Live Poets Society

Group 10

Mona Gandhi Nimay Kumar Rohan Saraogi Tom Huang

{mona09, nimay512, rsaraogi}@seas.upenn.edu zthuang@sas.upenn.edu

1. Introduction

In this project we have created a social cataloging application specifically for poetry lovers. Our aim was to make a platform that enables users to explore the world of poetry through an extensive collection of books, series, authors, and reviews.

The application is designed to offer a personalized experience to each user. By signing in, users can create and maintain their own virtual library of poetry books, rate them, and receive custom recommendations based on their past behaviour about new poetry books that they might enjoy.

In addition to these features, the application also seeks to encourage community engagement. This is achieved by providing information about author perceptions in the community, and highlighting the most active members of the community, which allows users to learn more about other members.

Overall, we hope the application can serve as a valuable resource for poetry enthusiasts of all levels, whether they are just starting out or are seasoned experts.

2. Architecture

The application consists of a backend MySQL DB Instance hosted on Amazon Relational Database Service (Amazon RDS), and a React.js frontend.

In particular, the following is the list of technologies used and their purposes:

1. **Python**: Data preprocessing
2. **Amazon RDS MySQL DB Instance**: Database
3. **DataGrip**: Writing/optimizing SQL queries
4. **Node.js**: Runtime environment for JavaScript code
5. **Express.js**: Managing routes and communication between the frontend and the database via RESTful APIs
6. **React.js/Material-UI**: Frontend design
7. **GitHub**: Version control and collaboration

3. Data

The data is from the Goodreads datasets scraped by UCSD. We have used the subset of the datasets specific to the poetry genre, and summarize each of datasets below.

3.1. Poetry Books

Description: The dataset contains data for poetry books, such as the book ID, title, description, authors, series etc.
Summary Statistics: The dataset contains 36,514 rows, 29 columns, and has a memory usage of ∼102 MB.
Usage: The dataset serves as a primary link between the datasets as it contains references to the book series and authors. We have used the dataset to reduce the series and author datasets to the subsets specific to the poetry genre, and to create the *Book* entities, and the *Similar_Books*, *In_Series*, and *Written_By* relationships.

3.2. Series

Description: The dataset contains data for all book series (i.e. not only for the poetry genre), such as the series ID, title, description etc.
Summary Statistics: The dataset contains 400,390 rows, 7 columns, and has a memory usage of ∼156 MB.
Usage: We have used the subset of the dataset specific to the poetry genre to create the *Series* entities, and the *In_Series* and *Series_By* relationships.

3.3. Poetry Interactions

Description: The dataset contains data for user interactions with poetry books, such as the user ID, book

ID, read at date, started at date etc.
Summary Statistics: The dataset contains 2,734,350 rows, 10 columns, and has a memory usage of ∼1543 MB.
Usage: We have used the dataset to create the *User* entities, and the *In_Library* relationships.

3.4. Poetry Reviews

Description: The dataset contains data for poetry book reviews, such as the review ID, user ID of the reviewer, book ID of the book reviewed, review text, rating, no. of votes, no. of comments etc.
Summary Statistics: The dataset contains 154,555 rows, 11 columns, and has a memory usage of ∼176 MB.
Usage: We have used the dataset to create the *Review* entities, and the *Review_By* and *Review_For* relationships.

3.5. Authors

Description: The dataset contains data for all authors (i.e. not only for the poetry genre), such as the author ID, name etc.
Summary Statistics: The dataset contains 829,529 rows, 5 columns, and has a memory usage of ∼86 MB.
Usage: We have used the subset of the dataset specific to the poetry genre to create the *Author* entities, and the *Written_By* and *Series_By* relationships.

4. Database

4.1. Preprocessing

The preprocessing code is contained in the notebook *preprocessing.ipynb* on GitHub. In no particular order, the following is a list of some of the key steps in our preprocessing:

- The raw data is in JSON format, containing embedded documents and lists. To store the data in a 1NF relational schema we have blown up the documents and lists and split them into separate relations.
- The books data contains URLs for book cover images, but ∼43% of the links are broken. We believe that including the book cover images is essential to the application's aesthetics, and hence have written a scraper to scrape missing book cover images from Goodreads. This has been possible as the books data also contains a column that links to the Goodreads page of the book. Our scraper retrieves the Goodreads page HTMLs and parses the book cover image links using XPath. After scraping, only two of the links are broken.
- We have subsetted the series and author data to only include entries specific to the poetry genre.
- To avoid redundancy, we have removed duplicate columns and columns with descriptive statistics (which are derivable from the remaining data). Regarding the latter, instead of storing the statistics, we have written our own queries to display the statistics as needed.
- Much of the data is in string format. Depending on the column, we have handled quotes/whitespaces in the strings, converted strings containing numbers to the numeric format/datetimes to the datetime format, and replaced empty strings with null values.
- We have removed illogical values eg. a negative no. of votes or comments for reviews.
- For users, the data only contains anonymized user IDs. We have used the Python *Faker* library to create dummy usernames, passwords, names, and emails for the users.
- We have ensured that foreign key constraints are met prior to ingestion into the database by manually comparing the dataframe columns.

4.2. ER Diagram

The following figure shows the ER diagram.

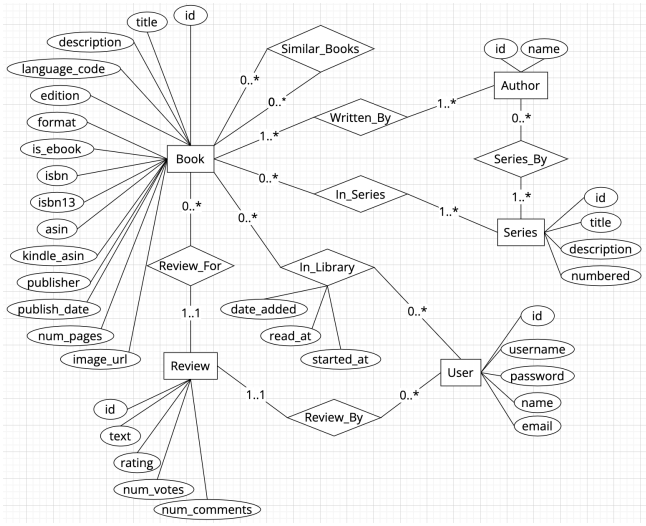

Figure 1. ER Diagram

4.3. Relational Schema

The DDL code is contained in the file *ddl.sql* on GitHub.

The following is the relational schema:

Book *(id, title, description, language_code, edition, format, is_ebook, isbn, isbn13, asin, kindle_asin, publisher, publish_date, num_pages, image_url)*
title NOT NULL

Similar_Books *(book_id1, book_id2)*
book_id1 FOREIGN KEY REFERENCES Book (id)
book_id2 FOREIGN KEY REFERENCES Book (id)

Series *(id, title, description, numbered)*
title NOT NULL

In_Series *(book_id, series_id)*
book_id FOREIGN KEY REFERENCES Book (id)
series_id FOREIGN KEY REFERENCES Series (id)

Author *(id, name)*
name NOT NULL

Written_By *(book_id, author_id)*
book_id FOREIGN KEY REFERENCES Book (id)
author_id FOREIGN KEY REFERENCES Author (id)

Series_By *(series_id, author_id)*
series_id FOREIGN KEY REFERENCES Series (id)
author_id FOREIGN KEY REFERENCES Author (id)

User *(id, username, password, name, email)*
username NOT NULL UNIQUE
password NOT NULL
name NOT NULL

In_Library *(user_id, book_id, date_added, read_at, started_at)*
user_id FOREIGN KEY REFERENCES User (id)
book_id FOREIGN KEY REFERENCES Book (id)

Review *(id, user_id, book_id, text, rating, num_votes, num_comments)*
user_id FOREIGN KEY REFERENCES User (id)
book_id FOREIGN KEY REFERENCES Book (id)
(user_id, book_id) NOT NULL UNIQUE
num_votes NOT NULL
num_comments NOT NULL

4.4. Relation Sizes

The following table shows the relation names and sizes in the database.

Relation	Cardinality
Book	36,512
Similar_Books	271,478
Series	613
In_Series	1,469
Author	23,104
Written_By	57,972
Series_By	2,094
User	377,799
In_Library	2,734,332
Review	154,552

Table 1. Relation Sizes

4.5. Normal Form Usage & Justification

The database is in BCNF. This is because each functional dependency is either trivial or the LHS of the functional dependency is a superkey. In particular,

1. All attribute domains are simple (integers, strings, booleans, datetimes) and none of them have relations as elements. Hence the database is in 1NF.

2. Each of the relations *Similar_Books*, *In_Series*, *Written_By*, and *Series_By* have two attributes, and the attributes together are the primary key. Hence, denoting the two attributes as X and Y, the only functional dependencies are trivial dependencies i.e. {X, Y} $\rightarrow$ X, {X, Y} $\rightarrow$ Y, and {X, Y} $\rightarrow$ {X, Y}.

3. Each of the *Book*, *Series*, *Author*, and *In_Library*, relations has one primary key and no alternate keys, and the attributes are only dependent on the primary key. Hence the respective minimal covers of the set of functional dependencies only contains dependencies of the form X $\rightarrow$ Y, where X is the primary key and Y is any other attribute.

4. The *User* relation has a primary key *id* and an alternate key *username*, and the attributes are only dependent on the candidate keys. Likewise, the *Review* relation has a primary key *id* and an alternate key (*user_id*, *book_id*), and the attributes are only dependent on the candidate keys. Hence the minimal cover of the set of functional dependencies in each case only contains dependencies of the form X $\rightarrow$ Y, where X is a candidate key, and Y is any other attribute.

5. Web Application Description

The application contains pages with different types of functionality, such as login/register pages, a user home page, pages to search by books/series/authors,

pages for book reviews, and a trending page with more information about the community. We summarize the functionality of the application pages below.

Login Page: The page allows existing users to login with their usernames and passwords, and also links to the register page for new users to sign up.

Register Page: The page allows new users to sign up with their usernames (required), passwords (required), names (required), and emails (optional). Users can navigate back to the login page from here, and successfully logging in will do that as well.

Home Page: The page shows users information about their accounts. Every time the page is visited or refreshed, a randomly selected "book of the moment" will be generated. The page contains a table of all the books in the user's library, and a list of 16 recommended books. The page also contains general information about the user, such as the number of books in their library, their average rating etc. Lastly, there is a logout button and logging out redirects the user to the login page.

Books Page: The books contains information about books eg. title, format, publisher, and publish date. Users can search by book titles and results are limited to 2000 entries to avoid network overload. Clicking on a book title will display a popup that contains an image of the book, the author(s), no. of pages, no. of reviews, average rating, as well as a link to a page that contains all reviews for that book. In addition, the popup will allow users to add/remove that book from their library, and if the book is in their library, users can rate it and save that rating.

Authors Page: The authors page contains information about all authors eg. name, number of works, average rating etc. Users can search for authors by their name.

Series Page: The series page contains information about all series. Users can search for series by their title. Also, when series is clicked, a popup shows all the books in that series.

Trending Page: The trending page contains information about the top 15 most active (i.e. with the most reviews) users. It also contains information author perceptions in the community based on the no. of users who dislike, are neutral towards, and like the authors. Users can search for authors by their name.

Review Page: The review page is accessible via each book information popup. It contains all the reviews and ratings available for the selected book.

Empty Page: This is the page that will be shown to users who access the site but are not logged in. It will provide a link for users to login or sign up.

6. API Specification

We summarize the routes below (to save space, we only mention route/query parameters if they are used).

Route: '/user_library/:user_id' (GET)
Description: Returns a user's library information
Route Parameters: user_id (str)
Response Type: JSON Array
Response Parameters: [{id (int), book_title (str), date_added (datetime)}, ...]

Route: '/user_recommendations/:user_id' (GET)
Description: Returns 16 book recommendations a given user hasn't read
Route Parameters: user_id (str)
Response Type: JSON Array
Response Parameters: [{book_id (int), book_title (str), image_url (str)}, ...]

Route: '/user_information/:user_id' (GET)
Description: Returns information about a user's reading habits
Route Parameters: user_id (str)
Response Type: JSON Array (of length 1)
Response Parameters: [{id (str), user_name (str), num_library (int), num_ratings (int), avg_rating (float), num_reviews (int), num_votes (int), num_comments (int)}]

Route: '/top_reviewers' (GET)
Description: Returns information about the top 15 users with the most reviews
Response Type: JSON Array
Response Parameters: [{id (str), user_name (str), num_library (int), num_ratings (int), avg_rating (float), num_reviews (int), num_votes (int), num_comments (int)}, ...]

Route: '/series_information/:series_id' (GET)
Description: Returns a series' information
Route Parameters: series_id (int)
Response Type: JSON Array (of length 1)
Response Parameters: [{title (str), description (str), numbered (bool), num_books (int)}]

Route: '/series_books/:series_id' (GET)
Description: Returns a series' books
Route Parameters: series_id (int)
Response Type: JSON Array
Response Parameters: [{id (int), title (str)}, ...]

Route: '/search_series' (GET)
Description: Returns series filtered by titles
Query Parameters: title (str) (default: '%')
Response Type: JSON Array
Response Parameters: [{id (int), title (str)}, ...]

Route: '/book_information/:book_id' (GET)
Description: Returns a book's information
Route Parameters: book_id (int)
Response Type: JSON Array (of length 1)
Response Parameters: [{title (str), author_name (str), description (str), format (str), publisher (str), publish_date (date), num_pages (int), num_reviews (int), avg_rating (float), is_ebook (int), isbn (str), isbn13 (str), asin (str), kindle_asin (str), image_url (str)}]

Route: '/random_book' (GET)
Description: Returns a random book of the moment
Response Type: JSON Array (of length 1)
Response Parameters: [{id (int)}]

Route: '/book_reviews/:book_id' (GET)
Description: Returns a book's review texts and ratings
Route Parameters: book_id (int)
Response Type: JSON Array
Response Parameters: [{text (str), rating (int)}, ...]

Route: '/book_authors/:book_id' (GET)
Description: Returns a book's authors
Route Parameters: book_id (int)
Response Type: JSON Array
Response Parameters: [{id (int), name (str)}, ...]

Route: '/search_books' (GET)
Description: Returns books filtered by titles
Query Parameters: title (str) (default: '%')
Response Type: JSON Array
Response Parameters: [{id (int), title (str), description (str), format (str), publisher (str), publish_date (date)}, ...]

Route: 'author_LND_statistics' (GET)
Description: Returns user perceptions (like/neutral/dislike) of authors based on their ratings
Response Type: JSON Array
Response Parameters: [{id (int), name (str), num_dislikes (int), num_neutral (int), num_likes (int)}, ...]

Route: 'search_authors' (GET)
Description: Returns authors filtered by names
Query Parameters: author_name (str) (default: '%')
Response Type: JSON Array
Response Parameters: [{id (int), author_name (str), num_books (int), num_reviews (int), num_ratings (int), avg_rating (float), min_rating (int), max_rating (int)}, ...]

Route: 'add_to_library/:user_id/:book_id' (POST)
Description: Adds a book to a user's library
Route Parameters: user_id (str), book_id (int)
Response Type: int
Response Parameters: status code (int)

Route: 'remove_from_library/:user_id/:book_id' (POST)
Description: Removes a book from a user's library
Route Parameters: user_id (str), book_id (int)
Response Type: int
Response Parameters: status code (int)

Route: 'check_in_library/:user_id/:book_id' (GET)
Description: Checks whether a book is in a user's library
Route Parameters: user_id (str), book_id (int)
Response Type: JSON Array (of length 1)
Response Parameters: [{user_id (str)}]

Route: 'get_rating/:user_id/:book_id' (GET)
Description: Returns a user's rating for a book
Route Parameters: user_id (str), book_id (int)
Response Type: JSON Array (of length 1)
Response Parameters: [{rating (int)}]

Route: 'update_review/:user_id/:book_id/:rating' (POST)
Description: Adds a rating for a book if not already rated, and updates the rating otherwise
Route Parameters: user_id (str), book_id (int), rating (int)
Response Type: int
Response Parameters: status code (int)

Route: '/register_user/:user_name/:email/:password/:name' (POST)
Description: Adds a new user
Route Parameters: user_name (str), email (str), password (str), name (str)

Response Type: int
Response Parameters: status code (int)

Route: 'handle_login/:user_name/:password' (POST)
Description: Attempts to log a user in
Route Parameters: user_name (str), password (str)
Response Type: int
Response Parameters: status code (int)

Route: '/get_user_id/:user_name' (GET)
Description: Returns the id of a user with a given username (used in login)
Route Parameters: user_name (str)
Response Type: JSON Array (of length 1)
Response Parameters: [{id (str)}]

7. Queries

We provide descriptions and the SQL code for 5 of the queries below. For the complex queries we provide optimized versions of the code (the unoptimized versions can be found in the Appendix section).

7.1. Book Information

7.1.1 Description

The query returns metadata and summary statistics (no. of reviews, average rating) for a given book. It is used in the books page.

7.1.2 Query

```
WITH book_trunc AS (
  SELECT id,
         title ,
         description ,
         format,
         publisher ,
         publish_date,
         num_pages,
         image_url
  FROM Book
  WHERE id = ${book_id}
)
SELECT B.title title,
       B.description  description ,
       B.format format,
       B.publisher  publisher ,
       B.publish_date publish_date,
       B.num_pages num_pages,
       COUNT(R.id) num_reviews,
       ROUND(AVG(R.rating), 1) avg_rating,
       B.image_url image_url
FROM book_trunc B
     JOIN Review R ON B.id = R.book_id
GROUP BY B.id
```

7.2. Book Recommendations

7.2.1 Description

The query recommends 16 books on a user's home page that the user hasn't read. In particular, for a given user U, the recommendation algorithm

1. Finds authors with ≥ 4.0 average rating from U
2. Given these authors, finds other users who have an average rating of 5 for at least one of the authors
3. Given these users, finds books in their library that they have rated ≥ 4.0 and aren't present in U's library
4. Returns 16 of these books

We note that it is possible for the query to return an empty result for example if U doesn't have any books in their library with a high enough rating. To address this, the query is part of an even larger query that ensures that 16 books are returned by filling in the blank spaces with the most reviewed books (across all users). We believe this is reasonable as the most reviewed books are a meaningful surrogate for the most popular books, and hence potentially enjoyable for U. We only include the query capturing the essence of the recommendation algorithm here and not the larger query as it is split into multiple functions. It can be found in the *routes.js* file on GitHub.

7.2.2 Query (Optimized)

```
WITH author_user AS (
    SELECT author_id,
           user_id ,
           AVG(rating) avg_rating
    FROM ABRU_View
    GROUP BY author_id, user_id
), top_authors AS (
    SELECT author_id
    FROM author_user
    WHERE user_id = '${user_id}' AND
          avg_rating >= 4.0
), similar_users AS (
    SELECT DISTINCT user_id
    FROM author_user
    WHERE user_id != '${user_id}' AND
          author_id IN (SELECT author_id FROM
               top_authors) AND
          avg_rating = 5.0
), similar_users_books AS (
    SELECT DISTINCT R.book_id book_id
    FROM Review R
         JOIN similar_users SU ON R.user_id = SU.user_id
    WHERE rating >= 4.0 AND
          R.book_id NOT IN (
            SELECT book_id
            FROM In_Library
            WHERE user_id = '${user_id}'
```

```
        )
    LIMIT 16
)
SELECT B.id book_id,
       B.title  book_title,
       B.image_url image_url
FROM Book B
     JOIN similar_users_books SUB ON B.id =
         SUB.book_id;
```

7.3. Author Perceptions

7.3.1 Description

For each author, the query performs a full outer join to return the no. of users that like, are neutral towards, and dislike the author. For a given author, we assume that a user dislikes the author if their average rating for the author is < 3, is neutral towards the author if their average rating for the author is 3, and like the author if their average rating for the author in > 3. The query is used in the trending page as a measure of authors' community perceptions.

7.3.2 Query (Optimized)

```
WITH author_user AS (
  SELECT author_id,
         author_name,
         user_id,
         AVG(rating) avg_rating
  FROM ABRU_View
  WHERE author_name LIKE '%${name}%'
  GROUP BY author_id, user_id
), dislikes AS (
  SELECT author_id,
         author_name,
         COUNT(user_id) num_dislikes
  FROM author_user
  WHERE avg_rating < 3.0
  GROUP BY author_id, author_name
), neutral AS (
  SELECT author_id,
         author_name,
         COUNT(user_id) num_neutral
  FROM author_user
  WHERE avg_rating = 3.0
  GROUP BY author_id, author_name
), likes AS (
  SELECT author_id,
         author_name,
         COUNT(user_id) num_likes
  FROM author_user
  WHERE avg_rating > 3.0
  GROUP BY author_id, author_name
), DN AS (
    SELECT D.author_id,
           D.author_name,
           num_dislikes,
           num_neutral
    FROM dislikes D
         LEFT JOIN neutral N ON D.author_id =
             N.author_id
    UNION
    SELECT N.author_id,
           N.author_name,
           num_dislikes,
           num_neutral
    FROM dislikes D
         RIGHT JOIN neutral N ON D.author_id =
             N.author_id
), DNL AS (
    SELECT DN.author_id,
           DN.author_name,
           num_dislikes,
           num_neutral,
           num_likes
    FROM DN
         LEFT JOIN likes ON DN.author_id =
             likes.author_id
    UNION
    SELECT likes.author_id,
           likes.author_name,
           num_dislikes,
           num_neutral,
           num_likes
    FROM DN
         RIGHT JOIN likes ON DN.author_id =
             likes.author_id
)
SELECT author_id id,
       author_name name,
       IFNULL(num_dislikes, 0) num_dislikes,
       IFNULL(num_neutral, 0) num_neutral,
       IFNULL(num_likes, 0) num_likes
FROM DNL
```

7.4. Author Statistics

7.4.1 Description

The query returns summary statistics about authors such as their no. of books/reviews/ratings, and min/-max/avg ratings. It is used in the author page to provide information about the authors.

7.4.2 Query (Optimized)

```
SELECT author_id id,
       author_name,
       COUNT(DISTINCT book_id) num_books,
       COUNT(review_id) num_reviews,
       COUNT(rating) num_ratings,
       ROUND(AVG(rating), 1) avg_rating,
       MIN(rating) min_rating,
       MAX(rating) max_rating
FROM ABRU_View
WHERE author_name LIKE '%${name}$%'
GROUP BY author_id;
```

7.5. Top Reviewers

7.5.1 Description

The query returns information about the 15 top reviewers (users with the most no. of reviews), such as the

no. of books in their library, the no. of reviews they have written, the no. of votes/comments they have received for their reviews etc. It is used in the trending page.

7.5.2 Query (Optimized)

Although originally a single query, the optimized query is split into three components to allow for re-usability in other parts of the application (details are provided in the Query Optimization & Performance Evaluation section).

As shown below, the first component returns information for a given user (this is reused in the user's homepage).

```
WITH user_trunc AS (
    SELECT id user_id,
           name user_name
    FROM User
    WHERE id = '${user_id}'
), in_library_trunc AS (
    SELECT user_id,
           COUNT(*) num_library
    FROM In_Library
    WHERE user_id = '${user_id}'
    GROUP BY user_id
), review_trunc AS (
    SELECT user_id,
           COUNT(rating) num_ratings,
           AVG(rating) avg_rating,
           COUNT(id) num_reviews,
           SUM(num_votes) num_votes,
           SUM(num_comments) num_comments
    FROM Review
    WHERE user_id = '${user_id}'
    GROUP BY user_id
)
SELECT UT.user_id AS id,
     UT.user_name,
     ILT.num_library,
     RT.num_ratings,
     RT.avg_rating,
     RT.num_reviews,
     RT.num_votes,
     RT.num_comments
FROM user_trunc UT
   JOIN in_library_trunc ILT ON UT.user_id = ILT.user_id
   JOIN review_trunc RT ON ILT.user_id = RT.user_id
```

As shown below, the second component returns the IDs of the 15 users with the most reviews.

```
SELECT user_id
FROM Review
GROUP BY user_id
ORDER BY COUNT(id) DESC
LIMIT 15
```

Finally, as shown below, the third component is JavaScript code that calls the first component for each of the user IDs returned by the second component and returns the aggregate results.

```
const topReviewers = async function(req, res) {
  let user_ids = await getTopReviewerIds();
  const arr = [];
  for (let i = 0; i < user_ids.length; i++){
    const user_id = user_ids[i].user_id;
    const user_information = await
          getUserInformation(user_id);
    arr.push(user_information[0]);
  }
  res.json(arr);
}
```

8. Query Optimization & Performance Evaluation

A significant challenge while working on the optimizations was that MySQL doesn't support query caching and materialized views. Despite this, we found workarounds and detail our query optimization efforts below.

8.1. Initial Steps

We begin by briefly mentioning that we have tried to adopt good query writing practices, such as using Common Table Expressions (CTEs), avoiding Cartesian products, removing unnecessary joins, and pushing selections and projections to minimize the sizes of intermediate results. Our queries have changed depending on the needs of our application and hence adhering to these practices has been a work-in-progress.

8.2. Author-Book-Review-User (ABRU) View

We observed that 3 complex queries had a similar and expensive initial setup that involved joining the *Author*, *Written_By*, and *Review* tables to generate aggregate statistics. These queries were expected to run frequently, and we did not anticipate updates to the data having a significant impact on the aggregate statistics. Hence we decided to create a materialized view for the setup, which the queries could utilize. The materialized view could be separately updated periodically at the backend to prevent the data from becoming stale, and this could be done at a lesser frequency compared to the frequency at which the complex queries are called[1].

Since MySQL did not support materialized views,

[1]We do not explicitly implement this update in the application as we are simulating a materialized view with a table here. Instead, we have mentioned this as part of our justification to use materialized views, and believe it could feasibly be done if we were to switch to a database provider with native support for these views.

Query	Before ABRU_View	After ABRU_View		
		Before AU_Index	After AU_Index	After AU_Index & Rating_Index
Book Recommendations	20.34	2.45	1.53	1.51
Author Perceptions	12.18	3.36	2.16	2.20
Author Statistics	96.42	1.21	0.3	0.31

Table 2. Performance evaluations for Book Recommendations, Author Perceptions, and Author Statistics queries (all values are averages of 10 runs in seconds)

Query	Unoptimized	Optimized
Top Reviewers	22.19	2.32

Table 3. Performance evaluations for Top Reviewers query (all values are averages of 10 runs in seconds)

we simulated them using a table that we called *ABRU_View*.

```
CREATE TABLE ABRU_View (
    SELECT A.id author_id,
           A.name author_name,
           R.book_id book_id,
           R.id review_id,
           R.user_id user_id,
           R.rating rating
    FROM Author A
        JOIN Written_By WB on A.id = WB.author_id
        JOIN Review R on WB.book_id = R.book_id
);
```

Coupled with some query optimizations (eg. removing unnecessary joins on the *User* table, pushing a LIMIT to slightly earlier in a query), introducing the view yielded substantial improvements in query runtimes. This can be seen in the *Before ABRU_View* and *After ABRU_View/Before AU_Index* columns in Table 2.

We also note that for the *Author Statistics* query, with the *ABRU_View* we were able to substantially simplify the query especially by removing joins and PARTITION BY statements. This explains the change in the query runtime from 96.42 s to 1.21 s.

8.3. Author-User (AU) Index

We observed that the complex queries that were now using *ABRU_View* were grouping by *author_id* or (*author_id*, *user_id*). Hence, to further improve query performance, we created a composite index on the view.

```
CREATE INDEX AU_Index ON ABRU_View(author_id,
    user_id);
```

On examining the EXPLAIN PLANs for the queries with and without the index, we observed that the index replaced a FULL TABLE SCAN for the group aggregations with a FULL INDEX SCAN. As seen in the *After AU_Index* column in Table 2, this yielded further improvements in query runtimes.

8.4. Rating_Index

We observed that the complex queries that were now using *ABRU_View* were making use of its *rating* column. So, as an experiment, we added an index for this column.

```
CREATE INDEX Rating_Index ON ABRU_View(rating);
```

As can be seen in the last column of Table 2, the index yielded no additional improvements in query runtime. On examining the queries we noticed that this was perhaps because the rating column was being used to compute the average rating, and hence the query could not really benefit from the index. This was confirmed when we looked at the query EXPLAIN PLANs and saw that they were not making use of the index.

8.5. Splitting Queries into Reusable Components

As mentioned in the Queries section, we intended to use the *Top Reviewers* query in the trending page to return information about the top 15 reviewers. However, we noticed that the query had a valuable component that returned information about a user based on their ID. This component could be reused in the user's home page and hence we decided to split the query into separate components and stitch them together using *ASYNC/AWAIT* in JavaScript. The first component could retrieve *user_id*s for the top reviewers, and the second component could asynchronously retrieve information about these users based on their *user_id*s. The improvements in query runtime can be seen in Table 3. They mainly came from query restructuring and not entirely from the split itself. However, we wanted to showcase this here as it was our attempt to both optimize a query and reuse it for multiple features.

9. Technical Challenges

We have faced several challenges while working on the project, and found different ways to address them. Some of them include:

- **Dataset**: Our goal was to find a dataset that was both interesting and manageable in size. While the Goodreads source we previously mentioned contained data for several genres eg. children's books, mystery etc., we found that some datasets were too large to work with locally. Ultimately, we chose the poetry dataset as it was the perfect size for the project.

- **Preprocessing**: During data preprocessing, we discovered that approximately 43% of the book cover image links were broken. As including book cover images was crucial to our application's aesthetics, we wrote a scraper to extract the links from Goodreads, ultimately reducing the number of broken image links to just two. Additionally, for the users we only had anonymized user IDs. To add more information for users we used Python's *Faker* library to generate dummy usernames, passwords, names, and emails.

- **Complex Queries**: Writing complex queries with runtimes suitable for query optimization was a challenging task. Our initial attempts at query writing yielded queries that had large runtimes, but the queries felt contrived and not particularly useful in the application. We spent a considerable amount of time brainstorming to come up with queries that were complex and would also meaningfully contribute to the user experience, such as the recommendation, author perception, author statistics, and top reviewers queries.

- **Query Optimization**: We encountered a challenge in query optimization due to MySQL's lack of query caching and materialized views. Despite some initial attempts at optimization that focused on making simple changes to the query structure, we found that the optimized queries still took considerable time to run, making them unsuitable for the application. However, we discovered that many of the queries had a similar initial setup. We leveraged this similarity by creating a materialized view for the setup (in particular, we simulated a materialized view using a table), and adding an index to it. This approach resulted in significant improvements to our query runtimes, making them feasible for use in the application.

10. Appendix

10.1. Book Recommendations Original (Unoptimized) Query

```
WITH author_user AS (
  SELECT A.id author_id,
         U.id  user_id,
         AVG(rating) AS avg_rating
  FROM Author A
       JOIN Written_By WB ON A.id = WB.author_id
       JOIN Review R ON WB.book_id = R.book_id
       JOIN User U ON R.user_id = U.id
  GROUP BY A.id, U.id
), top_authors AS (
  SELECT author_id
  FROM author_user
  WHERE user_id = '${user_id}' AND
        avg_rating >= 4.0
), similar_users AS (
  SELECT DISTINCT user_id
  FROM author_user
  WHERE user_id != '${user_id}' AND
        author_id IN (SELECT author_id FROM
             top_authors) AND
        avg_rating = 5.0
), similar_users_books AS (
  SELECT DISTINCT R.book_id book_id
  FROM Review R
       JOIN similar_users SU ON R.user_id = SU.user_id
  WHERE rating >= 4.0 AND
        R.book_id NOT IN (
          SELECT book_id
          FROM In_Library
          WHERE user_id = '${user_id}'
          )
)
SELECT B.id book_id,
     B.title  book_title,
     B.image_url image_url
FROM Book B
   JOIN similar_users_books SUB ON B.id = SUB.book_id
LIMIT 16;
```

10.2. Author Perceptions Original (Unoptimized) Query

```
WITH author_user AS (
 SELECT A.id author_id,
        A.name author_name,
        U.id  user_id,
        AVG(rating) avg_rating
 FROM Author A
      JOIN Written_By WB ON A.id = WB.author_id
      JOIN Review R ON WB.book_id = R.book_id
      JOIN User U on U.id = R.user_id
      WHERE A.name LIKE '%${name}%'
      GROUP BY A.id, U.id
), dislikes AS (
  SELECT author_id,
         author_name,
         COUNT(user_id) num_dislikes
  FROM author_user
  WHERE avg_rating < 3.0
  GROUP BY author_id, author_name
), neutral AS (
  SELECT author_id,
         author_name,
         COUNT(user_id) num_neutral
  FROM author_user
  WHERE avg_rating = 3.0
  GROUP BY author_id, author_name
```

```
), likes AS (
  SELECT author_id,
         author_name,
         COUNT(user_id) num_likes
  FROM author_user
  WHERE avg_rating > 3.0
  GROUP BY author_id, author_name
), DN AS (
    SELECT D.author_id,
           D.author_name,
           num_dislikes,
           num_neutral
    FROM dislikes D
         LEFT JOIN neutral N ON D.author_id =
             N.author_id
    UNION
    SELECT N.author_id,
           N.author_name,
           num_dislikes,
           num_neutral
    FROM dislikes D
         RIGHT JOIN neutral N ON D.author_id =
             N.author_id
), DNL AS (
    SELECT DN.author_id,
           DN.author_name,
           num_dislikes,
           num_neutral,
           num_likes
    FROM DN
         LEFT JOIN likes ON DN.author_id =
             likes.author_id
    UNION
    SELECT likes.author_id,
           likes .author_name,
           num_dislikes,
           num_neutral,
           num_likes
    FROM DN
         RIGHT JOIN likes ON DN.author_id =
             likes.author_id
)
SELECT author_id id,
       author_name name,
       IFNULL(num_dislikes, 0) num_dislikes,
       IFNULL(num_neutral, 0) num_neutral,
       IFNULL(num_likes, 0) num_likes
FROM DNL
```

10.3. Author Statistics Original (Unoptimized) Query

```
WITH author_reviews AS (
  SELECT A.id author_id,
         A.name author_name,
         WB.book_id book_id,
         R.rating rating,
         R.text text
  FROM Author A
       JOIN Written_By WB ON A.id = WB.author_id
       JOIN Review R ON WB.book_id = R.book_id
), author_rating_statistics AS (
  SELECT DISTINCT author_id,
                  author_name,
      AVG(rating) OVER (PARTITION BY author_id) AS
          avg_rating,
      MIN(rating) OVER (PARTITION BY author_id) AS
          min_rating,
      MAX(rating) OVER (PARTITION BY author_id) AS
          max_rating,
      COUNT(rating) OVER (PARTITION BY author_id)
          AS num_ratings,
      COUNT(text) OVER (PARTITION BY author_id)
          AS num_reviews
  FROM author_reviews
), author_num_works AS (
  SELECT author_id,
      COUNT(DISTINCT book_id) num_books
  FROM Written_By
  GROUP BY author_id
)
SELECT ARS.author_id author_id,
       ARS.author_name author_name,
       ANW.num_books num_books,
       ARS.num_reviews num_reviews,
       ARS.num_ratings num_ratings,
       ROUND(ARS.avg_rating, 1) avg_rating,
       ARS.min_rating min_rating,
       ARS.max_rating max_rating
FROM author_review_statistics ARS
     JOIN author_num_works ANW ON ARS.author_id =
         ANW.author_id;
```

10.4. Top Reviewers Original (Unoptimized) Query

```
WITH top_reviewer_ids AS (
    SELECT user_id
    FROM Review
    GROUP BY user_id
    ORDER BY COUNT(id) DESC
    LIMIT 15
), user_trunc AS (
    SELECT id user_id,
           name user_name
    FROM User
), in_library_trunc AS (
    SELECT user_id,
           COUNT(*) num_library
    FROM In_Library
    GROUP BY user_id
), review_trunc AS (
    SELECT user_id,
           COUNT(rating) num_ratings,
           AVG(rating) avg_rating,
           COUNT(id) num_reviews,
           SUM(num_votes) num_votes,
           SUM(num_comments) num_comments
    FROM Review
    GROUP BY user_id
), reviewer_info AS (
    SELECT UT.user_id,
           UT.user_name,
           ILT.num_library,
           RT.num_ratings,
           RT.avg_rating,
           RT.num_reviews,
           RT.num_votes,
           RT.num_comments
    FROM user_trunc UT
         JOIN in_library_trunc ILT ON UT.user_id =
             ILT.user_id
         JOIN review_trunc RT ON ILT.user_id =
```

```
                RT.user_id
)
SELECT RI.user_id,
       RI.user_name,
       RI.num_library,
       RI.num_ratings,
       RI.avg_rating,
       RI.num_reviews,
       RI.num_votes,
       RI.num_comments
FROM reviewer_info RI
     JOIN top_reviewer_ids TR ON TR.user_id =
          RI.user_id;
```