CONTENTS

1. ABSTRACT

The **Garage Management System** is a database-driven application designed to efficiently manage **vehicle servicing, customer records, and staff assignments** within a garage. The system aims to streamline operations by replacing traditional manual record-keeping with a structured **MySQL database**, ensuring **faster data retrieval, accurate service tracking, and improved management of resources**.

This system maintains detailed records of **vehicle owners, their vehicles, service requests, assigned specialists, and workers involved**. It allows garage owners to efficiently track **pending and completed services, calculate costs, and generate reports** using structured **SQL queries**. The system also ensures proper **assignment of specialists based on expertise**, reducing delays and enhancing service quality.

By implementing this **Garage Management System**, we aim to **optimize workflow, reduce errors in data handling, and improve customer satisfaction**. This project provides a practical approach to **database design, SQL query execution, and real-world problem-solving**, making it a valuable learning experience in **Database Management Systems (DBMS).**

2. INTRODUCATION

In today's fast-paced world, managing a garage efficiently requires a structured and automated system to handle **vehicle servicing, customer records, staff assignments, and billing**. Traditional manual record-keeping often leads to **errors, delays, and inefficiencies**, making it difficult for garage owners to keep track of service requests, vehicle histories, and staff availability.

The **Garage Management System** is designed to address these challenges by providing a **database-driven solution** for streamlined operations. This system ensures that all necessary data, including **vehicle details, owner information, service history, assigned specialists, and workers**, is stored securely and can be retrieved quickly when needed.

Using **MySQL as the database management system**, this project enables smooth **tracking of service requests, calculation of service costs, and management of staff assignments**. By executing structured **SQL queries**, the system helps in generating reports, analyzing garage performance, and improving decision-making for both garage owners and customers.

This project not only enhances **operational efficiency** but also serves as a **real-world implementation of database concepts**, helping us understand **data organization, retrieval, and optimization**. The goal is to develop a system that minimizes **manual errors, speeds up service operations, and improves customer satisfaction** through better management and record-keeping.

2.1 Problem Statement.

Managing a garage efficiently requires handling multiple aspects, including vehicle servicing, customer records, staff assignments, and service billing. Traditionally, garages rely on manual record-keeping or paper-based systems, which often lead to several challenges:

- Data Management Issues – Difficulty in maintaining accurate records of vehicle owners, service history, and staff assignments, leading to mismanagement.
- Service Tracking Problems – Lack of a proper system to track pending, ongoing, and completed service requests, causing delays and customer dissatisfaction.
- Billing and Cost Calculation Errors – Manual invoicing can result in calculation mistakes, affecting financial records and transparency.
- Resource Allocation Challenges – Improper assignment of specialists and workers, leading to inefficient workflow and extended service times.
- Lack of Data Retrieval and Reporting – Difficulty in quickly retrieving past service history, customer details, and financial reports, making decision-making slow and ineffective.

To address these issues, we propose a Garage Management System using MySQL as the database management system. This system provides a centralized database to store and manage all records efficiently, allowing for faster service tracking, accurate cost calculation, automated

staff allocation, and easy report generation. By implementing this system, we aim to enhance garage operations, minimize manual errors, and improve overall efficiency.

2.2 . Objectives

The primary objective of the **Garage Management System** is to develop an efficient **database-driven solution** to streamline garage operations, ensuring **accurate record-keeping, faster service tracking, and optimized resource allocation**. The specific objectives of this project are:

1. **Efficient Data Management** – To store and manage records of **vehicle owners, service history, staff details, and cost estimates** in a structured MySQL database.
2. **Service Tracking & Management** – To enable **real-time tracking of pending, ongoing, and completed services**, reducing delays and improving workflow.
3. **Automated Billing & Cost Calculation** – To minimize **human errors** by automating service charge calculations and invoice generation.
4. **Data Retrieval & Report Generation** – To execute **SQL queries for quick retrieval of customer details, vehicle history, and financial reports** for better decision-making.
5. **Reduce Manual Errors & Paperwork** – To eliminate **paper-based record-keeping**, making data storage and management more **secure, efficient, and scalablity.**

3. Data Types

MySQL uses many different data types broken into three categories: numeric, date and time, and string types.

Numeric Data Types:

MySQL uses all the standard ANSI SQL numeric data types, so if you're coming to MySQL from a different database system, these definitions will look familiar to you. The following list shows the common numeric data types and their descriptions:

- **INT** - A normal-sized integer that can be signed or unsigned. If signed, the allowable range is from -2147483648 to 2147483647. If unsigned, the allowable range is from 0 to 4294967295. You can specify a width of up to 11 digits.
- **TINYINT** - A very small integer that can be signed or unsigned. If signed, the allowable range is from -128 to 127. If unsigned, the allowable range is from 0 to 255. You can specify a width of up to 4 digits.
- **SMALLINT** - A small integer that can be signed or unsigned. If signed, the allowable range is from -32768 to 32767. If unsigned, the allowable range is from 0 to 65535. You can specify a width of up to 5 digits.
- **MEDIUMINT** - A medium-sized integer that can be signed or unsigned. If signed, the allowable range is from -8388608 to 8388607. If unsigned, the allowable range is from 0 to 16777215. You can specify a width of up to 9 digits.
- **BIGINT** - A large integer that can be signed or unsigned. If signed, the allowable range is from -9223372036854775808 to 9223372036854775807. If unsigned, the allowable range is from 0 to 18446744073709551615. You can specify a width of up to 20 digits.
- **FLOAT(M,D)** - A floating-point number that cannot be unsigned. You can define the display length (M) and the number of decimals (D). This is not required and will default to 10,2, where 2 is the number of decimals and 10 is the total number of digits (including decimals). Decimal precision can go to 24 places for a FLOAT.
- **DOUBLE(M,D)** - A double precision floating-point number that cannot be unsigned. You can define the display length (M) and the number of decimals (D). This is not required and will default to 16,4, where 4 is the number of decimals. Decimal precision can go to 53 places for a DOUBLE. REAL is a synonym for DOUBLE.
- **DECIMAL(M,D)** - An unpacked floating-point number that cannot be unsigned. In unpacked decimals, each decimal corresponds to one byte. Defining the display length (M) and the number of decimals (D) is required. NUMERIC is a synonym for DECIMAL.

Date and Time Types:

The MySQL date and time datatypes are:

- **DATE** - A date in YYYY-MM-DD format, between 1000-01-01 and 9999-12-31. For example, December 30th, 1973 would be stored as 1973-12-30.

- **DATETIME** - A date and time combination in YYYY-MM-DD HH:MM:SS format, between 1000-01-01 00:00:00 and 9999-12-31 23:59:59. For example, 3:30 in the afternoon on December 30th, 1973 would be stored as 1973-12-30 15:30:00.
- **TIMESTAMP** - A timestamp between midnight, January 1, 1970 and sometime in 2037. This looks like the previous DATETIME format, only without the hyphens between numbers; 3:30 in the afternoon on December 30th, 1973 would be stored as 19731230153000 (YYYYMMDDHHMMSS).
- **TIME** - Stores the time in HH:MM:SS format.
- **YEAR(M)** - Stores a year in 2-digit or 4-digit format. If the length is specified as 2 (for example YEAR(2)), YEAR can be 1970 to 2069 (70 to 69). If the length is specified as 4, YEAR can be 1901 to 2155. The default length is 4.

String Types:

Although numeric and date types are fun, most data you'll store will be in string format. This list describes the common string datatypes in MySQL.

- **CHAR(M)** - A fixed-length string between 1 and 255 characters in length (for example CHAR(5)), right-padded with spaces to the specified length when stored. Defining a length is not required, but the default is 1.
- **VARCHAR(M)** - A variable-length string between 1 and 255 characters in length; for example VARCHAR(25). You must define a length when creating a VARCHAR field.
- **BLOB or TEXT** - A field with a maximum length of 65535 characters. BLOBs are "Binary Large Objects" and are used to store large amounts of binary data, such as images or other types of files. Fields defined as TEXT also hold large amounts of data; the difference between the two is that sorts and comparisons on stored data are case sensitive on BLOBs and are not case sensitive in TEXT fields. You do not specify a length with BLOB or TEXT.
- **TINYBLOB or TINYTEXT** - A BLOB or TEXT column with a maximum length of 255 characters. You do not specify a length with TINYBLOB or TINYTEXT.
- **MEDIUMBLOB or MEDIUMTEXT** - A BLOB or TEXT column with a maximum length of 16777215 characters. You do not specify a length with MEDIUMBLOB or MEDIUMTEXT.
- **LONGBLOB or LONGTEXT** - A BLOB or TEXT column with a maximum length of 4294967295 characters. You do not specify a length with LONGBLOB or LONGTEXT.
- **ENUM** - An enumeration, which is a fancy term for list. When defining an ENUM, you are creating a list of items from which the value must be selected (or it can be NULL). For example, if you wanted your field to contain "A" or "B" or "C", you would define your ENUM as ENUM ('A', 'B', 'C') and only those values (or NULL) could ever populate that field.

4. Data Modeling using ER Model

To design a **Garage Management System**, we follow a structured approach using the **Entity-Relationship (ER) model**. This helps in visualizing how different entities interact within the system, ensuring an efficient and well-structured data.

4.1 Requirement Collection and Analysis

Before designing the database, we analyze the key components required for a **garage management system**. The system must handle information related to **garage owners, vehicle owners, vehicles, service requests, specialists, and workers**.
Key requirements:

- Store and manage details of **vehicle owners and their vehicles**.
- Keep track of **services, repairs, and maintenance history**.
- Assign **specialists and workers** to specific service requests.
- Generate **service bills and maintain transaction records**.
- Ensure **efficient data retrieval** through structured queries.

4.1.1 Entity Types, Entity Sets, Attributes, and Keys

Entity	Attributes (Keys in bold)
Garage Owner	**Owner_ID** (PK), Name, Contact, Address
Vehicle Owner	**Owner_ID** (PK), Name, Contact, Email, Address
Vehicle	**Vehicle_ID** (PK), Model, Type, Registration_No, **Owner_ID** (FK)
Service Request	**Service_ID** (PK), **Vehicle_ID** (FK), Date, Status, Total_Cost
Specialist	**Specialist_ID** (PK), Name, Expertise, Contact
Worker	**Worker_ID** (PK), Name, Contact, Experience
Assignment	**Assignment_ID** (PK), **Service_ID** (FK), **Specialist_ID** (FK), **Worker_ID** (FK), Work_Details
Bill	**Bill_ID** (PK), **Service_ID** (FK), Amount, Payment_Status

Table no:4.1.1 Entity Types & Attributes

Primary Keys (PK) uniquely identify each record.
Foreign Keys (FK) establish relationships between different entities.

ER Diagram Representation

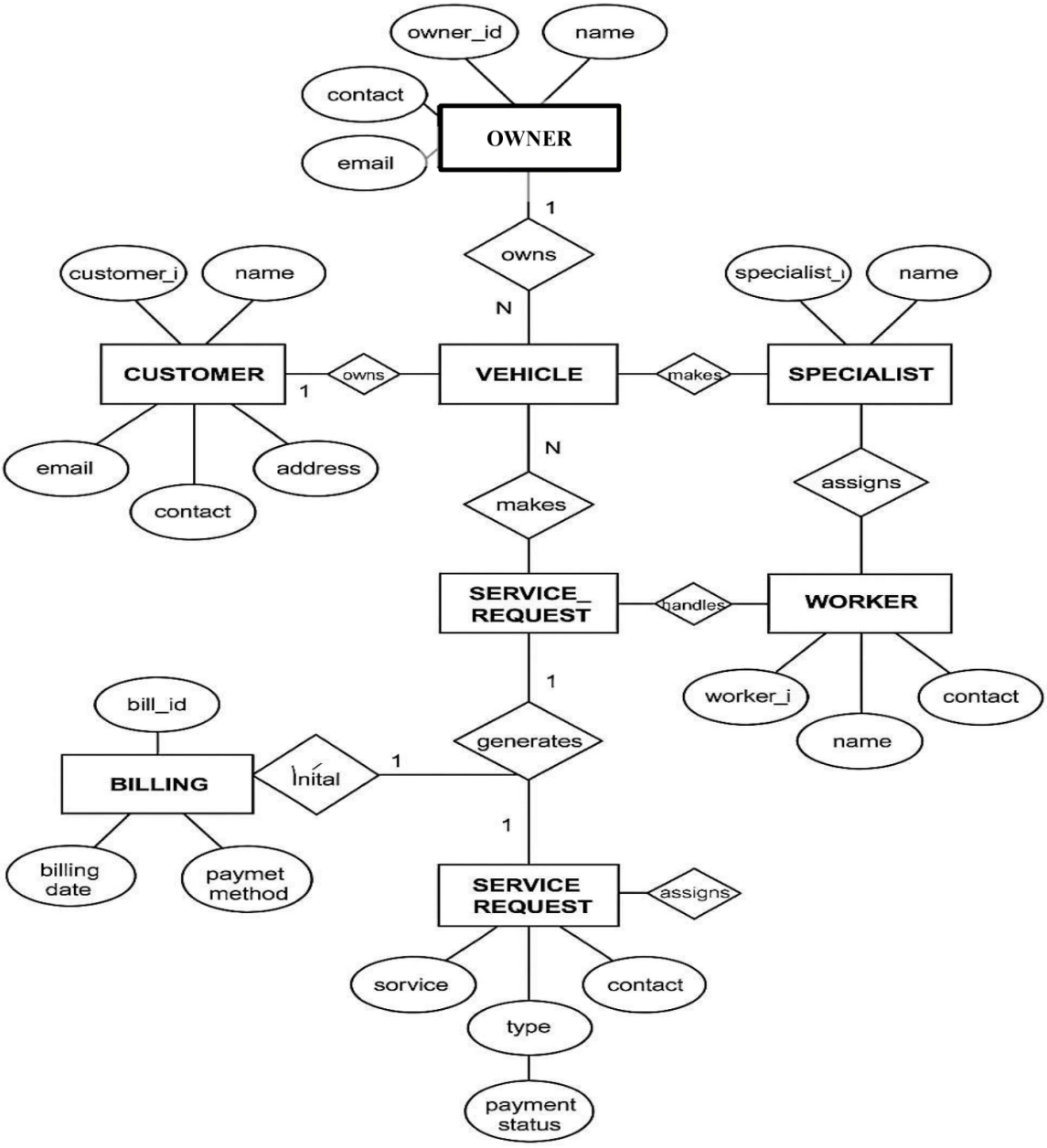


Fig 4.1.1-ER Diagram of Garage Management System

5. Relational Database Design Using ER-to-Relational Mapping

ER-to-Relational Mapping Steps

---

1. Entity Sets to Tables
Each entity becomes a table with its attributes and a primary key.

- Garage_Owner(owner_id, name, contact, email)
- Customer(customer_id, name, contact, email, address)
- Vehicle(vehicle_id, customer_id, model, registration_number, vehicle_type, last_service_date)

➞ Foreign Key: customer_id → Customer(customer_id)

- Specialist(specialist_id, name, expertise, contact)
- Worker(worker_id, name, assigned_specialist_id, contact)

➞ Foreign Key: assigned_specialist_id → Specialist(specialist_id)

- Service_Request(service_id, vehicle_id, specialist_id, worker_id, service_type, cost, status, payment_status, date)

➞ **Foreign Keys:**

- vehicle_id → Vehicle(vehicle_id)
- specialist_id → Specialist(specialist_id)
- worker_id → Worker(worker_id)

- Billing(bill_id, service_id, customer_id, billing_date, total_amount, payment_method, payment_status)

➞ **Foreign Keys:**

- service_id → Service_Request(service_id)
- customer_id → Customer(customer_id)

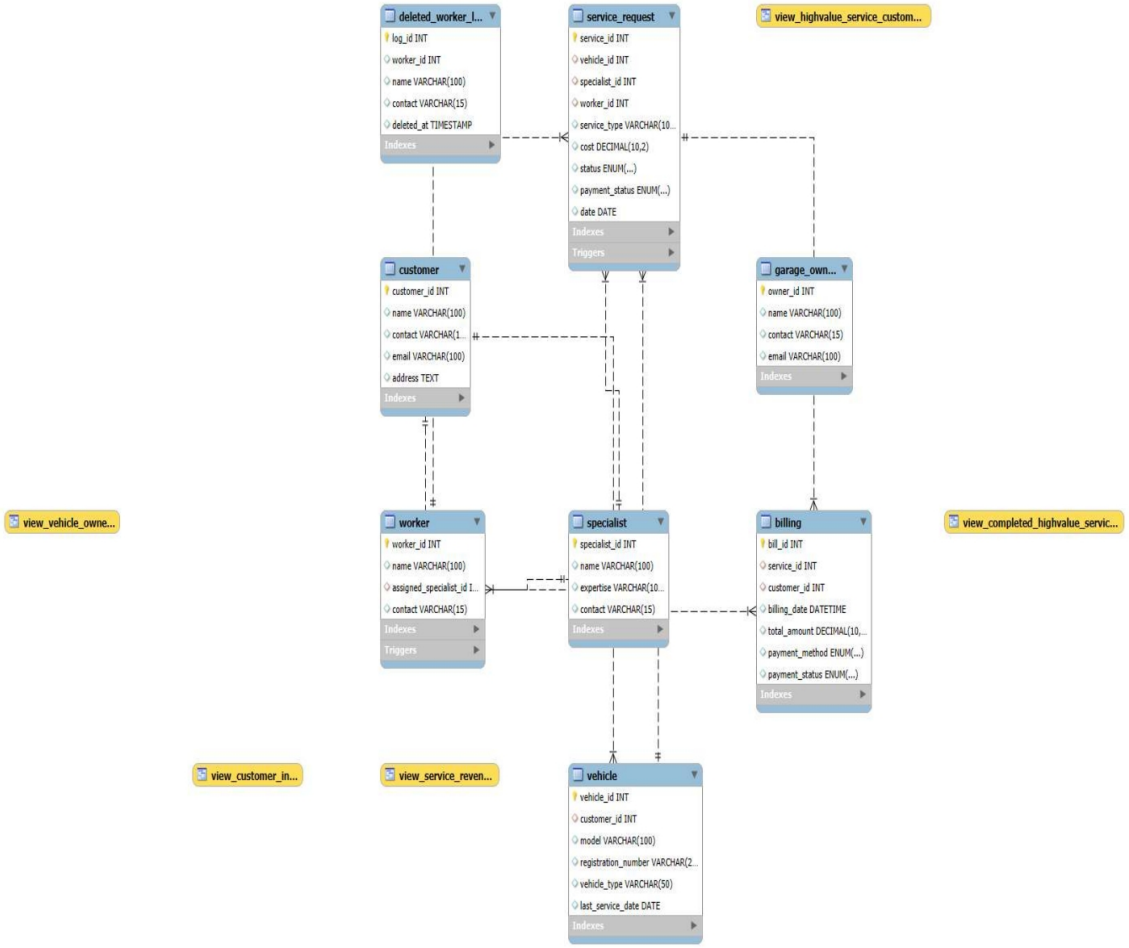
---

2. Relationships to Foreign Keys

Each relationship (e.g., *Customer owns Vehicle*, *Worker assigned to Specialist*) is implemented using foreign keys already present in tables.

- One-to-Many:
  - One Customer → Many Vehicles
  - One Specialist → Many Workers
  - One Vehicle → Many Service Requests
  - One Service Request → One Billing

SCHEMA DIAGRAM:-



.

Fig 5.1.1 Schema Diagram

6. Creating database using MySQL

An SQL schema is identified by a schema name, and includes an authorization identifier to indicate the user or account who owns the schema, as well as descriptors for each element in the schema. Schema elements include tables, constraints, views, domains, and other constructs (such as authorization grants) that describe the schema. A schema is created via the CREATE SCHEMA statement, which can include all the schema elements' definitions. Alternatively, the schema can be assigned a name and authorization identifier, and the elements can be defined later. The CREATE TABLE command is used to specify a new relation by giving it a name and specifying its attributes and initial constraints. The attributes are specified first, and each attribute is given a name, a data type to specify its domain of values, and any attribute constraints, such as NOT NULL. The key, entity integrity, and referential integrity constraints can be specified within the CREATE TABLE statement after the attributes are declared, or they can be added later using the ALTER TABLE command (see Chapter 5). Figure 4.1 shows sample data definition statements in SQL for the COMPANY relational database schema.

To create a database using MySQL, you typically write SQL statements to:

1. Create the database
2. Use the database
3. Create tables with keys and constraints
4. Insert sample data
5. (Optional) Create views, procedures, triggers, etc.

1. Create Database and Use It

```
CREATE DATABASE GarageDB;
USE GarageDB;
```

2. Create Tables Based on ER Diagram

📌 Garage Owner

```
CREATE TABLE Garage_Owner (
   owner_id INT AUTO_INCREMENT PRIMARY KEY,
   name VARCHAR(100),
   contact VARCHAR(15),
   email VARCHAR(100) UNIQUE
);
```

📌 Customer (Vehicle Owner)

```
CREATE TABLE Customer (
    customer_id INT AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(100),
    contact VARCHAR(15),
    email VARCHAR(100) UNIQUE,
    address TEXT
);
```

📌 Vehicle

```
CREATE TABLE Vehicle (
    vehicle_id INT AUTO_INCREMENT PRIMARY KEY,
    customer_id INT,
    model VARCHAR(100),
    registration_number VARCHAR(20) UNIQUE,
    vehicle_type VARCHAR(50),
    last_service_date DATE,
    FOREIGN KEY (customer_id) REFERENCES Customer(customer_id) ON DELETE
CASCADE
);
```

📌 Specialist

```
CREATE TABLE Specialist (
    specialist_id INT AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(100),
    expertise VARCHAR(100),
    contact VARCHAR(15)
);
```

📌 Worker

```
CREATE TABLE Worker (
    worker_id INT AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(100),
    assigned_specialist_id INT,
    contact VARCHAR(15),
    FOREIGN KEY (assigned_specialist_id) REFERENCES Specialist(specialist_id) ON
DELETE SET NULL
);
```

📌 **Service Request**

```
CREATE TABLE Service_Request (
   service_id INT AUTO_INCREMENT PRIMARY KEY,
   vehicle_id INT,
   specialist_id INT,
   worker_id INT,
   service_type VARCHAR(100),
   cost DECIMAL(10,2),
   status ENUM('Pending', 'In Progress', 'Completed'),
   payment_status ENUM('Pending', 'Completed', 'Failed') DEFAULT 'Pending',
   date DATE,
   FOREIGN KEY (vehicle_id) REFERENCES Vehicle(vehicle_id) ON DELETE CASCADE,
   FOREIGN KEY (specialist_id) REFERENCES Specialist(specialist_id) ON DELETE SET
NULL,
   FOREIGN KEY (worker_id) REFERENCES Worker(worker_id) ON DELETE SET NULL
);
```

📌 **Billing**

```
CREATE TABLE Billing (
   bill_id INT AUTO_INCREMENT PRIMARY KEY,
   service_id INT,
   customer_id INT,
   billing_date DATE DEFAULT (CURRENT_DATE),
   total_amount DECIMAL(10,2),
   payment_method ENUM('Cash', 'Card', 'UPI', 'Net Banking'),
   payment_status ENUM('Pending', 'Completed', 'Failed') DEFAULT 'Pending',
   FOREIGN KEY (service_id) REFERENCES Service_Request(service_id) ON DELETE
CASCADE,
   FOREIGN KEY (customer_id) REFERENCES Customer(customer_id) ON DELETE
CASCADE
);
```

3. Sample Data Insertion (Optional)

```
-- Insert sample Garage Owner
INSERT INTO Garage_Owner (name, contact, email) VALUES
('Rajesh Sharma', '9876543210', 'rajesh.sharma@example.com');



-- Insert sample Customers
INSERT INTO Customer (name, contact, email, address) VALUES
```

```
('Amit Verma', '9123456789', 'amit.verma@example.com', 'Sector 21, Gurugram, Haryana'),
('Pooja Iyer', '9898989898', 'pooja.iyer@example.com', 'Koramangala, Bengaluru, Karnataka');

-- Insert sample Vehicles
INSERT INTO Vehicle (customer_id, model, registration_number, vehicle_type,
last_service_date) VALUES
(1, 'Maruti Suzuki Swift', 'DL3CAP1234', 'Hatchback', '2025-03-10'),
(2, 'Hyundai Creta', 'KA01MK5678', 'SUV', '2025-02-20');

-- Insert sample Specialists
INSERT INTO Specialist (name, expertise, contact) VALUES
('Sandeep Yadav', 'Engine Repair', '9000011122'),
('Manish Tiwari', 'Electrical Work', '9112233445');

-- Insert sample Workers
INSERT INTO Worker (name, assigned_specialist_id, contact) VALUES
('Ravi Kumar', 1, '9223344556'),
('Anjali Mehta', 2, '9556677889');

-- Insert sample Service Requests
INSERT INTO Service_Request (vehicle_id, specialist_id, worker_id, service_type, cost, status,
payment_status, date) VALUES
(1, 1, 1, 'Clutch Overhaul', 8500.00, 'Pending', 'Pending', '2025-04-01'),
(2, 2, 2, 'Battery Replacement', 3200.00, 'In Progress', 'Completed', '2025-04-02');
```

7. Test Queries(Minimum 25 Queries)

-- INSERT, UPDATE, DELETE Queries

-- INSERT: Add a new customer
INSERT INTO Customer (name, contact, email, address)
VALUES ('Neha Singh', '9871234560', 'neha.singh@example.com', 'Vashi, Navi Mumbai');
SELECT*FROM Customer;

customer_id	name	contact	email	address
1	Amit Verma	9123456789	amit.verma@example.com	Sector 21, Gurugram, Haryana
2	Pooja Iyer	9898989898	pooja.iyer@example.com	Koramangala, Bengaluru, Karnataka
3	Neha Singh	9871234560	neha.singh@example.com	Vashi, Navi Mumbai
NULL	NULL	NULL	NULL	NULL

-- UPDATE: Change the status of a service
UPDATE Service_Request
SET status = 'Completed', payment_status = 'Completed'
WHERE service_id = 1;
SELECT * FROM Service_Request WHERE service_id = 1;

	service_id	vehicle_id	specialist_id	worker_id	service_type	cost	status	payment_status	date
▶	1	1	1	1	Clutch Overhaul	8500.00	Completed	Completed	2025-04-01
*	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL

-- DELETE: Remove a vehicle record
DELETE FROM Vehicle
WHERE registration_number = 'DL3CAP1234';
SELECT * FROM Vehicle
WHERE registration_number = 'DL3CAP1234';

vehicle_id	customer_id	model	registration_number	vehicle_type	last_service_date
NULL	NULL	NULL	NULL	NULL	NULL

-- Aggregate Functions
-- Total revenue from completed services
SELECT SUM(cost) AS total_revenue
FROM Service_Request

```
WHERE status = 'Completed';
```

	total_revenue
▶	8500.00

```
-- Average service cost by status
SELECT status, AVG(cost) AS avg_cost
FROM Service_Request
GROUP BY status;
```

status	avg_cost
In Progress	3200.000000

```
-- Number of vehicles per customer
SELECT customer_id, COUNT(*) AS total_vehicles
FROM Vehicle
GROUP BY customer_id;
```

customer_id	total_vehicles
2	1

```
-- Nested (Subqueries)
-- Get customers who own vehicles with service cost > ₹5000
SELECT name
FROM Customer
WHERE customer_id IN (
    SELECT v.customer_id
    FROM Vehicle v
    JOIN Service_Request sr ON v.vehicle_id = sr.vehicle_id
    WHERE sr.cost > 5000
);
```

	name
▶	Amit Verma

```
-- Get the model of the most expensive service
SELECT model
FROM Vehicle
WHERE vehicle_id = (
    SELECT vehicle_id
    FROM Service_Request
    ORDER BY cost DESC
    LIMIT 1
);
```

model
Maruti Suzuki Swift

-- JOIN Types
-- INNER JOIN: Vehicle info with their owners
SELECT v.model, v.registration_number, c.name AS owner_name
FROM Vehicle v
INNER JOIN Customer c ON v.customer_id = c.customer_id;

model	registration_number	owner_name
Maruti Suzuki Swift	DL3CAP1234	Amit Verma
Hyundai Creta	KA01MK5678	Pooja Iyer

-- LEFT JOIN: List all workers with their assigned specialist info (if any)
SELECT w.name AS worker_name, s.name AS specialist_name, s.expertise
FROM Worker w
LEFT JOIN Specialist s ON w.assigned_specialist_id = s.specialist_id;

worker_name	specialist_name	expertise
Ravi Kumar	Sandeep Yadav	Engine Repair
Anjali Mehta	Manish Tiwari	Electrical Work

-- RIGHT JOIN: All specialists and any assigned workers (if any)
SELECT s.name AS specialist_name, w.name AS worker_name
FROM Specialist s
RIGHT JOIN Worker w ON s.specialist_id = w.assigned_specialist_id;

specialist_name	worker_name
Sandeep Yadav	Ravi Kumar
Manish Tiwari	Anjali Mehta

-- FULL OUTER JOIN simulation using UNION (MySQL doesn't directly support FULL OUTER JOIN)
SELECT c.name AS customer_name, v.model
FROM Customer c
LEFT JOIN Vehicle v ON c.customer_id = v.customer_id
UNION
SELECT c.name AS customer_name, v.model
FROM Customer c
RIGHT JOIN Vehicle v ON c.customer_id = v.customer_id;

customer_name	model
Amit Verma	Maruti Suzuki Swift
Pooja Iyer	Hyundai Creta

-- NORMALIZATION—

-- 🔹 1NF: Ensure Atomicity (Already satisfied by schema)

-- Example: No multivalued attributes in Customer table
SELECT * FROM Customer;

customer_id	name	contact	email	address
1	Amit Verma	9123456789	amit.verma@example.com	Sector 21, Gurugram, Haryana
2	Pooja Iyer	9898989898	pooja.iyer@example.com	Koramangala, Bengaluru, Karnataka
NULL	NULL	NULL	NULL	NULL

-- 🔹 2NF: No partial dependency (already satisfied because there are no composite primary keys)
-- Just an example of dependency from Service_Request to Service_Type cost
SELECT service_type, AVG(cost) AS avg_cost
FROM Service_Request
GROUP BY service_type;

service_type	avg_cost
Clutch Overhaul	8500.000000
Battery Replacement	3200.000000

-- 🔹 3NF: Ensure no transitive dependency (each non-key depends only on the key)
-- Verify Specialist expertise depends only on specialist_id
SELECT specialist_id, name, expertise
FROM Specialist;

specialist_id	name	expertise
1	Sandeep Yadav	Engine Repair
2	Manish Tiwari	Electrical Work
NULL	NULL	NULL

-- 🔹 Functional Dependency Check using GROUP BY
-- Each registration_number is unique → should return 1 row per registration_number
SELECT registration_number, COUNT(*) AS duplicate_count
FROM Vehicle
GROUP BY registration_number
HAVING COUNT(*) > 1;

registration_number	duplicate_count

-- 🔹 Query showing dependency of vehicle on customer (vehicle belongs to only one customer)
SELECT v.vehicle_id, v.model, c.name AS customer_name
FROM Vehicle v
JOIN Customer c ON v.customer_id = c.customer_id;

vehicle_id	model	customer_name
1	Maruti Suzuki Swift	Amit Verma
2	Hyundai Creta	Pooja Iyer

-- 🔹 Check: No transitive dependency in Service_Request (e.g., worker_id → specialist_id is not allowed)
-- So, Service_Request has direct specialist_id assigned instead of deriving from worker
SELECT sr.service_id, sr.worker_id, sr.specialist_id, w.assigned_specialist_id
FROM Service_Request sr
JOIN Worker w ON sr.worker_id = w.worker_id
WHERE sr.specialist_id <> w.assigned_specialist_id;

service_id	worker_id	specialist_id	assigned_specialist_id

-- VIEWS--

CREATE VIEW View_Customer_Info AS
SELECT customer_id, name, contact, email
FROM Customer;

-- Example usage:
SELECT * FROM View_Customer_Info;

customer_id	name	contact	email
1	Amit Verma	9123456789	amit.verma@example.com
2	Pooja Iyer	9898989898	pooja.iyer@example.com

CREATE VIEW View_Vehicle_Owners AS
SELECT v.vehicle_id, v.model, v.registration_number, c.name AS owner_name, c.contact
FROM Vehicle v
JOIN Customer c ON v.customer_id = c.customer_id;

-- Example usage:
SELECT * FROM View_Vehicle_Owners;

vehicle_id	model	registration_number	owner_name	contact
1	Maruti Suzuki Swift	DL3CAP1234	Amit Verma	9123456789
2	Hyundai Creta	KA01MK5678	Pooja Iyer	9898989898

CREATE VIEW View_Service_Revenue AS
SELECT status, SUM(cost) AS total_revenue
FROM Service_Request
GROUP BY status;

-- Example usage:
SELECT * FROM View_Service_Revenue;

status	total_revenue
Completed	8500.00
In Progress	3200.00

```
-- First View: Completed and High-Value Services
CREATE VIEW View_Completed_HighValue_Services AS
SELECT sr.service_id, sr.vehicle_id, sr.cost
FROM Service_Request sr
WHERE sr.status = 'Completed' AND sr.cost > 5000;
SELECT * FROM View_Completed_HighValue_Services;
```

service_id	vehicle_id	cost
1	1	8500.00

```
-- Nested View: Join with Vehicle and Customer
CREATE VIEW View_HighValue_Service_Customers AS
SELECT chvs.service_id, chvs.cost, c.name AS customer_name, v.model
FROM View_Completed_HighValue_Services chvs
JOIN Vehicle v ON chvs.vehicle_id = v.vehicle_id
JOIN Customer c ON v.customer_id = c.customer_id;

-- Example usage:
SELECT * FROM View_HighValue_Service_Customers;
```

service_id	cost	customer_name	model
1	8500.00	Amit Verma	Maruti Suzuki Swift

```
-- Generate a bill for a completed service:--
 INSERT INTO Billing (service_id, customer_id, total_amount, payment_method,
payment_status)
SELECT sr.service_id, v.customer_id, sr.cost, 'UPI', 'Completed'
FROM Service_Request sr
JOIN Vehicle v ON sr.vehicle_id = v.vehicle_id
WHERE sr.service_id = 2;
SELECT * FROM Billing WHERE service_id = 2;
```

bill_id	service_id	customer_id	billing_date	total_amount	payment_method	payment_status
1	2	2	2025-04-13 17:31:07	3200.00	UPI	Completed
NULL	NULL	NULL	NULL	NULL	NULL	NULL

```
-- View all generated bills:--
  SELECT b.bill_id, c.name AS customer_name, b.total_amount, b.payment_method,
b.billing_date
FROM Billing b
JOIN Customer c ON b.customer_id = c.customer_id;
```

bill_id	service_id	customer_id	billing_date	total_amount	payment_method	payment_status
1	2	2	2025-04-13 17:31:07	3200.00	UPI	Completed
NULL	NULL	NULL	NULL	NULL	NULL	NULL

```
-- Total revenue collected through billing:
SELECT SUM(total_amount) AS total_revenue_collected
FROM Billing
WHERE payment_status = 'Completed';
```

total_revenue_collected
3200.00

```
-- Procedure to Get Service History of a Vehicle by Registration Number--
DELIMITER //
CREATE PROCEDURE GetVehicleServiceHistory(IN reg_no VARCHAR(20))
BEGIN
   SELECT sr.service_id, sr.service_type, sr.cost, sr.status, sr.payment_status, sr.date
   FROM Service_Request sr
   JOIN Vehicle v ON sr.vehicle_id = v.vehicle_id
   WHERE v.registration_number = reg_no;
END //
DELIMITER ;

-- Example: Call Procedure
CALL GetVehicleServiceHistory('DL3CAP1234');
```

service_id	service_type	cost	status	payment_status	date
1	Clutch Overhaul	8500.00	Completed	Completed	2025-04-01

```
-- TRIGGERS
--  Trigger: Auto-update last_service_date when a new service is marked 'Completed'
DELIMITER //

CREATE TRIGGER UpdateLastServiceDate
AFTER UPDATE ON Service_Request
FOR EACH ROW
BEGIN
   IF NEW.status = 'Completed' AND OLD.status <> 'Completed' THEN
      UPDATE Vehicle
      SET last_service_date = NEW.date
      WHERE vehicle_id = NEW.vehicle_id;
   END IF;
END //

DELIMITER ;
```

```
--  Trigger: Log deleted workers into a separate table --
-- First, create the log table:

CREATE TABLE Deleted_Worker_Log (
   log_id INT AUTO_INCREMENT PRIMARY KEY,
   worker_id INT,
   name VARCHAR(100),
   contact VARCHAR(15),
   deleted_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
Then the trigger:

DELIMITER //

CREATE TRIGGER LogDeletedWorker
BEFORE DELETE ON Worker
FOR EACH ROW
BEGIN
   INSERT INTO Deleted_Worker_Log (worker_id, name, contact)
   VALUES (OLD.worker_id, OLD.name, OLD.contact);
END //

DELIMITER ;
-- Trigger: Restrict adding a service request if the vehicle is already under "In Progress" service

DELIMITER //

CREATE TRIGGER PreventDuplicateService
BEFORE INSERT ON Service_Request
FOR EACH ROW
BEGIN
   DECLARE ongoing_count INT;

   SELECT COUNT(*) INTO ongoing_count
   FROM Service_Request
   WHERE vehicle_id = NEW.vehicle_id AND status = 'In Progress';

   IF ongoing_count > 0 THEN
      SIGNAL SQLSTATE '45000'
      SET MESSAGE_TEXT = 'This vehicle already has a service in progress.';
   END IF;
END //

DELIMITER ;
```

-- CURSOR Example--
--Cursor: Loop through all vehicles and print the vehicle ID and last service date--
--(for demonstration or debug – can be adapted to insert into a summary table)--

```
DELIMITER //

CREATE PROCEDURE PrintVehicleServiceDates()
BEGIN
   DECLARE done INT DEFAULT FALSE;
   DECLARE v_id INT;
   DECLARE v_model VARCHAR(100);
   DECLARE v_service_date DATE;

   DECLARE vehicle_cursor CURSOR FOR
      SELECT vehicle_id, model, last_service_date FROM Vehicle;

   DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = TRUE;

   OPEN vehicle_cursor;

   read_loop: LOOP
      FETCH vehicle_cursor INTO v_id, v_model, v_service_date;

      IF done THEN
         LEAVE read_loop;
      END IF;

      -- Just SELECTing in this example (or can insert into another table/log)
      SELECT CONCAT('Vehicle ID: ', v_id, ', Model: ', v_model, ', Last Service: ',
v_service_date) AS vehicle_info;
   END LOOP;

   CLOSE vehicle_cursor;
END //

DELIMITER ;

-- To run the cursor procedure:
CALL PrintVehicleServiceDates();
```

vehicle_info
Vehicle ID: 2, Model: Hyundai Creta, Last Service: 2025-02-20

8. Conclusion

The Garage Management System was developed to efficiently handle the day-to-day operations of a vehicle service garage. It provides a structured and user-friendly approach to managing service requests, customer details, vehicle information, specialist assignments, worker coordination, and billing processes. By implementing a relational database using MySQL, the system ensures data consistency, integrity, and security.

The integration of advanced SQL features like stored procedures, views, triggers, and cursors not only improves data processing but also enhances automation and reduces manual workload. This system minimizes operational delays and helps garage owners make informed decisions by maintaining a centralized and searchable database of all garage activities.

Overall, the project fulfills its objective of digitizing garage management, increasing efficiency, and improving customer service. In the future, this system can be expanded with features like real-time service tracking, SMS/email notifications, and mobile application integration to make it even more robust and user-centric