# Database Implementation

We used the following DDL commands to implement our database:

-- Load Publishers by creating unique IDs for each publisher

CREATE TABLE Publishers (

  PublisherId INT PRIMARY KEY,

PublisherName VARCHAR(8192),

);

CREATE TABLE Books (

    -- ISBN10 is always 10 characters long and it's unique

    ISBN            CHAR(10)        PRIMARY KEY,

    Title           VARCHAR(4096),

    Author          VARCHAR(255),

    YearPublished   INT,

    PublisherId   INT,

FOREIGN KEY (PublisherId) REFERENCES Publisher (PublisherId)

    SELECT \* FROM Books\_RAW NATURAL JOIN Publishers

);

-- Load the Users dataset

CREATE TABLE Users (

    Username VARCHAR(32) PRIMARY KEY,

DisplayName VARCHAR(32,

PasswordHash CHAR(64) -- We use SHA512 so hashes are 512 bits = 64 B

);

-- Load the Ratings dataset

CREATE TABLE Ratings (

    Username    VARCHAR(32) ,

    ISBN        CHAR(10),

    Rating      INT,

Description VARCHAR(255)

    FOREIGN KEY (Username)  REFERENCES Users (Username),

    FOREIGN KEY (ISBN)      REFERENCES Books (ISBN),

    PRIMARY KEY (Username, ISBN),

    -- The rating must be valid

    CHECK(Rating >= 0 AND Rating <= 10)

);

CREATE TABLE Authors (

    Name VARCHAR(255) PRIMARY KEY,

    Popularity INT

);

CREATE TABLE Friends (

    WantsRecs VARCHAR(32),

    GivesRecs VARCHAR(32),

    FOREIGN KEY (WantsRecs) REFERENCES Users (Username),

    FOREIGN KEY (GivesRecs) REFERENCES Users (Username)

);

Note that since the PublisherName can be quite large (up to **8192 B = 8 KB**), we made a separate Publishers table to avoid storing it too many times.

Here is a screenshot of the main tables. These tables were primarily implemented on GCP, but some testing was also done locally.

Text

Description automatically generatedText

Description automatically generated

Text

Description automatically generated with medium confidenceText

Description automatically generated

Text

Description automatically generated

# Advanced Queries

All code for the queries can be found at the following link: <https://github.com/tommasobassetto/cs411-book-data-loader/releases/tag/stage.3>

We implemented the following queries:

* RecommendFromAuthor
* RecommendFromPublisher
* RecommendFromFriends

These queries take as input the minimum rating for a “good” book, and the user to recommend for. The output is a table with new books that meet the following criteria:

1. The author/publisher is the same as that of a “good” book, or one of your friends considered the book a “good” book.
2. You have not already read this book. (Set operations were used to enforce this, as well as subqueries).

Note that we also joined on the Publishers table to get the PublisherName for all recommendation queries. All queries also return a score to order recommendations. For Author/Publisher, this is always 1, but for RecommendFromFriends it’s the sum of all of your friends’ rating of that book. (The Friends query doesn’t add a friend’s rating if they considered it a bad book.)

Screenshot: (Note that since Users only rarely review books, we use user 10030 for all the queries, as they have reviewed a large number of books. This prevents query output from being the empty set.)

Text

Description automatically generated

We also implemented RecommendFromSimilar, which takes as input the minimum rating for a “good” book, the minimum number of common books for a “similar” user, and the user to recommend for.

This query returns the books that meet the following criteria:

1. At least one “similar” user read the book and rated it “good”.

Screenshot: Text

Description automatically generated

The final procedure is RecommendFromAll. This takes the recommendations from all of the previous methods and sets the score to be equal to that author’s popularity (or NULL if not found). This operation uses multiple INSERT INTO to combine tables followed by a LEFT OUTER JOIN with the Authors table to get the popularity.

Graphical user interface, text

Description automatically generated

# Indexing Analysis

Since the access ISBN, Username, and Rating most often in our procedures (usually in the Ratings table), we tried setting an index for each of these columns inside the Ratings table.

## EXPLAIN ANALYZE No. 1 – Index on ISBN in Ratings

Command:

Text

Description automatically generated

Run without index:

Text

Description automatically generated

Cost = 0.15s

Run with Index:

Text

Description automatically generated

Cost = 0.13s.

While this is a small speedup over no index, there are two things to consider. First, creating the index itself takes about half a second, so this would only be worth it over a very large number of queries. Second, a speedup of 0.02s on this query is too small to determine if the index scan saved any time when running the query. Therefore, we decided to not use the index.

## EXPLAIN ANALYZE No. 2 – Index on Username in Ratings

Command:

Text

Description automatically generated

Without indexing (ISBN is part of the table’s primary key):

Text

Description automatically generated

With indexing:

Text

Description automatically generated

This request already takes 0.00s and we don’t use Username in any other significant WHERE, GROUP BY, or HAVING clauses, so we decided not to use this index.

## EXPLAIN ANALYZE No. 3 – Index on Rating in Ratings

// FIXME finish this section

Overall, we found that the queries already take so little time to run on our dataset, that any potential improvement from using an index is indistinguishable from a slight hardware speedup. Therefore, we decided not to use any indices for our project.