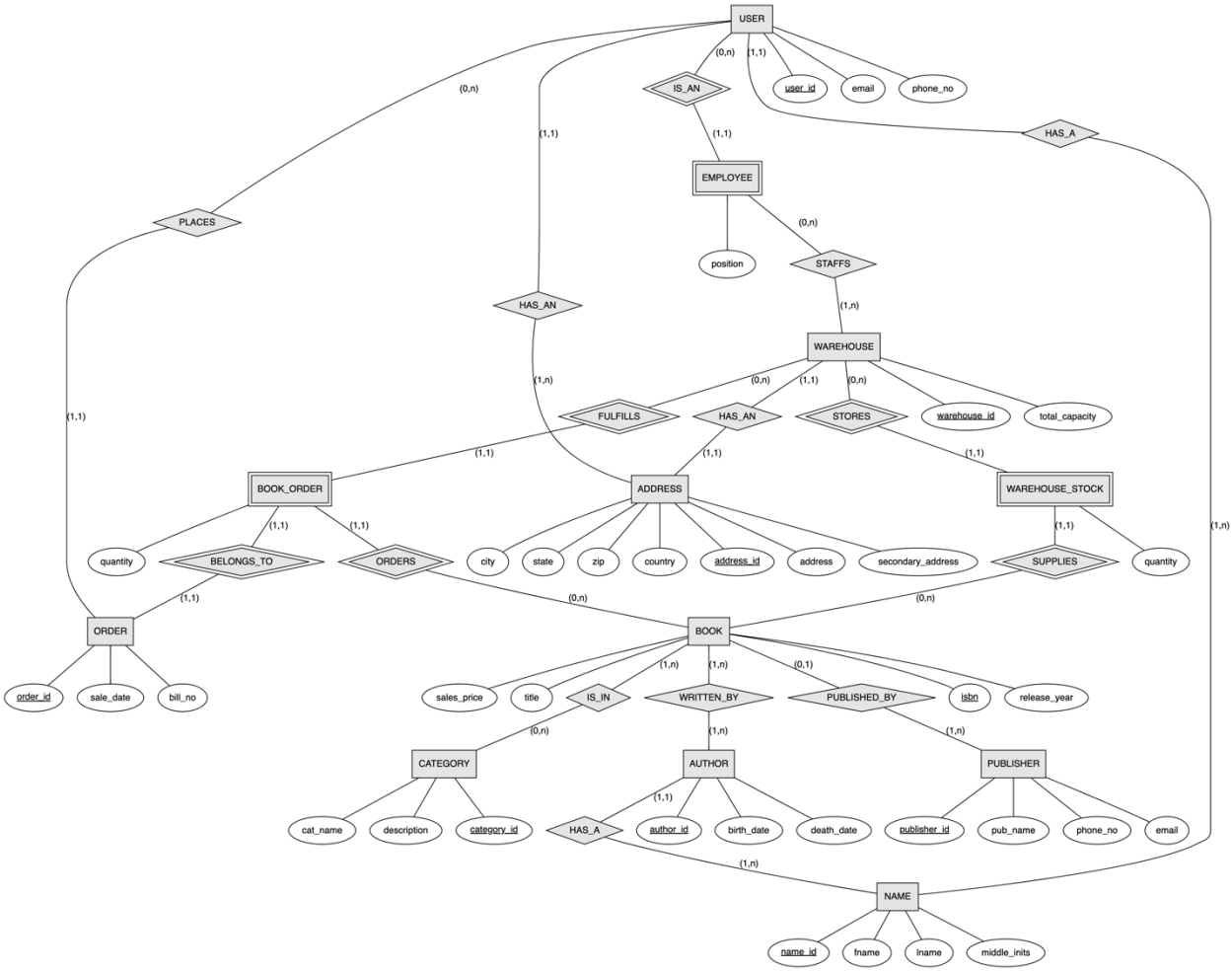
Bits & Books Database Report

By
Canaan Porter &
Alex Felderean

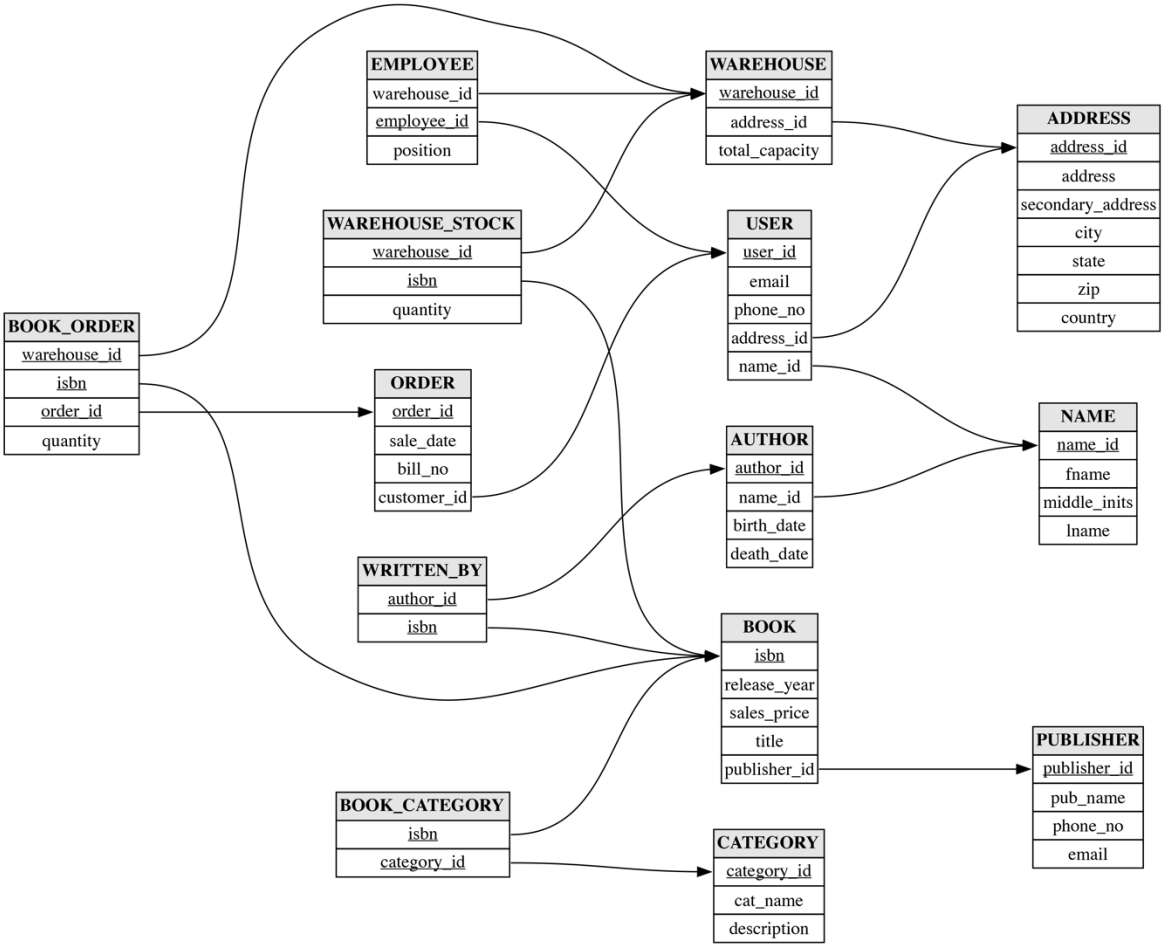
Table of Contents

1. Database Description

1.1 ER Diagram



1.2 Relational Diagram



1.3 Functional Dependencies & Normalization

Having a normalized database model with a minimal number of functional dependencies demonstrates good database practices for future scalability and long-term usability. The current database model will now be analyzed through its functional dependencies and resultant highest normal form to help reinforce its applicability into the future.

ADDRESS: For functional dependencies, the address ID attribute can determine the value of the primary and secondary addresses, city, state, zip code, and country attributes. Since address ID is a super key and determines all other attributes, with no partial or transitive dependencies, the table is BCNF normalized.

AUTHOR: For functional dependencies, the author ID attribute can determine the value of the first and last name, alongside middle initials attributes. Since author ID is a super key and determines all other attributes, with no partial or transitive dependencies, the table is BCNF normalized.

BOOK: For functional dependencies, the ISBN attribute can determine the value of the release year, sales price, title, and publisher ID attributes for each book. Since a book's ISBN is a super key and determines all other attributes, with no partial or transitive dependencies, the table is BCNF normalized.

BOOK_CATEGORY: The table has no functional dependencies, since it is a bridge table. Thus, the table is BCNF normalized.

BOOK_ORDER: For functional dependencies, the warehouse ID, ISBN, and order quantity attributes can all together determine the value of the quantity attribute. Since aforementioned determinants are a super key and determine all other attributes, with no partial or transitive dependencies, the table is BCNF normalized.

CATEGORY: For functional dependencies, the publisher ID attribute can determine the value of the name, phone number, and email attributes. Since publisher ID is a super key and determines all other attributes, with no partial or transitive dependencies, the table is BCNF normalized.

EMPLOYEE: For functional dependencies, the employee ID attribute can determine the value of the warehouse ID and position attributes. Since employee ID is a super key and determines all other attributes, with no partial or transitive dependencies, the table is BCNF normalized.

NAME: For functional dependencies, the name ID attribute can determine the value of the first name, middle initial, and last name attributes. Since name ID is a super key and determines all other attributes, with no partial or transitive dependencies, the table is BCNF normalized.

ORDER: For functional dependencies, the order ID attribute can determine the value of the sale date, total price, bill number, and customer ID attributes. Since the order ID is a super key and

determines all other attributes, with no partial or transitive dependencies, the table is BCNF normalized.

PUBLISHER: For functional dependencies, the publisher ID attribute can determine the value of the name, phone number, and email address attributes. The publisher email attribute also determines the publisher ID attribute. Thus, the table is 2NF normalized. However, since each publisher has its own completely unique set of data, requiring a new table to accommodate the email attribute would be over normalization relative to its current purpose.

USER: For functional dependencies, the user ID attribute can determine the value of the email, address ID, phone number, and name ID attributes. The email attribute value also determines the value of the user_id attribute. Thus, the table is 2NF normalized. However, since each user has its own completely unique set of data, requiring a new table to accommodate the email attribute would be over normalization relative to its current purpose.

WAREHOUSE: For functional dependencies, the warehouse ID attribute can determine the value of the address ID and total capacity attributes. Since warehouse ID is a super key and determines all other attributes, with no partial or transitive dependencies, the table is BCNF normalized.

WAREHOUSE_STOCK: For functional dependencies, the warehouse ID and ISBN attributes can together determine the value of the address ID and total capacity attributes. Since warehouse ID is a super key and determines all other attributes, with no partial or transitive dependencies, the table is BCNF normalized.

WRITTEN_BY: The table has no functional dependencies, since it is a bridge table. Thus, the table is BCNF normalized.

1.4 Views

Customer Book Orders View: This view allows a customer (where in the following SQL query, for customer with user_id = 50) to see all the orders that they have placed, with each order including the date, full book information, and quantity of each book purchased. This view is useful if the customer would like to see the history of all individual books purchased on the website, itemizing the cost of each item within an order for a thorough breakdown of their order activity.

Relational Algebra

$$\text{NET\_JOINED\_TABLE} \leftarrow \sigma_{\text{USER.user\_id = 50}} (\text{USER} \bowtie_{\text{user\_id = ORDER.customer\_id}} (\text{ORDER} \bowtie_{\text{order\_id = BOOK\_ORDER.order\_id}} (\text{BOOK} \bowtie_{\text{isbn = isbn}} \text{BOOK\_ORDER})))$$

$$\text{TOTAL\_SPENT\_TABLE} \leftarrow \rho_{(\text{TOTAL\_SPENT, isbn})} (\pi_{\text{BOOK\_ORDER.quantity * BOOK.sales\_price, BOOK.isbn}} (\text{NET\_JOINED\_TABLE}))$$

$$\text{CUSTOMER\_BOOK\_ORDERS\_VIEW} \leftarrow \pi_{\text{ORDER.order\_id, ORDER.sale\_date, BOOK\_ORDER.isbn, BOOK.title, BOOK.sales\_price, BOOK\_ORDER.quantity, TOTAL\_SPENT}} (\text{NET\_JOINED\_TABLE} \bowtie_{\text{BOOK.isbn = isbn}} \text{TOTAL\_SPENT\_TABLE})$$

```
-- SQL QUERY
CREATE VIEW CUSTOMER_BOOK_ORDERS
AS  SELECT O.order_id, O.sale_date, BO.isbn, B.title, B.sales_price,
           BO.quantity, BO.quantity * B.sales_price AS total_spent
    FROM "ORDER" O, BOOK_ORDER BO, BOOK B, USER U
    WHERE U.user_id = 50 AND U.user_id = O.customer_id AND
    O.order_id = BO.order_id AND B.isbn = BO.isbn
    ORDER BY O.sale_date DESC;
```

Resultant View Table

	order_id ▲ 1	sale_date	isbn	title	sales_price	quantity	total_spent
1	100	2012-06-15	0000385494149	Enduring Love	13.00	1	13
2	100	2012-06-15	0000609610570	Execution: The Discipline of Getti…	27.50	3	82.5
3	100	2012-06-15	0000061020648	Guards! Guards!	7.99	2	15.98
4	100	2012-06-15	0000738206679	Linked: The New Science of Networks	26.00	1	26
5	100	2012-06-15	0000684848155	UNDERWORLD: A NOVEL	16.00	2	32
6	100	2012-06-15	0001565924916	Ldap System Administration	39.95	2	79.9
7	171	2014-04-13	0000061020656	Pyramids	7.99	2	15.98
8	171	2014-04-13	0000142001740	The Secret Life of Bees	14.00	1	14
9	171	2014-04-13	0000471286168	Architecture: Form, Space, and Ord…	39.95	3	119.85000000000001
10	171	2014-04-13	0000072227710	How To Do Everything with Your Tab…	17.49	1	17.49

Customer Orders View: This view allows a customer (where in the following SQL query, for customer with user_id = 50) to see all the orders that they have placed, noting the bill number, sale date, and totals for books purchased and price respectively. This is useful for giving the customer an overlook view on their purchase history on the site without having to break down each order, like if a customer needed to check the order totals with their payment organization.

Relational Algebra

$$NET\_JOINED\_TABLE \leftarrow \sigma_{USER.user\_id = 50}(USER \bowtie_{user\_id = ORDER.customer\_id}(ORDER \bowtie_{order\_id = BOOK\_ORDER.order\_id}(BOOK \bowtie_{isbn = isbn} BOOK\_ORDER)))$$

$$TOTAL\_SPENT\_TABLE \leftarrow \rho_{(TOTAL\_SPENT, order\_id)}(BOOK\_ORDER.order\_id\ \mathbf{F}\ SUM\ BOOK\_TOTAL(\rho_{(BOOK\_TOTAL, order\_id)}(\pi_{BOOK\_ORDER.quantity * BOOK.sales\_price, BOOK\_ORDER.order\_id}(NET\_JOINED\_TABLE))))$$

$$TOTAL\_BOOKS\_PURCHASED\_TABLE \leftarrow \rho_{(quantity, order\_id, TOTAL\_BOOKS\_PURCHASED)}(isbn\ \mathbf{F}\ SUM\ quantity(\pi_{quantity, order\_id} BOOK\_ORDER))$$

$$CUSTOMER\_ORDERS\_VIEW \leftarrow \pi_{ORDER.order\_id, ORDER.sale\_date, ORDER.bill\_no, TOTAL\_BOOKS\_PURCHASED, TOTAL\_PRICE}(NET\_JOINED\_TABLE \bowtie_{BOOK\_ORDER.order\_id = order\_id}(TOTAL\_SPENT\_TABLE \bowtie_{order\_id = order\_id} TOTAL\_BOOKS\_PURCHASED\_TABLE))$$

```
-- SQL QUERY
CREATE VIEW CUSTOMER_ORDERS
AS  SELECT order_id, sale_date, bill_no, sum(quantity) as
total_books_purchased, sum(total_spent) as total_price
    FROM (SELECT O.order_id, O.sale_date, O.bill_no, B.sales_price,
        BO.quantity, BO.quantity * B.sales_price AS total_spent
        FROM "ORDER" O, BOOK_ORDER BO, BOOK B, USER U
        WHERE U.user_id = 50 AND U.user_id = O.customer_id AND
        O.order_id = BO.order_id AND B.isbn = BO.isbn)
    GROUP BY order_id
    ORDER BY sale_date DESC;
```

Resultant View Table

	order_id	sale_date	bill_no	total_books_purchased	total_price
1	479	2022-01-24	45025	17	283.75
2	259	2016-06-23	28490	13	610.8
3	171	2014-04-13	73414	22	901.98
4	100	2012-06-15	83089	11	249.38000000000002

Total Book Sales View: This view shows each book's details alongside the total copies and gross revenue produced through its time on the website. This is very useful for website management, who may want to run ads or promote these books to drive more customer interaction with the bookstore, and thus needs to know the most popular books on the site.

Relational Algebra

$$NET\_JOINED\_TABLE \leftarrow BOOK\_ORDER \bowtie_{isbn = BOOK.isbn} (BOOK \bowtie_{isbn = WRITTEN\_BY.isbn} (WRITTEN\_BY \bowtie_{author\_id = AUTHOR.author\_id} (AUTHOR \bowtie_{name\_id = name\_id} NAME)))$$

$$COPIES\_SOLD\_TABLE \leftarrow \rho_{(isbn, quantity, order\_id, COPIES\_SOLD)} (BOOK.isbn \; \mathbf{F} \; SUM\ BOOK\_ORDER.quantity (\pi_{BOOK.isbn, BOOK\_ORDER.quantity, BOOK\_ORDER.order\_id} NET\_JOINED\_TABLE))$$

$$TOTAL\_REVENUE\_TABLE \leftarrow \rho_{(isbn, TOTAL\_REVENUE)} (BOOK.isbn \; \mathbf{F} \; SUM\ ORDER\_TOTAL \rho_{(isbn, ORDER\_TOTAL)} (\pi_{BOOK.isbn, BOOK\_ORDER.quantity * BOOK.sales\_price} NET\_JOINED\_TABLE))$$

$$TOTAL\_BOOK\_SALES\_VIEW \leftarrow \pi_{BOOK.isbn, BOOK.title, NAME.fname, NAME.middle\_inits, NAME.lname, COPIES\_SOLD, TOTAL\_REVENUE} (NET\_JOINED\_TABLE \bowtie_{BOOK.isbn = isbn} (TOTAL\_REVENUE\_TABLE \bowtie_{isbn = isbn} COPIES\_SOLD\_TABLE))$$

```
-- SQL QUERY
CREATE VIEW TOTAL_BOOK_SALES
AS  SELECT B.isbn, B.title, N.fname || ' ' || N.middle_inits || '' || N.lname
as author_name,
          sum(BO.quantity) as copies_sold, sum(BO.quantity * B.sales_price)
as total_revenue
    FROM BOOK_ORDER BO, BOOK B, WRITTEN_BY WB, AUTHOR A, NAME N
    WHERE BO.isbn = B.isbn AND WB.isbn = B.isbn AND
          A.author_id = WB.author_id AND A.name_id = N.name_id
    GROUP BY B.isbn, B.title
    ORDER BY total_revenue DESC;
```

Resultant View Table

	isbn	title	author_name	copies_... 1	total_revenue
1	0000782140661	OCP: Oracle9i Certification Kit	Chip Dawes	295	30966.14999999999
2	0000517576600	Color: Natural Palettes for Painted Rooms	Donald Kaufman	248	12400
3	0000321125169	ColdFusion MX Web Application Construction Kit	Ben Forta	200	7697.999999999989
4	0000471363049	Intermediate Accounting	Donald E.Kieso	195	27144.000000000025
5	0000324107501	Advanced Accounting	Paul MarcusFischer	180	20331.000000000007
6	0000609610570	Execution: The Discipline of Getting Things Done	Larry Bossidy	177	4867.5
7	0000782140114	Access 2002 Developer's Handbook Set	Paul Litwin	159	11128.409999999985
8	0001880418568	Wolves of the Calla	Stephen King	146	5110
9	0000865659982	Composers' Houses	Gerard Gefen	144	7200
10	0000130323519	Financial Reporting and Analysis	Lawrence Revsine	141	17860.47

Author Sales View: This view shows each author registered within the database, alongside the number of copies of books sold through the platform with total gross revenue. This can be useful for website management staff who want to determine which authors are the most profitable on the platform and which authors have not performing well, which is useful in management decisions on who will continue to be supplied through the platform.

Relational Algebra

$$NET\_JOINED\_TABLE \leftarrow BOOK\_ORDER \bowtie_{isbn = BOOK.isbn} (BOOK \bowtie_{isbn = WRITTEN\_BY.isbn} (WRITTEN\_BY \bowtie_{author\_id = AUTHOR.author\_id} (AUTHOR \bowtie_{name\_id = name\_id} NAME)))$$

$$COPIES\_SOLD\_TABLE \leftarrow \rho_{(isbn, quantity, order\_id, COPIES\_SOLD)} (BOOK.isbn \; \mathbf{F} \; SUM\ BOOK\_ORDER.quantity (\pi_{BOOK.isbn, BOOK\_ORDER.quantity, BOOK\_ORDER.order\_id} NET\_JOINED\_TABLE))$$

$$TOTAL\_REVENUE\_TABLE \leftarrow \rho_{(isbn, TOTAL\_REVENUE)} (BOOK.isbn \; \mathbf{F} \; SUM\ ORDER\_TOTAL \rho_{(isbn, ORDER\_TOTAL)} (\pi_{BOOK.isbn, BOOK\_ORDER.quantity * BOOK.sales\_price} NET\_JOINED\_TABLE))$$

$$AUTHOR\_SALES\_VIEW \leftarrow \pi_{NAME.fname, NAME.middle\_inits, NAME.lname, COPIES\_SOLD, TOTAL\_REVENUE} (NET\_JOINED\_TABLE \bowtie_{BOOK.isbn = isbn} (TOTAL\_REVENUE\_TABLE \bowtie_{isbn = isbn} COPIES\_SOLD\_TABLE))$$

```
-- SQL QUERY
CREATE VIEW AUTHOR_SALES
AS  SELECT N.fname, N.middle_inits, N.lname,
           sum(BO.quantity) as copies_sold, sum(BO.quantity * B.sales_price)
as total_revenue
    FROM BOOK_ORDER BO, BOOK B, WRITTEN_BY WB, AUTHOR A, NAME N
    WHERE BO.isbn = B.isbn AND WB.isbn = B.isbn AND
          A.author_id = WB.author_id AND A.name_id = N.name_id
    GROUP BY A.author_id
    ORDER BY total_revenue DESC;
```

Resultant View Table

	fname	middle_inits	lname	copies_sold ▼ 1	total_revenue
1	Terry		Pratchett	327	2966.729999999993
2	Carl		Sagan	280	4409.789999999991
3	Steven		Pinker	243	4413.65
4	Ian		McEwan	235	3134
5	Stephen		King	225	3769.4799999999973
6	Nora		Roberts	221	4019.3999999999983
7	Don		DeLillo	215	3584.7999999999993
8	Nicholas		Sparks	209	2648.1799999999994
9	Edward	Osborne	Wilson	179	3316.499999999998
10	Ann		Patchett	168	2314.1499999999987

Warehouse Inventory View: This view shows the stock of every valid book within the database, alongside identifying details regarding the book and total inventory value held at each warehouse. This view is helpful for warehouse management staff who need to determine which books to restock or where certain stock is located within known warehouses.

NET_JOINED_TABLE ← WAREHOUSE ⋈ warehouse_id = WAREHOUSE.warehouse_id
(WAREHOUSE_STOCK ⋈ isbn = isbn BOOK)

INVENTORY_VALUE_TABLE ← ρ (INVENTORY_VALUE, warehouse_id) (π WAREHOUSE_STOCK.quantity * BOOK.sales_price, WAREHOUSE.warehouse_id (NET_JOINED_TABLE))

WAREHOUSE _INVENTORY_VIEW ← ρ (warehouse_id, isbn, title, inventory, inventory_value)
(π WAREHOUSE.warehouse_id, WAREHOUSE_STOCK.isbn, BOOK.title, WAREHOUSE_STOCK.quantity, INVENTORY_VALUE (NET_JOINED_TABLE ⋈ WAREHOUSE.warehouse_id = warehouse_id INVENTORY_VALUE_TABLE))

```
-- SQL QUERY
CREATE VIEW WAREHOUSE_INVENTORY
AS  SELECT W.warehouse_id, WS.isbn, B.title, WS.quantity as inventory,
        WS.quantity * B.sales_price as inventory_value
    FROM WAREHOUSE W, WAREHOUSE_STOCK WS, BOOK B
    WHERE W.warehouse_id = WS.warehouse_id AND B.isbn = WS.isbn;
```

Resultant View Table

	warehouse_id	isbn	title ▲ 1	inventory	inventory_value
1	1	0000394739698	A Field Guide to American Houses	36	898.1999999999999
2	2	0000394739698	A Field Guide to American Houses	108	2694.6
3	1	0001579550088	A New Kind of Science	47	2112.65
4	2	0001579550088	A New Kind of Science	113	5079.35
5	1	0000471288217	A Visual Dictionary of Architecture	16	639.2
6	2	0000471288217	A Visual Dictionary of Architecture	75	2996.25
7	1	0000446608955	A Walk to Remember	14	97.86
8	2	0000446608955	A Walk to Remember	48	335.52
9	1	0000072222743	A+ Certification All-in-One Exam Guide	146	8758.54
10	2	0000072222743	A+ Certification All-in-One Exam Guide	105	6298.95

1.5 Indexes

ADDRESS: The attributes address_id and state were indexed. The address_id attribute was indexed because it is the table's primary key, so join operations benefit from having efficient access to the attribute. Since address will be traversed frequently in relation to both user orders and company warehouses, indexing the address helps reduce computational load for these queries. The state attribute is also indexed because it is commonly searched for with queries. The state of warehouses or user addresses will likely be searched by location. During online transactions, state will frequently be called for computing taxes, which would benefit from efficient indexing of state. Likewise, state will also be queried for shipping proximity between user and warehouse, being another crucial area where indexing could help reduce computation load.

AUTHOR: The attributes author_id and birth_date were indexed. The author_id attribute was indexed because it is the table's primary key, so join operations benefit from having efficient access to the attribute. Since the author will be queried frequently in relation to their associated books and publishers, indexing the author helps reduce computational load for these queries. The birth_date attribute is also indexed because it will likely be commonly searched for with queries. Users will likely look for authors by generation, so queries searching for authors born on a specific year or decade reflect a need for indexing on both direct and ranged values.

BOOK: The attributes isbn, release_year, sales_price, and publisher_id were indexed. The publisher_id and isbn attributes are indexed because they are foreign and primary keys respectively, so join operations benefit from having efficient access to these attributes. Since book will be traversed frequently while browsing the website, placing orders, and viewing relations to other parts of the database, these indexes help reduce computational load for these queries. The sales_price and release_year attributes are also indexed because they are commonly searched for with queries. Books will likely be searched in relation to price either by rising price or within a range, like "find books that cost between than $25 and $35". This is equally applicable to queries in relation to the year the book was written, like "find all books written between 2001-2007". Organizing these values through indexing would help these types of queries, especially as the database grows.

BOOK_CATEGORY: The attributes isbn and category_id were indexed. The isbn and category_id attributes were indexed because they are the table's foreign key, so join operations benefit from having efficient access to the attribute. Since the book and book category will be queried frequently together, indexing the category IDs helps reduce computational load for these queries. Users will likely search for books within a specific category, like science fiction or nonfiction.

BOOK_ORDER: The attributes warehouse_id, isbn, and order_id were indexed. The warehouse_id, isbn, and order_id attributes were all indexed because they are all foreign or primary keys to the table, so join operations benefit from having efficient access to the attribute. Since all of these attributes will be frequently accessed during order creation and history retrieval, representing a major part of the website's service, it is important that the computational cost to running these queries is efficientized.

CATEGORY: No indexing was written for this category due to it containing only 9 tuples.

EMPLOYEE: The attributes employee_id, warehouse_id, and position were indexed. The employee_id and warehouse_id attributes were indexed because they are primary and foreign keys to the table respectively, so join operations benefit from having efficient access to the attribute. The position attribute was also indexed since it is likely queries involving employee rank will be frequently called. For instance, management may want to retrieve and send emails to employees within a specific position or set of positions. Thus, these positions can be queried respective to their use within the company to help retrieve employees by rank more efficiently.

NAME: The attributes name_id and lname were indexed. The name_id attribute was indexed because it is the table's primary key, so join operations benefit from having efficient access to the attribute. Since the name will be queried frequently in relation to users, authors, and employees of the company, indexing the name_id helps reduce computational load for these queries. The last name attribute is also indexed because it will likely be commonly searched for with queries, particularly in relation to author. Users often look for authors by last name when searching, similar to within library catalogs, so indexing the last names in alphabetical order, for instance, would greatly optimize the computational load required to search through authors.

ORDER: The attributes order_id and sale_date, and customer_id were indexed. The order_id and customer_id attributes were indexed because they are the table's primary and foreign keys respectively, so join operations benefit from having efficient access to the attribute. Since both these attributes will be queried frequently in relation to identifying online orders and customers they belong to, like during purchase history searches, indexing these attributes helps reduce computational load for these queries. The sale date attribute is also indexed because it will likely be commonly searched for with queries, particularly when users are looking at transaction history. Users will likely review their purchase on a particular day or see their transactions within a longer time span, so indexing these dates in numerical order, for instance, would greatly optimize the computational load required to search through purchase history on the site.

PUBLISHER: The attributes publisher_id and pub_name were indexed. The publisher_id attribute was indexed because it is the table's primary key, so join operations benefit from having efficient access to the attribute. Since the publishers will be queried frequently in relation to the authors they manage, indexing the publisher_id helps reduce computational load for these queries. The publisher's name attribute is also indexed because it will be commonly queried for in relation to books. Users will likely search for a particular publisher by name to query books published by the company, so indexing the publisher names in alphabetical order, for instance, would greatly optimize the computational load required to retrieve the correct publisher and pull relevant information from.

USER: The user table has indexes on the attributes user_id, name_id, address_id, email, and phone_no. Since the primary key of user may be used to join with another table, indexing the user_id attribute is a good way to improve performance. The name_id index is useful to join with

the name table, as it is a foreign key. This makes it faster to search by name through the bookstore. The address_id index is useful to join users with their connected address. This index can help with efficiency when querying users based on location, which can help in querying statistics on making the real-world from warehouse to user addresses. The index on email is useful when management may want to send marketing or order information to many customers based on website metrics. The index on phone_no makes it more efficient to search by area code, as well as similar benefits to the aforementioned email indexing.

WAREHOUSE: For the warehouse table, the attributes warehouse_id, address_id, and total_capacity are indexed. Since the primary key of warehouse may be called to join with another table, indexing the warehouse_id attribute is a good way to improve performance. The address_id index is useful since address_id is a foreign key. This allows for the address of the warehouse to be joined with the warehouse more efficiently. The total_capactiy index is useful when querying warehouses based on their ability to hold stock, which can help optimize distribution strategies based on book popularity and cost efficiency alongside other stats.

WAREHOUSE_STOCK: The warehouse_stock table has indexes on the isbn, warehouse_id and quantity attributes. The isbn index improves performance on searches, since books are likely to be queried in relation to the warehouse stocks to determine factors like shipping time, and thus knowing which warehouses contain which stock becomes very important for operation. Since the stock may frequently be looked at joined with the warehouse that operates it, indexing the warehouse_id attribute is a good way to improve performance. The index on quantity is useful for searching entries in relation to stock numbers, which will likely be used when determining if stock is low, what books to put on sale to empty stock, etc.

WRITTEN_BY: The written_by table has indexes on the isbn and author_id attributes. These attributes are foreign keys to book and author respectively; they are used to join together the many to many relationship between author and book. Since they are so commonly used in joins, it improves the efficiency of those joins to have indexes on both attributes.

1.6 Sample Transactions

1. A new user signs up to the website.

A useful and frequent transaction that will be conducted with the website will be when a new user is added to the database. This entire transaction works to populate tables and match accordingly with a new website user. It is a series of reads and writes that populates the ADDRESS, NAME, and USER tables accordingly based on inputted and existing information. The strings within the insert queries represent example input data that is representative of the type of data that belongs in each field. For the unique ID values, the maximum value was taken and incremented by one to ensure that the generated IDs were unique. This was to ensure that, if entries were ever removed from the website (ex. A user deleting their account), there is no chance that an ID number would be generated that conflicted from the remaining list. Overall, this properly represents the action of commands that would create a new user for the website, initializing all proper information to interact with the service.

```
-- Get the address registered first
INSERT INTO ADDRESS
VALUES ((SELECT max(ADDRESS.address_id) FROM ADDRESS) + 1, '4321 W Woodruff
Ave', 'Apt. 808', 'Columbus', 'OH', '43201', 'United States of America');
-- Then get the name registered
INSERT INTO NAME
VALUES ((SELECT max(NAME.name_id) FROM NAME) + 1, 'Alexander', 'M',
'Felderean');
-- Use the address and name IDs to then register the user
INSERT INTO USER
VALUES ((SELECT max(USER.user_id) FROM USER) + 1, (SELECT max(NAME.name_id)
FROM NAME), 'felderean.1@osu.edu', '6140000000', (SELECT
max(ADDRESS.address_id) FROM ADDRESS));
```

2. A new author releases their first book on the website

Another frequent transaction that will be conducted with the website will be a new book is added to the catalog by a new author. This is especially useful for a new website, which will have many new authors providing their services as the website grows in popularity. The series of writes depicted below populate tables accordingly based on established database dependencies, working up from tail ends of the database structure. For instance, Author had to come after the initialization of the name, since its reference depends on that entry. The strings within the insert queries represent example input data that is representative of the type of data that belongs in each field. For the unique ID values, the maximum value was taken and incremented by one to ensure that the generated IDs were unique. This was to ensure that, if entries were ever removed from the website (ex. An author discontinuing their services on the website), there is no chance that an ID number would be generated that conflicted from the remaining list. Overall, this properly represents the action of commands that would create a new author ready to sell a book for the website, initializing all proper information to interact with the service.

```
-- Get the name registered for the author first
INSERT INTO NAME
VALUES ((SELECT max(NAME.name_id) FROM NAME) + 1, 'Michael', 'C.',
'Townley');
-- Then register the author
INSERT INTO AUTHOR
VALUES ((SELECT max(AUTHOR.author_id) FROM AUTHOR) + 1, (SELECT
max(NAME.name_id) FROM NAME), '1965-06-24', '2013-09-17');
-- Then insert the book he wrote (the publisher is already established)
INSERT INTO BOOK
VALUES ('1725177362694', 2012, 59.99, 'The DeSanta Methods', (SELECT
PUBLISHER.publisher_id FROM PUBLISHER WHERE pub_name = 'Crown Pub'));
-- We can now link the book with the author
INSERT INTO WRITTEN_BY
VALUES ((SELECT max(AUTHOR.author_id) FROM AUTHOR), '1725177362694');
-- And assign the book within a category
INSERT INTO BOOK_CATEGORY
VALUES ('1725177362694', (SELECT CATEGORY.category_id FROM CATEGORY WHERE
cat_name = 'Literature & Fiction'));
```

3. An employee joins the company

Another frequent transaction that will be conducted with the website involves managing new employee staff. This is a necessary transaction to keep, as the company's staff will grow over time, and as such there needs to be a way to efficiently add workers to the database quickly. The series of writes depicted below populate tables accordingly based on established database dependencies, working up from tail ends of the database structure. For instance, employee had to come after the initialization of the address, since its reference depends on that entry. The strings within the insert queries represent example input data that is representative of the type of data that belongs in each field. For the unique ID values, the maximum value was taken and incremented by one to ensure that the generated IDs were unique. This was to ensure that, if entries were ever removed from the website, there is no chance that an ID number would be generated that conflicted from the remaining list. For instance, if an employee quits, we want to make sure new employees don't end up overlapping with concurrent employee IDs. Overall, this properly represents the action of commands that would create a new employee ready to be assigned to a warehouse for work, initializing all proper information to interact with the service.

```
-- Due to dependencies, get the address registered first
INSERT INTO ADDRESS
VALUES ((SELECT max(ADDRESS.address_id) FROM ADDRESS) + 1, '1425 E Leave St',
NULL, 'San Diego', 'CA', '15243', 'United States of America');
-- Then get the name registered
INSERT INTO NAME
VALUES ((SELECT max(NAME.name_id) FROM NAME) + 1, 'Imp', 'O.', 'Stern');
-- Create a new user for the employee
INSERT INTO USER
VALUES ((SELECT max(USER.user_id) FROM USER) + 1, (SELECT max(NAME.name_id)
FROM NAME), 'amon.g@postal.us', '8195231523', (SELECT max(ADDRESS.address_id)
FROM ADDRESS));
-- Finally, create the new employee
INSERT INTO EMPLOYEE
VALUES ((SELECT max(USER.user_id) FROM USER), 2, 'Logistics');
```

2. User Manual

The following section will explain how SQL can be used to search, add, modify, and delete entries from the Bits & Books database. It will start by describing the real-world entity represented by each relation in the database. This will include explanations of each attribute in the relation, as well as the domain of each attribute and any restrictions on the values they contain. After that, it will give a list of sample queries that can be executed on the database, along with up to the first 5 resulting tuples.

2.1 Real-World Entity Descriptions

ADDRESS: The ADDRESS relation contains all the information needed to find the location of a building, suite, apartment, PO box, etc. Each ADDRESS relation contains a unique, arbitrary integer as its primary key, address_id. This is referenced by the WAREHOUSE and USER relations. The address attribute represents the primary street address of the location and has a domain of string up to 100 characters. The secondary_address represents the specific apartment, suite, or room if a building is shared between multiple entities and is stored as a string of up to 50 characters. The city attribute represents the city of the location and is stored as a string of up to 30 characters. The state attribute is used to contain the 2-letter state or province code of the location, with a domain of 2-character strings. The zip attribute represents either the 5- or 10-digit zip/postal code of the location, stored as a string of up to 10 characters. Finally, the last attribute is the country, which contains a 3-digit country code. The only values these should contain are USA and CAN, as B&B only ships to those two countries. Each of these attributes must be not null except for secondary address, as not all locations will have an apartment or suite number.

AUTHOR: The AUTHOR relation represents an author that has written one or more book in the Bits & Books database. The primary key for this relation is the author_id, a unique, arbitrary integer assigned to each author. The name_id attribute is a foreign key that references the NAME relation, which gives provides the authors name. This must be not null and must have reference an existing name, as each author has a name. The other attributes in author are birth_date and death_date, which have a domain of DATE objects and represent the date of the authors birth and death respectively. There are no restrictions on these attributes, as many authors have not yet died, and some may have unknown birth dates.

BOOK: The BOOK relation represents a book that is sold on the Bits & Books website. The primary key of this relation is the isbn, or International Standard Book Number, which is a unique string of 13 characters. These characters are generally all numerical digits, apart from the case where the last digit is an X to represent a 10 for the check digit. The release_year attribute is a 4-character string, made up of all numerical values, that represents the year the book was originally published. The sales_price attribute is a decimal value with a precision of 10 and 2 digits after the decimal. It represents the price that each copy of the book is to be sold at. The title attribute is a string of up to 255 characters that holds the title of the book. Finally, the

last attribute of the BOOK relation is the integer, publisher_id. This attribute is a foreign key to the publisher relation; it references the PUBLISHER relation, which describes the company that publishes the book in each tuple. The publisher_id must exist for each book, as each book has a publisher, and it must reference an existing publisher in the database. The sales_price, release_year, and title must also not be null, as each book in the database has these qualities.

BOOK_CATEGORY: The BOOK_CATEGORY relation is a bridge table between the book and category relations. It allows for a book to be a part of multiple categories, as well as categories to contain many books. The only two attributes for this table are the foreign keys isbn (which references book) and category_id (references category). Neither of these attributes can be null, as the tuple would be a useless entry as it does not connect anything. The values of these attributes must also reference existing tuples in the referenced relations.

BOOK_ORDER: The BOOK_ORDER relation is used to store the quantity and identification of books, as well as the warehouse they are being shipped from for each order. Each BOOK_ORDER tuple contains the warehouse_id, isbn, and order_id to reference the WAREHOUSE they are being shipped from, the BOOK that is being ordered, and the ORDER that it is a part of. These values must all be not null and reference a valid tuple in the corresponding relation. The final attribute in the BOOK_ORDER relation is quantity, which specifies the number of copies for a specific book is placed in an order. This attribute is an integer and must be not null, as it only makes sense to keep track of a book order when at least 1 copy has been ordered.

CATEGORY: The CATEGORY relation represents the different categories that a book could belong to such as fantasy, romance, etc. This relation has the primary key category_id, which is a unique, arbitrary integer. It also has the attribute cat_name, a string of up to 20 characters, and a description of the category, a string of up to 255 characters.

EMPLOYEE: The EMPLOYEE relation is a subclass of users. While all users can access the customer facing site, you must have an EMPLOYEE tuple referencing your user_id to access the employee facing information. Each EMPLOYEE tuple has a foreign key, employee_id, which references the user_id of a tuple in USER. This connects the employee's USER account to the employee views. The employee_id attribute is required to be not null and referencing a valid user_id, as each employee must have a user account connected. For employees that work in a warehouse, the EMPLOYEE relation has a warehouse_id foreign key, which references the warehouse location that they work at. This value can be null as employees may not work at a warehouse, but if it is not null, it must reference a valid tuple in WAREHOUSE. The EMPLOYEE relation also has a position attribute, which is a string up to 25 characters. This position determines what level of access they have to the backend of the site.

NAME: The NAME relation is meant to represent the name of a person, and a tuple in NAME belongs to each tuple in the AUTHOR and USER relation. The name relation has a primary key called name_id, which is an arbitrary, unique integer assigned to each name. The name relation has the attributes fname and lname, which are both strings of up to 50 characters and correspond to a person's first and last names. The fname and lname attributes are required to

be not null, while the final attribute, middle_inits, does not need to contain a value. The middle_inits attribute is a string up to 10 characters and is used to contain a person's middle initials.

ORDER: The ORDER an instance of a user purchasing one or more books on the Bits & Books website. The relation has a primary key, order_id, which is a unique integer assigned at each order. The order has a sale_date attribute, with a domain of DATE object, used to keep track of when the order was placed. The sale_date is not null, as the date of each order is tracked when it is placed. The bill_no attribute keeps track of a third-party credit card company's bill processing number (integer) for the order and can also not be null. Finally, each order has a foreign key, customer_id, which references the user_id of the user that made the purchase.

PUBLISHER: The PUBLISHER relation represents a company or publishing group that prints, advertises, and distributes the books sold by the Bits & Books company. The primary key of this relation is publisher_id, which is a unique, arbitrary integer assigned to each publisher. The name attribute is a string of up to 50 characters and represents the official name of the publisher. The phone number and email to contact the company are stored in the attributes phone_no, a string of 10 characters, and email, a string of up to 30 characters, respectively. All these attributes are required to be non-null, as they are all necessary for Bits & Books to identify and contact the publishers of their books.

USER: The USER relation contains the information for anyone that uses the Bits & Books website, whether that is an employee or customer. Each user is identified with a user_id, which is a unique, arbitrary integer assigned to the user at the time of creating an account. Each user has an email and phone_no attribute to store their contact information, which have domains of a string up to 255 characters and a string of 10 characters respectively. A user is required to have an email in the database, but not a phone number. Like the AUTHOR relation, USER has a foreign key name_id, that references in a tuple in the NAME relation. This references tuple gives the first and last name, and potentially middle initials of the customer. The value in name_id must not be null, as each user has a name, and it must reference a valid tuple within NAME. Finally, the last attribute for each user is the foreign key address_id. This attribute references a tuple in the ADDRESS relation, which provides a home address or PO box for books to be shipped to. Since B&B is an online bookstore that ships its orders, this attribute is required to be not null and must reference a valid address tuple.

WAREHOUSE: The WAREHOUSE relation represents the locations in which Bits & Books stores books, and from where books are shipped to be delivered. Each warehouse has a unique, arbitrary warehouse_id integer as its primary key. It also has a foreign key, address_id, which references the ADDRESS relation to provide the location of the warehouse. This must be not null and reference a valid ADDRESS tuple, as each warehouse must have a shipping address. The last attribute is an integer called total_capacity. This attribute represents how many books that each warehouse location can hold. Total_capacity must also be not null, as each warehouse has a storage capacity.

WAREHOUSE_STOCK: The WAREHOUSE_STOCK relation represents the number of each book stored in each warehouse. The warehouse_id attribute is a foreign key that references the WAREHOUSE relation, specifying at which warehouse this stock is located. The isbn attribute is a foreign key to the BOOK relation, specifying which book the stock consists of. Both foreign keys must be not null and must reference a valid tuple in their corresponding relations. The final attribute in the WAREHOUSE_STOCK relation is quantity, which represents the number of books contained in the stock record. The quantity attribute must be not null, as even if there are no copies of a book in the warehouse, the quantity is 0. For example, if there are 35 copies of a book with isbn = "0001234567898" at warehouse_id = 1, a WAREHOUSE_STOCK tuple would look like (1, "0001234567898", 35).

WRITTEN_BY: The WRITTEN_BY relation connects an author to the books they have written. Each tuple in the relation has one author_id and one isbn, which are foreign keys that reference an AUTHOR and a BOOK respectively. These attributes correspond to the domains of the primary keys for their given relation, and both must reference an existing tuple. This relation allows a many to many relationship between authors and books, which is necessary as a book can be written by multiple authors, and an author can write multiple books.

2.2 Sample Queries

Simple Queries ----------

a. Find the titles of all books by Pratchett that cost less than $10

```
SELECT B.title, B.sales_price
FROM BOOK B
JOIN WRITTEN_BY WB on B.isbn = WB.isbn
JOIN AUTHOR A on A.author_id = WB.author_id
JOIN NAME N on A.name_id = N.name_id
WHERE N.lname = 'Pratchett' AND B.sales_price < 10;
```

---------- *Sample Output* ----------

title	*sales_price*
Small Gods	*7.99*
Going Postal	*7.99*
Pyramids	*7.99*
Guards! Guards!	*7.99*
Unseen Academicals	*7.99*

---

b. Give all the titles and their dates of purchase made by a single customer. In this sample query the customer is specified by USER_ID = 50.

```
SELECT B.title, O.sale_date
FROM BOOK B
JOIN BOOK_ORDER BO on B.isbn = BO.isbn
JOIN "ORDER" O ON O.order_id = BO.order_id
JOIN USER U on O.customer_id = U.user_id
WHERE U.user_id = 50;
```

---------- *Sample Output* ----------

title	*sale_date*
Enduring Love	2012-06-15
Execution: The Discipline of Getting Things Done	2012-06-15
Guards! Guards!	2012-06-15
Linked: The New Science of Networks	2012-06-15
UNDERWORLD: A NOVEL	2012-06-15

---

c. Find the titles and ISBNs for all books with less than 5 copies in stock
NOTE: only one book has less than 5 in stock (Pale Blue Dot).

```
SELECT B.title, B.isbn, sum(WS.quantity) AS stock
FROM BOOK B
JOIN WAREHOUSE_STOCK WS on B.isbn = WS.isbn
GROUP BY WS.isbn
HAVING sum(WS.quantity) < 5;
```

---------- *Sample Output* ----------

title	*isbn*	*stock*
Pale Blue Dot: A Vision of the Human Future in Space	0000345376595	4

---

d. Give all the customers who purchased a book by Pratchett and the titles of Pratchett books they purchased.

```
SELECT DISTINCT NUser.fname, NUser.lname, B.title
FROM NAME NUser
JOIN USER U on NUser.name_id = U.name_id
JOIN "ORDER" on U.user_id = "ORDER".customer_id
JOIN BOOK_ORDER BO on "ORDER".order_id = BO.order_id
JOIN BOOK B on BO.isbn = B.isbn
JOIN WRITTEN_BY WB on B.isbn = WB.isbn
JOIN AUTHOR A on WB.author_id = A.author_id
JOIN NAME NAuth on A.name_id = NAuth.name_id
WHERE NAuth.lname = 'Pratchett'
ORDER BY U.user_id;
```

---------- Sample Output ----------

fname	*lname*	*title*
Beula	Jerde	Going Postal
Marcy	Heathcote	Unseen Academicals
Art	Bednar	The Color of Magic
Darren	Kub	Small Gods
Darren	Kub	Guards! Guards!

----

e. Find the total number of books purchased by a single customer. In this sample query the customer is specified by USER_ID = 50.

```
SELECT U.user_id, N.fname, N.lname, sum(BO.quantity) as books_purchased
FROM USER U
JOIN NAME N on U.name_id = N.name_id
JOIN "ORDER" O on O.customer_id = U.user_id
JOIN BOOK_ORDER BO on O.order_id = BO.order_id
WHERE U.user_id = 50;
```

---------- Sample Output ----------

user_id	*fname*	*lname*	*books_purchased*
50	Clarence	Vandervort	63

----

f. Find the customer who has purchased the most books and the total number of books they have purchased.

```
SELECT U.user_id, N.fname, N.lname, sum(BO.quantity) as books_purchased
FROM USER U
JOIN NAME N on U.name_id = N.name_id
JOIN "ORDER" O on O.customer_id = U.user_id
JOIN BOOK_ORDER BO on O.order_id = BO.order_id
GROUP BY U.user_id, N.fname, N.lname
HAVING books_purchased = (SELECT max(total_books)
                          FROM (SELECT sum(B.quantity) as total_books
                                FROM USER U2
                                JOIN "ORDER" ON U2.user_id = "ORDER".customer_id
                                JOIN BOOK_ORDER B on "ORDER".order_id = B.order_id
                                GROUP BY user_id));
```

---------- *Sample Output* ----------

user_id	*fname*	*lname*	*books_purchased*
152	*Berneice*	*Huels*	*93*

Extra Queries ----------

a. Find the author who has sold the most books through B&B.

```
SELECT AName.fname, AName.middle_inits, AName.lname, sum(BO.quantity) as books_sold
FROM AUTHOR A
JOIN NAME AName on A.name_id = AName.name_id
JOIN WRITTEN_BY WB on A.author_id = WB.author_id
JOIN BOOK B on WB.isbn = B.isbn
JOIN BOOK_ORDER BO on B.isbn = BO.isbn
GROUP BY A.author_id
HAVING books_sold = (SELECT max(b_sold)
                     FROM (SELECT sum(BO.quantity) as b_sold
                           FROM AUTHOR A
                           JOIN WRITTEN_BY WB on A.author_id = WB.author_id
                           JOIN BOOK B on WB.isbn = B.isbn
                           JOIN BOOK_ORDER BO on B.isbn = BO.isbn
                           GROUP BY A.author_id));
```

---------- *Sample Output* ----------

fname	*middle_inits*	*lname*	*b_sold*
Terry	*N/A*	*Pratchett*	*327*

---

b. Check which warehouses have enough stock to order 40 copies of a book. The book chosen in this sample has isbn = 0000385494327

```
SELECT B.isbn, WAdd.address as warehouse_addr, WAdd.city, WAdd.state, WS.quantity as
stock
FROM BOOK B
JOIN WAREHOUSE_STOCK WS on B.isbn = WS.isbn
JOIN WAREHOUSE W on WS.warehouse_id = W.warehouse_id
JOIN ADDRESS WAdd on W.address_id = WAdd.address_id
WHERE B.isbn = '0000385494327' AND WS.quantity >= 40;
```

---------- Sample Output ----------

isbn	address	city	state	stock
0000385494327	3877 Cremin Estate	Wizafurt	MD	54

----

c. Determine the total revenue of the bookstore

```
SELECT  sum(B.sales_price * BO.quantity) as total_sales
FROM BOOK B
JOIN BOOK_ORDER BO on B.isbn = BO.isbn
JOIN "ORDER" O on BO.order_id = O.order_id;
```

---------- Sample Output ----------

total_sales
185839.71000000133

Advanced Queries ----------

a. Provide a list of customer names, along with the total dollar amount each customer has spent.

```
SELECT N.fname, N.lname, sum(B.sales_price * BO.quantity) as total_spent
FROM BOOK B
JOIN BOOK_ORDER BO on B.isbn = BO.isbn
JOIN "ORDER" O on BO.order_id = O.order_id
JOIN USER U on O.customer_id = U.user_id
JOIN NAME N on U.name_id = N.name_id
GROUP BY U.user_id
ORDER BY total_spent DESC;
```

---------- *Sample Output* ----------

fname	*lname*	*total_spent*
Betsy	Shields	3497.4199999999996
Clement	Rau	3218.41
Berneice	Huels	3001.4899999999993
Barrett	Larson	2895.13
Rene	Muller	2892.209999999999

----

b. Provide a list of customer names and e-mail addresses for customers who have spent more than the average customer.

```
SELECT N.fname, N.lname, U.email, sum(B.sales_price * BO.quantity) as total_spent
FROM BOOK B
JOIN BOOK_ORDER BO on B.isbn = BO.isbn
JOIN "ORDER" O on BO.order_id = O.order_id
JOIN USER U on O.customer_id = U.user_id
JOIN NAME N on U.name_id = N.name_id
GROUP BY U.user_id
HAVING total_spent > (SELECT avg(spent_per_c)
                      FROM ( SELECT sum(B.sales_price * BO.quantity) as spent_per_c
                             FROM BOOK B
                             JOIN BOOK_ORDER BO on B.isbn = BO.isbn
                             JOIN "ORDER" O on BO.order_id = O.order_id
                             JOIN USER U on O.customer_id = U.user_id
                             JOIN NAME N on U.name_id = N.name_id
                             GROUP BY U.user_id))
ORDER BY total_spent DESC;
```

---------- Sample Output ----------

fname	*lname*	*total_spent*
Betsy	Shields	3497.4199999999996
Clement	Rau	3218.41
Berneice	Huels	3001.4899999999993
Barrett	Larson	2895.13
Rene	Muller	2892.209999999999

----

c. Provide a list of the titles in the database and associated total copies sold to customers, sorted from the title that has sold the most individual copies to the title that has sold the least.

```
SELECT B.title, sum(BO.quantity) as copies_sold
FROM BOOK B
JOIN BOOK_ORDER BO on B.isbn = BO.isbn
GROUP BY B.isbn
ORDER BY copies_sold DESC;
```

---------- Sample Output ----------

title	*copies_sold*
The Vanished Man: A Lincoln Rhyme Novel	*76*
Twelve Times Blessed	*75*
Wolves of the Calla	*73*
Cabins & Camps	*72*

----

d. Provide a list of the titles in the database and associated dollar totals for copies sold to customers, sorted from the title that has sold the highest dollar amount to the title that has sold the smallest.

```
SELECT B.title, sum(BO.quantity * B.sales_price) as dollar_sold_total
FROM BOOK B
JOIN BOOK_ORDER BO on B.isbn = BO.isbn
GROUP BY B.isbn
ORDER BY dollar_sold_total DESC;
```

---------- Sample Output ----------

title	*dollar_sold_total*
Intermediate Accounting	*9047.999999999998*
Advanced Accounting	*6777*
Creating Documents with BusinessObjects 5.1	*6534*
OCP: Oracle9i Certification Kit	*6193.229999999999*
Financial Reporting and Analysis	*5953.490000000002*

----

e. Find the most popular author in the database (i.e. the one who has sold the most books)

```
SELECT AName.fname, AName.middle_inits, AName.lname, sum(BO.quantity) as books_sold
FROM AUTHOR A
JOIN NAME AName on A.name_id = AName.name_id
JOIN WRITTEN_BY WB on A.author_id = WB.author_id
JOIN BOOK B on WB.isbn = B.isbn
JOIN BOOK_ORDER BO on B.isbn = BO.isbn
GROUP BY A.author_id
HAVING books_sold = (SELECT max(b_sold)
                     FROM (SELECT sum(BO.quantity) as b_sold
                           FROM AUTHOR A
                           JOIN WRITTEN_BY WB on A.author_id = WB.author_id
                           JOIN BOOK B on WB.isbn = B.isbn
                           JOIN BOOK_ORDER BO on B.isbn = BO.isbn
                           GROUP BY A.author_id));
```

---------- Sample Output ----------

fname	*middle_inits*	*lname*	*books_sold*
Terry	*N/A*	*Pratchett*	*327*

----

f. Find the most profitable author in the database for this store (i.e. the one who has brought in the most money)

```
SELECT AName.fname, AName.middle_inits, AName.lname, sum(BO.quantity * B.sales_price)
as sales_total
FROM AUTHOR A
JOIN NAME AName on A.name_id = AName.name_id
JOIN WRITTEN_BY WB on A.author_id = WB.author_id
JOIN BOOK B on WB.isbn = B.isbn
JOIN BOOK_ORDER BO on B.isbn = BO.isbn
GROUP BY A.author_id
HAVING sales_total = (SELECT max(b_sold)
                      FROM (SELECT sum(BO.quantity * B.sales_price) as b_sold
                            FROM AUTHOR A
                            JOIN WRITTEN_BY WB on A.author_id = WB.author_id
                            JOIN BOOK B on WB.isbn = B.isbn
                            JOIN BOOK_ORDER BO on B.isbn = BO.isbn
                            GROUP BY A.author_id));
```

---------- Sample Output ----------

fname	*middle_inits*	*lname*	*sales_total*
Jeffrey	*M.*	*Wooldridge*	*9153.949999999997*

----

g. Provide a list of customer information for customers who purchased anything written by the most profitable author in the database.

```
SELECT UName.fname, UName.lname, U.email, U.phone_no, Ad.address, Ad.city, Ad.state,
Ad.country
FROM USER U
JOIN NAME UName on U.name_id = UName.name_id
JOIN ADDRESS Ad on U.address_id = Ad.address_id
JOIN "ORDER" O on U.user_id = O.customer_id
JOIN BOOK_ORDER BO on O.order_id = BO.order_id
JOIN BOOK B on BO.isbn = B.isbn
JOIN WRITTEN_BY WB on B.isbn = WB.isbn
JOIN AUTHOR A on WB.author_id = A.author_id
WHERE A.author_id = (SELECT A.author_id
                     FROM AUTHOR A
                     JOIN NAME AName on A.name_id = AName.name_id
                     JOIN WRITTEN_BY WB on A.author_id = WB.author_id
                     JOIN BOOK B on WB.isbn = B.isbn
                     JOIN BOOK_ORDER BO on B.isbn = BO.isbn
                     GROUP BY A.author_id
                     HAVING sum(BO.quantity * B.sales_price) = (SELECT max(b_sold)
                             FROM (SELECT sum(BO.quantity * B.sales_price) as b_sold
                                   FROM AUTHOR A
                                   JOIN WRITTEN_BY WB on A.author_id = WB.author_id
                                   JOIN BOOK B on WB.isbn = B.isbn
                                   JOIN BOOK_ORDER BO on B.isbn = BO.isbn
                                   GROUP BY A.author_id)))
                     GROUP BY U.user_id;
```

---------- *Sample Output* ----------

fname	*lname*	*email*	*phone_no*	*address*	*city*	*state*	*country*
Art	Bednar	Art.Bednar374@yahoo.com	7344067307	7282 Mervin Highway	Bradlyton	RI	USA
Darren	Kub	Darren.Kub824@hotmail.com	6169490792	66380 Bergnaum Throughway	Ondrickamouth	NV	USA
Landon	Goyette	Landon.Goyette625@yahoo.com	8038064828	864 Kelly Causeway	North Maynardburgh	HI	USA
Oscar	Beahan	Oscar.Beahan400@hotmail.com	5103145712	2168 Dorian Causeway	Lake Aileen	GA	USA
Barrett	Larson	Barrett.Larson489@gmail.com	8148162408	33392 Lucretia Motorway	Daughertyton	MA	USA

---

h. Provide the list of authors who wrote the books purchased by the customers who have spent more than the average customer.

```
SELECT AuthName.fname, AuthName.middle_inits, AuthName.lname
FROM BOOK B
JOIN BOOK_ORDER BO on B.isbn = BO.isbn
JOIN WRITTEN_BY WB on B.isbn = WB.isbn
JOIN AUTHOR A on WB.author_id = A.author_id
JOIN NAME AuthName on A.name_id = AuthName.name_id
GROUP BY A.author_id
HAVING sum(B.sales_price * BO.quantity) > (SELECT avg(spent_per_c)
                    FROM ( SELECT sum(B.sales_price * BO.quantity) as spent_per_c
                           FROM BOOK B
                           JOIN BOOK_ORDER BO on B.isbn = BO.isbn
                           JOIN "ORDER" O on BO.order_id = O.order_id
                           JOIN USER U on O.customer_id = U.user_id
                           JOIN NAME N on U.name_id = N.name_id
                           GROUP BY U.user_id));
```

---------- Sample Output ----------

fname	*middle_inits*	*lname*
Chip		*Dawes*
Biju		*Thomas*
Doug		*Stuns*
Matthew		*Weishan*
Joseph	*C.*	*Johnson*

2.3 INSERT Syntax

PUBLISHER: A PUBLISHER entry has 4 attributes, one of which is a primary key and none of which are foreign keys. The values for a publisher are 1) publisher_id, which must be a unique integer, 2) the publisher's name, 3) the publisher's email, and 4) the publisher's phone number, where the latter 2 attributes are strings. As the publisher_id will likely be used as a foreign key for a BOOK, it is good to keep track of the publisher_id while inserting a PUBLISHER. Below shows the format for inserting a publisher as well as a sample INSERT statement.

```
INSERT INTO PUBLISHER
VALUES (publisher_id: INTEGER, pub_name: VARCHAR(50), phone_no:
CHAR(10), email: VARCHAR(30));

INSERT INTO PUBLISHER
VALUES (145, 'Canaan Publishing Group', '6145001234',
'supply@canaanco.com');
```

BOOK: Each BOOK in the database has 5 attributes, one of which is a foreign key, and one many to many relationship maintained through a bridge table. The non-foreign key attributes isbn, release_year, sales_price, and title can be any values allowed by their domain, and isbn must be unique as it is the primary key. The publisher_id attribute is a foreign key that references the PUBLISHER table. The value entered for publisher_id must reference an already existing PUBLISHER in the database. If this book is published by a publisher not yet in the database, the publisher must be inserted into the database before the book, the directions for which are in the preceding section. To connect the credited authors to the book being inserted, keep reading up to the section WRITTEN_BY. A sample INSERT statement for a BOOK with an existing publisher with publisher_id = 1 is shown below.

```
INSERT INTO BOOK
VALUES (isbn: CHAR(13), release_year: INTEGER(4), sales_price:
DECIMAL(16,2), title: VARCHAR(255), publisher_id: INTEGER);

INSERT INTO BOOK
VALUES ('0009117382618', 2004, 24.99, 'Sample Title', 145);
```

NAME: Adding a NAME to the database require 4 attributes: a name_id, which is a unique integer, then a first name (up to 50 chars), middle initials (up to 10 chars), and last name (up to 50 chars). Below shows the format and a sample of adding a new NAME with name_id = 50. Since a NAME should always be referenced by either an AUTHOR or a USER, make sure to keep track of the name_id when adding a new NAME.

```
INSERT INTO NAME
VALUES (name_id: INTEGER, fname: VARCHAR(50), middle_inits:
VARCHAR(10), lname: VARCHAR(50));
```

```
INSERT INTO NAME
VALUES (50, 'Canaan', 'M.', 'Porter');
```

AUTHOR: Before you INSERT an AUTHOR to the database, you must add their name to the NAME table. How to do this is explained in the immediately preceding section titled “NAME”. Once the name has been added, you can use an insert statement with the following 2-4 values: 1) author_id, which is a unique integer, 2) the name_id of the NAME that was added for this author, 3) The birth_date of the author, 4) the death_date of the author. Values for 3 and 4 are optional, as many birthdates are unknown, and many authors are still alive. The formatting of how to INSERT an AUTHOR is shown below, as well as a sample statement using author_id = 100 and referencing the NAME with name_id = 50.

```
INSERT INTO AUTHOR
VALUES (author_id: INTEGER, name_id: INTEGER, birth_date: DATE,
death_date: DATE);

INSERT INTO AUTHOR
VALUES (50, 100, 12-24-1912, 5-8-1998);
```

WRITTEN_BY: Adding an entry to the WRITTEN_BY table allows you to connect an AUTHOR with a BOOK that they have written. There are only 2 attributes in a WRITTEN_BY entry, both foreign keys: 1) author_id, an integer referencing an AUTHOR, and 2) isbn, a string referencing a BOOK. These both must be valid references, so if you are in the process of adding books and their authors, make sure to add both the books and the authors before connecting them. Below shows the format of adding to the WRITTEN_BY table along with a sample INSERT statement, which connects our AUTHOR (author_id = 50) who has the name “Canaan M. Porter” to the book with ISBN = '0009117382618'.

```
INSERT INTO WRITTEN_BY
VALUES (author_id: INTEGER, isbn: CHAR(13));

INSERT INTO WRITTEN_BY
VALUES (50, '0009117382618');
```

ADDRESS: Adding an ADDRESS entry to the database takes 7 values: 1) address_id, a unique integer often referenced as a foreign key from USER and WAREHOUSE, 2) address, 3) secondary address, 4) city, 5) state, 6) zip, and 7) country. The secondary address is not required, so to insert an ADDRESS without a secondary address, just use NULL for the value of the secondary address. Shown below is the format for inserting an address into the database, along with a sample INSERT statement.

```
INSERT INTO ADDRESS
VALUES (address_id: INTEGER, address: VARCHAR(100),
```

```
secondary_address: VARCHAR(50), city: VARCHAR(30), state:
CHAR(2), zip: VARCHAR(10), country: CHAR(3));

INSERT INTO ADDRESS
VALUES (5, '1234 Lane Ave', NULL , 'Columbus', 'OH', '43210',
'USA');
```

USER: Adding a user requires 5 attributes, one of which is a primary key and 2 of which are foreign keys. The ordering for adding these attributes are 1) user_id, the primary key for USER, a unique integer, 2) name_id, a foreign key referencing the user's NAME, 3) the user's email, 4) the user's phone number, and 5) address_id, a foreign key referencing the user's address. If the user's name or address does not already exist in the database (some user's may have the same name/some user's may live together), then the corresponding NAME and ADDRESS must be added before the USER is inserted. Shown below is the format for adding a USER to the database, along with a user who has name_id = 100 and address_id = 5. Looking at our previous examples, this user is named 'Canaan M. Porter' and lives at 1234 Lane Ave, Columbus, OH 43210.

```
INSERT INTO USER
VALUES (user_id: INTEGER, name_id: INTEGER, email: VARCHAR(255),
phone_no: CHAR(10), address_id: INTEGER);

INSERT INTO USER
VALUES (51, 100, 'canaan@mail.com', '1234567898', 5);
```

2.4 DELETE Syntax

PUBLISHER: To delete an entry from the PUBLISHER table, you must first remove all BOOK entries that refer to the given PUBLISHER. For example, if you want to delete the PUBLISHER with publisher_id = 1, all BOOK entries with publisher_id = 1, must be deleted first. Deleting a BOOK has its own requirements, so continue reading the next sections before trying to delete any BOOK entries. After all referencing BOOKs have been removed, we can delete the PUBLISHER with publisher_id = X as follows:

```
DELETE FROM PUBLISHER
WHERE publisher_id = X;
```

BOOK: To delete a BOOK entry, all WRITTEN_BY and BOOK_CATEGORY entries that reference the BOOK's isbn must be deleted first. This can be done for a BOOK with isbn = Y with the queries:

```
DELETE FROM WRITTEN_BY
WHERE isbn = Y;
```

```
DELETE FROM BOOK_CAGTEGORY
WHERE isbn = Y;
```

From there, the BOOK can be deleted safely with:

```
DELETE FROM BOOK
WHERE isbn = Y;
```

To tie this back together to delete a publisher with publisher_id, we can chain together the 4 queries to remove the PUBLISHER with publisher_id = X:

```
DELETE FROM WRITTEN_BY
WHERE isbn IN (SELECT isbn
               FROM BOOK
               WHERE publisher_id = X);

DELETE FROM BOOK_CATEGORY
WHERE isbn IN (SELECT isbn
               FROM BOOK
               WHERE publisher_id = X);

DELETE FROM BOOK
WHERE isbn IN (SELECT isbn
               FROM BOOK
               WHERE publisher_id = X);

DELETE FROM PUBLISHER
WHERE publisher_id = X;
```

AUTHOR: To delete an AUTHOR from the database, all WRITTEN_BY entries that reference the AUTHOR's author_id must be deleted first. This can be done for an AUTHOR with author_id = Z with the query:

```
DELETE FROM WRITTEN_BY
WHERE author_id = Z;
```

Then the AUTHOR can be deleted safely with:

```
DELETE FROM AUTHOR
WHERE author_id = Z;
```

The AUTHOR's NAME should also be removed to save space, as long as there are no other USER's or AUTHOR's who also reference that name. All unreferenced names can be removed from the database with the following query:

```
DELETE FROM NAME
WHERE name_id NOT IN (SELECT name_id
       FROM NAME N, USER U, AUTHOR A
       WHERE N.name_id = U.name_id OR N.name_id = A.name_id);
```

USER: A USER entry is somewhat complicated to delete, as all of their ORDERs and the corresponding BOOK_ORDERS must be deleted first. If the USER is an EMPLOYEE, the corresponding EMPLOYEE entry must also be deleted first. These can all be deleted for a USER with user_id = V with the following sequence of queries:

```
DELETE FROM BOOK_ORDER
WHERE order_id IN (SELECT order_id
                   FROM "ORDER" O, USER U
                   WHERE O.customer_id = U.user_id
                   AND U.user_id = V);

DELETE FROM "ORDER"
WHERE order_id IN (SELECT order_id
                   FROM "ORDER" O, USER U
                   WHERE O.customer_id = U.user_id
                   AND U.user_id = V);

DELETE FROM EMPLOYEE
WHERE employee_id = V;
```

After these queries have been executed, you can safely delete the USER with:

```
DELETE FROM USER
WHERE user_id = V;
```

Again, after deleting a USER we may want to clean up any unused NAME or ADDRESSES. We can do so with the following queries:

```
DELETE FROM NAME
WHERE name_id NOT IN (SELECT N.name_id
       FROM NAME N, USER U, AUTHOR A
       WHERE N.name_id = U.name_id OR N.name_id = A.name_id);

DELETE FROM ADDRESS
WHERE address_id NOT IN (SELECT A.address_id
                         FROM ADDRESS A, USER U, WAREHOUSE W
                         WHERE A.address_id = U.address_id OR
A.address_id = W.address_id);
```

2.4 INSERT-DELETE SQL Walkthrough

This next section provides an easy step-by-step walkthrough for inserting and deleting from the tables in this database. Using the seeded database provided, you can execute each SQL statement listed here in order to see the order that each insertion and deletion must take. The SELECT statements are often in pairs, one before the delete and one after, so you can see the effect you have made on the database with each statement. Now that you understand the ordered dependencies of this database, feel free to mess around with these!

```
-- Creating New Publisher
INSERT INTO PUBLISHER
VALUES (1000, 'Canaan Publishing Group', '6145001234',
'supply@canaanco.com');

-- Checking that the new publisher was created
SELECT *
FROM PUBLISHER
WHERE publisher_id = 1000;

-- Creating New Book
INSERT INTO BOOK
VALUES ('0101010101010', 2005, 24.99, 'Sample Title', 1000);

-- Checking that the new book was created
SELECT *
FROM BOOK
WHERE isbn = '0101010101010';

-- Creating New Name
INSERT INTO NAME
VALUES (300, 'Canaan', 'Porter', 'M.');

-- Checking that the new name was created
SELECT *
FROM NAME
WHERE name_id = 300;

-- Creating New Author with name Canaan M. Porter
INSERT INTO AUTHOR
VALUES (500, 300, '1912-12-24', '1998-5-8');
```

```sql
-- Checking that the new AUTHOR was created
SELECT author_id, birth_date, death_date, fname, middle_inits,
lname
FROM AUTHOR, NAME
WHERE author_id = 500 AND AUTHOR.name_id = NAME.name_id;

-- Connecting Newly Created Author and Book
INSERT INTO WRITTEN_BY
VALUES (500, '0101010101010');

-- Checking that the connection was made
SELECT B.title, N.fname, N.lname, B.publisher_id
FROM BOOK B, WRITTEN_BY WB, AUTHOR A, NAME N
WHERE B.isbn = WB.isbn AND WB.author_id = A.author_id
AND A.name_id = N.name_id AND A.author_id = 500;

-- Connecting the new book to a category
INSERT INTO BOOK_CATEGORY
VALUES ('0101010101010', 2);

-- Checking that the connection was made
SELECT B.title, C.cat_name
FROM BOOK B, BOOK_CATEGORY BC, CATEGORY C
WHERE B.isbn = BC.isbn AND BC.category_id = C.category_id
AND B.isbn = '0101010101010';

-- Creating New Address
INSERT INTO ADDRESS
VALUES (1000, '1234 Lane Ave', NULL , 'Columbus', 'OH', '43210',
'USA');

-- Checking that the new address was created
SELECT *
FROM ADDRESS
WHERE address_id = 1000;

-- Creating New User with name Canaan M. Porter
INSERT INTO USER
VALUES (200, 300, 'canaan@mail.com', '1234567898', 1000);

-- Checking the new user
SELECT N.fname, N.middle_inits, N.lname, U.email, U.phone_no,
A.address, A.city, A.state
FROM USER U, NAME N, ADDRESS A
WHERE U.name_id = N.name_id AND U.address_id = A.address_id
AND U.user_id = 200;
```

```sql
-- DELETES
-- Observe the WRITTEN_BYs of the books from Canaan Publishing
Group
SELECT BOOK.isbn, author_id
FROM WRITTEN_BY, BOOK
WHERE WRITTEN_BY.isbn = BOOK.isbn
AND BOOK.publisher_id = 1000;

-- Deleting WRITTEN_BY entries on books from Canaan Publishing
Group
DELETE FROM WRITTEN_BY
WHERE isbn IN (SELECT isbn
               FROM BOOK
               WHERE publisher_id = 1000);

-- Make the same query as before, should have no results
SELECT BOOK.isbn, author_id
FROM WRITTEN_BY, BOOK
WHERE WRITTEN_BY.isbn = BOOK.isbn
AND BOOK.publisher_id = 1000;

-- Checking the book categories on books from Canaan Publishing
Group
SELECT BOOK.isbn, category_id
FROM BOOK_CATEGORY, BOOK
WHERE BOOK_CATEGORY.isbn = BOOK.isbn
AND BOOK.publisher_id = 1000;

-- Deleting BOOK_CATEGORY entries on books from Canaan
Publishing Group
DELETE FROM BOOK_CATEGORY
WHERE isbn IN (SELECT isbn
               FROM BOOK
               WHERE publisher_id = 1000);

-- Checking the same query, should be empty
SELECT BOOK.isbn, category_id
FROM BOOK_CATEGORY, BOOK
WHERE BOOK_CATEGORY.isbn = BOOK.isbn
AND BOOK.publisher_id = 1000;

-- Checking books written by Canaan Publishing Group
SELECT *
FROM BOOK
WHERE BOOK.publisher_id = 1000;
```

```sql
-- Deleting BOOK entries from Canaan Publishing Group
DELETE FROM BOOK
WHERE isbn IN (SELECT isbn
               FROM BOOK
               WHERE publisher_id = 1000);

-- Comparing the last query, should be empty now
SELECT *
FROM BOOK
WHERE BOOK.publisher_id = 1000;

-- Deleting WRITTEN_BY entries written by Canaan Porter
DELETE FROM WRITTEN_BY
WHERE author_id = 500;

-- Querying for the Canaan Publishing Group
SELECT *
FROM PUBLISHER
WHERE publisher_id = 1000;

-- Deleting Canaan Publishing Group
DELETE FROM PUBLISHER
WHERE publisher_id = 1000;

-- Making the same query, should be empty
SELECT *
FROM PUBLISHER
WHERE publisher_id = 1000;

-- Checking the book orders of USER 21
SELECT B.title, BO.quantity
FROM BOOK B, BOOK_ORDER BO, "ORDER" O, USER U
WHERE U.user_id = O.customer_id AND O.order_id = BO.order_id
AND BO.isbn = B.isbn AND U.user_id = 21;

-- Deleting all the book orders of USER #21
DELETE FROM BOOK_ORDER
WHERE order_id IN (SELECT order_id
                   FROM "ORDER" O, USER U
                   WHERE O.customer_id = U.user_id
                   AND U.user_id = 21);

-- Making the same query, should be empty
SELECT B.title, BO.quantity
FROM BOOK B, BOOK_ORDER BO, "ORDER" O, USER U
WHERE U.user_id = O.customer_id AND O.order_id = BO.order_id
AND BO.isbn = B.isbn AND U.user_id = 21;
```

```sql
-- Checking the orders of USER 21
SELECT O.order_id, O.sale_date
FROM "ORDER" O, USER U
WHERE U.user_id = O.customer_id AND U.user_id = 21;

-- Deleting all the orders of USER #21
DELETE FROM "ORDER"
WHERE order_id IN (SELECT order_id
                   FROM "ORDER" O, USER U
                   WHERE O.customer_id = U.user_id
                   AND U.user_id = 21);

-- Same query, should be empty now
SELECT O.order_id, O.sale_date
FROM "ORDER" O, USER U
WHERE U.user_id = O.customer_id AND U.user_id = 21;

-- Checking if User #1 is an employee
SELECT *
FROM EMPLOYEE, USER
WHERE employee_id = USER.user_id
AND user_id = 1;

-- Deleting the EMPLOYEE tuple for USER #1
DELETE FROM EMPLOYEE
WHERE employee_id = 1;

-- Checking if User #1 is an employee, should not be
SELECT *
FROM EMPLOYEE, USER
WHERE employee_id = USER.user_id
AND user_id = 1;

-- Checking USER #21
SELECT N.fname, N.middle_inits, N.lname, U.email, U.phone_no,
A.address, A.city, A.state
FROM USER U, NAME N, ADDRESS A
WHERE U.name_id = N.name_id AND U.address_id = A.address_id
AND U.user_id = 21;

-- Deleting USER #21
DELETE FROM USER
WHERE user_id = 21;

-- Checking USER #21 again, should be deleted
SELECT N.fname, N.middle_inits, N.lname, U.email, U.phone_no,
```

```
A.address, A.city, A.state
FROM USER U, NAME N, ADDRESS A
WHERE U.name_id = N.name_id AND U.address_id = A.address_id
AND U.user_id = 21;

-- Checking USER #1
SELECT N.fname, N.middle_inits, N.lname, U.email, U.phone_no,
A.address, A.city, A.state
FROM USER U, NAME N, ADDRESS A
WHERE U.name_id = N.name_id AND U.address_id = A.address_id
AND U.user_id = 1;

-- Deleting USER #1
DELETE FROM USER
WHERE user_id = 1;

-- Checking USER #1 again, should be deleted
SELECT N.fname, N.middle_inits, N.lname, U.email, U.phone_no,
A.address, A.city, A.state
FROM USER U, NAME N, ADDRESS A
WHERE U.name_id = N.name_id AND U.address_id = A.address_id
AND U.user_id = 1;

-- Checking USER #200
SELECT N.fname, N.middle_inits, N.lname, U.email, U.phone_no,
A.address, A.city, A.state
FROM USER U, NAME N, ADDRESS A
WHERE U.name_id = N.name_id AND U.address_id = A.address_id
AND U.user_id = 200;

-- Deleting USER #200 (Canaan)
DELETE FROM USER
WHERE user_id = 200;

-- Checking USER #200 again, should be deleted
SELECT N.fname, N.middle_inits, N.lname, U.email, U.phone_no,
A.address, A.city, A.state
FROM USER U, NAME N, ADDRESS A
WHERE U.name_id = N.name_id AND U.address_id = A.address_id
AND U.user_id = 200;

-- Checking all unused names
SELECT *
FROM NAME
WHERE name_id NOT IN (SELECT N.name_id
                      FROM NAME N, USER U, AUTHOR A
                      WHERE N.name_id = U.name_id OR N.name_id =
```

```
A.name_id);

-- Deleting all unused names
DELETE FROM NAME
WHERE name_id NOT IN (SELECT N.name_id
                      FROM NAME N, USER U, AUTHOR A
                      WHERE N.name_id = U.name_id OR N.name_id =
A.name_id);

-- Checking all unused names again, should be deleted
SELECT *
FROM NAME
WHERE name_id NOT IN (SELECT N.name_id
                      FROM NAME N, USER U, AUTHOR A
                      WHERE N.name_id = U.name_id OR N.name_id =
A.name_id);

-- Checking all unused addresses
SELECT *
FROM ADDRESS
WHERE address_id NOT IN (SELECT A.address_id
                      FROM ADDRESS A, USER U, WAREHOUSE W
                      WHERE A.address_id = U.address_id OR
A.address_id = W.address_id);

-- Deleting all unused addresses
DELETE FROM ADDRESS
WHERE address_id NOT IN (SELECT A.address_id
                      FROM ADDRESS A, USER U, WAREHOUSE W
                      WHERE A.address_id = U.address_id OR
A.address_id = W.address_id);

-- Checking all unused addresses again, should be deleted
SELECT *
FROM ADDRESS
WHERE address_id NOT IN (SELECT A.address_id
                      FROM ADDRESS A, USER U, WAREHOUSE W
                      WHERE A.address_id = U.address_id OR
A.address_id = W.address_id);

-- Checking for warehouse #20
SELECT * FROM WAREHOUSE
WHERE warehouse_id = 20;

-- Deleting warehouse #20
DELETE FROM WAREHOUSE
WHERE warehouse_id = 20;
```

```
-- Checking for warehouse #20 again, should be empty
SELECT * FROM WAREHOUSE
WHERE warehouse_id = 20;
```

---

3. Extra Features

3.1 CSV Parser/Generator

Bits & Books provided a csv file of the books that they sell, along with their corresponding authors, publishers, and other general information (release year, category, sales price). While this provided some of the specifications for what the database contains, the file was not formatted such that the data could be directly into the database. The rows of the csv file were not consistent, as some would contain a book's information, and some would include only an author's name. Rows containing only an author's name represent that the author is credited for writing the book that was last listed, and occurs when there are multiple authors credited to a book.

Since this provided data is inconsistent and unable to be dumped directly into our database, we used the Ruby programming language to create a class, *CsvParser*, that can parse a csv file into a ruby *hash* object, and inversely generate new csv files from *hash* object. Using this *CsvParser* class, we wrote a ruby reads in the given csv files, manipulates the data, then generates own csv files specified to the format of our database tables. These files can then be dumped directly into the database.

Along with providing the ability to easily generate the database from scratch, this class also provides the structural framework to add data from other csv files to an already existing database. This requires careful consideration of the data already in the database, as it must avoid adding duplicate entries such as the same author with a new author_id. This can be done by querying each entity to make sure that it does not exist before adding it to the csv, and eventually the database.

The full code for the CsvParser class can be found in Appendix C.1.

3.2 Database Seeding

While the books, authors, publishers, and categories to be stored in the database were provided in the given csv files, this was not enough data to simulate the functionality of the database for testing. Building on the ruby code that extracted the data from the provided csv file, we used

the RubyGem Faker with the CsvParser class to seed the database with sample data. After reading in and processes data from the given csv, the bb_parser.rb program uses a set of parameters and the given data to generate sample csvs to import into the remaining tables that in the database schema.

The following parameters are included:

- NUM_CUSTOMERS: The number of customers to be generated for the USER table.
- NUM_EMPLOYEES: The number of employees to be generated for the USER and EMPLOYEE tables.
- MAX_ORDERS_PER_CUSTOMER: The maximum number of ORDER tuples that will be generated per customer, along with a NAME per customer.
- MIN_ORDERS_PER_CUSTOMER: The minimum number of ORDER tuples that will be generated per customer, along with a NAME per customer.
- MAX_DIFF_BOOKS_PER_ORDER: The maximum number of BOOK_ORDERS to be generated for each ORDER.
- MIN_DIFF_BOOKS_PER_ORDER: The minimum number of BOOK_ORDERS to be generated for each ORDER.
- MAX_BOOK_ORDER_QUANTITY: The maximum value of the quantity attribute for each BOOK_ORDER generated.
- MIN_BOOK_ORDER_QUANTITY: The minimum value of the quantity attribute for each BOOK_ORDER generated.
- WAREHOUSES: The number of WAREHOUSE tuples to generate, along with an ADDRESS tuple for each WAREHOUSE.
- WAREHOUSES_IN_USE: The number of WAREHOUSE tuples that get WAREHOUSE_STOCK assigned to them.
- START_SALES_DATE: The earliest date that an ORDER tuple can have.
- END_SALES_DATE: The latest date that an ORDER tuple can have.
- RESERVE_SPACE: The amount of extra capacity that each WAREHOUSE ideally has (not stocked with books).

This seeding program can be run under the shell script *"gen_new_seed.sh"*, which uses the sqlite3 tool to create a new sqlite database, runs the schema script, then the seeding program, and finally adds each of the generated csv files to the created database. To use this script, relocate to the Controller directory of the Bits_And_Books_DB repository on GitHub, and run the following terminal command:

```
sh gen_new_seed.sh filename.sqlite
```

The full code for bb_parser.rb and gen_new_seed.sh can be found in Appendices C.2 and C.3 respectively.

All of this code can also be found in the GitHub for this project, linked below

3.3 GitHub Repository

Link to GitHub Repository: https://github.com/CPort28/Bits_And_Books_DB

APPENDIX A : Checkpoint Documents

A1.1 – Checkpoint 1: Version 1

Date: 1/25/2023

1. Based on the requirements given in the project overview, list the entities to be modeled in this database. For each entity, provide a list of associated attributes.
    - BOOK
        - ISBN (Primary Key)
        - Title
        - Release_Year
        - Sales_Price
        - Current_Stock (Derived)
        - Number_of_Copies_Sold (Derived)
    - CATEGORY
        - Category_Id (Primary Key)
        - Category_Name
        - Description
    - AUTHOR
        - Author_Id (Primary Key)
        - Author_First_Name
        - Author_Last_Name
    - PUBLISHER
        - Publisher_Id (Primary Key)
        - Publisher_Name
        - Publisher_Contact_Info
    - CUSTOMER
        - Customer_Id (Primary Key)
        - Email
        - Phone_Number
        - First_Name
        - Last_Name
        - Number_Books_Purchased (Derived)
        - Address
        - Total_Spent (Derived)
        - Password_Digest
    - ORDER
        - Order_ID (Primary Key)
        - Order_Date
        - Total_Price
    - BOOK_ORDER
        - Order_Id (Foreign Key)
        - Book_ISBN (Foreign Key)
        - Quantity

- PAYMENT
    - Card_Type
    - Bill_Number (Primary Key)
- EMPLOYEE
    - Employee_Id (Primary Key)
    - First_Name
    - Last_Name
    - Phone_Number
    - Email
    - Address
    - Password_Digest
- ACCESS_LEVEL
    - Level (Primary Key)
    - Permissions

2. Based on the requirements given in the project overview, what are the various relationships between entities? (For example, "CUSTOMER entities purchase BOOK entities").
    - AUTHOR entities write BOOK entities
    - PUBLISHER entities publish BOOK entities
    - STAFF entities have an ACCESS_LEVEL entity
    - CUSTOMER entities place ORDER entities
    - BOOK_ORDER entities belong to ORDER entities
    - BOOK_ORDER entities order BOOK entities
    - PAYMENT entities belong to ORDER entities
    - BOOK entities belong to a CATEGORY entity

3. Propose at least two additional entities that it would be useful for this database to model beyond the scope of the project requirements. Provide a list of possible attributes for the additional entities and possible relationships they may have with each other and the rest of the entities in the database. Give a brief, one sentence rationale for why adding these entities would be interesting/useful to the stakeholders for this database project.

    - WAREHOUSE:
        - **Attributes:**
            - Warehouse_Id (Primary Key)
            - Address
            - City
            - State
            - Country
            - Total_Capacity
        - **Relationships:**

        - WAREHOUSE entities have many WAREHOUSE_STOCK entities
        - WAREHOUSE entities staff EMPLOYEE entities
        - BOOK_ORDER entities are assigned to WAREHOUSE entities
    - WAREHOUSE_STOCK
        - **Attributes:**
            - Quantity
        - **Relationships:**
            - WAREHOUSE_STOCK entities belong to WAREHOUSE entities
            - WAREHOUSE_STOCK entities stocks a BOOK entity

Having access to which books are stored where, allows you to ship books quicker based on customer location

It also can notify you if you are running low on any books at a certain time so that you can restock them.

4. Give at least four examples of some informal queries/reports that it might be useful for this database might be used to generate. Include one example for each of the additional entities you proposed in question 3 above.

    1. How many copies of book X were sold in January 2023?
    2. At which warehouse locations is book X available?
    3. List all stocked books in the category X
    4. What is the name of the customer who made the purchase with bill_number x?

5. Suppose we want to add a new publisher to the database. How would we do that given the entities and relationships you've outlined above? Given your above description, is it possible to add a new publisher to your database without knowing the title of any books they have published? If not, revise your model to allow for publishers to be added as separate entities.

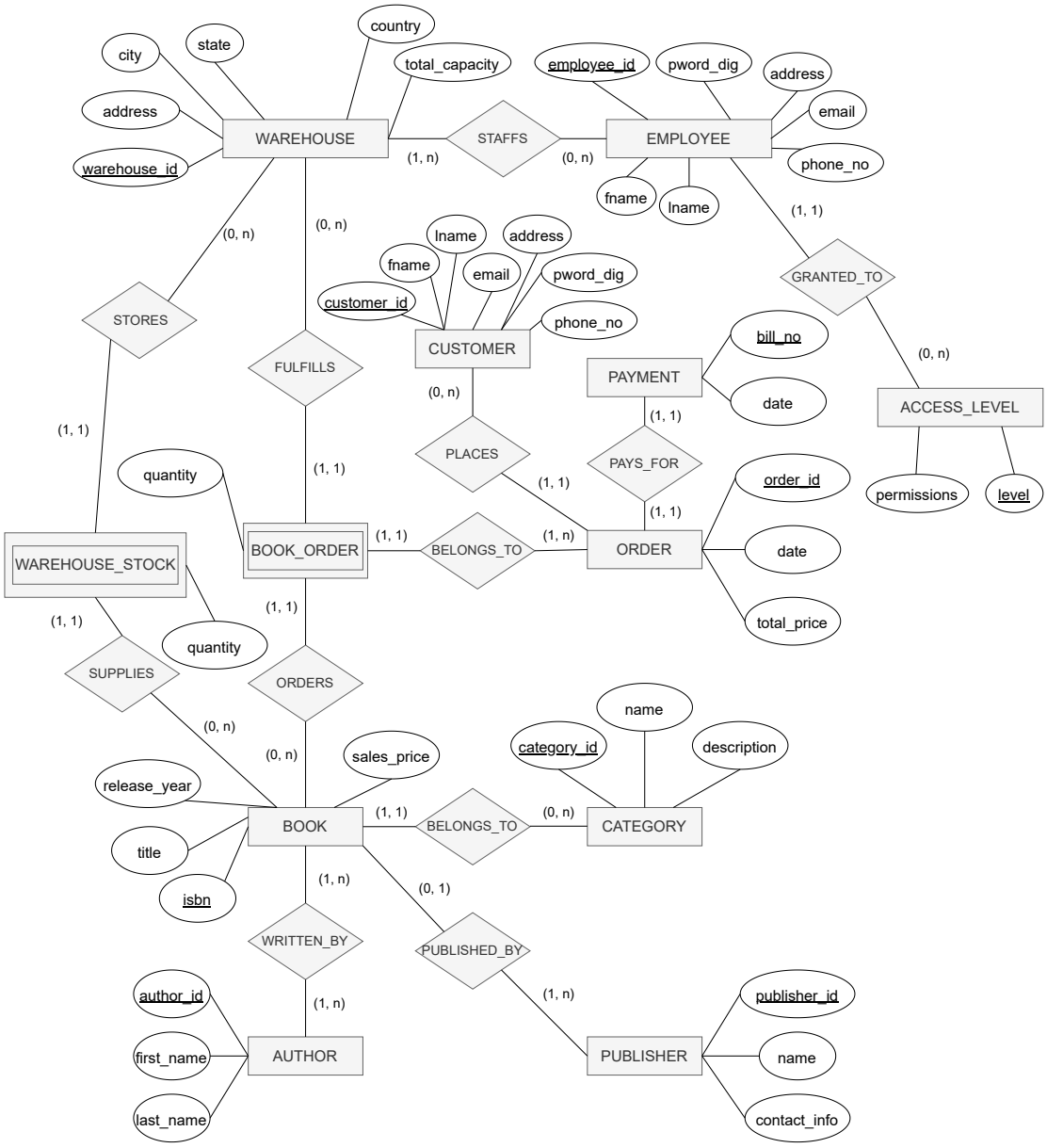
    You would add a new entry to the publisher table, providing the publisher's name and contact information (undecided format). It is possible to add a publisher without any of their books in our current model, as PUBLISHER entities can have many books, but are not required to have any.

6. Determine at least three other informal update operations and describe what entities would need to have attributes altered and how they would need to be changed given your above descriptions. Include one example for each of the additional entities you proposed in question 3 above.

    1. Adding a new entry to the order table. This would also require adding a new entry to the payment table, while also adding to the book_order table. The warehouse_stock quantity attribute will need to be updated according to how many books are purchased.

2. Adding a new entity to the warehouse table. This would require an update to the staff location, as at least one employee needs to be working at each warehouse. Warehouse stock would also need to be initialized (new entries to the warehouse stock table).
3. Update a book_order, such as changing the quantity of books or removing a book. The total price of the order would also need to be updated.

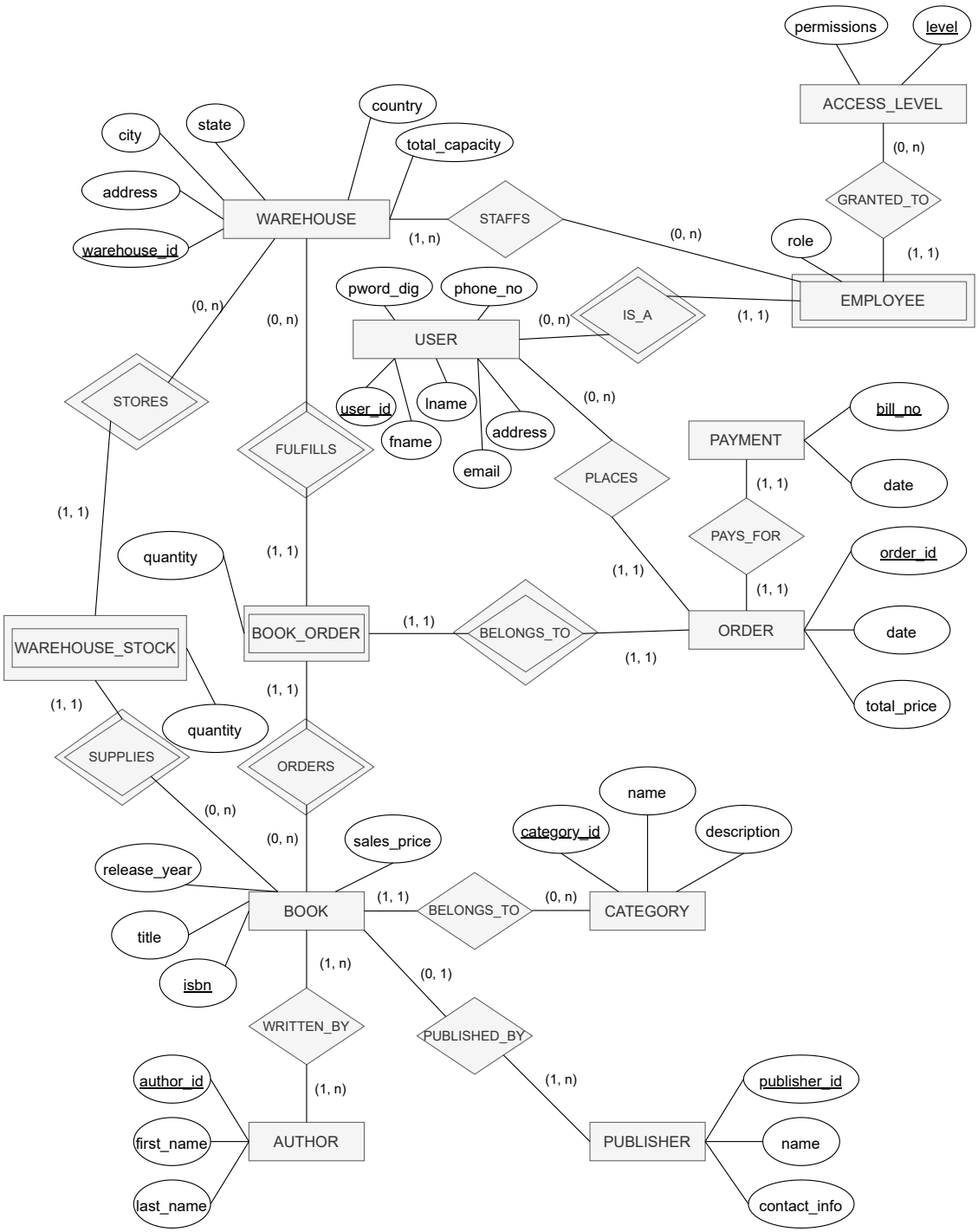
7. Provide an ER diagram for your database. Make sure you include all of the entities and relationships you determined in the questions above ***INCLUDING the entities for question 3 above***, and remember that ***EVERY*** entity in your model needs to connect to another entity in the model via some kind of relationship.



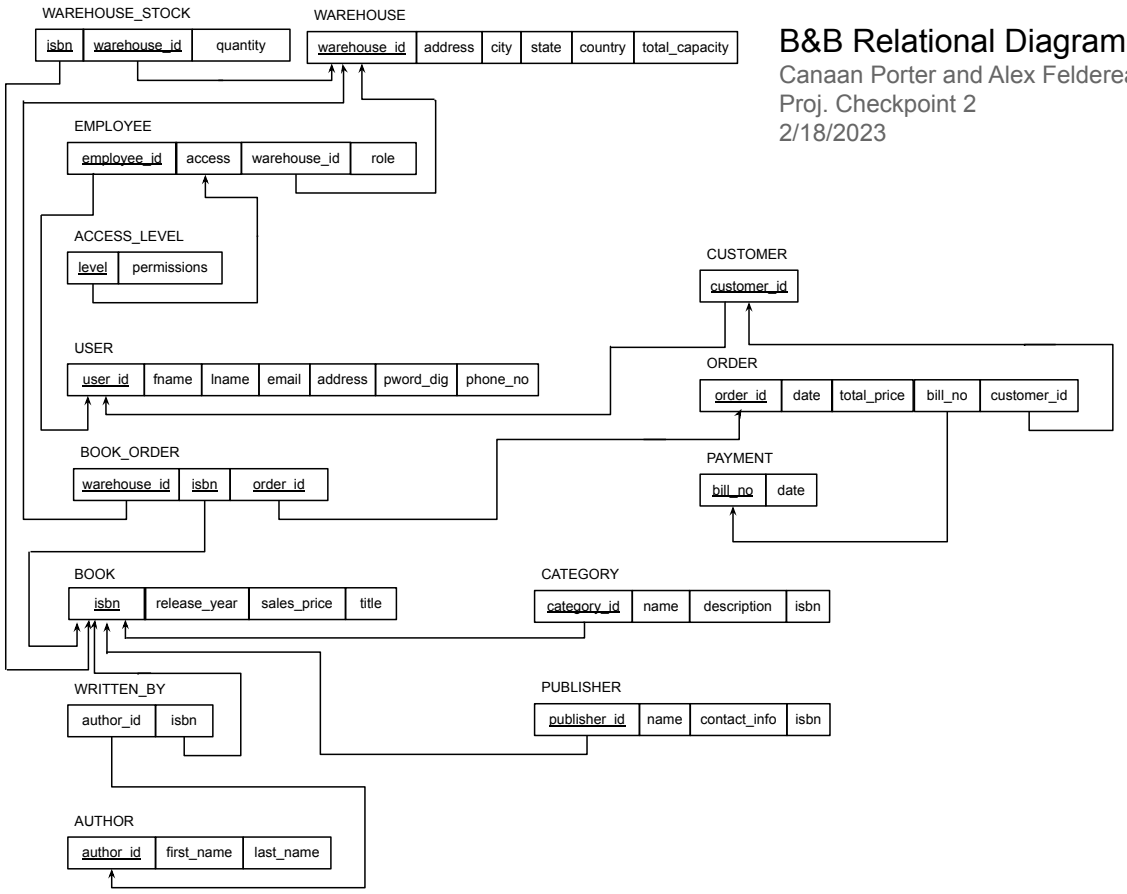
A2.1 – Checkpoint 2: Version 1

Date: 2/21/2023

1. Provide a current version of your ER Model as per Project Checkpoint 01. If you were instructed to change the model for Project Checkpoint 01, make sure you use the revised version of your ER Model.



2. Map your ER model to a relational schema. Indicate all primary and foreign keys.



3. Given your relational schema, provide the relational algebra to perform the following queries. If your schema cannot provide answers to these queries, revise your ER Model and your relational schema to contain the appropriate information for these queries:

   a. Find the titles of all books by Pratchett that cost less than $10:

   π_{title} (

   $\sigma_{\text{last\_name = 'Pratchett'} \wedge \text{sales\_price} < 10}$ (

   BOOK * WRITTEN_BY (

   WRITTEN_BY * AUTHOR)))

b. Give all the titles and their dates of purchases made by a single customer (you choose how to designate the customer): Customer identified by customer_id, "x"

$\pi_{title, date}$ (
$\sigma_{user\_id = x}$ (
USER * (
ORDER * (
BOOK_ORDER * BOOK))))

c. Find the titles and ISBNs for all books with less than 5 copies in stock:

ALL_WAREHOUSE_STOCK ← WAREHOUSE * WAREHOUSE_STOCK
BOOK_TOTAL_STOCK ← $_{isbn}$ **F** $_{SUM\ quantity}$ (ALL_WAREHOUSE_STOCK)
BOOKS_IN_STOCK ← BOOK_TOTAL_STOCK * BOOK
BOOKS_LOW_STOCK ← $\sigma_{Sum\_quantity < 5}$ (BOOKS_IN_STOCK)
RESULT ← $\pi_{title, isbn}$ (BOOKS_LOW_STOCK)

d. Give all the customers who purchased a book by Pratchett and the titles of Pratchett books they purchased

BOOKS_BY_PRATCHETT ← $\pi_{isbn, title}$ (
$\sigma_{last\_name = 'Pratchett'}$ (BOOK * (WRITTEN_BY * AUTHOR)))
RESULT ← $\pi_{first\_name, last\_name, title}$ (
USER $\bowtie_{user\_id = customer\_id}$ (
BOOK_ORDER * BOOKS_BY_PRATCHETT))

e. Find the total number of books purchased by a single customer (you choose how to designate the customer): Customer identified by user_id, "x"

ORDER_USER ← ORDER $\bowtie_{customer\_id = user\_id}$ (USER)
BOOK_ORDER_USER ← BOOK_ORDER * ORDER_USER
BOOKS_ORDERED_BY_USER ← $_{user\_id}$ **F** $_{SUM\ quantity}$
(BOOK_ORDER_USER)
RESULT ← $\pi_{Sum\_quantity}$ ($\sigma_{user\_id = x}$ (BOOKS_ORDERED_BY_USER))

f. Find the customer who has purchased the most books and the total number of books they have purchased:

ORDER_USER ← ORDER $\bowtie_{customer\_id = user\_id}$ USER

$$BOOK\_ORDER\_USER \leftarrow BOOK\_ORDER * ORDER\_USER$$
$$BOOKS\_ORDERED\_BY\_USER \leftarrow {}_{user\_id}\mathcal{F}_{SUM\ quantity}(BOOK\_ORDER\_USER)$$
$$MAX \leftarrow \mathcal{F}_{MAX\ Sum\_quantity}(BOOKS\_ORDERED\_BY\_USER)$$
$$RESULT \leftarrow \rho_{(fname, lname, books\_ordered)}(\pi_{first\_name, last\_name, Max\_Sum\_quantity}(BOOKS\_ORDERED\_BY\_USER \bowtie_{Sum\_quantity = Max\_Sum\_quantity} (MAX)))$$

4. Come up with three additional interesting queries that your database can provide. Give what the queries are supposed to retrieve in plain English and then as relational algebra. Your queries should include joins and at least one should include an aggregate function. At least one of your queries should use "extra" entities you added to your model in Checkpoint 01.

1. Find the author that has sold the most books through B&B

$$TOTAL\_SALES\_BY\_BOOK \leftarrow {}_{isbn}\mathcal{F}_{SUM\ quantity}(BOOK\_ORDER)$$
$$\rho_{(isbn, copies\_sold)}(TOTAL\_SALES\_BY\_BOOK)$$
$$BOOKS\_BY\_AUTHOR \leftarrow (WRITTEN\_BY * AUTHOR)$$
$$SALES\_BY\_AUTHOR\_BY\_BOOK \leftarrow (TOTAL\_SALES\_BY\_BOOK * BOOKS\_BY\_AUTHOR)$$
$$TOTAL\_AUTHOR\_SALES \leftarrow {}_{author\_id}\mathcal{F}_{SUM\ copies\_sold}(SALES\_BY\_AUTHOR)$$
$$MAX\_AUTHOR\_SALES \leftarrow \mathcal{F}_{MAX\ Sum\_copies\_sold}(TOTAL\_AUTHOR\_SALES)$$

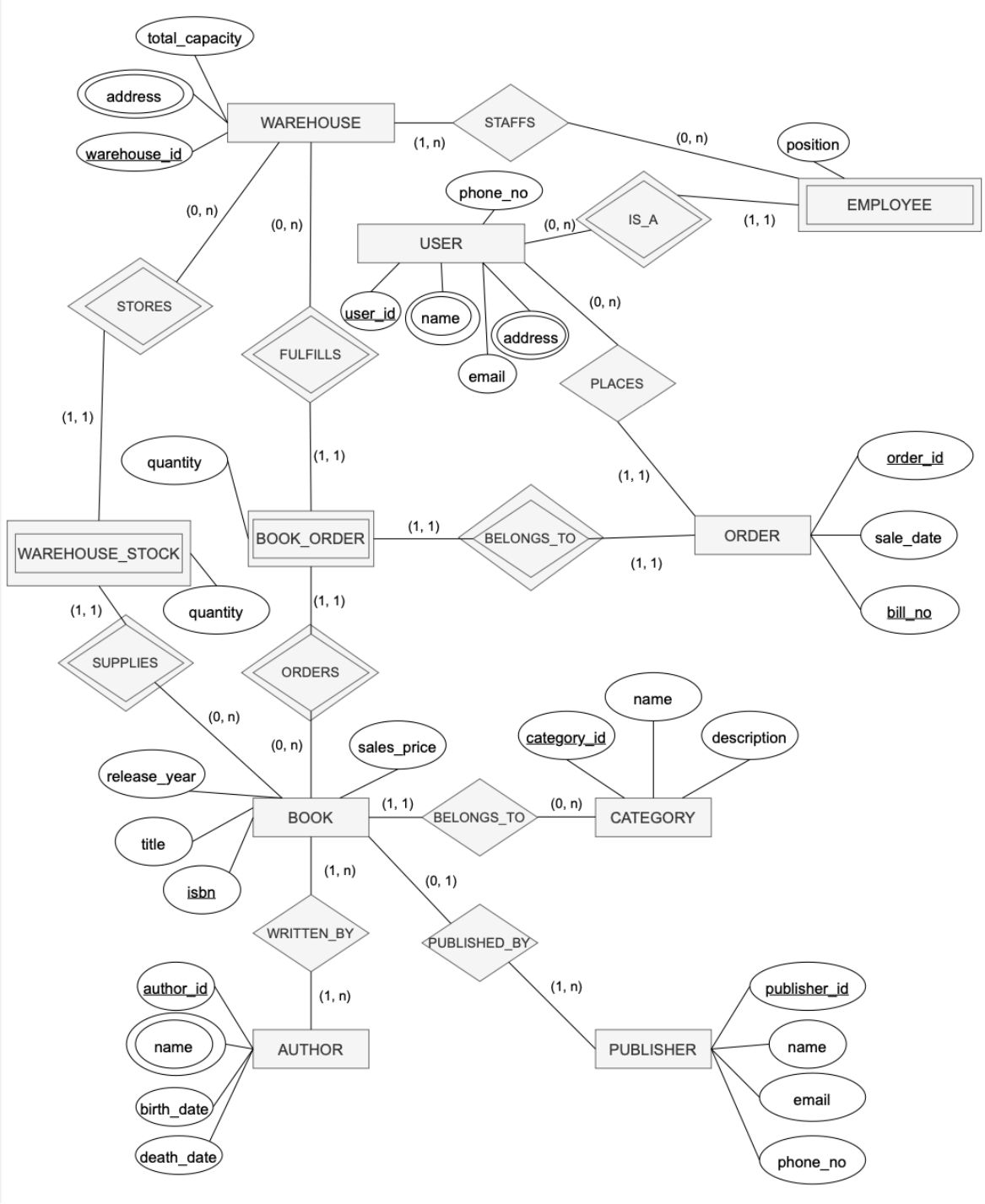
2. Check which warehouses have enough stock to order 10 copies of a book (given the isbn, "x")

$$TOTAL\_STOCK\_BY\_WAREHOUSE \leftarrow WAREHOUSE * WAREHOUSE\_STOCK$$
$$\sigma_{isbn = x \wedge quantity \geq 10}(TOTAL\_STOCK\_BY\_WAREHOUSE)$$

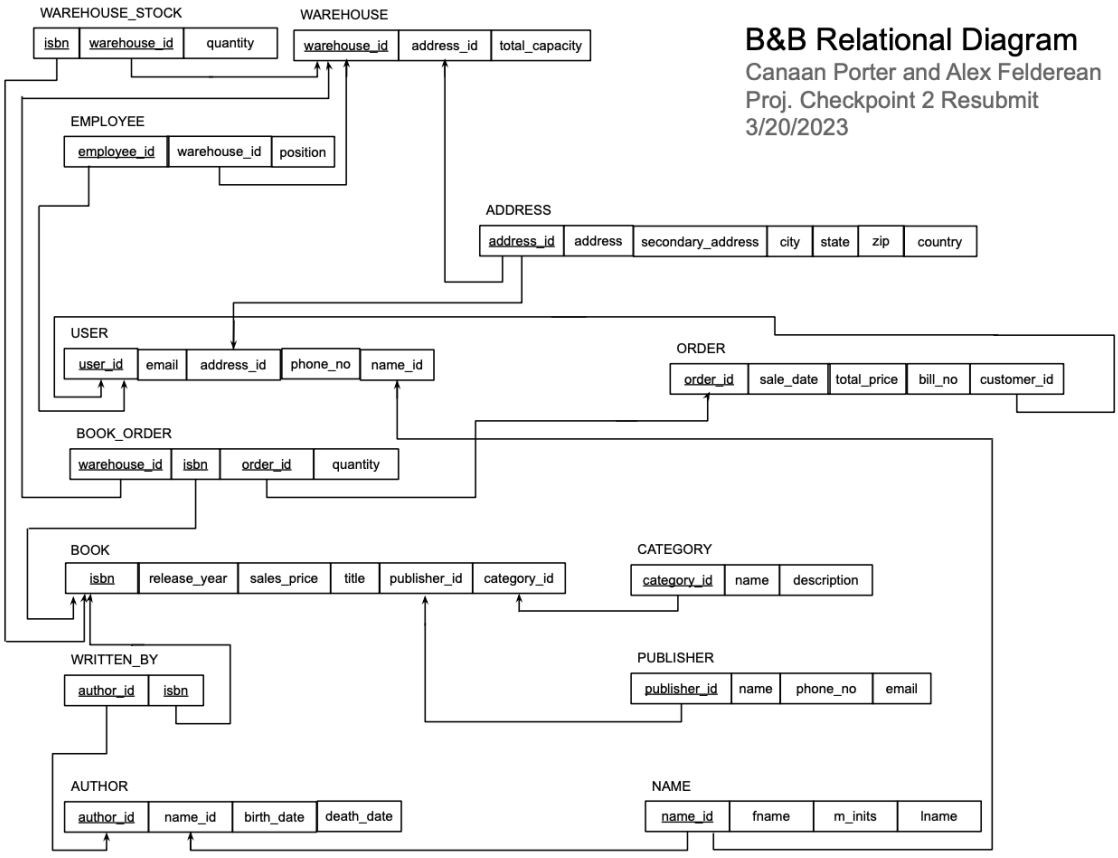
A2.2 – Checkpoint 2: Version 2

Date: 3/20/2023

1. Provide a current version of your ER Model as per Project Checkpoint 01. If you were instructed to change the model for Project Checkpoint 01, make sure you use the revised version of your ER Model.

2. 

3. Map your ER model to a relational schema. Indicate all primary and foreign keys.



4. Given your relational schema, provide the relational algebra to perform the following queries. If your schema cannot provide answers to these queries, revise your ER Model and your relational schema to contain the appropriate information for these queries:
   a. Find the titles of all books by Pratchett that cost less than $10:

$$\pi_{\text{title}} (\sigma_{\text{last\_name} = \text{'Pratchett'} \wedge \text{sales\_price} < 10} (\text{BOOK} * \text{WRITTEN\_BY} (\text{WRITTEN\_BY} * \text{AUTHOR})))$$

b. Give all the titles and their dates of purchases made by a single customer (you choose how to designate the customer): Customer identified by customer_id, "x"

$\pi_{\text{title, date}}$ (
$\sigma_{\text{user\_id = x}}$ (
USER * (
ORDER * (
BOOK_ORDER * BOOK))))

c. Find the titles and ISBNs for all books with less than 5 copies in stock:

ALL_WAREHOUSE_STOCK ← WAREHOUSE * WAREHOUSE_STOCK
BOOK_TOTAL_STOCK ← $_{\text{isbn}}$ **F** $_{\text{SUM quantity}}$ (ALL_WAREHOUSE_STOCK)
BOOKS_IN_STOCK ← BOOK_TOTAL_STOCK * BOOK
BOOKS_LOW_STOCK ← $\sigma_{\text{Sum\_quantity < 5}}$ (BOOKS_IN_STOCK)
RESULT ← $\pi_{\text{title, isbn}}$ (BOOKS_LOW_STOCK)

d. Give all the customers who purchased a book by Pratchett and the titles of Pratchett books they purchased

BOOKS_BY_PRATCHETT ← $\pi_{\text{isbn, title}}$ (
$\sigma_{\text{last\_name = 'Pratchett'}}$ (BOOK * (WRITTEN_BY * AUTHOR)))
RESULT ← $\pi_{\text{first\_name, last\_name, title}}$ (
USER $\bowtie_{\text{user\_id = customer\_id}}$ (
BOOK_ORDER * BOOKS_BY_PRATCHETT))

e. Find the total number of books purchased by a single customer (you choose how to designate the customer): Customer identified by user_id, "x"

ORDER_USER ← ORDER $\bowtie_{\text{customer\_id = user\_id}}$ (USER)
BOOK_ORDER_USER ← BOOK_ORDER * ORDER_USER
BOOKS_ORDERED_BY_USER ← $_{\text{user\_id}}$ **F** $_{\text{SUM quantity}}$ (BOOK_ORDER_USER)
RESULT ← $\pi_{\text{Sum\_quantity}}$ ($\sigma_{\text{user\_id = x}}$ (BOOKS_ORDERED_BY_USER))

f. Find the customer who has purchased the most books and the total number of books they have purchased:

$$ORDER\_USER \leftarrow ORDER \bowtie_{customer\_id = user\_id} USER$$

$$BOOK\_ORDER\_USER \leftarrow BOOK\_ORDER * ORDER\_USER$$

$$BOOKS\_ORDERED\_BY\_USER \leftarrow {}_{user\_id}\mathcal{F}_{SUM\ quantity}(BOOK\_ORDER\_USER)$$

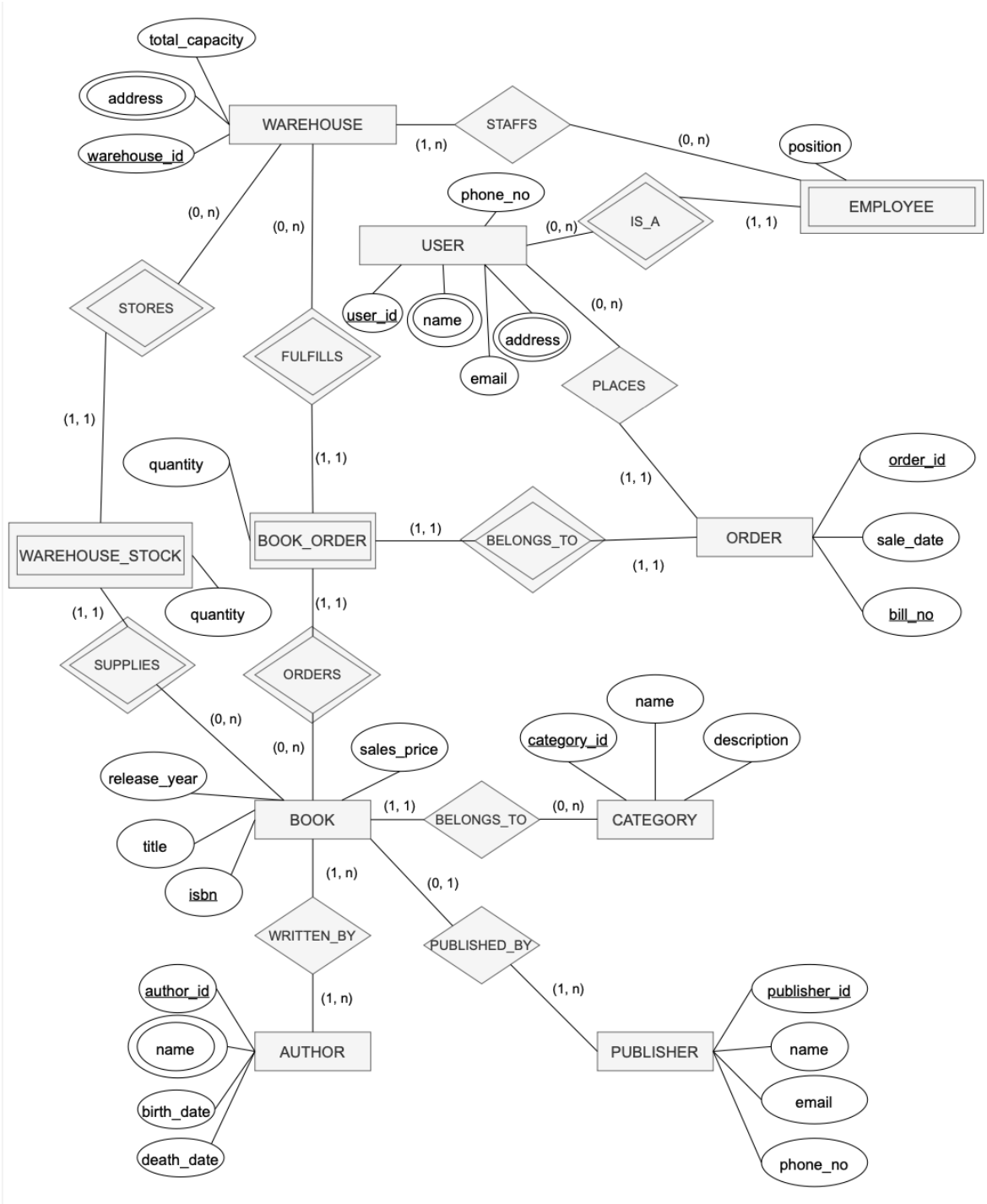
$$MAX \leftarrow \mathbf{F}_{MAX\ Sum\_quantity}(BOOKS\_ORDERED\_BY\_USER)$$

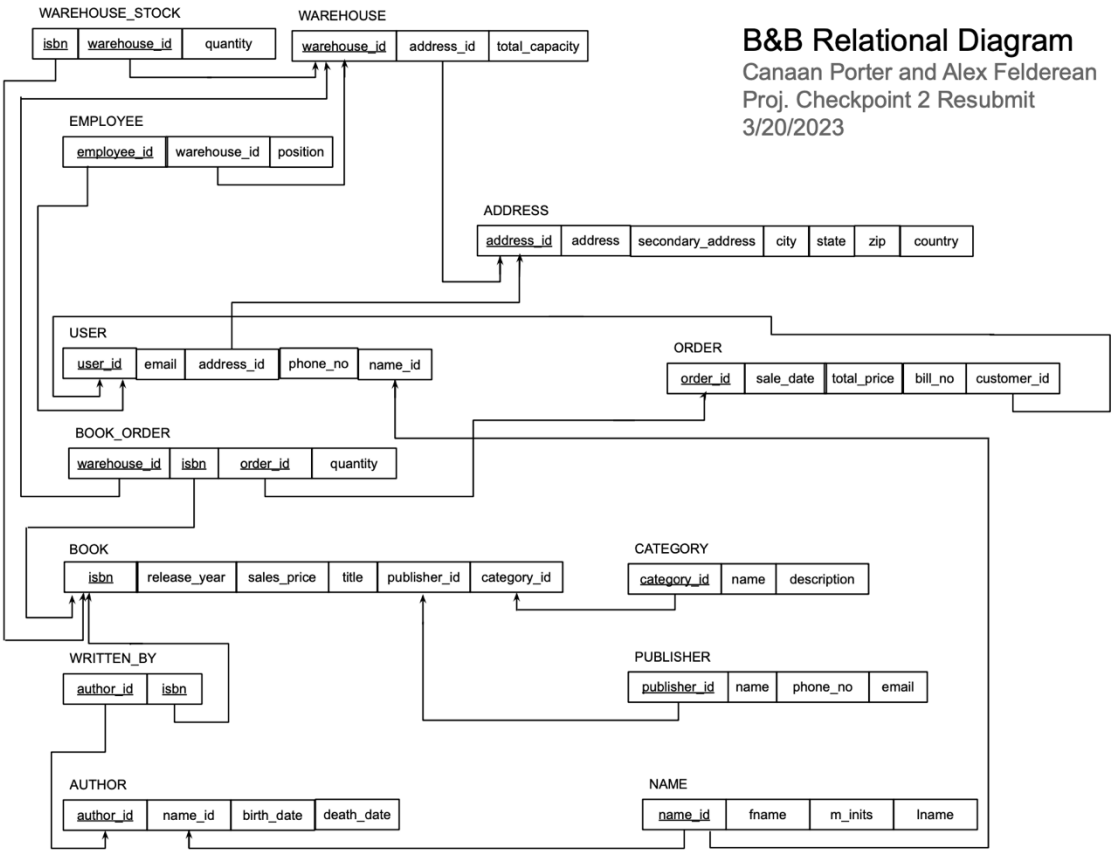
$$RESULT \leftarrow \rho_{(fname, lname, books\_ordered)}(\pi_{first\_name, last\_name, Max\_Sum\_quantity}(BOOKS\_ORDERED\_BY\_USER \bowtie_{Sum\_quantity = Max\_Sum\_quantity} (MAX)))$$

A3.1 – Checkpoint 3: Version 1

Date: 3/27/2023

1. Provide a current version of your ER Diagram and Relational Model as per Project Checkpoint 02. **If you were instructed to change the model for Project Checkpoint 02, make sure you use the revised versions of your models**





2. Given your relational schema, create a text file containing the SQL code to create your database schema. Use this SQL to create a database in SQLite. Populate this database with the data provided for the project as well as 20 sample records for each table that does not contain data provided in the original project documents.

```
CREATE TABLE CATEGORY (
    category_id INTEGER NOT NULL PRIMARY KEY,
    cat_name VARCHAR(20) NOT NULL,
    description VARCHAR(255)
);

CREATE TABLE PUBLISHER (
    publisher_id INTEGER NOT NULL PRIMARY KEY,
    pub_name VARCHAR(50) NOT NULL,
    phone_no INTEGER(10) NOT NULL,
    email VARCHAR(30) NOT NULL
);
```

```
CREATE TABLE BOOK (
    isbn VARCHAR(13) NOT NULL PRIMARY KEY,
    release_year INTEGER(4) NOT NULL,
    sales_price DECIMAL(10,2) NOT NULL,
    title VARCHAR(255) NOT NULL,
    publisher_id INTEGER NOT NULL,
    category_id INTEGER NOT NULL,
    FOREIGN KEY (publisher_id) REFERENCES PUBLISHER(publisher_id),
    FOREIGN KEY (category_id) REFERENCES CATEGORY(category_id)
);

CREATE TABLE AUTHOR (
    author_id INTEGER NOT NULL PRIMARY KEY,
    name_id INTEGER NOT NULL,
    birth_date DATE,
    death_date DATE
    FOREIGN KEY (name_id) REFERENCES NAME(name_id)
);

CREATE TABLE NAME (
    name_id INTEGER NOT NULL PRIMARY KEY,
    fname VARCHAR(50) NOT NULL,
    lname VARCHAR(50) NOT NULL,
    middle_inits VARCHAR(10)
)

CREATE TABLE WRITTEN_BY (
    author_id INTEGER NOT NULL,
    isbn VARCHAR(13) NOT NULL,
    FOREIGN KEY(author_id) REFERENCES AUTHOR(author_id),
    FOREIGN KEY(isbn) REFERENCES BOOK(isbn)
);

CREATE TABLE USER (
    user_id INTEGER NOT NULL PRIMARY KEY,
    name_id INTEGER NOT NULL,
    email VARCHAR(255) NOT NULL,
    phone_no INTEGER(10),
    address_id INTEGER NOT NULL,
    FOREIGN KEY(name_id) REFERENCES NAME(name_id)
    FOREIGN KEY(address_id) REFERENCES ADDRESS(address_id)
);
```

```
CREATE TABLE ADDRESS (
    address_id INTEGER NOT NULL PRIMARY KEY,
    address VARCHAR(100) NOT NULL,
    secondary_address VARCHAR(50),
    city VARCHAR(30) NOT NULL,
    state VARCHAR(2) NOT NULL,
    zip VARCHAR(10) NOT NULL,
    country VARCHAR(3) NOT NULL
)

CREATE TABLE "ORDER" (
    order_id INTEGER NOT NULL PRIMARY KEY,
    sale_date DATE NOT NULL,
    bill_no INTEGER NOT NULL,
    customer_id INTEGER NOT NULL,
    FOREIGN KEY (customer_id) REFERENCES USER (user_id)
);

CREATE TABLE WAREHOUSE (
    warehouse_id INTEGER NOT NULL PRIMARY KEY,
    address_id INTEGER NOT NULL,
    total_capacity INTEGER NOT NULL
    FOREIGN KEY(address_id) REFERENCES ADDRESS(address_id)
);

CREATE TABLE BOOK_ORDER (
    warehouse_id INTEGER NOT NULL,
    isbn VARCHAR(13) NOT NULL,
    order_id INTEGER NOT NULL,
    quantity INTEGER NOT NULL,
    FOREIGN KEY (warehouse_id) REFERENCES WAREHOUSE (warehouse_id),
    FOREIGN KEY (isbn) REFERENCES BOOK (isbn),
    FOREIGN KEY (order_id) REFERENCES "ORDER" (order_id)
);

CREATE TABLE WAREHOUSE_STOCK (
    isbn INTEGER(13) NOT NULL,
    warehouse_id INTEGER NOT NULL,
    quantity INTEGER,
    FOREIGN KEY (isbn) REFERENCES BOOK (isbn),
    FOREIGN KEY (warehouse_id) REFERENCES WAREHOUSE (warehouse_id)
);

CREATE TABLE EMPLOYEE (
    employee_id INTEGER NOT NULL,
    warehouse_id INTEGER,
    position VARCHAR(25) NOT NULL,
    FOREIGN KEY (employee_id) REFERENCES USER(user_id),
    FOREIGN KEY (warehouse_id) REFERENCES WAREHOUSE(warehouse_id)
);
```

3. Given your relational schema, provide the SQL to perform the following queries. If your schema cannot provide answers to these queries, revise your ER Model and your relational schema to contain the appropriate information for these queries.

```
-- a.    Find the titles of all books by Pratchett that cost less than $10
SELECT B.title
FROM BOOK B
JOIN WRITTEN_BY WB on B.isbn = WB.isbn
JOIN AUTHOR A on A.author_id = WB.author_id
JOIN NAME N on A.name_id = N.name_id
WHERE N.lname = 'Pratchett' AND B.sales_price < 10;

-------------------------------------------------------------------------------

-- b.    Give all the titles and their dates of purchase
-- made by a single customer (you choose how to designate the customer)
(USER_ID = 50)
SELECT B.title, O.sale_date
FROM BOOK B
JOIN BOOK_ORDER BO on B.isbn = BO.isbn
JOIN "ORDER" O ON O.order_id = BO.order_id
JOIN USER U on O.customer_id = U.user_id
WHERE U.user_id = 50;

-------------------------------------------------------------------------------

-- c.    Find the titles and ISBNs for all books with less than 5 copies
in stock
SELECT B.title, B.isbn, sum(WS.quantity) AS stock
FROM BOOK B
JOIN WAREHOUSE_STOCK WS on B.isbn = WS.isbn
GROUP BY WS.isbn
HAVING sum(WS.quantity) < 5;

-------------------------------------------------------------------------------

-- d.    Give all the customers who purchased a book by Pratchett
-- and the titles of Pratchett books they purchased
SELECT NUser.fname, NUser.lname, B.title
FROM NAME NUser
JOIN USER U on NUser.name_id = U.name_id
JOIN "ORDER" on U.user_id = "ORDER".customer_id
JOIN BOOK_ORDER BO on "ORDER".order_id = BO.order_id
JOIN BOOK B on BO.isbn = B.isbn
JOIN WRITTEN_BY WB on B.isbn = WB.isbn
JOIN AUTHOR A on WB.author_id = A.author_id
JOIN NAME NAuth on A.name_id = NAuth.name_id
WHERE NAuth.lname = 'Pratchett';
```

```
-------------------------------------------------------------------------------


-- e. Find the total number of books purchased by a single customer
-- (you choose how to designate the customer) (USER_ID = 50)
SELECT U.user_id, N.fname, N.lname, sum(BO.quantity) as books_purchased
FROM USER U
JOIN NAME N on U.name_id = N.name_id
JOIN "ORDER" O on O.customer_id = U.user_id
JOIN BOOK_ORDER BO on O.order_id = BO.order_id
WHERE U.user_id = 50;

-------------------------------------------------------------------------------

-- f. Find the customer who has purchased the most books and
-- the total number of books they have purchased
SELECT U.user_id, N.fname, N.lname, sum(BO.quantity) as books_purchased
FROM USER U
JOIN NAME N on U.name_id = N.name_id
JOIN "ORDER" O on O.customer_id = U.user_id
JOIN BOOK_ORDER BO on O.order_id = BO.order_id
GROUP BY U.user_id, N.fname, N.lname
HAVING books_purchased = (SELECT max(total_books)
                          FROM (SELECT sum(B.quantity) as total_books
                                FROM USER U2
                                JOIN "ORDER" ON U2.user_id = "ORDER".customer_id
                                JOIN BOOK_ORDER B on "ORDER".order_id = B.order_id
                                GROUP BY user_id));
```

4. For Project Checkpoint 02, you were asked to come up with three additional interesting queries that your database can provide. Give what those queries are supposed to retrieve in plain English, as relational algebra and then as SQL. Your queries should include joins and at least one should include an aggregate function, and they should be the same as the queries you outlined for Worksheet 02. If you were instructed to fix the queries in Checkpoint 02, make sure you use the fixed queries here.

```
-- Query 1:
  -- Find the author that has sold the most books through B&B

  -- Relational Algebra:
    -- TOTAL_SALES_BY_BOOK ← isbn F SUM quantity (BOOK_ORDER)
    -- ρ (isbn, copies_sold) (TOTAL_SALES_BY_BOOK)
    -- BOOKS_BY_AUTHOR ← (WRITTEN_BY ∗ AUTHOR)
    -- SALES_BY_AUTHOR_BY_BOOK ← (TOTAL_SALES_BY_BOOK ∗ BOOKS_BY_AUTHOR)
    -- TOTAL_AUTHOR_SALES ← author_id F SUM copies_sold (SALES_BY_AUTHOR)
    -- MAX_AUTHOR_SALES ← F MAX Sum_copies_sold (TOTAL_AUTHOR_SALES)
```

```
  -- SQL:
    FROM AUTHOR A
    JOIN NAME AName on A.name_id = AName.name_id
    JOIN WRITTEN_BY WB on A.author_id = WB.author_id
    JOIN BOOK B on WB.isbn = B.isbn
    JOIN BOOK_ORDER BO on B.isbn = BO.isbn
    GROUP BY A.author_id
    HAVING books_sold = (SELECT max(b_sold)
                         FROM (SELECT sum(BO.quantity) as b_sold
                               FROM AUTHOR A
                               JOIN WRITTEN_BY WB on A.author_id = WB.author_id
                               JOIN BOOK B on WB.isbn = B.isbn
                               JOIN BOOK_ORDER BO on B.isbn = BO.isbn
                               GROUP BY A.author_id));

-------------------------------------------------------------------------------------------

-- Query 2:
  -- Check which warehouses have enough stock to order 40 copies
  -- of a book(given the isbn, “x”)

  -- Relational Algebra:
    -- TOTAL_STOCK_BY_WAREHOUSE ← WAREHOUSE ∗ WAREHOUSE_STOCK
    -- σ isbn = x ∧ quantity ≥ 40 (TOTAL_STOCK_BY_WAREHOUSE)

  -- SQL:
    -- Book specified by isbn 596004478 in this example
    SELECT B.isbn, WAdd.address as warehouse_addr, WAdd.city, WAdd.state, WS.quantity
as stock
    FROM BOOK B
    JOIN WAREHOUSE_STOCK WS on B.isbn = WS.isbn
    JOIN WAREHOUSE W on WS.warehouse_id = W.warehouse_id
    JOIN ADDRESS WAdd on W.address_id = WAdd.address_id
    WHERE B.isbn = 596004478 AND WS.quantity >= 40;

-------------------------------------------------------------------------------------------

-- Query 3:
  -- Determine the total revenue of the bookstore

  -- Relational Algebra:
    -- ORDER_USER ← ORDER ⋈ customer_id = user_id USER
    -- BOOK_ORDER_USER ← BOOK_ORDER ∗ ORDER_USER
    -- RESULT ← F SUM total_price (BOOK_ORDER_USER)

  -- SQL:
    SELECT  sum(B.sales_price ∗ BO.quantity) as total_sales
    FROM BOOK B
    JOIN BOOK_ORDER BO on B.isbn = BO.isbn
    JOIN "ORDER" O on BO.order_id = O.order_id;
```

5. Given your relational schema, provide the SQL for the following more advanced queries. These queries may require you to use techniques such as nesting, aggregation using having clauses, and other techniques . If your database schema does not contain the information to answer to these queries, revise your ER Model and your relational schema to contain the appropriate information for these queries. **Note that if your database does contain the information but in non-aggregated form, you should NOT revise your model but instead figure out how to aggregate it for the query!**

```
-- a. Provide a list of customer names, along with the total dollar amount
each customer has spent.

SELECT N.fname, N.lname, sum(B.sales_price * BO.quantity) as total_spent
FROM BOOK B
JOIN BOOK_ORDER BO on B.isbn = BO.isbn
JOIN "ORDER" O on BO.order_id = O.order_id
JOIN USER U on O.customer_id = U.user_id
JOIN NAME N on U.name_id = N.name_id
GROUP BY U.user_id;
```

```
-- b. Provide a list of customer names and e-mail addresses for customers who have spent more
than the average customer.

SELECT N.fname, N.lname, U.email, sum(B.sales_price * BO.quantity) as total_spent
FROM BOOK B
JOIN BOOK_ORDER BO on B.isbn = BO.isbn
JOIN "ORDER" O on BO.order_id = O.order_id
JOIN USER U on O.customer_id = U.user_id
JOIN NAME N on U.name_id = N.name_id
GROUP BY U.user_id
HAVING total_spent > (SELECT avg(spent_per_c)
                      FROM ( SELECT sum(B.sales_price * BO.quantity) as spent_per_c
                             FROM BOOK B
                             JOIN BOOK_ORDER BO on B.isbn = BO.isbn
                             JOIN "ORDER" O on BO.order_id = O.order_id
                             JOIN USER U on O.customer_id = U.user_id
                             JOIN NAME N on U.name_id = N.name_id
                             GROUP BY U.user_id));
```

```
-- c. Provide a list of the titles in the database and associated total copies
sold to customers, sorted from the title that has sold the most individual
copies to the title that has sold the least.

SELECT B.title, sum(BO.quantity) as copies_sold
FROM BOOK B
JOIN BOOK_ORDER BO on B.isbn = BO.isbn
GROUP BY B.isbn
ORDER BY copies_sold DESC
```

```
-- d. Provide a list of the titles in the database and associated dollar
totals for copies sold to customers, sorted from the title that has sold the
highest dollar amount to the title that has sold the smallest.

SELECT B.title, sum(BO.quantity * B.sales_price) as dollar_sold_total
FROM BOOK B
JOIN BOOK_ORDER BO on B.isbn = BO.isbn
GROUP BY B.isbn
ORDER BY dollar_sold_total DESC;

-------------------------------------------------------------------------

-- e. Find the most popular author in the database
-- (i.e., the one who has sold the most books)

SELECT AName.fname, AName.middle_inits, AName.lname, sum(BO.quantity) as books_sold
FROM AUTHOR A
JOIN NAME AName on A.name_id = AName.name_id
JOIN WRITTEN_BY WB on A.author_id = WB.author_id
JOIN BOOK B on WB.isbn = B.isbn
JOIN BOOK_ORDER BO on B.isbn = BO.isbn
GROUP BY A.author_id
HAVING books_sold = (SELECT max(b_sold)
                     FROM (SELECT sum(BO.quantity) as b_sold
                           FROM AUTHOR A
                           JOIN WRITTEN_BY WB on A.author_id = WB.author_id
                           JOIN BOOK B on WB.isbn = B.isbn
                           JOIN BOOK_ORDER BO on B.isbn = BO.isbn
                           GROUP BY A.author_id));

-------------------------------------------------------------------------

-- f. Find the most profitable author in the database for this store
-- (i.e. the one who has brought in the most money)

SELECT AName.fname, AName.middle_inits, AName.lname, sum(BO.quantity * B.sales_price)
as sales_total
FROM AUTHOR A
JOIN NAME AName on A.name_id = AName.name_id
JOIN WRITTEN_BY WB on A.author_id = WB.author_id
JOIN BOOK B on WB.isbn = B.isbn
JOIN BOOK_ORDER BO on B.isbn = BO.isbn
GROUP BY A.author_id
HAVING sales_total = (SELECT max(b_sold)
                     FROM (SELECT sum(BO.quantity * B.sales_price) as b_sold
                           FROM AUTHOR A
                           JOIN WRITTEN_BY WB on A.author_id = WB.author_id
                           JOIN BOOK B on WB.isbn = B.isbn
                           JOIN BOOK_ORDER BO on B.isbn = BO.isbn
                           GROUP BY A.author_id));
```

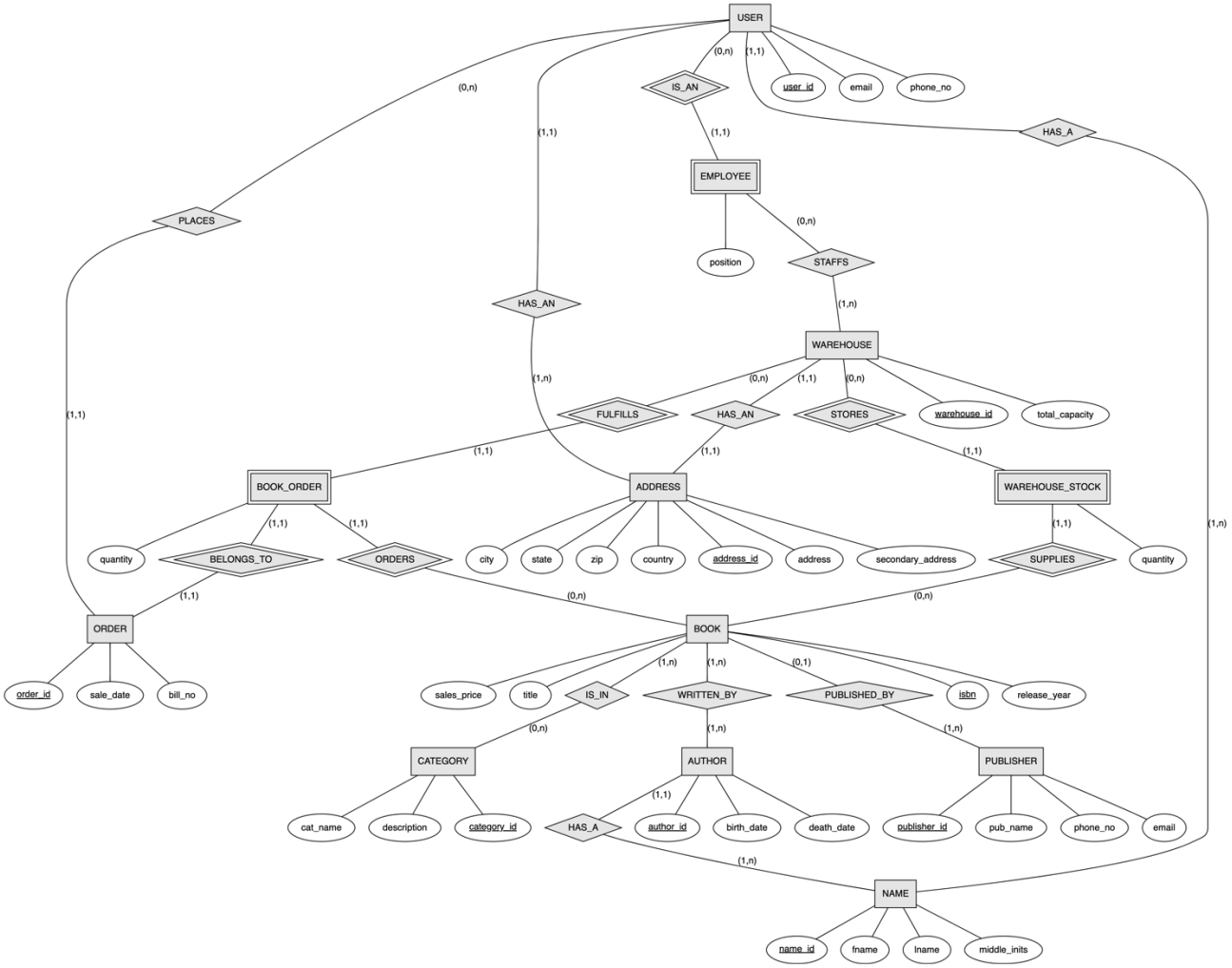
```
-- g. Provide a list of customer information for customers who
-- purchased anything written by the most profitable author in the database.

SELECT UName.fname, UName.lname, U.email, U.phone_no, Ad.address, Ad.city, Ad.state, Ad.country
FROM USER U
JOIN NAME UName on U.name_id = UName.name_id
JOIN ADDRESS Ad on U.address_id = Ad.address_id
JOIN "ORDER" O on U.user_id = O.customer_id
JOIN BOOK_ORDER BO on O.order_id = BO.order_id
JOIN BOOK B on BO.isbn = B.isbn
JOIN WRITTEN_BY WB on B.isbn = WB.isbn
JOIN AUTHOR A on WB.author_id = A.author_id
WHERE A.author_id = (SELECT A.author_id
                     FROM AUTHOR A
                     JOIN NAME AName on A.name_id = AName.name_id
                     JOIN WRITTEN_BY WB on A.author_id = WB.author_id
                     JOIN BOOK B on WB.isbn = B.isbn
                     JOIN BOOK_ORDER BO on B.isbn = BO.isbn
                     GROUP BY A.author_id
                     HAVING sum(BO.quantity * B.sales_price) = (SELECT max(b_sold)
                                  FROM (SELECT sum(BO.quantity * B.sales_price) as b_sold
                                        FROM AUTHOR A
                                        JOIN WRITTEN_BY WB on A.author_id = WB.author_id
                                        JOIN BOOK B on WB.isbn = B.isbn
                                        JOIN BOOK_ORDER BO on B.isbn = BO.isbn
                                        GROUP BY A.author_id)))
GROUP BY U.user_id;

-------------------------------------------------------------------------

-- h. Provide the list of authors who wrote the books purchased by the
customers who have spent more than the average customer.

SELECT AuthName.fname, AuthName.middle_inits, AuthName.lname
FROM BOOK B
JOIN BOOK_ORDER BO on B.isbn = BO.isbn
JOIN WRITTEN_BY WB on B.isbn = WB.isbn
JOIN AUTHOR A on WB.author_id = A.author_id
JOIN NAME AuthName on A.name_id = AuthName.name_id
GROUP BY A.author_id
HAVING sum(B.sales_price * BO.quantity) > (SELECT avg(spent_per_c)
                        FROM ( SELECT sum(B.sales_price * BO.quantity) as spent_per_c
                               FROM BOOK B
                               JOIN BOOK_ORDER BO on B.isbn = BO.isbn
                               JOIN "ORDER" O on BO.order_id = O.order_id
                               JOIN USER U on O.customer_id = U.user_id
                               JOIN NAME N on U.name_id = N.name_id
                               GROUP BY U.user_id));
```

A3.1 – Checkpoint 4: Version 1

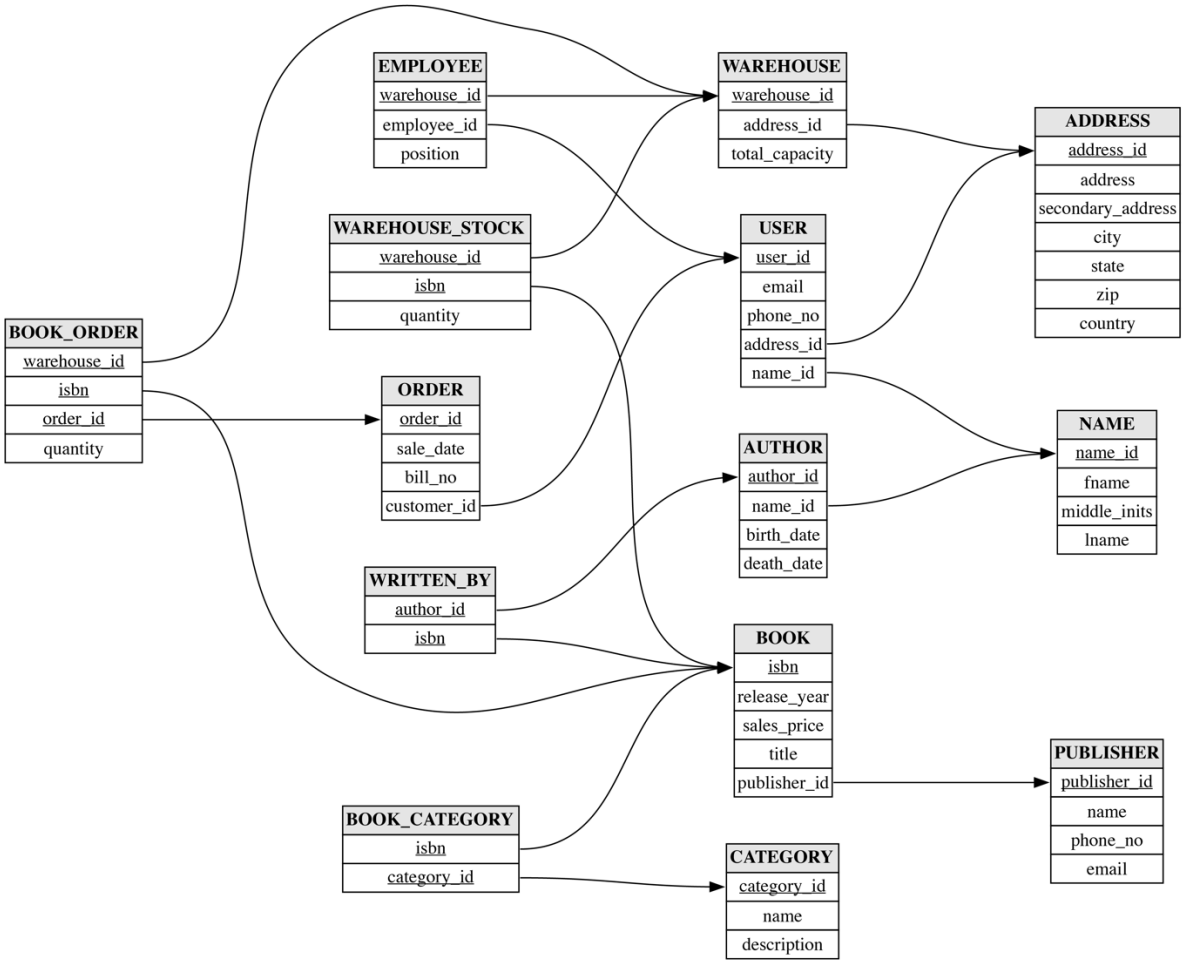
Date: 4/11/2023

1. Provide a current version of your ER Diagram and Relational Model as per Project Checkpoint 03. **If you were instructed to change the model for Project Checkpoint 03, make sure you use the revised versions of your models.**

ER Diagram:



Relational Model Diagram:



2. For each relation schema in your model, indicate the functional dependencies. Think carefully about what you are modeling here - make sure you consider all the possible dependencies in each relation and not just the ones from your primary keys. For example, a customer's credit card number is unique, and so will uniquely identify a customer even if you have another key in the same table (in fact, if the customer can have multiple credit card numbers, the dependencies can get even more involved).

   BOOK:

   {isbn} -> {release_year, sales_price, title, publisher_id}

   AUTHOR:

   {author_id} -> {name_id, birth_date, death_date}

NAME:

{name_id} -> {fname, middle_inits, lname}

CATEGORY:

{publisher_id} -> {name, phone_no, email}

WRITTEN_BY:

No functional dependencies (bridge table)

BOOK_CATEGORY:

No functional dependencies (bridge table)

BOOK_ORDER:

{warehouse_id, isbn, order_quantity} -> {quantity}

ORDER:

{order_id} -> {sale_date, total_price, bill_no, customer_id}

USER:

{user_id} -> {email, address_id, phone_no, name_id}

ADDRESS:

{address_id} -> {address, secondary_address, city, state, zip, country}

WAREHOUSE:

{warehouse_id} -> {address_id, total_capacity}

EMPLOYEE:

{employee_id} -> {warehouse_id, position}

WAREHOUSE_STOCK:

{isbn, warehouse_id} -> {quantity}

3. For each relation schema in your model, determine the highest normal form of the relation. If the relation is not in 3NF, rewrite your relation schema so that it is in at least 3NF.

BOOK: BCNF (isbn is a superkey and determines all other attributes, no partial or transitive dependencies)

AUTHOR: BCNF (author_id is a superkey and determines all other attributes, no partial or transitive dependencies)

NAME: BCNF (name_id is a superkey and determines all other attributes, no partial or transitive dependencies)

CATEGORY: BCNF (category_id is a superkey and determines all other attributes, no partial or transitive dependencies)

WRITTEN_BY: BCNF (binary relation)

BOOK_CATEGORY: BCNF (binary relation)

BOOK_ORDER: BCNF ({warehouse_id, isbn, order_id} is a superkey and determines all other attributes, no partial or transitive dependencies)

ORDER: BCNF (order_id is a superkey and determines all other attributes, no partial or transitive dependencies)

USER: BCNF (user_id is a superkey and determines all other attributes, no partial or transitive dependencies)

ADDRESS: BCNF (address_id is a superkey and determines all other attributes, no partial or transitive dependencies)

WAREHOUSE: BCNF (warehouse_id is a superkey and determines all other attributes, no partial or transitive dependencies)

EMPLOYEE: BCNF (employee_id is a superkey and determines all other attributes, no partial or transitive dependencies)

WAREHOUSE_STOCK: BCNF ({warehouse_id, isbn} is a superkey and determines all other attributes, no partial or transitive dependencies)

4. For each relation schema in your model that is in 3NF but not in BCNF, either rewrite the relation schema to BCNF or provide a short justification for why this relation should be an exception to the rule of putting relations into BCNF.

   All relations in the model are already in BCNF.

5. For your database, propose at least two interesting views that can be built from your relations. These views must involve joining at least two tables together each and must include some kind of aggregation in the view. Each view must also be able to be described by a one or two sentence description in plain English. Provide the code for constructing your views along with the English language description of what the view is supposed to be providing.

Customer Order Views:

These are 2 views that allow a customer (with user_id = X) to see all the orders that they have placed, the total price of each order, which books they purchased, and the quantity of each. The first view is the CUSTOMER_ORDERS view, which lists all entries of that specific user's orders, along with the derived total price and total number of books bought in the order. This view is ordered by most recent to least recent order (DESC by date).

CUSTOMER_ORDERS SQL:

```
CREATE VIEW CUSTOMER_ORDERS
AS  SELECT order_id, sale_date, bill_no, sum(quantity) as
total_books_purchased, sum(total_spent) as total_price
    FROM (SELECT O.order_id, O.sale_date, O.bill_no, B.sales_price,
        BO.quantity, BO.quantity * B.sales_price AS total_spent
        FROM "ORDER" O, BOOK_ORDER BO, BOOK B, USER U
        WHERE U.user_id = X AND U.user_id = O.customer_id AND
        O.order_id = BO.order_id AND B.isbn = BO.isbn)
    GROUP BY order_id
    ORDER BY sale_date DESC;
```

The second view is CUSTOMER_BOOK_ORDERS, which lists all the books that a customer has ordered, the order from which they were placed under, the book details, the number of each book bought, and amount spent on each book. The book orders are ordered from most to least recent (DESC by date).

CUSTOMER_BOOK_ORDERS SQL:

```
CREATE VIEW CUSTOMER_BOOK_ORDERS
AS  SELECT O.order_id, O.sale_date, BO.isbn, B.title, B.sales_price,
           BO.quantity, BO.quantity * B.sales_price AS total_spent
    FROM "ORDER" O, BOOK_ORDER BO, BOOK B, USER U
    WHERE U.user_id = X AND U.user_id = O.customer_id AND
    O.order_id = BO.order_id AND B.isbn = BO.isbn
    ORDER BY O.sale_date DESC;
```

Sale History View:

The online bookstore may want to analyze the top-selling books through their platform, which may be used to highlight frequently bought books to their customers. If we take the orders and sort them based on the books they contain, we can gather total sold counts that can be used to find the most successful products on the platform.

```
CREATE VIEW TOTAL_BOOK_SALES
AS  SELECT B.isbn, BO.order_id, B.title, sum(BO.quantity * B.sales_price) as
total_revenue
    FROM BOOK_ORDER BO, BOOK B
    WHERE BO.isbn = B.isbn
GROUP BY B.isbn, B.title
ORDER BY total_revenue DESC;
```

APPENDIX B : Final Schema

```sql
-- CREATING TABLES
-------------------------------------------------------------------------
CREATE TABLE CATEGORY (
    category_id INTEGER NOT NULL PRIMARY KEY,
    cat_name VARCHAR(20) NOT NULL,
    description VARCHAR(255)
);

CREATE TABLE PUBLISHER (
    publisher_id INTEGER NOT NULL PRIMARY KEY,
    pub_name VARCHAR(50) NOT NULL,
    phone_no CHAR(10) NOT NULL,
    email VARCHAR(30) NOT NULL
);

CREATE TABLE BOOK (
    isbn CHAR(13) NOT NULL PRIMARY KEY,
    release_year CHAR(4) NOT NULL,
    sales_price DECIMAL(10,2) NOT NULL,
    title VARCHAR(255) NOT NULL,
    publisher_id INTEGER NOT NULL,
    FOREIGN KEY (publisher_id) REFERENCES PUBLISHER(publisher_id)
);

CREATE TABLE BOOK_CATEGORY (
    isbn CHAR(13) NOT NULL,
    category_id INTEGER NOT NULL,
    FOREIGN KEY (isbn) REFERENCES BOOK(isbn),
    FOREIGN KEY (category_id) REFERENCES CATEGORY(category_id)
);

CREATE TABLE AUTHOR (
    author_id INTEGER NOT NULL PRIMARY KEY,
    name_id INTEGER NOT NULL,
    birth_date DATE,
    death_date DATE,
    FOREIGN KEY (name_id) REFERENCES NAME(name_id)
);

CREATE TABLE NAME (
    name_id INTEGER NOT NULL PRIMARY KEY,
    fname VARCHAR(50) NOT NULL,
    lname VARCHAR(50) NOT NULL,
    middle_inits VARCHAR(10)
);

CREATE TABLE WRITTEN_BY (
    author_id INTEGER NOT NULL,
```

```
    isbn CHAR(13) NOT NULL,
    FOREIGN KEY(author_id) REFERENCES AUTHOR(author_id),
    FOREIGN KEY(isbn) REFERENCES BOOK(isbn)
);

CREATE TABLE USER (
    user_id INTEGER NOT NULL PRIMARY KEY,
    name_id INTEGER NOT NULL,
    email VARCHAR(255) NOT NULL,
    phone_no CHAR(10),
    address_id INTEGER NOT NULL,
    FOREIGN KEY(name_id) REFERENCES NAME(name_id),
    FOREIGN KEY(address_id) REFERENCES ADDRESS(address_id)
);

CREATE TABLE ADDRESS (
    address_id INTEGER NOT NULL PRIMARY KEY,
    address VARCHAR(100) NOT NULL,
    secondary_address VARCHAR(50),
    city VARCHAR(30) NOT NULL,
    state CHAR(2) NOT NULL,
    zip VARCHAR(10) NOT NULL,
    country CHAR(3) NOT NULL
);

CREATE TABLE "ORDER" (
    order_id INTEGER NOT NULL PRIMARY KEY,
    sale_date DATE NOT NULL,
    bill_no INTEGER NOT NULL,
    customer_id INTEGER NOT NULL,
    FOREIGN KEY (customer_id) REFERENCES USER (user_id)
);

CREATE TABLE WAREHOUSE (
    warehouse_id INTEGER NOT NULL PRIMARY KEY,
    address_id INTEGER NOT NULL,
    total_capacity INTEGER NOT NULL,
    FOREIGN KEY(address_id) REFERENCES ADDRESS(address_id)
);

CREATE TABLE BOOK_ORDER (
    warehouse_id INTEGER NOT NULL,
    isbn CHAR(13) NOT NULL,
    order_id INTEGER NOT NULL,
    quantity INTEGER NOT NULL,
    FOREIGN KEY (warehouse_id) REFERENCES WAREHOUSE (warehouse_id),
    FOREIGN KEY (isbn) REFERENCES BOOK (isbn),
    FOREIGN KEY (order_id) REFERENCES "ORDER" (order_id)
```

```
);

CREATE TABLE WAREHOUSE_STOCK (
    isbn CHAR(13) NOT NULL,
    warehouse_id INTEGER NOT NULL,
    quantity INTEGER,
    FOREIGN KEY (isbn) REFERENCES BOOK (isbn),
    FOREIGN KEY (warehouse_id) REFERENCES WAREHOUSE (warehouse_id)
);

CREATE TABLE EMPLOYEE (
    employee_id INTEGER NOT NULL,
    warehouse_id INTEGER,
    position VARCHAR(25) NOT NULL,
    FOREIGN KEY (employee_id) REFERENCES USER(user_id),
    FOREIGN KEY (warehouse_id) REFERENCES WAREHOUSE(warehouse_id)
);



-- CREATING VIEWS
-------------------------------------------------------------------------
-- CUSTOMER_BOOK_ORDERS VIEW
CREATE VIEW CUSTOMER_BOOK_ORDERS
AS  SELECT O.order_id, O.sale_date, BO.isbn, B.title, B.sales_price,
        BO.quantity, BO.quantity * B.sales_price AS total_spent
    FROM "ORDER" O, BOOK_ORDER BO, BOOK B, USER U
    WHERE U.user_id = 50 AND U.user_id = O.customer_id AND
    O.order_id = BO.order_id AND B.isbn = BO.isbn
    ORDER BY O.sale_date DESC;

-- CUSTOMER_ORDERS VIEW
CREATE VIEW CUSTOMER_ORDERS
AS  SELECT order_id, sale_date, bill_no, sum(quantity) as total_books_purchased,
sum(total_spent) as total_price
    FROM (SELECT O.order_id, O.sale_date, O.bill_no, B.sales_price,
        BO.quantity, BO.quantity * B.sales_price AS total_spent
        FROM "ORDER" O, BOOK_ORDER BO, BOOK B, USER U
        WHERE U.user_id = 50 AND U.user_id = O.customer_id AND
        O.order_id = BO.order_id AND B.isbn = BO.isbn)
    GROUP BY order_id
    ORDER BY sale_date DESC;

-- TOTAL_BOOK_SALES VIEW
CREATE VIEW TOTAL_BOOK_SALES
AS  SELECT B.isbn, B.title, N.fname || ' ' || N.middle_inits || '' || N.lname as
author_name,
```

```
        sum(BO.quantity) as copies_sold, sum(BO.quantity * B.sales_price) as
total_revenue
    FROM BOOK_ORDER BO, BOOK B, WRITTEN_BY WB, AUTHOR A, NAME N
    WHERE BO.isbn = B.isbn AND WB.isbn = B.isbn AND
            A.author_id = WB.author_id AND A.name_id = N.name_id
    GROUP BY B.isbn, B.title
    ORDER BY total_revenue DESC;

-- AUTHOR_SALES VIEW
CREATE VIEW AUTHOR_SALES
AS  SELECT N.fname, N.middle_inits, N.lname,
        sum(BO.quantity) as copies_sold, sum(BO.quantity * B.sales_price) as
total_revenue
    FROM BOOK_ORDER BO, BOOK B, WRITTEN_BY WB, AUTHOR A, NAME N
    WHERE BO.isbn = B.isbn AND WB.isbn = B.isbn AND
            A.author_id = WB.author_id AND A.name_id = N.name_id
    GROUP BY A.author_id
    ORDER BY total_revenue DESC;

-- WAREHOUSE_INVENTORY VIEW
CREATE VIEW WAREHOUSE_INVENTORY
AS  SELECT W.warehouse_id, WS.isbn, B.title,
        WS.quantity as inventory, WS.quantity * B.sales_price as inventory_value
    FROM WAREHOUSE W, WAREHOUSE_STOCK WS, BOOK B
    WHERE W.warehouse_id = WS.warehouse_id AND B.isbn = WS.isbn;



-- CREATING INDEXES
-------------------------------------------------------------------------------
-- ADDRESS Indexes
CREATE UNIQUE INDEX IF NOT EXISTS address_id_idx ON ADDRESS(address_id);
CREATE INDEX IF NOT EXISTS state_idx ON ADDRESS(state);

-- AUTHOR Indexes
CREATE UNIQUE INDEX IF NOT EXISTS author_id_idx ON AUTHOR(author_id);
CREATE INDEX IF NOT EXISTS bdate_idx ON AUTHOR(birth_date);

-- BOOK Indexes
CREATE UNIQUE INDEX IF NOT EXISTS isbn_idx ON BOOK(isbn);
CREATE INDEX IF NOT EXISTS release_year_idx ON BOOK(release_year);
CREATE INDEX IF NOT EXISTS price_idx ON BOOK(sales_price);
CREATE INDEX IF NOT EXISTS publisher_idx ON BOOK(publisher_id);

-- BOOK_CATEGORY Indexes
CREATE INDEX IF NOT EXISTS book_cat_isbn_idx ON BOOK_CATEGORY(isbn);
CREATE INDEX IF NOT EXISTS cat_idx ON BOOK_CATEGORY(category_id);
```

```sql
-- BOOK_ORDER Indexes
CREATE INDEX IF NOT EXISTS book_order_warehouse_idx ON BOOK_ORDER(warehouse_id);
CREATE INDEX IF NOT EXISTS book_order_isbn_idx ON BOOK_ORDER(isbn);
CREATE INDEX IF NOT EXISTS book_order_idx ON BOOK_ORDER(order_id);

-- CATEGORY Indexes
-- No indexes for category, as it only contains 9 tuples

-- EMPLOYEE Indexes
CREATE UNIQUE INDEX IF NOT EXISTS employee_index ON EMPLOYEE(employee_id);
CREATE INDEX IF NOT EXISTS emp_warehouse_idx ON EMPLOYEE(warehouse_id);
CREATE INDEX IF NOT EXISTS position_idx ON EMPLOYEE(position);

-- NAME Indexes
CREATE UNIQUE INDEX IF NOT EXISTS name_idx ON NAME(name_id);
CREATE INDEX IF NOT EXISTS lname_idx ON NAME(lname);

-- ORDER Indexes
CREATE UNIQUE INDEX IF NOT EXISTS order_idx ON "ORDER"(order_id);
CREATE INDEX IF NOT EXISTS order_date_idx ON "ORDER"(sale_date);
CREATE INDEX IF NOT EXISTS order_customer_idx ON "ORDER"(customer_id);

-- PUBLISHER Indexes
CREATE UNIQUE INDEX IF NOT EXISTS publisher_id_idx ON PUBLISHER(publisher_id);
CREATE INDEX IF NOT EXISTS publisher_name_idx ON PUBLISHER(pub_name);

-- USER Indexes
CREATE UNIQUE INDEX IF NOT EXISTS user_id_idx ON USER(user_id);
CREATE INDEX IF NOT EXISTS user_name_idx ON USER(name_id);
CREATE INDEX IF NOT EXISTS user_addr_idx ON USER(address_id);
CREATE INDEX IF NOT EXISTS user_email_idx ON USER(email);
CREATE INDEX IF NOT EXISTS user_phone_idx ON USER(phone_no);

-- WAREHOUSE Indexes
CREATE UNIQUE INDEX IF NOT EXISTS warehouse_id_idx ON WAREHOUSE(warehouse_id);
CREATE INDEX IF NOT EXISTS warehouse_addr_idx ON WAREHOUSE(address_id);
CREATE INDEX IF NOT EXISTS warehouse_cap_idx ON WAREHOUSE(total_capacity);

-- WAREHOUSE_STOCK Indexes
CREATE INDEX IF NOT EXISTS warehouse_stk_isbn_idx ON WAREHOUSE_STOCK(isbn);
CREATE INDEX IF NOT EXISTS warehouse_stk_warehouse_idx ON
WAREHOUSE_STOCK(warehouse_id);
CREATE INDEX IF NOT EXISTS warehouse_stk_quantity_idx ON WAREHOUSE_STOCK(quantity);

-- WRITTEN_BY Indexes
CREATE INDEX IF NOT EXISTS written_by_isbn_idx ON WRITTEN_BY(isbn);
CREATE INDEX IF NOT EXISTS writteb_by_author_idx ON WRITTEN_BY(author_id);
```

APPENDIX C : CODE

C.1: csv_parser.rb

```ruby
class CsvParser

  ##
  # Initializes the CsvParser object
  #
  # @param filename : string
  #   the path and filename that you would like to read (including .csv)
  #
  # @param delimiter : char
  #   the char used to seperate columns in the csv
  #
  # @note :
  #   read_file! must be called after creating an object to finish initializing
  #
  def initialize(filename, delimiter)
    @filename = filename
    @delimiter = delimiter
    @columns = 0
    @rows = 0
    @hash_array = []
  end

  ##
  # Generates a csv file for any dataset
  #
  # @param filename : string
  #   The name of the file you want to generate, must end with ".csv"
  #
  # @param has_array : array(hash)
  #   An array of uniform hashed (same keys) that represent some form of data.
  #   The keys are the attibute names (column headers) and the values get
  #   written to a csv file by hash
  #
  def generate_csv(filename, hash_array, headers)
    file = File.open(filename, "w")
    attribute_names = hash_array[0].keys

    if headers == 1
      # Creating column headers
      attribute_names.each_with_index do |name,i|
        if i < (attribute_names.length - 1)
          file.write(name.to_s + @delimiter)
        else
          file.write(name.to_s + "\n")
        end
```

```
    end
  end

  # Exporting Data
  hash_array.each_with_index do |entity,i|
    attribute_names.each_with_index do |column,j|
      if j < (attribute_names.length - 1)
        file.write(entity[attribute_names[j]].to_s + @delimiter)
      else
        file.write(entity[attribute_names[j]].to_s + "\n")
      end
    end
  end

  file.close
end

##
# Allows you to change which file you are accessing
#
# @param filename : string
#   the path and filename that you would like to read (including .csv)
#
# @modifies :
#   resets hash_array to [], and changes the filename
#
# @requires :
#   read_file! must be called after changing the file
#
def change_file!(filename)
  @filename = filename
  @columns = 0
  @rows = 0
  @hash_array = []
end

##
# Reads in the given csv file into the CsvParser object
#
# @requires :
#   All columns must have unique names (the first row)
#
def read_file!
  # Setting the hash keys to the value of the first row
  begin
    file = File.open(@filename)
  rescue
    print "Error: Failed to open #{@filename}\n"
```

```
      exit
    end
    text = file.read
    file.close
    lines = text.split("\n")
    first_line = lines[0]
    lines.shift
    keys = first_line.split(@delimiter)
    @columns = keys.length

    # Writing the data into hashes
    lines.each_with_index do |line, line_num|
      hash = {}
      value_array = line.split(@delimiter)
      value_array.each_with_index do |value, column|
        hash[keys[column]] = value
      end
      @hash_array.push(hash)
      @rows = @rows + 1
    end
  end

  ##
  # Returns the number of columns in the CsvParser object
  #
  def get_column_count
    @columns
  end

  ##
  # Returns the number of rows in the CsvParser object
  #
  def get_row_count
    @rows
  end

  ##
  # Returns the hash array containing the data of the csv
  #
  # Hash array is set up such that
  #   [row1{"attr1" => value, "attr2" => value, ...}, row2{"attr1" => value, "attr2"
=> value, ...}, ...]
  #
  def get_hash_array
    @hash_array
  end

  ##
```

```ruby
# Returns an array containing the column names of the given csv file
#
def get_column_names
  @hash_array[0].keys
end

##
# Returns an array containing all the values of the given column name
#
# @param column_name : string
#   The identifying key of the column desired
#
def get_column(column_name)
  column_array = []
  @hash_array.each do |hash|
    column_array.push({column_name => hash[column_name]})
  end
  column_array
end

##
# Removes the specified columns from the CsvParser object
#
# @param column_names : array(string)
#
def drop_columns!(column_names)
  column_names.each do |c_name|
    @hash_array.each_with_index do |hash, index|
      hash = hash.except(c_name)
      @hash_array[index] = hash
    end
  end
end

##
# Prints the CsvParser object data as a table
#
# @param stream : IOStream
#   The stream that will be written to (defaults to STDOUT)
#
def fprint(stream = STDOUT)
  # Printing column headers
  keys = @hash_array[0].keys
  stream.write("1:\t")
  keys.each_with_index do |key, index|
    if index != keys.length - 1
      stream.write("#{key}\t")
    else
```

```
      stream.write("#{key}\n")
    end
  end

  i = 2
  # Printing data
  @hash_array.each do |hash|
    stream.write("#{i}:\t")
    hash.each_with_index do |value, index|
      if index != hash.size - 1
        stream.write("#{value}\t")
      else
        stream.write("#{value}\n")
      end
      i = i + 1
    end
  end
end

##
# Prints the given hash array data as a table
#
# @param stream : IOStream
#   The stream that will be written to (defaults to STDOUT)
#
# @param hash_array : array(hash)
#   The data to be printed
#
def fprint(stream = STDOUT, hash_array)
  # Printing column headers
  keys = hash_array[0].keys
  stream.write("1:\t")
  keys.each_with_index do |key, index|
    if index != keys.length - 1
      stream.write("#{key}\t")
    else
      stream.write("#{key}\n")
    end
  end

  i = 2
  # Printing data
  hash_array.each do |hash|
    stream.write("#{i}:\t")
    hash.each_with_index do |value, index|
      if index != hash.size - 1
        stream.write("#{value}\t")
      else
```

```
          stream.write("#{value}\n")
        end
        i = i + 1
      end
    end
  end

  ##
  # Returns the given array with the element given by the index removed
  #
  # @param arr : array
  #   The array to remove the index location from
  #
  # @param index_to_exclude : int
  #   The index of the element to be removed from the array
  #
  def remove_index(arr, index_to_exclude)
    arr[0,index_to_exclude].concat(arr[index_to_exclude+1..-1])
  end

  ##
  # Returns an array of hashes containing no nil rows of the given attribute
  #
  # @param array : array(hash)
  #   The array to remove the nil rows from
  #
  # @param attribute : string
  #   The given attribute to check for nil/empty string
  #
  def remove_nil_rows(array, attibute)
    indices_to_remove = []
    array.each_with_index do |hash, index|
      if ["", nil].include?(hash[attibute])
        indices_to_remove.push(index)
      end
    end

    flipped_indices = indices_to_remove.reverse
    flipped_indices.each do |index|
      array = remove_index(array, index)
    end
    array
  end

end
```

C.2: bb_parser.rb

```ruby
require_relative '../Model/csv_parser'
require 'faker'
require 'date'
require 'rubystats'
Faker::Config.locale = 'en-US'

# Config
NUM_CUSTOMERS = 150
NUM_EMPLOYEES = 20
MAX_ORDER_PER_CUSTOMER = 5
MIN_ORDER_PER_CUSTOMER = 0
MAX_DIFF_BOOKS_PER_ORDER = 10
MIN_DIFF_BOOKS_PER_ORDER = 1
MAX_BOOK_ORDER_QUANTITY = 3
MIN_BOOK_ORDER_QUANTITY = 1
WAREHOUSES_IN_USE = 2
TOTAL_WAREHOUSES = 20
START_SALES_DATE = "2010-01-01"
END_SALES_DATE = Date.today
RESERVE_SPACE = 500

name_hash_array = []
name_id_counter = 1
address_hash_array = []
address_id_counter = 1

def get_author_id(auth_id_arr, auth_name)
  author = auth_id_arr.select {|auth_hash| auth_hash["name"] == auth_name}.first
  author["author_id"]
end

# Reading in data from bits and books csv file
filename = "../Data/original_bb_data.csv"
parser = CsvParser.new(filename, ";")
parser.read_file!
parser.get_hash_array
parser.drop_columns!(["nil1", "nil2", "nil3\r"])

# Creating a CSV of Categories
cat_column = parser.get_column("Category")
category_array = []
category_names = []
cat_column.each do |category|
  # Add each unique category name to the array
```

```ruby
  if !category_names.include?(category["Category"]) && category["Category"] != ""
    category_names.push(category["Category"])
  end
end

# Add each category name, along with an id number to the category hash array
category_names.each_with_index do |cat_name, index|
  category_array.push({"category_id" => index + 1, "cat_name" => cat_name,
"description" => "N/A"})
end

# FOR TESTING:
# puts category_array

parser.generate_csv("../OutputHeaders/bb_categories.csv", category_array, 1)
parser.generate_csv("../OutputNoHeaders/bb_categories.csv", category_array, 0)

# Creating a Csv of Publishers
publisher_column = parser.get_column("Publisher")
publisher_array = []
publisher_names = []
publisher_column.each do |publisher|
  # Add each unique publisher name to the array
  if !publisher_names.include?(publisher["Publisher"]) && publisher["Publisher"] != ""
    publisher_names.push(publisher["Publisher"])
  end
end

# Adding id, unique publisher name, randomly generated phone number, and email
computed from publisher name
publisher_names.each_with_index do |publisher_name, index|
  pub_id = index + 1
  phone_no = Faker::PhoneNumber.cell_phone.delete("-. ()")
  lowercase_name = publisher_name.downcase
  email = "supply@" + lowercase_name.delete(" ,.") + ".org"
  publisher_array.push({"publisher_id" => pub_id, "pub_name" => publisher_name,
"phone_no" => phone_no, "email" => email})
end

# FOR TESTING:
# puts publisher_array

parser.generate_csv("../OutputHeaders/bb_publishers.csv", publisher_array, 1)
parser.generate_csv("../OutputNoHeaders/bb_publishers.csv", publisher_array, 0)

data_hash_array = parser.get_hash_array

# Setting all ISBN's to 13 characters
```

```ruby
data_hash_array.each_with_index do |row, index|
  if row["ISBN"] != ""
    while row["ISBN"].length < 13
      added_zero = row["ISBN"].prepend("0")
      data_hash_array[index]["ISBN"] = added_zero
    end
  end
end

# Creating a CSV of books
books_array = []
book_categories = []
data_hash_array.each_with_index do |row, index|
  # Only adding to the hash if the row has an isbn
  if row["ISBN"] != ""
    isbn = row["ISBN"]
    release_year = row["Year"]
    sales_price = row["Price"].delete("$").to_f
    title = row["Title"]
    # Getting the publisher ID of the book
    publisher_hash = publisher_array.select{|hash| hash["pub_name"] ==
row["Publisher"]}.first
    pub_id = publisher_hash["publisher_id"]
    # Getting the category ID of the book
    cat_hash = category_array.select{|hash| hash["cat_name"] == row["Category"]}.first
    cat_id = cat_hash["category_id"]
    book_categories.push({"isbn" => isbn, "category_id" => cat_id})
    if !books_array.any? {|h| h["isbn"] == isbn}
      books_array.push({"isbn" => isbn, "release_year" => release_year, "sales_price"
=> sales_price, "title" => title, "pub_id" => pub_id})
    end
  end
end

# FOR TESTING:
# puts books_array

parser.generate_csv("../OutputHeaders/bb_books.csv", books_array, 1)
parser.generate_csv("../OutputNoHeaders/bb_books.csv", books_array, 0)
parser.generate_csv("../OutputHeaders/bb_book_categories.csv", book_categories, 1)
parser.generate_csv("../OutputNoHeaders/bb_book_categories.csv", book_categories, 0)

# Creating a CSV of authors
author_array = []
author_names = []
auth_id_array = []
data_hash_array.each_with_index do |row, index|
  # Adding each unique author name to the array
```

```ruby
  if !author_names.include?(row["Author(s)"])
    author_names.push(row["Author(s)"])
  end
end

# Adding to the author hash array for each unique author name
author_names.each_with_index do |author, index|
  author_id = index + 1
  name = author
  # Getting fname, lname, and middle initials
  split_name = name.split(" ")
  first_name = split_name[0]
  last_name = split_name[split_name.length - 1]
  middle_inits = ""
  if split_name.length > 2
    middle_inits_array = split_name[1, split_name.length - 2]
    middle_inits_array.each_with_index do |init, index|
      if index < middle_inits_array.length - 1
        middle_inits = middle_inits + init + " "
      else
        middle_inits = middle_inits + init
      end
    end
  end

  # Generating appropriate birth dates and death dates
  bdate_start = '1930-01-01'
  bdate_end = '1980-12-31'
  birth_date = Faker::Date.between(from: bdate_start, to: bdate_end)

  ddate_start = '2010-01-01'
  ddate_end = Date.today
  death_date = Faker::Date.between(from: ddate_start, to: ddate_end)

  forty_years_in_days = 365 * 40
  sixty_years_in_days = 365 * 60

  # Getting a possible random death date (could be null based on age)
  days_alive = death_date - birth_date
  dead_prob = rand(0..100)
  if days_alive < forty_years_in_days
    if dead_prob >= 20
      death_date = ""
    end
  elsif days_alive < sixty_years_in_days
    if dead_prob >= 35
      death_date = ""
    end
```

```
  else
    if dead_prob >= 65
      death_date = ""
    end
  end

  author_array.push({"author_id" => author_id, "name_id" => name_id_counter,
    "birth_date" => birth_date, "death_date" => death_date})
  name_hash_array.push({"name_id" => name_id_counter, "fname" => first_name,
    "lname" => last_name, "middle_inits" => middle_inits})
  auth_id_array.push({"author_id" => author_id, "name" => name})
  name_id_counter = name_id_counter + 1
end

# FOR TESTING:
# puts author_array

parser.generate_csv("../OutputHeaders/bb_authors.csv", author_array, 1)
parser.generate_csv("../OutputNoHeaders/bb_authors.csv", author_array, 0)

# Generating Written By CSV
written_by_array = []
isbn_written_by = 0
data_hash_array.each_with_index do |row, index|
  # Each time a new book is seen, set the isbn
  if row["ISBN"] != ""
    isbn_written_by = row["ISBN"]
  end

  isbn = isbn_written_by
  author_id = get_author_id(auth_id_array, row["Author(s)"])
  hash_to_add = {"author_id" => author_id, "isbn" => isbn}
  if !written_by_array.include? hash_to_add
    written_by_array.push(hash_to_add) if !written_by_array.include? hash_to_add
  end

end

# FOR TESTING:
# puts written_by_array

parser.generate_csv("../OutputHeaders/bb_written_by.csv", written_by_array, 1)
parser.generate_csv("../OutputNoHeaders/bb_written_by.csv", written_by_array, 0)

# Generating users
user_counter = 1
customer_users_array = []
employee_users_array = []
```

```
# Generating employees
providers = ["bitsbooks.com"]
for i in 0...NUM_EMPLOYEES
  user_id = user_counter
  user_counter = user_counter + 1
  fname = Faker::Name.first_name
  lname = Faker::Name.last_name
  email = "#{fname}.#{lname}#{rand(0..999)}@#{providers.sample}"
  phone_no = Faker::PhoneNumber.cell_phone.delete("-. ()")
  address = Faker::Address.street_address
  secondary_address = ""
  if rand(1..10) == 1
    secondary_address = Faker::Address.secondary_address
  end
  city = Faker::Address.city
  country = "USA"
  state = Faker::Address.state_abbr
  zip = Faker::Address.postcode
  employee_users_array.push({"user_id" => user_id, "name_id" => name_id_counter,
"email" => email,
    "phone_no" => phone_no, "address_id" => address_id_counter})
  name_hash_array.push({"name_id" => name_id_counter, "fname" => fname, "middle_inits"
=> nil, "lname" => lname})
  address_hash_array.push({"address_id" => address_id_counter, "address" => address,
    "secondary_address" => secondary_address, "city" => city, "state" => state, "zip"
=> zip, "country" => country})
  name_id_counter = name_id_counter + 1
  address_id_counter = address_id_counter + 1
end

# Generating customers
providers = ["gmail.com", "hotmail.com", "yahoo.com"]
for i in 0...NUM_CUSTOMERS
  user_id = user_counter
  user_counter = user_counter + 1
  fname = Faker::Name.first_name
  lname = Faker::Name.last_name
  name = fname + " " + lname
  email = "#{fname}.#{lname}#{rand(0..999)}@#{providers.sample}"
  phone_no = Faker::PhoneNumber.cell_phone.delete("-. ()")
  address = Faker::Address.street_address
  secondary_address = ""
  if rand(1..10) == 1
    secondary_address = Faker::Address.secondary_address
  end
  city = Faker::Address.city
  if rand(1..50) == 1
    country = "CAN"
```

```
    possible_regions = ["QC", "ON", "MB", "SK", "AB", "NS", "NB", "NL", "PE", "BC",
"YT", "NT", "NU"]
    state = possible_regions.sample
  else
    country = "USA"
    state = Faker::Address.state_abbr
  end
  zip = Faker::Address.postcode
  customer_users_array.push({"user_id" => user_id, "name_id" => name_id_counter,
"email" => email,
    "phone_no" => phone_no, "address_id" => address_id_counter})
  name_hash_array.push({"name_id" => name_id_counter, "fname" => fname, "middle_inits"
=> nil, "lname" => lname})
  address_hash_array.push({"address_id" => address_id_counter, "address" => address,
    "secondary_address" => secondary_address, "city" => city, "state" => state, "zip"
=> zip, "country" => country})
  name_id_counter = name_id_counter + 1
  address_id_counter = address_id_counter + 1
end

# FOR TESTING:
# puts employee_users_array
# puts customer_users_array

parser.generate_csv("../OutputHeaders/bb_users.csv", employee_users_array +
customer_users_array, 1)
parser.generate_csv("../OutputNoHeaders/bb_users.csv", employee_users_array +
customer_users_array, 0)

order_counter = 1
bill_numbers = []
order_array = []
# Generating orders
customer_users_array.each_with_index do |customer, index|
  customer_id = customer["user_id"]
  for i in 0..rand(MIN_ORDER_PER_CUSTOMER..MAX_ORDER_PER_CUSTOMER)
    sale_date = Faker::Date.between(from: START_SALES_DATE, to: END_SALES_DATE)
    valid_bill_no = 0
    while valid_bill_no == 0 do
      bill_no = rand(10000..99999)
      if !bill_numbers.include?(bill_no)
        bill_numbers.push(bill_no)
        valid_bill_no = 1
      end
    end
    order_array.push({"order_id" => 0, "sale_date" => sale_date, "bill_no" => bill_no,
      "customer_id" => customer_id})
  end
```

```ruby
end

#Sorting by sale date then assigning order numbers
order_array.sort_by! {|order| order["sale_date"]}
order_array.each_with_index do |order, idx|
  order_array[idx]["order_id"] = order_counter
  order_counter = order_counter + 1
end

# FOR TESTING:
# puts order_array

parser.generate_csv("../OutputHeaders/bb_orders.csv", order_array, 1)
parser.generate_csv("../OutputNoHeaders/bb_orders.csv", order_array, 0)

# Generating Warehouses
warehouse_array = []
for i in 0...TOTAL_WAREHOUSES
  warehouse_id = i + 1
  address = Faker::Address.street_address
  city = Faker::Address.city
  state = Faker::Address.state_abbr
  zip = Faker::Address.postcode
  country = "USA"
  total_capacity = rand(2000..8000)
  warehouse_array.push({"warehouse_id" => warehouse_id, "address_id" =>
address_id_counter,
    "total_capacity" => total_capacity})
  address_hash_array.push({"address_id" => address_id_counter, "address" => address,
    "secondary_address" => nil, "city" => city, "state" => state, "zip" => zip,
"country" => country})
  address_id_counter = address_id_counter + 1
end

# FOR TESTING:
# puts warehouse_array

parser.generate_csv("../OutputHeaders/bb_warehouses.csv", warehouse_array, 1)
parser.generate_csv("../OutputNoHeaders/bb_warehouses.csv", warehouse_array, 0)

# Generating Warehouse Stock
warehouse_stock_array = []
warehouse_array.each_with_index do |warehouse, index|
  moving_capacity = warehouse["total_capacity"]
  average_book_stock = (warehouse["total_capacity"] - RESERVE_SPACE) /
books_array.length
  book_stock_dev = 50
  warehouse_id = warehouse["warehouse_id"]
```

```
  break if warehouse_id > WAREHOUSES_IN_USE
  norm = Rubystats::NormalDistribution.new(average_book_stock, book_stock_dev)
  books_array.each_with_index do |book, index2|
    isbn = book["isbn"]
    quantity = norm.rng.to_i.abs
    if moving_capacity < quantity
      quantity = moving_capacity
    end
    warehouse_stock_array.push({"isbn" => isbn, "warehouse_id" => warehouse_id,
"quantity" => quantity})
  end
end

# FOR TESTING:
# puts warehouse_stock_array

parser.generate_csv("../OutputHeaders/bb_warehouse_stock.csv", warehouse_stock_array,
1)
parser.generate_csv("../OutputNoHeaders/bb_warehouse_stock.csv",
warehouse_stock_array, 0)

# Generating Book Orders
book_order_array = []
order_array.each_with_index do |order, index|
  books_ordered = []
  order_id = order["order_id"]
  warehouse_id = rand(1..WAREHOUSES_IN_USE)
  for i in 0...rand(MIN_DIFF_BOOKS_PER_ORDER..MAX_DIFF_BOOKS_PER_ORDER)
    # selecting book
    unique_book = 0
    while unique_book == 0
      isbn = books_array.sample["isbn"]
      if !books_ordered.include?(isbn)
        unique_book = 1
        books_ordered.push(isbn)
      end
    end
    quantity = rand(MIN_BOOK_ORDER_QUANTITY..MAX_BOOK_ORDER_QUANTITY)
    book_order_array.push({"warehouse_id" => warehouse_id, "isbn" => isbn, "order_id"
=> order_id, "quantity" => quantity})
  end
end

# FOR TESTING:
# puts book_order_array

parser.generate_csv("../OutputHeaders/bb_book_orders.csv", book_order_array, 1)
parser.generate_csv("../OutputNoHeaders/bb_book_orders.csv", book_order_array, 0)
```

```ruby
positions = ["Information Technology", "Human Resources", "Logistics", "Manager",
"Transport", "Robotics Engineer"]

# Generating Employees
employee_array = []
employee_users_array.each_with_index do |emp_user, index|
  employee_id = emp_user["user_id"]
  warehouse_id = rand(1..WAREHOUSES_IN_USE)
  position = positions.sample
  employee_array.push({"employee_id" => employee_id, "warehouse_id" => warehouse_id,
"position" => position})
end

# FOR TESTING:
# puts employee_array

parser.generate_csv("../OutputHeaders/bb_employees.csv", employee_array, 1)
parser.generate_csv("../OutputNoHeaders/bb_employees.csv", employee_array, 0)

# Generating final name and address csvs
parser.generate_csv("../OutputHeaders/bb_names.csv", name_hash_array, 1)
parser.generate_csv("../OutputNoHeaders/bb_names.csv", name_hash_array, 0)
parser.generate_csv("../OutputHeaders/bb_addresses.csv", address_hash_array, 1)
parser.generate_csv("../OutputNoHeaders/bb_addresses.csv", address_hash_array, 0)
```

C.3 gen_new_seed.sh

```
ruby bb_parser.rb
sqlite3 ../db/$1 <<'END_SQL'
.read ../Schemas/BitsandBooks.sql
.mode csv
.separator ;
.import ../OutputNoHeaders/bb_categories.csv CATEGORY
.import ../OutputNoHeaders/bb_publishers.csv PUBLISHER
.import ../OutputNoHeaders/bb_books.csv BOOK
.import ../OutputNoHeaders/bb_book_categories.csv BOOK_CATEGORY
.import ../OutputNoHeaders/bb_authors.csv AUTHOR
.import ../OutputNoHeaders/bb_written_by.csv WRITTEN_BY
.import ../OutputNoHeaders/bb_users.csv USER
.import ../OutputNoHeaders/bb_orders.csv "ORDER"
.import ../OutputNoHeaders/bb_warehouses.csv WAREHOUSE
.import ../OutputNoHeaders/bb_book_orders.csv BOOK_ORDER
.import ../OutputNoHeaders/bb_warehouse_stock.csv WAREHOUSE_STOCK
.import ../OutputNoHeaders/bb_employees.csv EMPLOYEE
.import ../OutputNoHeaders/bb_addresses.csv ADDRESS
.import ../OutputNoHeaders/bb_names.csv NAME
.quit
END_SQL
```