Name: Alessandro Allegranzi

**Table of Contents**

[Project Direction Overview 2](#_Toc119333351)

[Use Cases and Fields 3](#_Toc119333352)

[Structural Database Rules 7](#_Toc119333353)

[Conceptual Entity-Relationship Diagram 9](#_Toc119333354)

[Full DBMS Physical ERD 11](#_Toc119333355)

[Stored Procedure Execution and Explanations 19](#_Toc119333356)

[Question Identification and Explanations 25](#_Toc119333357)

[Query Executions and Explanations 25](#_Toc119333358)

[Index Identification and Creations 29](#_Toc119333359)

[History Table Demonstration 35](#_Toc119333360)

[Data Visualizations 39](#_Toc119333361)

[Summary and Reflection 44](#_Toc119333362)

# Project Direction Overview

I plan to develop a mobile app that allows users to browse, buy, sell, and trade used books, called “BookSwap.” I am also considering “BookSwapp,” which sounds catchy. Users will be able to interact with each other, as well as dedicated used bookstores, on the app. I’ve always been a book lover, and used books especially also come with an added personality not present in new printings. E-commerce for books is very well-established at this point, even Amazon started off in the market, but it’s very much focused on new books. I would personally like an app geared more towards used books, or hard to find special editions, old comics books, etc., and believe many other potential users feel similarly.

A user based in the US might open up the app to browse the used book offerings of a bookstore in London they follow on the app that they wouldn’t normally have direct access to. They might be delighted to find a copy of a specific edition of an Italian novel’s translation they’d been searching for. They could then reach out to the store to purchase that copy. The same user might then notice their bookshelves are already full, so to make room for their new esoteric purchase they could log back into the app and list one of their older books for sale. Users can also interact directly with each other. Rather than searching a book store inventory, a user may browse the listings on the app from other users, and build a relationship with them through chatting and messaging each other within the app. Human users in the app will also be able to offer book trades instead of directly purchasing a book, which is a more personal way to share beloved literature with each other.

The BookSwap application would certainly involve a lot of mobile and web application logic that is beyond the purview of the course. In terms of the application’s database, it will need to store full user information like name, address, and billing information for the transactions that would take place in the app. I also plan to allow used bookstores to have a presence on the app, so storing their information will also be necessary. The database will also need to store a lot of book information like title, author, edition, any special notes, and price. I also plan to include messaging capability, so users and stores can chat and interact aside from direct purchase transactions.

# Use Cases and Fields

**New User Sign-up**

1. The user downloads the app from the app store.
2. The app requests the user create a log in and choose to create a bookstore or individual account.
3. The user creates an account name, selects a password, and provides basic personal information.

|  |  |  |
| --- | --- | --- |
| Field | What it stores | Why it’s needed |
| username | Name associated with the account. VARCHAR | Standard app feature to require an account name, especially to accommodate multiple accounts for the same user. |
| email | Email address. VARCHAR (need to research if postgres has a better-suited datatype) | Useful for communication with the user. |
| address\_id | Identifier for the account’s physical address. INTEGER | The app will involve billing and shipping, so a mailing address is necessary. I normalized this into a separate entity. |

Individual fields:

|  |  |  |
| --- | --- | --- |
| first\_name | Legal first name of the user. VARCHAR | Useful for displaying the user name to other users, also for correspondence from the app. |
| last\_name | Legal last name of the user. VARCHAR | Useful for displaying the user last name to other users, also for correspondence from the app. |

Bookstore Fields:

|  |  |  |
| --- | --- | --- |
| Field | What it stores | Why it’s needed |
| store\_name | Book store name associated with the account. VARCHAR | It’s necessary to connect the account to a real store. |

**Book Purchase**

1. An existing user opens the app and logs in.
2. They browse the books another user they follow as listed.
3. They select a used book they want to buy and place it in the cart.
4. They proceed to checkout and buy the book, selecting a payment type and delivery address.
5. The app sends a confirmation email for the purchase, and lists the transaction in the user’s in-app transaction history

|  |  |  |
| --- | --- | --- |
| Field | What it stores | Why it’s needed |
| purchase\_price | Price of transaction. MONEY | Useful to record price for order history and to process transactions. |
| payment\_option | The way the user paid for the book. VARCHAR | Useful to store information for credit cards for records and future transactions. |
| billing\_address\_id | Identifier for the billing address. INTEGER | Useful to process payments. |
| shipping\_address\_id | Identifier for the shipping address. INTEGER | Useful to actually send the book. |
| listing\_id | Identifier for purchased listing of book. INTEGER | It’s necessary to know which book is being purchased. |
| buyer\_id | Identifier or the user buying the book. INTEGER | It’s necessary to know who is buying the book for payment and shipping. |
| purchase\_time | Stores the time the purchase was finalized. The time the message was sent. TIMESTAMP (seems to be best datatype for postgres) | It's important to know when the transaction took place for reporting reasons and returns. |

**Book Listing**

1. An existing user opens the app and logs in.
2. They select the “list” option, and provide book details, including listing price, and photos.
3. The user hits the “list” button, and receives a confirmation email. The book appears in the user’s in-app listed books page, and is now viewable by other users.

|  |  |  |
| --- | --- | --- |
| Field | What it stores | Why it’s needed |
| title | Title of the book. VARCHAR | It’s necessary to know the title of the book for other users to see. |
| edition | Publishing edition. VARCHAR | The edition may affect the value of the book, so it’s important to display. |
| publication\_year | Year the specific copy was printed. DATE | The year of publication also could affect the book’s value. |
| image | A picture of the book. Datatype TBD, maybe blob | Seeing the visual condition of the book is also important for prospective buyers. |
| list\_price | Listed price for the book. MONEY | A list price is necessary to set the value of the book and let other users know what they need to pay to purchase it. |
| condition | Rated condition the comic is in, like mint. VARCHAR | For assessing comic book values it’s important to know the condition. |
| ISBN | integer | International Standard Book Number. It’s an important identifier for specific book products. |
| author\_id | Identifier for the book author. INTEGER | It’s necessary to know the author of the book for other users to see. I’ve normalized this field into a separate author entity. |
| first\_name | The author’s first name. varchar | It’s necessary to know the author’s name. |
| last\_name | The author’s last name. varchar | It’s necessary to know the author’s name. |
|  |  |  |

**Comic Book Listing**

1. An existing user opens the app and logs in.
2. They select the “list” option, and select the comic book button to list a comic book specifically. They input additional information like condition, issue, and artists information, which a normal book would not include.
3. The user hits the “list” button, and receives a confirmation email. The comic book appears in the user’s in-app listed books page, and is now viewable by other users.

|  |  |  |
| --- | --- | --- |
| Field | What it stores | Why it’s needed |
| issue\_no | Issue of the comic. VARCHAR | Comic issues are important. I think I will opt to go VARCHAR for this, since more information than just issue number might be important, like whether it is a collection, or info like that. |
| artist\_id | Identifier for the artist of the comic. INTEGER | It’s necessary to know who drew the comics in additon to the author. I normalized this field into a separate table. |
| first\_name | The artist’s first name. varchar | It’s necessary to know the artist’s name. |
| last\_name | The artist’s last name. varchar | It’s necessary to know the artist’s name. |
|  |  |  |

**Return**

1. A user is not happy with a purchase made on the app within 14 days of making the purchase.
2. Through the app they select a past purchase from their order history and start the return.
3. A return is created

|  |  |  |
| --- | --- | --- |
| Field | What it stores | Why it’s needed |
| purchase\_id | ID of the original purchase the user wants to return. INTEGER | It's necessary to know which purchase is being returned. |
| received \_date | When the return was was received by the original seller. TIMESTAMP | It’s important to record when the item was received by the original seller. |
| completed\_date | When the refund was fulfilled by the original seller. TIMESTAMP | It’s important to know when the return was completed and the refund made to close out the return. |

I moved away from the social media follow/like functionality for now, and instead pivoted to allowing returns for purchased books. A purchase would already have all the information pertinent to the users involved, so I’ thinking connecting the return to users directly isn’t necessary. As long as we can tie the return to a purchase, we can get all the information we need.

**Book Trade**

1. A user finds a used book they want, but they do not want to pay the listed price.
2. They message the listing user and offer a book of equal value for exchange.
3. One of the users initiates a trade proposal through the application, and if the other user agrees, the trade is enacted.
4. Both users receive confirmation emails with details of the exchange and destination mailing addresses.

|  |  |  |
| --- | --- | --- |
| Field | What it Stores | Why it’s needed |
| offer\_listing\_id | The id’s of the books being traded for each other. Similar to the above, I’m not completely certain the best way to store a trade or if it would be a join table. INTEGER | It is necessary to know which books are being traded for each other. |
| desired\_listing\_id | The id’s of the books being traded for each other. Similar to the above, I’m not completely certain the best way to store a trade or if it would be a join table. INTEGER | It is necessary to know which books are being traded for each other. |
| offered\_time | The date the trade was first offered. timestamp | It’s important to track this in case we want to implement some sort of trade deadline. |
| accepted\_time | The date the trade was accepted. timestamp | It’s important to know when the trade was accepted for tracking purposes, and to move forward with the trade. |

**Chat Between Users**

1. A user selects the chat functionality and selects a destination user.
2. The user types a message and sends it to the other user.
3. The user can then open their message chat and see past chat activity.

|  |  |  |
| --- | --- | --- |
| Field | What it Stores | Why it’s needed |
| content | The content of the message sent to another user. TEXT | It’s necessary to store the text body to display in the app’s chat history and to send to the other user. |
| sender\_id | Identifier for the user sending the message. INTEGER | It’s important to know where the message came from to display to the user receiving the message. |
| receiver\_id | Identifier for the user receiving the message. INTEGER | It’s important to know where the message is going to. |
| message\_subject | Subject line for message | Useful to allow users to enter subjects for their messages, and track that information along with text. |
| message\_datetime | The time the message was sent. TIMESTAMP (seems to be best datatype for postgres) | It’s important to track when the message was sent, also to show users. |

# Structural Database Rules

**Use Case: Account creation**

1. An account must be either an individual account or a bookstore account.

This business rule sets up specialization-generalization for accounts. Account is the super type, and individual and book store are the 2 subtypes. The relationship is totally complete and disjoint, as an account must be either an individual or bookstore account and cannot be neither.

**Use Case: Book Listing**

1. An account may make 0 to many listings; a listing belongs to only one account.
2. A listing contains only one book; a book may be listed 0 to many times.
3. A book must have one to many authors; an author may write one to many books.

I reworked the listing rule based on iteration 2 feedback (and please let me know if I’m still missing the mark). An account (either individual or book store) has a 0 to many relationship with listings. Listings must be tied to a specific account though. On the other side, a listing must contain 1 book, but a book may have 0 to many listings. For example, if the same edition of a book appears in multiple copies, or even if the same book is re-sold later.

**Use Case: Comic Book Listing**

1. A book may be a comic book, or not.
2. A comic book must have one to many artists; an artist may draw one to many comic books.

I decided to add a comic book subtype to book. Comics have some specific info to them that makes those pretty different from traditional books, like artists and issue/formal condition information. For now this relationship is partially complete. A book can still be just a book. Over time I can add more subtypes to the book type, like manuscripts, or maps, there are all sorts of different collectible book niches. For now though, I am only focusing on the one comic book subtype.

**Use Case: Messaging**

1. An individual sends 0 to many Messages. A Message involves exactly 2 individuals.

I’ve scaled back on the wider social media capabilities on the app, but I would still like to allow communication between individuals to store in a simple message table. This rule makes it clear that an individual has a 0 to many relationship with messages, and one message must be tied to two individuals, to have a sender and a receiver. I am choosing for now to leave book stores out of the messaging capabilities of the platform, and to allow this level of interaction only for users.

**Use Case: Book Purchase**

1. An account (individual or bookstore) may make many purchases. A purchase must be made by one account (individual or bookstore).
2. Every purchase must have 1 or more listings. A listing can have 0 to 1 purchases.

I framed database rule 5 as a relationship at the account level, since both individuals and book stores can be involved. Please let me know if it would instead be better to phrase this as 2 different rules for the 2 subtypes. I’m also establishing the relationship between purchases and listings rather than books directly, as I think it makes more sense. My mistake was originally trying to set up a relationship between 2 user accounts and books to purchases, but the feedback clarified I can connect listings to purchases, rather than books, which already contain the listing account’s information. A listing can have no purchases, if it has not been bought yet, or one purchase. A purchase can contain 1 or several books.

**Use Case: Book Trade**

1. An individual may make 0 to many trades. A trade must involve 2 individuals.
2. A trade must contain 2 listings. A listing can be involved in 0 to many trades.

Similar to purchases, long term I do think I can consider trades for multiple books, or uneven trades, like two-for-ones. However, for this project I’m just going to keep it simple and only allow one-for-one trades on books. These two rules explains that a trade connects two users and two listings. I made the relationship between a listing and trades 0 to many, because until the trade is finalized, the listing may be involved in multiple trade requests.

**Use Case: Return**

1. A purchase may have one return; a return must have one purchase.

The purchase relationship between two accounts would already contain much of the information necessary for a return. I think it makes more sense to build out a rule that ties the return to a specific purchase. I reworded the business rule to be more clear on the entity relationship based on feedback from iteration 2.

**Additional rules from Normalization:**

1. A purchase must have one payment option; a payment option may be used in many purchases.
2. A purchase must have one billing address; a billing address may be used in many purchases.
3. A purchase must have a shipping address; a shipping address may be used in many purchases.
4. An account must have one address; an address can be associated with multiple accounts.

**History Table Structural Rule:**

1. A listing may have many list price changes; each list price change is connected to one listing.

# Conceptual Entity-Relationship Diagram

The Structural Database Rules I have developed thus far are:

1. An account must be either an individual account or a bookstore account.
2. An account may make 0 to many listings; a listing belongs to only one account.
3. A listing contains only one book; a book may be listed 0 to many times.
4. A book may be a comic book, or not.
5. A book must have one to many authors; an author may write one to many books.
6. A comic book must have one to many artists; an artist may draw one to many comic books.
7. An individual sends 0 to many Messages. A Message involves exactly 2 individuals.
8. An account (individual or bookstore) may make many purchases. A purchase must be made by one account (individual or bookstore).
9. Every purchase must have 1 or more listings. A listing can have 0 to 1 purchases.
10. An individual may make 0 to many trades. A trade must involve 2 individuals.
11. A trade must contain 2 listings. A listing can be involved in 0 to many trades.
12. A purchase may have one return; a return must have one purchase.
13. A purchase must have one payment option; a payment option may be used in many purchases.
14. A purchase must have one billing address; a billing address may be used in many purchases.
15. A purchase must have a shipping address; a shipping address may be used in many purchases.
16. An account must have one address; an address can be associated with multiple accounts.
17. a listing may have many list price changes; each list price change is connected to one listing.

The ERD I developed from the above rules is ([LucidChart Link](https://lucid.app/lucidchart/4f70b3dd-07ee-454c-8e22-ba9c137f80af/edit?viewport_loc=-213%2C-377%2C1887%2C1162%2C0_0&invitationId=inv_043b6bbd-baed-494d-81f9-f6a0aa3feab6)):

Diagram

Description automatically generated with medium confidence

I changed a decent amount of the initial ERD for iteration 3. I reworked the original 2 different user types into subtypes under the Account supertype. The relationship is totally complete and disjoint, as an account must be either an individual or bookstore account and cannot be neither.

Both book stores and individuals can list books, so I left that relationship at the supertype Account level. A Listing must be tied to an account; the constraint is shown by the “1..1” on the diagram. Accounts, however, do not have a mandatory relationship with listings, and can have many listings, which is shown by the “0..\*”.

Similarly, both individuals and book stores can purchase books. A Purchase must be tied to 1 account and one or more listings.

Only individual accounts can message each other and trade with each other. I used “2..2” in the ERD diagram to show that mandatory relationship. Individuals, on the other hand, have an optional relationship to messages and trades, and can make 0 to many of them.

Returns have a mandatory one to one relationship to purchases, but a purchase is not necessarily tied to a return.

The last entity in the diagram is the book entity. Books may be tied to one to many listings. I have also diagrammed a comic book subtype. The relationship is not complete, as in the future I may add more subtypes, and right now a book could just be a book, without being a comic book.

After reading over the iteration 4 feedback, I further normalized the relationships and split off the shipping address, billing address, and payment info fields from the purchase entity into their own individual entities with one to many relationships with purchases. I also created an author and artist entity, related to book and comic book, respectively, which all display a many to many relationships. I also included the necessary bridging tables. I also added some structural database rules around these new entities and relationships.

As part of the last project iteration, I added a history table to track list price changes for the Listing entity.

# Full DBMS Physical ERD

The below table contains the proposed attributes for the tables in the BookSwap database. In terms of character length, I used [this document](https://webarchive.nationalarchives.gov.uk/ukgwa/+/http:/www.cabinetoffice.gov.uk/media/254290/GDS%20Catalogue%20Vol%202.pdf) for UK standards online, and decided to use that as a base:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Table | Attribute | Datatype | Reasoning | Example Data |
| Account | account\_id (primary key) | integer | I’m not expecting PK numbers that would require bigint to store the id’s. Postgres docs say integer is usually the best numeric choice for a field. Docs actually suggested I use serial, which auto increments, but given part of the requirements is the sequences I did integer. | 1 |
| Account | username | varchar(50) | The UK standards doc specifies 35 as the limit for single names, but I went a little more generous to be safe. | “Reader99” |
| Account | email | varchar(64) | Again, went a little more generous with the length than the UK standards. | “customer1@coolmail.net” |
| Account | address | varchar(255) | Also went more conservative here and gave this character count a larger cushion than I found online. | “123 Memory Lane, New York, NY, 34567, USA” |
| Account | account\_type | varchar(10) | 10 is enough characters to contain both current account types, and I don’t foresee adding more types later. | “individual” for an individual account |
| Individual | account\_id (PK + foreign key) | integer | \*Same as the above PK integer field. | 1 |
| Individual | first\_name | varchar(50) | The UK standards doc specifies 35 as he limit, but I went a little more generous to be safe. | “Harry” |
| Individual | last\_name | varchar(50) | The UK standards doc specifies 35 as he limit, but I went a little more generous to be safe. | “Styles” |
| Bookstore | account\_id (PK + foreign key) | integer | \*Same as other PK’s |  |
| Bookstore | store\_name | varchar(100) | Fairly arbitrary, but I think this is a very generous amount to contain all possible bookstores, and not take up too much DB space. | “The Friendly Book Shop” |
| Book | book\_id (primary key) | integer | \*same as other PK’s | 1 |
| Book | isbn (allow null) | bigint | ISBN’s are 10 or 13 digits, so need to go with bigint. | 1234567891 |
| Book | title | varchar(200) | Titles could theoretically get pretty long, so going with 200 as the character limit. Don’t think it’s the case to go for a text type. | “***No Matter How Much You Promise to Cook or Pay the Rent You Blew It Cauze Bill Bailey Ain’t Never Coming Home Again***  ” |
| Book | edition | varchar(100) | Went with a character type over numeric here, as not all editions are neatly numbered. | “First US printing” |
| Book | publication\_year | date | It’s a date | 2002 |
| Book | Image (allow null) | bytea | Postgres does not support BLOB type, and I read bytea is the best option to use for something like an image file. https://www.postgresql.org/docs/current/datatype-binary.html | '\xDEADBEEF' |
| Book | condition | text | I did not want to arbitrarily limit condition descriptions, so went with the text type. Postgres docs say it’s very performant on par with varchar. If space ends up being a concern I can look at other types like JSON maybe, but I think this should be OK for the use case. | “Excellent overall condition. Some scratches on the cover.” |
| Book | Book\_type | varchar(20) allow null | Right now the only possible value is comic book, but to be safe I’m using 20 characters for future subtypes. | “comic book” |
| Comic\_book | book\_id (PK + foreign key) | integer | \*as other synthetic keys | 1 |
| Comic\_book | issue\_no | integer | Just needs to count issues | 1 |
| Listing | listing\_id | Integer (primary key) | \*same as other primary keys | 1 |
| Listing | list\_price | money | Postgres has a money type for prices, and the docs recommend to use it | $25.00 |
| Listing | account\_id (foreign key) | integer | \*same as other synthetic keys | 1 |
| Listing | book\_id (foreign key) | integer | \*same as other synthetic keys | 1 |
| Listing | purchase\_id (foreign key allow null) | integer | \*same as other synthetic keys | 1 |
| Purchase | Purchase\_id (primary key) | integer | \*same as other primary keys | 1 |
| Purchase | purchase\_price | money | Postgres has a money type for prices, and the docs recommend to use it | $25.00 |
| Purchase | payment\_option\_id (foreign key) | integer | \*same as other synthetic keys | 1 |
| Purchase | billing\_address\_id (foreign key) | integer | \*same as other synthetic keys | 1 |
| Purchase | shipping\_address\_id (foreign key) | integer | \*same as other synthetic keys | 1 |
| Purchase | buyer\_id (foreign key) | integer | \*same as other synthetic keys | 1 |
| Purchase | purchase\_time | timestamp | Originally designed this to be date, but for reporting knowing the exact time of the transaction is preferable, so using a timestamp. | 1999-01-08 04:05:06 |
| Return | Return\_id | integer | \*same as other synthetic keys | 1 |
| Return | purchase\_id (foreign key) | integer | \*same as other synthetic keys | 1 |
| Return | received \_date (allow null) | date | Date type seems to fit here as we don’t need to know the exact time. | 2023-02-12 |
| Return | completed\_date (allow null) | date | Date type seems to fit here as we don’t need to know the exact time. | 2023-02-12 |
| Trade | trade\_id (primary key) | integer | \*same as other primary keys | 1 |
| Trade | offer\_listing\_id (foreign key) | integer | \*same as other synthetic keys | 1 |
| Trade | desired\_listing\_id (foreign key) | integer | \*same as other synthetic keys | 1 |
| Trade | offered\_time | timestamp | Timestamp seems to be better here as it’s very likely trades might be offered and accepted on the same day. Probably useful to have the added granularity. | 1999-01-08 04:05:06 |
| Trade | accepted\_time (allow null) | timestamp | Timestamp seems to be better here as it’s very likely trades might be offered and accepted on the same day. Probably useful to have the added granularity. | 1999-01-08 04:05:06 |
| Message | message\_id (primary key) | integer | \*same as other primary keys | 1 |
| Message | content | text | Given that messages can be fairly long, this seemed like the best data type. | “Hello, I am writing to enquire about a book you listed…” |
| Message | sender\_id (foreign key) | integer | \*same as other synthetic keys | 1 |
| Message | receiver\_id (foreign key) | integer | \*same as other synthetic keys | 1 |
| Message | subject | varchar(200) | Here I want to limit the subject at 200 characters. | “Inquiry about Sherlock Holmes volume” |
| Message | send\_time | timestamp | Using timestamp because for messages we need to store hours and minutes, as messages can go back and forth quickly. | 1999-01-08 04:05:06 |
| Book\_author | book\_id (foreign key) | integer | \*same as other synthetic keys | 1 |
| Book\_author | author\_id (foreign key) | integer | \*same as other synthetic keys | 1 |
| Author | author\_id (primary key) | integer | \*same as other primary keys | 1 |
| Author | first\_name | varchar(50) | The UK standards doc specifies 35 as he limit, but I went a little more generous to be safe. | “Harry” |
| Author | last\_name | varchar(50) | The UK standards doc specifies 35 as he limit, but I went a little more generous to be safe. | “Styles” |
| Comic\_book\_artist | comic\_book\_id (foreign key) | integer | \*same as other synthetic keys | 1 |
| Comic\_book\_artist | artist\_id (foreign key) | integer | \*same as other synthetic keys | 1 |
| Artist | artist\_id (primary key) | integer | \*same as other primary keys | 1 |
| Artist | first\_name | varchar(50) | The UK standards doc specifies 35 as the limit, but I went a little more generous to be safe. | “Harry” |
| Artist | last\_name | varchar(50) | The UK standards doc specifies 35 as the limit, but I went a little more generous to be safe. | “Styles” |
| Payment\_option | payment\_option\_id | integer | \*same as other primary keys | 1 |
| Payment\_option | Type\_name | Varchar(20) | 20 characters should be plenty for payment options like credit card or paypal. | “paypal” |
| Address | Address\_id | integer | \* same as other primary keys | 1 |
| Address | street | varchar(50) | 50 should be enough to handle street number and name | “123 Memory Lane” |
| Address | city | varchar(25) | 25 should be enough for the city. The longest city name in the US is 17 characters. | “Boston” |
| Address | state | varchar(2) | Just the standard 2 initial field. | “MA” |
| Address | zip\_code | integer | Integer should be sufficient for a standard 5 digit zipcode | “12345” |
| List\_price\_change | list\_price\_change\_id | integer | \*same as other synthetic primary keys | 1 |
| List\_price\_change | listing\_id | integer | \*Same as other foreign keys | 1 |
| List\_price\_change | old\_list\_price | money | Money is the type I’ve used in the rest of the project to store prices. | 12.50 |
| List\_price\_change | new\_list\_price | money | Money is the type I’ve used in the rest of the project to store prices. | 12.50 |
| List\_price\_change | change\_time | timestamp | In case a list price changes more than once in one day, I think timestamp is a good data type to use over date. | 1999-01-08 04:05:06 |

I feel like the above fields represent the core attributes for the entities involved in my BookSwap app. I have further normalized the initial entities and have added some tables and attributes for iteration 5. Address, Author, and Artist are now independent tables. I have also included bridging tables for books and authors, and comic books and artists, since those involve many to many relationships. The image data type I used works in theory, but in practice I’ll see if maybe just storing a url string for the image address is more feasible for a project like this.

The full Physical ERD is in Lucid Chart [here](https://lucid.app/lucidchart/719e5232-f44c-4e21-805e-21d46986106e/edit?viewport_loc=-982%2C-5%2C2225%2C1247%2C0_0&invitationId=inv_0f015035-7063-40c1-bbbf-b1af11127f4a). Image below:

Timeline

Description automatically generated with medium confidence

The attributes in the above columns are all mapped out in the new, complete physical ERD, which still contains the primary keys and foreign keys from the previous iteration.

I’ve incorporated some feedback from iteration 4, and further normalized the relationships. The Author and Artist entities are connected to books and comic books through join tables, which contain the appropriate foreign keys. I’ve also created an Address entity, with appropriate fields for a standard US address. I then placed foreign keys in the Account and Purchase entity for addresses. Because Book\_author and Comic\_book\_artist are purely bridging tables, I left out a synthetic primary key in those tables, and am opting for a composite key of the 2 foreign keys. The same book shouldn’t have the same author twice, or same artist twice, so I went with a composite key.

**References:** Looked up a lot of datatype info in the postgres docs: <https://www.postgresql.org/docs/current/datatype.html>

Here are some sample screen shots of the SQL code. I will also attach the full sql file with my submission:

A picture containing table

Description automatically generated

Text

Description automatically generated

A picture containing text

Description automatically generated

My drop table statements were producing errors in PgAdmin, so I added IF EXISTS to the commands, so it would just send a notice without failing if run and the tables didn’t exist. I also had to use CASCADE for some of the drop table commands due to the constraint dependencies on foreign keys.

# Stored Procedure Execution and Explanations

**New User Sign-up**

1. The user downloads the app from the app store.
2. The app requests the user create a log in and choose to create a bookstore or individual account.
3. The user creates an account name, selects a password, and provides basic personal information.

For this use case, I will implement a stored procedure to add a new account for an individual or bookstore.

Graphical user interface, text, application

Description automatically generated

Because I normalized addresses into their own table, and have 2 subtypes to contend with, this stored procedure had to include some conditional logic. First, it checks for the existence of the entered address info. If it finds it, then we are selecting that address key to insert into the account, and if not, we create a new Address entry. Next, we have to account for 2 different account types. Depending on the argument, we either create a new individual account, or bookstore account. I assigned default values to the first and last name, and store name arguments, since in some cases they would be blank depending on the subtype.

Screenshot of the execution:

Graphical user interface, text, application

Description automatically generated with medium confidence

**Chat Between Users**

1. A user selects the chat functionality and selects a destination user.
2. The user types a message and sends it to the other user.
3. The user can then open their message chat and see past chat activity.

Graphical user interface, text, application, email

Description automatically generated

This stored procedure was much more straightforward to implement. The only real decision involved was which time function to use to get the message send time. Reading [this](https://www.postgresql.org/docs/current/functions-datetime.html#FUNCTIONS-DATETIME-CURRENT), I opted for clock\_timestamp(). Depending on the application design, it may be better to not get the time dynamically, as there may be some lag between the message being sent, and it being saved to the DB. For right now though, I’m opting with getting the current time at time of saving the new entry.

Screenshot:

A picture containing diagram

Description automatically generated

**Book Listing**

1. An existing user opens the app and logs in.
2. They select the “list” option, and provide book details, including listing price, and photos.
3. The user hits the “list” button, and receives a confirmation email. The book appears in the user’s in-app listed books page, and is now viewable by other users.

Graphical user interface, text, application

Description automatically generated

For this use case, I created a stored procedure to add a new book entry. ISBN and images I left as optional arguments as they are nullable in the table. This requires inserting a new entry in the book table, before creating a new entry in the Listing table, due to the foreign key constraint on book\_id. I also left the author aspect out of it. Since a book could have multiple authors, I decided not to include author entry and Book\_author entry creation with this stored procedure, since the number of author fields could be variable.

Example entry:

Graphical user interface, text, application

Description automatically generated

**Comic Book Listing**

1. An existing user opens the app and logs in.
2. They select the “list” option, and select the comic book button to list a comic book specifically. They input additional information like condition, issue, and artists information, which a normal book would not include.
3. The user hits the “list” button, and receives a confirmation email. The comic book appears in the user’s in-app listed books page, and is now viewable by other users.

Graphical user interface, text, application, email

Description automatically generated

Very similar to the book listing sproc; this includes some more arguments and makes a new entry into the comic\_book table as well.

Graphical user interface, text, application

Description automatically generated

The PDF doc was sort of unclear on whether I needed to include 2 or 3 stored procedures in this document. I went extra safe with 4. \*the sql file features a few more.

**INSERTS:**

I’ll include some example screenshots, but the full list of inserts will be included in the attached sql file. The constraint on not being able to manually enter foreign keys was difficult in some cases. I got it to mostly work, but had to make some concessions to domain logic to make the data inserts reasonable and not require dozens and dozens of params and long chains of conditional logic.

Table

Description automatically generated

Text

Description automatically generated

# Question Identification and Explanations

1. Which books have been returned and what is their condition and book type?

Returns are a concern for e-commerce in general because of the overhead involved. Our case is a little different, since BookSwap does not directly sell or restock items, but our individual users and bookstores may want to know if there are any sort of patterns to returns. This way we can find the titles of returned books, see what conditions they were in, and see if comics books or traditional books are more likely to be returned.

1. Which individual users have been involved with the most message interactions?

User engagement is important in any app. This question would help us find the most active individual users in terms of messaging and direct interaction with other users.

1. Which Bookstores have sold the most in terms of dollars on site?

Bookstores on the app may want to know how much they’ve sold. Internally, it would be good to know which bookstores are performing better in order to maintain that relationship with the vendor and give other Bookstore tips on what sort of books the most successful stores list and sell.

1. What have we sold on the app in the last 24 hours?

Again, the site is meant to trade and sell old books, so we are not a traditional retailer. However, I think something like find out how many books have been sold in the last 24 hours is a useful and reasonable query. An alarmingly low number of sales might point to a technical problem going on with the app, for example.

1. How many listings belong to user X and which of them have been sold?

I added one more query to try and find some indexes. This would be very useful information for a user’s profile or history page. Gathering and showing all listings, and which of them have been sold, would be good to know for an account holder.

# Query Executions and Explanations

1. Which books have been returned and what is their condition and book type?

Graphical user interface, application

Description automatically generated

This query involves joining 4 tables to find out which books have been returned. The query starts with the Book table, and travels through Listing and Purchase to get to the return table. You can see that a few returns have been started on site. Reading through the condition and book type columns, with enough large scale data we could start to make some assessments on patterns.

1. Which individual users have been involved with the most message interactions?

Table

Description automatically generated with medium confidence

The query involves the Account supertype and Individual subtype. Even though individuals only can send messages, I opted to grab the username from the supertype as well as I thought showing the username and full name would be useful.

1. Which Bookstores have sold the most in terms of dollars on site?

Graphical user interface, application

Description automatically generated

This query features a join on at least 2 tables and an order by clause from requirement group 1 and contains at least one aggregate function from requirement group 2. I am using the SUM function to calculate the total dollar amount of sales for each bookstore.

1. What have we sold on the app in the last 24 hours?

Graphical user interface, text

Description automatically generated

This query does not match the requirements in the doc, but my original 3 queries did not lead to any indexes, so I added another useful query. By joining purchase, listing, and book, we are able to gather the books sold in the last X days. For this query, I chose 24 hours as the time interval, as I did in the question. I found the interval method online [here](https://www.postgresql.org/docs/current/functions-datetime.html#:~:text=interval%20to%20a%20date-,date,-%272001%2D09%2D28%27%20%2B%20interval):

1. How many listings belong to user X and which of them have been sold?

Graphical user interface, application

Description automatically generated

I stretched out to one more query looking for a third index candidate. Here I take information from account, book, and listing, to gather information on a single user’s listed books, and which have been sold so far. This query features a WHERE clause that we can use to target the specific account’s listings. I got the CASE WHEN expression off [WF3school](https://www.w3schools.com/sql/sql_case.asp).

# Index Identification and Creations

The primary keys that are already indexed are:

Account.acount\_id,

Address.address\_id,

Bookstore.account\_id,

Individual.account\_id,

Message.message\_id,

Trade.trade\_id,

Listing.listing\_id,

Book.book\_id,

Author.author\_id,

Book\_author.[book\_id][author\_id] \*composite PK

Comic\_book.book\_id,

Artist.artist\_id,

Comic\_book\_artist.[book\_id][artist\_id] \*composite PK

Purchase.purchase\_id,

Return.return\_id,

Payment\_Option.payment\_option\_id

List\_price\_change.list\_price\_change\_id

Foreign Keys:

|  |  |  |
| --- | --- | --- |
| Column | Unique? | Description |
| Account.address\_id | Not unique | Multiple people and accounts can share an address, so this isn’t unique. |
| Message.sender\_id | Not unique | The same user could send many messages. |
| Message.recipient\_id | Not unique | The same user could receive many messages. |
| Trade.trader\_id | Not unique | The same user could offer multiple trades |
| Trade.tradee\_id | Not unique | The same user could receive multiple trade requests. |
| Trade.offer\_listing\_id | Not unique | The same listing could be offered in more than one trade. |
| Trade.desired\_listing\_id | Not unique | The same listing could be desired in more than one trade. |
| Listing.account\_id | Not unique | Account will be able to make many listings, so this isn’t unique. |
| Listing.book\_id | Not unique | It may not be common, but it’s possible the same book could be listen multiple times. |
| Purchase.buyer\_id | Not unique | Users will be able to make many purchases. |
| Purchase.payment\_option\_id | Not unique | The payment options are not unique to any one transaction. |
| Purchase.billing\_address\_id | Not unique | Just like users, addresses will be tied to multiple purchases. |
| Purchase.shipping\_address\_id | Not unique | Just like users, addresses will be tied to multiple purchases. |
| Return.purchase\_id | Unique | This one will be unique, as one purchase will not be returned multiple times. |
| List\_price\_change.listing\_id | Not Unique | For iteration 6 I am also adding an index on the listing id foreign key in the list price change history table. This will not be unique as the same listing will be able to have many list price changes. |

Additional Keys:

Purchase.purchase\_price

Purchase.purchase\_time

Account.username

The above queries show three fairly clear indexes. On the purchase table, I will add an index on purchase\_price and purchase\_time. Both will be non unique, as they could hold the same value. Granted, for purchase time the odds are very low, but if we end up with concurrent calls or distributed systems we don’t want to set that as a unique index. Looking forward, I think indeing on purchase\_price will be especially useful and likely speed up many other queries throughout the lifetime of the app.

I will also add an index on the username column of the Account table, as that looks like it will be a common column in WHERE causes to retrieve user info. Profile page loads, for example, will be very interested in gathering info for specific users and presenting it on the app.

Index Creation:

Foreign Keys:

Graphical user interface, text, application

Description automatically generated with medium confidence

A picture containing table

Description automatically generated

Graphical user interface, text, application, email

Description automatically generated

I used the same naming convention for all foreign key indexes, which is Table\_keyname\_idx. I based that on the pdf guide doc and think it is descriptive enough. All but one of the indexes I created are not unique. Only the last script for the Return\_purchase\_id\_idx index is different because it is a unique index.

Graphical user interface, application

Description automatically generated

As a quick example here is the PgAdmin UI showing the foreign key indexes on table Message.

Additional Indexes:

Graphical user interface, text, application, email

Description automatically generated

I used the same naming conventions as I did for foreign keys for these additional indexes. Additionally, the last index on username is unique since usernames should not be repeated for multiple users.

# History Table Demonstration

I decided it would be useful to create a history table to track changes in list prices for books on the app. Tracking initial list prices, how often they change, and how long it takes to sell, are all good information points to collect for users. An interesting feature to possibly add in the future is price suggestions for new listings based on past history with similar titles, or different editions of the same book. That’s still a ways ahead, but for now tracking list price changes will do. The new structural rule for the history table is:

A listing may have many list price changes; each list price change is connected to one listing.

I have already updated the [logical](#_Conceptual_Entity-Relationship_Diag) and [physical](#_Full_DBMS_Physical) ERD’s to reflect the new history tables. I’ve also added the appropriate attributes to the table attribute list, but just to show them here they are:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| List\_price\_change | list\_price\_change\_id | integer | \*same as other synthetic primary keys | 1 |
| List\_price\_change | listing\_id | integer | \*Same as other foreign keys | 1 |
| List\_price\_change | old\_list\_price | money | Money is the type I’ve used in the rest of the project to store prices. | 12.50 |
| List\_price\_change | new\_list\_price | money | Money is the type I’ve used in the rest of the project to store prices. | 12.50 |
| List\_price\_change | change\_time | timestamp | In case a list price changes more than once in one day, I think timestamp is a good data type to use over date. | 1999-01-08 04:05:06 |

Table creation:

Graphical user interface, text, application

Description automatically generated

Sequence creation:

Graphical user interface, text, application

Description automatically generated

Trigger Creation:

Graphical user interface, text, application, email

Description automatically generated

I followed the pdf example for postgres to define the function to actually carry out the history table insertion, and the trigger that calls the function when list\_price is updated on Listing.

I selected an id from the Listing table and update the list price:

Table

Description automatically generated

Graphical user interface, text, application

Description automatically generated

I then checked the new history table to make sure everything worked correctly.

Graphical user interface, application

Description automatically generated

Tested out a couple more:

Table

Description automatically generated

You can see by the results we are now tracking list price changes in the Listing table with the new history table. We are tying the change to the appropriate Listing row with the foreign key and storing the old and new values for list\_price. We are also capturing the time of change with decent granularity using a full timestamp. Many list price changes per day won’t be a problem in reporting.

# Data Visualizations

**Payment Options:**

Given that the Bookswap App is a marketplace for used book purchases, I think gathering data around transactions and purchases is clearly going to be an area of need. Collecting data on how users pay for purchases is something I think will be very useful, so I asked the question, “what percentage of total transactions are made up by each payment option.” This information can be useful in terms of considering additional payment options, and maybe even removing options that are rarely used.

To answer the above question, I developed a query:

Graphical user interface, application

Description automatically generated

I gathered both the number and percentages of transactions that were enacted with each payment type, but because I am interested in looking at percentages of the whole, visually, I have decided to create a pie chart in EXCEL to visualize the data.

Chart, pie chart

Description automatically generated

The pie chart very clearly shows that the majority of transactions on site are carried out with credit cards. I think most e-commerce transactions are for credit cards, so that makes sense. Interestingly, PayPal ranks lower than BNPL provider Affirm. The sample size of purchases is very small (and fake), but this will be an interesting visualization to track over time. This information will be very useful during the life of the app. Supporting different payment options costs engineering effort to make sure the site can actually support them, but also directly costs money and entails agreements with credit card or other providers. Knowing which payment options are most commonly used can help better target and streamline the payment process for the app and bring savings to operations costs. It can also help inform possible, additional payment options to consider in the future.

**Purchases Over Time:**

Sticking with the purchase theme, I also decided to track purchases over the past month. This will be very useful to track and will provide very important information relating to app use and user activity patterns. Eventually being able to compare purchases month over month will also provide a lot of insight on historical user patterns. It will help direct marketing campaigns and advertising spend. From an operational standpoint, it could also point to problems with the app, if purchases for a month were to drop suspiciously and precipitously, for example.

To answer the question, I developed a query:

Graphical user interface, text, application

Description automatically generated

I found information on the date\_trunc() function [here](https://www.w3resource.com/PostgreSQL/date_trunc-function.php). It seemed like a very useful function to help group results by days. The results only have the number of purchases as a measure, so I created a column chart in Excel:

Chart

Description automatically generated

The data is fairly limited, but we can see that the 20th of February was the most active in terms of purchases. Once we are able to compare months and track larger trends, this information can help inform resource allocation for application infrastructure and drive down costs. If we would ever need to plan some sort of extended down time for maintenance, this graph would help drive the timing of those decisions, as well. It can also help drive advertisement pricing on the app (if we ever implement that) or drive our own marketing initiatives.

I thought another interesting way to measure purchases was purchases by daily hour ranges over x amount of time, in this case sticking to one month. This may reveal information about what time of the day is the most active for users in terms of purchases.

I developed a query to answer the question:

Graphical user interface, application

Description automatically generated

Grouping by hour ranges across days was an interesting problem, and I researched the extract() function [here](https://www.w3resource.com/PostgreSQL/extract-function.php#:~:text=The%20extract%20function()%20is,extracted%20from%20the%20source%20value.). The function does exactly what I needed it to do, so it ended up being a nice solution.

Similar to the monthly version, I made a column chart in EXCEL to display to visualize the data:

Chart, bar chart

Description automatically generated

Even though we don’t quite have a meaningful data set, we can definitely see some trends from the data presented in column chart form. Mainly, it looks like the early afternoon is the most active time for purchases on site. The 2 pm to 4pm range accounts for almost 65% of all purchases on site. This information can be very useful for marketing to help drive ads on the app (if we ever decide to add ads to the app). Additionally, this is extremely useful information for operations. Knowing when the most purchases happen can help us better allocate server and database resources and drive down costs. We could also use this information to better monitor site performance for issues if purchase volume were to drop below historical/expected amounts.

# Summary and Reflection

My application will allow users to browse, buy, sell, and trade used books, and will need a database to support all those activities. Buying new books online is common, but the process can also be very impersonal. With the BookSwap app, users can follow each other, search for hard-to-find books, and build a community based on a mutual interest in literature.

For the first iteration, I created the basic concept of the application, and used some user stories to plot out necessary fields for the database component tables. For the second iteration, I wrote out structural database rules that informed an ERD for my database. In iteration 3, I defined a hierarchy of supertypes and subtypes. The exercise was very helpful, as in iteration 2 one of the hardest parts for me was defining the relationships for 2 different user types. Reformatting those relationships as 2 subtypes under account made those relationships much clearer. I also mapped out primary and foreign key constraints for all the entities in my ERD, which help more clearly show those relationships. In iteration 4, I listed out all the table attributes and specified their datatypes. I then filled in attribute information to complete by physical ERD diagram. I also wrote out and tested the sql script necessary to generate all the tables and sequences for the database.

For week 5, I normalized the ERD further based on feedback, by creating new entities for addresses, payment types, artists, and authors. I also wrote out stored procedures to help populate rows in the database and inserted some data to work with. I then came up with some questions and queries to analyze the data in my database. Finally, I added indexes in foreign keys, and a few other keys, to make my database more performant. In the last iteration, I created a history table and a trigger to track listing price changes. I then created some data visualizations for interesting questions regarding my database and data around purchase activity.

The SQL script I have written has all the scripts necessary to implement my database design in Postgresql. It contains table and sequence creations scripts, some stored procedures and commands to insert dummy data into the database, and history table and trigger creations. It also includes indexes to help performance of table scans. I have also written out some queries to answer interesting questions about the BookSwap app data and to create visualizations.

Looking back on the whole project, I feel good about the database design overall. I believe it will be able to serve the application level of the BookSwap app very well moving forward. As always, I am open to feedback on any issues with my design or choices. Thank you very much to the teaching team to the support and comments so far, that have helped guide me along in creating what I think is a solid database design project.