NYU, Tandon School of Engineering

November 20, 2024

CS-GY 6083

Principles of Database Systems

Section A, Fall 2024

Homework #4

Submitted by:

Khwaab Thareja

N15911999

Kt3180

Guided By: Prof Phyllis Frankl

Answer 1

Brand	ID	Sz	Clr	Email	ODate	BPrice	PPaid	Stat	InStk	Ord	BPts	Phone
Nike	A001	9	Black	khwaab@ex.com	2024-11-10	100	90	Deliv	50	1	200	123-456-789
Nike	A001	9	Black	ansh@ex.com	2024-11-12	100	95	Shipped	50	2	150	234-567-890
Adidas	Samba	10	White	raj@ex.com	2024-11-11	120	120	Ordered	20	1	300	345-678-901
Adidas	Samba	9	Black	Null	Null	120	Null	Null	15	Null	0	Null
Puma	C003	8	Blue	neeha@ex.com	Null	90	Null	Null	30	Null	250	456-789-012
Puma	C003	9	Red	shourya@ex.com	2024-11-16	90	88	Deliv	20	1	180	567-890-123

Requirements:

1. **Two customers have ordered the same brand/style/color/size:**
    - Khwaab and Ansh both ordered Nike shoes, style A001, size 9, color Black. This shows that the table can handle multiple customers ordering the same shoe type.
2. **Multiple colors and/or sizes for a brand/style:**
    - The Puma brand with styleID C003 appears in different colors (Blue and Red), demonstrating that the schema supports tracking different variations of the same brand/style.
3. **Customer without an order:**

- Neeha@ex.com is listed in the table with no orderDate, pricePaid, or other order details. This indicates that it's possible to have customers represented without any orders.

4. **Shoes in stock that haven't been ordered:**
   - Adidas shoes with styleID Samba, size 9, color Black are in stock (qInStock > 0) but have qOrdered = 0, meaning no one has ordered them yet.

Issues Highlighted by This Schema:

1. **Repetition and Redundancy:**
   - The same information, such as brand, styleID, and basePrice, is stored in multiple rows. This increases storage needs and the potential for errors if data is updated inconsistently.
2. **Combining Different Types of Data:**
   - Customer data (like email, phone, bonusPts) is mixed with inventory data (such as qInStock and basePrice), making the schema harder to manage. This structure does not clearly separate orders from the inventory.
3. **Null or Missing Values:**
   - Rows like the Adidas B002 (size 9, Black) entry show missing values for email, orderDate, and pricePaid, indicating incomplete data when shoes are only in stock but not ordered. This can create problems for data analysis and reporting.
4. **Data Consistency Issues:**
   - The basePrice is stored redundantly for each instance of a shoe type with the same brand and styleID. If the basePrice changes, updating it in every related row becomes necessary, which can lead to inconsistencies.

Anomalies Observed:

1. **Delete Anomaly:**
   - If a customer's order is removed from the table (e.g., Khwaab's order for Nike shoes), it could also delete important data about the shoe's inventory, such as qInStock and basePrice. This highlights how data loss can occur if orders and inventory are managed together.
2. **Insert Anomaly:**
   - Adding a new type of shoe (e.g., black size 9 Adidas Sambas) without linking it to an order is difficult because email, orderDate, and other order-specific fields need to be filled. This complicates adding new inventory entries when no customer has yet ordered that shoe.

Answer 2

A functional dependency is called trivial if the right-hand side (dependent attributes) is already part of the left-hand side (determinant attributes).

Example: In the ShoeOrder table, an example is:

- {brand,styleID,size,color, qInStock} → {qInStock}

Since qInStock is already included in the left-hand side, this is a trivial dependency.

A superkey is a set of attributes that can uniquely identify a row in the table.

Example: In the ShoeOrder table, email,orderDate,brand,styleID,size,color together can uniquely identify a row. This means:

- {email,orderDate,brand,styleID,size,color}→ {basePrice,pricePaid,status,qInStock,qOrdered, bonusPts,phone}

This dependency does not violate BCNF because the left-hand side is a superkey.

Answer 3

a. Dependency for Shoe Type and Inventory:

Each unique shoe type is identified by brand, styleID, size, and color, which determines the quantity in stock (qInStock).

Functional Dependency:

{brand,styleID,size,color} → {qInStock}

b. Dependency for Base Price:

The basePrice of a shoe depends on brand and styleID, so all shoes with the same brand and styleID will have the same basePrice, no matter the size or color.

Functional Dependency:

{brand,styleID}→{basePrice}

c. Dependency for Customer Information:

A customer's email uniquely identifies them and determines their phone number and bonusPts.

Functional Dependency:

{email}→{phone,bonusPts}

These show how the attributes relate to each other in the ShoeOrder table.

Answer 4

To find the canonical cover of functional dependencies, we need to simplify the dependencies by removing any unnecessary parts while keeping their original meaning.

Original Functional Dependencies:

1. {brand,styleID,size,color}→{qInStock}
2. {brand,styleID}→{basePrice}
3. {email}→{phone,bonusPts}

Steps for Simplification:

1. Remove extra attributes:
    - Check if any attributes on the left side can be removed without changing what the dependencies mean.
2. Break down dependencies:
    - Split dependencies with more than one attribute on the right side (e.g., split {email}→{phone,bonusPts}
3. Combine similar dependencies:
    - Group dependencies that have the same left side.

After simplifying, we have:

1. {brand,styleID,size,color}→{qInStock}
2. {brand,styleID}→{basePrice}
3. Split {email}→{phone,bonusPts} into:
    - {email}→{phone}
    - {email}→{bonusPts}

Final Canonical Cover:

The final set of simplified dependencies is:

1. {brand,styleID,size,color}→{qInStock}
2. {brand,styleID}→{basePrice}
3. {email}→{phone}
4. {email}→{bonusPts}

This is the minimal set of dependencies that keeps all the original information but without any unnecessary parts.

Answer 5:

To find a candidate key for the ShoeOrder table, we need to identify the smallest set of attributes that can uniquely identify each row.

Functional Dependencies:

1. {brand,styleID,size,color}→{qInStock}
2. {brand,styleID}→{basePrice}
3. {email}→{phone,bonusPts}

A candidate key should be a minimal set of attributes that can uniquely identify each record.

1. Shoe Details:
    - The combination of brand, styleID, size, and color uniquely identifies a type of shoe.
2. Customer Details:
    - email is unique for each customer and identifies their phone and bonusPts.
3. Order Details:
    - Adding orderDate to email helps identify specific orders from the same customer.

The combination of email, orderDate, brand, styleID, size, and color is a candidate key because it can uniquely identify each row in the table.

Candidate Key:

{email,orderDate,brand,styleID,size,color}

Unique: This set ensures each record is unique by including customer details (email), order details (orderDate), and specific shoe details (brand, styleID, size, color).

Minimal: If any attribute is removed, it won't be able to uniquely identify a record.

Therefore candidate key for the ShoeOrder table is:

{email,orderDate,brand,styleID,size,color}

Answer 6

To show that the ShoeOrder schema is not in BCNF, we need to check if the left side of each functional dependency is a superkey (a set of attributes that uniquely identifies a row).

Functional Dependencies:

1. {brand,styleID,size,color}→{qInStock}
2. {brand,styleID}→{basePrice}
3. {email}→{phone,bonusPts}

Checking BCNF:

1. Dependency 1: {brand,styleID,size,color}→{qInStock}
    - Is it a superkey? No, because this combination only identifies a specific type of shoe, not the whole row. This violates BCNF.
2. Dependency 2: {brand,styleID}→{basePrice
    - Is it a superkey? No, as brand and styleID only determine basePrice but do not identify the full record. This also violates BCNF.
3. Dependency 3: {email}→{phone,bonusPts}
    - Is it a superkey? No, email identifies customer information but not the full row with order and shoe details. This does not meet BCNF.

The ShoeOrder schema is not in BCNF because there are dependencies where the left side is not a superkey. To put the schema in BCNF, we would need to break it down into smaller tables where each dependency's left side is a superkey.

Answer 7

To decompose the ShoeOrder schema into a set of schemas that are in BCNF, we will go step by step, identifying functional dependencies that violate BCNF and decomposing the schema accordingly.

The following functional dependencies violate BCNF because their left-hand sides are not superkeys:

1. {brand,styleID,size,color} → {qInStock}
2. {brand,styleID} → {basePrice}
3. {email} → {phone,bonusPts}

No we will have to Decompose Based on Violating Dependencies

Decomposition 1:

Functional Dependency Used: {brand,styleID}→{basePrice}

Resulting Schemas:

1. ShoeDetails(brand, styleID, basePrice)
2. OrderDetails(brand, styleID, size, color, email, orderDate, pricePaid, status, qInStock, qOrdered, bonusPts, phone)

Decomposition 2:

Schema to Decompose: OrderDetails

Functional Dependency Used: {email}→{phone,bonusPts}

Resulting Schemas:

1. Customer(email, phone, bonusPts)
2. Order(brand, styleID, size, color, email, orderDate, pricePaid, status, qInStock, qOrdered)

Decomposition 3:

Schema to Decompose: Order

Functional Dependency Used: {brand,styleID,size,color}→{qInStock}

Resulting Schemas:

1. Inventory(brand, styleID, size, color, qInStock)
2. OrderRecord(brand, styleID, size, color, email, orderDate, pricePaid, status, qOrdered)

Final BCNF Decomposed Schemas:

1. ShoeDetails(brand, styleID, basePrice):
    - Represents unique shoe types and their base prices.
2. Customer(email, phone, bonusPts):
    - Contains customer contact information and bonus points.
3. Inventory(brand, styleID, size, color, qInStock):
    - Keeps track of inventory quantities for each type of shoe.
4. OrderRecord(brand, styleID, size, color, email, orderDate, pricePaid, status, qOrdered):
    - Represents customer orders and associated details.

Hence, We decomposed the original ShoeOrder schema into four schemas:

1. ShoeDetails(brand, styleID, basePrice)
2. Customer(email, phone, bonusPts)
3. Inventory(brand, styleID, size, color, qInStock)
4. OrderRecord(brand, styleID, size, color, email, orderDate, pricePaid, status, qOrdered)

Each of these schemas is now in BCNF, as each non-trivial functional dependency's left side is a superkey.

Answer 8:

Decomposed BCNF Schemas:

1. ShoeDetails(brand, styleID, basePrice)
2. Customer(email, phone, bonusPts)
3. Inventory(brand, styleID, size, color, qInStock)
4. OrderRecord(brand, styleID, size, color, email, orderDate, pricePaid, status, qOrdered)

Data Stored in Decomposed Schemas:

1. ShoeDetails Table:

Brand	ID	BPrice
Nike	A001	100
Adidas	Samba	120
Puma	C003	90

2. Customer Table:

Email	Phone	BPts
khwaab@ex.com	123-456-789	200
ansh@ex.com	234-567-890	150
raj@ex.com	345-678-901	300
neeha@ex.com	456-789-012	250
shourya@ex.com	567-890-123	180

3. Inventory Table:

Brand	ID	Sz	Clr	InStk
Nike	A001	9	Black	50
Adidas	Samba	10	White	20
Adidas	Samba	9	Black	15
Puma	C003	8	Blue	30
Puma	C003	9	Red	20

4. OrderRecord Table:

Brand	ID	Sz	Clr	Email	ODate	PPaid	Stat	Ord
Nike	A001	9	Black	khwaab@ex.com	2024-11-10	90	Deliv	1
Nike	A001	9	Black	ansh@ex.com	2024-11-12	95	Shipped	2
Adidas	B002	10	White	raj@ex.com	2024-11-11	120	Ordered	1
Puma	C003	8	Blue	neeha@ex.com	2024-11-15	85	Deliv	1
Puma	C003	9	Red	shourya@ex.com	2024-11-16	88	Deliv	1

Addressing Anomalies:

Redundancy: The decomposed tables remove data duplication. For example, basePrice is stored once in ShoeDetails, and customer details are in Customer, reducing repetition.

Update Anomalies: In the decomposed schema, changes (e.g., updating phone or basePrice) only need to be made in one table, preventing inconsistencies.

Insertion Anomalies: New shoe types, customers, or inventory entries can be added without needing complete order data. For example, a new shoe type can be added to ShoeDetails alone.

Deletion Anomalies: Deleting an order from OrderRecord will not remove information from ShoeDetails, Customer, or Inventory, preserving critical data.

This decomposition ensures data integrity, reduces anomalies, and maintains BCNF standards.

Answer 9

a. The new rule means that for all shoes with the same brand and styleID bought on the same orderDate, the pricePaid is always the same. This can be written as:

{brand,styleID,orderDate}→{pricePaid}

b. We need to check if this new dependency breaks BCNF in the tables we created earlier and decompose further if necessary.

Current Tables:

1. ShoeDetails(brand, styleID, basePrice)
2. Customer(email, phone, bonusPts)
3. Inventory(brand, styleID, size, color, qInStock)
4. OrderRecord(brand, styleID, size, color, email, orderDate, pricePaid, status, qOrdered)

BCNF Check: The new dependency {brand,styleID,orderDate}→{pricePaid} applies to OrderRecord and is not a superkey, so it violates BCNF.

New Tables:

1. OrderPrice(brand, styleID, orderDate, pricePaid) — Stores consistent pricePaid for each brand, styleID, and orderDate.
2. OrderRecord(brand, styleID, size, color, email, orderDate, status, qOrdered) — Holds the remaining order information.

c. The new decomposition is not dependency-preserving because all dependencies need a join between OrderPrice and OrderDetails to check.

The decomposed schema satisfies **BCNF**, but it is **not dependency-preserving** because the candidate key {email,orderDate,brand,styleID,size,color} is split across multiple tables. This means some dependencies involving the candidate key, while theoretically intact, cannot be directly enforced without recombining tables.

Answer 10

To make sure the schema follows BCNF with the new rule, we need to handle the dependency where all purchases on the same orderDate with the same basePrice have the same pricePaid.

a. Functional Dependency:

The new rule creates this dependency:

{orderDate,basePrice}→{pricePaid}

This means that on any given orderDate, if the basePrice is the same, then the pricePaid must also be the same.

b. Checking for BCNF and Decomposition:

We need to check if this dependency breaks BCNF in the current tables from part (7) and decompose if needed.

Current Tables:

1. ShoeDetails(brand, styleID, basePrice)
2. Customer(email, phone, bonusPts)
3. Inventory(brand, styleID, size, color, qInStock)
4. OrderRecord(brand, styleID, size, color, email, orderDate, pricePaid, status, qOrdered)

BCNF Check: The dependency {orderDate,basePrice}→{pricePaid} applies to OrderRecord.

BCNF Violation:

- {orderDate,basePrice}\{orderDate, basePrice\}{orderDate,basePrice} is not a superkey in OrderRecord because it doesn't uniquely identify all columns like email or shoe details. This breaks BCNF.

Decomposition: We need to split OrderRecord:

New Tables:

1. OrderPrice(orderDate, basePrice, pricePaid) — Keeps pricePaid consistent for each orderDate and basePrice.
2. OrderDetails(brand, styleID, size, color, email, orderDate, basePrice, status, qOrdered) — Contains other order information.

c. Is It Dependency-Preserving?

This decomposition is not fully dependency-preserving because to check the dependency {orderDate,basePrice}→{pricePaid}, you need to join OrderPrice and OrderDetails.

Explanation:

- The dependency involving pricePaid requires data from both tables to verify.
- Dependencies like {email}→{phone,bonusPts} stay preserved because Customer is unchanged.

Final BCNF Tables:

1. ShoeDetails(brand, styleID, basePrice)
2. Customer(email, phone, bonusPts)
3. Inventory(brand, styleID, size, color, qInStock)
4. OrderPrice(orderDate, basePrice, pricePaid)
5. OrderDetails(brand, styleID, size, color, email, orderDate, basePrice, status, qOrdered)

The schema now meets BCNF requirements for the new dependency {orderDate,basePrice}→{pricePaid}. However, it is not dependency-preserving for this rule, as checking it needs joining tables. This is a common trade-off when ensuring BCNF.