Database Project Phase 2

Final Report

Topic: Urban Point

Team: 17

Mentor: Ghada Al-Naimi

Authors:

Iroda Ibrohimova
Madina Mirzatayeva

Carnegie Mellon University Qatar

Fall 2025

Contents

1. Introduction

UrbanPoint is a digital platform that connects customers with local businesses by providing redeemable offers, brand information, subscription plans, and a seamless in-app payment and redemption experience. To support these operations, UrbanPoint requires a well-designed database capable of managing user accounts, business entities, outlet locations, offers, payments, redemptions, and analytical reporting features.

This report presents the full design and implementation of the UrbanPoint database as part of Phase 2 of the course project. Building on the initial conceptual design from Phase 1, we refine the UML model, develop a complete relational schema, identify functional dependencies, justify normalization up to BCNF, and translate the final design into a physical model. In addition, we define a comprehensive set of user stories (operational and analytical) that illustrate how different system roles interact with the database to perform real tasks such as browsing offers, adding outlets, analyzing revenue patterns, and generating targeted marketing lists.

The goal of this report is to provide a clear and complete explanation of how the UrbanPoint database was modeled, structured, and validated. By following the standard database design process and applying normalization principles, we ensure that the final schema is logically sound, free of redundancy, supports efficient queries, and accurately reflects the functional needs of the platform. The result is a scalable and maintainable database foundation suitable for both day-to-day operations and advanced analytical workloads.

2. UML Conceptual Model

Figure 1 shows our final UML class diagram for the UrbanPoint system, including entities, attributes, generalization hierarchies, and associations.

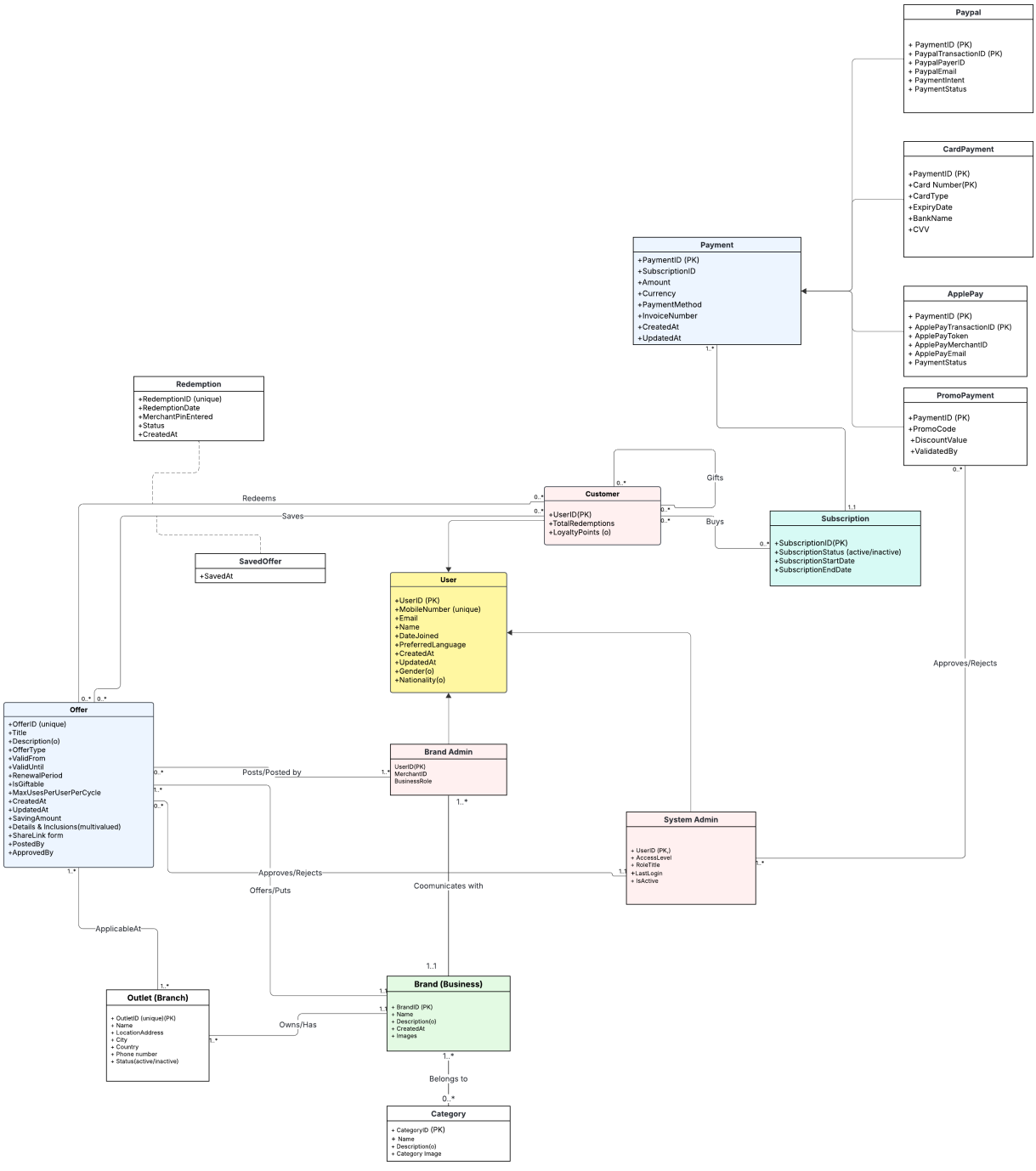

Figure 1: UML conceptual model for UrbanPoint.

2.2 Updated Final User Stories

Story Type	Actor	I Want	So That
Complex Operational	Customer	To create an account (Name + Mobile + Email)	I can start redeeming offers.
Simple Operational	Customer	To browse brands by category	I can easily discover businesses matching my interests.
Simple Operational	Customer	To view active outlets in a specific city	I can find open business locations where I plan to visit.
Simple Analytical	Customer	To see how many offers I have saved overall	I can understand how actively I use the save-for-later feature.
Simple Operational	Brand Admin	To add a new outlet branch	Customers can find our physical locations.
Complex Operational	Brand Admin	To view all outlets for my brand with full details	I can get a full overview of my brand's presence.
Complex Analytical (NEW)	System Admin	To view revenue and transaction counts grouped by payment method	I can identify the most popular channel and plan maintenance or promotions.
Complex Analytical (Window Function)	System Admin	To view all outlets ranked by their total redemptions	I can identify the highest-performing outlets across the platform.
Simple Operational	System Admin	To generate a contact list of users filtered by a specific gender	I can send targeted SMS marketing campaigns.
Simple Analytical (Window Function)	Customer	To see the maximum number of redemptions I made in a single day	I can understand when I was most active on the app.

Table 1: Final list of user stories for UrbanPoint.

3. Relational Model

Following the conversion rules discussed in class, we mapped all entities, relationships, and generalization structures in our UML conceptual model into relations. We use the following adornments:

- single underline: primary key (PK)
- dotted underline: foreign key (FK)
- double underline: attribute that is both PK and FK

3.1 Entity Relations

User(UserID, MobileNumber, Email, Name, DateJoined, PreferredLanguage, CreatedAt, UpdatedAt, Gender, Nationality)
Customer(UserID, TotalRedemptions, LoyaltyPoints)
BrandAdmin(UserID, MerchantID, BusinessRole)
SystemAdmin(UserID, AccessLevel, RoleTitle, LastLogin, IsActive)
Category(CategoryID, Name, Description, CategoryImage)
Brand(BrandID, CategoryID, Name, Description, CreatedAt, Images)
Outlet(OutletID, BrandID, Name, LocationAddress, City, Country, PhoneNumber, Status)

3.2 Offer and Applicability Relations

Offer(OfferID, BrandID, Title, Description, OfferType, ValidFrom, ValidUntil, RenewalPeriod, IsGiftable, MaxUsesPerUserPerCycle, CreatedAt, UpdatedAt, SavingAmount, DetailsAndInclusions, ShareLink, PostedBy, ApprovedBy)
Offer_Outlet(OfferID, OutletID)

3.3 Customer–Offer Interaction Relations

SavedOffer(UserID, OfferID, SavedAt)
Gifting(GiftID, OfferID, UserID, GiftDate, RecipientPhone, GiftStatus)
Redemption(RedemptionID, OfferID, UserID, OutletID, RedemptionDate, MerchantPinEntered, Status, CreatedAt)

3.4 Subscription and Payment Relations

Subscription(SubscriptionID, UserID, SubscriptionStatus, SubscriptionStartDate, SubscriptionEndDate)
Payment(PaymentID, SubscriptionID, Amount, Currency, PaymentMethod, InvoiceNumber, CreatedAt, UpdatedAt)
PaypalPayment(PaymentID, PaypalTransactionID, PaypalPayerID, PaypalEmail, PaymentIntent, PaymentStatus)
CardPayment(PaymentID, CardNumber, CardType, ExpiryDate, BankName, CVV)
ApplePay(PaymentID, ApplePayTransactionID, ApplePayToken, ApplePayMerchantID, ApplePayEmail, PaymentStatus)
PromoPayment(PaymentID, PromoCode, DiscountValue, ValidatedBy)

4. Functional Dependencies

The functional dependencies (FDs) are derived from the domain semantics of UrbanPoint. Each FD is written in the form $X \rightarrow Y$. We also indicate the highest normal form each relation satisfies. Our standing assumption is that the primary key of each relation uniquely identifies a tuple and that no additional business rules introduce extra dependencies beyond those listed.

4.1 Users and Role Subtypes

User
FDs:

- UserID → MobileNumber, Email, Name, DateJoined, PreferredLanguage, CreatedAt, UpdatedAt, Gender, Nationality
- MobileNumber is unique ⇒ MobileNumber → UserID (alternate key)

Normal form: **BCNF** (all determinants are keys/candidate keys).

Customer
FD: UserID → TotalRedemptions, LoyaltyPoints
Normal form: **BCNF**.

BrandAdmin
FD: UserID → MerchantID, BusinessRole
Normal form: **BCNF**.

SystemAdmin
FD: UserID → AccessLevel, RoleTitle, LastLogin, IsActive
Normal form: **BCNF**.

4.2 Business Entities

Category
FD: CategoryID → Name, Description, CategoryImage
Normal form: **BCNF**.

Brand
FD: BrandID → CategoryID, Name, Description, CreatedAt, Images
Normal form: **BCNF**. (BrandID is the only determinant; the foreign key CategoryID does not determine BrandID.)

Outlet
FD: OutletID → BrandID, Name, LocationAddress, City, Country, PhoneNumber, Status
Normal form: **BCNF**.

4.3 Offers and Applicability

Offer
FD: OfferID → BrandID, Title, Description, OfferType, ValidFrom, ValidUntil, RenewalPeriod,
IsGiftable, MaxUsesPerUserPerCycle, CreatedAt, UpdatedAt, SavingAmount,
DetailsAndInclusions, ShareLink, PostedBy, ApprovedBy
Normal form: **BCNF** (OfferID is the only determinant).

Offer_Outlet
FDs:

- (OfferID, OutletID) → OfferID, OutletID (only trivial FDs)

Normal form: **BCNF**. (The composite key is the only determinant.)

4.4 Customer–Offer Interactions

SavedOffer
FD: (UserID, OfferID) → SavedAt
Normal form: **BCNF**. (Neither UserID nor OfferID alone determines the tuple.)

Gifting
FD: GiftID → OfferID, UserID, GiftDate, RecipientPhone, GiftStatus
Normal form: **BCNF**. (GiftID is a surrogate key for each gifting event.)

Redemption
FD: RedemptionID → OfferID, UserID, OutletID, RedemptionDate, MerchantPinEntered, Status, CreatedAt
Normal form: **BCNF**.

4.5 Subscription and Payments

Subscription
FD: SubscriptionID → UserID, SubscriptionStatus, SubscriptionStartDate, SubscriptionEndDate
Normal form: **BCNF**.

Payment
FD: PaymentID → SubscriptionID, Amount, Currency, PaymentMethod, InvoiceNumber, CreatedAt, UpdatedAt
Normal form: **BCNF**.

PaypalPayment
FD: PaymentID → PaypalTransactionID, PaypalPayerID, PaypalEmail, PaymentIntent, PaymentStatus
Normal form: **BCNF**.

CardPayment
FD: PaymentID → CardNumber, CardType, ExpiryDate, BankName, CVV
Normal form: **BCNF**.

ApplePay
FD: PaymentID → ApplePayTransactionID, ApplePayToken, ApplePayMerchantID, ApplePayEmail, PaymentStatus
Normal form: **BCNF**.

PromoPayment
FDs:

- PaymentID → PromoCode, DiscountValue, ValidatedBy

Normal form: **BCNF**.

5. Normalization

Our final relational schema is already in **BCNF** because, from the earliest stages of converting the UML conceptual model into the relational and physical schema, we systematically eliminated all partial, transitive, and hidden dependencies. We enforced clear primary keys, represented all many-to-many relationships using associative entities, and ensured each generalization hierarchy used a PK–FK structure. As a result, for every functional dependency $X \rightarrow Y$ that holds in any relation, the determinant X is a **superkey**, satisfying the definition of BCNF.

This section explains **why each relation individually satisfies BCNF**, using the functional dependencies presented in Section 4.

5.1 BCNF Justification for Individual Relations

User

- FD: UserID $\rightarrow$ all other attributes
- MobileNumber is also unique: MobileNumber $\rightarrow$ UserID

Both *UserID* and *MobileNumber* are candidate keys; every determinant is a superkey. **Therefore, User is in BCNF**.

Customer

- FD: UserID $\rightarrow$ TotalRedemptions, LoyaltyPoints

UserID is the primary key and the only determinant. **Customer is in BCNF**.

BrandAdmin

- FD: UserID $\rightarrow$ MerchantID, BusinessRole

UserID is the PK and determinant. **BrandAdmin is in BCNF**.

SystemAdmin

- FD: UserID $\rightarrow$ AccessLevel, RoleTitle, LastLogin, IsActive

UserID is the PK and the only determinant. **SystemAdmin is in BCNF**.

Category

- FD: CategoryID $\rightarrow$ Name, Description, CategoryImage

CategoryID is the PK and a superkey. **Category is in BCNF**.

Brand

- FD: BrandID → CategoryID, Name, Description, CreatedAt, Images

BrandID is the only determinant; the foreign key CategoryID does not determine any other attribute. **Brand is in BCNF**.

Outlet

- FD: OutletID → BrandID, Name, LocationAddress, City, Country, PhoneNumber, Status

OutletID is the PK; no attribute determines OutletID. **Outlet is in BCNF**.

Offer

- FD: OfferID → all non-key attributes

OfferID is the PK and the only determinant. **Offer is in BCNF**.

Offer_Outlet

- Only FDs are trivial: (OfferID, OutletID) → OfferID, OutletID

The composite key is the only determinant. **Offer_Outlet is in BCNF**.

SavedOffer

- FD: (UserID, OfferID) → SavedAt

The composite key determines all attributes; no part of the key determines the other part. **SavedOffer is in BCNF**.

Gifting

- FD: GiftID → OfferID, UserID, GiftDate, RecipientPhone, GiftStatus

GiftID is a surrogate key and is the only determinant. **Gifting is in BCNF**.

Redemption

- FD: RedemptionID → OfferID, UserID, OutletID, RedemptionDate, MerchantPinEntered, Status, CreatedAt

RedemptionID is the PK and the only determinant. **Redemption is in BCNF**.

Subscription

- FD: SubscriptionID → UserID, SubscriptionStatus, SubscriptionStartDate, SubscriptionEndDate

SubscriptionID is the PK; no other determinant exists. **Subscription is in BCNF**.

Payment

- FD: PaymentID → SubscriptionID, Amount, Currency, PaymentMethod, InvoiceNumber, CreatedAt, UpdatedAt

PaymentID is the PK. **Payment is in BCNF**.

PaypalPayment

- FD: PaymentID → PaypalTransactionID, PaypalPayerID, PaypalEmail, PaymentIntent, PaymentStatus

PaymentID is both PK and FK; it determines all attributes. **PaypalPayment is in BCNF**.

CardPayment

- FD: PaymentID → CardNumber, CardType, ExpiryDate, BankName, CVV

PaymentID is the PK and sole determinant. **CardPayment is in BCNF**.

ApplePay

- FD: PaymentID → ApplePayTransactionID, ApplePayToken, ApplePayMerchantID, ApplePayEmail, PaymentStatus

PaymentID is the PK and a superkey. **ApplePay is in BCNF**.

PromoPayment

- FD: PaymentID → PromoCode, DiscountValue, ValidatedBy

PaymentID is the PK and sole determinant. **PromoPayment is in BCNF**.

5.2 Final Remarks on Normalization

Because every non-trivial FD in every relation has a determinant that is a superkey, the schema already satisfies BCNF. No further decomposition is necessary. This confirms that the transformation from conceptual to relational and physical design preserved full normalization and eliminated all anomalies related to update, insertion, and deletion.

6. Physical Model

Figure 2 presents the final physical database schema for UrbanPoint, as implemented in Vertabelo. This model includes all normalized relations, primary and foreign key constraints, associative entities, and generalization structures. It reflects the final BCNF-compliant schema described in Sections 3, 4, and 5.

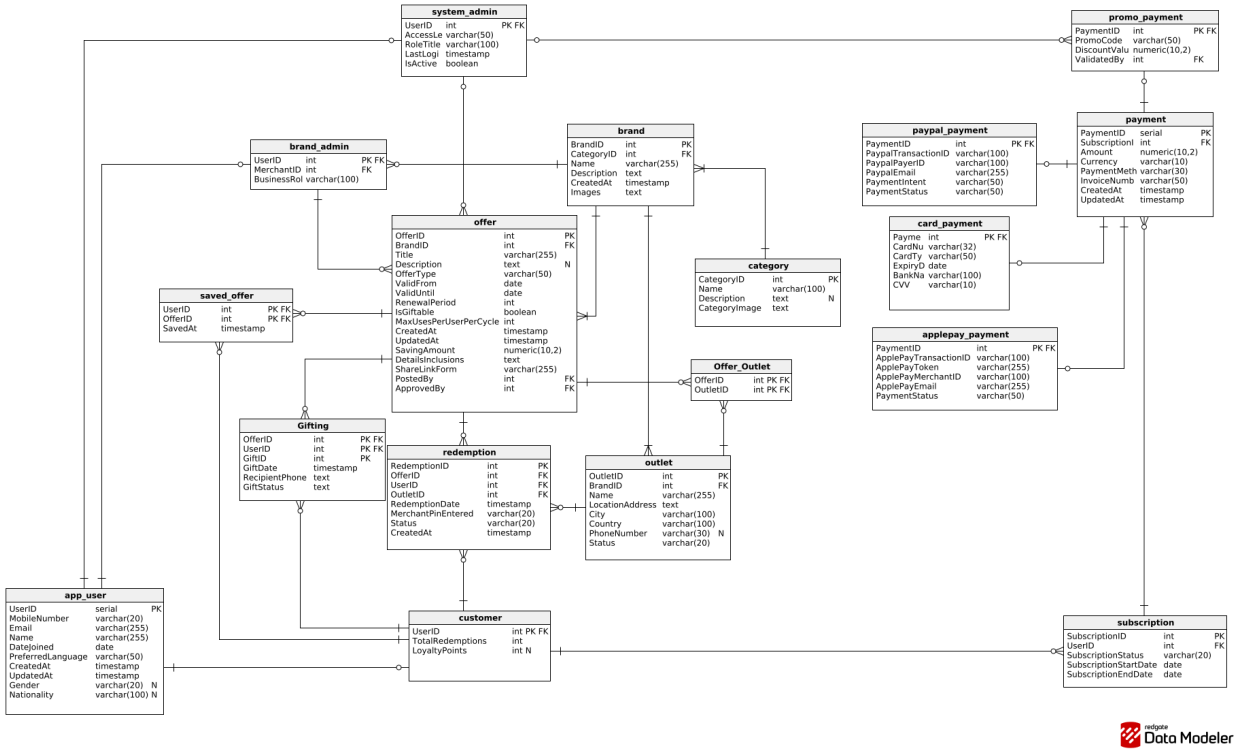

Figure 2: Final physical model for the UrbanPoint database (Vertabelo).

Assumptions

In developing the Phase 2 implementation, we made the following assumptions regarding the physical data model and system behavior:

- **Data Integrity and Uniqueness:** We assume all CSV files contain valid, non-conflicting data that satisfies primary key and foreign key constraints. We further assume that natural keys, such as `MobileNumber` in the `app_user` table and `InvoiceNumber` in the `payment` table, must be unique across the system to prevent duplicate real-world entries.

- **Generalization and Specialization Strategy:** We assumed a Generalization/Specialization strategy for users and payments. We require that every specialized user role (*Customer*, *Brand Admin*, *System Admin*) must first exist as a generalized `app_user`. Similarly, we enforce that all specialized payment methods (PayPal, Card, ApplePay) are subtypes of a generalized `payment` entity to ensure financial data centralization.

- **Manual Key Assignment:** Since the schema does not use `SERIAL` or auto-incrementing fields, we assume all new primary keys are manually computed by the application using a `MAX(id)+1` strategy. We assume no concurrent inserts occur that could generate conflicting IDs during this phase.

- **Identity Through Associations:** We determined that for specific **Associations**, a single ID is insufficient to define uniqueness. Therefore, we assumed that `Offer_Outlet`, `saved_offer`, and `Gifting` must use composite Primary Keys (e.g., `OfferID` + `OutletID`) to physically prevent duplicate associations at the database level.

- **Role-Based Schema Authority:** While the application layer handles session access, we assumed the database must enforce authority via specific Foreign Keys. For instance, we mandate that an `offer` can only be posted by a `brand_admin` and approved by a `system_admin` by linking strictly to those specialized tables rather than the generalized user table.

- **Hierarchical Dependencies:** We assumed that child entities cannot exist without their parents. Specifically, `offer` and `outlet` records are strictly dependent on a valid `BrandID`, and payments are strictly tied to a valid `SubscriptionID`.

- **Transaction and Timestamp Validity:** We assume that a `redemption` is a valid "triangulation" of a User, an Offer, and an Outlet, enforced by three simultaneous Foreign Keys. Furthermore, all timestamps in this table are assumed to reflect the correct local time of the event, enabling valid time-based analytical queries.

- **Offer–Outlet Mapping:** We assume that every outlet associated with a brand participates in at least one offer through the `offer_outlet` association. This ensures that analytical queries (e.g., outlet rankings) can compute valid aggregates without encountering orphaned outlets.

Triggers and Their Relationship to User Stories

To satisfy the project requirement of implementing and demonstrating database triggers, we implemented three triggers designed to enforce critical business rules. These triggers are directly tied to specific user stories and support correct system behavior.

Trigger 1: Prevent Duplicate Mobile Numbers

Trigger Name: `trg_unique_mobile`
User Story Supported: *US1 – Register New Customer (Complex Operational)*

This trigger ensures that no two users can register with the same mobile number. When a new row is inserted into `app_user`, the trigger checks whether the number already exists and raises an exception if so. This enforces the core business rule required by US1 and prevents inconsistent application data.

Trigger 2: Prevent Duplicate Outlet Identifiers

Trigger Name: `trg_no_duplicate_outletid`
User Story Supported: *US5 – Add New Outlet (Simple Operational)*

Since the database does not use auto-incrementing keys, outlet insertion relies on the application generating a new `OutletID`. This trigger ensures that even if a programming error or a race condition occurs, the database will reject any duplicate ID. It provides a safety layer for US5, which demonstrates adding new outlets to the system.

Trigger 3: Prevent Duplicate Outlet Identifiers

Trigger Name: `trg_no_duplicate_outletid`
User Story Supported: *US5 – Add New Outlet (Simple Operational)*

Similarly, outlet insertion relies on the application generating a new `OutletID` via a `MAX(OutletID)+1` strategy. The `trg_no_duplicate_outletid` trigger ensures that any attempt to insert a duplicate `OutletID` is rejected. This protects the integrity of the `outlet` table and ensures that US5, which demonstrates adding new outlets to the system, remains robust even if the manual ID generation were to fail.

Manual Identifier Generation

The system follows a controlled approach to primary key creation.

UserID and CustomerID Assignment

When registering a new customer (US1), the application determines the next `UserID` by querying:

```
SELECT MAX(UserID) FROM app_user
```

The new ID is computed as `MAX + 1` and used for both the insertion into `app_user` and the corresponding row in `customer`. This ensures key consistency and complies with schema constraints.

OutletID Assignment

Similarly, when adding a new outlet (US5), the system executes:

```
SELECT MAX(OutletID) FROM outlet
```

and assigns `MAX + 1` to the new outlet. The trigger `trg_no_duplicate_outletid` adds a database-level safeguard, rejecting inserts where the manually generated identifier conflicts with an existing record.

Rationale

Manual key generation provides full transparency over the identifier space and demonstrates the developer's control over schema constraints. The combination of application-level ID generation and trigger-based enforcement guarantees correctness while respecting the project rule against auto-incrementing columns.