Report for Final Project template

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Title of the Project: Book Application

Name of the students: David Buchheit

Intro

The domain I decided to use for this application was on Books. I picked this domain since there is a lot you can with books such as finding new ones, rating, and having a to-do list. I decided to pick this domain since I was able to find a large set of books with reviews for a lot of them.

This application is a book application. This application has a list of books you can search for by different criteria. Upon finding a book you can save it for later. Books you have stored can be deleted, marked as read, and find books similar to it.

As I was the only member in the group I did everything the project and was definitely a lot of work. This report is organized by implementation -> data -> prototype -> practical use -> conclusion.

Web Architecture

For the application I built it from Laravel 4.2. Laravel is a PHP framework that does a lot of annoying stuff in PHP for you. For example, instead of doing <?php echo($var)?> you can just do {{$var}} and it will print the var to webpage. I decided to pick this as my architecture since I honestly don’t have any experience with 100% raw PHP on a website and didn’t have the time to learn it. However, the place I work at uses Laravel for web development and I know to do that. This architecture is also more related to real world applications since Laravel is the most popular PHP framework and raw PHP is rarely used in real world applications anymore.

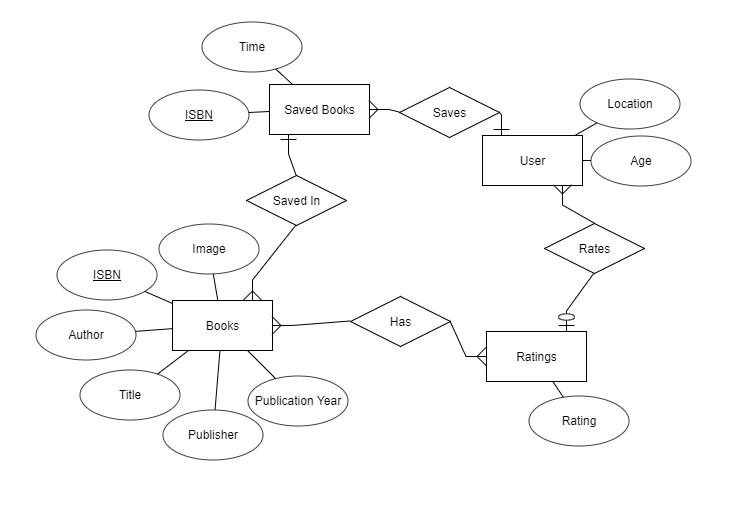
DBMS Architecture

I am using MariaDB as my DBMS which is like MySQL. The framework I’m using messes with how you do queries. It uses something called Eloquent which makes writing queries much easier. You sort of build your query and add stuff to it. For example, selecting everything from the table using Eloquent is simply: DB::connection(DATABASE)->table(TABLE)->get() or to limit your parameters you can do down the rows and columns you can do DB::connection(DATABASE)->table(TABLE)->where(‘Author’, ‘=’, ‘John’)->select(‘ISBN’, ‘BookTitle’)->get(). You can chain together other basic operators like joins, orWheres, unions, and orderBy but overall I really liked this implementation of doing the queries, It definitely makes a lot of the generating of queries much easier to generate and read. Some disadvantages of this though is that it can be harder to do some of the MySQL specific functionalities built in since raw SQL has to be put in a special way but in the end should work out. One of the other things that it can’t do is Stored Procedures. They are supported however the data you get back from it is much harder to work with and when I tried to use a stored procedure as an experiment to see the differences, the difference in speed/time was negligible and the average time difference was around 10% but with how fast I was able to get the queries down to, it didn’t really matter. 50ms -> 45ms isn’t much I think. The data I got back was much different than what I normally expect from a normal query I generate. Overall with the stored procedures I didn’t think the hassle of writing/implementing them was worth the hassle for such a little performance boost. Overall the technology I used for the SQL made my life much easier for me to implement them on the front end.

Dataset

The dataset I found has a massive collection on books. The dataset can be found at: <http://www2.informatik.uni-freiburg.de/~cziegler/BX/> . Actually, while writing this report and searching for what dataset I used for my data I found from my dataset didn’t properly import and I only had 1/3rd of the book reviews I should have. It went from about 350,000 reviewsto 1,149,780 reviews. Which I guess is really helpful and definitely something that was originally worrying me since I thought the dataset I had was originally small for the amount of books I had. The dataset I have now is BookRatings(n=1,149,780), Books(n = 271,379), Users(n=278,858). I had to clean the data by dropping some rows that I did not need for my application because it’s going more in depth than I ever would. For the tables I also had to add some data that was not originally there such as adding keys, and indexes. The dataset has standard information you would find on a book such as author, isbn, title etc. One thing I did like from the dataset that some of the books had working urls that you could grab an image off amazon from. However about 30% were broken so its sort of a gamble on if they worked. The ratings table had a lot of user ratings on books. The ratings are from 0-10 where 0 is lowest.

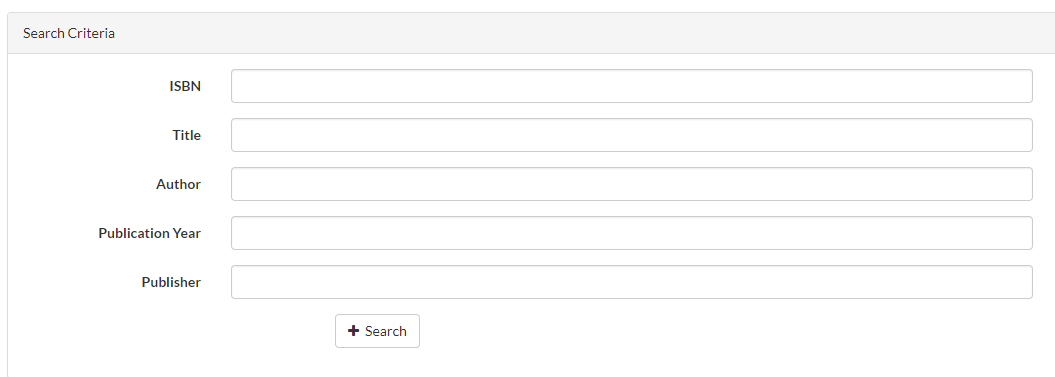
ER Diagram



Relational Model

Book-Ratings(User-ID, ISBN, User-ID, BookRating)  
Books(ISBN, BookTitle, BookAuthor, YearOfPublication, Publisher, ImageURLS, ImageURLM, ImageURLL)  
SavedBooks(id, ISBN, time)  
Users(User-ID, Location, Age)

Implementation

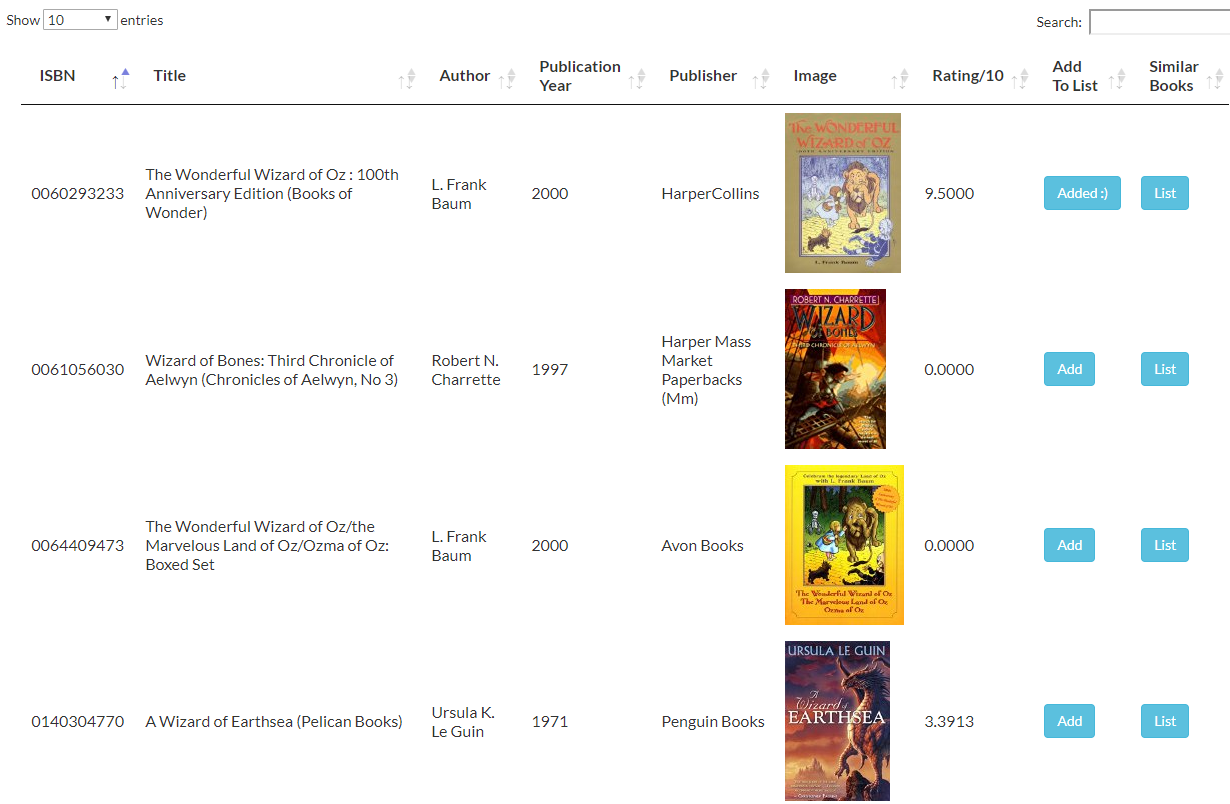


This is the main implementation of searching. Here you can search for criteria. You only need to input at least 1 criteria. It uses full text search which sacrifices accuracy for speed. For example, searching for ‘Wizard Of Oz’ will give you ‘The Wizard in the Tree’ as a result. I don’t know the exact algorithm for fulltext but I believe it is because ‘Wizard of Oz’ and ‘The Wizard in the tree’ contain Wizard. The query for this search is:

select `b`.\*, AVG(br.BookRating) as Rating from `books` as `b` left join `book-ratings` as `br` on `br`.`ISBN` = `b`.`ISBN` where MATCH(b.ISBN) AGAINST('ISBN') and MATCH(b.BookTitle) AGAINST(' BookTitle ') and MATCH(b.BookAuthor) AGAINST(‘BookAuthor’) and `b`.`YearOfPublication` = YEAR and MATCH(b.Publisher) AGAINST('PUBLISHER') group by `ISBN` limit 500

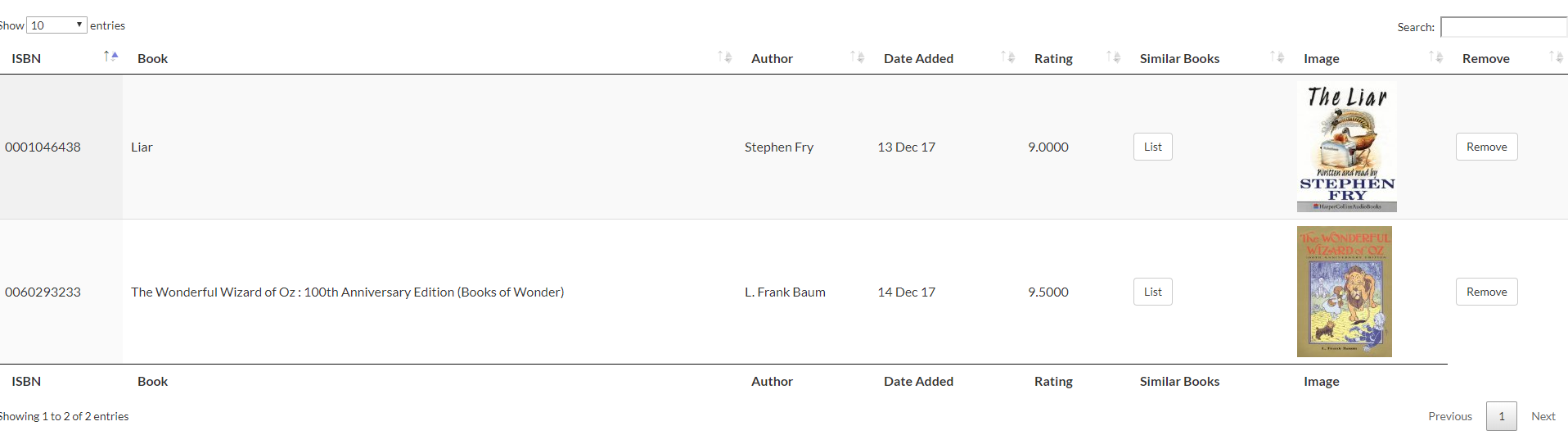
Some points in this is that I’m limiting the results to only 500 because grabbing more than 500 can take too long and if you’re searching through more than 500 items you should probably refine your search some more. I had to use left join on ratings since a book may not be rated and if I used inner join it just wouldn’t show those books.

Upon searching you’ll get this:



This uses the jQuery addon called datatables to display the data a clean format. I had pagination, sorting, searching and quick table formatting. This table contains all the information stored in the DB and some generated. The rating row comes from the average of all the book ratings under the book-ratings table. The image is also stored in the database and will be displayed if the link isn’t broken. The ‘Add To List’ button add the book to your own list of stored books. The “Similar Books” finds books that people who rated your book well also rated other books well.

Upon adding to list and going to inventory page you’ll get this:



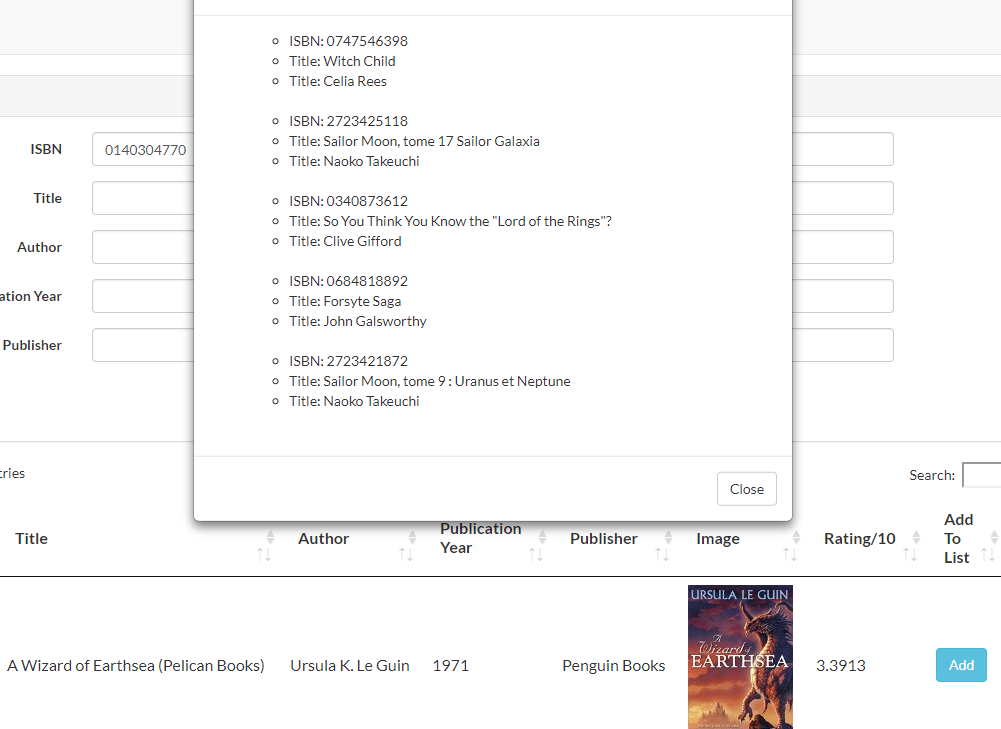
It the same format as the search page but has less ‘useless’ information display. It only shows the image, author, rating and title. The publication info I don’t believe is relevant for just having it on the page. The remove button deletes the book from your inventory off ISBN. There are some things I left out of this page such as your rating and date read. I decided to leave these out because of time restraints.

The basic SQL for grabbing your books is:

select `b`.`ISBN`, `b`.`BookTitle` as `Title`, `b`.`BookAuthor` as `Author`, `b`.`Publisher`, `b`.`ImageURLM` as `Image`, `sb`.`Time`, AVG(br.BookRating) as Rating from `books` as `b` inner join `savedbooks` as `sb` on `sb`.`ISBN` = `b`.`ISBN` left join `book-ratings` as `br` on `br`.`ISBN` = `sb`.`ISBN` group by `ISBN`

I had to use left join on ratings since a book may not be rated and if I used inner join it just wouldn’t show those books. This table also uses an average so you need to use a group by and since ISBN is the global key for every table, it gets the group by.

Clicking the similar to button gives you a modal like this:



I picked this book since it has a good sample size of reviews which gives me some good results for the similar to. Similar to in theory finds books that people who also liked this book good also liked. How I implemented it was finding all books that people who rated this book also voted on. I then limited the query on where the new ratings found and greater than or equal to this (book rating – 2). The system doesn’t work on books where no one has voted on so it’s not completely perfect. I don’t have the knowledge to make a proper recommendation system and this is the best I could think of.

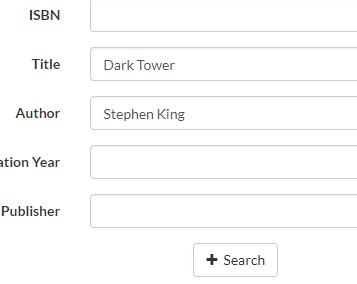
Query I used looks like:

select distinct `b`.`ISBN`, `b`.`BookTitle` as `Title`, `b`.`BookAuthor` as `Author`, `b`.`Publisher`, `b`.`ImageURLM` as `Image` from `book-ratings` as `br1` inner join `book-ratings` as `br2` on `br1`.`User-ID` = `br2`.`User-ID` inner join `books` as `b` on `b`.`ISBN` = `br2`.`ISBN` where `br1`.`ISBN` = ? and `br2`.`BookRating` >= ? and `br2`.`BookRating` > ? order by `br1`.`BookRating` desc limit 5

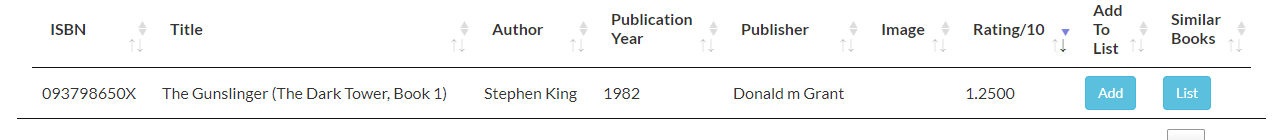
I used a self join to get all the books the person also liked and limited the book results to 5 since getting above 5 recommendations made no sense since it’s only recommendations and not demands.

Evaluation

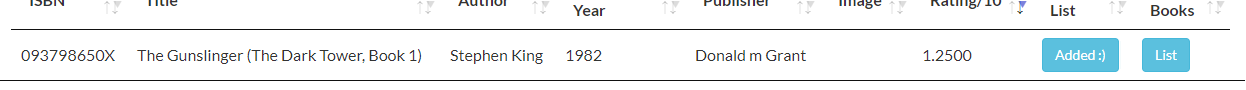
I’m going to take you on a trip of some examples that should happen. If you want to look up “The Dark Tower” by Stephen King. (The image on this book does not work)



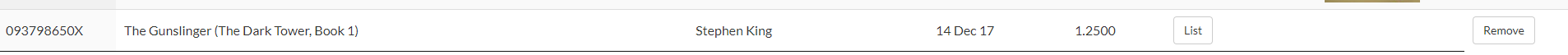
After some searching you find the book:



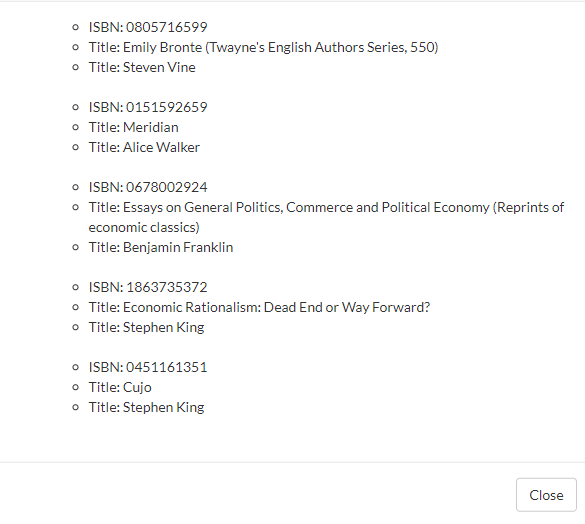
And you decide to add it to your list of books you want saved.



You find it in your inventory and start to read it.



You’ve now read it and want to find books similar to it!

You get 

Ideally it would show other Dark Tower series books but I guess it doesn’t. People who liked this book must’ve also liked books about economics or something.

Now that you’ve read the book you delete it from your inventory, never to be seen again.

The queries and results went perfectly as normal and as expected up until the similar results. I have never been able to perfectly find results that match what you should expect but I guess that’s the nature of prediction stuff. The image on this book also didn’t work since Amazon took down the links. I can’t display a “not found” as a link that doesn’t work since the link doesn’t go to a /404/ it just displays a 1x1 pixel of nothing. Another thing that doesn’t work is the Date Added. The result up top says Dec 14th due to time zones. We’re in CST but the unix time stamp I’m storing in the DB takes the time in GMT -0. I believe I can fix this by changing the timezone in php but I don’t know how to do that with Laravel.

I always tested my queries with about 2 or 3 test cases before leaving them as perfectly fine since with data this large you need multiple cases to see if anything goes wrong. Like originally if a book didn’t have a rating it would just display ‘null’ in that area and I had to set a default value for the query in case that happened. I wouldn’t have discovered that if I didn’t try out multiple cases.

Conclusion

I learned a lot from this application.

The #1 thing I gained from this application was how slow joins can be on large datasets. When I originally joined on the data-reviews for ratings the query wouldn’t even run in time as it would time out. I fixed this by adding the column I’m joining on as an index key for the new table. While that sped it up enough for the table to be able to read in a time for it to not time out, it was still incredibility slow. I originally thought that having a column as a primary key would give it the performance boost like a index key would, but it does not. A primary key only means that it is unique. Adding an index key for ISBN on all my tables made the joins down from not loading at all to loading searches in 300ms. Viewing the structure for the table on phpMyAdmin shows that the index key and primary key are all store in btrees with cardinalities in the 100000s or even 1mil for the case of ISBN. Which is relevant which since it is exactly what you talked about in class about how queries can be sped up for indexing.

I also learned some minor performance boosters. For example, Like %string% is extremely slow for the database to calculate. I was using a lot of like%% in my program and that is what was also slowing down my query by a good amount. I believe that is because like%% compared every single character. After searching I was able to figure out you can replace the like%% with fulltext trees. Fulltext trees are much faster than like%%. There is a slight disadvantage to fulltext though. A fulltext tree sacrifices accuracy for the speed it gains. I threw an example of this under the search page in implementation. The speed increase for the application sped up the query from 300ms to 50ms for searching. I was never able to get it faster than 50ms on my benchmark test.

Some other lessons learned was making sure I don’t send too much data at once. Originally my rows were max size (255) and by cutting down the size to as little as I could without breaking anything I was able slightly speed up my query by ~10%. I believe this is because it had to grab all the data from the tables and that is slow.

I also learned that you should check what type of join you’re doing. A full join on every element is extremely slow so you should check. Most of my joins are inner joins as I just want the intersections between two tables.

The way I had the tables wasn’t probably the most efficient. There are 16,550 distinct publishers, and 99,198 distinct authors. With 271,379 rows in books I could in theory assign publishers and authors into another table and assign ids to join off on. I didn’t do this because honestly the work it would require is a bit too much for me to do for a minor storage difference. There is also probably a time trade off too since it would have to join another table and that would take time compared to having it there.