Group 44 - Amazing Reads

Team Name: The Bookworms
Title: Amazing Reads

Group members:
Renisa Pati, Rut Vyas, Sam Thudium, Tanvi Gupta

1. Introduction and Project Goals

In today's age of information overload, there are so many books available that it can be overwhelming for readers to find a book they would like to read. Through our web application we wish to help readers find a book that suits their interests, tastes, and preferences. The readers would be able to explore and discover new books, authors, and genres they may have never heard of before. Recommendations of new books based on their preferences and books they have liked in the past would make their reading experience more personalized.

In order to accomplish this goal, we turned to the popular book sharing site, Goodreads. We chose to use data from this source for several reasons including (1) the vast amount of information that it has on books and authors, (2) the extensive culture of user reviews that exists for the site provides us with millions of text reviews for the books in our database, and (3) the source data included other interesting information such as book genres and book series information. Altogether, we think this dataset has a compelling structure for a relational database project and the granularity of the data we use has allowed us to compute interesting statistics or retrieve subsets of the data using non-trivial predicates.

Finally, though it was not the primary goal of our project, we wanted to produce an aesthetic website that facilitates easy and intuitive user navigation. As a group without any background in front-end development, this was an arduous task, but we are incredibly proud of the final product. We hope the underlying goal of this project shines through: connecting people to new stories.

2. Architecture

Data Pre-processing: Python on Google Colab
Database: MySQL hosted in AWS/EC2
Front-end of Web App: React and Material-UI
Back-end of Web App: Node.js
Testing: Jest and SuperTest

3. Data

3.1 Description of the Dataset

- **Books :** It's a dataset with all the information about different books like authorid, average rating, similar books, language, publisher, etc where each row is a unique book.
- **Reviews :** This is a dataset with complete multilingual review text and ratings by user.
- **Detailed authors info:** Dataset with information about the name, rating count, average rating of different authors.
- **Fuzzy genres:** This a very fuzzy version of book genres. These tags are extracted from users' popular shelves by a simple keyword matching process.
- **Series:** Information about series of a book as an aggregate like the name and number of books in that series.

3.2 A link to the dataset:

https://sites.google.com/eng.ucsd.edu/ucsdbookgraph/home?authuser=0

3.3 Pre-processing:

All of these files are in json format. We will pre-process and filter them. Thus, separate lists within a json attribute will possibly be converted into new tables.

We will begin building the database using a subset of the following:

Table name	Size	Cardinality	Arity
Detailed Authors	105.9 MB	829528	5
Reviews	5GB	~15M	11
Books	2GB	2.36M	20-25
Fuzzy Genres	200 MB	2.36 M	2
Series	24.3 MB	400K	7

4. Database

Data was cleaned on Google Colab using the pandas library. Efforts were made to replace missing values where an intuitive value existed. One of the common procedures that this data requires was unnesting the JSON documents into a relational data structure. We took the approach of breaking these fields off into separate tables rather than repeating information in a single table. For instance, for books with more than one author, this unwinding would produce multiple rows for a single book. We separated this as a relational table between authors and books, called Written_By. Each attribute of every relation, thus, is functionally dependent only on the primary key of the relation. These efforts allowed us to produce normalized schemas, which are in 3NF (shown in detail in **Appendix E**). A final ER Diagram can be seen in **Appendix A**.

5. Web App Description

Signup: We prepended a regular user authentication facility to our application where a new user can sign up and their details shall get inserted into the database containing User Information. If they are an already existing user, they can click on the message that redirects them to the SignIn page.

SignIn: For existing users, we have a SignIn page where they can enter their credentials, get it validated against their existing information. Once validated, they are redirected to the homepage. The SignIn page also provides a clickable message where it lets new users navigate through the Signup page first before Signing in to the application.

Homepage: Upon entering the website, users will be greeted with a banner welcoming them to AMAZINGREADS. The page helps users find books using several mechanisms, including a Surprise Me button, which takes them to a random book page, a Recommended Books section, which holds books similar to those they have liked in the past, the top 10 books of the month, and the user's personal reading list. Upon clicking any of these, the user will be sent to that book's page for more information. Should they wish to explore genres or authors, the navigation bar will allow them to do so.

Book page: The book page displays all the important information that a user might want to learn about a new book. Basic meta information is displayed including the genres and series to which the book belongs. The genre attributes are clickable and will redirect users to that genre's page while the series title will pop out a modal showing the user the other books in that series. Similar books to the current one are shown if they exist and a chart displaying the rating history of the book compared to that of the genre that the book is in, again conditioned on the existence of the rating history data. Finally, the reviews that have been written about the book are displayed below.

Authors page: The authors page displays a paginated table of all authors who have written a book stored in the database. They're ranked by a weighted average of the average author rating, so many of the user's favorite authors may appear towards the top or on the first few pages of the table. Here the user can compare some metrics about each author such as average rating and number of reviews. Clicking on an author's name will pop out a modal which displays more information about the author as well as all of the books that they have written. If clicked, these books redirect the user to that book's page.

Genres page: This page displays the top books that are part of the selected genre. The page is reachable either from the navigation bar dropdown menu or from a book page, where the user can click on a genre name. Each book in the genre that is displayed will take the user to that book's page.

6. Complex Queries

★ **Route:** `/book_recs_rand_genre/:user_id`

Function: Find two random genres and select the top 2 rated books within those genres and return these as recommended books for the user. Used to populate the Recommended books section. This query was optimized by refactoring the joining methodology and pushing selects and projects.

```
WITH genre_random AS (
  SELECT bg.genre_id AS genre_id
  FROM Users_Liked u
  JOIN Book_Genres bg ON u.book_id = bg.book_id
  WHERE user_id = 1
  ORDER BY RAND()
  LIMIT 2
    )
(SELECT g.genre_name, b.title, b.image_url
    FROM Genres g
        NATURAL JOIN Book_Genres bg
      JOIN
        (SELECT * FROM genre_random
         LIMIT 1) g_user ON bg.genre_id = g_user.genre_id
        JOIN Book b ON bg.book_id = b.book_id
    ORDER BY b.average_rating
    LIMIT 2)
    UNION
    (SELECT g.genre_name, b.title,b.image_url
    FROM Genres g
        NATURAL JOIN Book_Genres bg
      JOIN
        (SELECT * FROM genre_random
         LIMIT 1,1) g_user ON bg.genre_id= g_user.genre_id
        JOIN Book b ON bg.book_id = b.book_id
    ORDER BY b.average_rating
    LIMIT 2);
```

★ **Route:** `rating_history`

Function: Find the yearly average rating of the current book for all years in which it has received reviews. Return this along with the yearly average rating of all books that are part of the same genre. A cached table was created to hold the average rating reviewers had given books in each genre in each year. This was an immensely expensive query so we opted to store it and retrieve from it rather than recalculate every time. The remainder of the query was further optimized by limiting the size of sub-tables by pushing sections down.

```
WITH ya_filtered AS (SELECT *
```

```
    FROM yearly_average ya
    WHERE genre_id = (SELECT genre_id
                      FROM Book_Genres
                      WHERE book_id = ${curr_id}
                      LIMIT 1)
    )
  SELECT ya2.genre_id, r.year_added,
         ROUND(AVG(r.rating), 2) AS average_rating,
         COUNT(*) AS review_count,
         ROUND(ya2.yearly_average, 2) AS yearly_average,
         (AVG(r.rating) > yearly_average) AS gt_yearly_avg
  FROM (SELECT * FROM Reviews WHERE book_id = ${curr_id}) r
    JOIN ya_filtered ya2 ON ya2.year_added = r.year_added
  GROUP BY r.year_added, ya2.genre_id
  ORDER BY ya2.genre_id, year_added;
```

★ **Route:** /top_ten_books_month

Function: First find the top 100 books of the month based on reviews submitted during the current month. Then take 10 of these randomly and obtain author information on the randomly selected books to obtain the book and author information of 10 books of the month. This query was optimized by creating an index on book_id, month_added, and rating, and resulted in huge improvements in query runtime, likely because the computation could be done at the data entry level rather than the data record level.

```
WITH month_top_reviewed AS (
    SELECT r.book_id , ROUND(COUNT(*) * AVG(rating), 2) AS wt_avg
    FROM Reviews r
    WHERE r.month_added = MONTH(CURRENT_TIMESTAMP)
    GROUP BY r.book_id
    ORDER BY wt_avg DESC
    LIMIT 100 ),
  b_top_ten AS (
    SELECT tt.book_id, b.title, b.image_url, tt.wt_avg
    FROM month_top_reviewed tt
    JOIN (SELECT book_id, title, image_url
        FROM Book) b ON tt.book_id = b.book_id
    ORDER BY RAND()
    LIMIT 10)
  SELECT btt.*, GROUP_CONCAT(DISTINCT A.name ORDER BY A.name DESC
SEPARATOR ', ') AS authors
  FROM b_top_ten btt
      JOIN Written_By wb ON btt.book_id = wb.book_id
        JOIN (SELECT author_id, name FROM Authors) A ON A.author_id =
wb.author_id
  GROUP BY btt.book_id
```

```
  ORDER BY btt.wt_avg DESC;
```

★ **Route:** /surprise_me/:user_id

Function: The 'Surprise Me!' query takes in the current user's id, accesses the genres of books they have liked and selects the genre from which they have read the fewest books. It then finds the average rating of books in that genre and returns a random book_id with an average rating less than the genre-average rating. This query was optimized purely by rethinking the strategy taken and then carefully pushing selections and projections. The joins were ordered carefully as well to avoid large intermediate results. This was also very successful in reducing the runtime of this query.

```
SELECT book_id
FROM Book b
WHERE b.average_rating <
      (SELECT AVG(average_rating) AS avg_genre_rating
        FROM
  (SELECT MIN(genre_id) AS genre_id
   FROM (SELECT genre_id, COUNT(genre_id)
         FROM (SELECT ul.book_id, bg.genre_id
   FROM (SELECT book_id FROM Users_Liked WHERE user_id = ${user_id) ul
              JOIN Book_Genres bg ON ul.book_id = bg.book_id) t1
                        NATURAL JOIN Genres g
                    GROUP BY genre_id) t2) t3
INNER JOIN Book_Genres bg ON bg.genre_id = t3.genre_id
INNER JOIN Book B on bg.book_id = B.book_id)
ORDER BY RAND()
LIMIT 1;
```

★ **Route:** /book_author_series/:book_id

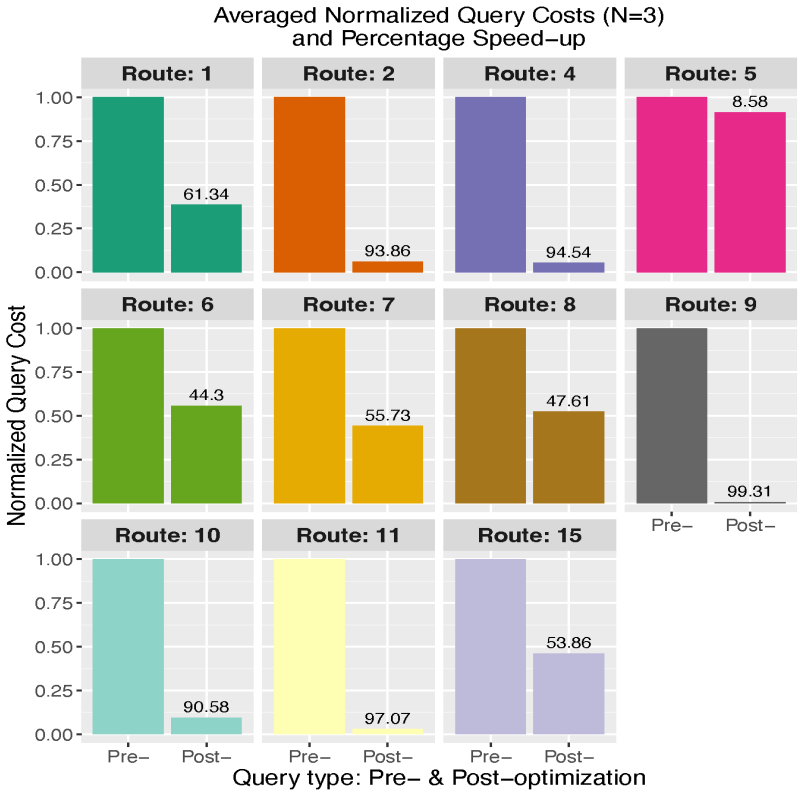
Function: Given a book_id, find information about the author who wrote it as well as books that are part of the same series. This is used to populate the Book Series modal on the book page.

```
SELECT a.name AS author, w.role, bs.title AS series_title
  FROM (SELECT book_id, series_id FROM Book WHERE book_id = ${curr_id})
b
    INNER JOIN Written_By w ON w.book_id = b.book_id
    INNER JOIN Authors a ON w.author_id = a.author_id
    INNER JOIN Book_Series bs ON b.series_id = bs.series_id;
```

7. Performance Evaluation

Below are the performance metrics before and after optimizing our queries. On average after optimization, the queries cost approximately 75% less than they did in the unoptimized state.

However, if weighted by the proportion of the pre-optimization runtime, the average reduction in query cost is >98%. (**Appendix C**). Costs were calculated using MySQL's built in timer, which returns the query runtime in milliseconds. In the graph below, the actual time costs are displayed as a normalized proportion of the pre-optimization cost. Raw outputs for the actual rows and times for each query can be viewed in **Appendix C**. The float value above each of the post-optimization bars represents the percentage speed-up achieved by the query.



These improvements were achieved using a combination of indexing, pushing selections and projections, creating CTEs, and optimizing the ordering of joins. For example, several queries (i.e Route 9) rely on calculating a weighted average rating for the purposes of ordering results. By adding an index on these fields, we were able to perform such calculations at the data entry level rather than the data record level. Route 11, 'Surprise Me!', is a good example of a query optimized purely through basic rules of query optimization - pushing selections, projections, and correctly ordering joins; the query makes extensive use of subqueries in which the join keys are selected and only the necessary columns are projected. The result is that the most expensive join with the Book table is made as small as possible. This resulted in a >98% reduction in query time.

On the front end, we were able to get fairly fast responses when fetching data using our queries. The more complex queries took some time to return data, but they are still fast enough to facilitate good user navigation of the website. Times were averaged across multiple client requests and were recorded using Google Chrome's built-in network timing monitor (**Appendix E**).

8. Technical Challenges

Data management: Processing and managing the large amount of data we had was challenging and time-consuming. This data required unnesting the JSON documents into a relational data structure. We took the approach of breaking these fields off into separate tables rather than repeating information in a single table. These efforts allowed us to produce normalized schemas. Also, we had to limit the scope to the Mysteries-Thriller-Crime subset. We made this decision upon the advice of Professor Davidson. Even then, this subset is still quite large and allows plenty of room for query optimization.

Performance issues: Handling large datasets affects the performance of an application and we faced this issue too. Several of the final relations contain hundreds of thousands, if not more than a million rows, making joins quite expensive. Since the normalized form of our database requires joins quite frequently, many unoptimized queries were quite heavy, with runtimes of the originals ranging from 6 seconds to well over 60 seconds. Clearly, these times do not allow for a smooth user experience on the front end of our application, so we had to perform extensive optimization to improve these times.

User experience: Dealing with large datasets can be overwhelming for users, particularly if they are presented with too much information at once. We tried to design intuitive user interfaces. We implemented features like pagination and ordering and provided meaningful data visualizations to help users navigate the data effectively. Additionally, none of us come from a web development background, so we had never used React or Node.js before. Understanding how to link pages to one another was a significant hurdle, as well as implementing non-trivial features like the dropdown menu of the NavBar, Modals, and image carousels. Lots of hours went into understanding how these elements fit together and demonstrates our determination to make the most out of this project.

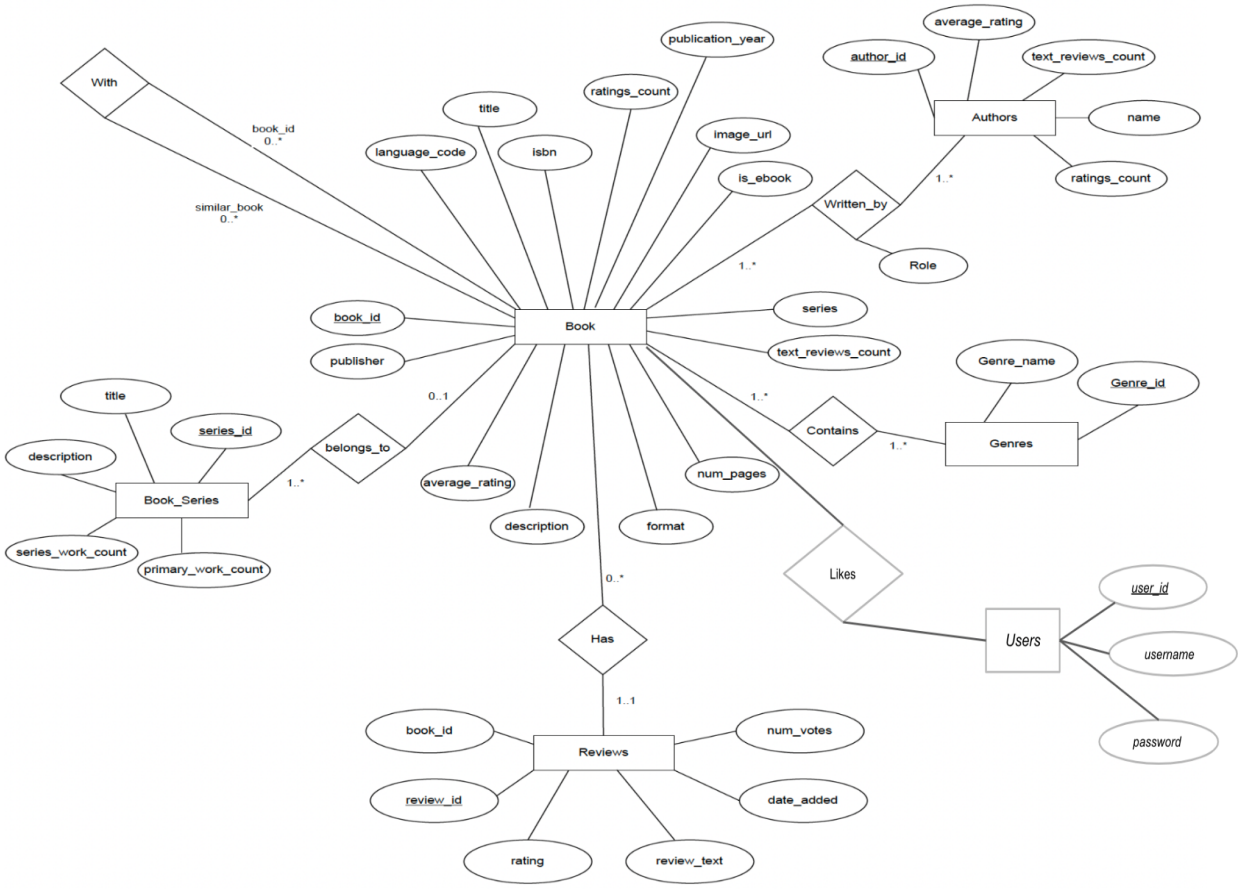
9. Extra Credit

We implemented two extra credit tasks for this project. The first was backend route testing which we implemented using Jest and Supertest, a popular open-source testing framework for JavaScript code. These tests cover each of the main queries used in our web application and helped us to ensure that our queries were working properly on the front end.

We also implemented a login page for users and store their information in the Users table. The page is composed of both a sign up and sign in page both of which are navigable between each other. Upon signing in or signing up, the user should be redirected to the homepage of the app.

10. Appendix A: ER Diagram

Final ER Diagram for the relational database.



11. Appendix B: Relational Schemas

Below are the relational schemas of the final populated tables in our database along with the number of instances in each.

- **Authors**(author_id, name, average_rating, text_reviews_count, ratings_count)
- **Book**(isbn, book_id, text_reviews_count, series_id, language_code, is_ebook, average_rating, description, format, publisher, num_pages, publication_year, image_url, ratings_count, title)
- **Book_Genres**(book_id, genre_id, num_votes)
- **Book_Series**(description, title, series_id, primary_work_count)
- **Genres**(genre_id, genre_name)
- **Reviews**(book_id, review_id, rating, review_text, year_added, month_added, day_added, num_votes)
- **Similar_Books**(book_id, similar_book_id)
- **Written_By**(book_id, author_id, role)
- **Users**(user_id, username, password)
- **Users_Liked**(user_id, book_id)

Table	**Cardinality**
Authors	829529
Book	219235
Book_Genres	1128293
Book_Series	400390
Genres	16
Reviews	1849235
Similar_Books	443622
Written_By	283692
Users	1 (until more sign up!)
Users_Liked	7

12. Appendix C: Raw Query Costs

All times below are reported in milliseconds.

Route	Time	Cost1	Cost2	Cost3	AverageCost
1	Unoptimized	627.739	569	894	696.913
1	Optimized	237.729	231	339.5	269.4096667
2	Unoptimized	524	168	481	391
2	Optimized	33	20	19	24
4	Unoptimized	32743.238	35325	33980	34016.07933
4	Optimized	1728	1994	1851	1857.666667
5	Unoptimized	233	329	207	256.3333333
5	Optimized	185	166	352	234.3333333
6	Unoptimized	387	146	239	257.3333333
6	Optimized	136	148	146	143.3333333
7	Unoptimized	340	365	298	334.3333333
7	Optimized	145	132	167	148
8	Unoptimized	423	150	410	327.6666667
8	Optimized	158	129	228	171.6666667
9	Unoptimized	420000	420000	420000	420000
9	Optimized	3930	3159	1571	2886.666667
10	Unoptimized	6346.232	3515	2321	4060.744
10	Optimized	474	216	458	382.6666667
11	Unoptimized	162000	128000	120165	136721.6667
11	Optimized	4148	4120	3733	4000.333333
15	Unoptimized	1138.06	890	987	1005.02
15	Optimized	254.117	571	566	463.7056667

Weighted average query costs calculated in Excel as: `SUMPRODUCT(A:A, B:B) / SUM(B:B)`

where column A holds the post-optimization query time (in milliseconds) for each route and column B is the pre-optimization query time (in milliseconds) for each route

13. Appendix D: Front-end client response times

All times below are reported in milliseconds.

Route	Description	Time 1	Time 2	Time 3	Average
1	get genre	3560	967.8	382.29	1636.697
2	get book	628.1	324.84	336.9	429.947
3	get reviews	323.62	206.4	205.96	245.327
4	rating history	714.78	546.98	383.78	548.513
5	get books in series	134.7	304.97	327.62	255.763
6	get author/series title	357.7	241.75	260.87	286.773
7	get book genres	741.09	578.8	410.4	576.763
8	similar books	599.08	294.44	300.86	398.127
9	top 10 of the month	10880	4170	7410	7486.667
10	book recommendations	6880	8900	9910	8563.333
11	surprise me	13610	11570	3590	9590.000
12	get author details	614.86	411.83	353.7	460.130
13	get books by author	657.55	485.08	390.27	510.967
14	get user liked books	479.87	94.33	115.42	229.873
15	get authors ordered	5140	4620	4400	4720.000

14. Appendix D: Front-end client response times

- **Authors**(<u>author_id</u>, name, average_rating, text_reviews_count, ratings_count)
  F_{Author} = {author_id → name, ratings_count, text_reviews_count, average_rating}

- **Book**(isbn, <u>book_id</u>, text_reviews_count, series_id, language_code, is_ebook, average_rating, description, format, publisher, num_pages, publication_year, image_url, ratings_count, title)
  F_{Book} = {book_id → isbn, text_reviews_count, series_id, language_code, is_ebook, average_rating, description, format, publisher, num_pages, publication_year, image_url, ratings_count, title }

- **Book_Genres**(<u>book_id</u>, <u>genre_id</u>, num_votes)
  F_{Book_Genres} = {book_id, genre_id → num_votes}

- **Book_Series**(description, title, <u>series_id</u>, primary_work_count)
  F_{Book_Series} = {series_id → description, title, primary_work_count}

- **Genres**(<u>genre_id</u>, genre_name)
  F_{Genres} = {genre_id → genre_name}

- **Reviews**(book_id, <u>review_id</u>, rating, review_text, year_added, month_added, day_added, num_votes)
  $F_{Reviews}$ = {book_id, review_id → rating, review_text, year_added, month_added, day_added, num_votes}

- **Similar_Books**(<u>book_id</u>, <u>similar_book_id</u>)
  Both are keys, so this is in BCNF

- **Written_By**(<u>book_id</u>, <u>author_id</u>, role)
  $F_{Written\_By}$ = {book_id, author_id → role}

- **Users**(<u>user_id</u>, username, password)
  F_{Users} = {user_id → username, password}

- **Users_Liked**(<u>user_id</u>, <u>book_id</u>)
  Both are keys, so this is in BCNF