fk_ratings_user_id FOREIGN KEY (UserId)
REFERENCES Users(UserId);


-- ATTRIBUTE 2: Normalize ISBN and Price,
Step 1) Move the ISBN and Price from the
Ratings table to the Books table
ALTER TABLE Books ADD ISBN VARCHAR(20);
ALTER TABLE Books ADD Price VARCHAR(20);

--Step 2)  Create Temporary Table for Book
Details
```

```sql
-- A temporary table TempBookDetails is created to store BookId, ISBN, and Price from the Ratings table where both ISBN and Price are not NULL.
CREATE TEMPORARY TABLE TempBookDetails AS
SELECT r.BookId, r.ISBN, r.Price
FROM Ratings r
WHERE r.ISBN IS NOT NULL AND r.Price IS NOT NULL;

CREATE INDEX idx_temp_bookdetails_bookid ON
TempBookDetails(BookId);

-- Step 3) Update Books Table with ISBN and Price
-- This step updates the Books table by setting the ISBN and Price columns to the corresponding values from the TempBookDetails table.
UPDATE Books b
JOIN TempBookDetails t ON b.BookId = t.BookId
SET b.ISBN = t.ISBN, b.Price = t.Price;

DROP TEMPORARY TABLE TempBookDetails;

ALTER TABLE Ratings DROP COLUMN ISBN;
ALTER TABLE Ratings DROP COLUMN Price;
```

3.2.3. Third Normal Form (3NF)

The third normal form states that all non-key attributes are functionally dependent only on the primary key, meaning no transitive dependencies exist[6].

For the case of all tables, all the attributes are already only dependent on the primary key of each table and no transitive dependencies appear to exist.

[6] Michael Kaufmann, Andreas Meier (2023). SQL and NoSQL Databases. Springer International, p.40

3.3 Conceptual Model (ER-Diagram, Chen 1976) After Normalization

Our Conceptual Model expands on the initial Entity-Relationship Diagram of the datasets as they were loaded. It takes into consideration the necessary normalizations as well as missing values in the data sets. Attributes correspond to the ones already described in detail in section 2.3 “Data Sources”.

Here are more details on the conceptual model itself with *Books* and *Ratings* as the two main entity sets we will focus on to easily understand the logic of the model:

- *Books* is an entity set of all books with the attributes *Book ID #* (Identification Key), *Publisher ID #* (Foreign Key), *Title, Date of Publishing, Amount of Ratings, ISBN* and *Price*. In terms of relationships:
    - Each book has its own unique Metadata contained in the *Meta Data* entity set. This relationship is mapped by a unique foreign key attribute *Book ID #*. Its other attributes are *Book Description, Image Link, Preview Link* and *Info Link*.
    - Each book may or may not have its own unique publisher contained in the *Publishers* entity set. This is because some authors do not use publishers and this information is sometimes missing in our data sets. This relationship is mapped by a unique identification key attribute *Publisher ID #*. Its other attribute is *Name*.
    - Each book was written by one or multiple authors contained in the *Authors* entity set identified by the identification key *Author ID #* and with a *Name* attribute. Each author has written at least one book. As a consequence, the relationship set *wrote* connecting *Authors* to *Books* contains the foreign key attributes *Book ID #* and *Author ID #*.
    - In the data set, some books have missing genres, so we say that there is none, one or multiple genres which may describe one or multiple books. The *Genres* entity set contains *Genre ID #* as the identification key and *Name* as an attribute. Its complex-complex relationship set with *Books* contains the foreign key attributes *Book ID #* and *Genre ID #*.

- Each book has one or multiple ratings, however, as books will keep being added to the data set in the future, there might be some that do not have reviews yet. Therefore, each rating is associated to exactly one book through the use of *Book ID #* as a foreign key, but books can have none, one or several ratings. The identification key of *Ratings* is *Ratings*

ID #, and its other attributes are *Review Helpfulness, Review Score* and *Review Time*. It also has another foreign key *User ID #*. In terms of other relationships:

- The *Users* entity set has *User ID #* as its primary key attribute, as well as attributes *User Code* and *Profile Name.* Indeed, each review is associated with one user, but each user might not necessarily have posted a review.
- Like *Books*, each rating in the *Ratings* entity set has its own unique meta data in a *Meta Data* entity set. This relationship is mapped by a unique foreign key attribute *Rating ID #*. Its other attributes are *Review Summary* and *Review Text.*

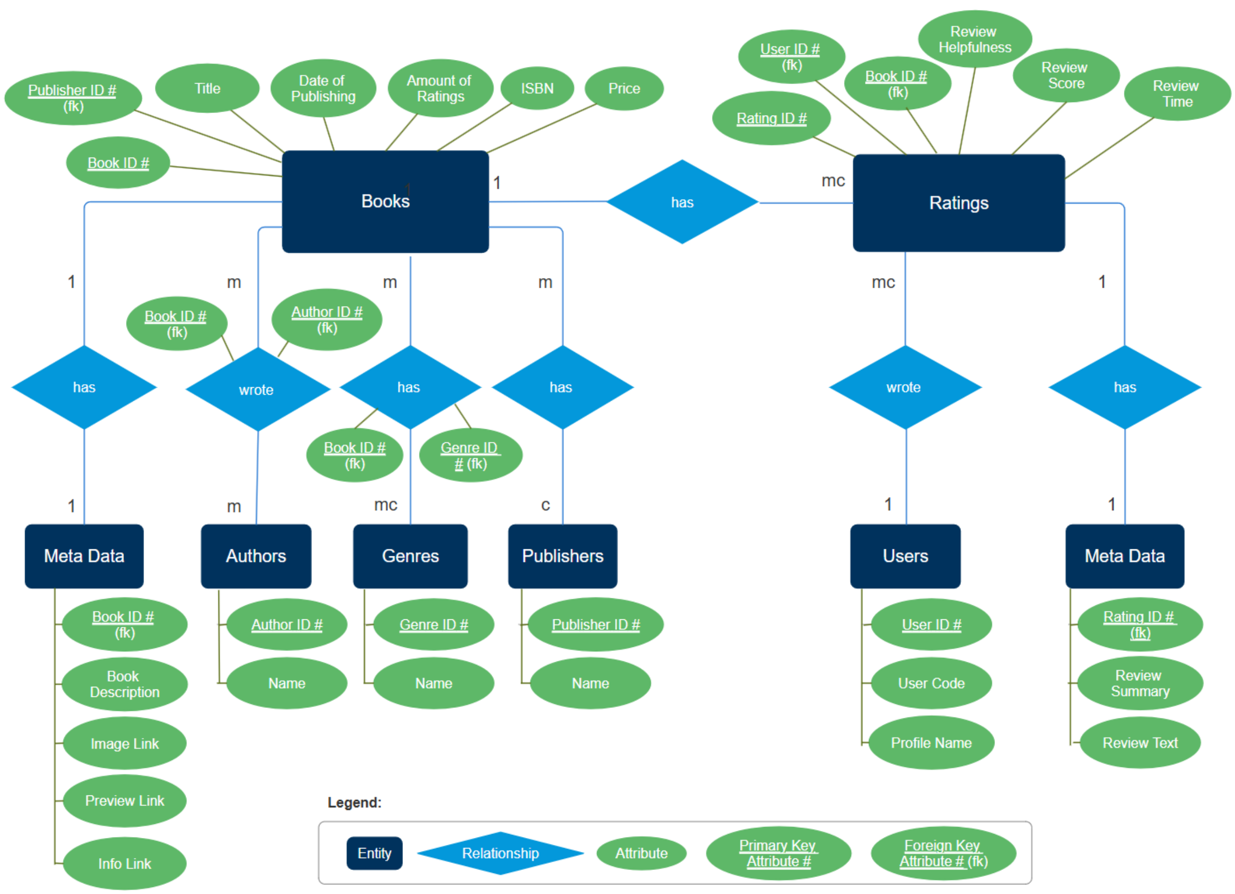

Figure 2: Entity Relationship Diagram (Chen Notation)

3.4. Final DB Schema After Normalization

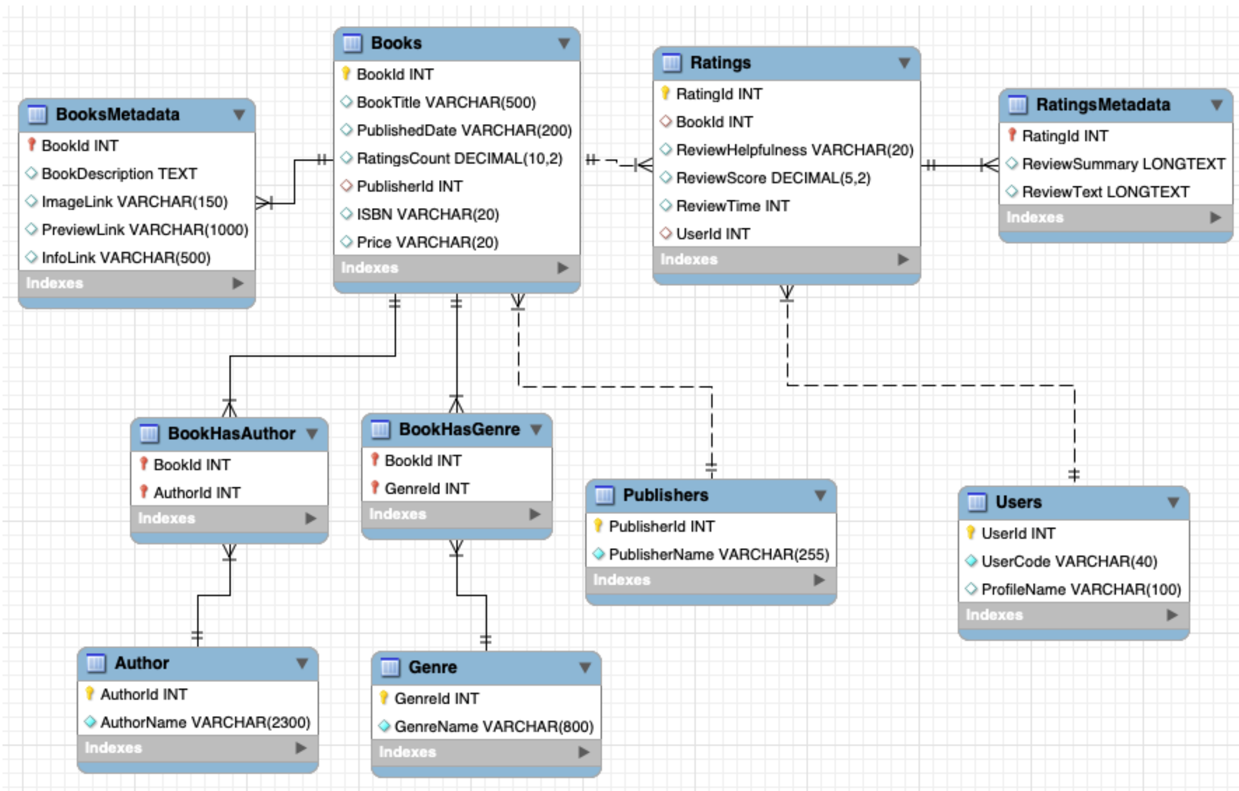

Figure 3: Entity Relationship (ER) diagram after DB schema and DDL

3.3. Relationship between model and schema

Entity sets and relationship sets have different names in the DB schema, but otherwise the logic remains the same:

- *BooksMetadata* contains metadata for the books.
- *RatingsMetadata* contains metadata for the ratings.
- *BookHasAuthor* is a connecting table which corresponds to the *has* relationship set from the Entity Relationship Diagram between *Books* and *Author.*
- *BookHasGenre* is a connecting table which corresponds to the *has* relationship set from the Entity Relationship Diagram between *Books* and *Genre*.
- In the DB Schema, it is immediately visible that the optional relationships are the ones between *Books* and *Genre*, *Books* and *Publisher* and *Ratings* and *Users* for reasons already detailed in section 3.3.

3.4. Verbal description of the most important data classes and attributes

The **Books** entity is important for the recommendation system as each book is uniquely identified by *BookId*, and includes attributes shown for the recommendation query which are: *BookTitle*, *ISBN*, *Price* and *PublishedDate*. For joining with other tables, the attribute *PublisherId* is used.

The **BooksMetadata** entity has additional details shown for the recommender query. This is linked by the *BookId* attribute. Important attributes shown in the query are the *ImageLink* which contains the cover image for the book, and *BookDescription* to give a textual description of the book.

The **Ratings** entity holds all the user ratings data, where each rating is identified by the *RatingId* and includes the foreign key *BookId* to refer to the book it is rated. It also includes the attribute *ReviewScore* which is very important as it determines if a book is highly rated or not, which is one of the main parts of the recommender algorithm. And the attribute *UserId* which will be needed to track the books rated equally by other users.

The **Users** entity is used to track the users who rate books. It is uniquely identified by *UserId*, which is how the recommender algorithm associates the ratings with specific users and identifies users who liked particular books.

The **Genres** entity will categorize books as an optional argument for the query. Each genre is identified by a unique *GenreId* and a *GenreName*. Then it is linked to a book via the **BookHasGenre** table using both the *GenreId* and the *BookId*.

The **Authors** entity works similarly to the **Genre** table. With the unique identifier being *AuthorId* and *AuthorName*, which are linked to the **BookHasAuthor** table using the *AuthorId* and the *BookId*.

The **Publisher** entity contains unique publishers, which is linked through the foreign key *PublisherId* in the **Books** table and *PublisherId* as its primary key in this **Publisher** table. The *PublisherName* is the attribute collected to show to the user for the recommender query.

4. Loading & Transforming the Data

4.1. System components, relationships and data flows incl. access

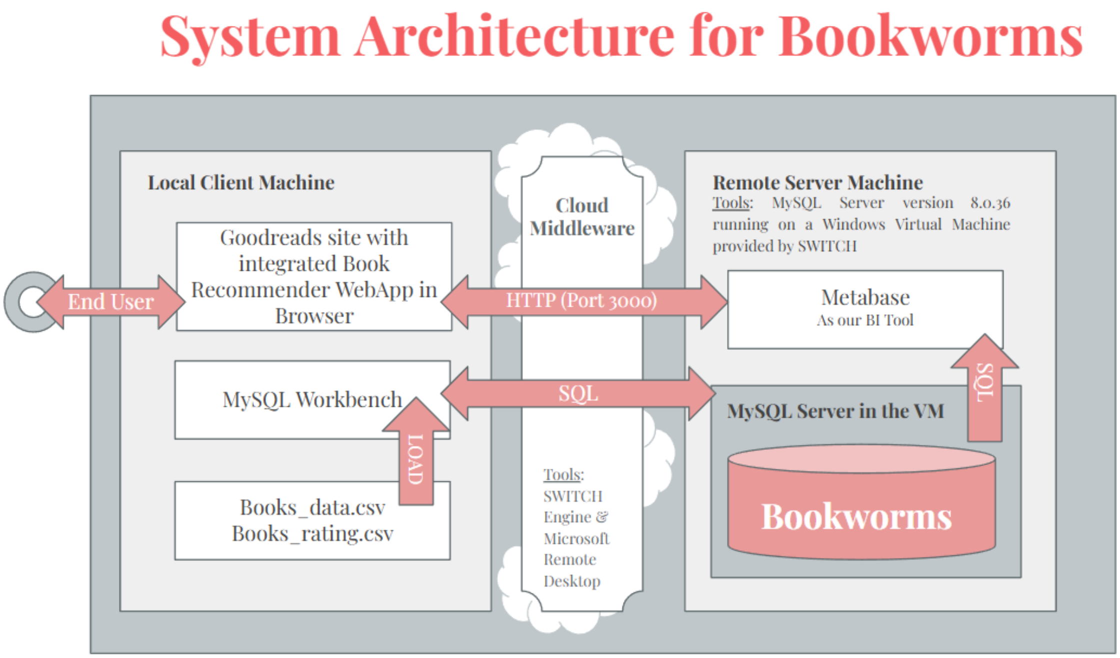

Figure 4: System Architecture

As our database server, we used the MySQL server located in our Virtual Machine. This is where data storage, retrieval and manipulation using SQL queries took place. SQL was our language interface and as data administrators we used it to interact with the server from MySQL Workbench situated in the local machine. The Workbench allowed us to design and query our database after we loaded and transformed our data from our two csv files ‘Books_data.csv’ and ‘Books_rating.csv’ in our tool.

Users will interact with the database via the recommender web application situated on the Goodreads website. This web application would use Metabase to provide book recommendations. The user inputs the title of a book they want to receive similar recommendations for and/or a specific genre and Metabase will connect to the MySQL server to run queries and retrieve data for the user. More specifically, the web browser will send an HTTP request to the web server. The Metabase server will then construct SQL queries based on this request, forward it to the MySQL server which executes them and sends the results back to Metabase. The Metabase server then processes the data and provides the information which is displayed in the web browser to the client.

4.2. Data loading

4.2.1. Initial Loading

We had to set up our system before loading the data. First, we set the “DBMS connection read timeout interval” to 0 under Preferences > SQL Editor. We also configured the connection to allow loading of local files under the Advanced tab of the Connection tab. Furthermore, we also edited the my.ini file by defining the file path as empty and setting the net_read_timeout and max_allowed_packet to high values to be on the safe side.

```
[mysqld]
secure-file-priv=""
net_read_timeout=7000000
max_allowed_packet=500M
```

Figure 5: Screenshot of [mysqld] in the my.ini file

We also specified the parameters below before loading the data:

```sql
SHOW VARIABLES LIKE 'secure_file_priv';
SET GLOBAL local_infile = true;
SHOW GLOBAL VARIABLES LIKE 'local_infile';
```

To load the information from the csv files into the tables we created (see Section 3 : Data Model & Database Schema), we specified that columns are separated by commas, lines are terminated by a space and that the headers from the files had to be ignored. We also specified that fields were enclosed by apostrophes.

During the loading, to temporarily store data from the columns, we defined user variables as temporary stores. We immediately retrieved data thereafter with the SET command, which also allowed us to run initial transformation steps.

For the Books table, we properly capitalized book titles. We also avoided keeping null values for *RatingsCount* since we set them to 0 as a default. One the next page is the code we used specifically for the *Books* table that showcases our process.

Loading data into the Books table

```
LOAD DATA LOCAL INFILE 'C:/ProgramData/MySQL/MySQL Server
8.0/Uploads/books_data.csv'

INTO TABLE Books
FIELDS TERMINATED BY ','
OPTIONALLY ENCLOSED BY '"' -- Because some titles are enclosed by "", others not
LINES TERMINATED BY '\n'
IGNORE 1 LINES
(
    @var_Title,
    @var_description,
    @var_authors,
    @var_image,
    @var_previewLink,
    @var_publisher,
    @var_publishedDate,
    @var_infoLink,
    @var_categories,
    @var_RatingsCount
)
SET
 -- Normalize movie title to capitalize first letter
    BookTitle = CONCAT(UPPER(SUBSTRING(@var_Title, 1, 1)),
LOWER(SUBSTRING(@var_Title FROM 2))),
    BookDescription = @var_description,
    Authors = @var_authors,
    ImageLink = @var_image,
    PreviewLink = @var_previewLink,
    Publisher = @var_publisher,
    PublishedDate = @var_publishedDate,
    InfoLink = @var_infoLink,
    Genres = @var_categories,
    RatingsCount = IF(@var_RatingsCount = '' OR @var_RatingsCount IS NULL, 0,
@var_RatingsCount);

CREATE INDEX idx_books_book_id ON Books(BookId); -- To make operations faster
CREATE INDEX idx_books_book_title ON Books(BookTitle); -- To make operations faster
```

For the *Ratings* table, we also properly capitalized book titles in exactly the same way as for the *Books* table. We also made sure to use the FROM_UNIXTIME function on the *ReviewTime* column to convert it to DATETIME.

Since the loading of millions of ratings into the *Ratings* table can become very slow if we have key constraints, we investigated the possibility of dropping primary and foreign keys in the *Ratings* table before loading the data. However, our trials showed very little improvement : 381 seconds instead of 385 seconds to load all reviews. Furthermore, since the additional queries to delete key constraints were taking some time to execute, we decided to not drop and then re-define key constraints.

Loading data into the Ratings table

```
LOAD DATA LOCAL INFILE 'C:/ProgramData/MySQL/MySQL Server
8.0/Uploads/Books_rating.csv'
INTO TABLE Ratings
FIELDS TERMINATED BY ','
ENCLOSED BY '"'
LINES TERMINATED BY '\n'
IGNORE 1 LINES
(
    @var_Id,
    @var_Title,
    @var_Price,
    @var_UserId,
    @var_ProfileName,
    @var_ReviewHelpfulness,
    @var_ReviewScore,
    @var_ReviewTime,
    @var_ReviewSummary,
    @var_ReviewText
)
SET
    ISBN = @var_Id,
    BookTitle = CONCAT(UPPER(SUBSTRING(@var_Title, 1, 1)),
LOWER(SUBSTRING(@var_Title FROM 2))),
    Price = @var_Price,
    UserId = @var_UserId,
    ProfileName = @var_ProfileName,
    ReviewHelpfulness = @var_ReviewHelpfulness,
    ReviewScore = @var_ReviewScore,
    ReviewTime = FROM_UNIXTIME(@var_ReviewTime),
    ReviewSummary = @var_ReviewSummary,
    ReviewText = @var_ReviewText;

CREATE INDEX idx_ratings_rating_id ON Ratings(RatingId); -- To make operations
faster
CREATE INDEX idx_ratings_book_title ON Ratings(BookTitle); -- To make operations
faster
CREATE INDEX idx_ratings_book_id ON Ratings(BookId); -- To make operations faster
```

4.2.2. Duplicate Removal

Upon inspection of the *Books* dataset, it was noted that several book titles had duplicate values, only differentiated by a case change on some letters, while keeping all the rest of the data entirely duplicated. This required the removal of the duplicated book title values using the following SQL query:

```sql
DELETE b1
FROM Books b1
LEFT JOIN (
   SELECT MIN(BookId) AS BookId
   FROM Books
   GROUP BY BookTitle
) AS b2 ON b1.BookId = b2.BookId
WHERE b2.BookId IS NULL;
```

Note that since the above table did not have a primary key, SQL Workbench refused to execute the query. As a consequence, we had to disable Safe Updates under Preferences > SQL Editor. It was reactivated afterwards:

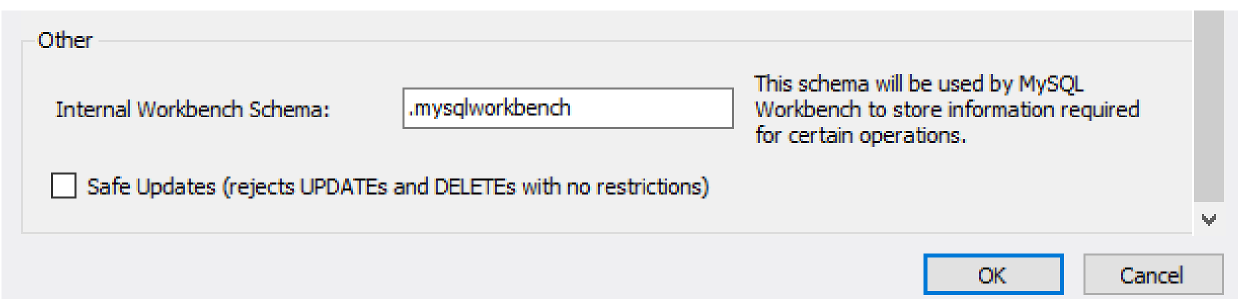

Figure 6: Screenshot of Safe Updates under Preferences > SQL Editor

4.3. Data transformations

4.3.3. Data Integrity

To make queries perform better, the *Ratings* table was assigned a foreign key called *BookId* which references the *BookId* from the *Books* table. But since on the original datasets this relationship was not assigned, we had to add this relationship with the following query, which uses a temporary join table to speed up the update process:

```sql
CREATE TEMPORARY TABLE TempRatingsUpdate AS
SELECT r.RatingId, b.BookId
FROM Ratings r
JOIN Books b ON r.BookTitle = b.BookTitle;

CREATE INDEX idx_temp_ratings_update ON TempRatingsUpdate(RatingId);

UPDATE Ratings r
JOIN TempRatingsUpdate t ON r.RatingId = t.RatingId
SET r.BookId = t.BookId;

DROP TEMPORARY TABLE TempRatingsUpdate;
```

4.3.2. Vertical Partitioning

Right after loading the data, vertical partitioning on attributes with long types of data (i.e. review texts, URLs) were split into their own dependent smaller table. This is done to improve performance, manage storage more efficiently, and optimize the use of indexes. The attributes that required vertical partition to a metadata table from each original dataset were:

- **Books dataset**: The attributes/columns *BookDescription*, *ImageLink*, *PreviewLink*, and *InfoLink* were partitioned to the new table *BooksMetadata*.
- **Ratings dataset**: The attributes/columns *ReviewSummary* and *ReviewText* were partitioned to the new table *RatingsMetadata*.

```sql
-- CREATE THE METADATA TABLE FOR BOOKS
CREATE TABLE BooksMetadata (
    BookId INT PRIMARY KEY,
    BookDescription TEXT,
    ImageLink VARCHAR(150),
    PreviewLink VARCHAR(1000),
    InfoLink VARCHAR(500),
    CONSTRAINT fk_booksmetadata_book_id FOREIGN KEY (BookId) REFERENCES
Books(BookId)
);
INSERT INTO BooksMetadata (BookId, BookDescription, ImageLink, PreviewLink,
InfoLink)
SELECT BookId, BookDescription, ImageLink, PreviewLink, InfoLink
FROM Books;
ALTER TABLE Books
DROP COLUMN BookDescription,
DROP COLUMN ImageLink,
DROP COLUMN PreviewLink,
DROP COLUMN InfoLink;

-- CREATE THE METADATA TABLE FOR RATINGS
CREATE TABLE RatingsMetadata (
    RatingId INT PRIMARY KEY,
    ReviewSummary LONGTEXT,
    ReviewText LONGTEXT,
    CONSTRAINT fk_ratingsmetadata_rating_id FOREIGN KEY (RatingId) REFERENCES
Ratings(RatingId)
);
INSERT INTO RatingsMetadata (RatingId, ReviewSummary, ReviewText)
SELECT RatingId, ReviewSummary, ReviewText
FROM Ratings;
ALTER TABLE Ratings
DROP COLUMN ReviewSummary,
DROP COLUMN ReviewText;
```

4.3.3. Adding BookId to the Ratings table

As a last step, we created a temporary table *TempRatingsUpdate* Where we added the attributes *RatingId* and *BookId*. Using an index on *RatingId* for speed, we then joined it to *Ratings* on the *RatingId*. Lastly, we dropped our temporary table and we dropped *BookTitle* from *Ratings*.

5. Analyzing & Evaluating Data

5.1. Verbal description of query

To query our database, we first created a materialized view containing every *BookId - UserId* pair in which a user rated the book a score of 4 or more out of 5. An index was immediately created as well on the two columns *BookId* and *UserId* to quickly access the different rows of this view.

Next, we set our desired genre as a temporary variable. This field can remain blank if the user only wants to receive recommendations based on the book title he defines.

The query itself consists of several blocks (parts 3 to 6) which we will describe in detail.

In part 3, we create a Common Table Expression (CTE) named *LikedByUsers*. The *UserId* of users who rated the input book 4 or more out of 5 are selected using the materialized view. **Selection** (SELECT), **joining** (JOIN) and **projection** (WHERE) are present in this part.

In part 4, a second CTE is created and named *RecommendedBooks*. It contains the *BookId*, *BookTitle*, *RatingsCount* and *GoodRatingsCount* of books that were rated 4 or more by the users selected in part 3. The input book is omitted. **Joining** (JOIN), **Aggregation** (COUNT) and **grouping** (GROUP BY) appear in this part.

In part 5, a third CTE computes the recommendations. To do so, the weighted score is calculated for each book selected in part 4. In the **case when** (CASE WHEN) the desired Genre is left blank, the score is calculated as *(number of good ratings * number of good ratings) / total number of ratings.* Otherwise, books containing the selected genre will have their score multiplied by 7 to bring them into the recommended list : (*(number of good ratings * number of good ratings) / total number of ratings)*7.* Then the top 50 books are selected and ordered by descending score.

In part 6, our selection is re-ordered and narrowed down even further through the use of a subquery. A new column called TopPicks is created where we place the row numbers of our books in descending order of weight. These row numbers are created with the function ROW_NUMBER() OVER and serve to select the top 10 books we will return to the user. Inside of the query, we select the information to be displayed to the user: *WeightedScore, BookTitle, Authors, GenreName, PublisherName, Price, ISBN, PublishedDate, ImageLink, BookDescription, GoodRatingsCount, RatingsCount*. If the book has several authors, they are all displayed separated by a comma.

5.2. Show connection between query and use case

The query is central to the use case. It allows the calculation of the key value *WeightedScore* which may be adjusted depending on if a genre is selected by the user. Following this calculation, it retrieves the top 10 recommended books which help the user decide what to read next. As the books recommended are all within the database of Goodreads, this enhances their recommendation service and allows the user to easily access the books' pages on the website, then read detailed reviews or purchase the books.

5.3. Query in database syntax

On the next pages is the full query for our book recommender system.

```sql
-- Set up for the Query
-- 1. Creating a materialized view with an index on UserID and BookID
CREATE TABLE IF NOT EXISTS Bookslikedbyusers AS SELECT BookId, UserId FROM
    ratings
WHERE
    ReviewScore >= 4;

CREATE INDEX IX_UserID_BookId ON Bookslikedbyusers(UserId, BookId);

—- 2. Setting the Genre input (Note that in Metabase this is removed):
SET @desiredGenre = ''; - User can input a genre or leave this field blank

-- Query
-- 3. Getting the users that rated the chosen book 4 or more
WITH
  LikedByUsers AS (
    SELECT DISTINCT
      blu.UserId
    FROM
      Bookslikedbyusers blu
      JOIN books b ON blu.BookId = b.BookId
    WHERE
      b.BookTitle = 'Carrie'- User inputs the book that they like
  ),

-- 4. Getting the other books that the preselected users rated 4 or more, while counting the total number of ratings and the number of good ratings for each book
 RecommendedBooks AS (
    SELECT
      b.BookId,
      b.BookTitle,
      bhg.GenreId,
      g.GenreName,
      COUNT(r.RatingId) AS RatingsCount,
      COUNT(DISTINCT r.UserId) AS GoodRatingsCount
    FROM
      ratings r
      JOIN books b ON r.BookId = b.BookId
      JOIN bookhasgenre bhg ON b.BookId = bhg.BookId
      JOIN genre g ON bhg.GenreId = g.GenreId
    WHERE
```

```
      r.UserId IN (
        SELECT
          UserId
        FROM
          LikedByUsers
      )
      AND b.BookTitle <> 'Carrie'
    GROUP BY
      b.BookId,
      b.BookTitle,
      bhg.GenreId,
      g.GenreName
  ),

-- 5. Getting the recommended books by calculating a weighted score for each book,
ordering the results and limiting to only the top 50
 TopRecommendedBooks AS (
    SELECT
      rb.BookId,
      rb.BookTitle,
      rb.RatingsCount,
      rb.GoodRatingsCount,
      rb.GenreName,
      CASE
       WHEN @desiredGenre IS NULL OR @desiredGenre='' THEN (rb.GoodRatingsCount *
rb.GoodRatingsCount) / rb.RatingsCount
   WHEN rb.GenreName = @desiredGenre THEN ((rb.GoodRatingsCount *
rb.GoodRatingsCount)/ rb.RatingsCount)*7
       ELSE (rb.GoodRatingsCount * rb.GoodRatingsCount) / rb.RatingsCount
      END AS WeightedScore
    FROM
      RecommendedBooks rb
    ORDER BY
      WeightedScore DESC
    LIMIT
      50)
-- 6. Select the top 10 books and the data to be displayed
SELECT *
FROM
(
SELECT
```

```
ROW_NUMBER() OVER (ORDER BY trb.WeightedScore DESC) AS TopPicks,
  trb.WeightedScore,
  b.BookTitle,
  GROUP_CONCAT(a.AuthorName SEPARATOR ', ') AS Authors,
  rb.GenreName,
  p.PublisherName,
  b.Price,
  b.ISBN,
  b.PublishedDate,
  bm.ImageLink,
  bm.BookDescription,
  rb.GoodRatingsCount,
  rb.RatingsCount
FROM
  TopRecommendedBooks trb
  JOIN Books b ON trb.BookId = b.BookId
  JOIN BookHasAuthor bha ON b.BookId = bha.BookId
  JOIN Author a ON bha.AuthorId = a.AuthorId
  JOIN Publishers p ON b.PublisherId = p.PublisherId
  JOIN BooksMetadata bm ON b.BookId = bm.BookId
  JOIN RecommendedBooks rb ON b.BookId = rb.BookId
GROUP BY
  trb.WeightedScore,
  b.BookId,
  b.BookTitle,
  rb.GenreName,
  p.PublisherName,
  b.Price,
  b.ISBN,
  b.PublishedDate,
  bm.ImageLink,
  bm.BookDescription,
  rb.GoodRatingsCount,
  rb.RatingsCount) subquery
  ORDER BY TopPicks
  LIMIT 10;
```

6. Efficiency & Query Performance

6.1. Analysis of runtime bottlenecks of your queries

The recommender query involves several operations, including multiple joins, subqueries and aggregations. These potential bottlenecks were considered in writing the query:

- **Full Table Scans:** Avoid queries that do not use indexes on full table scans.
- **Multiple Joins:** Joining several large tables like *Ratings*, *Books*, *BookHasGenre*, and *Genre* can cause performance overhead without proper relationships and indexes.
- **Aggregation:** The `GROUP BY` clause and aggregation functions like `COUNT()` and `SUM()` also benefit from the indexes.
- **Filters and Sorting:** Filters like the `WHERE` clause and sorting like the `ORDER` can be very slow without the use of indexes. SARGable queries are also used when possible, especially in the main database query, to take advantage of those indexes.

6.2. Optimization measures used

The following optimization measures and techniques were implemented in the query detailed on point 5.1. Query in database syntax.

6.2.1. Materialized Views and Execution plan

The table *Bookslikedbyusers* is created as a **materialized view** to store results of the subquery that identifies books rated 4 or more by users. This will avoid repeated computation and speed the main query:

```
CREATE TABLE IF NOT EXISTS Bookslikedbyusers AS SELECT BookId, UserId FROM ratings
WHERE ReviewScore >= 4;
```

As a consequence, the execution plan results in a full table scan:

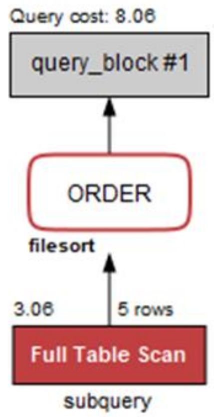

Figure 7 : Execution Plan of the query

It also appears at a first glance that the execution times are longer compared to a regular view, since the cost is greater (the amount of rows is more important).

Here is the regular view that was defined for this test:

```
CREATE OR REPLACE VIEW BooksRatings4 AS
SELECT BookId, UserId
FROM ratings r
WHERE r.ReviewScore >=4;
```

Here are the Result Grids:

id	select_type	table	par	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	PRIMARY	<derived2>	NULL	ALL	NULL	NULL	NULL	NULL	5	100.00	Using filesort
2	DERIVED	<derived4>	NULL	ALL	NULL	NULL	NULL	NULL	2	100.00	Using temporary; Using filesort
2	DERIVED	bm	NULL	eq_ref	PRIMARY	PRIMARY	4	rb.BookId	1	100.00	NULL
2	DERIVED	b	NULL	eq_ref	PRIMARY,idx_books_book_id,fk_books_publisher_id	PRIMARY	4	rb.BookId	1	100.00	Using where
2	DERIVED	p	NULL	eq_ref	PRIMARY	PRIMARY	4	bookworms.b.PublisherId	1	100.00	NULL
2	DERIVED	bha	NULL	ref	PRIMARY,AuthorId	PRIMARY	4	rb.BookId	1	100.00	Using index
2	DERIVED	a	NULL	eq_ref	PRIMARY	PRIMARY	4	bookworms.bha.AuthorId	1	100.00	NULL
2	DERIVED	<derived3>	NULL	ref	<auto_key0>	<auto_key0>	4	rb.BookId	2	100.00	NULL
3	DERIVED	<derived4>	NULL	ALL	NULL	NULL	NULL	NULL	2	100.00	Using filesort
4	DERIVED	<subquery...	NULL	ALL	NULL	NULL	NULL	NULL	NULL	100.00	Using where; Using temporary; Using filesort
4	DERIVED	r	NULL	ref	idx_ratings_book_id,fk_ratings_user_id	fk_ratings_user_id	5	<subquery5>.UserId	2	100.00	Using where
4	DERIVED	bhg	NULL	ref	PRIMARY,GenreId	PRIMARY	4	bookworms.r.BookId	1	100.00	Using index
4	DERIVED	g	NULL	eq_ref	PRIMARY	PRIMARY	4	bookworms.bhg.GenreId	1	100.00	NULL
4	DERIVED	b	NULL	eq_ref	PRIMARY,idx_books_book_id,idx_books_book_title	PRIMARY	4	bookworms.r.BookId	1	77.35	Using where
5	MATERIALIZED	<derived6>	NULL	ALL	NULL	NULL	NULL	NULL	4	100.00	NULL
6	DERIVED	b	NULL	ref	PRIMARY,idx_books_book_id,idx_books_book_title	idx_books_book_title	2003	const	1	100.00	Using index; Using temporary
6	DERIVED	r	NULL	ref	idx_ratings_book_id,fk_ratings_user_id	idx_ratings_book_id	5	bookworms.b.BookId	13	33.33	Using where

Figure 8 : Result Grid of the query using regular views obtained by using the EXPLAIN keyword[7]

[7] See SQL script called *Reco_Regular_Materialized_Views_Comparison_with_genres* for the full code

id	select_type	table	par	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	PRIMARY	<derived2>	NULL	ALL	NULL	NULL	NULL	NULL	5	100.00	Using filesort
2	DERIVED	<derived4>	NULL	ALL	NULL	NULL	NULL	NULL	2	100.00	Using temporary; Using filesort
2	DERIVED	bm	NULL	eq_ref	PRIMARY	PRIMARY	4	rb.BookId	1	100.00	NULL
2	DERIVED	b	NULL	eq_ref	PRIMARY,idx_books_book_id,fk_books_publisher_id	PRIMARY	4	rb.BookId	1	100.00	Using where
2	DERIVED	p	NULL	eq_ref	PRIMARY	PRIMARY	4	bookworms.b.PublisherId	1	100.00	NULL
2	DERIVED	bha	NULL	ref	PRIMARY,AuthorId	PRIMARY	4	rb.BookId	1	100.00	Using index
2	DERIVED	a	NULL	eq_ref	PRIMARY	PRIMARY	4	bookworms.bha.AuthorId	1	100.00	NULL
2	DERIVED	<derived3>	NULL	ref	<auto_key0>	<auto_key0>	4	rb.BookId	2	100.00	NULL
3	DERIVED	<derived4>	NULL	ALL	NULL	NULL	NULL	NULL	2	100.00	Using filesort
4	DERIVED	<subquery…	NULL	ALL	NULL	NULL	NULL	NULL	NULL	100.00	Using where; Using temporary; Using filesort
4	DERIVED	r	NULL	ref	idx_ratings_book_id,fk_ratings_user_id	fk_ratings_user_id	5	<subquery5>.UserId	2	100.00	Using where
4	DERIVED	bhg	NULL	ref	PRIMARY,GenreId	PRIMARY	4	bookworms.r.BookId	1	100.00	Using index
4	DERIVED	b	NULL	eq_ref	PRIMARY,idx_books_book_id,idx_books_book_title	PRIMARY	4	bookworms.r.BookId	1	77.35	Using where
4	DERIVED	g	NULL	ALL	PRIMARY	NULL	NULL	NULL	10902	0.01	Using where; Using join buffer (hash join)
5	MATERIALIZED	<derived6>	NULL	ALL	NULL	NULL	NULL	NULL	238821	100.00	NULL
6	DERIVED	b	NULL	ref	PRIMARY,idx_books_book_id,idx_books_book_title	idx_books_book_title	2003	const	1	100.00	Using index; Using temporary
6	DERIVED	blu	NULL	index	IX_UserID_BookId	IX_UserID_BookId	10	NULL	2388216	10.00	Using where; Using index; Using join buffer …

Figure 9 : Result Grid of the query using materialized views obtained by using the EXPLAIN keyword

However, looking closer at the Query Stats, we notice that on the server side the materialized view takes less time which might mean that information is accessed more efficiently:

Query Statistics

Timing (as measured at client side):
Execution time: 0:00:0.00000000

Timing (as measured by the server):
Execution time: 0:00:0.00277310
Table lock wait time: 0:00:0.00000600

Errors:
Had Errors: NO
Warnings: 2

Rows Processed:
Rows affected: 0
Rows sent to client: 17
Rows examined: 0

Temporary Tables:
Temporary disk tables created: 0
Temporary tables created: 7

Joins per Type:
Full table scans (Select_scan): 0
Joins using table scans (Select_full_join): 0
Joins using range search (Select_full_range_join): 0
Joins with range checks (Select_range_check): 0
Joins using range (Select_range): 0

Sorting:
Sorted rows (Sort_rows): 0
Sort merge passes (Sort_merge_passes): 0
Sorts with ranges (Sort_range): 0
Sorts with table scans (Sort_scan): 0

Index Usage:
No Index used

Other Info:
Event Id: 57
Thread Id: 312

Figure 10 : Query Stats of the regular view obtained using the EXPLAIN keyword

Query Statistics

Timing (as measured at client side):
Execution time: 0:00:0.01600000

Timing (as measured by the server):
Execution time: 0:00:0.00160920
Table lock wait time: 0:00:0.00000600

Errors:
Had Errors: NO
Warnings: 2

Rows Processed:
Rows affected: 0
Rows sent to client: 17
Rows examined: 0

Temporary Tables:
Temporary disk tables created: 0
Temporary tables created: 7

Joins per Type:
Full table scans (Select_scan): 0
Joins using table scans (Select_full_join): 0
Joins using range search (Select_full_range_join): 0
Joins with range checks (Select_range_check): 0
Joins using range (Select_range): 0

Sorting:
Sorted rows (Sort_rows): 0
Sort merge passes (Sort_merge_passes): 0
Sorts with ranges (Sort_range): 0
Sorts with table scans (Sort_scan): 0

Index Usage:
No Index used

Other Info:
Event Id: 63
Thread Id: 312

Figure 11 : Query Stats of the materialized view obtained using the EXPLAIN keyword

6.5.2. WHERE clauses

In our query to return book recommendation, the first CTE that we defined used a non-sargable WHERE clause. This was done so that the results for exactly the book title that was entered would be returned. We did not want to include users who rated books whose title contained “Carrie”, for example. Afterwards, we made sure to use ‘<>’ for ‘not equal to’ and abided to sargable practices by utilizing indexed columns and efficient joins and aggregations as described below.

6.5.3. Indexes

Indexes are created on key columns to speed up the data retrieval. In this case an index is created on the same *Bookslikedbyusers* table to facilitate the quick lookups based on *UserId* and *BookId.* Looking at the Result Grid for the Materialized View above, we can clearly see that it is utilized by our query and that about 10% of the information is accessed to find the needed data.

6.5.4. Efficient Joins and Aggregations

The use of indexed columns in the join conditions and groupings improves the efficiency of the operation. The query uses joins with the *Books*, *BookHasGenre*, and *Genre* tables. This was achieved through a series of `CREATE INDEX` DDL commands performed during the entire DDL and Data Transformation steps of the database project. With the inclusion of these indexes, the need to perform full table scans is reduced. Additionally, aggregations like `COUNT()` and `SUM()` also benefit from the indexing.

6.5.5. Temporary Tables

Along the whole process, temporary tables were also used to reduce the need to compute results multiple times. In the query, *Bookslikedbyusers* is created to store the books that users rated 4 or more.

6.3. Correlation between measures and runtime optimization

Indexing: By creating indexes on columns such as *BookId*, *UserId,* and *GenreId*, the query execution became much faster. Indexes allowed the database engine to quickly locate the necessary rows without performing full table scans. This drastically reduced the number of rows processed and improved the speed of join and filtering operations. Specifically, indexing alone lowered the query execution time from 3540 seconds (almost one hour) to 312 seconds.

Materialized Views: The use of the materialized view *Bookslikedbyusers* to store precomputed results of highly-rated books significantly reduced the computation load. Instead of recalculating these results for each query execution, the materialized view provided immediate access to the required data, cutting down the execution time from 310 seconds to 4.5 seconds.

7. Visualization & Decision Support

7.1. Visualization of query results

Below are the list output and the graph output of a query in Metabase. The input book is "Carrie", a horror novel by successful writer Stephen King and the genre has been left empty. The recommendations include only books by the same author, and the BookDescription field hints that the genres are also horror, fiction and mystery, indicating that they are of high relevance.

Recommendations 6 and 7 are the same book but with different title spelling. Some duplicates were treated during the loading phase, but a more thorough and detailed cleaning of the book data would be necessary to prevent these kinds of issues and improve the quality of the recommendations. However, due to the ELT nature of this project as well as time restrictions, this aspect of data quality was not investigated further.

This question is written in SQL. Enter the title of the book you like *: Carrie Enter the genre of books you prefer *: NULL

TopPicks	WeightedScore	BookTitle	Authors	GenreName	PublisherName
1	9.91	Firestarter (signet book ser.)	Stephen King	Fiction	Berkley
2	9	The colorado kid	Stephen King	Fiction	Titan Books (US, CA)
3	6.21	Penguin readers level 6: misery (penguin longman reader level 6)	Robin Waterfield, Stephen King	Readers	Longman
4	6.11	The drawing of the 3: the dark tower ii	Stephen King	Fiction	Donald m Grant Pub Incorporated
5	5.5	The dead zone	Stephen King	Fiction	Simon and Schuster
6	5.44	Running man	Stephen King	Fiction	Simon and Schuster
7	5.44	The running man	Stephen King	Fiction	Simon and Schuster
8	5.32	Needful things	Stephen King	Fiction	Simon and Schuster
9	5.19	Night shift (signet)	Stephen King, Stephen King	American	Signet Book
10	5.19	Night shift (signet)	Stephen King, Stephen King	Horror tales	Signet Book

Figure 12: Results of the query (part 1)

Price	ISBN	PublishedDate	ImageLink
	B000OUGBYA	1981	http://books.google.com/books/content?id=dFaLU4a3B28C&printsec=frontcover&img=1&zoom=1&source=gbs_api
	0743550404	2019-05-07	http://books.google.com/books/content?id=DPIOEAAAQBAJ&printsec=frontcover&img=1&zoom=1&source=gbs_api
	0582418291	2008	http://books.google.com/books/content?id=I_BLPwAACAAJ&printsec=frontcover&img=1&zoom=1&source=gbs_api
	B000HM6M30	1987	http://books.google.com/books/content?id=x2SDPwAACAAJ&printsec=frontcover&img=1&zoom=1&source=gbs_api
	B000NPQ9P2	2016-04-12	http://books.google.com/books/content?id=O53jCwAAQBAJ&printsec=frontcover&img=1&zoom=1&edge=curl&source=gbs_api
	0450056422	2016-04-19	http://books.google.com/books/content?id=JX_0CwAAQBAJ&printsec=frontcover&img=1&zoom=1&edge=curl&source=gbs_api
	B0006Y0EN8	2016-04-19	http://books.google.com/books/content?id=JX_0CwAAQBAJ&printsec=frontcover&img=1&zoom=1&edge=curl&source=gbs_api
	B000CC0HE0	2016-05-03	http://books.google.com/books/content?id=jAUODAAAQBAJ&printsec=frontcover&img=1&zoom=1&edge=curl&source=gbs_api
	0451170113	1979	http://books.google.com/books/content?id=p9TRiBVo7TcC&printsec=frontcover&img=1&zoom=1&source=gbs_api
	0451170113	1979	http://books.google.com/books/content?id=p9TRiBVo7TcC&printsec=frontcover&img=1&zoom=1&source=gbs_api

Figure 13 : Results of the query (part 2)

BookDescription	GoodRatingsCount	RatingsCount
Andy and Vicky McGee's eight-year-old daughter, Charlie, has the ability to set things on fire and a secret government agency is de…	31	97
Stephen King's bestselling unsolved mystery, THE COLORADO KID -- inspiration for the TV series HAVEN -- returns to bookstores …	9	9
Contemporary / British English A story by Stephen King -- the master of horror. Paul Sheldon is Annie Wilkes's favourite writer. She…	32	165
After his confrontation with the man in black at the end of The Gunslinger , Roland awakes to find three doors on the beach of Mid-…	18	53
A man awakens from a 5-year coma to discover he has powers to see visions of the past, present and future, a power which drives h…	22	88
A desperate man attempts to win a reality TV game where the only objective is to stay alive in this #1 national bestseller from Step…	14	36
A desperate man attempts to win a reality TV game where the only objective is to stay alive in this #1 national bestseller from Step…	14	36
Now available for the first time in a mass-market premium paperback edition—master storyteller Stephen King presents the classi…	22	91
A chilling collection of twenty horror stories.	20	77
A chilling collection of twenty horror stories.	20	77

Figure 14: Results of the query (part 3)

The rank calculated in the query and the weighted score are displayed first, followed by all information that may be needed by the user to find the book. That includes the title, author, ISBN, publisher and release date, the price and the cover image. In addition, the book description plays an important role in helping the user decide if it appeals to them. Finally, the number of good ratings and the number of total ratings are displayed for transparency.

If the user selects a genre they would prefer to be recommended to them, for example "History", based on preferences of the same users who liked the book they indicated, then the recommendations change and History books appear in the top 10.

This question is written in SQL.

Enter the title of the book you like *: Carrie

Enter the genre of books you prefer *: History

TopPicks	WeightedScore	BookTitle	Authors	GenreName	PublisherName
1	21	Mayflower : a story of courage, community, and war	Nathaniel Philbrick	History	Penguin
2	14	The w effect: bush's war on women	Laura Flanders	History	Feminist Press at CUNY
3	9.91	Firestarter (signet book ser.)	Stephen King	Fiction	Berkley
4	9.33	As jesus cared for women: restoring women then and now	Christina Page	History	Basic Books
5	9.33	Lest innocent blood be shed	Philip P. Hallie	History	Harper Collins
6	9.33	Blowout (library edition)	Rachel Maddow	History	Crown Publishing Group (NY)
7	9.33	Tao te ching (chinese classics (hong kong).)	老子, Dim Cheuk Lau	History	
8	9.33	The soldier and the state: the theory and politics of civil-military relations	Samuel Phillips Huntington, Samuel P. Huntington	History	Harvard University Press
9	9.33	The hoax: library edition	Various	History	Routledge
10	9	The colorado kid	Stephen King	Fiction	Titan Books (US, CA)

Figure 15 : Results of the query with “History” selected as a preferred genre

The weighted score changed since the scores of books with the “History” genre were multiplied by 7 to make them appear in the top 10 ranking. Two of the books are still novels by Stephen King, which means that we still have familiar and highly relevant recommendations at the top while introducing some new suggestions with an unrelated genre which were liked by the same readers who liked Stephen King books.

The graph visualization on the next page puts two figures into perspective: the weighted score and the total number of ratings. The weighted score is the main figure to be considered when looking for a recommendation. However, some readers might be interested in comparing the total number of ratings, which gives an indication about the overall popularity of the book. For example, the visualization shows that the third book, “Firestarter” by Stephen King, has a much higher number of ratings than the other books with the History genre, suggesting that it might be the most representative of the user’s preferences.

Figure 16: Visualization of the results for the book “Carrie” and genre “History”

7.2. Decision recommendations for the persona

Lisa takes note of the other books by Stephen King that she didn’t know, and it makes sense to her that they would have more reviews than the other books in the list since they are so popular. But she mainly turns her attention to the first book, “Mayflower: A story of courage, community and war” by Nathaniel Philbrick. She has always wanted to learn more about early colonial America and had never heard of this author nor his book. This is precisely what she wanted to find through the recommendations : a book about History which was rated highly by people with similar tastes as hers and with a decent amount of good ratings. Furthermore, the summary of the book makes her want to read it. So she will start with this one and if she likes it, hopefully she will use the recommender system again with “Mayflower” and keep discovering new books to read.

7.3. Connection between visualization and original use case

Both the list and the graph should be presented to the user as they include insights for decision support. The first allows a clear representation of the scores as well as useful book information, and the second is more adapted for comparison between books. As demonstrated by the persona, not only the better rated book can be interesting, but also how popular it is, as measured by the amount of reviews. It is much more pleasant for users to view a graph where they get an overview of all the books suggested, which they can compare quickly and easily.

8. Conclusions & Lessons Learned

Throughout this project, we have managed to create a functioning book recommender system that not only allows the user to input a book that they like and obtain relevant suggestions, but also returns a more diversified book selection through the input of a preferred genre. Additionally, with a visualization of each of the top 10 books plotted against their *WeightedScore* and *GoodRatingsCount*, the user can analyze how popular the books really are and quickly weigh options as to which book to select. If the user needs more details, he can switch back to the table of results and consult descriptions, prices, images and other information.

In terms of the work that went into constructing our book recommender system using MySQL in the SQL Workbench, one of the most important lessons that we learned was the importance of futureproofing our database system. Indeed, it might not be used the same way tomorrow as it is today. As a consequence, we made sure to keep all the books in our database, even those with no reviews, so that future reviews may be linked to pre-existing books in the *Books* table. Likewise, information may need to be added to or deleted from tables, such as a newly published book. This put in perspective the necessity of normalizing our tables to prevent mistakes and excessive data entry work. Another good practice we retained was definitely the use of indexes to speed up data retrieval as execution times were drastically shortened by more than 10 times as a result.

As a next step, we could standardize the genres of the books in our database and include sub-genres as an additional attribute to the *Genre* table. Finally, we could explore the actual implementation of the book recommender system on a website such as Goodreads to fully test its performance.