DATA 514 Section 4 Worksheet - Jonathan Jacobs

Entity Relationship Diagrams

Question 1

Entities and Attributes:

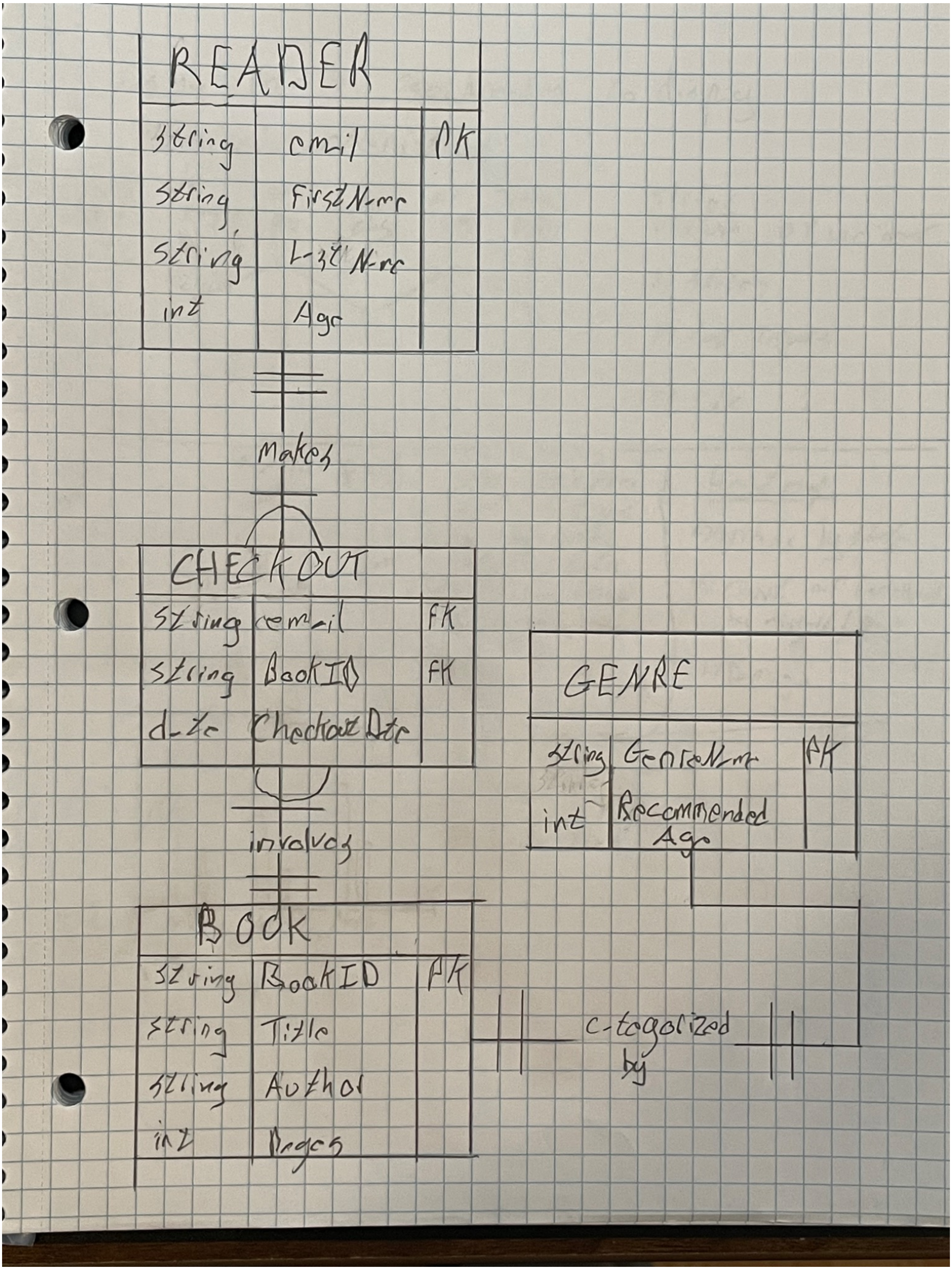
1. **Book**
    - Attributes: BookID (PK), Title, Author, Genre, Pages, RecommendedAge
    - The inclusion of `RecommendedAge` suggests a normalization where `Genre` is linked to a typical `RecommendedAge` for the genre.
2. **Reader**
    - Attributes: Email (PK), FirstName, LastName, Age
3. **Checkout**
    - Attributes: Email (FK), BookID (FK), CheckoutDate
    - This entity represents the many-to-many relationship between books and readers, where a book can be checked out multiple times and readers can check out multiple books. The inclusion of `CheckoutDate` directly in this entity allows us to track when each book is checked out by a reader.
4. **Genre**
    - Attributes: GenreName (PK), RecommendedAge
    - This entity is normalized from the `Book` entity to manage the recommended age for each genre effectively.

Relationships:

- **Books to Genre**: One-to-Many (One genre can categorize many books, a book belongs to one genre).
- **Readers to Checkout**: One-to-Many (A reader can have multiple checkout records, a checkout record belongs to one reader).
- **Books to Checkout**: One-to-Many (A book can be checked out multiple times, each checkout record refers to one book).

Assumptions:

- Each book has only one genre.
- The recommendation age is consistent per genre and is stored with the genre.
- Readers are uniquely identified by their email addresses.


READER
string email PK
string FirstName
string LastName
int Age
makes
CHECKOUT
string email FK
string BookID FK
date CheckoutDate
involves
BOOK
string BookID PK
string Title
string Author
int Pages
GENRE
string GenreName PK
int Recommended Age
categorized by

Question 2

```sql
-- Table for storing genres and their recommended ages
CREATE TABLE Genre (
    GenreName VARCHAR(255) PRIMARY KEY,
    RecommendedAge INT NOT NULL
);

-- Table for storing books
CREATE TABLE Book (
    BookID VARCHAR(255) PRIMARY KEY,
    Title VARCHAR(255) NOT NULL,
    Author VARCHAR(255) NOT NULL,
    Pages INT NOT NULL,
    GenreName VARCHAR(255),
    FOREIGN KEY (GenreName) REFERENCES Genre(GenreName)
);

-- Table for storing reader information
CREATE TABLE Reader (
    Email VARCHAR(255) PRIMARY KEY,
    FirstName VARCHAR(255) NOT NULL,
    LastName VARCHAR(255) NOT NULL,
    Age INT NOT NULL
);

-- Table for storing checkout information, tracking which reader checked out which book and when
CREATE TABLE Checkout (
    Email VARCHAR(255),
    BookID VARCHAR(255),
    CheckoutDate DATE NOT NULL,
    PRIMARY KEY (Email, BookID, CheckoutDate),
    FOREIGN KEY (Email) REFERENCES Reader(Email),
    FOREIGN KEY (BookID) REFERENCES Book(BookID)
);
```

Explanation

1. **Genre Table**: Stores the genres and their corresponding recommended ages.
2. **Book Table**: Contains details about each book, including a foreign key linking to the `Genre` table to categorize each book.
3. **Reader Table**: Maintains information on each reader, identified uniquely by their email.
4. **Checkout Table**: Records each instance of a book being checked out by a reader, including the date. This table has composite keys to uniquely identify each checkout record and foreign keys linking to both `Reader` and `Book`.

Question 3

```sql
CREATE TABLE Ingredient (
    iid INT PRIMARY KEY,
```

```sql
    name VARCHAR(255),
    allergen VARCHAR(255) -- Assuming allergen is just a text field, not a foreign key to an allergen table
);

CREATE TABLE Dish (
    did INT PRIMARY KEY,
    name VARCHAR(255),
    description TEXT,
    category VARCHAR(255)
);

CREATE TABLE Order (
    oid INT PRIMARY KEY
);

CREATE TABLE DishOrder (
    oid INT,
    did INT,
    num INT,
    PRIMARY KEY (oid, did),
    FOREIGN KEY (oid) REFERENCES Order(oid),
    FOREIGN KEY (did) REFERENCES Dish(did)
);

CREATE TABLE IngredientIn (
    iid INT,
    did INT,
    PRIMARY KEY (iid, did),
    FOREIGN KEY (iid) REFERENCES Ingredient(iid),
    FOREIGN KEY (did) REFERENCES Dish(did)
);
```

Explanation

1. **Ingredient Table**: Holds the data for each ingredient, which is identified by a unique identifier `iid`.
2. **Dish Table**: Contains the details for each dish offered, with each dish having a unique identifier `did`.
3. **Order Table**: Keeps track of customer orders where each order is uniquely identified by `oid`.
4. **DishOrder Table**: Acts as a junction table between `Dish` and `Order`, denoting which dishes have been ordered and in what quantity. It uses composite keys made of `oid` and `did` for unique identification and includes `num` to indicate how many of each dish was ordered.
5. **IngredientIn Table**: Functions as a junction table to manage the many-to-many relationship between `Ingredients` and `Dishes`, indicating which ingredients are used in which dishes. It also uses a composite key consisting of `iid` and `did`.

Functional Dependencies

Question 1

Within the given relation, we can observe the following functional dependencies:

1. The `license_plate` uniquely identifies each row, which implies the following functional dependency:
    - `license_plate → car_type, car_color, is_electric, is_yellow` The closure of `{license_plate}` therefore includes all the attributes in the relation, as the license plate number is unique to each vehicle.
    - Closure: `{license_plate}+ = {car_type, car_color, is_electric, is_yellow, license_plate}`
2. The combination of `car_type` and `car_color` is observed to determine the `is_electric` attribute, given that within the dataset, no two vehicles of the same type and color have different `is_electric` values. This leads to the second functional dependency:
    - `car_type, car_color → is_electric` The closure for the combination `{car_type, car_color}` includes the `is_electric` attribute.
    - Closure: `{car_type, car_color}+ = {car_type, car_color, is_electric}`

These conclusions are drawn from the explicit content of the dataset. It's important to note that the dataset as provided is a snapshot, and these dependencies hold true for the data at hand. Should the dataset be expanded or varied, the functional dependencies and closures may require reassessment.

BCNF Decomposition

To solve the problem of decomposing the relation R(A, B, C, D, E) into BCNF, we start by calculating the closures for the sets {A}, {B}, {D}, and {BD}. Then, using the given functional dependencies, we decompose the relation into BCNF, ensuring each relation in the decomposition adheres to the BCNF conditions.

Step 1: Calculate Closures

- **{A}+**: Given that A → C, the closure of {A} would include A and C. Since no other functional dependencies provide additional attributes based on A or C, the closure is {A, C}.
- **{B}+**: No functional dependencies have B on the left-hand side. Hence, B alone cannot determine any other attributes. The closure of {B} is {B}.
- **{D}+**: Given that D → E, the closure of {D} includes D and E. Again, there are no additional attributes determined by D or E alone, so the closure is {D, E}.
- **{BD}+**: Since BD → A and D → E, from BD we can determine A and E. Therefore, BD determines all attributes, A, B, C, D, E (because A → C also applies). Thus, the closure of {BD} is {A, B, C, D, E}.

Step 2: Decompose into BCNF

BCNF requires that for every functional dependency X → Y in a relation, X should be a superkey. We check each functional dependency and decompose if it violates this rule:

1. **Check and Decompose for A → C**:
    - In R(A, B, C, D, E), A → C is not a superkey because {A}+ = {A, C} does not include all attributes of R.
    - **Decompose**:
        - R1: (A, C) where A is a key.
        - R2: (A, B, D, E) with A removed as it is fully functionally dependent on C in R1.

2. **Check and Decompose R2 for BD → A**:
    - In R2(A, B, D, E), BD → A does meet the condition of BD being a superkey (since {BD}+ = {A, B, D, E} includes all attributes in R2). Thus, no further decomposition based on BD → A is required.
3. **Check R2 for D → E**:
    - In R2(A, B, D, E), D → E does not meet the BCNF condition as D is not a superkey (since {D}+ = {D, E}).
    - **Decompose**:
        - R3: (D, E) where D is a key.
        - R4: (A, B, D) with E removed.

Final Decomposition:

- **R1: (A, C)** where A is a key.
- **R3: (D, E)** where D is a key.
- **R4: (A, B, D)** where BD is a key.

Each resulting relation now satisfies BCNF because the determining attribute set in each functional dependency of each relation is a superkey of that relation. This decomposition ensures no redundancy based on the functional dependencies provided.