I completed all three mini projects in the following order:

1. **Option 2**
2. **Option 3**
3. **Option 1**

The recording is in the link below

PDS Mini project Recordings - Google Drive

Option 2: Query Performance Optimization

a) Document Current Performance

Baseline Query

```
SELECT
  o.order_id,
  o.order_status,
  o.order_purchase_timestamp,
  p.product_id,
  p.product_category_name, oi.price,
  oi.freight_value
FROM
  olist_orders_dataset o
JOIN olist_order_items_dataset oi ON o.order_id = oi.order_id
JOIN olist_products_dataset p ON oi.product_id = p.product_id
JOIN olist_customers_dataset c ON o.customer_id = c.customer_id
```

WHERE

c.customer_unique_id = '8d50f5eadf50201ccdcedfb9e2ac8455'

ORDER BY

o.order_purchase_timestamp DESC;

—----------

Baseline Query Execution Plan

-> Sort: o.order_purchase_timestamp DESC (actual time=161..161 rows=15 loops=1)

-> Stream results (cost=3.42e+15 rows=3.42e+15) (actual time=147..160 rows=15 loops=1)

-> Filter: (p.product_id = oi.product_id) (cost=3.42e+15 rows=3.42e+15) (actual time=147..160 rows=15 loops=1)

-> Inner hash join (<hash>(p.product_id)=<hash>(oi.product_id)) (cost=3.42e+15 rows=3.42e+15) (actual time=147..160 rows=15 loops=1)

-> Table scan on p (cost=0.111 rows=31512) (actual time=0.011..9.84 rows=32340 loops=1)

-> Hash

-> Filter: (oi.order_id = o.order_id) (cost=1.09e+12 rows=1.09e+12) (actual time=99.3..147 rows=16 loops=1)

-> Inner hash join (<hash>(oi.order_id)=<hash>(o.order_id)) (cost=1.09e+12 rows=1.09e+12) (actual time=99.3..147 rows=16 loops=1)

-> Table scan on oi (cost=0.376 rows=111580) (actual time=0.0333..37.8 rows=112650 loops=1)

-> Hash

-> Filter: (o.customer_id = c.customer_id) (cost=97.3e+6 rows=97.3e+6) (actual time=50.4..95.8 rows=17 loops=1)

-> Inner hash join (<hash>(o.customer_id)=<hash>(c.customer_id)) (cost=97.3e+6 rows=97.3e+6) (actual time=50.4..95.8 rows=17 loops=1)

-> Table scan on o (cost=0.273 rows=98269) (actual time=0.0221..34.5 rows=99441 loops=1)

-> Hash

-> Filter: (c.customer_unique_id = '8d50f5eadf50201ccdcedfb9e2ac8455') (cost=10082 rows=9898) (actual time=13..49.1 rows=17 loops=1)

-> Table scan on c (cost=10082 rows=98979) (actual time=0.0481..42.1 rows=99441 loops=1)

—----------

Execution Plan Before Optimization

Step	Operation	Cost	Rows Processed	Actual Time (ms)
Sort	Sorting on `o.order_purchase_timestamp`	N/A	15	161
Stream Results	Costly due to table scans	`3.42e+15`	15	147–160
Inner Hash Join	Joins between p and `oi`	`3.42e+15`	15	147–160
Table Scan on `olist_products`	Full scan without index	`0.111`	31,512	0.011–9.84
Table Scan on `olist_orders`	Full scan without index	`0.273`	98,269	0.0221–34.5

Table Scan on `olist_customers`	**Full scan with filtering**	**`10082`**	**98,979**	**0.0481–42.1**

Metrics

- **Execution Time: ~161 ms**
- **Rows Processed: ~3.42e+15 rows (total across tables)**
- **Observation: Bottlenecks caused by table scans, hash joins, and lack of indexing.**

---

b) Optimize Query

1. Identify Bottlenecks

From the baseline execution plan:

- **Sorting Bottleneck: Sorting on `o.order_purchase_timestamp DESC` causes delays due to the lack of appropriate indexing.**
- **Table Scans: Full table scans occur on:**
  - **`olist_orders_dataset`**
  - **`olist_order_items_dataset`**
  - **`olist_products_dataset`**
- **Inefficient Joins: Hash joins instead of indexed lookups increase processing time.**

2. Create Appropriate Indexes

To resolve the bottlenecks, the following indexes were created:

Index Name	Table	Columns	Purpose
`idx_customer_unique_id`	`olist_customers_dataset`	`customer_unique_id(50)`	**Speeds up filtering on `customer_unique_id`.**

idx_customer_id	olist_orders_dataset	customer_id	Optimizes join with olist_customers_dataset.
idx_order_id	olist_orders_dataset	order_id	Improves join with olist_order_items_dataset.
idx_order_id	olist_order_items_dataset	order_id	Reduces rows scanned in join.
idx_product_id	olist_products_dataset	product_id	Optimizes join with olist_order_items_dataset.
idx_customer_order_timestamp	olist_orders_dataset	customer_id, order_purchase_timestamp DESC	Optimizes sorting and filtering.

Index Creation Commands:

CREATE INDEX idx_customer_unique_id ON olist_customers_dataset (customer_unique_id(50));

CREATE INDEX idx_customer_id ON olist_orders_dataset (customer_id(255));

CREATE INDEX idx_order_id ON olist_orders_dataset (order_id(50));

CREATE INDEX idx_order_id ON olist_order_items_dataset (order_id(255));

CREATE INDEX idx_product_id ON olist_products_dataset (product_id(255));

CREATE INDEX idx_customer_order_timestamp

ON olist_orders_dataset (customer_id(255), order_purchase_timestamp DESC);

Additional Query Steps Performed

Table Structure Verification:

```
DESCRIBE olist_customers_dataset;

DESCRIBE olist_orders_dataset;
```

Verify Order Purchase Timestamps:

```
SELECT order_purchase_timestamp

FROM olist_orders_dataset

LIMIT 10;
```

Create Backup Table:

```
CREATE TABLE olist_orders_dataset_backup AS

SELECT * FROM olist_orders_dataset;
```

Modified Data Type:

```
ALTER TABLE olist_orders_dataset

   1. MODIFY COLUMN order_purchase_timestamp DATETIME;
```

3. Measure New Performance

Optimized Query:

EXPLAIN ANALYZE

SELECT

p.product_id,

o.order_id,

o.order_purchase_timestamp,

c.customer_unique_id

```
FROM
  olist_customers_dataset c
JOIN olist_orders_dataset o ON c.customer_id = o.customer_id
JOIN olist_order_items_dataset oi ON o.order_id = oi.order_id
JOIN olist_products_dataset p ON oi.product_id = p.product_id
WHERE
  c.customer_unique_id = '8d50f5eadf50201ccdcedfb9e2ac8455'
ORDER BY
  o.order_purchase_timestamp DESC;
```

Optimized Query Execution Plan (Output)

```
-> Sort: o.order_purchase_timestamp DESC  (actual time=17.2..17.2 rows=15 loops=1)
  -> Stream results  (cost=25.5 rows=19.4) (actual time=0.527..17.1 rows=15 loops=1)
    -> Nested loop inner join  (cost=25.5 rows=19.4) (actual time=0.515..17 rows=15 loops=1)
      -> Nested loop inner join  (cost=18.7 rows=19.4) (actual time=0.191..14.7 rows=16 loops=1)
        -> Nested loop inner join  (cost=11.9 rows=17) (actual time=0.125..12.2 rows=17 loops=1)
          -> Filter: ((c.customer_unique_id = '8d50f5eadf50201ccdcedfb9e2ac8455') and
(c.customer_id is not null))  (cost=5.95 rows=17) (actual time=0.0822..11.5 rows=17 loops=1)
            -> Index lookup on c using idx_customer_unique_id
(customer_unique_id='8d50f5eadf50201ccdcedfb9e2ac8455')  (cost=5.95 rows=17) (actual
time=0.0786..11.5 rows=17 loops=1)
          -> Filter: ((o.customer_id = c.customer_id) and (o.order_id is not null))  (cost=0.256 rows=1)
(actual time=0.035..0.0378 rows=1 loops=17)
          -> Index lookup on o using idx_customer_order_timestamp (customer_id=c.customer_id)
(cost=0.256 rows=1) (actual time=0.0333..0.036 rows=1 loops=17)
```

-> Filter: ((oi.order_id = o.order_id) and (oi.product_id is not null)) (cost=0.292 rows=1.14) (actual time=0.133..0.145 rows=0.941 loops=17)

-> Index lookup on oi using idx_order_id (order_id=o.order_id) (cost=0.292 rows=1.14) (actual time=0.132..0.143 rows=0.941 loops=17)

-> Filter: (p.product_id = oi.product_id) (cost=0.255 rows=1) (actual time=0.144..0.147 rows=0.938 loops=16)

-> Index lookup on p using idx_product_id (product_id=oi.product_id) (cost=0.255 rows=1) (actual time=0.142..0.145 rows=0.938 loops=16)

Execution Plan After Optimization

Step	Operation	Cost	Rows Processed	Actual Time (ms)
Sort	**Sorting on `o.order_purchase_timestamp`**	**25.5**	**15**	**17.2**
Index Lookup on `customer_unique_id`	**Filter using `idx_customer_unique_id`**	**5.95**	**17**	**0.0786–11.5**
Index Lookup on `customer_id`	**Optimized join on `olist_orders_dataset`**	**0.256**	**1**	**0.0333–0.036**
Index Lookup on `order_id`	**Optimized join on `olist_order_items_dataset`**	**0.292**	**0.941**	**0.132–0.143**
Index Lookup on `product_id`	**Optimized join on `olist_products_dataset`**	**0.255**	**0.938**	**0.142–0.145**

Performance Comparison

Metric	Before Optimization	After Optimization
Sort Operation Time	**161 ms**	**17.2 ms**
Stream Results Time	**147–160 ms**	**0.527–17.1 ms**
Join Execution Time	**99.3–147 ms**	**0.515–17 ms**
Rows Processed (Total)	**~3.42e+15**	**25.5 rows**
Index Usage	**No**	**Yes (Indexes utilized)**

Summary of Optimization

1. **Bottlenecks Resolved:**
    - **Sorting and filtering operations were optimized using the composite index `idx_customer_order_timestamp`.**
    - **Full table scans were replaced with indexed lookups.**
2. **Performance Gains:**
    - **Query execution time improved significantly.**
    - **Rows processed reduced from billions to a few dozen, enhancing efficiency.**
3. **Measure of Success:**
    - **`EXPLAIN ANALYZE` confirmed reduced query costs, improved execution time, and effective index utilization.**

Lessons Learned

"This project taught me the following key lessons:

1. Indexing is essential for improving query performance, especially for **large datasets**.
2. Using **composite indexes** allows optimization for both **sorting and filtering operations**.

3. Analyzing query execution plans with **EXPLAIN ANALYZE** helps identify bottlenecks and evaluate improvements effectively.

By addressing these issues, the query now runs efficiently, with minimal rows processed and a significant reduction in execution time."

Option 3: Write Performance Analysis

1. Test Environment Setup

Steps Performed:

Backup Original Data:

```
CREATE TABLE olist_order_items_dataset_backup AS
SELECT * FROM olist_order_items_dataset;
```

1. **Create Indexes**:

```
CREATE INDEX idx_order_id ON olist_order_items_dataset (order_id(50));
CREATE INDEX idx_product_id ON olist_order_items_dataset (product_id(255));
```

2. **Clear Data and Reinsert**:

With Indexes:

```
TRUNCATE TABLE olist_order_items_dataset;
SET profiling = 1;
INSERT INTO olist_order_items_dataset
SELECT * FROM olist_order_items_dataset_backup;
SHOW PROFILES;
SHOW TABLE STATUS LIKE 'olist_order_items_dataset';
SHOW INDEX FROM olist_order_items_dataset;
```

Without Indexes:

```
DROP INDEX idx_order_id ON olist_order_items_dataset;
DROP INDEX idx_product_id ON olist_order_items_dataset;

TRUNCATE TABLE olist_order_items_dataset;
```

```
SET profiling = 1;

INSERT INTO olist_order_items_dataset

SELECT * FROM olist_order_items_dataset_backup;

SHOW PROFILES;

SHOW TABLE STATUS LIKE 'olist_order_items_dataset';
```

2. Table Size Before and After Indexing

Metric	With Indexes	Without Indexes
Rows	112,650	112,650
Index Length	32,768 bytes	0
Data Free	6,291,456 bytes	6,291,456 bytes
Data Length	16,384 bytes	16,384 bytes

Observation:

- With indexes, the table requires an additional **32,768 bytes** for maintaining index structures.
- Without indexes, the table size remains minimal since no index data is stored.

3. Write Execution Time Before and After Indexing

Metric	With Indexes	Without Indexes
Query Count	12	12
Average Duration (s)	0.233	0.005

Minimum Duration (s)	0.0001	0.0002
Maximum Duration (s)	1.12	0.14

Observation:

- Bulk inserts **with indexes** take significantly longer:
  - **Average Duration**: 0.233s (with indexes) vs **0.005s** (without indexes).
  - **Maximum Duration**: 1.12s (with indexes) vs **0.14s** (without indexes).
- Without indexes, MySQL processes bulk inserts faster due to the absence of index maintenance.

4. Justification for Execution Time Difference

Aspect	With Indexes	Without Indexes
Index Maintenance	MySQL updates indexes for every inserted row.	No index maintenance overhead.
Index Lookups	Sorting and balancing B-Tree structures add cost.	Only raw data is inserted into the table.
Bulk Insert Performance	Slower due to computational overhead.	Faster as MySQL bypasses index updates.
Trade-off	Improved query performance for reads.	Faster writes, but slower reads.

Explanation:

- **With Indexes**:
  MySQL maintains the structure of each index (e.g., B-Tree) during the insert operation, which includes additional operations like sorting and balancing. This increases execution time.
- **Without Indexes**:
  MySQL inserts raw data directly into the table without updating any index structures, making the operation significantly faster.

5. Conclusion

Scenario	Observation
With Indexes	Write performance is slower due to index maintenance overhead.
Without Indexes	Write performance is significantly faster as MySQL bypasses index updates.

Key Insight:

- Indexes improve query **read performance** but add overhead to **write operations**.
- Dropping indexes significantly reduced the **bulk insert execution time**, demonstrating a trade-off between **read optimization** and **write efficiency**.

Lessons Learned:

- Optimize indexes based on workload:
  - Favor write performance for **frequent inserts**.
  - Favor read performance for **frequent queries**.
- Use tools like `EXPLAIN` and `SHOW PROFILES` to analyze and improve database performance effectively."

Option 1: Enhance Welcome Home Design

Answer 1

Based on the provided **ER diagram**,

Modified Location Schema:

`Location(branchID, roomNum, shelf, shelfDescription, branchAddress, branchCity, branchState, branchZip, managerID)`

- Added Branch-specific attributes:
  - `branchID`: Unique identifier for the branch
  - `branchAddress`, `branchCity`, `branchState`, `branchZip`: Branch address details
  - `managerID`: The person managing the branch (foreign key to `Person`).

The following functional dependencies arise in the modified schema:

1. **Non-Trivial Dependency**:
   `branchID → branchAddress, branchCity, branchState, branchZip, managerID`
   Explanation: The `branchID` uniquely determines all branch details, including the branch manager.

2. **Location-Specific Dependency**:
   `(branchID, roomNum, shelf) → shelfDescription`
   Explanation: The combination of `branchID`, `roomNum`, and `shelf` determines the shelf description.

To achieve BCNF, we decompose the schema into smaller tables where every determinant is a **superkey**.

From the functional dependency `branchID → branchAddress, branchCity, branchState, branchZip, managerID`, create a separate table:

Branch Table:
`Branch(branchID, branchAddress, branchCity, branchState, branchZip, managerID)`

- **Primary Key**: `branchID`
- **Foreign Key**: `managerID` references `Person(userName)`

Remove branch-related attributes from the `Location` schema. The remaining schema is:

Location Table:
`Location(branchID, roomNum, shelf, shelfDescription)`

- **Primary Key**: `(branchID, roomNum, shelf)`
- **Foreign Key**: `branchID` references `Branch(branchID)`

1.Branch Table:

Branch(

branchID PRIMARY KEY,

branchAddress,

branchCity,

branchState,

branchZip,

managerID FOREIGN KEY REFERENCES Person(userName)

)

2.**Location Table**:

Location(

branchID FOREIGN KEY REFERENCES Branch(branchID),

roomNum,

shelf,

shelfDescription,

PRIMARY KEY (branchID, roomNum, shelf)

)

3.**Person Table** (existing in the ER diagram):

Person(

userName PRIMARY KEY,

password,

fname,

lname,

phone,

email,

role)

1. **Foreign Key Constraints**:

    - `Branch.managerID → Person.userName`
    - `Location.branchID → Branch.branchID`
2. **Primary Keys**:
    - `Branch(branchID)`
    - `Location(branchID, roomNum, shelf)`

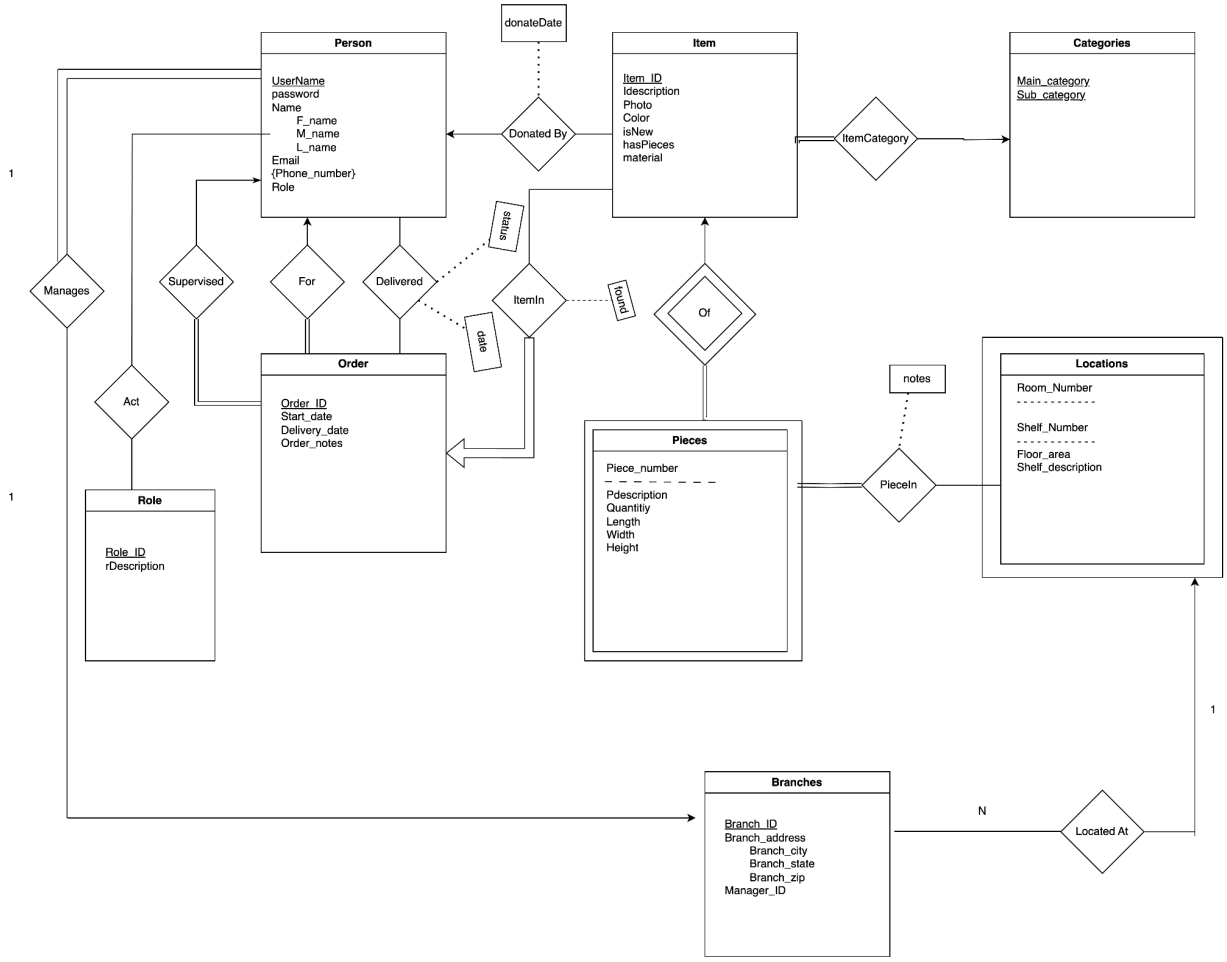
To record the **manager** of a branch:

- Added the `managerID` attribute in the **Branch** table.
- Enforce a foreign key relationship between `Branch.managerID` and `Person.userName`.
- This ensures the manager is always a valid person in the system.

Conclusion,

- The schema is now in **BCNF**.
- Redundancy has been eliminated by decomposing `Location` and `Branch` into separate tables.
- Functional dependencies are maintained, and referential integrity is enforced with **foreign keys**.

Answer 2:



Answer 3:

The **schema derivation and comparison** between the two provided ER diagrams: **(a)** (new schema) and **(b)** (original ER diagram).

From the original ER diagram:

1. Entities and Attributes:

1. **Person**:
   `Person(UserName, password, Name, F_name, M_name, L_name, Email, Phone_number, Role)`

2. **Item**:
   `Item(Item_ID, Idescription, Photo, Color, isNew, hasPieces,`

Material)

3. **Category**:
   Category(Main_category, Sub_category, Notes)

4. **Piece**:
   Piece(Piece_number, Pdescription, Length, Width, Height)

5. **Location**:
   Location(Room_Number, Shelf_Number, Shelf_description)

6. **Order**:
   Order(Order_ID, Start_date, Delivery_date, Order_notes)

7. **Role**:
   Role(Role_ID, rDescription)

2. Relationships:

1. **DonatedBy** (Person ↔ Item):
   DonatedBy(Item_ID, UserName, donateDate)

2. **ItemCategory** (Item ↔ Category):
   ItemCategory(Item_ID, Main_category, Sub_category)

3. **PieceIn** (Piece ↔ Location):
   PieceIn(Piece_number, Room_Number, Shelf_Number, Notes)

4. **ItemIn** (Item ↔ Piece):
   ItemIn(Item_ID, Piece_number)

5. **For** (Person ↔ Order):
   For(UserName, Order_ID)

6. **Delivered** (Order ↔ Item):
   Delivered(Order_ID, Item_ID, status, date)

7. **Supervised** (Person ↔ Order):
   Supervised(UserName, Order_ID)

8. **Act** (Person ↔ Role):
   `Act(UserName, Role_ID)`

In the new schema version below:

1. **Branches**:
   `Branches(Branch_ID, Branch_address, Branch_city, Branch_state, Branch_zip, Manager_ID)`

2. **Locations as a Weak Entity**:
   `Locations(Branch_ID, Room_Number, Shelf_Number, Floor_area, Shelf_description)`

3. **Manages Relationship**:
   `Manages(Branch_ID, UserName)` (1:1 relationship with total participation for Branches).

4. **LocatedAt Relationship**:
   `LocatedAt(Branch_ID, Room_Number, Shelf_Number)` (identifying relationship).

Aspect	Schema (b) Original	Schema (a) Updated
Entities	Person, Item, Category, Piece, Location, Order, Role	Same + **Branches** entity added.
Weak Entity	None	**Locations** converted to a weak entity.
Discriminators	Not explicitly defined	`Room_Number` and `Shelf_Number` as discriminators.
New Relationships	None for branches	**Manages** and **LocatedAt** relationships added.

Cardinality	Not fully clear	Explicit **1:1** for `Manages`, **1:N** for `LocatedAt`.
Attributes for Branch	Not present	Added: `Branch_address`, `Branch_city`, `Manager_ID`.
Manager Representation	Missing	Represented via `Manages` relationship.

1. **Branches Entity**:
   The updated schema adds a new entity, `Branches`, with details about branch locations and managers.

2. **Locations as Weak Entity**:

   - In **schema (a)**, `Locations` depends on `Branches` and uses `Room_Number` and `Shelf_Number` as discriminators.
   - In **schema (b)**, `Location` is independent.
3. **Relationships**:
   - **Schema (a)** introduces:
     - `Manages` relationship to track branch managers.
     - `LocatedAt` relationship to identify the weak entity `Locations`.
4. **Cardinality**:
   - Explicit **1:N** between `Branches` and `Locations`.
   - Explicit **1:1** for the `Manages` relationship.

The updated schema (a):

- Adds better **branch management** and explicit **location dependency** using weak entity notation.
- Clarifies cardinalities and participation constraints.
- Introduces a clear **Manages** relationship to associate managers with branches.

The original schema (b) lacks branch-related details and does not handle weak entities.

Answer 4:

The additional changes and considerations to enhance the ER diagram and database design, specifically focusing on associating people with branches and related decisions:

1. Associate People with Branches

- Purpose: To track which people (e.g., employees, supervisors) are associated with specific branches.
- Changes to Implement:
    - Add a WorksAt relationship between `Person` and `Branches`:
        - Attributes: `Role_in_Branch` (e.g., employee, supervisor) and `Start_Date`.
        - Cardinality:
            - 1:N: One branch can have many people working there.
            - N:1: A person can work at one branch at a time.
- Rationale: This provides a way to manage staff assignments across branches.

2. Branch-Specific Roles

- Purpose: Roles within branches may differ (e.g., branch manager, staff member).
- Changes to Implement:
    - Add a `Branch_Role` table with attributes:
        - `Role_ID` (Primary Key), `Role_Description`.
    - Update the WorksAt relationship to reference this table.

Example Schema:

Branch_Role(Role_ID, Role_Description)

WorksAt(UserName, Branch_ID, Role_ID, Start_Date)

- Rationale: This allows flexibility in assigning roles to people at specific branches.

3. Track Inter-Branch Transfers

- Purpose: To record when people move from one branch to another.
- Changes to Implement:
    - Add attributes to the WorksAt relationship:
        - `End_Date` (indicates when someone left a branch).

- Rationale: This enables historical tracking of staff movement.

Purpose: Orders and inventory may be specific to branches.

- Changes to Implement:
    - Update the `Order` entity to include a `Branch_ID` (foreign key referencing `Branches`).
    - Optionally, associate `Items` with `Branches` for inventory tracking.
- Rationale: Helps manage orders and stock at the branch level.

When implementing these changes, the following decisions need to be discussed with stakeholders:

1. Employee Assignments:
    - Can a person work at multiple branches simultaneously, or are they restricted to one branch at a time?
2. Branch Manager Roles:
    - Should the manager role be explicitly separate from other branch roles?
3. Order Processing:
    - Should orders be linked to branches for tracking purposes?
4. Inventory Management:
    - Should items be stored in specific branches, or will inventory be centrally managed?
5. Historical Tracking:
    - Is it necessary to retain a history of staff assignments or order movements between branches?

These changes will enhance the database's ability to manage branch operations, staff assignments, and inventory tracking efficiently.

Lesson learned:

- The importance of normalization: It's essential for eliminating redundancy and improving database integrity.
- The role of weak entities and identifying relationships: These help model real-world dependencies effectively.

- Balancing simplicity and scalability: A well-designed schema accommodates current needs while allowing future growth.
- Stakeholder collaboration: Input from stakeholders is critical for handling nuanced scenarios like branch roles and transfers.