Takeaway Sheet 4 (Lectures 10-12) - Jonathan Jacobs

Question 1

For the MAKES table decision regarding your Payroll and Regist examples, a MAKES table is not relevant. The notion of a "MAKES" table generally applies to relationships between manufacturers and their products, such as companies producing specific goods. In the case of the Payroll and Regist data, the tables detail employees and their cars, not a product-manufacturing relationship. Therefore, there is no need for a MAKES table in this context, as the tables do not describe a manufacturing process or a relationship between products and producers.

Question 2

Entity Sets, Attributes, and Relationships

For the Payroll and Regist examples:

- **Entity Sets and Attributes**:
  - **Payroll Table**:
    - **UserID**: Unique identifier for each employee.
    - **Name**: Name of the employee.
    - **Job**: Job title or position of the employee.
    - **Salary**: Salary of the employee.
  - **Regist Table**:
    - **UserID**: Links to the UserID in the Payroll table, indicating which employee owns which car.
    - **Car**: The model or name of the car owned by the employee.
- **Relationships and Multiplicities**:
  - **Payroll to Regist Relationship**:
    - **Multiplicity**: One-to-Many (1:M)
      - Each employee (UserID in Payroll) can own zero or more cars (entries in Regist), but each car entry in the Regist table refers back to exactly one employee in the Payroll table.
      - This relationship establishes a typical one-to-many linkage where an employee can have multiple car registrations, each uniquely linked back to them via the UserID.

Question 3

Based on the ER diagram, the two rows for purchasing chips are disallowed under the given setup. The diagram likely imposes constraints ensuring that a buyer always purchases a specific product from the same company. This arrangement prevents entries that depict the same buyer purchasing the same product type from different companies, which would violate the diagram's rules and undermine data consistency and integrity. Allowing such entries would necessitate redesigning the primary keys to include

company identifiers, making each combination of `buyerID`, `productName`, and `companyID` unique to accommodate multiple suppliers for the same product.

Question 4

1. **Phone Numbers Uniqueness**
    - Phone numbers do not uniquely identify a person. Multiple people could share the same phone number (e.g., family members or company phones). Hence, phone numbers should not be used as a primary key in a database that requires unique identifiers for each person.
2. **UW Student Numbers Uniqueness**
    - UW student numbers uniquely identify a student among UW students. Each student number is assigned to only one student, making it a suitable candidate for a primary key in a database managing student information.

Question 5

To determine whether "Name -> Job" holds as a functional dependency in the dataset, we need to check if a person's name uniquely determines their job title. This dependency would hold if every instance of a name is associated with only one job title throughout the dataset.

To verify this, I can run a SQL query that looks for any names associated with multiple job titles. Here's the query I would use:

```
SELECT Name, COUNT(DISTINCT Job) AS JobCount
FROM YourTable
GROUP BY Name
HAVING COUNT(DISTINCT Job) > 1;
```

In this query:

- I select each name along with the count of distinct job titles associated with that name.
- I group the results by the name to aggregate job titles per individual.
- The `HAVING COUNT(DISTINCT Job) > 1` condition filters out any names linked to only one job title.

If the query returns no results, it means "Name -> Job" holds, as every name is consistently linked to a single job title. If the query returns any rows, it indicates that the same name has different job titles in the dataset, so the dependency does not hold.

Question 6

1. **ID -> Name**: This functional dependency holds for the specific relation shown. Since each ID is unique to a person, it correctly determines the name. This dependency should be satisfied by any relation that includes ID and Name as attributes, where ID is a unique identifier.

2. **Name -> Car**: This dependency does not hold. The data example shows that an individual (e.g., Magda) can have more than one car (Civic and Ferrari), indicating that knowing a name alone does not uniquely determine the car.

3. **Name -> Job**: This dependency appears to hold in the given data, as each name is associated with a single job. However, this might not be generally satisfied in all relations if a person can have more than one job role in different contexts or time periods.

Question 7

1. **Closure of {Name} ({Name}+):**
    - If Name determines attributes like Job, then the closure of {Name} would include these attributes. Assuming the typical dependency Name → Job:
        - {Name}+ = {Name, Job}

2. **Closure of {Name, Signature} ({Name, Signature}+):**
    - Combining Name and Signature, if these determine additional attributes such as Project or Task, the closure would encompass these. Based on possible dependencies:
        - {Name, Signature}+ = {Name, Signature, Project, Task}

Question 8

1. **Initial Table: Restaurants(id, name, rating, popularity, rec?)**
    - FDs: `id -> name, rating`

2. **Decomposition Process:**
    - Decompose based on the FD `id -> name, rating`. This FD suggests that the restaurant ID uniquely determines the restaurant's name and its rating.
    - Resulting Relations:
        - **R1(id, name, rating):** This relation holds the FD `id -> name, rating` as `id` is the key for this relation.
        - **R2(id, popularity, rec?):** This relation retains the `id` to link back to R1 but does not necessarily carry any of the FDs from the initial table. It includes attributes not covered by the key FD in R1.

3. **Remaining Functional Dependencies:**
    - In **R1(id, name, rating)**, the FD `id -> name, rating` remains valid and is preserved post-decomposition because `id` is a candidate key for this relation.
    - **R2(id, popularity, rec?)** may need further analysis to determine if any new FDs apply specifically to this subset of attributes or if additional decomposition is necessary based on other FDs like `rating -> popularity` and potentially `popularity -> rec?`, if applicable.

Question 9

To decompose the `Employees(ID, Name, Car, ParkingLot, Job, Salary)` table based on the functional dependencies (FDs) provided, I'll first consider typical dependencies that might exist in an

employee database:

1. **ID → Name, Job, Salary**: This dependency suggests that an employee's ID uniquely determines their name, job, and salary.
2. **Car → ParkingLot**: If each car is associated with a specific parking lot, this dependency would apply.
3. **Name → ID**: Though less common due to potential name duplications, if names are unique within the organization, this might be a valid dependency.

Given these assumptions, here's how I would decompose the table:

Decomposition Process:

1. **First Decomposition based on ID → Name, Job, Salary**:
    - **Relation 1 (R1):** `ID, Name, Job, Salary`
    - **Relation 2 (R2):** `ID, Car, ParkingLot` (retaining ID to maintain a link to the employee details).
2. **Second Decomposition based on Car → ParkingLot (if valid)**:
    - **From R2 Decompose Further:**
    - **Relation 2a (R2a):** `Car, ParkingLot`
    - **Relation 2b (R2b):** `ID, Car` (Used to maintain which cars are assigned to which employees if Car → ParkingLot is applicable).

Remaining Functional Dependencies:

- **In R1 (ID, Name, Job, Salary)**:
    - The FD `ID → Name, Job, Salary` remains valid as ID is the key for R1 and determines all other attributes in this relation.
- **In R2a (Car, ParkingLot)**:
    - If `Car → ParkingLot` was assumed, then this FD is preserved here, with Car being the key for R2a.
- **In R2b (ID, Car)**:
    - This relation does not preserve any specific FDs but maintains the link between employees and their cars if multiple car assignments are possible.

This approach ensures normalization by minimizing redundancy and preventing update anomalies, allowing changes in one part of the database to reflect accurately across all related parts. If adjustments are needed based on the specific FDs you're working with, feel free to refine this decomposition accordingly.