COMP 3005 Final Project

Look Inna Book

Alex Djordjevic, Evan Maxted and Malick Sylla

COMP 3005 - Database Management Systems
Carleton University
December 11, 2022

2.1 Conceptual Design

We designed the database based off the directions given in the problem statement, as well as our understanding of existing bookstores. Companies like Chapters and Barnes and Noble served as industry examples, but unlike these two our Look-Inna-Book operates only in the online space, not brick-and-mortar. Because of this, there were many things our database did not need to model. In general, we wanted to be intentional about our data – only storing the information required for the operation of our bookstore.

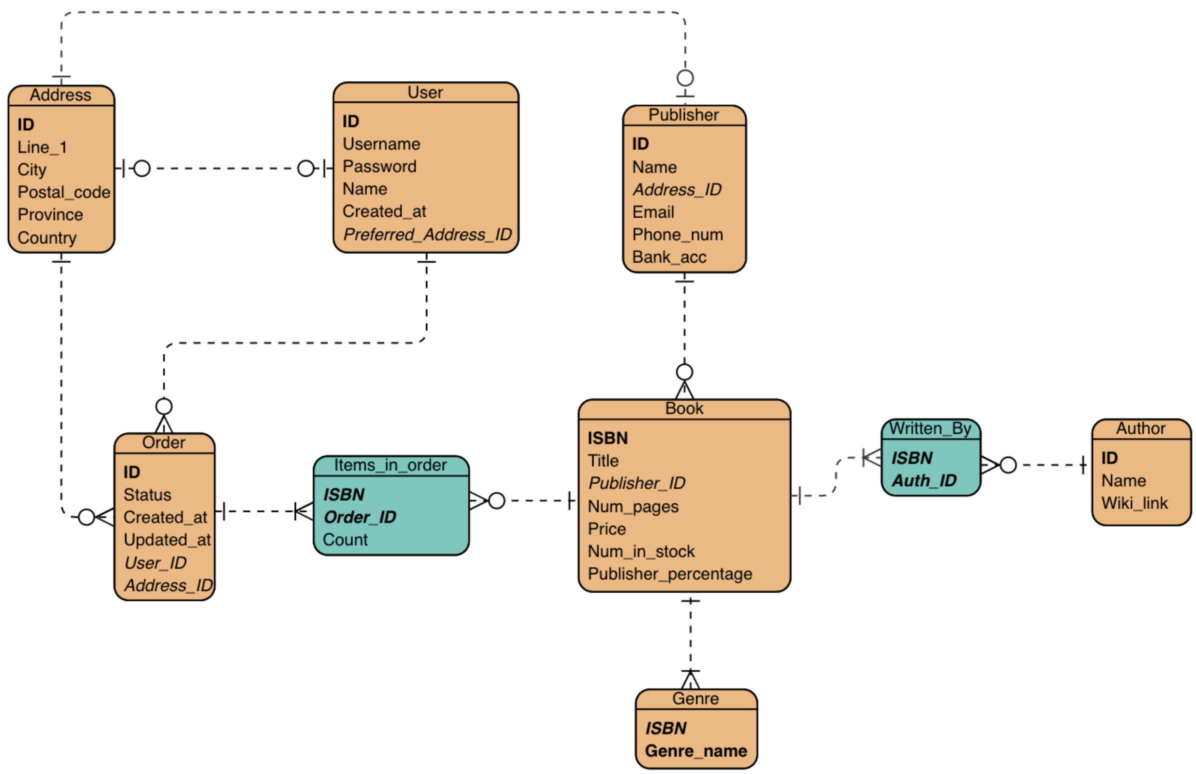


Figure 1: The ER Diagram. Blue tables represent intersection records.

2.1.1 Entity and Attribute Design

We first set out to create our entities, which would model real-world objects necessary for our business. Then, we created the sets of attributes which would define and give meaning to each of our entities. One major design decision was the inclusion of a numerical id identifying each table. This was done for efficiency in both retrieval time as well as space complexity. If our tables were to be indexed, it would be more cost-effective to store integers than it would be to store strings. If we were to scale, this space-saving technique would be beneficial, despite the cost of adding an extra attribute in cases where another candidate key exists. In addition, it is quicker to retrieve an integer from memory than it would be to retrieve a longer string, potentially stored in different sections and thus requiring multiple accesses.

Entities	Attributes
address	{id, line_1, city, country, province, postal_code} This list of attributes encompassed the set of data usually required to fulfill an online order. An address is uniquely identifiable by the combination of a postal code and a line-1, however as mentioned above we chose to include a numerical primary key to identify the records.
user	{id, username, password, name, created_at, preferred_address_id} This list of attributes contains both attributes that would be user-facing, as well as attributes that are beneficial for our business. A username allows for a user-friendly identification for each user who signs up, but having a numerical identification is cost-effective for the reasons discussed earlier. In addition, decoupling user ids and usernames allows us for flexibility – in the event that we scale, and unique usernames become more tedious to create, we can allow for duplicates whilst maintaining uniqueness because of the internal user id that backs each record. The preferred_address_id is a foreign key referencing an address – it represents that user's preferred address for order fulfillment.
publisher	{id, name, address_id, email, phone_num, bank_acc} This list of attributes comprises the set of information from a publisher that our business would need. These attributes allow for communication, identification, and payment with/of/to all of the publishers selling on our platform. Address_id is a foreign key referencing the address table and storing the physical location of the publishing company.
author	{id, name, wiki_link} The author table stores attributes required for the author's identification within our system, as well as for display to our users. We store a wiki_link, so we can fetch data from their Wikipedia page and display it to our users.
book	{isbn, title, publisher_id, num_pages, price, num_in_stock, publisher_percentage} The book entity's attributes allow us to display information to the user like title, page count and stock, which can help inform their buying decision. We also store a publisher_percentage, which is necessary for payment to the publishers whose books we sell.
genre	{isbn, genre_name} This list of attributes identifies a genre of book in our business. It was designed to allow for groupings – essentially grouping books of the same genre together, to help with discoverability and make it easier for the user to find a book they're interested in.
order	{id, status, created_at, updated_at, user_id, address_id} The order entity stores attributes required for a user to successfully purchase and receive a book. This includes a reference to the user itself, as well as the address the order should be delivered to. We will keep track of an order status to ensure every is fulfilled, and that unfulfilled orders are being monitored and troubleshooted, not left in an empty state.

2.1.2 Relationships

Participants	Cardinality	Participation Type
address/user This relationship is stored by the foreign key in user and models a user's preferred delivery address.	One-to-One	Each participant only participates partially in the relationship.
address/publisher This relationship is stored by the foreign key in publisher and models a publisher's physical location.	One-to-One	Address participates partially, but a publisher participates totally in the relationship, because a publisher must have a physical address.
address/order This relationship is stored by the foreign key in the order table, and it models the order's delivery location – i.e. the address of the user who placed it	Many-to-One	Address participates partially in the relationship, but order participates totally, since an order requires an address so it can be fulfilled.
user/order This is the most important relationship in our database – it keeps track of orders placed by a user. The relationship is stored by the foreign key in the order table.	Many-to-One	User participates partially, but order participates totally since it belongs to a user.
order/book This relationship is stored by a relationship table – items_in_order. It models the fact that an order can be comprised of many books, and vice versa.	Many-to-Many	Order participates totally in the relationship, while book participates partially. An order must contain at least one book, while a book need not be contained in any order.
book/publisher This relationship is stored by the foreign key in the book table. It models the fact that a book is published by a publisher.	Many-to-One	Publisher participates partially in the relationship, while book participates totally. This is because a book must belong to a publisher
book/author This relationship is stored in a relationship table – written_by. It models the fact that an author can write many books, and a book can be co-authored by many authors.	Many-to-Many	Author participates partially in the relationship while book participates totally. An author does not need to have written a book in our system, but a book requires an author.
book/genre This relationship is stored by the foreign key in the genre table. It represents the mapping between a particular book and a particular genre.	Many-to-One	Both participants participate totally in the relationship.

2.1.3 Intersection Tables

Table Name	Attributes
items_in_order	{<u>isbn</u>, <u>order_id</u>, count} This relationship table stores the id of both the book and order involved, as well as a count of how many copies of the book were ordered.
written_by	{<u>isbn</u>, <u>auth_id</u>} This relationship table stores the id of both the book involved and the author who wrote it.

2.2 Reduction to Relation Schemas

Relation Schemas:

- *address*(<u>id</u>, line_1, city, postal_code, province, country)
- *user*(<u>id</u>, username, password, name, created_at, preferred_address_id)
- *publisher*(<u>id</u>, name, address, email, phone_num, bank_acc)
- *book*(<u>isbn</u>, title, publisher_id, num_pages, price, num_in_stock, publisher_percentage)
- *order*(<u>id</u>, status, created_at, updated_at, user_id, address_id)
- *author*(<u>id</u>, name, wiki_link)
- *genre*(<u>isbn</u>, <u>genre_name</u>)
- *items_in_order*(<u>isbn</u>, <u>order_id</u>, count)
- *written_by*(<u>isbn</u>, <u>auth_id</u>)

2.3 Normalization of Relation Schemas

- *address*(<u>id</u>, line_1, city, postal_code, province, country)

$$\begin{aligned} F = \{ & \\ & id \rightarrow line\_1,\ postal\_code \\ & postal\_code \rightarrow city,\ country,\ province \\ & \} \end{aligned}$$

$postal\_code^{+} = \{city,\ country,\ province,\ postal\_code\}$ which is non-trivial and not a superkey so this relation violates BCNF.

Following the BCNF decomposition algorithm on the violation of $postal\_code \rightarrow city,\ country,\ province$ we can decompose this relation into:

address(<u>id</u>, line_1, postal_code), postal_address(<u>postal_code</u>, city, country, province)

$id^{+} = \{id,\ line\_1,\ postal\_code\}$ which makes id a superkey on the address relation so it is in BCNF

$postal\_code^+ = \{city,\ country,\ province,\ postal\_code\}$ which makes postal_code a superkey on the postal_address relation so it is in BCNF

Therefore, a decomposition of address(<u>id</u>, line_1, city, postal_code, province, country) on our functional dependencies F is:

address(<u>id</u>, line_1, postal_code), postal_address(<u>postal_code</u>, city, country, province)

- *user*(<u>id</u>, username, password, name, created_at, preferred_address_id)

$$
\begin{aligned}
F = \{ & \\
& username \to id,\ password,\ name,\ created\_at,\ preferred\_address\_id \\
& id \to username \qquad \}
\end{aligned}
$$

$id^+ = username^+ = \{id,\ username,\ password,\ name,\ created\_at,\ preferred\_address\_id\}$ Since id and username are superkeys, our relation is in BCNF.

- *publisher*(<u>id</u>, name, address_id, email, phone_num, bank_acc)

$$
\begin{aligned}
F = \{ & \\
& id \to email \\
& email \to id,\ address\_id,\ name,\ phone\_num,\ bank\_acc \\
& bank\_acc \to email \\
& phone\_num \to bank\_acc \\
\} &
\end{aligned}
$$

$email^+ = \{id,\ name,\ address\_id,\ email,\ phone\_num,\ bank\_acc\}$ so email is a superkey, and with transitivity we can see id, bank_acc, and phone_num are all superkeys on publisher too, so the relation is in BCNF.

- *book*(<u>isbn</u>, title, publisher_id, num_pages, price, num_in_stock, publisher_percentage)

$$
\begin{aligned}
F = \{ & \\
& isbn \to title,\ publisher\_id,\ num\_pages,\ price,\ num\_in\_stock,\ publisher\_percentage \\
\} &
\end{aligned}
$$

$isbn^+ = \{isbn,\ title,\ publisher\_id,\ num\_pages,\ price,\ num\_in\_stock,\ publisher\_percentage\}$ since isbn is a superkey on book, the relation is in BCNF

- *order*(<u>id</u>, status, created_at, updated_at, user_id, address_id)

$$
\begin{aligned}
F = \{ & \\
& id \to status,\ created\_at,\ updated\_at,\ user\_id,\ address\_id \\
\} &
\end{aligned}
$$

$id^+ = \{id,\ status,\ created\_at,\ updated\_at,\ user\_id,\ address\_id\}$ since id is a superkey on order, the relation is in BCNF

- *author*(<u>id</u>, name, wiki_link)

$$F = \{$$
$$id \rightarrow name,\ wiki\_link$$
$$\}$$

$id^+ = \{id,\ name,\ wiki\_link\}$ so id is a superkey on author so it is in BCNF

- *genre*(<u>isbn</u>, <u>genre_name</u>)

$$F = \{\}$$

since all the functional dependencies on genre are trivial it is in BCNF

- *items_in_order*(<u>isbn</u>, <u>order_id</u>, count)

$$F = \{$$
$$isbn,\ order\_id \rightarrow count$$
$$\}$$

$\{isbn,\ order\_id\}^+ = \{isbn,\ order\_id,\ count\}$ since {isbn, order_id} is a superkey on items_in_order, the relation is in BCNF

- *written_by*(<u>isbn</u>, <u>auth_id</u>)

$$F = \{\}$$

since all the functional dependencies on written_by are trivial it is in BCNF

2.4 Database Schema Diagram

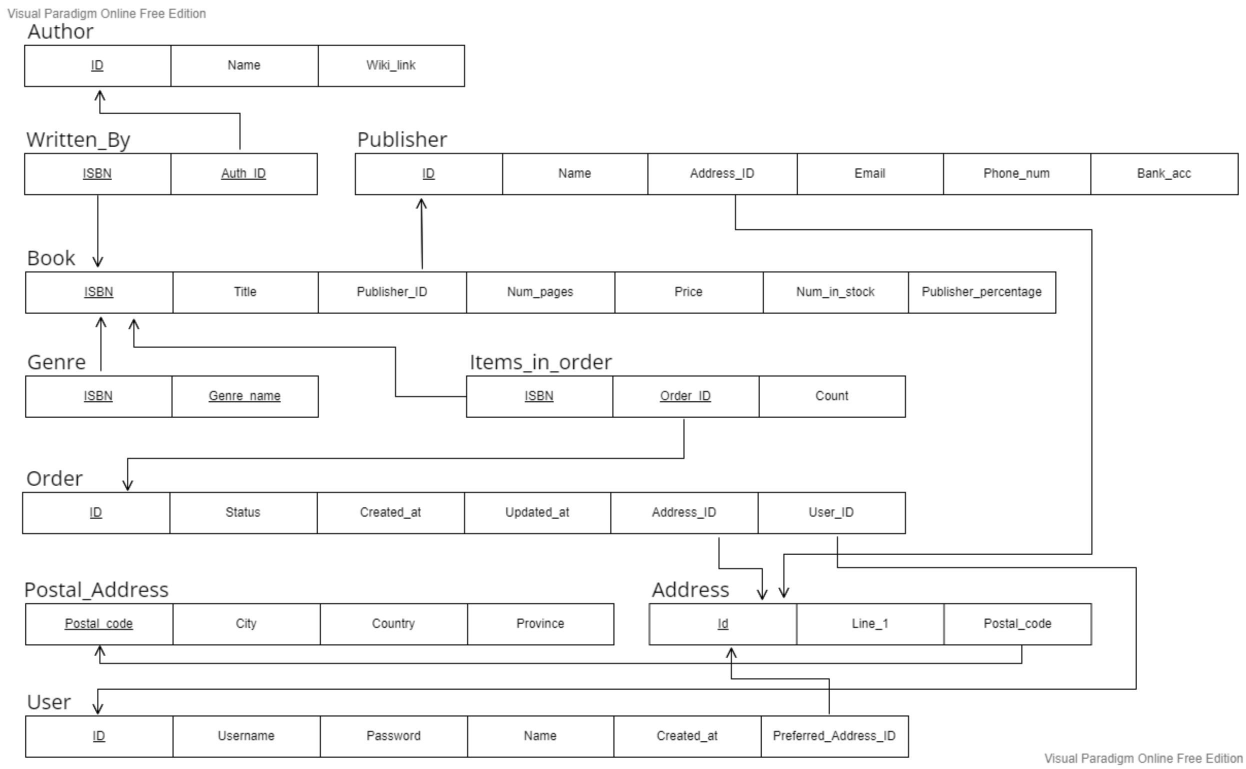
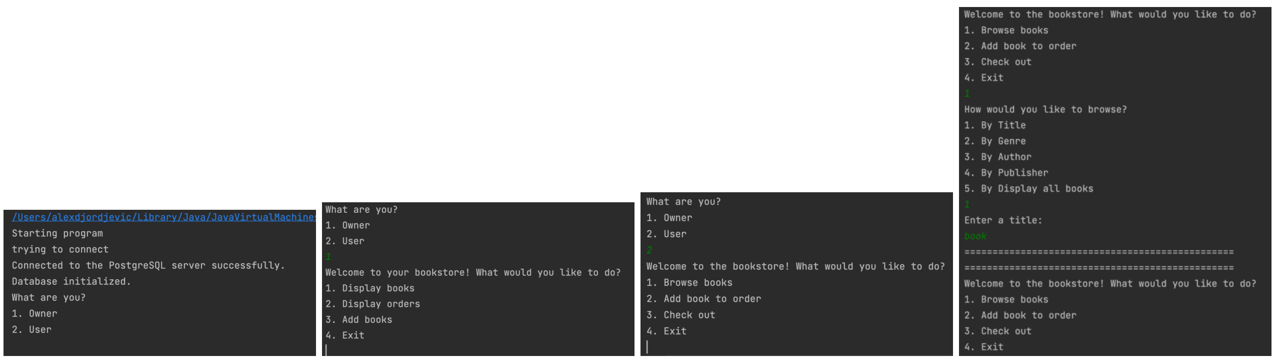


Figure 2: Database Schema Diagram

2.5 Implementation

We implemented the look-inna-book bookstore application as a command-line application with Java as the programming language and postgresql as the database. The postgresql instance runs on a server and the Java application connects directly to it. There is a Java class for each database relation in addition to a few SQL files that define the database.

The implementation architecture we followed was MVC - Model View Controller. Our Model was all the classes combined to define our database structure. Our View was the command-line where the end-user was able to interact and make decisions. Our Controller was our main function that handled the calls between the View and Model back and forth with the database.



2.6 Bonus Features

Related Searching

Our queries were built to not only match exact strings, but to also match similar strings. This was done using the sql "like" keyword. The "like" keyword defines two strings as "like" each other if one string is a substring of the other.

2.7 GitHub

https://github.com/alexatshopify/look-inna-book