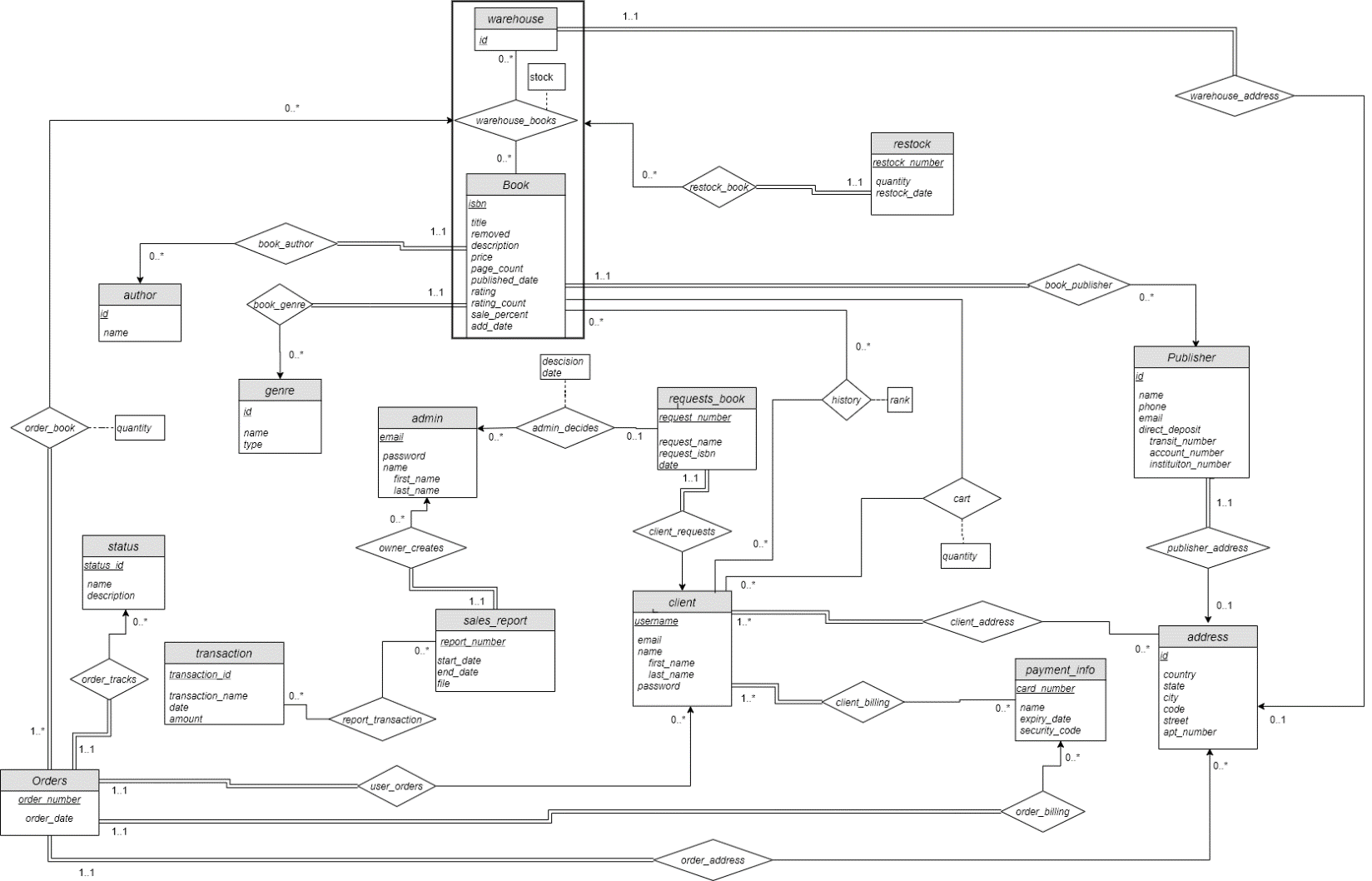
COMP 3005 Winter 2020 Project

Sharjeel Ali #101070889

March 28th, 2020

# Conceptual Design



The ER diagram that was created is shown above. For this section, an entire walkthrough that includes all explanations and assumptions about the design shown above will be explained below.

1. **Book**

The book entity contains all the relevant book information that can be derived from the primary key (isbn). These attributes are the title, description, price, page\_count, published\_date, rating, rating\_count, sale\_percent, removed and add\_date. The reason for why these attributes are all put into the single book entity is because they are all unique to the book (isbn).

Book has three main “has a” relationships, where the book derives attributes from other entities. These are the author, publisher and genre (Relationships being book\_author, book publisher and book\_genre). To make matters simpler in terms of designing the book entity, the assumption was made that each book most have an author, genre and publisher (Total participation), and contain at most one author, publisher and genre (So each book has a single genre, author and publisher). The book entity itself derives the ids of each of the three tables (Ids will be explained further below in genre and publisher sections). The publisher, author and genre all share a many to one relation (From their point of view) to the book, as an author, publisher or genre could potentially have more than one book that it has a relation with (a “fantasy” genre could have thousands of books, a publisher could publish many books, and an author could write many books).

1. **Author**

The author entity is very simplistic and contains just an id and name attribute. The reason for name not being a composite type (i.e. first\_name, last\_name) is due to it being not important. For the bookstore vision, the client or admin (owner) does not need to access only the author’s first or last name, and needs both whenever they access the author, therefore name is not a composite type. The same can be said of why the author entity does not contain any extra information, such as email, phone, address, etc. As all this information is once again irrelevant for the bookstore and would realistically be things that the publisher would know, not the store. The reason for why an id attribute exists as the primary key is that there could be books written by authors who share the same full name (Although highly unlikely). As explained above, the author has a partial participation relation with a book, where the author can write many books (many to one).

1. **Genre**

The genre entity is very similar to the author entity described above, in that they both share an id and a name. Although the bookstore would a unique genre name (i.e. fantasy, sci fi, etc.), the reason for the id attribute is to make the author, genre and publisher entities all very similar, which would be a great benefit in terms of overall efficiency when implementing the schema. This entity also contains another attribute, being type. Type in this case refers to it being fiction or nonfiction (Which as an avid book reader is important to state, as both fiction and nonfiction contain a wide plethora of genres such as history, sciences, fantasy, sci fi, etc.)

1. **Publisher**

The publisher entity is also structurally like genre and author described above, in that it contains an id as a primary key, as well as name (Being the publisher name). However, as the requirements stated, the bookstore must store details about each publisher such as name, address, email address, phone number(s) and banking account, so therefore the publisher entity is much more complex than author and genre. Each publisher entity contains a name, phone, email and direct deposit information. The phone number is a part of the publisher entity due to the assumption that a publisher can have at most one phone number, so there are no multi valued phone numbers described here. The email address is also unique to each publisher and can also be put inside the publisher entity.

As for banking account information, this was done through a realistic standpoint. Due to the bookstore/owner needing to give a percentage of each book sale to the publisher, this means the publisher receives money from the bookstore. In a real-world scenario this is done through direct deposits. The original design of the ER diagram included the direct deposit information as being a separate entity that publisher would have a total participation relation with. However, due to each direct deposit entity being tied to only a single publisher through another total participation relation, and the direct deposit only being used for publisher, this meant having a separate entity was redundant. Therefore, the direct deposit was transformed into a composite attribute and placed inside the publisher entity. Each direct\_deposit attribute contains the transit number (Unique to each bank location), intuition number(Unique to each bank, i.e. CIBC, RBC, TD), and the account number (Unique to each account within the bank location and bank itself).

The final attributes that a publisher requires is the address. This was done through creating a separate entity address. The publisher has a total participation “has an associated” relation with address, where it derives the address id from the address entity. The reason for this association type relation is due to a publisher only requiring a single address (One to many). Due to this relationship being similar to the earlier model for direct deposit, the reason for why publisher address isn’t a composite attribute in the publisher is due to the address being used by many other entities, so it would be more organized to have it be a separate entity and not composite.

1. **Client**

The client entity is the entity that contains all the user account information. In this case, this entity contains the username, email, the name and password. The reason for having both a username and password is due to scalability and personalization. A large aspect about schemas is their scalability, so having a username attribute could be useful if in the future, the bookstore allowed clients to add book reviews, etc. The name attribute is a composite attribute in that it contains both the first name and last name, which sometimes both the full name or either the first name are required (i.e. when ordering the full name could appear, however the account page could simply display the user’s first name).

Per the requirements, each client has their own checkout/cart, which is defined as the relation between book and client, and as a result, this relation stores all the books and their quantities that the client adds to their cart. Because the requirements state that the client needs to be registered to checkout, this was interpreted in this segment; The client can only add books to their carts if they’re signed in (As the cart relationship requires both isbn and client username). The client can also choose to not have any books in their cart, which means they share a partial participation with book, and the same can be said of book, in that it does not need to have a book in every client’s cart. As a result, a client can add as many books as they wish (As well as their quantities), and a book can appear in an unlimited number of client’s carts. Therefore, this is defined as a many to many relationship.

Another very similar relationship to cart that client has with book is the history relationship. This is a bonus feature that was added. Essentially, the client can view their viewing history (The books they viewed). Each book entity can be viewed by any number of clients, and so book partially participates with client. However, to conserve space, the history only stores a client’s ten most recently viewed books, therefore the client’s relationship with history is defined as at most 10 books.

The client must also contain address and billing information, this means client must have a total participation relation with both address and billing. This was made under the assumption based on the requirements that since the client inserts shipping and billing information on registration, that each client must be totally participating with these. However, the requirements also state that upon checkout, the client can choose another shipping or billing information than the one used in registration. This therefore meant that the client should be able to add an unlimited amount of shipping or billing information, and therefore the client shares a many to many relationship with both. The billing information was interpreted as payment information (i.e. card number). Both the payment information and address can belong to any client entity as a many to many relationship. This is due to the possibility of multiple clients living in the same address or using the same card numbers.

The client can also order an unlimited amount of book orders, and this is shown through the orders – client relationship through client orders relation, where a client can have a many to one relationship with orders. As another bonus feature, the client can also request a book (Described in detail below) through a book request. In this case, a client can request an unlimited number of books to be added to the bookstore, meaning this is a many to one relation (Similar to orders), in that a book request can only belong to one client.

1. **Admin**

The admin entity stores the admins or owner of the bookstore. The admin exists as a separate entity to client due to admins not being able to order books, have a cart, or book viewing history. The admin entity is also fairly like the client entity, in that both share an email, password and a composite name attribute (Formed of first\_name and last\_name). However, the admin entity does not contain a username attribute, instead using the email attribute as the primary key, contrary to the client entity. This is due to there being no need for admin scalability, as admins would not be anonymous (through usernames) and would not write book reviews (As a future addition example). Therefore, username is not needed. Each admin has the option of adding or removing a book, per the requirements. However, this is unnecessary to show as a relationship with book due to no data being shared between them (i.e. author entity shares author id with book, etc.), as well as this add/remove relationship just being an action, therefore no need to add any new relationship in this case.

The admin however still has two relationships, both being bonuses. The first being that an admin can approve a book request from a client, and therefore, by approving a book request, they can add the book into the bookstore (Once again, this is an action so its not being shown as a relationship), or they can reject the request (This is seen in admin\_decides relation, with the decision and date). Each admin entity can decide on any number of book requests, making it a many relationship (0..1), and they do not need to participate in these book request decisions, therefore it is a partial participation relationship. The admin entity also partially participates with the sales\_report entity, which is the second bonus. The admin can request a sales report to be made and can request an unlimited amount of sales reports if they so desire, therefore making the cardinality for admin none to many/unlimited sales requests.

1. **transaction**

The transaction entity is another bonus feature. This entity stores all transactions that the admin has entered. However, the transaction entity does not store a book sale transaction or sending money to the publisher transaction. This is due to both scenarios being stored within the order entity relationships themselves, so a query can be performed to extract the necessary information from them, therefore making it redundant to add book and publisher sales into the transaction entity. However, the reason for the existence of the transaction entity is for cases where the admin wishes to enter other sales or fees, such as taxes or rent, which would be included in the sales reports. The reason for why there is no relation between transaction and admin is the same reasoning used to describe the admin adding or removing books from above. The entity itself has a transaction\_id, name, date and amount. The way each transaction entity would differentiate between debit or credit transactions is through the amount (i.e. if amount< 0, it is debit, removing money from the bookstore).

1. **Sales\_Report**

The sales report entity is another bonus feature, which is used in addition to the requirement of the admin/owner accessing the book sales reports. The design will allow the owner to view a page showcasing various types of book sales and charts. However, these can be downloaded as an actual pdf report, like a Carleton audit a student can request through Carleton Central. These are then also saved to the database, for the admin to access at any time they wish. The sales report entity itself contains four attributes, being the report number, start date, end date and actual pdf file. When the admin creates a sales report, they input the start date and end date, to make it more specialized and like an actual sales report (i.e. a report showing sales from March 15th to April 5th ). The sales report also has another attribute, being the admin email, which it derives from the admin\_creates relationship, which it totally participates in, and can contain at most one admin (As only a single admin can request a report).

As for the sales report data such as the transactions and book and publisher sales, these do not need to be stored in a separate relationship for each of them (i.e. report\_transaction, report\_booksale, report\_publisherfees). This is due to the start and end date attributes. Because the transaction information is already saved in other entities (order and transaction), then all the sales report needs to do is simply view and organize the transaction data into charts by selecting transactions from specific dates (start-end dates). Therefore, it would be redundant to include these. Another reason is storage management. If for example, the admin requested sales from March 15th to April 15th, and this range included over 10,000 book sales, this would mean the database would need to replicate and store the transactions in a relationship between sales\_report and orders, and this would be very inefficient for multiple reports that span over the same days. Therefore, it is safer to simply not include a relationship between sales\_report and transaction/orders.

1. **Warehouse**

The warehouse entity was created due to future scalability for the bookstore. Per the requirements, a book needs to be shipped from a warehouse that also contains a quantity (In this case stock). So therefore, the most efficient option was creating a warehouse entity to store all the books and their stocks. The warehouse itself is defined by just a single attribute, being its id. However, it has multiple relationships with other entities. The first relationship that warehouse has is with address through warehouse\_address. Logically, a warehouse needs to have an address, and therefore warehouse totally participates with the address table, and derives the address id to include within the warehouse entity (Described in section 2). The warehouse can also have at most one address logically, and a specific address can have at most one warehouse as well, therefore this is a one to one relationship.

The second relationship that warehouse contains is warehouse\_books with books. This relationship defines each book stored in the warehouse through isbn and warehouse id, as well as the stock for each book. Because some warehouses could potentially be empty, the warehouse entity partially participates with book, but can store an unlimited number of books (Though unlikely, for obvious reasons). A book can also belong to multiple warehouses and must belong to a given warehouse as well, meaning a book totally participates with warehouse. However, the requirements state that all books are shipped (and stored) in only one warehouse. Although the design allows books to be stored in multiple warehouses, the actual implementation will only use a single warehouse to meet the requirements listed.

Finally, the warehouse and book entities need to be aggregated into a single entity for use with the order and restock entities (This is the bold outline/box around warehouse and book). Aggregation means treating a relationship (In this case warehouse\_books, and its entities warehouse and book) as a single entity, that can be used by other relations, giving the other relationships and entities access to the specific isbn and warehouse id.

1. **Orders**

The orders entity stores all orders done by the client. The entity contains just two attributes, being the order\_number, and order\_date, however it has many relationships between other entities. The most important being the client\_orders relationship. In this case, orders totally participate in this relationship, to derive the username of the client that submitted the order (As every order must be tied to a client). This is also means order can have at most one user to be tied to, and is therefore a 1..1 relationship, making client\_orders one to many (From orders point of view).

Another two important relationships that order has is order\_address and order\_billing between address and payment info. This is due to the requirement that states a client must insert shipping and billing information into the order, and as such it is assumed that the order entity would totally participate with both these entities as well. And similarly, to client\_orders relationship, order could have at most one address and payment information attached to it (Whereas payment info and address can belong to many different orders). This once again makes these two relationships one to many (From orders point of view).

Another requirement was showing tracking information. This is done via the status entity and order\_tracks relationship. The status entity itself contains the status id, the name of the status (i.e. shipped), and the description of the status. The order must participate with status as every order needs tracking information, making order a totally participating relation, and once again a one to many with status. The status id of an order would however update and change as the order progresses (i.e. shipped, out for delivery, delivered, etc.). Due to making the status and tracking simple, the order will not actually track the location (As this is different book to book), and the assumption was made that this tracking can simply be where the order is currently in the stages of ordering to delivery.

The 2nd most important relation for order to have is with book, in that each order must contain the books the client ordered as well as their quantity, which is defined as the relationship between order and the aggregated warehouse\_books described above. The set of books and quantity that the client ordered is taken directly from their cart, but this is not shown as a relationship between order\_book and cart due to the cart being dynamic and could potentially contain different books than what the order\_book contains. Each order must be totally participating with a book (i.e. each order contains at least one book), whereas the aggregated warehouse\_books does not need to participate in order (As there are certain books that are never ordered from a warehouse). In this case, order\_book would contain the order\_number, isbn, warehouse\_id and quantity ordered from the relationship. The reason for having a relationship with the aggregated warehouse\_book is simply due to making sure each book’s quantity ordered actually exists in the warehouse (i.e. It is not possible to order 10 books if only 4 exist in the warehouse\_books), therefore aggregation is needed. Per the requirements stating that to assume all books are shipped from only one warehouse, the implementation will only include just one warehouse to order books from, fulfilling the requirement, however due to scalability it is still possible for an order to contain books from different warehouses in the future. Therefore, the order\_book relation is a many to many relationship.

1. **Restock**

Per the requirements, a book must be restocked if its quantity reaches a value below a certain threshold, as well as email the publisher. The reason for restock being an entity instead of just an unlisted action is due to restock sending a publisher email, which the assumption is that the admin should be able to see when an email has been sent to a given publisher, as a record. The restock entity itself contains a restock\_number, the quantity to restock, as well as the restock date. However, it also has a “has a” total participation relationship with the aggerated warehouse\_books, where it requires the isbn and the specific warehouse id of where to restock to. This also means that the aggregated warehouse\_books doesn’t need to participate with restock.

1. **Request\_book**

Request book is the bonus feature described above. Essentially, a client can request a book to be added to the bookstore by entering either the book isbn to be added or the book title. This means both must be stored directly in the request\_book entity, as well as the request\_number and date of request. It also must have a “has a” total participation relation with client, as request\_book needs the username of the client who requested the book. And it also has a relationship with admin through admin\_decides, in that an admin can approve/reject a book request. Due to requests not being answered until an admin manually approves/rejects the book, request\_book partially participates with this relationship, but can have at most one decision being made (Meaning one to many form the request\_book side).

# 2. Reduction to Relation Schemas

Relations:

1. *author(id, name)*
2. *genre(id, name, type)*
3. *publisher(id, phone, email, address\_id, transit\_number, account\_number, institution\_number)*
4. *book(isbn, title, description, price, page\_count, published\_date, add\_date, rating, rating\_count, sale\_percent, author\_id, genre\_id, publisher\_id)*
5. *orders(order\_number, username, order\_date, status\_id, card\_number, address\_id)*
6. *status(status\_id, name, description)*
7. *order\_book(order\_number, isbn, warehouse\_id, quantity)*
8. *cart(username, isbn, quantity)*
9. *client(username, email, first\_name, last\_name, password)*
10. *admin(email, first\_name, last\_name, password)*
11. *sales\_report(report\_number, admin\_email, start\_date, end\_date, file)*
12. *report\_transaction(report\_number, transaction\_id)*
13. *transaction(transaction\_id, transaction\_name, transaction\_type, amount, date)*
14. *warehouse(id, address\_id)*
15. *warehouse\_books(warehouse\_id, isbn, stock)*
16. *address(id, country, state, city, code, street, apt\_number)*
17. *client\_address(username, address\_id)*
18. *payment\_info(card\_number, name, expiry\_date, security\_code)*
19. *client\_billing(username, card\_number)*
20. *restock(restock\_number, isbn, warehouse\_id, quantity, restock\_date)*
21. *request\_book(request\_number, username, request\_name, request\_isbn, date)*
22. *admin\_decides(email, request\_number, decision, date )*
23. *history(username, isbn, rank)*

test

# 3. Normalization of Relation Schemas

For the following section, all relations will be reduced to BCNF form. For this section, before I reduce, and normalize my relations, the following relations from section 2 above cannot be normalized any further due to them containing only two attributes, where the primary key can determine the other attribute.

Each relation that can be normalized will follow under the following three steps:

1. Check if R is in BCNF Using the simplified test (If *R* has not been decomposed yet)
   1. Find a non-trivial dependency *a 🡪 b* such that it causes a violation of BCNF
   2. Computer *a+*
   3. Verify that *a+* includes all attributes of *R*, proving it is a superkey of *R*.
2. Decompose *R* into BCNF form
3. *author(id, name)*

Functional dependencies:

*F = {*

*­id 🡪 name*

*}*

This relation is already normalized due to the following rule:

*For every set of attributes a ⊆ Ri , check that a+ (the attribute closure of a) either includes no attribute of Ri - a, or includes all attributes of R­i*

Because there is only two attributes, it cannot be normalized any further.

1. *genre(id, name, type)*

Functional dependencies:

*F = {*

*Id 🡪 name, type*

*genre\_name 🡪 id, type*

*}*

This relation is already normalized due to the following rule:

*For every set of attributes, a ⊆ Ri , check that a+ (the attribute closure of a) either includes no attribute of Ri - a, or includes all attributes of R­i*

Because there are only two attributes, it cannot be normalized any further.

1. *publisher(id, name, phone, email, address\_id, transit\_number, account\_number, institution\_number)*

Functional dependencies:

F = {

*Id🡪 name, phone, email, address\_id, dd\_id*

*phone🡪 name, Id, email, address\_id, dd\_id*

*email 🡪 name, Id, phone, address\_id, dd\_id*

*address\_id 🡪 name, Id, phone, email, dd\_id*

*dd\_id 🡪 name, Id, phone, email, address\_id*

*name 🡪 id, phone, email, address\_id, dd\_id*

}

We know that by looking at the attributes, that each of them can be used to identify the other in this case (As there only exists a single phone, email, address and direct deposit bank information for a given publisher). Therefore, each functional dependency contains a superkey. Using the following rule, we can prove it is normalized:

*For every set of attributes, a ⊆ Ri , check that a+ (the attribute closure of a) either includes no attribute of Ri - a, or includes all attributes of R­i*

Therefore, each *a+* of every *a🡪b* includes all attributes of R. Therefore, we can say publisher is normalized.

1. *book(isbn, title, description, price, page\_count, published\_date, add\_date, rating, rating\_count, sale\_percent, author\_id, genre\_id, publisher\_id)*

Functional dependencies:

*F = {*

*isbn🡪 title, description, price, page\_count, published\_date, add\_date, rating, rating\_count, sale\_percent, author\_id, genre\_name, publisher\_id*

*}*

Note: The rating and ratings\_count. These two although one would assume are dependent upon each other, and therefore not a superkey, these are not functional dependencies of book, due to multiple books having the same count but different ratings and vise versa.

Another important note is author\_id and publisher\_id. Author\_id does not determine publisher\_id. This is made under the assumption that although rare, some authors switch publishers during their writing careers, so this functional dependency cannot exist.

We know that by looking at the attributes, that the only attribute that can determine another is isbn, which determines the rest of the attributes. Using the following rule, we can prove it is normalized:

*For every set of attributes, a ⊆ Ri , check that a+ (the attribute closure of a) either includes no attribute of Ri - a, or includes all attributes of R­i*

Therefore, each *a+* of every *a🡪b* (In this case just the isbn🡪) includes all attributes of R. Therefore, we can say book is normalized.

1. *orders(order\_number, username, order\_date, status\_id, card\_number, address\_id)*

Functional dependencies:

*F = {*

*Order\_number-> username, order\_date, status\_id, card\_number, address\_id*

*}*

The important realization about this is the card\_number, address\_id and username. These three attributes do not share a functional dependency (i.e. *card\_number 🡪 username, address\_id 🡪 username*). This is under the assumption that there could be the possibility of multiple orders sharing the same address but being ordered by different users (i.e. family members or roommates). Therefore, requiring username in this relation is important and is then as a functional dependency, but *card\_number 🡪 username, address\_id 🡪 username* are not however.

We know that by looking at the attributes, that the only attribute that can determine another is order\_number, which determines the rest of the attributes. Using the following rule, we can prove it is normalized:

*For every set of attributes, a ⊆ Ri , check that a+ (the attribute closure of a) either includes no attribute of Ri - a, or includes all attributes of R­i*

Therefore, each *a+* of every *a🡪b* (In this case just the order\_number🡪) includes all attributes of R. Therefore, we can say orders is normalized.

1. *status(status\_id, name, description)*

*F = {*

*Status\_id 🡪 name, description*

*Name🡪 status\_id, description*

*Description🡪name, status\_id*

*}*

This one is simple and similar to genre normalized above. We know that by looking at the attributes, that every attribute is defined as the *a* in *a🡪b*, and that every *b* contains the rest of the attributes (*R-a*), meaning each attribute determines every other attributes. We can use the following proof to check:

*For every set of attributes, a ⊆ Ri , check that a+ (the attribute closure of a) either includes no attribute of Ri - a, or includes all attributes of R­i*

Therefore, each *a+* of every *a🡪b* includes all attributes of R. Therefore, we can say status is normalized.

1. *order\_book(order\_number, isbn, warehouse\_id, quantity)*

*F = {*

*Order\_number, isbn 🡪 warehouse\_id, quantity*

*}*

1. *cart(username, isbn, quantity)*

*F = {*

*username, isbn 🡪 quantity*

*}*

1. *client(username, email, first\_name, last\_name, password)*

*F = {*

*Username🡪 email, first\_name, last\_name, password*

*Email 🡪 username, first\_name, last\_name, password*

*}*

We know that by looking at the attributes, that there are only two functional dependencies which are username and email that determine everything else. Using the following rule, we can prove it is normalized:

*For every set of attributes, a ⊆ Ri , check that a+ (the attribute closure of a) either includes no attribute of Ri - a, or includes all attributes of R­i*

Therefore, each *a+* of every *a🡪b* includes all attributes of R. Therefore, we can say clients is normalized.

1. *admin(email, first\_name, last\_name, password)*

*F = {*

*Email 🡪 first\_name, last\_name, password*

*}*

We know that by looking at the attributes, that there are only two functional dependencies which is just email. Which determines everything else, making email a superkey. Using the following rule, we can prove it is normalized:

*For every set of attributes, a ⊆ Ri , check that a+ (the attribute closure of a) either includes no attribute of Ri - a, or includes all attributes of R­i*

Therefore, each *a+* of every *a🡪b* (In this case just the email🡪) includes all attributes of R. Therefore, we can say admin is normalized.

1. *sales\_report(report\_number, admin\_email, start\_date, end\_date, file)*

*F = {*

*Report\_number 🡪 admin\_email, start\_date, end\_date*

*}*

We know that by looking at the attributes, that there are only two functional dependencies which is just report\_number. Which determines everything else, making report\_number a superkey. Using the following rule, we can prove it is normalized:

*For every set of attributes, a ⊆ Ri , check that a+ (the attribute closure of a) either includes no attribute of Ri - a, or includes all attributes of R­i*

Therefore, each *a+* of every *a🡪b* (In this case just the report\_number 🡪) includes all attributes of R. Therefore, we can say sales\_report is normalized.

1. *report\_transaction(report\_number, transaction\_id)*

This table does not have any functional dependencies. This is due to how report\_transaction is set up. The report\_number and transaction\_id are both needed to get a unique row, meaning simply using report\_number or transaction\_id will not determine the other (As they are both many-many).

Because the relation only contains two attributes, we can use the following rule to prove it is normalized:

*For every set of attributes, a ⊆ Ri , check that a+ (the attribute closure of a) either includes no attribute of Ri - a, or includes all attributes of R­i*

This rule holds, therefore it is normalized.

1. *transaction(transaction\_id, transaction\_name, amount, date)*

*F = {*

*Transaction\_id 🡪 transaction\_name, transaction\_type, amount, date*

*}*

We know that by looking at the attributes, that there is only one functional dependency which is just transaction\_id. Which determines everything else, making transaction\_id a superkey. Using the following rule, we can prove it is normalized:

*For every set of attributes, a ⊆ Ri , check that a+ (the attribute closure of a) either includes no attribute of Ri - a, or includes all attributes of R­i*

Therefore, each *a+* of every *a🡪b* (In this case just the transaction\_id 🡪) includes all attributes of R. Therefore, we can say transaction is normalized.

1. *warehouse(id, address\_id)*

*F = {*

*id 🡪 address\_id,  
address\_id 🡪 id*

*}*

This relation is similar to publisher from above in that all attributes can determine the author (As a warehouse can have at most one address, and that address can contain just the warehouse). We can also use the following rule to prove it is normalized:

*For every set of attributes, a ⊆ Ri , check that a+ (the attribute closure of a) either includes no attribute of Ri - a, or includes all attributes of R­i*

Therefore, each *a+* of every *a🡪b* includes all attributes of R. Therefore, we can say warehouse is normalized.

1. *warehouse\_books(warehouse\_id, isbn, stock)*

*F = {*

*Warehouse\_id, isbn 🡪 stock*

*}*

1. *address(id, country, state, city, code, street, apt\_number)*

*F = {*

*Id 🡪 state, city, code, street, apt\_number,*

*}*

This relation contains a single functional dependency, which is determined by ID. However, we can further add more dependencies:

We know that code🡪 city, state.

We know

1. *client\_address(username, address\_id)*

This relation has no functional dependencies. This is because a username cannot determine an address (As a user can have multiple addresses), and an address cannot determine user (As an address can have multiple users listed).

1. *payment\_info(card\_number, name, expiry\_date, security\_code)*

*F = {*

*Card\_number 🡪 name, expiry\_date, security\_code,*

*}*

1. *client\_billing(username, card\_number)*

This relation has no functional dependencies. This is because a username cannot determine a card number (As a user can have multiple cards), and a card cannot determine user (As a card can have multiple users listed).

1. *restock(restock\_number, isbn, warehouse\_id, quantity, restock\_date)*

*F = {*

*Restock\_number, isbn, warehouse\_id 🡪 quantity, restock\_date*

*}*

1. *request\_book(request\_number, username, request\_name, request\_isbn, date)*

*F = {*

*Request\_number 🡪 username, status, request\_name, request\_isbn, date*

*}*

We know that by looking at the attributes, that there is only one functional dependency which is just *Request\_number*. Which determines everything else, making *Request\_number* a superkey. Using the following rule, we can prove it is normalized:

*For every set of attributes, a ⊆ Ri , check that a+ (the attribute closure of a) either includes no attribute of Ri - a, or includes all attributes of R­i*

Therefore, each *a+* of every *a🡪b* (In this case just the *Request\_number* 🡪) includes all attributes of R. Therefore, we can say request\_book is normalized.

1. *admin\_decides(email, request\_number, decision, date)*

*F = {*

*Email, request\_number 🡪 decision, date*

*}*

1. *history(username, isbn, rank)*

*F = {*

*username, isbn 🡪 rank*

*}*

Mention discussion with email and username and how it is normalized for admin and client. Assumption that admin emails are constant and cannot be changed, but client ones can.

# 4. Database Schema Diagram

# 5. Implementation

# 6. Bonus Features

1. **Fuzzy search**
2. **Bestsellers**
3. **Recently viewed**
4. **Requesting books**
5. **GUI Webpage**
6. **Recently released**
7. **Download PDF sales reports**
8. **Transactions**

# 7. Github Repository

<https://github.com/SharjeelAliBCS/comp3005W20-project>

# Availability

# 7. Website link

The website is hosted on Heroku as well, feel free to create an account (Just do not put personal information as the database is not secure):

<http://book-sanctum.herokuapp.com/>