UNIVERSITY OF GONDAR

COLLEGE OF INFORMATICS

DEPARTMENT OF COMPUTER SCIENCE

FUNDAMENTAL DATABASE PROJECT

TITLE: PARKING MANAGEMENT SYSTEM

Student Name	*ID*	*Student Name*	*ID*
Birhanu Fiseha	*GUR/02498/15*	*Biniam Ambachew*	*GUR/01387/16*
Dagmawit Mesfin	*GUR/02065/16*	*Solomon Setegne*	*GUR/01568/16*
Ermiays lakew	*GUR/01774/16*	*Temesgen Dessie*	*GUR/ 01339/16*
Atinaw Dessie	*GUR/22848/16*	*Getachew Mola*	*GUR/01150/16*
Bihoney Gebremeskel	*GUR/01020/16*	*Frewold Hadgu*	*GUR/03222/15*

Submitted to: Mr. Rediet

Submissiom Date: 05/07/2017 E.C

Table of Contents

1 Introduction 1

1.1 Background 1

1.2 Statement of the Problem 1

1.3 Objectives 2

1.3.1 General Objective 2

1.3.2 Specific Objectives 2

2 Methodology 2

2.1 Data Collection Techniques 2

2.2 Database Modeling Approach 2

2.3 Hardware and Software Requirements 2

3 Phase 1: Requirements Analysis and ER Diagram 3

3.1 Entities and Attributes 3

3.2 Entity-Relationship (ER) Model 4

3.2.1 Entities and Their Relationships 4

3.2.2 Entity Relationship (ER) Model Summary 6

3.2.3 Constraints and Business Rules 7

3.2.4 Entity-Relationship Diagram (ER Diagram) 7

3.3 Enhanced ER (EER) Diagram 9

3.3.1 Specialization and Generalization (Inverse Relationship) 9

3.3.2 Aggregation (Whole-Part Relationship) 9

3.4 Mapping ER Diagram to a Relational Schema: Key Rules 11

4 Phase 2: Database Design and Normalization 13

4.1 Define the Relational Schema for the Database 13

4.2 Normalize the Schema to Third Normal Form (3NF) 14

4.2.1 Step 1: First Normal Form (1NF) 15

4.2.2 Step 2: Second Normal Form (2NF) 15

4.2.3 Step 3: Third Normal Form (3NF) 15

4.3 Specify Primary Keys, Foreign Keys, and Constraints 20

5 Phase 3: SQL Implementation 21

5.1 Implement the Database Schema Using SQL (DDL Statements) ... 21

5.2 Populate the Database with Sample Data (DML Statements) ... 24

5.3 3.Write SQL Queries to Retrieve and Manipulate Data (e.g., Nested Queries, Joins, Aggregation) ... 27

5.3.1 Query 1: Retrieve all reservations for a specific user(using joints) ... 27

5.3.2 Query 2: Count the number of reservations per parking slot(using aggregation)... 27

5.3.3 Query 3: Find available parking slots not currently reserved(using nested query) 28

5.3.4 Query 4: Calculate total payment amount per user (using aggregation with join) . 29

5.3.5 Query 5: List vehicles and their entry/exit times for a specific slot (using join with nested query) ... 29

5.3.6 Query 6: Total reservations and payments by user with aggregation and nested query ...30

6 Phase 4: Advanced Features (Bonus) ... 31

6.1 1. Create Views for Frequently Accessed Data ... 31

6.2 Implement Indexes to Optimize Query Performance ... 32

6.3 Write Stored Procedures or Triggers for Automation (Optional, for Advanced Students)..33

6.3.1 -- Stored procedure to book a parking slot ... 33

6.3.2 -- Trigger to update slot status on reservation confirmation ... 35

6.3.3 . Test and Check for Stored Procedure Execution. ... 35

6.3.4 Test and Check for Trigger Execution. ... 37

Conclusion ...39

References ...40

1 Introduction

1.1 Background

Parking management is a crucial aspect of urban infrastructure, ensuring the efficient allocation of parking spaces, reducing congestion, and improving user experience. Traditional parking management systems often rely on manual processes, which can lead to inefficiencies, errors, and inconvenience for users. In many cases, parking lots operate on a first-come, first-served basis without real-time monitoring, leading to wasted time and frustration for drivers searching for available spaces.

Manual parking systems pose several challenges, including:

- **Inefficiency**: Without automation, tracking available slots and managing reservations is cumbersome.
- **Security Risks**: Unauthorized access and parking violations are difficult to monitor.
- **Lack of Data-Driven Insights**: Parking lot owners lack analytics on peak hours, usage trends, and revenue insights.
- **Limited Payment Flexibility**: Many systems rely on cash transactions, making payment tracking and financial management difficult.

To address these challenges, this project aims to design and implement a **Parking Management System (PMS)** using a relational database. The system will integrate real-time slot availability tracking, online reservations, automated entry/exit logs, and digital payments. The database-driven approach will improve efficiency, security, and customer convenience while providing valuable insights for parking lot operators.

1.2 Statement of the Problem

The inefficiencies of manual and outdated parking systems result in operational bottlenecks, customer dissatisfaction, and revenue loss. The following issues highlight the need for an automated system:

1. **Operational Inefficiencies**:
    - Manual tracking leads to errors in slot availability and reservations.
    - Long queues due to slow manual entry and exit processing.
2. **Security and Accessibility Issues**:
    - Unauthorized parking is difficult to track.
    - No proper log of vehicle entries and exits.
3. **Limited Payment and Booking Options**:
    - Users have to pay in cash, with no digital payment integration.
    - No online reservation system, leading to uncertainty about available spaces.

4. **Lack of Data Insights**:
   - No way to analyze peak hours, revenue trends, or user behavior.
   - Inability to optimize pricing or expand based on data-driven decisions.

This project seeks to solve these challenges by developing a modern Parking Management System that ensures **real-time slot tracking, online reservations, secure vehicle logging, and digital payments**.

1.3 Objectives

1.3.1 General Objective

To design, develop, and implement a **Parking Management System (PMS)** that enhances efficiency, security, and user experience while providing valuable data insights for effective parking space management.

1.3.2 Specific Objectives

1. **To create a centralized database** for storing user, vehicle, parking slot, reservation, and transaction data.
2. **To implement an online reservation system** allowing users to pre-book parking slots.
3. **To automate entry and exit tracking** for better security and real-time slot availability updates.
4. **To integrate digital payments** for convenient and secure transactions.
5. **To generate reports and analytics** for parking operators to optimize pricing and space utilization.

2 Methodology

2.1 Data Collection Techniques

- Conducting surveys with drivers and parking lot managers to gather requirements.
- Analyzing existing parking management solutions to identify best practices.

2.2 Database Modeling Approach

- Using **Entity-Relationship (ER) modeling** to design the database, ensuring a well-structured schema.
- **Normalization (up to 3NF or BCNF)** to eliminate redundancy and ensure data integrity.
- Implementing **referential integrity** using primary and foreign keys.

2.3 Hardware and Software Requirements

- **Hardware**: Server with storage and processing capabilities for database operations and concurrent user access.

- **Software**:
    - **DBMS**: MySQL or PostgreSQL for database management.
    - **Web Technologies**: HTML, CSS, JavaScript for UI.
    - **Backend**: Python (Django/Flask) or PHP for API and database interactions.

3 Phase 1: Requirements Analysis and ER Diagram

For the **Parking Management System**, the following entities and relationships have been identified:

3.1 Entities and Attributes

1. **Users (Stores user details)**
    - user_id (Primary Key)
    - full_name
    - email
    - phone_number
    - role (Admin, Employee, Customer)
    - password_hash
2. **Parking Slots (Represents individual parking spaces)**
    - slot_id (Primary Key)
    - slot_number
    - location (Area/Floor/GPS)
    - status (Available, Occupied, Reserved)
    - price_per_hour
3. **Vehicles (Tracks vehicle information)**
    - vehicle_id (Primary Key)
    - user_id (Foreign Key → Users)
    - license_plate
    - vehicle_type (Car, Bike, Truck, etc.)
    - brand
    - model
    - color
4. **Parking Reservations (Handles parking bookings)**
    - reservation_id (Primary Key)
    - user_id (Foreign Key → Users)
    - slot_id (Foreign Key → Parking Slots)

- vehicle_id (Foreign Key → Vehicles)
- start_time
- end_time
- status (Pending, Confirmed, Completed, Cancelled)

5. **Payments (Stores transaction details)**
    - payment_id (Primary Key)
    - user_id (Foreign Key → Users)
    - reservation_id (Foreign Key → Parking Reservations)
    - amount_paid
    - payment_method (Cash, Credit Card, Mobile Payment)
    - payment_date
    - payment_status (Pending, Paid, Failed)

6. **Entry & Exit Logs (Records vehicle movements)**
    - log_id (Primary Key)
    - vehicle_id (Foreign Key → Vehicles)
    - slot_id (Foreign Key → Parking Slots)
    - entry_time
    - exit_time

3.2 Entity-Relationship (ER) Model

To design a well-structured **ER model**, we need to define **entity relationships, cardinalities, and constraints** between different entities. Below is a **detailed explanation of each relationship**, including how they interact within the **Parking Management System**.

3.2.1 Entities and Their Relationships

1. Users (user_id) ⇄ Vehicles (vehicle_id)

Relationship: One-to-Many (1:M)

- A **User** can own **multiple Vehicles**.
- A **Vehicle** belongs to only **one User**.

Foreign Key Constraint:

- Vehicles.user_id references Users.user_id.

2. Users (user_id) ⇄ Parking Reservations (reservation_id)

Relationship: One-to-Many (1:M)

- A **User** can make **multiple Reservations**.

- A **Reservation** belongs to **one User**.

Foreign Key Constraint:

- Parking_Reservations.user_id references Users.user_id.

3. Vehicles (vehicle_id) ⇄ Parking Reservations (reservation_id)

Relationship: One-to-One (1:1) or One-to-Many (1:M)

- A **Vehicle** is linked to **one Reservation** at a time (1:1).
- A **User** may book multiple Reservations for the same or different Vehicles (1:M).

Foreign Key Constraint:

- Parking_Reservations.vehicle_id references Vehicles.vehicle_id.

4. Parking Slots (slot_id) ⇄ Parking Reservations (reservation_id)

Relationship: One-to-Many (1:M)

- A **Parking Slot** can have **multiple Reservations** over time.
- A **Reservation** is associated with **one Parking Slot** at a given time.

Foreign Key Constraint:

- Parking_Reservations.slot_id references Parking_Slots.slot_id.

5. Parking Reservations (reservation_id) ⇄ Payments (payment_id)

Relationship: One-to-One (1:1)

- A **Reservation** has **one Payment record**.
- A **Payment** is linked to **only one Reservation**.

Foreign Key Constraint:

- Payments.reservation_id references Parking_Reservations.reservation_id.

6. Users (user_id) ⇄ Payments (payment_id)

Relationship: One-to-Many (1:M)

- A **User** can make **multiple Payments** for different Reservations.
- A **Payment** is associated with only **one User**.

Foreign Key Constraint:

- Payments.user_id references Users.user_id.

7. Vehicles (vehicle_id) ⇄ Entry & Exit Logs (log_id)

Relationship: One-to-Many (1:M)

- A **Vehicle** can have **multiple Entry and Exit Logs** over time.
- Each **Log entry** is associated with **one Vehicle**.

Foreign Key Constraint:

- Entry_Exit_Logs.vehicle_id references Vehicles.vehicle_id.

8. Parking Slots (slot_id) ⇄ Entry & Exit Logs (log_id)

Relationship: One-to-Many (1:M)

- A **Parking Slot** can have **multiple vehicle entries and exits** recorded over time.
- Each **Log entry** is associated with **one Parking Slot**.

Foreign Key Constraint:

- Entry_Exit_Logs.slot_id references Parking_Slots.slot_id.

3.2.2 Entity Relationship (ER) Model Summary

Entity A	Entity B	Relationship	Cardinality
Users	Vehicles	Owns	1:M
Users	Reservations	Makes	1:M
Vehicles	Reservations	Used in	1:1 or 1:M
Parking Slots	Reservations	Assigned to	1:M
Reservations	Payments	Paid for	1:1
Users	Payments	Makes	1:M
Vehicles	Entry Logs	Enters/Exits	1:M
Parking Slots	Entry Logs	Logs entries/exits	1:M

3.2.3 Constraints and Business Rules

1. A parking slot can only be reserved if its status is "Available".
2. A user cannot reserve more than one slot at the same time for the same vehicle.
3. A reservation cannot be "Confirmed" unless payment is successful.
4. An entry log is created when a vehicle enters a slot, and an exit log is recorded when it leaves.
5. A user cannot book a parking slot for more than 24 hours unless explicitly allowed by the system.

3.2.4 Entity-Relationship Diagram (ER Diagram)

This ER diagram represents the relationships between different entities in the Parking Management System. It includes:

1. Users – Users register and manage their vehicles.

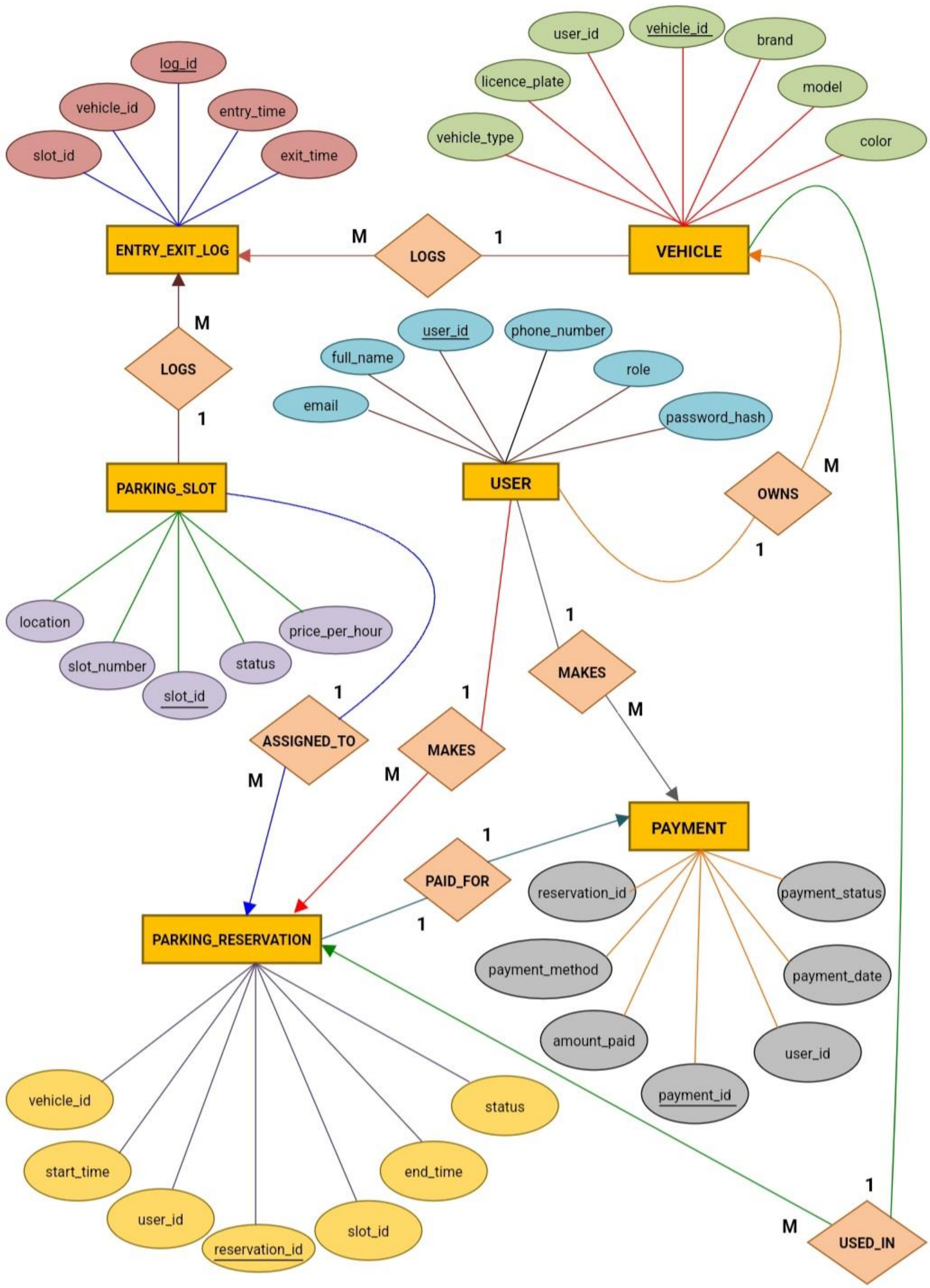
2. Vehicles – Each vehicle is linked to a user and can enter/exit the parking lot.

3. Parking Slots – Represent individual parking spaces with availability status.

4. Parking Reservations – Users can reserve slots for their vehicles.

5. Payments – Handles transactions for reservations.

6. Entry & Exit Logs – Tracks vehicle movements.



log_id
vehicle_id
entry_time
slot_id
exit_time
ENTRY_EXIT_LOG
M
LOGS
1
VEHICLE
user_id
vehicle_id
brand
licence_plate
model
vehicle_type
color
M
LOGS
1
PARKING_SLOT
user_id
phone_number
full_name
role
email
password_hash
USER
OWNS
M
1
location
slot_number
slot_id
status
price_per_hour
1
MAKES
M
1
ASSIGNED_TO
M
1
MAKES
M
PAYMENT
1
PAID_FOR
1
PARKING_RESERVATION
reservation_id
payment_status
payment_method
payment_date
amount_paid
user_id
payment_id
vehicle_id
status
start_time
end_time
user_id
reservation_id
slot_id
1
M
USED_IN

3.3 Enhanced ER (EER) Diagram

Enhancements in the EER Diagram:

3.3.1 Specialization and Generalization (Inverse Relationship)

The User entity is specialized into three subclasses:

- Admin
- Employee
- Customer

This specialization represents an ISA relationship, meaning each user falls into one of these roles.

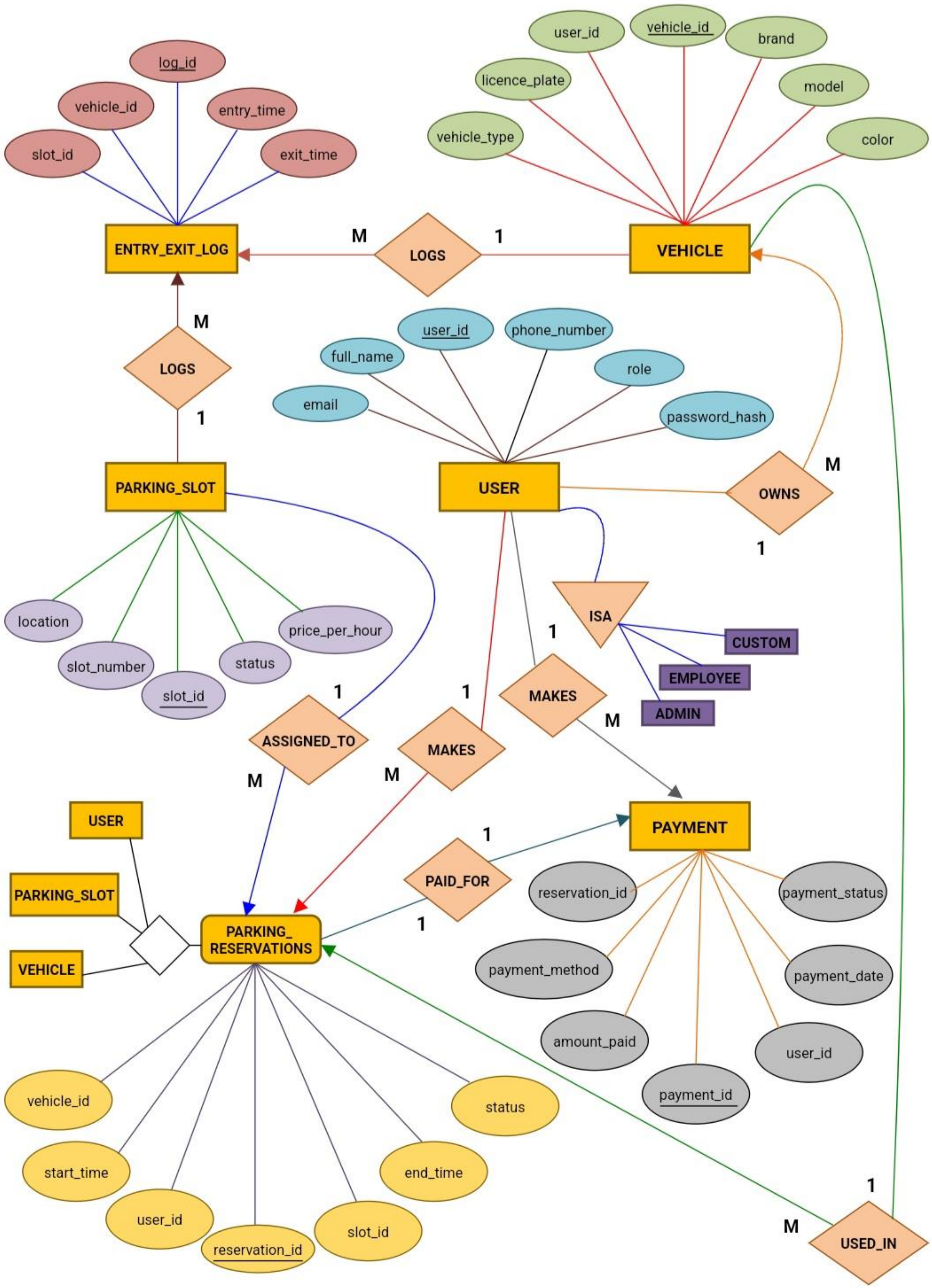
Specialization and Generalization are inverse operations. While specialization breaks down a general entity (like User) into more specific subclasses (Admin, Employee, Customer), generalization would take these specific subclasses and combine them into a more general entity.

Notation: The inverted triangle (▽) with the “ISA” label is used for both specialization and generalization. The diagram shows this inverse relationship, demonstrating that the User entity can be broken down (specialized) into roles and, if needed, could be generalized back into a more abstract form.

3.3.2 Aggregation (Whole-Part Relationship)

The Parking Reservation entity represents an aggregation of multiple entities:

- User
- Vehicle
- Parking Slot



log_id
vehicle_id
entry_time
slot_id
exit_time
ENTRY_EXIT_LOG
M
LOGS
1
user_id
vehicle_id
brand
licence_plate
model
vehicle_type
color
VEHICLE
M
LOGS
1
PARKING_SLOT
user_id
full_name
phone_number
role
email
password_hash
USER
OWNS
M
1
location
slot_number
slot_id
status
price_per_hour
ISA
CUSTOM
EMPLOYEE
ADMIN
1
ASSIGNED_TO
M
1
MAKES
M
1
MAKES
M
PAYMENT
1
PAID_FOR
1
USER
PARKING_SLOT
VEHICLE
PARKING_
RESERVATIONS
reservation_id
payment_status
payment_method
payment_date
amount_paid
user_id
payment_id
vehicle_id
start_time
user_id
reservation_id
slot_id
end_time
status
1
M
USED_IN

3.4 Mapping ER Diagram to a Relational Schema: Key Rules

When converting an Enhanced ER (EER) Diagram into a Relational Database Schema, follow these rules to ensure proper mapping.

1. Mapping Entities to Tables

- Each entity in the ER diagram becomes a table in the relational schema.

Example: The USER entity is mapped to the USERS table.

2. Mapping Attributes to Columns

- Each attribute of an entity becomes a column in the table.
- Simple attributes are directly mapped (e.g., full_name, email).
- Composite attributes should be broken into separate columns.

Multivalued attributes require a separate table with a foreign key relationship.

3. Mapping Primary Keys (PKs)

- Each entity must have a primary key that uniquely identifies records in the table.

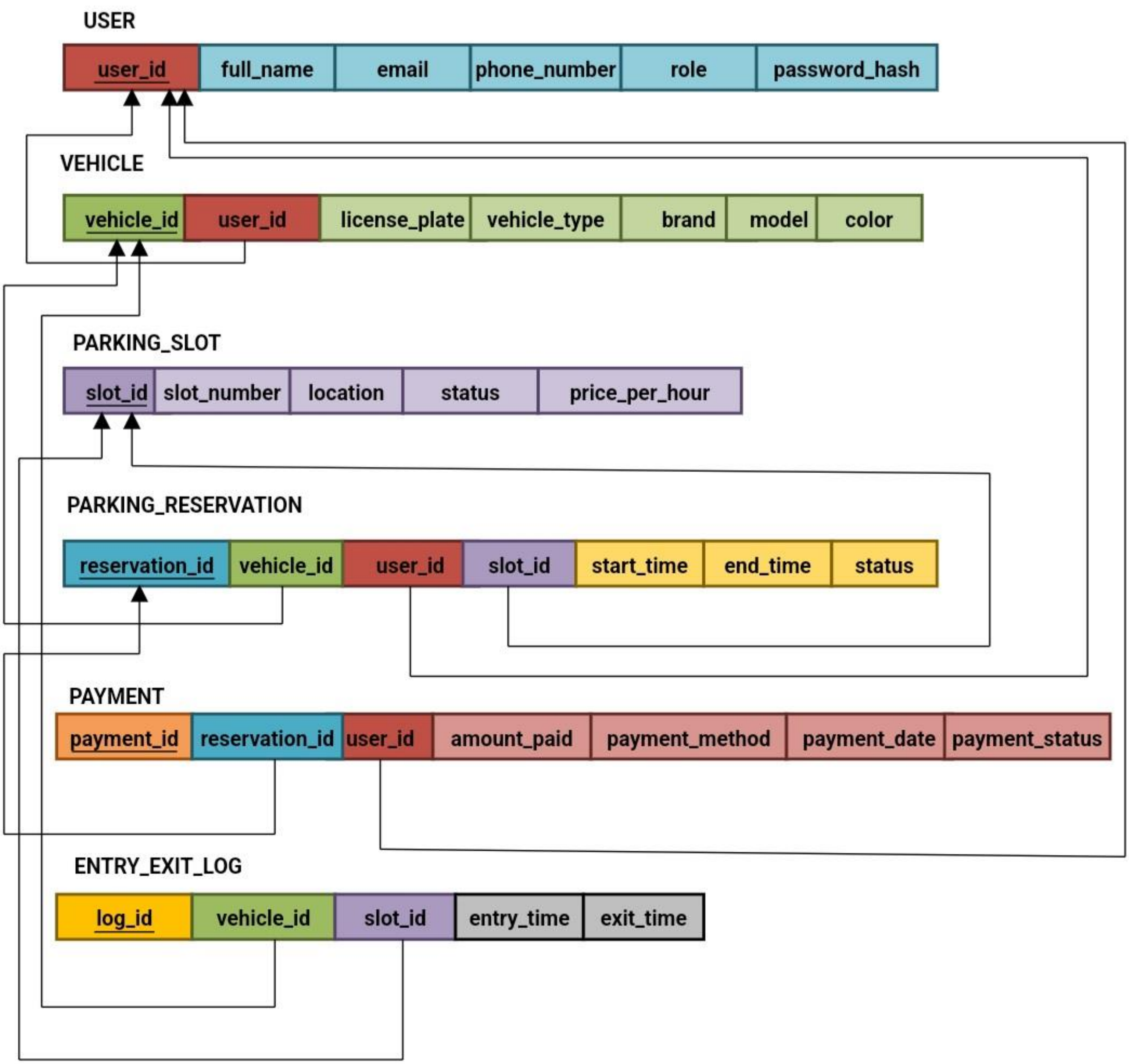
Example: user_id is the PK for the USERS table.

4. Mapping Relationships to Foreign Keys (FKs)

- Relationships between entities are mapped using foreign keys (FKs).
- The foreign key (FK) is placed in the table representing the "many" side of a one-to-many relationship.
- The foreign key references the primary key (PK) of the related table.

Example: The PARKING_RESERVATIONS table has user_id (FK), slot_id (FK), and vehicle_id (FK), referencing their respective PKs.

Example: The VEHICLES table has user_id (FK), linking each vehicle to an owner in the USER table.


USER
user_id
full_name
email
phone_number
role
password_hash
VEHICLE
vehicle_id
user_id
license_plate
vehicle_type
brand
model
color
PARKING_SLOT
slot_id
slot_number
location
status
price_per_hour
PARKING_RESERVATION
reservation_id
vehicle_id
user_id
slot_id
start_time
end_time
status
PAYMENT
payment_id
reservation_id
user_id
amount_paid
payment_method
payment_date
payment_status
ENTRY_EXIT_LOG
log_id
vehicle_id
slot_id
entry_time
exit_time

4 Phase 2: Database Design and Normalization

This phase focuses on designing the relational schema for the Parking Management System, normalizing it to Third Normal Form (3NF), and specifying primary keys, foreign keys, and constraints to ensure entity integrity and referential integrity. The design is based on a conceptual understanding of parking management needs, including users, vehicles, parking slots, reservations, payments, and entry/exit logs, with relationships modeled to support efficient data storage and retrieval.

4.1 Define the Relational Schema for the Database

The relational schema for the Parking Management System includes the following tables, reflecting the key entities, attributes, and relationships necessary for managing parking operations:

- Users Table

  Attributes: user_id (Primary Key), full_name, email, phone_number, role, password_hash
  Description: Stores user details, including their role (e.g., Admin, Employee, Customer) and authentication information, to manage individuals interacting with the system.

- Vehicles Table

  Attributes: vehicle_id (Primary Key), user_id (Foreign Key referencing Users), license_plate, vehicle_type, brand, model, color
  Description: Tracks vehicle information associated with users, including vehicle identification and type, to facilitate parking reservations.

- Parking_Slots Table

  Attributes: slot_id (Primary Key), slot_number, location, status, price_per_hour
  Description: Represents individual parking spaces, including their location, availability status, and hourly pricing, to manage space allocation.

- Reservations Table

  Attributes: reservation_id (Primary Key), user_id (Foreign Key referencing Users), vehicle_id (Foreign Key referencing Vehicles), slot_id (Foreign Key referencing Parking_Slots), start_time, end_time, status
  Description: Handles parking bookings, linking users, vehicles, and parking slots with time-based reservations and status tracking.

- Payments Table

Attributes: payment_id (Primary Key), reservation_id (Foreign Key referencing Reservations), user_id (Foreign Key referencing Users), amount_paid, payment_date, payment_status
Description: Stores transaction details for parking reservations, including payment amounts, dates, and status.

- Payment_Methods Table

Attributes: method_id (Primary Key), method_name
Description: Lists available payment methods (e.g., Cash, Credit Card, Mobile Payment) for transactions.

- Payment_Transactions Table

Attributes: payment_id (Foreign Key referencing Payments), method_id (Foreign Key referencing Payment_Methods), PRIMARY KEY (payment_id, method_id)
Description: Links payments to specific payment methods, handling potential many-to-many relationships.

- Entry_Exit_Logs Table

Attributes: log_id (Primary Key), vehicle_id (Foreign Key referencing Vehicles), slot_id (Foreign Key referencing Parking_Slots), entry_time, exit_time
Description: Records vehicle movements, tracking entry and exit times for parking slots to ensure security and monitoring.

This schema captures the entities and relationships needed for a parking management system, including one-to-many relationships (e.g., a user can own multiple vehicles, a parking slot can have multiple reservations over time) and one-to-one relationships (e.g., a reservation links to one payment).

4.2 Normalize the Schema to Third Normal Form (3NF)

To ensure data integrity and eliminate redundancy, the schema is normalized through the following steps:

4.2.1 Step 1: First Normal Form (1NF)

Objective: Ensure all attributes are atomic, each table has a primary key, and there are no repeating groups.

- Process: Review each table to confirm atomic values and unique identifiers.

  - All attributes (e.g., full_name, email, license_plate) are single, indivisible values.
  - Each table has a primary key (e.g., user_id for Users, vehicle_id for Vehicles).
  - No repeating groups exist (e.g., Vehicles lists one vehicle per row).

Result: The schema is in 1NF, requiring no changes at this step.

4.2.2 Step 2: Second Normal Form (2NF)

- Objective: Ensure all non-key attributes are fully functionally dependent on the entire primary key (remove partial dependencies).
- Process: Check for composite primary keys and partial dependencies.
  - No table has a composite primary key; each uses a single primary key (e.g., user_id, vehicle_id).
  - All non-key attributes depend on their respective primary keys (e.g., full_name depends on user_id in Users, license_plate on vehicle_id in Vehicles).
- Result: The schema is in 2NF, as there are no partial dependencies.

4.2.3 Step 3: Third Normal Form (3NF)

- Objective: Ensure no transitive dependencies exist (non-key attributes do not depend on other non-key attributes).
- Process: Analyze each table for transitive dependencies:
- Users Table:
  - Attributes: user_id (PK), full_name, email, phone_number, role, password_hash
  - Dependencies: user_id → full_name, email, phone_number, role, password_hash
  - Check: role might imply additional attributes (e.g., permissions), but as a simple attribute (e.g., 'Admin', 'Customer'), it depends directly on user_id. No transitive dependencies exist, so Users is in 3NF.
  - The resulting 3NF schema for the Users table is presented below:

	user_id [PK] integer	full_name character varying (100)	email character varying (100)	phone_number character varying (20)	role character varying (50)	password_hash character varying (255)
1	1	Birhanu Fiseha	birhanu.fiseha@email.com	+251911234567	Customer	hashed_password1
2	2	Biniam Ambachew	biniam.ambachew@email.com	+251922345678	Customer	hashed_password2
3	3	Dagmawit Mesfin	dagmawit.mesfin@email.com	+251933456789	Customer	hashed_password3
4	4	Solomon Setegne	solomon.setegne@email.com	+251944567890	Customer	hashed_password4
5	5	Ermias Lakew	ermias.lakew@email.com	+251955678901	Customer	hashed_password5
6	6	Temesgen Dessie	temesgen.dessie@email.com	+251966789012	Customer	hashed_password6
7	7	Atinaw Dessie	atinaw.dessie@email.com	+251977890123	Customer	hashed_password7
8	8	Getachew Mola	getachew.mola@email.com	+251988901234	Customer	hashed_password8
9	9	Bihoney Gebremeskel	bihoney.gebremeskel@email.com	+251999012345	Customer	hashed_password9
10	10	Frewold Hadgu	frewold.hadgu@email.com	+251900123456	Customer	hashed_password10

- Vehicles Table:
  - Attributes: vehicle_id (PK), user_id (FK), license_plate, vehicle_type, brand, model, color
  - Dependencies: vehicle_id → user_id, license_plate, vehicle_type, brand, model, color
  - No transitive dependencies (e.g., user_id doesn't determine license_plate indirectly). Vehicles is in 3NF.
  - The resulting 3NF schema for the Vehicles table is presented below

	vehicle_id [PK] integer	user_id integer	license_plate character varying (20)	vehicle_type character varying (50)	brand character varying (50)	model character varying (50)	color character varying (50)
1	1	1	ABC123	Car	Toyota	Camry	Blue
2	2	2	XYZ789	Car	Honda	Civic	Red
3	3	3	DEF456	Car	Ford	Focus	Black
4	4	4	GHI789	Truck	Isuzu	D-Max	White
5	5	5	JKL012	Car	BMW	3 Series	Silver
6	6	6	MNO345	Car	Mercedes	C-Class	Grey
7	7	7	PQR678	Bike	Honda	CBR	Yellow
8	8	8	STU901	Car	Audi	A4	Green
9	9	9	VWX234	Car	Volkswagen	Golf	Purple
10	10	10	YZA567	Car	Hyundai	Tucson	Orange

- Parking_Slots Table:
  - Attributes: slot_id (PK), slot_number, location, status, price_per_hour
  - Dependencies: slot_id → slot_number, location, status, price_per_hour
  - No transitive dependencies (e.g., location doesn't determine price_per_hour indirectly). Parking_Slots is in 3NF.
  - The resulting 3NF schema for the Parking_Slots table is presented below:

	slot_id [PK] integer	slot_number character varying (20)	location character varying (100)	status character varying (20)	price_per_hour numeric (10,2)
1	1	P001	Level 1	Available	5.00
2	2	P002	Level 1	Available	5.00
3	3	P003	Level 2	Available	5.50
4	4	P004	Level 2	Available	5.50
5	5	P005	Level 3	Available	6.00

- Reservations Table:
  - Attributes: reservation_id (PK), user_id (FK), vehicle_id (FK), slot_id (FK), start_time, end_time, status
  - Dependencies: reservation_id → user_id, vehicle_id, slot_id, start_time, end_time, status
  - No transitive dependencies (e.g., user_id doesn't determine start_time indirectly). Reservations is in 3NF.
  - The resulting 3NF schema for the Reservations table is presented below:

	reservation_id [PK] integer	user_id integer	vehicle_id integer	slot_id integer	start_time timestamp without time zone	end_time timestamp without time zone	status character varying (20)
1	1	1	1	1	2025-02-25 08:00:00	2025-02-25 10:00:00	Confirmed
2	2	2	2	1	2025-02-25 10:30:00	2025-02-25 12:30:00	Confirmed
3	3	3	3	2	2025-02-25 09:00:00	2025-02-25 11:00:00	Confirmed
4	4	4	4	2	2025-02-25 11:30:00	2025-02-25 13:30:00	Confirmed
5	5	5	5	3	2025-02-25 12:00:00	2025-02-25 14:00:00	Confirmed
6	6	6	6	3	2025-02-25 14:30:00	2025-02-25 16:30:00	Confirmed
7	7	7	7	4	2025-02-25 13:00:00	2025-02-25 15:00:00	Confirmed
8	8	8	8	4	2025-02-25 15:30:00	2025-02-25 17:30:00	Confirmed
9	9	9	9	5	2025-02-25 16:00:00	2025-02-25 18:00:00	Confirmed
10	10	10	10	5	2025-02-25 18:30:00	2025-02-25 20:30:00	Confirmed

- Payments Table:
  - Attributes: payment_id (PK), reservation_id (FK), user_id (FK), amount_paid, payment_date, payment_status
  - Dependencies: payment_id → reservation_id, user_id, amount_paid, payment_date, payment_status
  - Issue: The ER diagram and initial schema included payment_method, which could create a transitive dependency if it determines other attributes (e.g., transaction fees). To achieve 3NF, normalize payment_method into separate tables (Payment_Methods and Payment_Transactions), as shown in the schema.
  - The resulting 3NF schema for the Payments table is presented below:

	payment_id [PK] integer	reservation_id integer	user_id integer	amount_paid numeric (10,2)	payment_date timestamp without time zone	payment_status character varying (20)
1	1	1	1	10.00	2025-02-25 08:15:00	Paid
2	2	2	2	10.00	2025-02-25 10:45:00	Paid
3	3	3	3	10.00	2025-02-25 09:15:00	Paid
4	4	4	4	10.00	2025-02-25 11:45:00	Paid
5	5	5	5	11.00	2025-02-25 12:15:00	Paid
6	6	6	6	11.00	2025-02-25 14:45:00	Paid
7	7	7	7	11.00	2025-02-25 13:15:00	Paid
8	8	8	8	11.00	2025-02-25 15:45:00	Paid
9	9	9	9	12.00	2025-02-25 16:15:00	Paid
10	10	10	10	12.00	2025-02-25 18:45:00	Paid

- Payment_Methods Table:
  - Attributes: method_id (PK), method_name
  - Dependencies: method_id → method_name
  - No transitive dependencies. Payment_Methods is in 3NF.
  - The resulting 3NF schema for the Payment_Methods table is presented below::

	method_id [PK] integer	method_name character varying (50)
1	1	Credit Card
2	2	Mobile Payment
3	3	Cash

- Payment_Transactions Table:
  - Attributes: payment_id (FK), method_id (FK)
  - Dependencies: (payment_id, method_id) → none (PK only references)
  - No transitive dependencies. Payment_Transactions is in 3NF.
  - The resulting 3NF schema for the Payment_Transactions table is presented below:

	payment_id [PK] integer	method_id [PK] integer
1	1	1
2	2	2
3	3	3
4	4	1
5	5	2
6	6	3
7	7	1
8	8	2
9	9	3
10	10	1

- Entry_Exit_Logs Table:
  - Attributes: log_id (PK), vehicle_id (FK), slot_id (FK), entry_time, exit_time
  - Dependencies: log_id → vehicle_id, slot_id, entry_time, exit_time
  - No transitive dependencies (e.g., vehicle_id doesn't determine entry_time indirectly). Entry_Exit_Logs is in 3NF.
  - The resulting 3NF schema for the Entry_Exit_Logs table is presented below:

	log_id [PK] integer	vehicle_id integer	slot_id integer	entry_time timestamp without time zone	exit_time timestamp without time zone
1	1	1	1	2025-02-25 08:05:00	2025-02-25 10:05:00
2	2	2	1	2025-02-25 10:35:00	2025-02-25 12:35:00
3	3	3	2	2025-02-25 09:05:00	2025-02-25 11:05:00
4	4	4	2	2025-02-25 11:35:00	2025-02-25 13:35:00
5	5	5	3	2025-02-25 12:05:00	2025-02-25 14:05:00
6	6	6	3	2025-02-25 14:35:00	2025-02-25 16:35:00
7	7	7	4	2025-02-25 13:05:00	2025-02-25 15:05:00
8	8	8	4	2025-02-25 15:35:00	2025-02-25 17:35:00
9	9	9	5	2025-02-25 16:05:00	2025-02-25 18:05:00
10	10	10	5	2025-02-25 18:35:00	[null]

- Result: The schema, after normalizing payment_method, is in 3NF, with no redundancy or transitive dependencies.

4.3 Specify Primary Keys, Foreign Keys, and Constraints

Based on the 3NF schema, the following keys and constraints ensure data integrity:

- Primary Keys
  - Users: user_id
  - Vehicles: vehicle_id
  - Parking_Slots: slot_id
  - Reservations: reservation_id
  - Payments: payment_id
  - Payment_Methods: method_id
  - Payment_Transactions: payment_id, method_id (composite primary key)
  - Entry_Exit_Logs: log_id
- Foreign Keys
  - Vehicles: user_id (references Users.user_id)
  - Reservations: user_id (references Users.user_id), vehicle_id (references Vehicles.vehicle_id), slot_id (references Parking_Slots.slot_id)
  - Payments: reservation_id (references Reservations.reservation_id), user_id (references Users.user_id)
  - Payment_Transactions: payment_id (references Payments.payment_id), method_id (references Payment_Methods.method_id)
  - Entry_Exit_Logs: vehicle_id (references Vehicles.vehicle_id), slot_id (references Parking_Slots.slot_id)
- Constraints
  - Entity Integrity: Ensure each table has a unique primary key (e.g., user_id must be unique in Users).
  - Referential Integrity: Ensure foreign keys reference valid primary keys (e.g., user_id in Vehicles must exist in Users).
  - Additional Constraints (based on business rules for a parking system):
    - A parking slot can only be reserved if its status is “Available” (enforced via Status in Parking_Slots, e.g., CHECK (status IN (‘Available’, ‘Occupied’, ‘Reserved’))).
    - A user cannot reserve more than one slot at the same time for the same vehicle (enforced via checks in Reservations on start_time, end_time, vehicle_id, and slot_id, potentially with a trigger or stored procedure).
    - A reservation cannot be “Confirmed” unless payment is successful (enforced via Status in Reservations and Payments.payment_status, e.g., CHECK (status IN (‘Pending’, ‘Confirmed’, ‘Completed’, ‘Cancelled’)) and business logic linking Payments.payment_status = ‘Paid’).

- An entry log is created when a vehicle enters a slot, and an exit log is recorded when it leaves (enforced via entry_time and exit_time in Entry_Exit_Logs, potentially with triggers).
- A user cannot book a parking slot for more than 24 hours unless explicitly allowed (enforced via start_time and end_time in Reservations, e.g., CHECK (end_time – start_time <= INTERVAL '24 hours')).

This 3NF schema ensures a robust, efficient, and scalable database design for the Parking Management System, supporting efficient data storage, retrieval, and management.

5 Phase 3: SQL Implementation

5.1 Implement the Database Schema Using SQL (DDL Statements)

Note: This section presents the SQL Data Definition Language (DDL) statements to create the database and tables for the Parking Management System, based on the 3NF schema from Phase 2. The statements define the structure, primary keys, foreign keys, and constraints (e.g., CHECK, UNIQUE) for the users, vehicles, parking_slots, reservations, payments, payment_methods, payment_transactions, and entry_exit_logs tables, ensuring data integrity and normalization to Third Normal Form (3NF). These statements are formatted for execution in PostgreSQL and documented here for clarity.

```
-- Create the database
CREATE DATABASE parking_management_system_db;

-- Connect to the database
\c parking_management_system_db

-- Create Users table
CREATE TABLE users (
   user_id SERIAL PRIMARY KEY,
   full_name VARCHAR(100) NOT NULL,
   email VARCHAR(100) UNIQUE NOT NULL,
   phone_number VARCHAR(20) NOT NULL,
   role VARCHAR(50) NOT NULL CHECK (role IN ('Customer')),
   password_hash VARCHAR(255) NOT NULL
);

-- Create Vehicles table
CREATE TABLE vehicles (
   vehicle_id SERIAL PRIMARY KEY,
```

```sql
    user_id INT,
    license_plate VARCHAR(20) UNIQUE NOT NULL,
    vehicle_type VARCHAR(50) NOT NULL,
    brand VARCHAR(50),
    model VARCHAR(50),
    color VARCHAR(50),
    FOREIGN KEY (user_id) REFERENCES users(user_id) ON DELETE SET NULL
);

-- Create Parking_Slots table
CREATE TABLE parking_slots (
    slot_id SERIAL PRIMARY KEY,
    slot_number VARCHAR(20) UNIQUE NOT NULL,
    location VARCHAR(100) NOT NULL,
    status VARCHAR(20) NOT NULL CHECK (status IN ('Available', 'Occupied', 'Reserved')),
    price_per_hour DECIMAL(10, 2) NOT NULL CHECK (price_per_hour >= 0)
);

-- Create Reservations table
CREATE TABLE reservations (
    reservation_id SERIAL PRIMARY KEY,
    user_id INT,
    vehicle_id INT,
    slot_id INT,
    start_time TIMESTAMP NOT NULL,
    end_time TIMESTAMP NOT NULL,
    status VARCHAR(20) NOT NULL CHECK (status IN ('Pending', 'Confirmed', 'Completed',
'Cancelled')),
    CHECK (end_time > start_time),
    CHECK (end_time - start_time <= INTERVAL '24 hours'),
    FOREIGN KEY (user_id) REFERENCES users(user_id) ON DELETE SET NULL,
    FOREIGN KEY (vehicle_id) REFERENCES vehicles(vehicle_id) ON DELETE SET NULL,
    FOREIGN KEY (slot_id) REFERENCES parking_slots(slot_id) ON DELETE SET NULL
);

-- Create Payments table
CREATE TABLE payments (
    payment_id SERIAL PRIMARY KEY,
    reservation_id INT,
    user_id INT,
```

```sql
    amount_paid DECIMAL(10, 2) NOT NULL CHECK (amount_paid > 0),
    payment_date TIMESTAMP NOT NULL,
    payment_status VARCHAR(20) NOT NULL CHECK (payment_status IN ('Pending', 'Paid',
'Failed')),
    FOREIGN KEY (reservation_id) REFERENCES reservations(reservation_id) ON DELETE
CASCADE,
    FOREIGN KEY (user_id) REFERENCES users(user_id) ON DELETE SET NULL
);

-- Create Payment_Methods table
CREATE TABLE payment_methods (
    method_id SERIAL PRIMARY KEY,
    method_name VARCHAR(50) UNIQUE NOT NULL CHECK (method_name IN ('Credit
Card', 'Mobile Payment', 'Cash'))
);

-- Create Payment_Transactions table
CREATE TABLE payment_transactions (
    payment_id INT,
    method_id INT,
    PRIMARY KEY (payment_id, method_id),
    FOREIGN KEY (payment_id) REFERENCES payments(payment_id) ON DELETE
CASCADE,
    FOREIGN KEY (method_id) REFERENCES payment_methods(method_id) ON DELETE
SET NULL
);

-- Create Entry_Exit_Logs table
CREATE TABLE entry_exit_logs (
    log_id SERIAL PRIMARY KEY,
    vehicle_id INT,
    slot_id INT,
    entry_time TIMESTAMP NOT NULL,
    exit_time TIMESTAMP,
    CHECK (exit_time IS NULL OR exit_time > entry_time),
    FOREIGN KEY (vehicle_id) REFERENCES vehicles(vehicle_id) ON DELETE SET NULL,
    FOREIGN KEY (slot_id) REFERENCES parking_slots(slot_id) ON DELETE SET NULL
);
```

5.2 Populate the Database with Sample Data (DML Statements)

Note: This section includes SQL Data Manipulation Language (DML) statements to populate the Parking Management System database with sample data, reflecting ten customers (users), their vehicles, parking slots, reservations, payments, payment methods, payment transactions, and entry/exit logs. The data, dated February 25, 2025, mirrors the normalized schema and demonstrates realistic parking activities, ensuring referential integrity and consistency with the DDL structure. These statements are provided as text for documentation and can be executed in PostgreSQL for testing.

```
-- Insert users (all as customers)
INSERT INTO users (full_name, email, phone_number, role, password_hash) VALUES
('Birhanu Fiseha', 'birhanu.fiseha@email.com', '+251911234567', 'Customer',
'hashed_password1'),
('Biniam Ambachew', 'biniam.ambachew@email.com', '+251922345678', 'Customer',
'hashed_password2'),
('Dagmawit Mesfin', 'dagmawit.mesfin@email.com', '+251933456789', 'Customer',
'hashed_password3'),
('Solomon Setegne', 'solomon.setegne@email.com', '+251944567890', 'Customer',
'hashed_password4'),
('Ermias Lakew', 'ermias.lakew@email.com', '+251955678901', 'Customer', 'hashed_password5'),
('Temesgen Dessie', 'temesgen.dessie@email.com', '+251966789012', 'Customer',
'hashed_password6'),
('Atinaw Dessie', 'atinaw.dessie@email.com', '+251977890123', 'Customer', 'hashed_password7'),
('Getachew Mola', 'getachew.mola@email.com', '+251988901234', 'Customer',
'hashed_password8'),
('Bihoney Gebremeskel', 'bihoney.gebremeskel@email.com', '+251999012345', 'Customer',
'hashed_password9'),
('Frewold Hadgu', 'frewold.hadgu@email.com', '+251900123456', 'Customer',
'hashed_password10');

-- Insert vehicles
INSERT INTO vehicles (user_id, license_plate, vehicle_type, brand, model, color) VALUES
(1, 'ABC123', 'Car', 'Toyota', 'Camry', 'Blue'),
(2, 'XYZ789', 'Car', 'Honda', 'Civic', 'Red'),
(3, 'DEF456', 'Car', 'Ford', 'Focus', 'Black'),
(4, 'GHI789', 'Truck', 'Isuzu', 'D-Max', 'White'),
(5, 'JKL012', 'Car', 'BMW', '3 Series', 'Silver'),
(6, 'MNO345', 'Car', 'Mercedes', 'C-Class', 'Grey'),
(7, 'PQR678', 'Bike', 'Honda', 'CBR', 'Yellow'),
(8, 'STU901', 'Car', 'Audi', 'A4', 'Green'),
```

```sql
(9, 'VWX234', 'Car', 'Volkswagen', 'Golf', 'Purple'),
(10, 'YZA567', 'Car', 'Hyundai', 'Tucson', 'Orange');

-- Insert parking slots
INSERT INTO parking_slots (slot_number, location, status, price_per_hour) VALUES
('P001', 'Level 1', 'Available', 5.00),
('P002', 'Level 1', 'Available', 5.00),
('P003', 'Level 2', 'Available', 5.50),
('P004', 'Level 2', 'Available', 5.50);

-- Insert reservations
INSERT INTO reservations (user_id, vehicle_id, slot_id, start_time, end_time, status) VALUES
(1, 1, 1, '2025-02-25 08:00:00', '2025-02-25 10:00:00', 'Confirmed'),
(2, 2, 1, '2025-02-25 10:30:00', '2025-02-25 12:30:00', 'Confirmed'),
(3, 3, 2, '2025-02-25 09:00:00', '2025-02-25 11:00:00', 'Confirmed'),
(4, 4, 2, '2025-02-25 11:30:00', '2025-02-25 13:30:00', 'Confirmed'),
(5, 5, 3, '2025-02-25 12:00:00', '2025-02-25 14:00:00', 'Confirmed'),
(6, 6, 3, '2025-02-25 14:30:00', '2025-02-25 16:30:00', 'Confirmed'),
(7, 7, 4, '2025-02-25 13:00:00', '2025-02-25 15:00:00', 'Confirmed'),
(8, 8, 4, '2025-02-25 15:30:00', '2025-02-25 17:30:00', 'Confirmed'),
(9, 9, 1, '2025-02-25 16:00:00', '2025-02-25 18:00:00', 'Confirmed'),
(10, 10, 2, '2025-02-25 18:30:00', '2025-02-25 20:30:00', 'Confirmed');

-- Insert payments
INSERT INTO payments (reservation_id, user_id, amount_paid, payment_date, payment_status)
VALUES
(1, 1, 10.00, '2025-02-25 08:15:00', 'Paid'),
(2, 2, 10.00, '2025-02-25 10:45:00', 'Paid'),
(3, 3, 10.00, '2025-02-25 09:15:00', 'Paid'),
(4, 4, 10.00, '2025-02-25 11:45:00', 'Paid'),
(5, 5, 11.00, '2025-02-25 12:15:00', 'Paid'),
(6, 6, 11.00, '2025-02-25 14:45:00', 'Paid'),
(7, 7, 11.00, '2025-02-25 13:15:00', 'Paid'),
(8, 8, 11.00, '2025-02-25 15:45:00', 'Paid'),
(9, 9, 10.00, '2025-02-25 16:15:00', 'Paid'),
(10, 10, 10.00, '2025-02-25 18:45:00', 'Paid');

-- Insert payment methods
INSERT INTO payment_methods (method_name) VALUES
('Credit Card'),
```

```sql
('Mobile Payment'),
('Cash');

-- Insert payment transactions
INSERT INTO payment_transactions (payment_id, method_id) VALUES
(1, 1), -- Credit Card
(2, 2), -- Mobile Payment
(3, 3), -- Cash
(4, 1), -- Credit Card
(5, 2), -- Mobile Payment
(6, 3), -- Cash
(7, 1), -- Credit Card
(8, 2), -- Mobile Payment
(9, 3), -- Cash
(10, 1); -- Credit Card

-- Insert entry/exit logs
INSERT INTO entry_exit_logs (vehicle_id, slot_id, entry_time, exit_time) VALUES
(1, 1, '2025-02-25 08:05:00', '2025-02-25 10:05:00'),
(2, 1, '2025-02-25 10:35:00', '2025-02-25 12:35:00'),
(3, 2, '2025-02-25 09:05:00', '2025-02-25 11:05:00'),
(4, 2, '2025-02-25 11:35:00', '2025-02-25 13:35:00'),
(5, 3, '2025-02-25 12:05:00', '2025-02-25 14:05:00'),
(6, 3, '2025-02-25 14:35:00', '2025-02-25 16:35:00'),
(7, 4, '2025-02-25 13:05:00', '2025-02-25 15:05:00'),
(8, 4, '2025-02-25 15:35:00', '2025-02-25 17:35:00'),
(9, 1, '2025-02-25 16:05:00', '2025-02-25 18:05:00'),
(10, 2, '2025-02-25 18:35:00', NULL); -- Vehicle still parked

Verification queries to check inserted data
SELECT * FROM users LIMIT 10;
SELECT * FROM vehicles;
SELECT * FROM parking_slots;
SELECT * FROM reservations;
SELECT * FROM payments;
SELECT * FROM payment_methods;
SELECT * FROM payment_transactions;
SELECT * FROM entry_exit_logs;
```

5.3 Write SQL Queries to Retrieve and Manipulate Data (e.g., Nested Queries, Joins, Aggregation)

5.3.1 Query 1: Retrieve all reservations for a specific user(using joints)

Note: This query retrieves all parking reservations for a specific user (e.g., Birhanu Fiseha, user_id = 1) to demonstrate user-specific parking activity. It uses JOINs to combine data from the reservations, users, vehicles, and parking_slots tables, showing reservation details like start time, end time, slot number, location, status, and vehicle license plate. See Figure 1 for the query results, captured as an image to visually confirm the output.

```
SELECT r.reservation_id, r.start_time, r.end_time, ps.slot_number, ps.location, ps.status, v.license_plate
FROM reservations r
JOIN users u ON r.user_id = u.user_id
JOIN vehicles v ON r.vehicle_id = v.vehicle_id
JOIN parking_slots ps ON r.slot_id = ps.slot_id
WHERE u.user_id = 1; -- Birhanu Fiseha's reservations
```

	reservation_id integer	start_time timestamp without time zone	end_time timestamp without time zone	slot_number character varying (20)	location character varying (100)	status character varying (20)	license_plate character varying (20)
1	1	2025-02-25 08:00:00	2025-02-25 10:00:00	P001	Level 1	Available	ABC123

Figure 1: Reservations for Birhanu Fiseha. (2025, February 25).

5.3.2 Query 2: Count the number of reservations per parking slot(using aggregation)

Note: This query aggregates the number of reservations per parking slot to analyze parking usage patterns. It uses a LEFT JOIN between parking_slots and reservations, grouped by slot details (slot_id, slot_number, location), and ordered for clarity. The result helps identify high-traffic slots, useful for management decisions. See Figure 2 for the query results, captured as an image to visually confirm the output.

```
SELECT ps.slot_id, ps.slot_number, ps.location, COUNT(r.reservation_id) AS total_reservations
FROM parking_slots ps
LEFT JOIN reservations r ON ps.slot_id = r.slot_id
GROUP BY ps.slot_id, ps.slot_number, ps.location
ORDER BY ps.slot_id;
```

	slot_id [PK] integer	slot_number character varying (20)	location character varying (100)	total_reservations bigint
1	1	P001	Level 1	2
2	2	P002	Level 1	2
3	3	P003	Level 2	2
4	4	P004	Level 2	2
5	5	P005	Level 3	2

Figure 2: Reservations per parking slot. (2025, February 25).

5.3.3 Query 3: Find available parking slots not currently reserved(using nested query)

Note: This query identifies parking slots available for booking on February 25, 2025, by checking slots with a status of 'Available' and excluding those reserved (status 'Confirmed' or 'Pending') during that day. It uses a nested query to filter reservations and ensures real-time availability for users. See Figure 3 for the query results, captured as an image to visually confirm the output.

```
SELECT slot_id, slot_number, location, status
FROM parking_slots ps
WHERE status = 'Available'
AND slot_id NOT IN (
   SELECT slot_id
   FROM reservations
   WHERE status IN ('Confirmed', 'Pending')
   AND (start_time <= '2025-02-25 23:59:59' AND end_time >= '2025-02-25 00:00:00')
);
```

Data Output Messages Notifications

	slot_id [PK] integer	slot_number character varying (20)	location character varying (100)	status character varying (20)

Figure 3: Available parking slots. (2025, February 25).

5.3.4 Query 4: Calculate total payment amount per user (using aggregation with join)

Note: This query calculates the total payment amount for each user using SUM and GROUP BY, combined with a JOIN between users and payments tables, demonstrating aggregation with join operations for financial analysis of user parking activity. See Figure 4 for the query results, captured as an image to visually confirm the output.

```
SELECT u.user_id, u.full_name, SUM(p.amount_paid) AS total_payments
FROM users u
JOIN payments p ON u.user_id = p.user_id
GROUP BY u.user_id, u.full_name
ORDER BY u.user_id;
```

Data Output Messages Notifications

	user_id [PK] integer	full_name character varying (100)	total_payments numeric
1	1	Birhanu Fiseha	10.00
2	2	Biniam Ambachew	10.00
3	3	Dagmawit Mesfin	10.00
4	4	Solomon Setegne	10.00
5	5	Ermias Lakew	11.00
6	6	Temesgen Dessie	11.00
7	7	Atinaw Dessie	11.00
8	8	Getachew Mola	11.00
9	9	Bihoney Gebremeskel	12.00
10	10	Frewold Hadgu	12.00

Figure 4: Total payment amount per user with joins (2025, February 25)

5.3.5 Query 5: List vehicles and their entry/exit times for a specific slot (using join with nested query)

Note: This query lists vehicles and their entry/exit times for slot 1 on February 25, 2025, using a JOIN between vehicles and entry_exit_logs and a nested condition in the WHERE clause to filter by date, showcasing join with nested query functionality for tracking vehicle activity in a specific slot. See Figure 5 for the query results, captured as an image to visually confirm the output.

```
SELECT v.vehicle_id, v.license_plate, v.vehicle_type, e.entry_time, e.exit_time
FROM vehicles v
JOIN entry_exit_logs e ON v.vehicle_id = e.vehicle_id
WHERE e.slot_id = 1
AND e.entry_time >= '2025-02-25 00:00:00' AND (e.exit_time IS NULL OR e.exit_time <=
'2025-02-25 23:59:59');
```

Data Output | Messages | Notifications

Showing rows: 1 to 2

	vehicle_id integer	license_plate character varying (20)	vehicle_type character varying (50)	entry_time timestamp without time zone	exit_time timestamp without time zone
1	1	ABC123	Car	2025-02-25 08:05:00	2025-02-25 10:05:00
2	2	XYZ789	Car	2025-02-25 10:35:00	2025-02-25 12:35:00

Figure 5: Vehicles and entry/exit times for slot 1 with join and nested query (2025, February 25).

5.3.6 Query 6: Total reservations and payments by user with aggregation and nested query

Note: This query aggregates the total number of reservations and payment amounts per user, using COUNT and SUM with a nested query to filter only completed reservations, combined with JOINs across users, reservations, and payments tables, demonstrating aggregation with nested query functionality for user activity analysis. See Figure 6 for the query results, captured as an image to visually confirm the output.

```
SELECT u.user_id, u.full_name, COUNT(r.reservation_id) AS total_reservations,
SUM(p.amount_paid) AS total_payments
FROM users u
JOIN reservations r ON u.user_id = r.user_id
JOIN payments p ON r.reservation_id = p.reservation_id
WHERE r.reservation_id IN (
   SELECT reservation_id
   FROM reservations
   WHERE status = ‘Completed’
)
GROUP BY u.user_id, u.full_name
ORDER BY u.user_id;
```

Data Output Messages Notifications

	user_id [PK] integer	full_name character varying (100)	total_reservations bigint	total_payments numeric

Figure 6: Total reservations and payments by user with aggregation and nested query (Fiseha, 2025

6 Phase 4: Advanced Features (Bonus)

6.1 Create Views for Frequently Accessed Data

Note: This section creates SQL views to simplify access to frequently queried data in the Parking Management System, improving readability and performance for common operations. Views abstract complex queries, such as listing available parking slots or user reservations, making them reusable for end-users or administrators. See Figure 1 for the result of querying available_parking_slots, and Figure 2 for user_reservations, captured as images to visually confirm the output.

```
-- View for available parking slots
CREATE VIEW available_parking_slots AS
SELECT slot_id, slot_number, location, status, price_per_hour
FROM parking_slots
WHERE status = 'Available';

SELECT * FROM available_parking_slots;
```

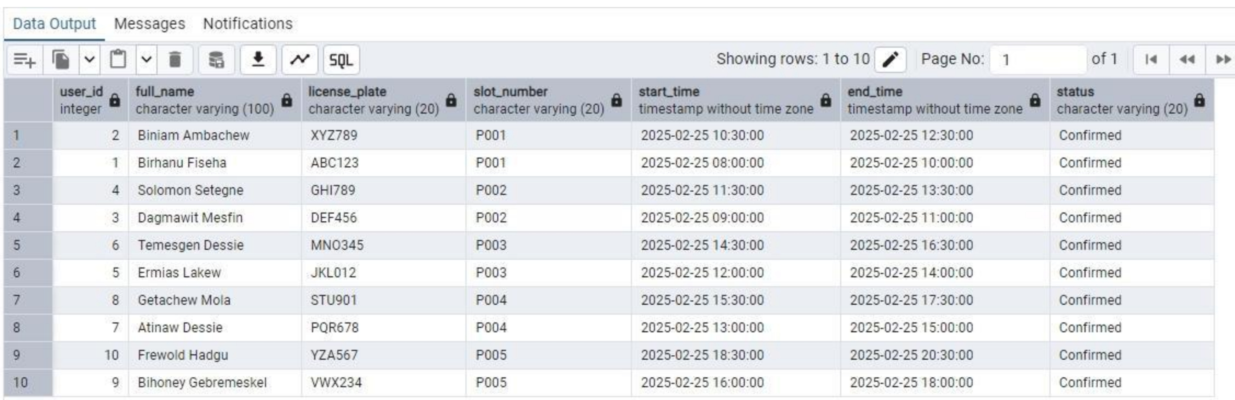
Data Output Messages Notifications

	slot_id integer	slot_number character varying (20)	location character varying (100)	status character varying (20)	price_per_hour numeric (10,2)
1	1	P001	Level 1	Available	5.00
2	2	P002	Level 1	Available	5.00
3	3	P003	Level 2	Available	5.50
4	4	P004	Level 2	Available	5.50
5	5	P005	Level 3	Available	6.00

Figure 1: Available parking slots view (2025, February 25).

```
-- View for user reservations with vehicle and slot details
CREATE VIEW user_reservations AS
SELECT u.user_id, u.full_name, v.license_plate, ps.slot_number, r.start_time, r.end_time, r.status
FROM users u
JOIN reservations r ON u.user_id = r.user_id
JOIN vehicles v ON r.vehicle_id = v.vehicle_id
JOIN parking_slots ps ON r.slot_id = ps.slot_id
WHERE r.status IN ('Confirmed', 'Pending');

SELECT * FROM user_reservations;
```



	user_id integer	full_name character varying (100)	license_plate character varying (20)	slot_number character varying (20)	start_time timestamp without time zone	end_time timestamp without time zone	status character varying (20)
1	2	Biniam Ambachew	XYZ789	P001	2025-02-25 10:30:00	2025-02-25 12:30:00	Confirmed
2	1	Birhanu Fiseha	ABC123	P001	2025-02-25 08:00:00	2025-02-25 10:00:00	Confirmed
3	4	Solomon Setegne	GHI789	P002	2025-02-25 11:30:00	2025-02-25 13:30:00	Confirmed
4	3	Dagmawit Mesfin	DEF456	P002	2025-02-25 09:00:00	2025-02-25 11:00:00	Confirmed
5	6	Temesgen Dessie	MNO345	P003	2025-02-25 14:30:00	2025-02-25 16:30:00	Confirmed
6	5	Ermias Lakew	JKL012	P003	2025-02-25 12:00:00	2025-02-25 14:00:00	Confirmed
7	8	Getachew Mola	STU901	P004	2025-02-25 15:30:00	2025-02-25 17:30:00	Confirmed
8	7	Atinaw Dessie	PQR678	P004	2025-02-25 13:00:00	2025-02-25 15:00:00	Confirmed
9	10	Frewold Hadgu	YZA567	P005	2025-02-25 18:30:00	2025-02-25 20:30:00	Confirmed
10	9	Bihoney Gebremeskel	VWX234	P005	2025-02-25 16:00:00	2025-02-25 18:00:00	Confirmed

Figure 2: User reservations view (2025, February 25).

6.2 Implement Indexes to Optimize Query Performance

Note: This section adds indexes to frequently queried columns in the Parking Management System to enhance query performance, particularly for joins and searches. Indexes on foreign keys and status fields improve efficiency for operations like finding reservations or available slots.See Figure 3 for the confirmation of index creation, captured as an image to visually confirm the output.

```
-- Index on user_id in reservations for faster joins

CREATE INDEX idx_reservations_user_id ON reservations(user_id);

-- Index on vehicle_id in reservations for faster joins

CREATE INDEX idx_reservations_vehicle_id ON reservations(vehicle_id);

-- Index on slot_id in reservations for faster slot-related queries
```

CREATE INDEX idx_reservations_slot_id ON reservations(slot_id);

-- Index on status in parking_slots for faster status checks

CREATE INDEX idx_parking_slots_status ON parking_slots(status);

-- Index on user_id in payments for faster joins

CREATE INDEX idx_payments_user_id ON payments(user_id);

Figure 3: Index creation confirmation (2025, February 25).

6.3 Write Stored Procedures or Triggers for Automation (Optional, for Advanced Students)

Note: This section implements a stored procedure and trigger to automate parking management tasks, enhancing system efficiency and enforcing business rules. The stored procedure books a parking slot, and the trigger updates slot status upon reservation confirmation, ensuring data consistency and automation for advanced functionality. See Figures 4 and 5 for the execution results, captured as images to visually confirm the output.

6.3.1 -- Stored procedure to book a parking slot

CREATE OR REPLACE PROCEDURE book_parking_slot(

```
  p_user_id INT, p_vehicle_id INT, p_slot_id INT, p_start_time TIMESTAMP, p_end_time
TIMESTAMP
)
LANGUAGE plpgsql
AS $$
BEGIN
  IF EXISTS (
    SELECT 1
    FROM parking_slots ps
    WHERE ps.slot_id = p_slot_id AND ps.status != 'Available'
  ) THEN
    RAISE EXCEPTION 'Slot % is not available', p_slot_id;
  END IF;

  IF EXISTS (
    SELECT 1
    FROM reservations r
    WHERE r.slot_id = p_slot_id
    AND r.status IN ('Confirmed', 'Pending')
    AND (r.start_time <= p_end_time AND r.end_time >= p_start_time)
  ) THEN
    RAISE EXCEPTION 'Slot % is already reserved for the given time', p_slot_id;
  END IF;

  INSERT INTO reservations (user_id, vehicle_id, slot_id, start_time, end_time, status)
  VALUES (p_user_id, p_vehicle_id, p_slot_id, p_start_time, p_end_time, 'Pending');

  UPDATE parking_slots
  SET status = 'Reserved'
  WHERE slot_id = p_slot_id;

  RAISE NOTICE 'Parking slot booked successfully';
EXCEPTION
  WHEN OTHERS THEN
    RAISE EXCEPTION 'Booking failed: %', SQLERRM;
END;
$$;
```

6.3.2 -- Trigger to update slot status on reservation confirmation

```
CREATE OR REPLACE FUNCTION update_slot_status()
RETURNS TRIGGER AS $$
BEGIN
   IF NEW.status = 'Confirmed' AND OLD.status = 'Pending' THEN
      UPDATE parking_slots
      SET status = 'Occupied'
      WHERE slot_id = NEW.slot_id;
   END IF;
   IF NEW.status IN ('Completed', 'Cancelled') AND OLD.status = 'Confirmed' THEN
      UPDATE parking_slots
      SET status = 'Available'
      WHERE slot_id = NEW.slot_id;
   END IF;
   RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER trg_update_slot_status
AFTER UPDATE ON reservations
FOR EACH ROW
EXECUTE FUNCTION update_slot_status();
```

6.3.3 . Test and Check for Stored Procedure Execution.

Objective: Verify the book_parking_slot procedure works, booking a parking slot and updating its status to ‘Reserved’, then capture the output for the image.

SQL Code to Test:

-- Book a parking slot for Dagmawit Mesfin (user_id = 3, vehicle_id = 3, slot_id = 1) on Feb 25, 2025

CALL book_parking_slot(3, 3, 1, '2025-02-25 09:00:00', '2025-02-25 11:00:00');

Figure 4a: Stored procedure success message. (2025, February 25).

-- Verify the reservation was created

SELECT * FROM reservations WHERE reservation_id = (SELECT MAX(reservation_id) FROM reservations);

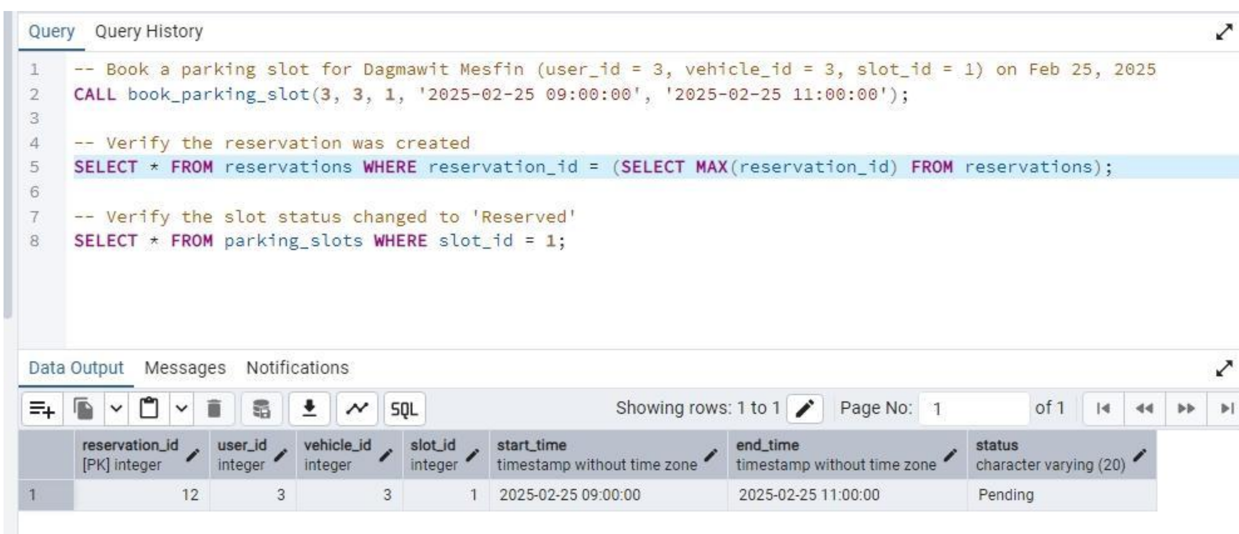


Figure 4b: New reservation entry after booking. (2025, February 25)

-- Verify the slot status changed to 'Reserved'

SELECT * FROM parking_slots WHERE slot_id = 1;

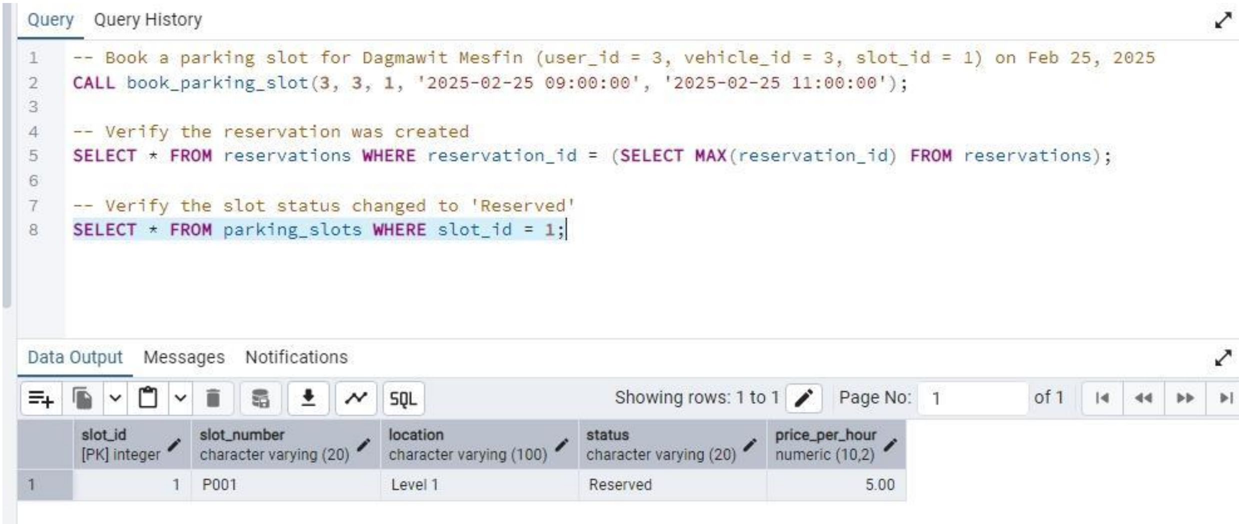


Figure 4c: Updated slot status after booking.

6.3.4 Test and Check for Trigger Execution.

Objective: Verify the update_slot_status trigger updates the parking_slots.status to ‘Occupied’ when a reservation status changes to ‘Confirmed’, then capture the result for the image.

SQL Code to Test

-- Update the reservation status to 'Confirmed' (using the latest reservation_id from the stored procedure)

UPDATE reservations SET status = 'Confirmed' WHERE reservation_id = (SELECT MAX(reservation_id) FROM reservations);

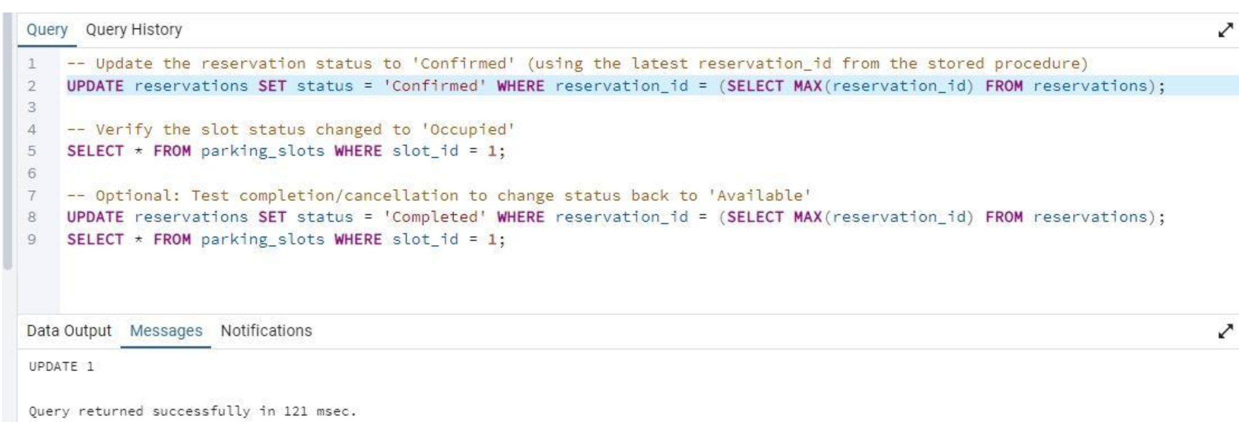


Figure 5a: Trigger update command execution. (2025, February 25).

-- Verify the slot status changed to 'Occupied'

SELECT * FROM parking_slots WHERE slot_id = 1;

Figure 5b: Slot status updated to ‘Occupied’. (2025, February 25)

-- Optional: Test completion/cancellation to change status back to 'Available'

UPDATE reservations SET status = 'Completed' WHERE reservation_id = (SELECT MAX(reservation_id) FROM reservations);

SELECT * FROM parking_slots WHERE slot_id = 1;

Figure 5c: Slot status updated to ‘Available’. (2025, February 25).

Conclusion

The Parking Management System (PMS) developed in this project successfully addresses the inefficiencies and challenges associated with traditional manual parking systems. By leveraging a relational database approach, the system provides a robust framework for managing parking slots, reservations, vehicle tracking, payments, and entry/exit logs. The design process, encompassing requirements analysis, entity-relationship modeling, normalization to Third Normal Form (3NF), and SQL implementation, ensures data integrity, scalability, and efficiency. The integration of advanced features such as views, indexes, stored procedures, and triggers further enhances the system's functionality, automating key operations like slot booking and status updates while optimizing query performance.

The system meets its general objective of improving parking management efficiency, security, and user experience while delivering actionable data insights for parking operators. Specific objectives, including centralized data storage, online reservations, automated tracking, digital payments, and analytics generation, were achieved through a structured methodology and database-driven solutions. The sample data and SQL queries demonstrate practical application, providing real-time insights into slot availability, user activity, and financial transactions. This project not only showcases the application of database concepts but also lays the foundation for a scalable, real-world parking management solution that can be extended with additional features like a user interface or mobile app integration in future iterations.

References

- Elmasri, R., & Navathe, S. B. (2015). Fundamentals of Database Systems (7th ed.). Pearson.
- Silberschatz, A., Korth, H. F., & Sudarshan, S. (2019). Database System Concepts (7th ed.). McGraw-Hill Education.
- Date, C. J. (2003). An Introduction to Database Systems (8th ed.). Addison-Wesley.
- PostgreSQL Global Development Group. (2025). PostgreSQL Documentation. Retrieved from https://www.postgresql.org/docs/.
- Hoffer, J. A., Ramesh, V., & Topi, H. (2016). Modern Database Management (12th ed.). Pearson.