VEHICLE RENTAL MANAGEMENT SYSTEM

A DBMS PROJECT REPORT

*Submitted by*

**Ananya Singh (RA2311003010035)**

**Ritisha Katiyar (RA2311003010042)**

**Saanvi Patnaik (RA2311003010057)**

*Under the Guidance of*

DR. R. MANGALAGOWRI

Associate Professor Department of Computing Technologies

*in partial fulfillment of the requirements for the degree of*

BACHELOR OF TECHNOLOGY

in

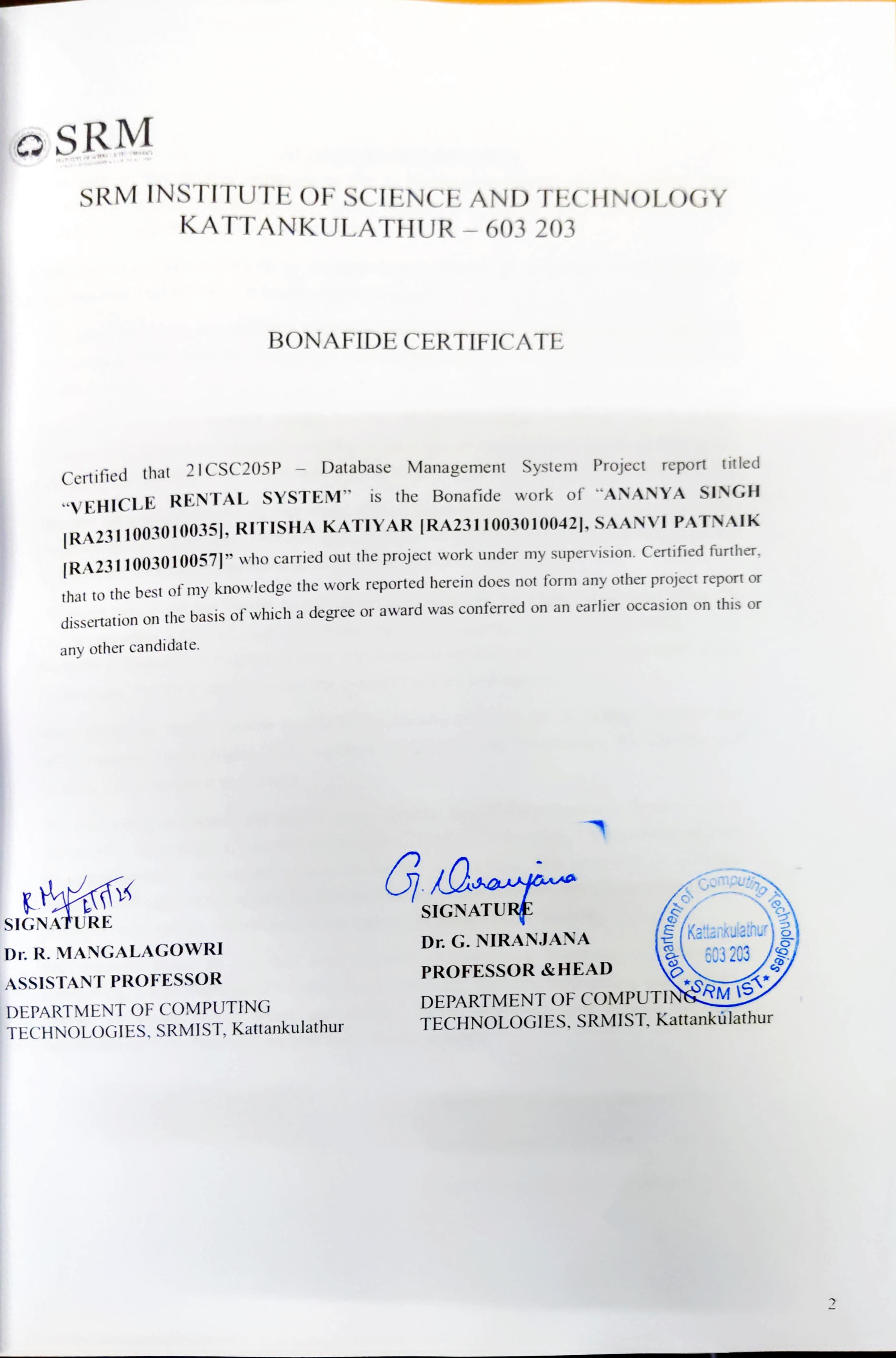
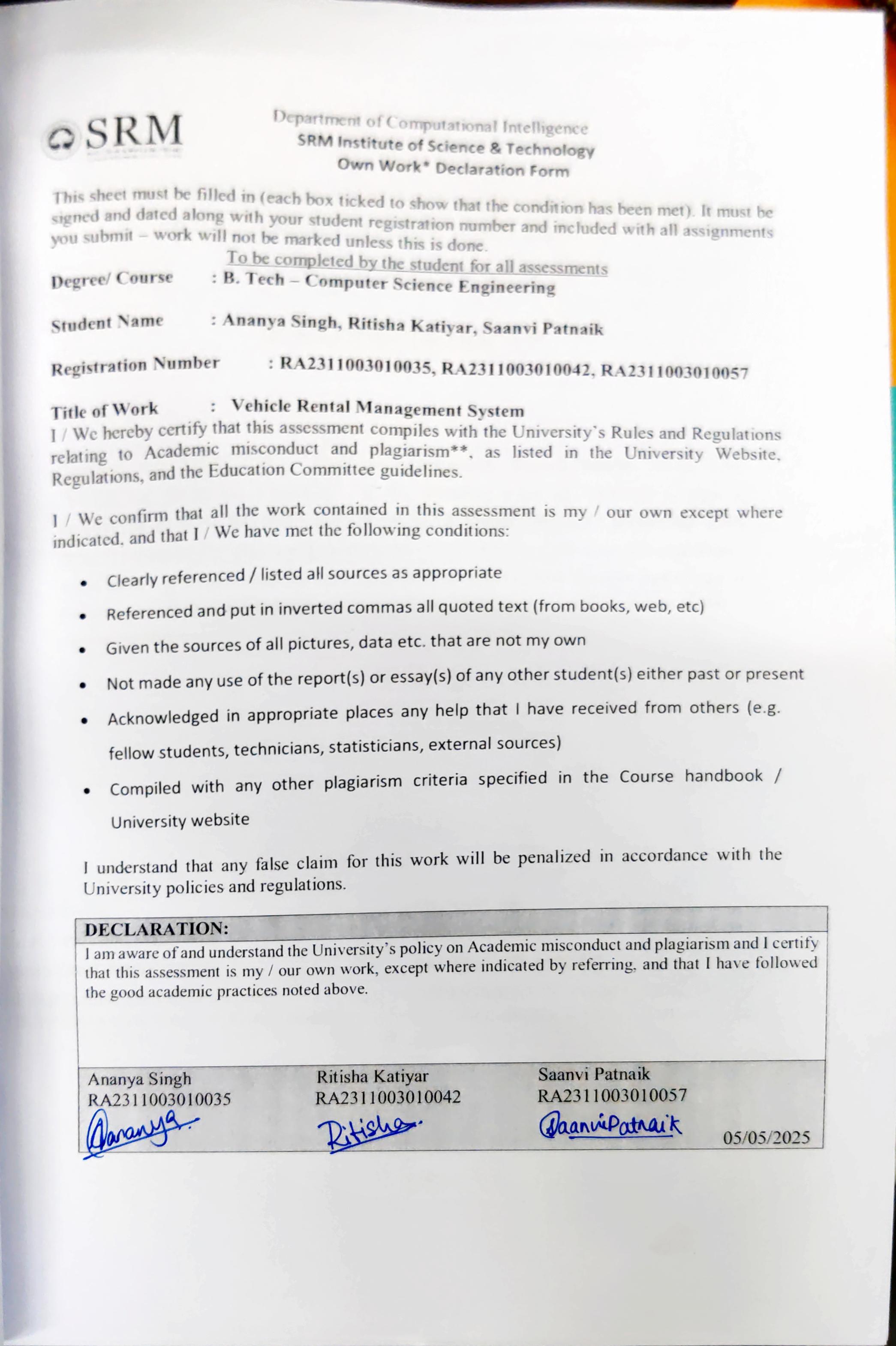
COMPUTER SCIENCE ENGINEERING

DEPARTMENT OF COMPUTING TECHNOLOGIES, COLLEGE OF ENGINEERING AND TECHNOLOGY

SRM INSTITUTE OF SCIENCE ANDTECHNOLOGY

KATTANKULATHUR- 603 203

MAY 2025



# ACKNOWLEDGEMENTS

We express our humble gratitude to **Dr. C. Muthamizhchelvan**, Vice-Chancellor, SRM Institute of Science and Technology, for the facilities extended for the project work and his continued support.

We extend our sincere thanks to **Dr. Leenus Jesu Martin M,** Dean-CET, SRM Institute of Science and Technology, for his invaluable support.

We wish to thank **Dr. Revathi Venkataraman**, Professor and Chairperson, School of Computing, SRM Institute of Science and Technology, for her support throughout the project work.

We encompass our sincere thanks to, **Dr. M. Pushpalatha**, Professor and Associate Chairperson, School of Computing and **Dr. C. Lakshmi,** Professor and Associate Chairperson, School of Computing, SRM Institute of Science and Technology, for their invaluable support.

We are incredibly grateful to our Head of the Department, **Dr. G. Niranjana**, Professor, Department of Computing Technologies, SRM Institute of Science and Technology, for her suggestions and encouragement at all the stages of the project work.

We want to convey our thanks to our DBMS Coordinators, **Dr. Muthu Kumaran A M J** and **Dr. Jayapradha J** Department of Computing Technologies, and our Audit Professor **Dr. Malathy C,** Department of Networking and Communications, SRM Institute of Science and Technology, for their inputs during the project reviews and support.

We register our immeasurable thanks to our Faculty Advisors, **Dr. G. Ramya**, Department of Computing Technologies, SRM Institute of Science and Technology, for leading and helping us to complete our course.

Our inexpressible respect and thanks to our faculty, **Dr. R. Mangalagowri,** Department of Computing Technologies, S.R.M Institute of Science and Technology, for providing us with an opportunity to pursue our project under her mentorship. She provided us with the freedom and support to explore the research topics of our interest. Her passion for solving problems and making a difference in the world has always been inspiring.

We sincerely thank all the staff and students of Computing Technologies, School of Computing, S.R.M Institute of Science and Technology, for their help during our project. Finally, we would like to thank our parents, family members, and friends for their unconditional love, constant support and encouragement.

Ananya Singh [RA2311003010035]

Ritisha Katiyar [RA2311003010042]

Saanvi Patnaik [RA2311003010057]

**TABLE OF CONTENTS**

**ABSTRACT**  **iv**

**LIST OF FIGURES vi**

**LIST OF TABLES [ if required] vii**

**ABBREVIATIONS viii**

**CHAPTER NO. TITLE PAGE NO.**

**1. INTRODUCTION** 9

1.1 Introduction 9

1.2 Objectives 10

1.3 System Requirements 11

**2. MODULES** 12

2.1 Module 1: Vehicle Management 12

2.2 Module 2: Booking Management 17

2.3 Module 3: User Management 18

2.4 Module 4: Viewing Available Cars 20

**3. SYSTEM ARCHITECTURE** 22

3.1 ER Diagram 22

3.2 Related Schema 24

3.3 SQL Queries 28

3.4 Normalization 87

3.5 Transaction & Concurrency Control 91

**4. RESULTS** 96

**5. CONCLUSION & FUTURE ENHANCEMENTS** 101

**APPENDIX I: SAMPLE SOURCE CODE** 103

### ABSTRACT

The Vehicle Rental System is a comprehensive, user-centric platform designed to streamline the process of renting various types of vehicles, including cars, motorcycles, bicycles, and commercial vehicles. The system caters to both customers and administrators, providing a seamless interface for managing vehicle availability, bookings, payments, and maintenance schedules. It aims to enhance customer convenience by offering online reservations, real-time vehicle tracking, and flexible rental options tailored to individual needs. The core of the system is a centralized database that maintains detailed records of vehicles, customers, transactions, and rental histories. Users can create accounts, browse available vehicles based on categories, and make instant bookings with transparent pricing and rental terms. Advanced search and filter functionalities allow customers to select vehicles based on parameters such as model, price, availability, and location. The system supports multiple payment gateways, ensuring secure and versatile transaction options. From an administrative perspective, the Vehicle Rental System provides tools for inventory management, scheduling regular vehicle maintenance, handling customer feedback, and generating financial and operational reports. It also incorporates a notification system to alert administrators and customers about upcoming bookings, due returns, and promotional offers. To improve operational efficiency, the system includes features like automated invoicing, GPS-based vehicle tracking, and digital contract signing. Security and data privacy are prioritized through encryption and access controls, ensuring that user data and payment information remain protected. Scalability is another key aspect, allowing the system to handle increasing numbers of users and vehicles without compromising performance. The system is designed to be accessible across multiple platforms, including web and mobile devices, providing users with flexibility and ease of use.

# 

# LIST OF FIGURES

**CHAPTER NO TITLE PAGE NO.**

|  |  |  |
| --- | --- | --- |
| **2** | vehicle Management | **14** |
| **2** | Booking management | **16** |
| **2** | Admin Login | **18** |
| **2** | Admin Sign up | **19** |
| **2** | View available cars | **21** |
| **2** | booking management | **23** |
| **3** | ER Diagram | **22** |
| **3** | Relational Schema | **24** |
| **3** | Relational Schema | **27** |
| **3** | Table Creation | **28** |
| **3** | Inserting Data | **29** |
| **3** | Unormalised Table | **31** |
| **3** | 1NF Form | **32** |
| **3** | 2NF Form | **33** |
| **3** | Normalisation | **34** |
| **4** | All Tables | **41** |
| **4** | All cars in the Inventory | **41** |
| **4** | Available Cars for Rent | **42** |
| **4** | Rent a Car | **42** |
| **4** | Return a Rented Car | **43** |

# ABBREVIATIONS

VRN – Vehicle Registration Number

VIN – Vehicle Identification Number

ODO – Odometer Reading

DLN – Driver’s License Number

DLI – Driver’s License Issuer

RNT – Rental Number

RCD – Rental Contract Date

RET – Return Date

INS – Insurance Number

TPA – Third Party Administrator

FLS – Fuel Level Start

FLE – Fuel Level End

MIL – Mileage

BKP – Booking Payment

DEP – Deposit

LOC – Location Code

STS – Rental Status

CAT – Vehicle Category (e.g., SUV, Sedan)

**CHAPTER 1: INTRODUCTION**

* 1. **Introduction**

A vehicle rental system is a comprehensive software platform designed to manage and streamline the process of renting vehicles to customers, providing an organized and efficient way for both rental companies and customers to handle bookings, payments, and vehicle tracking. The system allows customers to browse available vehicles, make reservations, specify rental periods, and complete payments, often through a user-friendly online portal or mobile application. For rental companies, the system offers powerful tools for managing vehicle inventories, scheduling, maintenance records, and customer databases, ensuring that each vehicle is properly maintained and available when needed. The vehicle rental system also integrates features such as automated billing and invoicing, GPS tracking, fuel management, insurance processing, and reporting, helping businesses monitor their fleet and optimize operations. Additionally, it typically supports functionalities like driver verification, license validation, and dynamic pricing based on rental duration, vehicle type, or demand. Security is a key component, ensuring sensitive customer and payment data are protected through encryption and compliance with data protection regulations. Advanced vehicle rental systems may include analytics to forecast demand, track performance metrics, and enhance customer service through automated notifications and feedback collection. The system reduces manual work, minimizes errors, and speeds up transaction times, making the rental process more convenient and transparent. It benefits customers by providing flexibility and easy access to vehicles for personal or business use, whether for short-term needs or long-term leasing. For the rental company, it ensures better asset management, higher customer satisfaction, and improved profitability. In today’s fast-paced and competitive market, such a system is essential for vehicle rental businesses to stay efficient, scalable, and responsive to customer needs. Furthermore, as environmental concerns grow, many vehicle rental systems are expanding to include electric vehicles (EVs) and integrating sustainability tracking features to monitor emissions and promote eco-friendly travel options. Overall, a vehicle rental system serves as the backbone of a modern rental business, linking customers and companies through a seamless digital platform that enhances the overall rental experience, fosters loyalty, and supports business growth.

* 1. **Objectives**

The primary objective of Project is to design and develop a fully functional, secure, and scalable platform that automates the key operations of a car rental service. Traditional paper-based or spreadsheet-driven methods are often error-prone, time- consuming, and inefficient in handling increasing customer demands and large volumes of data. This project aims to eliminate such limitations by leveraging a structured relational database model using MySQL, enabling centralized data management, real-time updates, and enhanced operational control. By introducing system automation and enforcing role-based access, the solution ensures smooth interactions between customers, staff, and administrators. The system is not only tailored to fulfill current business requirements but also lays the groundwork for advanced future features such as online payments, mobile support, and data analytics. The following objectives outline the specific goals that guided the development of the system:

• Streamline Booking and Reservation Process: To provide customers with a simple, user-friendly platform for booking vehicles, reducing time spent on manual reservations and ensuring accurate availability of vehicles.

• Efficient Fleet Management: To enable rental companies to effectively manage their vehicle inventory, including tracking the status, maintenance schedules, and location of each vehicle in real-time.

• Enhance Customer Experience: To offer personalized services such as easy booking, flexible rental periods, transparent pricing, and convenient payment options, ensuring a seamless experience for users.

• Automated Billing and Payments: To simplify the payment process by automating billing, invoicing, and integrating multiple payment gateways, ensuring accurate and timely financial transactions.

• Improve Vehicle Utilization: To maximize the utilization of rental vehicles by managing demand, optimizing vehicle allocation, and scheduling rentals efficiently.

• Track Vehicle Usage and Maintenance: To monitor vehicle usage, ensure regular maintenance, and keep detailed logs of each vehicle’s mileage and condition, promoting better upkeep and longevity of the fleet.

• Enhance Security and Compliance: To implement security measures such as secure user authentication, driver verification, and compliance with legal and insurance requirements, reducing risks for both the company and customers.

• Optimize Fleet Cost and Revenue Management: To track costs associated with vehicle acquisition, maintenance, and operation, while also maximizing revenue through dynamic pricing, discounts, and promotions.

• Data-Driven Insights and Reporting: To generate reports and analytics on rental trends, customer preferences, fleet performance, and financial metrics, helping the business make informed decisions and identify growth opportunities

## System Requirements

### Hardware Requirements

* Processor: Intel Core i3 or higher
* RAM: Minimum 4 GB (8 GB recommended)
* Storage: At least 500 MB free disk space
* Display: 1024×768 resolution or higher
* Network: Required for server access or multi-user operation

### Software Requirements

* Operating System: Windows 10/11, Linux, or macOS
* Local Server Stack: XAMPP (Apache, MySQL, PHP)
* Database: MySQL Server with phpMyAdmin
* Code Editor: Visual Studio Code or equivalent
* Web Browser: Chrome, Firefox, or Edge (latest version)
* Optional Tools: MySQL Workbench for database modeling and ER diagrams

# CHAPTER 2: MODULES

### Module 1: Vehicle Management

The Vehicle Management Module is responsible for managing the core asset of the car rental business: its fleet of vehicles. This module provides the functionalities to add, update, delete, and view vehicle information, ensuring that the database accurately reflects the current inventory.

#### Functions:

* + **Add New Vehicles:**
    - Allows administrators to add new vehicles to the system.
    - Captures essential vehicle details, including:
      * Vehicle Title/Name
      * Brand (references tbl\_brands)
      * Description/Overview
      * Price Per Day
      * Fuel Type
      * Model Year
      * Seating Capacity
      * Vehicle Images (multiple images - Vimage1 to Vimage5)
      * Features (e.g., Air Conditioner, Power Door Locks - boolean flags)
    - Inserts data into the tbl\_vehicles table.
    - Validates data inputs to ensure accuracy and completeness (e.g., ensuring the brand exists, price is a positive number, year is a valid year).

#### Update Vehicle Information:

* + - Enables administrators to modify existing vehicle information.
    - Provides a user interface to edit vehicle details.
    - Updates the corresponding record in the tbl\_vehicles table.
    - Handles updates to any of the vehicle attributes mentioned above.
    - Ensures that updates are applied correctly and that data integrity is maintained.

#### Delete Vehicles:

* + - Allows administrators to remove vehicles from the system.
    - Provides a mechanism to select the vehicle to be deleted.
    - Deletes the corresponding record from the tbl\_vehicles table.
    - May include a confirmation step to prevent accidental deletions.
    - Considers the implications of deleting a vehicle that may be associated with existing or past bookings in the tblbooking table (e.g., may need to handle these bookings or prevent deletion if there are active rentals).

#### View Available Vehicles:

* + - Provides functionality to display a list of vehicles that are currently available for rent.
    - This may involve filtering vehicles based on their availability status (which may be derived from the tblbooking table).
    - Presents vehicle information in a user-friendly format, including relevant details like brand, model, price, and availability.
    - May support sorting and filtering of vehicles based on various criteria (e.g., price, brand, vehicle type).

#### Manage Vehicle Features:

* + - Allows administrators to manage the features offered by each vehicle.
    - This involves setting boolean flags for features like Air Conditioner, Power Door Locks, etc., in the tbl\_vehicles table.
    - Provides a user interface to easily enable or disable these features for each vehicle.

#### Interactions with Database:

* + The module primarily interacts with the tbl\_vehicles table.
  + It also reads from the tbl\_brands table to get the list of available brands when adding or updating vehicles.
  + Potentially interacts with the tblbooking table to determine vehicle availability and to handle deletion of vehicles with existing bookings.

#### Business Rules:

* + Requires that a vehicle must be associated with a valid brand from the tbl\_brands table.
  + Ensures that prices and other numerical values are within acceptable ranges.
  + May implement rules to prevent the deletion of vehicles that are currently rented.
  + Maintains data consistency between tbl\_vehicles and other related tables.



Fig 2.1.1. vehicle Management

### Module 2: Booking Management

#### The Booking Management Module handles the core functionality related to vehicle reservations. It allows users to initiate, manage, and track bookings, while also ensuring that vehicles are available and not double-booked during the selected rental period.

#### Functions:

* + **Create Booking:**
    - Allows users to select a vehicle and book it for a specific duration.
    - Captures essential booking details, including:
    - Inserts the booking into the tblbooking table.
    - Validates availability of the vehicle for the chosen date range before confirming booking.

#### Modify Booking Dates:

* + - Allows users or admins to change the rental start or end dates.
    - Rechecks vehicle availability for the new date range.
    - Updates tblbooking with the revised schedule and recalculates rental cost..

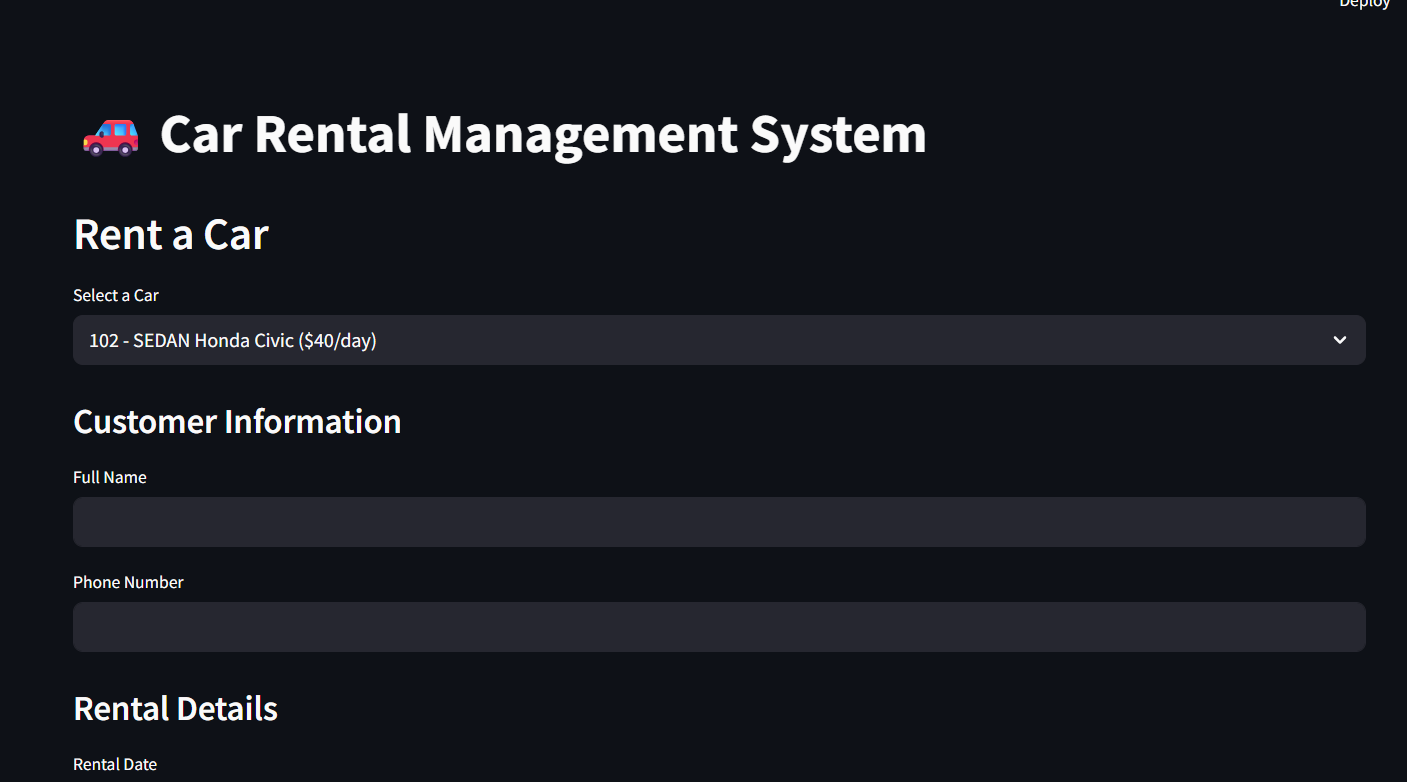
#### Interactions with Database:

#### Reads and writes to the tblbooking table.

#### References tblusers for customer details.

#### References tbl\_vehicles to link bookings with specific vehicles.

#### May read from tblpayment or similar (if implemented) for booking-related payment tracking



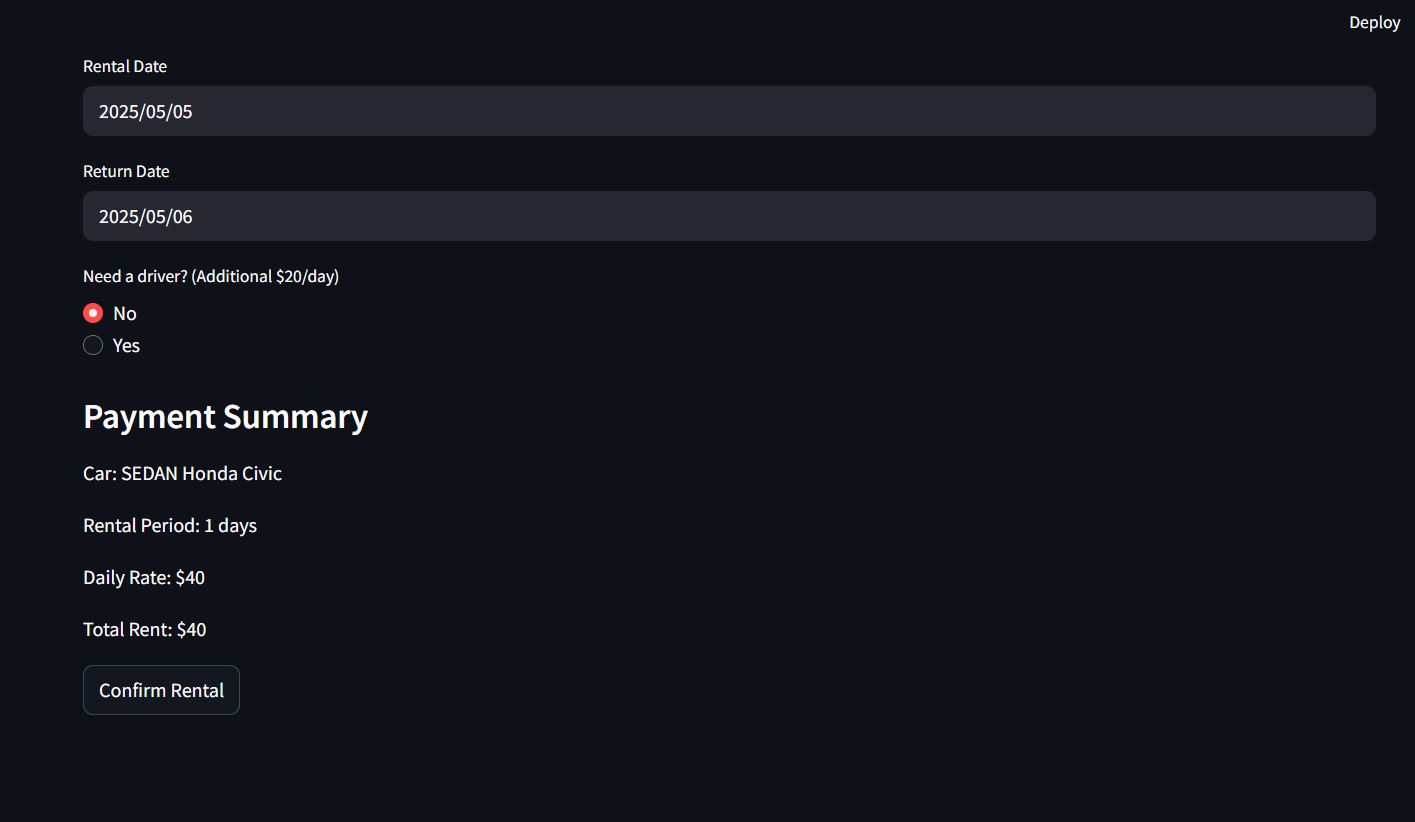


Fig 2.2.1. Booking management

### Module 3: User Management

The User Management Module is responsible for managing customer information within the car rental system. This includes handling user registration, authentication, profile management, and password management. The module ensures secure and efficient management of user data, which is crucial for the overall functionality of the system.

* Functions:

#### User Registration and Account Creation:

* + - Allows new users to register and create accounts in the system.
    - Captures user details such as full name, email address, contact number, date of birth, address, city, and country.
    - Stores user credentials (email and password) securely in the tbl\_users table.
    - Validates user input to ensure data accuracy and completeness.
    - May send a confirmation email to the user upon successful registration.

#### User Login and Authentication:

* + - Enables registered users to log in to the system using their credentials (email and password).
    - Authenticates users against the data stored in the tbl\_users table.
    - Manages user sessions to maintain login status.
    - Implements security measures to protect against unauthorized access.

#### User Profile Management (View, Edit):

* + - Allows users to view and edit their profile information.
    - Retrieves user details from the tbl\_users table for display.
    - Enables users to update their personal information, such as address, contact number, etc.
    - Ensures that updated information is saved correctly in the tbl\_users table.

#### Password Management (Reset, Change):

* + - Provides functionality for users to manage their passwords.
    - Allows users to change their existing passwords.
    - Implements password reset functionality for users who have forgotten their passwords.
    - Ensures that new passwords meet the defined complexity requirements.
    - Handles password encryption and storage securely.

#### Interactions with Database:

* + The module primarily interacts with the tbl\_users table.

#### Business Rules:

* + Ensures that email addresses are unique to prevent duplicate accounts.
  + Enforces password complexity requirements (e.g., minimum length, special characters) to enhance security.
  + Handles user authentication and authorization to control access to different parts of the system.

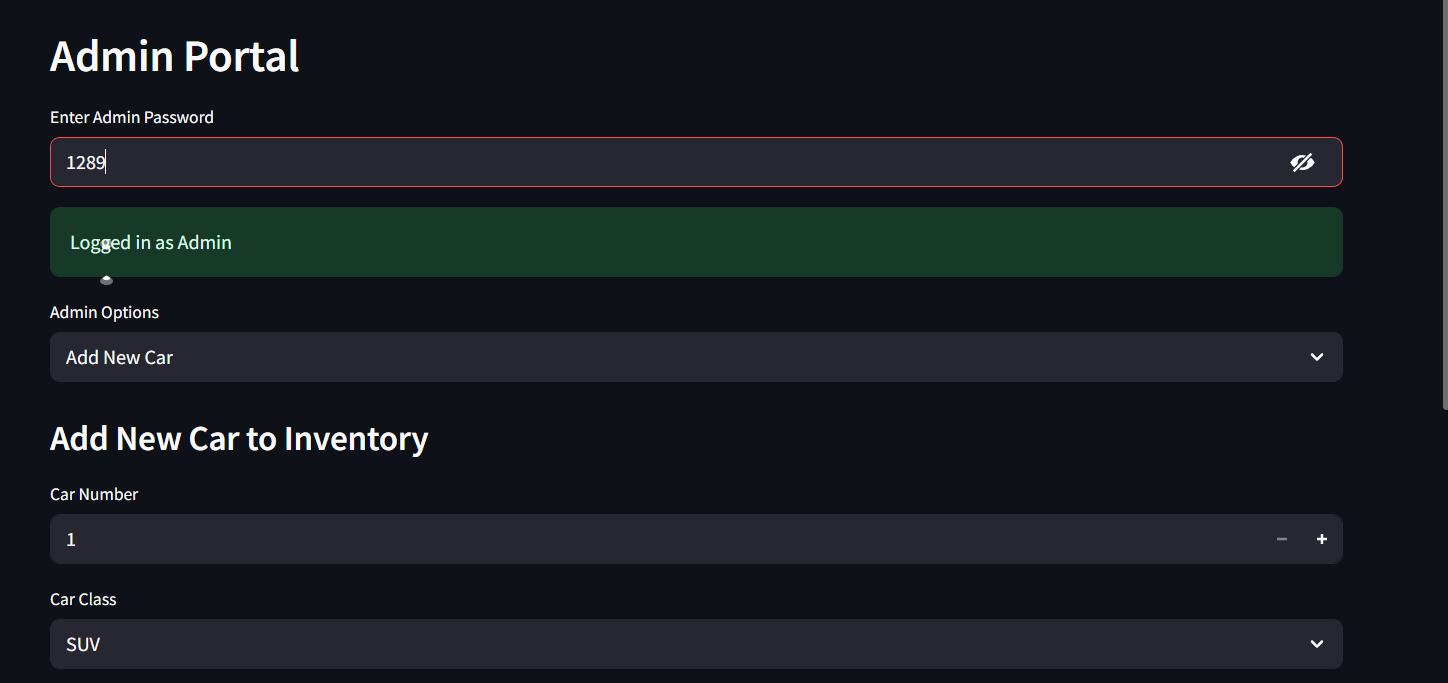


Fig 2.3.1 Admin Login

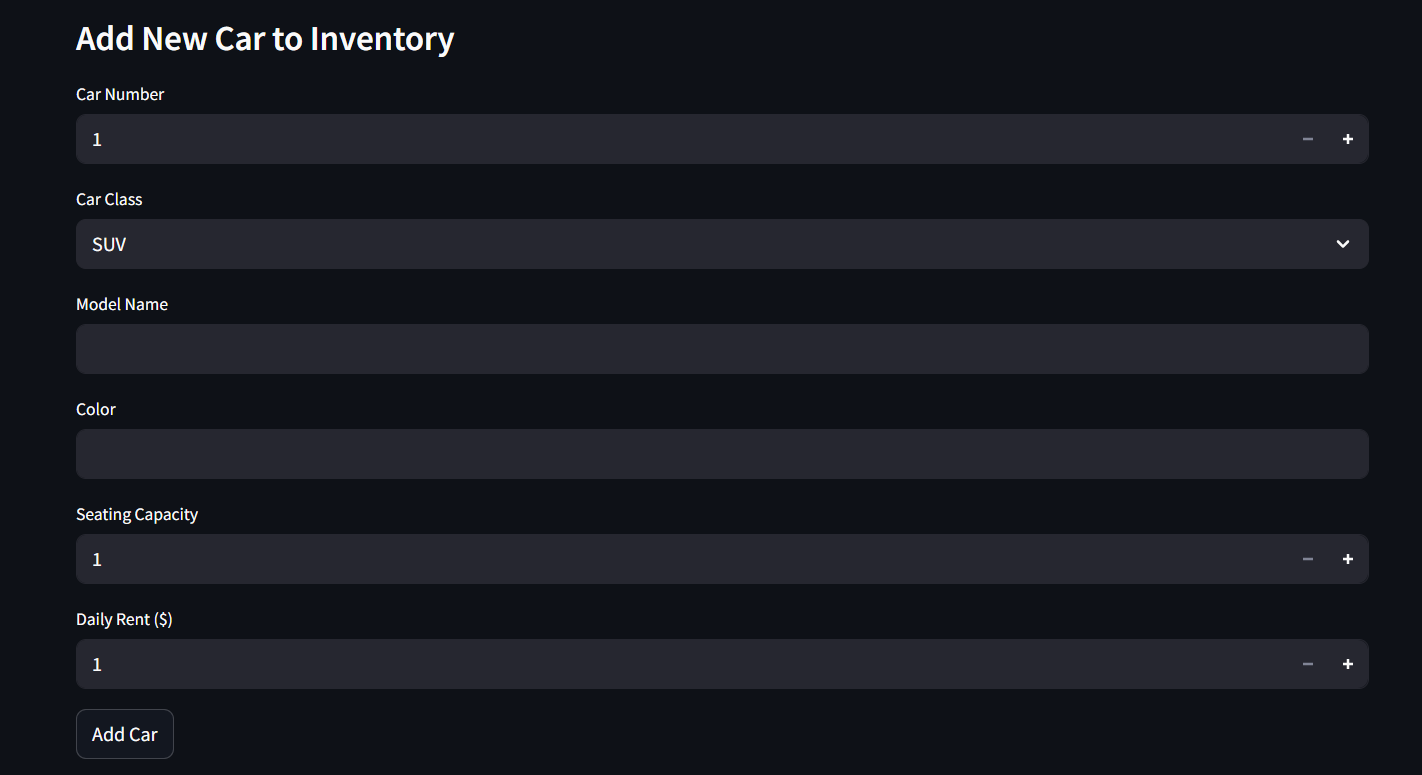


Fig 2.3.2 Admin Sign up

**2.4. Module 4: View Available Cars**

The Booking Management Module handles vehicle reservations.

Key functions include:

### Functions:

* Show Available Cars:
* Filters out cars that are currently rented (based on tblbooking records).
* Displays real-time availability based on booking start and end dates.

### Search & Filter:

### Users can search by vehicle type, price, features.

### Allows filtering based on availability date ranges.

### Interaction with Database:

* Reads from tbl\_vehicles and tblbooking.
* Performs date comparisons to determine current availability.

### Business Rules:

### A vehicle is available only if no active or overlapping booking exists for the current date.

### Rented vehicles should be hidden from this view.



Fig 2.4.1 View available cars

# CHAPTER 3: SYSTEM ARCHITECTURE

A diagram of a company

AI-generated content may be incorrect.**3.1 ER Diagram**

Fig 3.1. ER Diagram

The ER diagram consists of multiple entities and their interconnections, which are described below:

**3.1.1 Entities and Attributes**

* Employee (Orange Box): Manages vehicle registrations and has attributes like Employee ID, Name, Salary, Contact Details, and Responsibilities.
* Customer (Blue Box): Represents the users who rent vehicles, with attributes such as Customer ID, Name, Contact Details, Address, and Driving License Information.
* Vehicle (Green Box): The core asset in the rental system, with attributes like Vehicle ID, Model, Condition, Daily Price, Plate Number, and Mileage.
* Reservation (Red Box): Captures booking details, including Reservation ID, Pickup Date, Return Date, Number of Days, and Cancellation Details.
* Rent (Yellow Box): Stores rental transaction details like Rent ID, Payment Method, Refund, Down Payment, and Damage Compensation.

**3.1.2 Relationships Between Entities**

* An Employee registers Vehicles and is responsible for their management.
* A Customer makes a Reservation for a Vehicle, specifying rental duration and location.
* A Vehicle is chosen for Reservation based on availability.
* The Rent transaction is linked to a Customer and a Vehicle, involving payment and potential compensation for damages.

A Customer pays for the Rent, which includes attributes like Payment Date and Payment Method

A screenshot of a computer

AI-generated content may be incorrect.**3.2. Relational Schema**

Fig 3.2.1. Relational Schema

**3.2.1 EMPLOYEES**

This table stores information about the employees managing the system.

* EmpID (Primary Key) – Unique identifier for each employee.
* FName, LName – First and last names of the employee.
* HouseNo, City, Country, Address, ContactNo – Employee's residential details and contact number.
* Salary – Monthly salary of the employee.
* JoinedDate – Date when the employee joined.
* Responsibility – Role or responsibilities of the employee in the system.

Relationships:

* An employee registers vehicles into the system.
* An employee manages vehicle-related records.

**3.2.2 VEHICLE**

This table stores details of the vehicles available for rental.

* VehicleID (Primary Key) – Unique identifier for each vehicle.
* Mileage – Distance the vehicle has traveled.
* PlateNo – Vehicle's registration number.
* DailyPrice – Rental price per day.
* Condition – Vehicle's current condition (e.g., "Good", "Needs Maintenance").
* Model – The make and model of the vehicle.

Relationships:

* A vehicle is registered by an employee.
* A vehicle is chosen for a reservation.
* A vehicle is involved in a rental transaction.

**3.2.3 CUSTOMER**

This table stores customer details.

* CustID (Primary Key) – Unique identifier for each customer.
* FName, LName – First and last names of the customer.
* HouseNo, City, Country, Address, ContactNo – Customer’s residential details and contact information.
* DrivingLicence – Customer’s driving license number.

Relationships:

* A customer registers in the system to make reservations.
* A customer makes reservations for vehicles.
* A customer pays for rental transactions.

**3.2.4 RESERVATION**

This table stores details about vehicle reservations made by customers.

* ReserveID (Primary Key) – Unique identifier for each reservation.
* ReserveDate – The date on which the reservation was made.
* ReturnDate – The date when the vehicle is expected to be returned.
* NoOfDays – Duration of the reservation.
* PickupDate – The date when the vehicle is picked up.
* PickupLocation – Location from where the vehicle is picked up.
* CancelationDetails – Information about reservation cancellations (if any).

Relationships:

* A customer makes a reservation.
* A vehicle is chosen for a reservation.

**3.2.5 RENT**

This table stores payment and rental transaction details.

* RentID (Primary Key) – Unique identifier for each rent transaction.
* PayDate – Date when payment was made.
* TotalPay – The total amount paid by the customer.
* PayMethod – Mode of payment (e.g., Cash, Credit Card, UPI).
* DownPay – Advance payment made before renting the vehicle.
* Refund – Amount refunded in case of cancellation.
* DamageCompensation – Additional charges for vehicle damage.

Relationships:

* A customer pays for the rent.

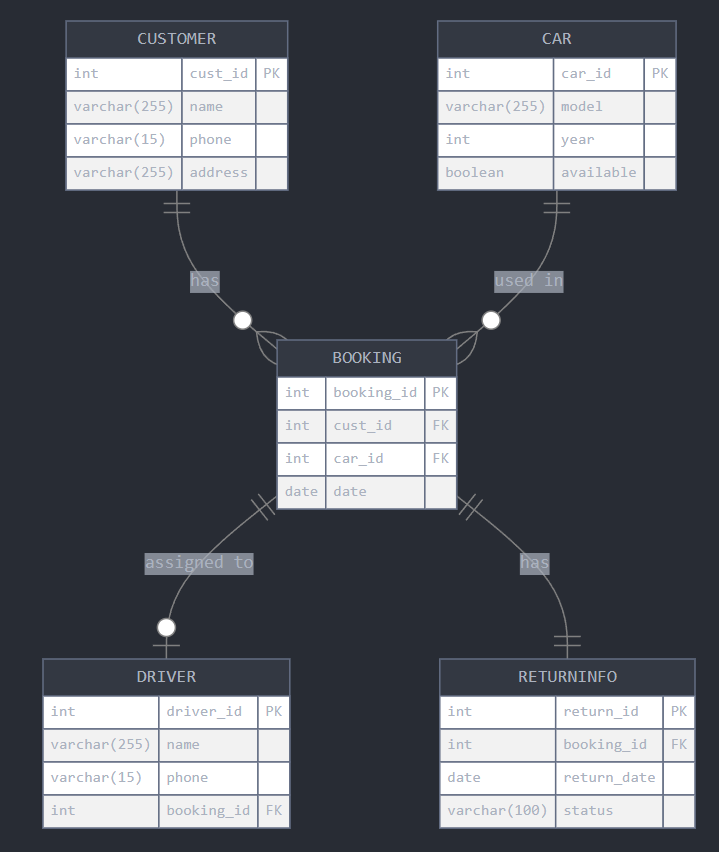


Fig 3.2.2. Relational Schema

The diagram shows:

1. **Entities** with their attributes:
   * Customer (customer\_id, name, contact\_info, address)
   * Booking (booking\_id, customer\_id, start\_date, end\_date, total\_cost, status)
   * Car (car\_id, booking\_id, model, license\_plate, status)
   * Driver (driver\_id, booking\_id, name, license\_number, contact\_info)
   * Return (return\_id, booking\_id, return\_date, condition\_notes, additional\_charges)
2. **Relationships**:
   * One Customer can make many Bookings (one-to-many)
   * One Booking can involve many Cars (one-to-many)
   * One Booking may have one optional Driver (one-to-one optional)
   * One Booking has exactly one Return record (one-to-one)

## SQL Queries

## Create Database

## create database car\_rental;

## use car\_rental;

### Create Tables

### Cars:

### Admin

### Rentings

### Inserting values

### 

Fig 3.3.1. Table Creation

### 

Fig 3.3.2. Inserting Data

### Search Available Vehicles by Date and Type

SELECT \* FROM Vehicles

WHERE Vehicle\_Type = 'SUV'

AND Current\_Status = 'Available'

AND Vehicle\_ID NOT IN (

SELECT Vehicle\_ID FROM Reservations

WHERE ('2025-06-01' BETWEEN Start\_Date AND End\_Date)

OR ('2025-06-05' BETWEEN Start\_Date AND End\_Date)

);

3.3.3 Make a New Reservation

INSERT INTO Reservations (Customer\_ID, Vehicle\_ID, Start\_Date, End\_Date, Pickup\_Location, Dropoff\_Location, Total\_Price, Booking\_Status)

VALUES (1, 5, '2025-06-01', '2025-06-05', 'Downtown Branch', 'Downtown Branch', 300.00, 'Confirmed');

### 

### Process a Payment

INSERT INTO Payments (Reservation\_ID, Payment\_Date, Amount, Payment\_Method, Transaction\_Status)

VALUES (10, NOW(), 300.00, 'Credit Card', 'Successful');

### View Customer Booking History

SELECT R.Reservation\_ID, V.Make, V.Model, R.Start\_Date, R.End\_Date, R.Total\_Price, R.Booking\_Status

FROM Reservations R

JOIN Vehicles V ON R.Vehicle\_ID = V.Vehicle\_ID

WHERE R.Customer\_ID = 1;

### List All Active Reservations

SELECT R.Reservation\_ID, C.Full\_Name, V.Make, V.Model, R.Start\_Date, R.End\_Date

FROM Reservations R

JOIN Customers C ON R.Customer\_ID = C.Customer\_ID

JOIN Vehicles V ON R.Vehicle\_ID = V.Vehicle\_ID

WHERE R.Booking\_Status = 'Confirmed'

AND R.End\_Date >= CURDATE();

### Cancel a Reservation and Update Vehicle Status

* Update the reservation status

UPDATE Reservations

SET Booking\_Status = 'Cancelled'

WHERE Reservation\_ID = 10;

* Update the vehicle status

UPDATE Vehicles

SET Current\_Status = 'Available'

WHERE Vehicle\_ID = (SELECT Vehicle\_ID FROM Reservations WHERE Reservation\_ID = 10);

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Ex.No.** | **Name of the Exercise** | **Page No.** | **Date of Completion** | **Faculty Sign.** |
| 1. | Creating Database Table | 05 |  |  |
| 2. | Working with Data Manipulation commands | 08 |  |  |
| 3. | Integrity Constraints | 13 |  |  |
| 4. | Basic Select Statements | 17 |  |  |
| 5. | SQL Functions | 21 |  |  |
| 6. | Joining Tables | 28 |  |  |
| 7. | Sub Queries | 31 |  |  |
| 8. | Views | 35 |  |  |
| 9. | Advanced Select Statements | 38 |  |  |
| 10. | Additional Queries | 41 |  |  |
| 11. | Basic of PL/SQL | 60 |  |  |

Appendix – A : Codd’s Rules

Appendix – B : Answer for Additional Queries Bibliography

Index

**I N T R O D U C T I O N TO S Q L**

# SQL

* Structured Query Language (SQL) is a language that all commercial RDBMS implementations understand.
* SQL is a non-procedural language.
* SQL is a Unified Language. It provides statements for a variety of tasks.
* Developed in a prototype relational database management **System R** by IBM in mid 1970s and Oracle Corporation introduced the first commercial SQL in 1979.

**SQL Vs SQL \*PLUS**

|  |  |
| --- | --- |
| ***SQL*** | ***SQL \*PLUS*** |
| * A language * ANSI standard * Keyword cannot be abbreviated * Statements manipulate data and table definitions in the database * SQL statements are stored in buffer | * An environment * Oracle proprietary * Keywords can be abbreviated * Commands do not allow manipulation of values in the database * SQL\*PLUS statements are not stored in Buffer |

**Oracle RDBMS Limits**

|  |  |
| --- | --- |
| ***Item*** | ***Limit*** |
| Tables in a database | No limit |
| Rows in a table | No limit |
| Columns in a table | 254 |
| Characters in a row | 1,26,495 |
| Characters in character field | 240 |
| Digits in a number field | 1057 |
| Significant digits in a number filed | 40 |
| Ranges of values in a date field | 1-JAN-4712 BC *to* 31-DEC-4712 AD |
| Indexes on a table | No limit |
| Tables or views joined in a query | No limit |
| Levels of nested sub queries | 255 |
| Characters in name | 30 |

**Naming Rules**

* Begin with a letter and followed by zero or more of characters A-Z, 0-9, \_, $, #
* Not case sensitive

## Data types

* Integers Decimal numbers--- NUMBER, INTEGER.

Number is an oracle data type. Integer is an ANSI data type. The syntax for NUMBER is NUMBER(P, S) p is the precision and s is the scale. P can range from 1 to 38 and s from -84 to +127

* Floating point numbers FLOAT
* Fixed length character strings CHAR (len) Fixed length character data of length len bytes.
* Variable length character strings Varchar2(len)

Variable length character string having maximum length *len* bytes.

* Dates DATE
* Character data of variable size up to 32760 characters LONG
* Raw binary data, size bytes long. Maximum size is 32767 bytes RAW (size)

## Constants/Literals

* ANSI standard defines format for literals
* Numeric: 21, -32, $0.75,1.2E4
* String: enclosed within single quote
* Date Format: 12-mar-03

## Operators

* Arithmetic operators like +, -, \*, /
* Logical operators: AND, OR
* Relational operators: =, <=,>=, < >
* The Arithmetic operators are used to calculate something like given in the example below:

Select \* from employee where sal \* 1.1 > 1000;

* The logical operators are used to combine conditions like: Select \* from employee where (sal > 1000 AND age > 25);
* The above two examples also illustrate use of relational operators

# NULL

* Missing/unknown/inapplicable data represented as a **null** value

## Statements

* SQL has three flavours of statements.

*DDL is Data Definition Language statements. Some examples:*

* CREATE - to create objects in the database
* ALTER - alters the structure of the database
* DROP - delete objects from the database
* TRUNCATE - remove all records from a table
* COMMENT - add comments to the data dictionary

*DML is Data Manipulation Language statements. Some examples:*

* SELECT - retrieve data from a database
* INSERT - insert data into a table
* UPDATE - updates existing data within a table
* DELETE - deletes all records from a table, the space for the records remains
* CALL - call a PL/SQL or Java subprogram
* EXPLAIN PLAN - explain access path to data
* LOCK TABLE - control concurrency

*TCL (Transaction Control Language) is a DML*

* COMMIT - save work done
* SAVEPOINT - identify a point in a transaction to which you can later roll back
* ROLLBACK - restore database to original since the last COMMIT
* SET TRANSACTION - Change transaction options like what rollback segment to use

*DCL is Data Control Language statements. Some examples:*

* GRANT - gives user's access privileges to database
* REVOKE - withdraw access privileges given with the GRANT command

## Database Objects

* Table -- Basic unit of storage; composed of rows and columns
* View -- Logically represents subsets of data from one or more tables
* Sequence -- Generates primary key values
* Index -- Improves the performance of some queries
* Synonym -- Gives alternative names to objects

# SQL \*PLUS EDITING COMMANDS

EDIT filename Invokes text editor

L  List lines of the current buffer from first to last

C/old/new  Change *old* to *new* in the current line of the current buffer. (If *old* is prefixed with ‘…’ then it matches everything up to including the first

occurrence of *old*. If *old* is suffixed with ‘…’ then it matches everything including and after the first occurrence of *old*.)

A *text*  Add *text* to the end of current line in current buffer. DEL  Delete the current line in current buffer.

SAVE *name* CREATE/ REPLACE/ APPEND  Save the content of current buffer in to the file *name*. If filetype is missing, then CREATE is the default.

@*filename*  Run the file *filename* which contains SQL statements. The file may contain any commands that may be interactively.

RUN  Displays and run the command in the buffer

/  Run the command in the buffer without displaying.

START *file*  Run the *file*. The file may contain any commands that may not be interactively.

\* \* \*

|  |  |  |
| --- | --- | --- |
| **Ex.No. 1** | **CREATING DATABASE TABLE** | **Date:** |

# CREATE TABLE

## CREATE TABLE *tablename* (column\_name data\_ type constraints, …)

* Used to create a table by defining its structure, the data type and name of the various columns, the relationships with columns of other tables etc.

**Q1)** Create the tables DEPT and EMP as described below

**SQL> CREATE TABLE dept057 (deptno int, dname varchar(255), loc varchar(255));**

**DEPT**

|  |  |  |
| --- | --- | --- |
| *Column name* | *Data type* | *Description* |
| DEPTNO | Number | Department number |
| DNAME | Varchar | Department name |
| LOC | Varchar | Department Location |

**SQL> CREATE TABLE emp057 (empno int, ename varchar(255), job char, mgr int, hiredate date, sal int, comm int, deptno int);**

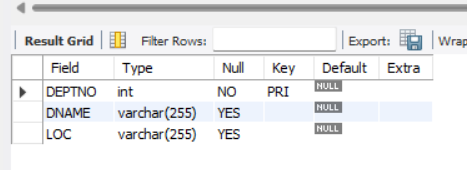
**EMP**

|  |  |  |
| --- | --- | --- |
| *Column name* | *Data type* | *Description* |
| EMPNO | Number | Employee number |
| ENAME | Varchar | Employee name |
| JOB | Char | Designation |
| MGR | Number | Manager’s EMP.No. |
| HIREDATE | Date | Date of joining |
| SAL | Number | Basic Salary |
| COMM | Number | Commission |
| DEPTNO | Number | Department Number |

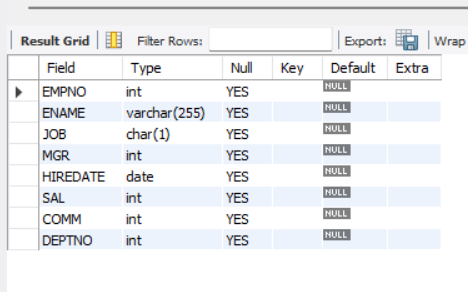
**Q2)** Confirm table creation

**desc *table-name;***

**SQL> DESC dept057;**

****

**SQL> DESC emp057;**

****

**Q3)** List name of the tables created by the user

## SQL> show tables;

**Q4)** Describe tables owned by the user

## SQL> desc table table\_name;

**Q5)** View distinct object types owned by the user

## SQL> SELECT DISTINCT object\_type FROM user\_objects;

**Q6)** View tables, views, synonyms, and sequences owned by the user

**SQL> SELECT \* FROM user\_catalog;**

# ALTER TABLE

* *Add Column*

**ALTER TABLE *table* ADD (*column data type* [DEFAULT *expr*] [, *column data type*] ...);**

* *Drop Column*

## ALTER TABLE *table* DROP column column\_name;

* *Modify Column*

**ALTER TABLE *table* MODIFY (*column data type* [DEFAULT *expr*] [, *column data type*] ...);**

**Q7)** Add new columns COMNT and MISCEL in DEPT table of character type.

**SQL >** **ALTER TABLE dept057 ADD (COMNT char, MISCEL char);**

**Q8)** Modify the size of column LOC by 15 in the DEPT table

**SQL >** **ALTER TABLE dept057 MODIFY (loc varchar (15));**

**Q9)** Set MISCEL column in the DEPT table as unused

**SQL >** **ALTER TABLE dept057 SET UNUSED (MISCEL);**

**Q10)** Drop the column COMNT from the table DEPT

**SQL >** **ALTER TABLE dept057 DROP COLUMN COMNT;**

**Q11)** Drop unused columns in DEPT table

**SQL > ALTER TABLE dept057 DROP UNUSED COLUMNS;**

**Q12)** Rename the table DEPT to DEPT12

**SQL >** **ALTER TABLE dept057 RENAME TO dept12;**

**Q13)** Remove all the rows in the table DEPT12 (Presently no records in DEPT12)

**SQL >** **DELETE FROM dept12;**

# ADDING COMMENTS TO THE TABLES

* You can add comments to a table or column by using the COMMENT statement.
* Comments can be viewed through the data dictionary views.
  + ALL\_COL\_COMMENTS
  + USER\_COL\_COMMENTS
  + ALL\_TAB\_COMMENTS
  + USER\_TAB\_COMMENTS

## Syntax:

**COMMENT ON TABLE** table **IS** 'xxxxxxxxxx';

**Q14)** Add some comment to the table DEPT12 and also confirm the inclusion of comment

**SQL >** **COMMENT ON TABLE dept12 IS ‘first comment’;**

**SQL >** **SELECT \* FROM all\_tab\_comments WHERE table\_name=‘dept12’;**

**Q15)** Delete the table DEPT12 from the database.

**SQL >** **DROP TABLE dept12;**

**Q16)** Confirm the removal of table DEPT12 from the database.

**SQL >** **SELECT \* FROM user\_tables WHERE table\_name=‘dept12’;**

***Verified by***

**Date:**

**Staff In-charge Sign:**

|  |  |  |
| --- | --- | --- |
| **Ex.No. 2** | **Working with Data Manipulation commands** | **Date :** |

# DML

* A DML statement is executed when you:
  + Add new rows to a table
  + Modify existing rows in a table
  + Remove existing rows from a table
* A *transaction* consists of a collection of DML statements that form a logical unit of work.

**Data for EMP table**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| EMPNO | ENAME | JOB | MGR | HIREDATE | SAL | COMM | DEPTNO |
| 7369 | SMITH | CLERK | 7902 | 17-DEC-80 | 800 |  | 20 |
| 7499 | ALLEN | SALESMAN | 7698 | 20-FEB-81 | 1600 | 300 | 30 |
| 7521 | WARD | SALESMAN | 7698 | 22-FEB-81 | 1250 | 500 | 30 |
| 70576 | JONES | MANAGER | 78057 | 02-APR-81 | 2975 |  | 20 |
| 7654 | MARTIN | SALESMAN | 7698 | 28-SEP-81 | 1250 | 1400 | 30 |
| 7698 | BLAKE | MANAGER | 78057 | 01-MAY-81 | 2850 |  | 30 |
| 7782 | CLARK | MANAGER | 78057 | 09-JUN-81 | 2450 |  | 10 |
| 7788 | SCOTT | ANALYST | 70576 | 19-APR-87 | 3000 |  | 20 |
| 78057 | KING | PRESIDENT |  | 17-NOV-81 | 5000 |  | 10 |
| 7844 | TURNER | SALESMAN | 7698 | 08-SEP-81 | 1500 | 0 | 30 |
| 7876 | ADAMS | CLERK | 7788 | 23-MAY-87 | 1100 |  | 20 |
| 7900 | JAMES | CLERK | 7698 | 03-DEC-81 | 950 |  | 30 |
| 7902 | FORD | ANALYST | 70576 | 03-DEC-81 | 3000 |  | 20 |
| 7934 | MILLER | CLERK | 7782 | 23-JAN-82 | 1300 |  | 10 |

**Data for DEPT table**

|  |  |  |
| --- | --- | --- |
| DEPTNO | DNAME | LOC |
| 10 | ACCOUNTING | NEW YORK |
| 20 | RESEARCH | DALLAS |
| 30 | SALES | CHICAGO |
| 40 | OPERATIONS | BOSTON |

# INSERT STATEMENT

* Add new rows to a table by using the INSERT statement.

1. **INSERT INTO *table* VALUES*(value1, value2,..*)*;***
   * Only one row is inserted at a time with this syntax.
   * List values in the default order of the columns in the table
   * Enclose character and date values within single quotation marks.
   * Insert a new row containing values for each column
2. **INSERT INTO *table(column1, column2,..).***

**VALUES*(value1, value2,..*)*;***

* + Rows can be inserted with NULL values either
    - by omitting column from the column list or
    - by specifying NULL in the value field.

1. **INSERT INTO *table(column1, column2,...)***

**VALUES*(&value1,& value2,..*)*;***

* + Substitution variable(&) helps us to write an interactive script for inserting rows

**Q1)** Insert the rows of DEPT table using syntax (i)

**SQL> INSERT INTO dept057 VALUES (10, ‘ACCOUNTING’, ‘NEW YORK’); INSERT INTO dept057 VALUES (20, ‘RESEARCH’, ‘DALLAS’); INSERT INTO dept057 VALUES (30, ‘SALES’, ‘CHICAGO’); INSERT INTO dept057 VALUES (40, ‘OPERATIONS’, ‘BOSTON’);**

**Q2)** Insert first & second rows of EMP table using syntax (ii)

**SQL>** **INSERT INTO emp057 (empno, ename, job, mgr, hiredate, sal, deptno) VALUES (7369, ‘SMITH’, ‘CLERK’, 7902, ‘1980-12-17’, 800, 20); INSERT INTO emp057 (empno, ename, job, mgr, hiredate, sal, comm, deptno) VALUES (7499, ‘ALLEN’, ‘SALESMAN’, 7698, ‘1981-02-26’, 1600, 300, 30);**

**Q3)** Insert the remaining rows of EMP table using syntax (iii).

**SQL>** **INSERT INTO emp057 (empno, ename, job, mgr, hiredate, sal, comm, deptno) VALUES (‘&empno’, ‘&ename’, ‘&job’, ‘&mgr’, ‘&hiredate’, ‘&sal’, ‘&comm’, ‘&deptno’):**

**Q4)** Create a table MANAGER with the columns *mgr-id, name, salary* and *hiredate*

### SQL> CREATE TABLE manager\_057(mgr-id number(5) primary key, name varchar(20), sal number(5), hiredate date);

**Q5)** Insert values into the table MANAGER by copying the values from EMP table where the designation of the employee is ‘MANAGER’

**SQL> INSERT INTO *manager\_057* SELECT *empno,ename,sal,hiredate***

**FROM emp WHERE job=’MANAGER’;**

# UPDATE STATEMENT

* Modify existing rows with the UPDATE statement.
* Update more than one row at a time, if required.
* All rows in the table are modified if you omit the WHERE clause

**UPDATE *table* SET *column* = *value*, *column* = *value, ….* WHERE *condition*;**

**Q6)** Change the LOC of all rows of DEPT table by ‘NEW YORK’

**SQL>** **UPDATE dept057 SET loc=‘NEW YORK’;**

**Q7)** Change the LOC=’DALLAS’ for deptno=20 in DEPT table.

# SQL>

**UPDATE dept057 SET loc=‘DALLAS’ WHERE deptno=20;**

# DELETE STATEMENT

* You can remove existing rows from a table by using the DELETE statement.
* All rows in the table are deleted if you omit the WHERE clause.

**DELETE FROM *table* WHERE *condition*;**

**Q8)** Delete the rows from EMP table whose employee name = ‘PAUL’

# SQL>

**DELETE FROM emp057 WHERE ename=‘PAUL’**

# SELECT STATEMENT

* To perform a query we use select statement

**SELECT [DISTINCT] {\*, *column* [*alias*],...} FROM *table;***

* + *Select* Clause determines what columns
  + *From* Clause determines which table.
  + *Where* Clause specifies the conditions

**Q9)** List all the columns and rows of the table DEPT

# SQL>

**SELECT \* FROM dept057;**

**Q10)** List the name of the employee and salary of EMP table

# SQL>

**SELECT ename, sal FROM emp057;**

**Q11)** Without duplication, list all names of the department of DEPT table.

# SQL>

**SELECT DISTINCT dname FROM dept057;**

**Q12)** Find out the name of an employee whose EMPNO is 7788.

# SQL>

**SELECT ename FROM dept057 WHERE empno=7788;**

**Q13)** As a copy of DEPT table, create DEPT1 table using select command.

## SQL> CREATE TABLE dept1\_057 AS SELECT \* FROM dept;

**Q14)** List ename and sal of EMP table with the column headings NAME and SALARY

**SQL> SELECT ename as name, sal salary FROM emp057;**

# DATABASE TRANSACTIONS

* Begin when the first executable SQL statement is executed
* End with one of the following events:
* COMMIT or ROLLBACK
* DDL or DCL statement executes (automatic commit)
* User exits
* System crashes

**TCL (**Transaction Control Language)

* TCL Statements are COMMIT, ROLLBACK & SAVE POINT.
* State of Data Before COMMIT or ROLLBACK
  + The previous state of the data can be recovered.
  + The current user can review the results of the DML operations by using the SELECT statement.
  + Other users *cannot* view the results of the DML statements by the current user.
  + The affected rows are *locked*; other users cannot change the data within the affected rows.
* State of Data After COMMIT
  + Data changes are made permanent in the database.
  + The previous state of the data is permanently lost.
  + All users can view the results.
  + Locks on the affected rows are released; those rows are available for other users to manipulate.
  + All savepoints are erased.

**Q15)** Change LOC=’CHICAGO’ for deptno=30 in DEPT table and COMMIT the transaction.

**SQL>** **UPDATE dept057 SET loc=‘CHICAGO’ WHERE deptno=30;**

**SQL>** **COMMIT;**

* State of Data After ROLLBACK
  + Discard all pending changes by using the ROLLBACK statement.
  + Data changes are undone.
  + Previous state of the data is restored.
  + Locks on the affected rows are released.

**Q16)** Delete all the rows from EMP table and ROLLBACK the transaction.

**SQL>** **DELETE FROM emp057;**

**SQL>** **ROLLBACK;**

* Rolling Back to a Marker
  + Create a marker within a current transaction by using

<SAVEPOINT savepoint-name>

* + Roll back to that marker by using <ROLLBACK TO savepoint-name>

**Q17)** Do the following operations one after another

1. Change LOC=’BOSTON’ for deptno=40 in DEPT table

**SQL>** **UPDATE dept057 SET loc=‘BOSTON’ WHERE deptno=40;**

1. Create SAVEPOINT in the name ‘update\_over’

**SQL>** **SAVEPOINT update\_over;**

1. Insert another row in DEPT table with your won values

**SQL>** **INSERT INTO dept057 VALUES (33, ‘MANUFACTURING’, ‘SHENZHEN’);**

1. Rollback the transaction upto the point ‘update\_over’

**SQL>** **ROLLBACK TO update\_over;**

***Verified by***

**Date :**

**Staff In-charge Sign :**

|  |  |  |
| --- | --- | --- |
| **Ex.No. 3** | **I N T E G R I T Y C O N S T R A I N T S** | **Date :** |

# CONSTRAINTS

* Constraints enforce rules at the table level.Constraints prevent the deletion of a table if there are dependencies.
* The following constraint types are valid in Oracle:
  + NOT NULL
  + UNIQUE Key
  + PRIMARY KEY
  + FOREIGN KEY
  + CHECK
* Name a constraint or the Oracle Server will generate a name by using the SYS\_C*n* format.
* Create a constraint:
  + At the same time as the table is created
  + After the table has been created
* Define a constraint at the column or table level.
* View a constraint in the data dictionary.

# DEFINING CONSTRAINTS

* Column constraint level

***column* [CONSTRAINT *constraint\_name*] *constraint\_type***

* Table constraint level

**[CONSTRAINT *constraint\_name*] *constraint\_type*(*column*)**

**CREATE TABLE *table* (*column data type, column\_constraint*,**

**…** **,**

**… … …,**

### table\_constraint);

**Q1)** Create table EMP1 with columns similar to EMP table and create NOT NULL (column) constraint for DEPTNO column and PRIMARY KEY (table) constraint for EMPNO column.

**SQL> CREATE TABLE emp1\_057( empno number(4), ename varchar2(10), job char(20), mgr number(10),**

**hiredate date, sal number(5), comm number(5), deptno number(7,2) NOT NULL,**

**CONSTRAINT emp1\_pk PRIMARY KEY (empno));**

**NOT NULL Constraint**

* Ensures that null values are not permitted for the column

## CHECK Constraint

* Defines a condition that each row must satisfy

## UNIQUE Constraint

* Prevent the duplication of values within the rows of a specified column

## PRIMARY KEY Constraint

* Avoids duplication of rows and does not allow NULL values

## FOREIGN KEY Constraint

* To establish a ‘parent-child’ or a ‘master-detail’ relationship between two tables having a common column, we make use of Foreign key (referential integrity) constraints.
* To do this we should define the column in the parent table as primary key and the same column in the child table as a foreign key referring to the corresponding parent entry.

*FOREIGN KEY*

* + Defines the column in the child table at the table constraint level

*REFERENCES*

* + Identifies the table and column in the parent table

*ON DELETE CASCADE*

* + Allows deletion in the parent table and deletion of the dependent rows in the child table

# ADDING A CONSTRAINT

* Add or drop, but not modify, a constraint
* Add a NOT NULL constraint by using the MODIFY clause

**ALTER TABLE *table* ADD CONSTRAINT *const-name* cons-*type* (*column*); Q2)** Add NOT NULL constraint to the columns ENAME and JOB of EMP table.

## SQL> ALTER TABLE emp057 MODIFY(ename varchar2(20) NOT NULL,

**job char(20) NOT NULL);**

**Q3)** Add Primary key constraint to the column EMPNO of EMP table

## SQL> ALTER TABLE emp057 ADD CONSTRAINT emp\_pk PRIMARY KEY(empno);

**Q4)** Add Primary key constraint to the column DEPTNO of DEPT table

**SQL>** **ALTER TABLE dept057 ADD CONSTRAINT dept\_pk PRIMARY KEY (deptno);**

**Q5)** Add Unique key constraint to the column DNAME of DEPT table

**SQL>** **ALTER TABLE dept057 ADD CONSTRAINT dept\_uname UNIQUE (dname);**

**Q6)** Add Check constraint to the table EMP to restrict the values of EMPNO lies between 7000 and 8000.

## SQL> ALTER TABLE emp057 ADD CONSTRAINT emp\_ck CHECK(empno BETWEEN 7000 AND 8000)

**Q7)** Add Foreign key constraint to the column DEPTNO of EMP table references DEPTNO of DEPT table.

## SQL> ALTER TABLE emp057 ADD CONSTRAINT emp\_fk FOREIGN KEY(deptno) REFERENCES DEPT(deptno);

**Q8)** Add a Foreign key constraint to the EMP1 table indicating that a manager must already exist as a valid employee in the EMP1 table.

**SQL>** **ALTER TABLE emp1\_057 ADD CONSTRAINT manager\_fk FOREIGN KEY**

**(mgr) REFERENCES emp1\_057(empno);**

# DROPING CONSTRAINTS

* Removing constraints from the table

**ALTER TABLE *table* DROP CONSTRAINT *const-name;***

**Q9)** Remove the Manager constraint (added in Q8) from EMP table

**SQL>** **ALTER TABLE emp057 DROP CONSTRAINT manager\_fk;**

**Q10)** Remove the primary key constraint on the DEPT table and drop the associated foreign key constraint on the EMP.DEPTNO column.

## SQL> ALTER TABLE dept057 DROP PRIMARY KEY CASCADE;

**DISABLE and ENABLE Constraint**

* Execute the DISABLE clause of the ALTER TABLE statement to deactivate an integrity constraint.
* Apply the CASCADE option to disable dependent integrity constraints.
* Activate an integrity constraint currently disabled in the table definition by using the ENABLE clause.

A UNIQUE or PRIMARY KEY index is automatically created if you enable a UNIQUE key or PRIMARY KEY constraint.

**Q11)** Disable the primary key constraint of EMP table.

## SQL> ALTER TABLE emp057 DISABLE CONSTRAINT emp\_pk CASCADE;

**Q12)** Enable the primary key constraint of EMP table.

# SQL>

**ALTER TABLE emp057 ENABLE CONSTRAINT emp\_pk CASCADE;**

**Q13)** Query the USER\_CONSTRAINTS table to view all constraint definitions and names

## SQL> SELECT constraint\_name, constraint\_type, search\_condition FROM user\_constraints

**WHERE table\_name = 'EMP057';**

**Q14)** View the columns associated with the constraint names in the USER\_CONS\_COLUMNS view

## SQL> SELECT constraint\_name, column\_name FROM user\_cons\_columns

**WHERE table\_name = 'EMP057';**

***Verified by***

**Date :**

**Staff In-charge Sign :**

|  |  |  |
| --- | --- | --- |
| **Ex.No. 4** | **BASIC SELECT STATEMENTS** | **Date :** |

## Arithmetic Operators

+ Addition

- Subtraction

\* Multiplication

/ Division

## Comparison Operators

= Equal to

<> Not Equal to

< Less than

> Greater than

<= Less than or equal to

>= Greater than or equal to IN (List) Match any of list of values

LIKE Match a character pattern (%  any no. of characters, -  One Character) IS NULL Is a null value

BETWEEN…AND… Between two values

## Logical Operators

AND Returns TRUE if *both* component conditions are TRUE OR Returns TRUE if *either* component condition is TRUE NOT Returns TRUE if the following condition is FALSE

## Concatenation Operator ( || )

* Concatenates the Columns of any data type.
* A Resultant column will be a Single column.

## Operator Precedence

|  |  |
| --- | --- |
| *Order Evaluated* | *Operators* |
| 1 | Parenthesis |
| 2 | All Arithmetic Operators (Multiplication and Division followed by Addition and subtraction) |
| 3 | All Comparison Operators |
| 4 | NOT |
| 5 | AND |
| 6 | OR |

**Where Clause**

* Specify the Selection of rows retrieved by the WHERE Clause

**SELECT *column1, column2, …***

**FROM *table***

**WHERE *condition*;**

* The WHERE clause follows the FROM clause

## Order by Clause

* Sort rows specified by the order ASC / DESC **SELECT *column1, column2, … …* FROM *table***

**ORDER BY *sort-column* DESC;**

* Sorts *table* by *sort-column* in descending order
* Omitting the keyword DESC will sort the table in ascending order

*Note :*

* AS Keyword between the column name and the actual alias name
* Date and character literal values must be enclosed within single quotation marks
* Default date format is 'DD-MON-YY'
* Eliminate duplicate rows by using the DISTINCT keyword

**Q1)** Update all the records of *manager* table by increasing 10% of their salary as bonus.

**SQL>** **UPDATE manager\_057 SET sal=sal\*1.10;**

**Q2)** Delete the records from *manager* table where the salary less than 2750.

**SQL>** **DELETE FROM manager\_057 WHERE sal < 2750;**

**Q3)** Display each name of the employee as “Name” and annual salary as “Annual Salary” (Note: Salary in *emp* table is the monthly salary)

**SQL>** **SELECT ename AS "Name", (sal \* 12) AS "Annual Salary" FROM emp057;**

**Q4)** List concatenated value of name and designation of each employee.

**SQL>** **SELECT CONCAT(ename, ' - ', job) AS "Employee Details" FROM emp057;**

**Q5)** List the names of Clerks from *emp* table.

# SQL>

SELECT ename AS “CLERKS” FROM emp057 WHERE job = ‘CLERK’;

**Q6)** List the Details of Employees who have joined before 30 Sept 81.

# SQL>

**SELECT \* FROM emp057 WHERE hiredate < '1981-09-30';**

**Q7)** List names of employees who’s employee numbers are 7369,78057,7934,7788.

# SQL>

SELECT ename FROM emp057 WHERE empno IN (7369, 78057, 7934, 7788);

**Q8)** List the names of employee who are not Managers.

# SQL>

# SELECT ename FROM emp057 WHERE job <> ’MANAGER’;

**Q9)** List the names of employees not belonging to dept no 30,40 & 10

# SQL>

# SELECT ename FROM emp057 WHERE deptno NOT IN (30, 40,10)

**Q10)** List names of those employees joined between 30 June 81 and 31 Dec 81.

# SQL>

# SELECT \* FROM emp057 WHERE hiredate BETWEEN '1981-06-30' AND '1981-12-31';

**Q11)** List different designations in the company.

# SQL>

SELECT DISTINCT job FROM emp057;

**Q12)** List the names of employees not eligible for commission.

# SQL>

# SELECT ename FROM emp057 WHERE comm IS NULL;

**Q13)** List names and designations of employee who does not report to anybody

# SQL>

**SELECT ename, job FROM emp057 WHERE mgr IS NULL**

**Q14)** List all employees not assigned to any department.

**SQL> SELECT \* FROM emp057 WHERE deptno IS NULL;**

**Q15)** List names of employee who are eligible for commission.

# SQL>

**SELECT ename FROM emp057 WHERE comm IS NOT NULL;**

**Q16)** List employees whose name either start or end with ‘s’.

# SQL>

**SELECT ename FROM emp057 WHERE ename LIKE 'S%' OR ename LIKE '%s';**

**Q17)** List names of employees whose names have ‘i’ as the second character.

# SQL>

**SELECT ename FROM emp057 WHERE ename LIKE '\_i%';**

**Q18)** Sort *emp* table in ascending order by *hire-date* and list *ename, job, deptno* and

*hire-date.*

# SQL>

**SELECT ename, job, deptno, hire-date FROM emp057 ORDER BY hire-date ASC;**

**Q19)** Sort *emp* table in descending order by annual salary and list *empno, ename, job*

and *annual-salary.* (Note : Salary in *emp* table is the monthly salary)

# SQL>

**SELECT empno, ename, job, (sal\*12) AS annual\_salary FROM emp057 ORDER BY (sal\*12) DESC;**

**Q20)** List *ename, deptno* and *sal* after sorting *emp* table in ascending order by *deptno*

and then descending order by *sal.* (Note : Sorting by multiple coluns)

# SQL>

### Verified by

**SELECT ename, deptno, sal FROM emp057 ORDER BY deptno ASC, sal DESC;**

**Date :**

**Staff In-charge Sign :**

|  |  |  |
| --- | --- | --- |
| **Ex. No. 5** | **S Q L F U N C T I O N S** | **Date :** |

SQL functions are of two types

## Single row functions or scalar functions

* + Returns only one value for every row queried in the table
  + Can be used in Select clause and where clause
  + It can be broadly classified into 5 categories
    - Date Functions
    - Character Functions
    - Conversion functions
    - Numeric functions
    - Miscellaneous functions

## Group functions or multiple-row functions

Discussed in the next exercise (ie.; Ex. No.6)

**Note :** The exercises that follow mostly uses system table ‘dual’. It is a table which is automatically created by Oracle along with the data dictionary. Dual table has one column defined to be of varchar2 type and contains only one row with value ‘x’.

**SCALAR FUNCTIONS**

**Q1)** List the hiredate of employees who work in deptno 20 in a format like ‘WEDNESDAY JANUARY 12, 1983’

(Hint: DAY : Day of the week, MONTH : Name of the month, DD: Day of the month, and YYYY : Year)

**SQL> SELECT FORMAT(hiredate, 'dddd MMMM dd, yyyy') AS fmt\_hiredate FROM emp057 WHERE deptno = 20;**

**Q2)** Display the hiredate with time of employess who work in deptno 20.

**SQL> SELECT FORMAT(hiredate, 'dddd MMMM dd, yyyy HH:mm') AS fmt\_hiredate FROM emp057 WHERE deptno = 20;**

**Q3)** Each employee receives a salary review after every 150 days of service. Now list employee name, hiredate and first salary review date of each employee who work in dept no 20.

**SQL> SELECT ename, hiredate, hiredate+150 AS "First Salary Review Date" FROM emp057 WHERE deptno = 20;**

**Q4. Date Functions**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Functions** | **Value Returned** | | **Input** | | | **Output** |
| add\_months(d,n) | ‘n’ months added to date ‘d’. | | Select add\_months(sysdate,2) from dual; | | | **2025-057-15** |
| last\_day(d) | Date corresponding to the last day of the month | | Select last\_day(sysdate) from dual; | | | **2025-03-31** |
| to\_date(str,’format’) | Converts the string ina given format into Oracle date. | | Select to\_date(’10-02-09’,’dd-mm-yy’) from dual; | | | **2009-02-10** |
| to\_char(date,’format’) | Reformats date according to format | | Select to\_char(sysdate,’dy yyyy’) from dual; | dd | mon | **sat 15 mar 2025** |
| months\_between(d1,d2) | No. of months between two dates | | Select months\_between(sysdate, to\_date(’10-10-07’,’dd-mm-yy’) )  from dual; | | | **209.1852400686977299**  **88057205786977299880573** |
| next\_day(d,day) | Date of the ‘day’ that immediately follows the date ‘d’ | | Select next\_day(sysdate,’wednesday’) from dual; | | | **2025-03-19** |
| round(d,’format’) | Date will  rounded to nearest day. | be the | Select round(sysdate,’year’) from dual; | | | **2025-01-01** |
| Select round(sysdate,’month’) from dual; | | | **2025-03-01** |
|  | Select round(sysdate,’day’) from dual; | | | **2025-03-16** |
|  |  | Select round(sysdate) from dual; | | | **2025-03-16** |
| trunc(d,’format’); | Date will truncated to nearest day. | be the | Select trunc(sysdate,’year’) from dual; | | | **2025-01-01** |
| Select trunc(sysdate,’month’) from dual; | | | **2025-03-01** |
|  | Select trunc(sysdate,’day’) from dual; | | | **2025-03-09** |
|  |  | Select trunc(sysdate) from dual; | | | **2025-03-15** |
| greatest(d1,d2,…) | Picks latest of list of dates | | Select greatest(sysdate, to\_date(‘02-10- 06’,’dd-mm-yy’), to-date(’12-07- 12’,’dd-mm-yy’)) from dual; | | | **2025-03-15** |
| Date Arithmetic | Add /Subtract no. of days to a date | | Select sysdate+25 from dual; | | | **2025-04-09** |
| Select sysdate-25 from dual; | | | **2025-02-18** |
| Subtract one date from another, producing a no. of days | | Select sysdate - to\_date(‘02-10-06’,’dd- mm-yy’) from dual; | | | **67057.871643518518518**  **51851851851851851852** |

**Q5. Character Functions**

|  |  |  |  |
| --- | --- | --- | --- |
| **Functions** | **Value Returned** | **Input** | **Output** |
| initcap(char) | First letter of each word capitalized | Select initcap(‘database management’) from dual; | **Database Management** |
| lower(char) | Lower case | Select lower(‘WELCOME’) from dual; | **welcome** |
| upper(char) | Upper case | Select upper(‘srmist’) from dual; | **SRMIST** |
| ltrim(char, set) | Initial characters removed up to the character not in set. | Select ltrim(‘lordourgod’,’lord’) from dual; | **urgod** |
| rtrim(char, set) | Final characters removed after the last character not in set. | Select rtrim(‘godlovesyou’,’you’) from dual; | **godloves** |
| translate(char, from, to) | Translate ‘from’ by ‘to’ in char. | Select translate(‘jack’,’j’,’b’) from dual; | **back** |
| replace(char, search, repl) | Replace ‘search’ string by ‘repl’ string in ‘char’. | Select replace(‘jack and jue’,’j’,’bl’) from dual; | **black and blue** |
| substr(char, m, n) | Substring of ‘char’ at ‘m’ of size ‘n’ char long. | Select  substr(‘wages of sin is death’,10,3) from dual; | **sin** |

**Q6. Conversion Functions**

|  |  |  |  |
| --- | --- | --- | --- |
| **Functions** | **Value Returned** | **Input** | **Output** |
| to\_date(str,’format’) | Converts the string ina given format into Oracle date. | Select to\_date(’10-02-09’,’dd-mm-yy’) from dual; | **2009-02-10** |
| to\_char(date,’format’) | Reformats date  according to format | Select to\_char(sysdate,’dy dd mon yyyy) from dual; | **sat 15 mar 2025** |
| to\_char(number,’format’) | Display number value as a char. | Select to\_char(12345.5,’L099,999.99’) from dual; | **$012,345.50** |
| to\_number(char) | Char string to number form | Select to\_number(‘123’) from dual; | **123** |

**Q7. Numeric Functions**

|  |  |  |  |
| --- | --- | --- | --- |
| **Functions** | **Value Returned** | **Input** | **Output** |
| Abs(n) | Absolute value of n | Select abs(-15) from dual; | **15** |
| Ceil(n) | Smallest int >= n | Select ceil(33.645) from dual; | **34** |
| Cos(n) | Cosine of n | Select cos(180) from dual; | **-.59846006905778581389** |
| Cosh(n) | Hyperbolic cosine of n | Select cosh(0) from dual; | **1** |
| Exp(n) | en | Select exp(2) from dual; | **7.3890576098930650227** |
| Floor(n) | Largest int <= n | Select floor(100.2) from dual; | **100** |
| Ln(n) | Natural log of n (base e) | Select ln(5) from dual; | **1.609437912434100374**  **600759333226187605753** |
| Log(b,n) | Log n base b | Select log(2,64) from dual; | **5.999999999999999999** |
| Mod(m,n) | Remainder of m divided by n | Select mod(17,3) from dual; | **2** |
| Power(m,n) | m power n | Select power(5,3) from dual; | **125** |
| Round(m,n) | m rounded to n decimal places | Select round(125.67854,2) from dual; | **125.68** |
| Sign(n) | If n<0, -1 if n=0, 0  otherwise 1. | Select sin(-19) from dual; |  |
| Sin(n) | Sin of n | Select sin(90) from dual; | **.8905796663600575789057** |
| Sinh(n) | Hyperbolic sin of n | Select sinh(45) from dual; | **17467135528742547674.01** |
| Sqrt(n) | Square root of n | Select sqrt(7) from dual; | **2.6457513110645905790** |
| Tan(n) | Tangent of n | Select tan(45) from dual; | **1.6197751905743861549** |
| Tanh(n) | Hyperbolic tangent of n | Select tanh(60) from dual; | **1** |
| Trunc(m,n) | m truncated to n decimal places | Select trunc(125.5764,2) from dual; | **125.57** |

**Q8. Miscellaneous Functions**

|  |  |  |  |
| --- | --- | --- | --- |
| **Functions** | **Value Returned** | **Input** | **Output** |
| Uid | User id | Select uid from dual; | **73373** |
| User | User name | Select user from dual; | **APEX\_PUBLIC\_USER** |
| Vsize(n) | Storage size of v | Select vsize(‘hello’) from dual; | **5** |
| NVL(exp1,exp2) | Returns exp1 if not null, otherwise returns exp2. | Select nvl(comm,50) from emp where empno=7369; | **50** |

**GROUP FUNCTIONS**

## Common Group Functions

* AVG : Average value of a set
* COUNT : Numbers of non null values
* MAX : Maximum of a set
* MIN : Minimum of a set
* STDDEV : Standard Deviation of a set
* SUM : Sum of a set
* VARIANCE : Variance of a set

## Syntax :

SELECT *column*, *group\_function(column)*

FROM *table*

[WHERE *condition*]

[GROUP BY *group\_column\_or\_expression*] [HAVING *group\_condition*]

[ORDER BY *column*];

* Group functions ignore null values
* *Group by* Clause is used to modularize rows in a table into smaller groups
* Columns that are not a part of the Group Functions should be included in the Group by clause
* Any column or expression in the SELECT list that is not an aggregate function must be in the GROUP BY clause
* Group Functions cannot be placed in the where clause
* HAVING clause is to restrict groups Groups satisfying the HAVING condition are displayed
* Order of evaluation of the clauses :
  + WHERE clause
  + GROUP BY clause
  + HAVING clause

**Q9)** Find number of rows in the table EMP

**SQL > SELECT COUNT(\*) FROM emp057;**

**Q10)** Find number of designations available in EMP table.

# SQL>

**SELECT COUNT(DISTINCT job) FROM emp057;**

**Q11)** Find number of employees who earn commission in EMP table.

# SQL>

**SELECT COUNT(COMM) FROM emp057;**

**Q12)** What is the difference between the following queries

**SQL > select *count(comm)* from *emp057*;**

**SQL > select *count(nvl(comm,0))* from *emp057*;**

**COUNT(comm) counts only non-null values, whereas COUNT(NVL(comm, 0)) count all the values as NVL replaces null values.**

**Q13)** Find the total salary paid to the employees.

# SQL>

**SELECT SUM(sal) FROM emp057;**

**Q14)** Find maximum, minimum and average salary in EMP table.

# SQL>

**SELECT MAX(sal), MIN(sal), AVG(sal) FROM emp057;**

**Q15)** Find number of employees who work in department number 30

**SQL> select *count(\*)* from *emp***

## where deptno=30;

**Q16)** Find the maximum salary paid to a ‘CLERK’

# SQL>

**SELECT MAX(sal) FROM emp057 WHERE job = ‘CLERK’;**

**Q17)** List the department numbers and number of employees in each department

# SQL>

**SELECT deptno, COUNT(\*) FROM emp057 GROUP BY deptno;**

**Q18)** List the jobs and number of employees in each job. The result should be in the descending order of the number of employees.

**SQL> select *job, count(\*)* from *emp057***

**group by *job***

**order by *count\*)* desc;**

**Q19)** List the total salary, maximum and minimum salary and average salary of the employees jobwise.

**SQL> SELECT job, SUM(sal), MAX(sal), MIN(sal), AVG(sal) FROM**

**emp057 GROUP BY job;**

**Q20)** List the total salary, maximum and minimum salary and average salary of the employees jobwise, for department 20 and display only those rows having an average salary > 1000.

**SQL> select *job,sum(sal), max(sal), min(sal), avg(sal)***

**from *emp057***

**group by *job, deptno***

### having deptno=20 and avg(sal) > 1000;

**Q21)** List the job and total salary of employees jobwise, for jobs other than ‘PRESIDENT’ and display only those rows having total salary > 5000.

# SQL>

**SELECT job, SUM(sal) FROM emp057 WHERE job <> ‘PRESIDENT’ GROUP BY job**

**HAVING SUM(sal) > 5000;**

**Q22)** List the job**,** number of employees and average salary of employees jobwise.

Display only the rows where the number of employees in each job is more than two.

# SQL>

### Verified by

**SELECT job, COUNT(\*), AVG(sal) FROM emp \_057 GROUP BY job**

**HAVING COUNT(\*) > 2;**

**Date :**

**Staff In-charge Sign :**

|  |  |  |
| --- | --- | --- |
| **Ex. No. 6** | **JOINING TABLES** | **Date :** |

## Set operators

* Set operators combine the results of two queries into a single.

|  |  |
| --- | --- |
| **Operator** | **Function** |
| Union | Returns all distinct rows selected by either query |
| Union all | Returns all rows selected by either query including duplicates |
| Intersect | Returns only rows that are common to both the queries |
| Minus | Returns all distinct rows selected only by the first query and not by the second. |

**Q1)** Create the following tables :

depositor(cus\_name,acno) & borrower(cus\_name,loanno)

**SQL> CREATE TABLE depositor\_057 (cus\_name varchar(255), acno int);**

**SQL> CREATE TABLE borrower\_057 (cus\_name varchar(255), loanno int);**

**Q2)** List the names of distinct customers who have either loan or account

**SQL> SELECT DISTINCT cus\_name FROM depositor\_057 UNION SELECT DISTINCT cus\_name FROM borrower\_057;**

**Q3)** List the names of customers (with duplicates) who have either loan or account

**SQL> (select *cus\_name* from *borrower\_057*)**

**union all (select *cus\_name* from *depositor\_057*)**

**Q4)** List the names of customers who have both loan and account

# SQL>

**SELECT DISTINCT cus\_name FROM depositor\_057 INTERSECT SELECT DISTINCT cus\_name FROM**

**borrower\_057;**

**Q5)** List the names of customers who have loan but not account

**SQL> SELECT DISTINCT cus\_name FROM borrower\_057 MINUS SELECT DISTINCT cus\_name FROM depositor\_057;**

## Joins

* Used to combine the data spread across tables

## Syntax

SELECT *table1.column, table2.column*

FROM *table1, table2*

WHERE *table1.column1* = *table2.column2*;

* A JOIN Basically involves more than one Table to interact with.
* Where clause specifies the JOIN Condition.
* Ambiguous Column names are identified by the Table name.
* If join condition is omitted, then a **Cartesian product** is formed. That is all rows in the first table are joined to all rows in the second table

## Types of Joins

* Inner Join (Simple Join) : It retrieves rows from 2 tables having a common column.
  + Equi Join : A join condition with relationship = .
  + Non Equi Join : A join condition with relationship other than = .
* Self Join : Joining of a table to itself
* Outer Join : Returns all the rows returned by simple join as well as

those rows from one table that do not match any row from the other table. The symbol (+) represents outer joins.

**Q6)** List *empno, ename, deptno* from *emp* and *dept* tables.

# SQL>

**SELECT e.empno, e.ename, e.deptno FROM emp057 e INNER JOIN dept057 d ON e.deptno = d.deptno;**

**Q7)** Create a table *Salgrade* with the following data .

## Grade Losal Hisal

|  |  |  |
| --- | --- | --- |
| 1 | 700 | 1400 |
| 2 | 1401 | 2000 |
| 3 | 2001 | 5000 |
| 4 | 5001 | 9999 |

Now, list *ename, sal* and *salgrade* of all employees.

# SQL>

**CREATE TABLE salgrade\_057 (grade int, losal int, hisal int);**

**INSERT INTO salgrade\_057 VALUES (1, 700, 1400), (2, 1401, 2000), (3, 2001, 5000), (4, 5001,**

**9999);**

**SELECT e.ename, e.sal, s.grade AS salgrade\_057;**

**Q8)** List *ename, deptno* and *deptname* from *emp* and *dept* tables, including the rows of

*emp* table that does not match with any of the rows in *dept* table.

# SQL>

**SELECT e.ename, e.deptno, d.deptname FROM emp057 e LEFT JOIN dept057 d ON e.deptno ==.deptno;**

**Q9)** List *ename, deptno* and *deptname* from *emp* and *dept* tables, including the rows of

*dept* table that does not match with any of the rows in *emp* table.

# SQL>

**SELECT e.ename, e.deptno, d.deptname FROM emp057 e RIGHT JOIN dept057 d ON e.deptno = d.deptno;**

**Q10)** List the names of the employee with name of his/her manager from *emp* table.

# SQL>

### Veried by

**SELECT e.ename, m.ename FROM emp057 e LEFT JOIN emp057 m ON e.mgr = m.empno;**

**Date :**

**Staff In-charge Sign :**

|  |  |  |
| --- | --- | --- |
| **Ex. No. 7** | **S U B Q U E R I E S** | **Date :** |

* Nesting of queries, one within the other is termed as sub query.

## Syntax

SELECT *select\_list*

FROM *table*

WHERE *expr operator* ( SELECT *select\_list*

FROM *table*);

* The subquery (inner query) executes once before the main query.
* The result of the subquery is used by the main query (outer query).

## Guidelines for Subqueries

* Enclose subqueries in parentheses.
* Place subqueries on the right side of the comparison operator.
* Do not add an ORDER BY clause to a subquery.
* Use single-row operators with single-row subqueries.
* Use multiple-row operators with multiple-row subqueries.

## Single-Row Subqueries

* Return only one row
* Use single-row comparison operators (ie; relational operators)

## Multiple-Row Subqueries

* Return more than one row
* Use multiple-row comparison operators

|  |  |
| --- | --- |
| ***Operator*** | ***Meaning*** |
| **IN** | Equal to **any** member in the list |
| **ANY** | Compare value to **each value** returned by the subquery |
| **ALL** | Compare value to **every value** returned by the subquery |

## Note:

‘**=any**’ is equivalent to ‘**in**’ ‘**!=all**’ is equivalent to ‘**not in**’

**Q1)** List the name of the employees whose salary is greater than that of employee with empno 70576.

**SQL> select *ename* from *emp057***

**where *sal* > (select *sal* from *emp057***

**where *empno*=70576);**

**Q2)** List the name of the employees whose job is equal to the job of employee with empno 7369 and salary is greater than that of employee with empno 7876.

# SQL>

**SELECT ename FROM emp057**

**WHERE job = (SELECT job FROM emp057 WHERE empno = 7369) AND sal > (SELECT sal FROM emp057 WHERE empno = 7876);**

**Q3)** List the *ename,job,sal* of the employee who get minimum salary in the company.

**SQL> select *ename, job, sal* from *emp057***

**where *sal* = (select *min(sal)* from *emp057*);**

**Q4)** List *deptno & min(salary)* departmentwise, only if *min(sal)* is greater than the

*min(sal)* of *deptno* 20.

**SQL> select *deptno, min(sal)* from *emp057***

**group by *deptno***

**having *min(sal)* > (select *min(sal)* from *emp057***

**where *deptno* = 20);**

**Q5)** List *empno, ename, job* of the employees whose *job* is not a ‘CLERK’ and whose *salary* is less than at least one of the salaries of the employees whose *job* is ‘CLERK’.

**SQL> select *empno, ename, job* from *emp057***

**where *sal* < any (select *sa* from *emp057***

## where *job* = 'CLERK') and *job* <> ' CLERK ';

**Q6)** List *empno, ename, job* of the employees whose salary is greater than the average salary of each department.

**SQL> SELECT e.empno, e.ename, e.job FROM emp057 e**

**WHERE e.sal > (SELECT AVG(sal) FROM emp057 WHERE deptno = e.deptno);**

**Q7)** Display the *name, dept. no, salary,* and *commission* of any employee whose *salary* and *commission* matches both the *commission* and *salary* of any employee in department 30.

**SQL> select *ename, deptno, sal, comm***

**from *emp057***

**where (*sal, nvl(comm,-1)*) in ( select *sal, nvl(comm,-1)***

**from *emp057***

**where *deptno* = 30);**

**Q8)** List *ename sal, deptno, average salary* of the dept where he/she works, if salary of the employee is greater than his/her department average salary.

**SQL> select *a.ename, a.sal, a.deptno, b.salavg***

**from *emp057 a*, ( select *deptno, avg(sal) salavg***

**from *emp057***

**group by *deptno*) *b***

**where *a.deptno = b.deptno***

**and *a.sal > b.salavg*;**

**Q9)** Execute and Write the output of the following query in words.

**SQL> with *summary* as**

**(select *dname,sum(sal) as dept\_total* from *emp057 a , dept057 b***

**where *a.deptno = b.deptno***

**group by *dname*);**

**select *dname,dept\_total* from *summary***

**where *dept\_total* > (select *sum(dept\_total)\*1/3* from *summary*) order by *dept\_total* desc;**

**dname dept\_total**

**RESEARCH 10875**

**Q10)** List *ename, job, sal* of the employees whose salary is equal to any one of the salary of the employee ‘SCOTT’ and ‘WARD’.

**SQL> SELECT ename, job, sal FROM emp057**

**WHERE sal IN (SELECT sal FROM emp057 WHERE ename IN ('SCOTT', 'WARD'));**

**Q11)** List *ename, job, sal* of the employees whose salary and job is equal to the employee ‘FORD’.

# SQL>

**SELECT ename, job, sal FROM emp057**

**WHERE (sal, job) = (SELECT sal, job FROM emp057 WHERE ename = ‘FORD’);**

**Q12)** List *ename, job, deptno, sal* of the employees whose job is same as ‘JONES’ and

*salary* is greater than the employee ‘FORD’.

# SQL>

**SELECT ename, job, deptno, sal FROM emp057**

**WHERE job = (SELECT job FROM emp057 WHERE ename = ‘JONES’) AND sal > (SELECT sal FROM emp057 WHERE ename = ‘FORD’);**

**Q13)** List *ename, job* of the employees who work in *deptno* 10 and his/her *job* is any one of the job in the department ‘SALES’.

# SQL>

**SELECT ename, job FROM emp057 WHERE deptno = 10 AND job IN (SELECT job FROM emp057**

**WHERE deptno = (SELECT deptno WHERE dname = ‘SALES’)**

**);**

**Q14)** Execute the following query and write the result in word

**SQL> select *job,ename,empno,deptno* from *emp057 s***

## where exists (select \* from *emp057*

**where *s.empno=mgr)***

**order by *empno*;**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **job** | **ename** | **empno** | **deptno** | **job** | **ename** | **empno** | **deptno** |
| **CLERK** | **SMITH** | **7369** | **20** | **ANALYST** | **SCOTT** | **7788** | **20** |
| **SALESMAN** | **ALLEN** | **7499** | **30** | **SALESMAN** | **TURNER** | **7844** | **30** |
| **SALESMAN** | **WARD** | **7521** | **30** | **CLERK** | **ADAMS** | **7876** | **20** |
| **MANAGER** | **JONES** | **70576** | **20** | **CLERK** | **JAMES** | **7900** | **30** |
| **SALESMAN** | **MARTIN** | **7654** | **30** | **ANALYST** | **FORD** | **7902** | **20** |
| **MANAGER** | **BLAKE** | **7698** | **30** | **CLERK** | **MILLER** | **7934** | **10** |
| **MANAGER** | **CLARK** | **7782** | **10** |  |  |  |  |

### Verified by

**Date :**

**Staff In-charge Sign :**

|  |  |  |
| --- | --- | --- |
| **Ex. No. 8** | **VIEWS** | **Date :** |

**VIEWS**

* An Imaginary table contains no data and the tables upon which a view is based are called base tables.
* Logically represents subsets of data from one or more tables

## Advantages of view

* To restrict database access
* To make complex queries easy
* To allow data independence
* To present different views of the same data

## Syntax

**CREATE [OR REPLACE] VIEW *view* [*col1 alias*, *col2 alias,*...]**

**AS *subquery***

## [ WITH CHECK OPTION [ CONSTRAINT *constraint* ] ] [ WITH READ ONLY ]

* You embed a subquery within the CREATE VIEW statement.
* The subquery can contain complex SELECT syntax.
* The subquery cannot contain an ORDER BY clause.

**Q1)** Create a view *empv10* that contains *empno, ename, job* of the employees who work in *dept* 10. Also describe the structure of the view.

**SQL> create view *empv10\_057* as**

**select *empno, ename, job* from *emp057***

**where *deptno=10;***

**SQL> desc *empv10\_057;***

**Q2)** Create a view with column aliases *empv30* that contains *empno, ename, sal* of the employees who work in *dept* 30. Also display the contents of the view.

# SQL >

**CREATE VIEW empv30\_057 AS**

**SELECT empno, ename, sal FROM emp057 WHERE deptno = 30;**

**SQL> select \* from *empv30\_057;***

## Rules for Performing DML Operations on a View

* You can perform DML operations on simple views.
* You **cannot remove** a row/ **modify** data/ **add** data if the view contains the following
  + Group functions
  + A GROUP BY clause
  + The DISTINCT keyword
* You cannot modify data in a view if it contains columns defined by expressions or it contains ROWNUM pseudo column
* You cannot add data if any NOT NULL columns in the base tables that are not selected by the view

**Q3)** Update the view *empv10* by increasing 10% salary of the employees who work as ‘CLERK’. Also confirm the modifications in *emp table.*

**SQL > update *empv10\_057* set *sal = sal+0.10\*sal***

**where *job=’CLERK’*;**

**SQL > select *empno,ename,job,sal* from emp057;**

**Q4)** Modify the view *empv10* which contains the data *empno, ename, job, sal*. Add an alias for each column name.

**SQL > create or replace view *empv10\_057 (employee\_no, employee\_name, job, salary)* as**

**select *empno, ename, job,sal* from *emp057* where *deptno=10;***

**Q5)** Using *emp* table, **c**reate a view *pay* which contains *ename, monthly\_sal, annual\_sal, deptno.*

# SQL>

**CREATE VIEW pay\_057 AS**

**SELECT ename, sal AS monthly\_sal, monthly\_sal\*12 AS annual\_sal, deptno FROM emp057;**

**Q6)** Create a view *dept\_stat* which contains *department no., department name, minimum salary, maximum salary, total salary.*

# SQL>

**CREATE VIEW dept\_stat\_057 AS**

**SELECT deptno, dname, MIN(sal), MAX(sal), SUM(sal) FROM dept057;**

## With check option

* You can ensure that DML on the view stays within the domain of the view by using the WITH CHECK OPTION.

**Q7)** Execute the following query and then try to delete the row with dept no 20. Now write in words that you understand

**SQL> create or replace view *empv20\_057***

**as select \* from *emp057* where *deptno = 20***

**with check option constraint *empv20\_ck;***

## Denying DML Operations

* You can ensure that no DML operations occur by adding the WITH READ ONLY option to your view definition.

**Q8)** Create a view *empv10* with all the details of employees who work in dept no. 10.

Also ensure that no DML operations can be done with the view.

**SQL> create or replace view *empv10\_057***

**as select \* from *emp057* where *deptno = 20***

**with read only;**

**Deleting Views**

**Syntax**

**DROP VIEW view*\_name*;**

**Q9)** Delete the view *empv20*.

**SQL> DROP VIEW empv20\_057;**

|  |  |  |
| --- | --- | --- |
| **Ex. No. 9** | **ADVANCED SELECT STATEMENTS** | **Date :** |

Questions Q1 to Q18 pertain to a database with the following tables.

**Suppliers - S (Supplyno, Name, Status, City)**

**Parts - P (Partno, Pname, Colour, Weight, City) Projects - J (Jobno, Jname, City)**

**Shipment - SPJ (Supplyno, Partno, Jobno, Qty)**

The significance of an SPJ record is that the specified supplier supplies the specified part to the specified project in the specified quantity (and the combination Supplyno-Partno- Jobno uniquely identifies such a record).

**Q1)** Get full details of all projects in London.

# SQL>

**SELECT \* FROM j\_057 WHERE city = ‘LONDON’;**

**Q2)** Get Supplyno for suppliers who supply project J1.

# SQL>

**SELECT DISTINCT spj.supplyno FROM spj\_057 JOIN j\_057 as j ON spj.jobno = j.jobno**

**Q3)** Get all part-color/part-city combinations.

# SQL>

**SELECT DISTINCT colour, city FROM p\_057;**

**Q4)** Get all Supplyno/Partno/Jobno triples such that all are co-located.

**SQL> SELECT spj.supplyno, spj.partno, spj.jobno FROM spj\_057 JOIN s\_057 ON spj.supplyno = s.supplyno**

**JOIN p ON spj.partno = p.partno JOIN j ON spj.jobno = j.jobno**

**WHERE s.city = p.city AND p.city = j.city;**

**Q5)** Get al Supplyno, Partno, Jobno triples such that they are not all co-located.

**SQL> SELECT spj.supplyno, spj.partno, spj.jobno FROM spj\_057 JOIN s ON spj.supplyno = s.supplyno**

**JOIN p\_057 ON spj.partno = p.partno JOIN j\_057 ON spj.jobno = j.jobno WHERE s.city != p.city OR p.city != j.city;**

**Q6)** Get Partno for parts supplied by a supplier in London.

**SQL> SELECT spj.partno FROM spj\_057 as spj JOIN s\_057 as s ON spj.supplyno = s.supplyno WHERE s.city = ‘LONDON’;**

**Q7)** Get all pairs of cities such that a supplier in the first city supplies to a Project in the second city.

**SQL> SELECT s.city, j.city FROM s\_057 as s JOIN spj\_057 as spj ON s.supplyno = spj.supplyno JOIN j\_057 ON j.jobno =**

**Q8)** Get Jobno for projects supplied by at least one supplier not in the same city.

# SQL>

**SELECT DISTINCT spj\_057.jobno FROM spj\_057 JOIN s\_057 ON spj\_057.supplyno = s\_057.supplyno JOIN j ON spj\_057.jobno = j\_057.jobno**

**WHERE s\_057.city <> j\_057.city;**

**Q9)** Get all pairs of part numbers such that some supplier supplies both the indicated parts.

# SQL>

**SELECT spj1\_057.partno, spj2\_057.partno FROM spj\_057 spj1\_057**

**JOIN spj\_057 spj2\_057 ON spj1\_057.supplyno = spj2\_057.supplyno AND spj1\_057.partno < spj2\_057.partno;**

**Q10)** Get the total quantity of part P1 supplied by S1.

# SQL>

**SELECT SUM(spj\_057.qty) FROM spj\_057 JOIN s\_057 on s.supplyno = spj.supplyno JOIN p on p.partno = spj.partno**

**WHERE p.pname = ‘P1’ and s.name = ‘S1’;**

**Q11)** For each part supplied to a project, get the Partno, Jobno and corresponding total quantity.

# SQL>

**SELECT partno, jobno, SUM(qty) FROM spj\_057 GROUP BY partno, jobno;**

**Q12)** Get Partno of parts supplied to some project in an average quantity > 320.

# SQL>

**SELECT partno FROM spj\_057 GROUP BY partno, jobno; HAVING AVG(qty) > 320;**

**Q13)** Get project names for projects supplied by supplier S1.

# SQL>

**SELECT j.jname FROM j\_057**

**JOIN spj\_057 ON spj.jobno = j.jobno JOIN s\_057 ON s.supplyno = spj.supplyno WHERE s.name = ‘S1’;**

**Q14)** Get colors of parts supplied by S1.

# SQL>

**SELECT DISTINCT p.colour FROM spj\_057 JOIN p\_057 ON p.partno = spj.partno**

**JOIN s\_057 ON s.supplyno = spj.supplyno WHERE s.name = ‘S1’;**

**Q15)** Get Jobno for projects using at least one part available from supplier S1.

# SQL>

**SELECT DISTINCT spj.jobno FROM spj\_057 JOIN s\_057 ON s.supplyno = spj.supplyno WHERE s.name = ‘S1’;**

**Q16)** Get supplier numbers for suppliers supplying at least one part supplied by at least one supplier who supplies at least one red part.

**SQL> SELECT DISTINCT s2.supplyno FROM spj\_057 s2\_057**

**WHERE s2.partno IN (SELECT DISTINCT spj.partno FROM spj\_057 WHERE spj.supplyno IN ( SELECT DISTINCT spj.supplyno FROM spj\_057**

**JOIN p\_057 ON spj.partno = p.partno WHERE p.colour = 'Red'**

**Q17)** Get supplier numbers for suppliers with a status lower than that of supplier S1.

**SQL> SELECT supplyno FROM s\_057**

**WHERE status < (SELECT status FROM s\_057 WHERE name = 'S1');**

**Q18)** Get project numbers for projects not supplied with any red part by any London supplier.

# SQL>

**SELECT DISTINCT spj.jobno FROM spj\_057 WHERE NOT EXISTS (**

**SELECT 1 FROM spj\_057 spj2\_057**

**JOIN p\_057 ON spj2.partno = p.partno JOIN s\_057 ON spj2.supplyno = s.supplyno**

**WHERE spj.jobno = spj2.jobno AND p.colour = 'Red' AND s.city = 'London'**

**);**

***Verified by***

**Date :**

**Staff In-charge Sign :**

|  |  |  |
| --- | --- | --- |
| **Ex. No. 10** | **ADDITIONAL QUERIES** | **Date :** |

## Solve the queries Q1 to Q32 by considering the following tables

**Table 1 : STUDIES (PNAME, SPLACE, COURSE, CCOST)**

Hint : First 3 columns are of type varchar and the 4th one is number

## Table 2 : SOFTWARE(PNAME, TITLE, DEVIN, SCOST, DCOST, SOLD)

Hint : First 3 columns are of type varchar and the last 3 are of numbers

## Table 3 : PROGRAMMER(PNAME, DOB, DOJ, SEX, PROF1, PROF2, SAL)

Hint : Pname : varchar, dob & doj : date, sex : char, prof1 & prof2 : varchar, sal : number.

# LEGEND :

PNAME – Programmer Name, SPLACE – Study Place, CCOST – Course Cost, DEVIN – Developed in, SCOST – Software Cost, DCOST – Development Cost, PROF1 – Proficiency 1

**Q1)** Find out the selling cost average for packages developed in Oracle.

**SQL> SELECT AVG(scost) FROM software\_057 WHERE devin=‘Oracle’;**

**Q2)** Display the names, ages and experience of all programmers.

**SQL> SELECT pname, TRUNC(MONTHS\_BETWEEN(SYSDATE, dob) / 12) AS age,**

**TRUNC(MONTHS\_BETWEEN(SYSDATE, doj) / 12) AS yoe FROM programmer\_057;**

**Q3)** Display the names of those who have done the PGDCA course.

**SQL> SELECT pname FROM studies\_057 WHERE course = ‘PGDCA’;**

**Q4)** What is the highest number of copies sold by a package?

**SQL> SELECT MAX(sold) FROM software\_057;**

**Q5)** Display the names and date of birth of all programmers born in April.

**SQL> SELECT pname, dob FROM programmer\_057 WHERE TO\_CHAR(dob, 'MM') = ‘04’;**

**Q6)** Display the lowest course fee.

**SQL> SELECT MIN(ccost) FROM studies\_057;**

**Q7)** How many programmers have done the DCA course.

# SQL>

**SELECT COUNT(\*) FROM studies\_057 WHERE course = ‘DCA’;**

**Q8)** How much revenue has been earned through the sale of packages developed in C.

# SQL>

**SELECT SUM(scost) FROM software\_057 WHERE devin = ‘C’;**

**Q9)** Display the details of software developed by Rakesh.

# SQL>

**SELECT \* FROM software\_057 WHERE pname = ‘Rakesh’;**

**Q10)** How many programmers studied at Pentafour.

# SQL>

**SELECT COUNT(\*) FROM studies\_057 WHERE splace = ‘Pentafour’;**

**Q11)** Display the details of packages whose sales crossed the 5000 mark.

# SQL>

**SELECT \* FROM software\_057 WHERE sold > 5000;**

**Q12)** Find out the number of copies which should be sold in order to recover the development cost of each package.

# SQL>

**SELECT CEILING(dcost / scost) FROM software\_057;**

**Q13)** Display the details of packages for which the development cost has been recovered.

# SQL>

**SELECT \* FROM software\_057 WHERE scost \* sold > dcost;**

**Q14)** What is the price of costliest software developed in VB?

# SQL>

**SELECT MAX(scost) FROM software\_057 WHERE devin = ‘VB’;**

**Q15)** How many packages were developed in Oracle ?

**SQL> SELECT COUNT(\*) FROM software\_057 WHERE devn = ‘Oracle’;**

**Q16)** How many programmers studied at PRAGATHI?

# SQL>

**SELECT COUNT(\*) FROM studies\_057 WHERE splace = ‘Pragathi’;**

**Q17)** How many programmers paid 10000 to 15000 for the course?

# SQL>

**SELECT COUNT(\*) FROM studies\_057 WHERE ccost BETWEEN 10000 AND 15000;**

**Q18)** What is the average course fee?

# SQL>

**SELECT AVG(ccost) FROM studies\_057;**

**Q19)** Display the details of programmers knowing C.

# SQL>

**SELECT \* FROM programmer\_057 WHERE prof1 = ‘C’ OR prof2 = ‘C’;**

**Q20)** How many programmers know either C or Pascal?

# SQL>

**SELECT COUNT(\*) FROM programmer\_057**

**WHERE prof1 IN ('C', 'Pascal') OR prof2 IN ('C', 'Pascal');**

**Q21)** How many programmers don’t know C and C++?

# SQL>

**SELECT COUNT(\*) FROM programmer\_057**

**WHERE prof1 NOT IN ('C', 'C++') AND prof2 NOT IN ('C', 'C++');**

**Q22)** How old is the oldest male programmer?

# SQL>

**SELECT TRUNC(MONTHS\_BETWEEN(SYSDATE, MIN(dob)) / 12) FROM programmer\_057**

**WHERE sex = ‘Male’;**

**Q23)** What is the average age of female programmers?

# SQL>SELECT AVG(TRUNC(MONTHS\_BETWEEN(SYSDATE, dob) / 12)) FROM

**programmer\_057 WHERE sex = ‘Female’;**

**Q24)** Calculate the experience in years for each programmer and display along with their names in descending order.

# SQL>

**SELECT TRUNC(MONTHS\_BETWEEN(SYSDATE, doj) / 12) AS yoe, pname ORDER BY pname DESC;**

**Q25)** Who are the programmers who celebrate their birthdays during the current month?

# SQL>

**SELECT \* FROM programmer\_057 WHERE TO\_CHAR(dob, 'MM') = TO\_CHAR(SYSDATE, 'MM');**

**Q26)** How many female programmers are there?

# SQL>

**SELECT COUNT(\*) FROM programmer\_057 WHERE sex = ‘Female’;**

**Q27)** What are the languages known by the male programmers?

# SQL>

**SELECT DISTINCT prof1 FROM programmer\_057 WHERE sex = 'Male' UNION SELECT DISTINCT prof2 FROM programmer\_057 WHERE sex = 'Male';**

**Q28)** What is the average salary?

# SQL>

**SELECT AVG(sal) FROM programmer\_057;**

**Q29)** How many people draw 5000 to 7500?

# SQL>

**SELECT COUNT(\*) FROM programmer\_057 WHERE sal BETWEEN 5000 AND 7500;**

**Q30)** Display the details of those who don’t know C, C++ or Pascal.

**SQL> SELECT \* FROM programmer\_057**

**WHERE prof1 NOT IN (‘C’, ‘C++’, ‘Pascal’) AND prof2 NOT IN (‘C’, ‘C++’, ‘Pascal’);**

**Q31)** Display the costliest package developed by each programmer.

**SQL> SELECT \* FROM software\_057 s**

**WHERE scost = (SELECT MAX(scost) FROM software\_057 WHERE pname = s.pname);**

**Q32)** Produce the following output for all the male programmers Programmer

Mr. Arvind – has 15 years of experience

**SQL> SELECT 'Mr. ' || pname || ' – has ' || TRUNC(MONTHS\_BETWEEN(SYSDATE, doj) / 12) || ' years of experience' AS programmer FROM programmer\_057 WHERE sex = ‘Male’;**

## Solve Queries from Q33 to 47 using the table *dept* and *emp*.

**Q33)** List all the employees who have at least one person reporting to them.

**SQL> SELECT \* FROM emp057 WHERE empno IN (SELECT DISTINCT mgr FROM emp057);**

**Q34)** List the employee details if and only if more than 10 employees are present in department no 10.

**SQL> SELECT \* FROM emp057**

**WHERE (SELECT COUNT(\*) FROM emp057 WHERE deptno = 10) > 10;**

**Q35)** List the name of the employees with their immediate higher authority.

**SQL> SELECT e.ename, m.ename FROM emp057 e JOIN emp057 m ON m.empno = e.mgr;**

**Q36)** List all the employees who do not manage any one.

**SQL> SELECT \* FROM emp057 WHERE empno NOT IN (SELECT DISTINCT mgr FROM**

**emp057);**

**Q37)** List the employee details whose salary is greater than the lowest salary of an employee belonging to deptno 20.

# SQL>

**SELECT \* FROM emp057 WHERE sal > (SELECT MIN(sal) FROM emp057 WHERE deptno**

**= 20);**

**Q38)** List the details of the employee earning more than the highest paid manager.

# SQL>

**SELECT \* FROM emp057 WHERE sal > (SELECT MAX(sal) FROM emp057 WHERE**

**empno IN (SELECT DISTINCT mgr FROM emp057));**

**Q057)** List the highest salary paid for each job.

# SQL>

**SELECT job, MAX(sal) FROM emp057 GROUP BY job;**

**Q40)** Find the most recently hired employee in each department.

# SQL>

**SELECT \* FROM emp057 e WHERE hiredate = (SELECT MAX(hiredate) FROM emp057 WHERE deptno = e.deptno);**

**Q41)** In which year did most people join the company? Display the year and the number of employees.

# SQL>

**SELECT TO\_CHAR(doj, 'YYYY'), COUNT(\*) FROM emp**

**GROUP BY TO\_CHAR(doj, 'YYYY') ORDER BY COUNT(\*) DESC FETCH FIRST 1 ROWS ONLY;**

**Q42)** Which department has the highest annual remuneration bill?

# SQL>

**SELECT deptno, SUM(sal \* 12) FROM emp057 GROUP BY deptno ORDER BY SUM(sal \* 12) DESC**

**FETCH FIRST 1 ROWS ONLY;**

**Q43)** Write a query to display a ‘\*’ against the row of the most recently hired employee.

**SQL> SELECT \*,**

**CASE**

**WHEN hiredate = (SELECT MAX(hiredate) FROM emp057) THEN '\*' ELSE ''**

**END**

**FROM emp057;**

**Q44)** Write a correlated sub-query to list out the employees who earn more than the average salary of their department.

# SQL>

**SELECT \* FROM emp057 e**

**WHERE sal > (SELECT AVG(sal) FROM emp057 WHERE deptno = e.deptno);**

**Q45)** Find the nth maximum salary.

# SQL>

**SELECT sal FROM (SELECT sal, ROWNUM FROM emp057 ORDER BY sal DESC)**

**WHERE rnum = n;**

**Q46)** Select the duplicate records (Records, which are inserted, that already exist) in the EMP table.

# SQL>

**SELECT \* FROM emp057**

**GROUP BY empno, ename, job, mgr, hiredate, sal, comm, deptno HAVING COUNT(\*) > 1;**

**Q47)** Write a query to list the length of service of the employees (of the form n years and m months).

**SQL> SELECT empno, ename, FLOOR(MONTHS\_BETWEEN(SYSDATE, hiredate) / 12) ||**

**‘ years and ’ || MOD(MONTHS\_BETWEEN(SYSDATE, hiredate), 12) ‘ months’ FROM emp057;**

**Q48)** Create the following tables with constraints specified below.

### customer (cust\_id, cust\_name, annual\_revenue, cust\_type)

Constraints are :

*cust\_id* must be between 100 and 10,000

*annual\_revenue* defaults to Rs. 20,000

*cust\_type* must be manufacturer, wholesaler, or retailer

**CREATE TABLE customer\_057 (**

# SQL>

**cust\_id INT CHECK (cust\_id BETWEEN 100 AND 10000), cust\_name VARCHAR(255),**

**annual\_revenue INT DEFAULT 20000,**

**cust\_type VARCHAR(50) CHECK (cust\_type IN ('manufacturer', 'wholesaler', 'retailer'))**

**);**

### shipment (shipment\_no, cust\_id, weight, truck\_no, destination, ship\_date)

Constraints are :

foreign key: cust\_id references customer, on deletion cascade foreign key: truck\_# references truck, on deletion set to null foreign key: destination references city, on deletion set to null weight must be under 1000 and defaults to 10

**CREATE TABLE shipment\_057 ( shipment\_no INT PRIMARY KEY,**

**weight INT DEFAULT 10 CHECK (weight < 1000),**

**SQL> cust\_id INT, truck\_no INT, destination VARCHAR(255), ship\_date DATE,**

**CONSTRAINT fk\_cust\_id FOREIGN KEY (cust\_id) REFERENCES customer(cust\_id) ON DELETE CASCADE,**

**CONSTRAINT fk\_truck\_no FOREIGN KEY (truck\_no) REFERENCES truck(truck\_no) ON DELETE SET NULL, CONSTRAINT fk\_destination FOREIGN KEY (destination) REFERENCES city(city\_name) ON DELETE SET NULL**

**);**

1. ***truck (truck\_no, driver\_name)***

# SQL>

**CREATE TABLE truck\_057 ( truck\_no INT PRIMARY KEY, driver\_name VARCHAR(255)**

**);**

1. ***city (city\_name, population)***

# SQL>

**CREATE TABLE truck\_057 (**

**city\_name VARCHAR(255) PRIMARY KEY,**

**population INT**

**);**

## Answer Questions Q49 to Q67 using the tables created in Q48.

**Q49)** What are the names of customers who have sent packages (shipments) to Sioux City?

**SQL> SELECT c.cust\_name FROM customer\_057 c JOIN shipment\_057 s ON c.cust\_id = s.cust\_id WHERE s.destination = 'Sioux City';**

**Q50)** To what destinations have companies with revenue less than $1 million sent packages?

# SQL>

**SELECT DISTINCT s.destination FROM shipment\_057 s JOIN customer c ON s.cust\_id = c.cust\_id**

**WHERE c.annual\_revenue < 1000000;**

**Q51)** What are the names and populations of cities that have received shipments weighing over 100 pounds?

# SQL>

**SELECT DISTINCT ci.city\_name, ci.population FROM city\_057 ci JOIN shipment\_057 s ON s.destination = ci.city\_name**

**WHERE s.weight > 100;**

**Q52)** Who are the customers having over $5 million in annual revenue who have sent shipments weighing less than 1 pound?

# SQL>

**SELECT \* FROM customer\_057 c**

**JOIN shipment\_057 s ON s.cust\_id = c.cust\_id**

**WHERE c.annual\_revenue > 5000000 AND s.weight < 1;**

**Q53)** Who are the customers having over $5 million in annual revenue who have sent shipments weighing less than 1 pound or have sent a shipment to San Francisco?

# SQL>

**SELECT \* FROM customer\_057 c JOIN shipment\_057 s ON s.cust\_id =**

**c.cust\_id WHERE c.annual\_revenue > 5000000**

**Q54)** Who are the drivers who have delivered shipments for customers with annual revenue over $20 million to cities with populations over 1 million?

# SQL>

**SELECT DISTINCT t.driver\_name FROM shipment\_057 s JOIN truck t ON s.truck\_no = t.truck\_no**

**JOIN customer\_057 c ON s.cust\_id = c.cust\_id JOIN city\_057 ci ON s.destination = ci.city\_name**

**WHERE c.annual\_revenue > 20000000 AND ci.population > 1000000;**

**Q55)** List the cities that have received shipments from customers having over $15 million in annual revenue.

# SQL>

**SELECT DISTINCT s.destination FROM shipment\_057 s JOIN customer\_057 c ON s.cust\_id = c.cust\_id**

**WHERE c.annual\_revenue > 15000000;**

**Q057)** List the names of drivers who have delivered shipments weighing over 100 pounds.

# SQL>

**SELECT DISTINCT t.driver\_name FROM shipment\_057 s JOIN truck\_057 t ON s.truck\_no = t.truck\_no**

**WHERE s.weight > 100;**

**Q57)** List the name and annual revenue of customers who have sent shipments weighing over 100 pounds.

# SQL>

**SELECT cust\_name, annual\_revenue FROM customer\_057 c JOIN shipment\_057 s ON s.cust\_id = c.cust\_id**

**WHERE s.weight > 100;**

**Q58)** List the name and annual revenue of customers whose shipments have been delivered by truck driver Jensen.

# SQL>

**SELECT cust\_name, annual\_revenue FROM customer\_057 c JOIN shipment\_057 s ON s.cust\_id = c.cust\_id**

**JOIN truck\_057 t ON s.truck\_no = t.truck\_no WHERE t.driver\_name = ‘Jensen’;**

**Q59)** List customers who had shipments delivered by every truck. ( use NOT EXISTS)

# SQL>

**SELECT c.cust\_name FROM customer\_057 c**

**WHERE NOT EXISTS (SELECT 1 FROM truck\_057 t WHERE NOT EXISTS (**

**SELECT 1 FROM shipment\_057 s**

**WHERE s.cust\_id = c.cust\_id AND s.truck\_no = t.truck\_no**

**));**

**Q60)** List cities that have received shipments from every customer. ( use NOT

EXISTS)

**SQL> SELECT ci.city\_name FROM city\_057 ci**

**WHERE NOT EXISTS (SELECT 1 FROM customer\_057 c WHERE NOT EXISTS (**

**SELECT 1 FROM shipment\_057 s**

**WHERE s.cust\_id = c.cust\_id AND s.destination = ci.city\_name**

**Q61)** List drivers who have delivered shipments to every city. (use NOT EXISTS)

# SQL>

**SELECT DISTINCT t.driver\_name FROM truck\_057 t**

**WHERE NOT EXISTS (SELECT 1 FROM city\_057 ci WHERE NOT EXISTS (**

**SELECT 1 FROM shipment\_057 s**

**WHERE s.destination = ci.city\_name AND s.truck\_no = t.truck\_no**

**));**

**Q62)** Customers who are manufacturers or have sent a package to St. Louis.

# SQL>

**SELECT \* FROM customer\_057 c**

**JOIN shipment\_057 s ON s.cust\_id = c.cust\_id**

**WHERE c.cust\_type = ’manufacturer’ AND s.destination = ‘St. Louis’;**

**Q63)** Cities of population over 1 million which have received a 100-pound package from customer 311.

# SQL>

**SELECT \* FROM city\_057 ci**

**JOIN shipment\_057 s ON s.destination = ci.city\_name**

**WHERE ci.population > 1000000 AND s.weight > 100 AND s.cust\_id = 311;**

**Q64)** Trucks driven by Jake Stinson which have never delivered a shipment to Denver.

# SQL>

**SELECT t.truck\_no FROM truck\_057 t WHERE t.driver\_name = 'Jake Stinson' AND NOT EXISTS (**

**SELECT 1 FROM shipment\_057 s WHERE s.truck\_no = t.truck\_no AND c.city\_name = 'Denver'**

**);**

**Q65)** Customers with annual revenue over $10 million which have sent packages under 1 pound to cities with population less than 10,000.

# SQL>

**SELECT \* FROM customer\_057 c**

**JOIN shipment\_057 s ON s.cust\_id = c.cust\_id JOIN city\_057 ci ON s.destination = ci.city\_name**

**WHERE c.annual\_revenue > 10000000 AND s.weight < 1 AND ci.population < 10000;**

**Q66)** Create views for each of the following:

1. Customers with annual revenue under $1 million.
2. Customers with annual revenue between $1 million and $5 million.
3. Customers with annual revenue over $5 million.

# SQL> SQL> SQL>

**CREATE VIEW revenue\_lt\_1m\_057 AS**

**SELECT \* FROM customer\_057 WHERE annual\_revenue < 1000000;**

**CREATE VIEW revenue\_b\_1m\_5m\_057 AS**

**SELECT \* FROM customer\_057 WHERE annual\_revenue BETWEEN 1000000 AND 5000000;**

**CREATE VIEW revenue\_gt\_1m\_057 AS**

**SELECT \* FROM customer\_057 WHERE annual\_revenue > 1000000;**

**Q67)** Use these views to answer the following queries:

1. Which drivers have taken shipments to Los Angeles for customers with revenue over $5 million?
2. What are the populations of cities which have received shipments from customers with revenue between $1 million and $5 million?
3. Which drivers have taken shipments to cities for customers with revenue under $1 million, and what are the populations of those cities?

**SELECT DISTINCT t.driver\_name FROM truck\_057 t JOIN shipment\_057 s ON s.truck\_no**

# SQL> SQL> SQL>

### Verified by

**t.truck\_no JOIN revenue\_gt\_5m\_057 c ON s.cust\_id = c.cust\_id WHERE s.destination = 'L Angeles';**

**SELECT ci.population FROM city\_057 ci**

**JOIN shipment\_057 s ON s.destination = ci.city\_name JOIN revenue\_b\_1m\_5m\_057 c ON s.cust\_id = c.cust\_id;**

**SELECT DISTINCT t.driver\_name, ci.population FROM truck\_057 t JOIN shipment\_057 s ON s.truck\_no = t.truck\_no**

**JOIN customers\_under\_1m\_057 c ON s.cust\_id = c.cust\_id JOIN city\_057 ci ON s.destination = ci.city\_name;**

**Date :**

**Staff In-charge Sign :**

# P L / S Q L I N T R O D U C T I O N

**PL/SQL**

* PL/SQL bridges the gap between database technology and procedural programming languages.
* PL/SQL uses the facilities of the sophisticated RDBMS and extends the standard SQL database language
* Not only PL/SQL allow you to insert, delete, update and retrieve data, it lets you use procedural techniques such as looping and branching to process the data.
* Thus PL/SQL provides the data manipulating power of SQL with the data processing power of procedural languages

## Advantage of PL/SQL

PL/SQL is a completely portable, high performance transaction processing language. It provides the following advantages :

* + Procedural Capabilities
    - It supports many of constructs like constants, variable, selection and iterative statements
  + Improved Performance
    - Block of SQL statements can be executed at a time
  + Enhanced Productivity
    - PL/SQL brings added functionality to non procedural tools such as SQL Forms.
  + Portability
    - PL/SQL can run anywhere the RDBMS can run
  + Integration with RDBMS
    - Most PL/SQL variables have data types native to the RDBMS data dictionary. Combined with the direct access to SQL, these native data type declarations allow easy integration of PL/SQL with RDBMS.

## Character Set

It is either ASCII or EBCDIC format

## Identifiers

It begins with a letter and can be followed by letters, numbers, $ or #. Maximum size is 30 characters in length.

## Variable Declaration

The data types (number, varchar2, real, date, …) discussed in SQL are all applicable in PL/SQL.

Ex. Salary *Number(7,2);* Sex *Boolean;* Count *smallint :=0;*

Tax *number default 750;*

Name *varchar2(20) not null;*

## Constant declaration

Ex. Phi *Constant Number(7,2)* := 3.1417;

## Comment

Line can be commented with double hyphen at the beginning of the line. Ex. - - This is a comment line

## Assignments

Variable assignment sets the current value of a variable. You can assign values to a variable as follows

1. Assignment operator (:=)

Ex. d := b\*b – 4\*a\*c;

1. Select … into statement

Ex. *Select* sal *into* salary *from* emp *where* empno=7655;

## Operators

Operators used in SQL are all applicable to PL/SQL also.

## Block Structure

PL/SQL code is grouped into structures called blocks. If you create a stored procedure or package, you give the block of PL/SQL code a name. If the block of PL/SQL code is not given a name, then it is called an anonymous block.

The PL/SQL block divided into three section: declaration section, the executable section and the exception section

The structure of a typical PL/SQL block is shown in the listing:

***declare***

< declaration section >

***begin***

< executable commands>

***exception***

<exception handling>

***end;***

53

*Declaration Section :*

Defines and initializes the variables and cursor used in the block

*Executable commands :*

Uses flow-control commands (such as IF command and loops) to execute the commands and assign values to the declared variables

*Exception handling :*

Provides handling of error conditions

## Declaration Using attributes

### %type attribute

The %TYPE attribute provides the data type of a variable, constant, or database column. Variables and constants declared using %TYPE are treated like those declared using a data type name.

For example in the declaration below, PL/SQL treats debit like a REAL(7,2) variable.

credit *REAL(7,2);*

debit credit*%TYPE;*

The %TYPE attribute is particularly useful when declaring variables that refer to database columns. You can reference a table and column, or you can reference an owner, table, and column.

my\_dname dept.dname%TYPE;

Using %TYPE to declare my\_dname has two advantages.

* First, you need not know the exact datatype of dname.
* Second, if the database definition of dname changes, the datatype of my\_dname changes accordingly at run time.

### %rowtype attribute

The %ROWTYPE attribute provides a record type that represents a row in a table (or view). The record can store an entire row of data selected from the table or fetched by a cursor.

DECLARE

emp\_rec emp%ROWTYPE;

... BEGIN

SELECT \* INTO emp\_rec FROM emp WHERE ...

...

END;

Columns in a row and corresponding fields in a record have the same names and data types.

The column values returned by the SELECT statement are stored in fields. To reference a field, you use the dot notation.

IF emp\_rec.deptno = 20 THEN ...

In addition, you can assign the value of an expression to a specific field. emp\_rec.ename := 'JOHNSON';

A %ROWTYPE declaration cannot include an initialization clause. However, there are two ways to assign values to all fields in a record at once.

First, PL/SQL allows aggregate assignment between entire records if their declarations refer to the same table or cursor.

DECLARE

dept\_rec1 dept%ROWTYPE; dept\_rec2 dept%ROWTYPE;

…..

BEGIN

.. …..

dept\_rec1 := dept\_rec2;

…..

END;

Second, you can assign a list of column values to a record by using the SELECT and FETCH statement, as the example below shows. The column names must appear in the order in which they were defined by the CREATE TABLE or CREATE VIEW statement.

DECLARE

dept\_rec dept%ROWTYPE;

….

BEGIN

SELECT deptno, dname, loc INTO dept\_rec FROM dept WHERE deptno = 30;

….

END;

However, you cannot assign a list of column values to a record by using an assignment statement. Although you can retrieve entire records, you cannot insert them.

For example, the following statement is illegal:

INSERT INTO dept VALUES (dept\_rec); -- illegal

## Creating and Executing PL/SQL Programs

Edit your PL/SQL program in your favourite editor as text file.

Execute the following command once for a session to get displayed the output.

SQL> *set serveroutput on;*

Now execute the program using the following command. SQL> *start* filename; (or) SQL> @filename;

Note : Give absolute path of the filename if you saved the file in some directory.

Ex. SQL> *start* z:\plsql\ex11; (or) SQL> @ z:\plsql\ex11;

## Control Structures

### IF Statements

There are three forms of IF statements: IF-THEN, IF-THEN-ELSE, and IF- THEN-ELSIF. The third form of IF statement uses the keyword ELSIF (NOT ELSEIF) to introduce additional conditions, as follows:

IF condition1 THEN

sequence\_of\_statements1;

ELSIF condition2 THEN sequence\_of\_statements2;

ELSE

sequence\_of\_statements3;

END IF;

### LOOP and EXIT Statements

There are three forms of LOOP statements. They are LOOP, WHILE-LOOP, and FOR-LOOP.

### LOOP

The simplest form of LOOP statement is the basic (or infinite) loop, which encloses a sequence of statements between the keywords LOOP and END LOOP, as follows:

LOOP

sequence\_of\_statements3;

...

END LOOP;

With each iteration of the loop, the sequence of statements is executed, then control resumes at the top of the loop. If further processing is undesirable or impossible, you can use the EXIT statement to complete the loop. You can place one or more EXIT statements anywhere inside a loop, but nowhere outside a loop. There are two forms of EXIT statements: EXIT and EXIT-WHEN.

The EXIT statement forces a loop to complete unconditionally. When an EXIT statement is encountered, the loop completes immediately and control passes to the next statement.

LOOP

...

IF ... THEN

...

EXIT; -- exit loop immediately END IF;

END LOOP;

-- control resumes here

The EXIT-WHEN statement allows a loop to complete conditionally. When the EXIT statement is encountered, the condition in the WHEN clause is evaluated. If the condition evaluates to TRUE, the loop completes and control passes to the next statement after the loop.

LOOP

….

EXIT WHEN i>n; -- exit loop if condition is true

….

END LOOP;

….

Until the condition evaluates to TRUE, the loop cannot complete. So, statements within the loop must change the value of the condition.

Like PL/SQL blocks, loops can be labeled. The label, an undeclared identifier enclosed by double angle brackets, must appear at the beginning of the LOOP statement, as follows:

<<label\_name>> LOOP

sequence\_of\_statements;

...

END LOOP [label\_name];

Optionally, the label name can also appear at the end of the LOOP statement.

With either form of EXIT statement, you can complete not only the current loop, but any enclosing loop. Simply label the enclosing loop that you want to complete, then use the label in an EXIT statement.

<<outer>> LOOP

... LOOP

...

EXIT outer WHEN ... -- exit both loops

END LOOP;

...

END LOOP outer;

### WHILE-LOOP

The WHILE-LOOP statement associates a condition with a sequence of statements enclosed by the keywords LOOP and END LOOP, as follows:

WHILE condition LOOP sequence\_of\_statements;

...

END LOOP;

Before each iteration of the loop, the condition is evaluated. If the condition evaluates to TRUE, the sequence of statements is executed, then control resumes at the top of the loop. If the condition evaluates to FALSE or NULL, the loop is bypassed and control passes to the next statement. Since the condition is tested at the top of the loop, the sequence might execute zero times.

### FOR-LOOP

Whereas the number of iteration through a WHILE loop is unknown until the loop completes, the number of iterations through a FOR loop is known before the loop is entered. FOR loops iterate over a specified range of integers. The range is part of an iteration scheme, which is enclosed by the keywords FOR and LOOP.

FOR counter IN [REVERSE] lower\_bound**..**upper\_bound LOOP sequence\_of\_statements;

...

END LOOP;

The lower bound need not be 1. However, the loop counter increment (or decrement) must be 1. PL/SQL lets you determine the loop range dynamically at run time, as the following example shows:

SELECT COUNT(empno) INTO emp\_count FROM emp; FOR i IN 1..emp\_count LOOP

...

END LOOP;

The loop counter is defined only within the loop. You cannot reference it outside the loop. You need not explicitly declare the loop counter because it is implicitly declared as a local variable of type INTEGER.

The EXIT statement allows a FOR loop to complete prematurely. You can complete not only the current loop, but any enclosing loop.

### GOTO and NULL statements

Unlike the IF and LOOP statements, the GOTO and NULL statements are not crucial to PL/SQL programming. The structure of PL/SQL is such that the GOTO statement is seldom needed. Occasionally, it can simplify logic enough to warrant its use. The NULL statement can make the meaning and action of conditional statements clear and so improve readability.

BEGIN

...

GOTO insert\_row;

...

<<insert\_row>>

INSERT INTO emp VALUES ...

END;

A GOTO statement cannot branch into an IF statement, LOOP statement, or sub- block. A GOTO statement cannot branch from one IF statement clause to another. A GOTO statement cannot branch out of a subprogram. Finally, a GOTO statement cannot branch from an exception handler into the current block.

The NULL statement explicitly specifies inaction; it does nothing other than pass control to the next statement. It can, however, improve readability. Also, the NULL statement is a handy way to create stubs when designing applications

from the top down.

**\* \* \***

|  |  |  |
| --- | --- | --- |
| **Ex. No. 11** | **B A S I C P L / S Q L** | **Date :** |

**Q1)** Write a PL/SQL Block to find the maximum of 3 Numbers Declare

a number;

b number;

c number;

Begin

dbms\_output.put\_line('Enter a:'); a:=&a; dbms\_output.put\_line('Enter b:'); b:=&b; dbms\_output.put\_line('Enter c:'); c:=&c;

if (a>b) and (a>c) then

dbms\_output.putline('A is Maximum'); elsif (b>a) and (b>c) then

dbms\_output.putline('B is Maximum');

End;

/

else end if;

dbms\_output.putline('C is Maximum');

**Q2)** Write a PL/SQL Block to find the sum of odd numbers upto 100 using loop statement

Declare

**sum NUMBER := 0; i NUMBER := 1;**

Begin

**WHILE i <= 100 LOOP**

**sum := sum + i; i := i + 2;**

**END LOOP;**

**DBMS\_OUTPUT.PUT\_LINE('Sum of odd numbers up to 100: ' || sum\_odd);**

End;

/

**Q3)** Write a PL/SQL block to get the salary of the employee who has empno=7369 and update his salary as specified below

* if his/her salary < 2500, then increase salary by 25%
* otherwise if salary lies between 2500 and 5000, then increase salary by 20%
* otherwise increase salary by adding commission amount to the salary.

Declare

Salary number(5);

Begin

End;

/

Select sal into salary from emp where empno=7369;

*-- complete remaining statements*

**IF (salary < 2500) THEN**

**UPDATE emp057 SET sal = salary \* 1.25 WHERE empno = 7369; ELSIF (salary BETWEEN 2500 AND 500 THEN**

**UPDATE emp057 SET sal = salary \* 1.20 WHERE empno = 7369; ELSE**

**UPDATE emp057 SET sal = salary + NVL(comm, 0) WHERE empno = 7369; END IF;**

**Q4)** Write a PL/SQL Block to modify the department name of the department 71 if it is not ‘HRD’.

Declare

deptname dept.dname%type;

Begin -- complete the block

**SELECT dname INTO deptname FROM dept057 WHERE deptno = 71; IF (dname <> ‘HRD’) THEN**

**UPDATE dept057 SET dname = ‘HRD’ WHERE deptno = 71; END IF;**

End;

/

**C U R S O R**

A cursor is a temporary work area created in the system memory when a SQL statement is executed. A cursor contains information on a select statement and the rows of data accessed by it. This temporary work area is used to store the data retrieved from the database, and manipulate this data. A cursor can hold more than one row, but can process only one row at a time.

There are two types of cursors in PL/SQL. They are Implicit cursors and Explicit cursors.

## Implicit cursors

These are created by default when DML statements like, INSERT, UPDATE, and DELETE statements are executed.

Oracle provides few attributes called as implicit cursor attributes to check the status of DML operations. The cursor attributes available are %FOUND,

%NOTFOUND, %ROWCOUNT, and %ISOPEN.

For example, When you execute INSERT, UPDATE, or DELETE statements the cursor attributes tell us whether any rows are affected and how many have been affected.

When a SELECT... INTO statement is executed in a PL/SQL Block, implicit cursor attributes can be used to find out whether any row has been returned by the SELECT statement. PL/SQL returns an error when no data is selected.

### Implicit Cursor Attributes

%FOUND

The return value is TRUE, if the DML statements like INSERT, DELETE and UPDATE affect at least one row or if SELECT ….INTO statement return at least one row. Ex. SQL%FOUND

%NOTFOUND

The return value is FALSE, if DML statements affect at least one row or if SELECT. …INTO statement return at least one row.

Ex. SQL%NOTFOUND

%ROWCOUNT

Return the number of rows affected by the DML operations Ex. SQL%ROWCOUNT

**Q5)** Write a PL/SQL Block, to update salaries of all the employees who work in deptno 20 by 15%. If none of the employee’s salary are updated display a message *'None of the salaries were updated'*. Otherwise display the total number of employee who got salary updated.

Declare

num number(5);

Begin

update emp set sal = sal + sal\*0.15 where deptno=20; if SQL%NOTFOUND then

dbms\_output.put\_line('none of the salaries were upated');

elsif SQL%FOUND then

num := SQL%ROWCOUNT;

dbms\_output.put\_line('salaries for ' || num || 'employees are updated'); end if;

End;

## Explicit cursors

They must be created when you are executing a SELECT statement that returns more than one row. Even though the cursor stores multiple records, only one record can be processed at a time, which is called as current row. When you fetch a row the current row position moves to next row.

There are four steps in using an Explicit Cursor.

* DECLARE the cursor in the declaration section.
* OPEN the cursor in the Execution Section.
* FETCH the data from cursor into PL/SQL variables or records in the Execution Section.
* CLOSE the cursor in the Execution Section before you end the PL/SQL Block.

*Declaring Cursor* :

**CURSOR cursor\_name IS select\_statement;**

*Opening Cursor* :

**OPEN cursor\_name;**

*Fetching Cursor* :

**FETCH cursor\_name INTO variable-list/record-type;**

*Closing Cursor* :

**CLOSE cursor\_name;**

### Explicit Cursor Attributes

%FOUND

TRUE, if fetch statement returns at least one row. Ex. Cursor\_name%FOUND

%NOTFOUND

TRUE, , if fetch statement doesn’t return a row. Ex. Cursor\_name%NOTFOUND

%ROWCOUNT

The number of rows fetched by the fetch statement. Ex. Cursor\_name%ROWCOUNT

%ISOPEN

TRUE, if the cursor is already open in the program. Ex. Cursor\_name%ISOPEN

**Q6)** Create a table *emp\_grade* with columns *empno & grade*. Write PL/SQL block to insert values into the table *emp\_grade* by processing *emp* table with the following constraints.

If sal <= 1400 then grade is ’C’

Else if sal between 1401 and 2000 then the grade is ‘B’ Else the grade is ‘A’.

**SQL>** create table emp\_grade\_057(empno number, grade char(1)); Declare Emp\_rec emp%rowtype;

Cursor c is select \* into emp\_rec from emp;

Begin

Open c;

If c%ISOPEN then Loop

Fetch c into emp\_rec;

If c%notfound then Exit; Endif; If emp\_rec.sal <= 1400 then

Insert into emp\_grade\_057 values(emp\_rec.empno,’C’); Elsif emp\_rec.sal between 1401 and 2000 then

Insert into emp\_garde\_057 values(em\_rec.empno,’B’); Else

Insert into emp\_garde\_057 values(em\_rec.empno,’A’); Endif

End loop;

Else

Open c;

Endif;

End;

**Q7)** Write a PL/SQL block to do the following :

1. Total wages of the company (Sum of the salaries and commission values of all the employees in *emp* table)
2. Total number of highly paid employees. (Employees with salary > 2000)
3. Total number of employees who get commission that is higher than their salary.

**DECLARE**

**salary NUMBER; commission NUMBER; total NUMBER := 0; highpaid NUMBER := 0;**

**comm\_gt\_sal NUMBER := 0;**

**CURSOR c IS SELECT sal, NVL(comm, 0) AS comm FROM emp057; BEGIN**

**OPEN c;**

**IF c%ISOPEN THEN LOOP**

**FETCH c INTO salary, commission;**

**IF c%NOTFOUND THEN EXIT; END IF;**

**total := total + salary + commission;**

**IF salary > 2000 THEN highpaid := highpaid + 1; END IF;**

**IF commission > salary THEN comm\_gt\_sal := comm\_gt\_sal + 1; END IF; END LOOP;**

**ELSE**

**OPEN c;**

**END IF;**

**DBMS\_OUTPUT.PUT\_LINE('Total wages of the company: ' || total); DBMS\_OUTPUT.PUT\_LINE('Number of highly paid employees (salary > 2000): ' || highpaid); DBMS\_OUTPUT.PUT\_LINE('Number of employees with commission > salary: ' || comm\_gt\_sal);**

**END;**

**/**

**Q8)** Write a PL/SQL block to find the name and salary of first five highly paid employees.

**DECLARE**

**empname emp.ename%TYPE; salary NUMBER;**

**CURSOR c IS SELECT ename, sal FROM emp057 ORDER BY sal DESC**

**FETCH FIRST 5 ROWS ONLY; BEGIN**

**DBMS\_OUTPUT.PUT\_LINE('Top 5 Highly Paid Employees:'); OPEN c;**

**IF c%ISOPEN THEN LOOP**

**FETCH c INTO empname, salary; EXIT WHEN c%NOTFOUND;**

**DBMS\_OUTPUT.PUT\_LINE('Name: ' || empname || ', Salary: ' || salary); END LOOP;**

**ELSE**

## Cursor for loop

Cursor for loop automatically opens a cursor, fetches each row and closes the cursor when all rows have been processed.

Ex. egin

Cursor s1 is select .. .. ..

*For* var *in* s1

*Loop*

-- statements ---

*End loop;*

.. .. .. .. ..

**Q9)** Solve the program in question number Q4 using cursor for…loop

**DECLARE**

**deptname dept.dname%TYPE;**

**CURSOR c IS SELECT dname FROM dept057 WHERE deptno = 71; BEGIN**

**FOR r IN c LOOP**

**deptname := r.dname;**

**IF deptname <> 'HRD' THEN**

**UPDATE dept057 SET dname = 'HRD' WHERE deptno = 71; END IF; END LOOP;**

**END;**

**/**

**Q10)** Write a PL/SQL block to find the names of employees & job and total number of employees who have more than 28 years of service in the company.(Use for loop)

**DECLARE**

**empcount NUMBER := 0; empname emp.ename%TYPE; jobname emp.job%TYPE;**

**CURSOR c IS SELECT ename, job, FLOOR(MONTHS\_BETWEEN(SYSDATE, hiredate) / 12) AS yoe FROM emp057 WHERE FLOOR(MONTHS\_BETWEEN(SYSDATE, hiredate) / 12) > 28;**

**BEGIN**

**FOR r IN c LOOP**

**empname := r.ename; jobname := r.job; empcount := empcount + 1;**

**DBMS\_OUTPUT.PUT\_LINE('Name: ' || empname || ', Job: ' || jobname); END LOOP;**

**DBMS\_OUTPUT.PUT\_LINE('No. of Employees: ' || empcount); END;**

**/**

**II. T R I G G E R**

A trigger is a PL/SQL block structure which is fired when DML statements like Insert, Delete and Update is executed on a database table. A trigger is triggered automatically when an associated DML statement is executed.

## Syntax of Trigger

**CREATE [OR REPLACE ] TRIGGER trigger\_name**

**{BEFORE | AFTER | INSTEAD OF }**

**{INSERT [OR] | UPDATE [OR] | DELETE}**

**[OF col\_name]**

**ON table\_name**

**[REFERENCING OLD AS o NEW AS n] [FOR EACH ROW]**

**WHEN (condition) BEGIN**

**-- SQL Statements**

**END;**

CREATE [OR REPLACE ] TRIGGER trigger\_name

This clause creates a trigger with the given name or overwrites an existing trigger with the same name.

BEFORE | AFTER | INSTEAD OF

This clause indicates at what time the trigger should get fired. i.e for example: before or after updating a table. INSTEAD OF is used to create a trigger on a view. Before and after cannot be used to create a trigger on a view.

INSERT [OR] | UPDATE [OR] | DELETE

This clause determines the triggering event. More than one triggering events can be used together separated by OR keyword. The trigger gets fired at all the specified triggering event.

OF col\_name

This clause is used with update triggers. This clause is used when you want to trigger an event only when a specific column is updated.

ON table\_name

This clause identifies the name of the table/view to which the trigger is associated.

REFERENCING OLD AS o NEW AS n

This clause is used to reference the old and new values of the data being changed. By default, you reference the values as **:old.column\_name** or

**:new.column\_name**. The reference names can also be changed from old (or new) to any other user-defined name. You cannot reference old values when inserting a record, or new values when deleting a record, because they do not exist.

FOR EACH ROW

This clause is used to determine whether a trigger must fire when each row gets affected ( i.e. a Row Level Trigger) or just once when the entire SQL statement is executed (i.e.statement level Trigger).

WHEN (condition)

This clause is valid only for row level triggers. The trigger is fired only for rows that satisfy the condition specified.

## Types of Triggers

There are two types of triggers based on which level it is triggered.

* *Row level trigger* : An event is triggered for each row updated, inserted or deleted.
* *Statement level trigger* : An event is triggered for each SQL statement executed.

*Before and After Triggers* : Since triggers occur because of events, they may be set to occur immediately *before* or *after* those events. Within the trigger, we are able to refer *old* and *new* values involved in transactions. *Old* refers to the data as it existed prior to the transaction. *New* refer to the data that the transaction creates.

## Q11) Before Update : Row level Trigger

Employees may get promoted and continue servicing with new designation. To maintain the job history of the employees, create a table *job\_history* with columns *empno, ename, job, pro\_date*, and create a trigger to update the table *job\_history* whenever there is an updation in *job* column of any row in *emp* table.

Create or replace trigger *job\_history\_trigger*

Before update of *job*

on *emp*

For each row Begin

Insert into *job\_history\_057* values(:old.empno,:old.ename,:old.job,sysdate);

End;

## Q12) Before Insert : Row level Trigger

Create a trigger to convert *employee name* into upper case, before we insert any row into the table *emp* with *employee name* in either case.

(Hint. :new.ename refers the value that is to be inserted)

**CREATE OR REPLACE TRIGGER uppercase\_ename\_trigger BEFORE INSERT ON emp057**

**FOR EACH ROW BEGIN**

**:NEW.ename := UPPER(:NEW.ename); END;**

## Q13) After delete : Row level Trigger

Consider tables *dept* and *deptold* with same structure. Create a trigger to move the row into second table whenever a row is removed from first table.

**CREATE OR REPLACE TRIGGER move\_deleted\_dept\_trigger AFTER DELETE ON dept057**

**FOR EACH ROW BEGIN**

**INSERT INTO deptold VALUES ROW (:OLD); END**

**Q14)** Create a trigger which will not allow you to enter duplicate or null values in column *empno* of *emp* table.

create or replace trigger *dubb*

before insert on *emp057*

for each row declare

cursor *c1* is select \* from *emp057*; *x emp057*%rowtype;

begin

fetch *c1* into *x*;

if *:new.empno = x.empno* then

dbms\_output.put\_line('*you entered duplicated no*'); elseif :new.empno is null then

dbms\_output.put\_line('*you empno is null*');

end;

nd if;

## Q15) Before Insert/Update/Delete : Statement level Trigger

Create a database trigger that allows changes to employee table only during the business hours(i.e. from 8 a.m to 5 p.m.) from Monday to Friday. There is no restriction on viewing data from the table

Create or replace trigger *time\_check* Before insert or update or delete on *emp057* Begin

if *to\_number(to\_char(sysdate,'hh24')) < 8 or to\_number(to\_char(sysdate,'hh24')) >= 17 or*

*to\_char(sysdate,'DY') = ‘SAT'' or to\_char(sysdate,'day') = 'SUN'* then raise\_application\_error(-20004,'you can access only between 8am to 5pm on Monday to Friday');

end if;

end;

## Information about Triggers

We can use the data dictionary 'USER\_TRIGGERS' to obtain information about any trigger. The below statement shows the structure of 'USER\_TRIGGERS'.

**SQL>** desc user\_triggers;

**Q16)** Find the trigger type, trigger event and table name of the trigger ‘*time\_check’.*

**SQL>** select trigger\_type, trigger\_event, table\_name

from user\_triggers where trigger\_name = ‘TIME\_CHECK’;

## Enabling and Disabling Triggers

Syntax : **ALTER TRIGGER *trigger\_name* ENABLE | DISABLE** (or)

**ALTER TABLE *table\_name* ENABLE | DISABLE ALL TRIGGERS;**

**Q17)** Disable the trigger *‘job\_history\_trigger’*

**SQL> ALTER TRIGGER job\_history\_trigger DISABLE;**

**Q18)** Disable all the triggers of *emp* table.

**SQL> ALTER TRIGGER emp057 DISABLE ALL TRIGGERS;**

**Q19)** Drop the trigger *‘dubb’*

**SQL>** drop trigger *dub*;

### Verified by

**Date :**

**Staff In-charge Sign :**

**A P P E N D I X - A**

**CODD’S RULES**

Codd’s rule provides a method for therotical evalution of a product, that claims to be a Relational Data Base Management System.

## Rule 1 : The Information Rule

All information should be represented as data values in the rows and columns of a table.

## Rule 2 : The Guaranteed Access Rule

Every item of data must be logically addressable by specifying a combination of the table name, the primary key value and the column name.

## Rule 3 : The systematic Treatment of Null values

It is fundamental to the DBMS that NULL values are supported in the representation of missing and inapplicable information. This support for null values must be consistent throughout the DBMS and independent of data types.

## Rule 4 : The database Description Rule

A description of the database is held and maintained using the same logical structures used to define the data. Thus allowing users with appropriate authority to query such information in the same way and using the same language as they would for other data in the database.

## Rule 5 : The Comprehensive Sub-Language Rule

There must be at least one unified language whose statements can be expressed as character strings confirming to some well defined syntax. It should essentially encapsulate the following :-

* Data Definition Language
* Data Manipulation Language
* View Definition Language
* Integrity Constraints
* Authorization or Data Control Language
* Transaction boundaries

## Rule 6 : View Update Rule

All views that are theoretically updateable must be updateable by the system.

## Rule 7 : The Insert, Update and Delete Rule

The capability of handling a base relation or in fact a derived relation as a single operand must hold good for all retrieve, insert, update and delete activity.

## Rule 8 : The Physical Data Independence Rule

User access to the database via terminal monitors or application programs must remain logically consistent whenever changes to the storage representation or access methods to the data are changed.

## Rule 9 : The logical Data Independence Rule

Application programs and terminal activities must remain logically unimpaired whenever information preserving changes of any kind that are theoretically permitted are made to the base tables.

## Rule 10 : Integrity Independence Rule

All integrity constraints should be stored in the system catalog or in the database as table.

## Rule 11 : Distribution Rule

According to this rule a DBMS which claims to be relational must have distribution independence.

## Rule 12 : No subversion Rule

Different levels of the language cannot bypass the integrity rules and constraints. That is, if a DBMS supports a lower level language that permits, then it should not bypass any integrity rules any constraints defined in the higher level.

**A P P E N D I X - B**

**A N S W E R F O R A D D I T I O N A L Q U E R I E S**

**Q1)** SELECT AVG(SCOST) FROM SOFTWARE WHERE DEVIN = 'ORACLE';

**Q2)** SELECT PNAME,TRUNC(MONTHS\_BETWEEN(SYSDATE,DOB)/12) "AGE", TRUNC(MONTHS\_BETWEEN(SYSDATE,DOJ)/12) "EXPERIENCE" FROM PROGRAMMER;

**Q3)** SELECT PNAME FROM STUDIES WHERE COURSE = 'PGDCA';

**Q4)** SELECT MAX(SOLD) FROM SOFTWARE;

**Q5)** SELECT PNAME, DOB FROM PROGRAMMER WHERE DOB LIKE '

%APR%';

**Q6)** SELECT MIN(CCOST) FROM STUDIES;

**Q7)** SELECT COUNT(\*) FROM STUDIES WHERE COURSE = 'DCA';

**Q8)** SELECT SUM(SCOST\*SOLD-DCOST) FROM SOFTWARE GROUP BY DEVIN HAVING DEVIN = 'C';

**Q9)** SELECT \* FROM SOFTWARE WHERE PNAME = 'RAKESH';

**Q10)** SELECT \* FROM STUDIES WHERE SPLACE = 'PENTAFOUR';

**Q11)** SELECT \* FROM SOFTWARE WHERE SCOST\*SOLD-DCOST > 5000;

**Q12)** SELECT CEIL(DCOST/SCOST) FROM SOFTWARE;

**Q13)** SELECT \* FROM SOFTWARE WHERE SCOST\*SOLD >= DCOST;

**Q14)** SELECT MAX(SCOST) FROM SOFTWARE GROUP BY DEVIN HAVING DEVIN = 'VB';

**Q15)** SELECT COUNT(\*) FROM SOFTWARE WHERE DEVIN = 'ORACLE'; **Q16)** SELECT COUNT(\*) FROM STUDIES WHERE SPLACE = 'PRAGATHI'; **Q17)** SELECT COUNT(\*) FROM STUDIES WHERE CCOST BETWEEN 10000

AND 15000;

**Q18)** SELECT AVG(CCOST) FROM STUDIES;

**Q19)** SELECT \* FROM PROGRAMMER WHERE PROF1 = 'C' OR PROF2 = 'C';

**Q20)** SELECT \* FROM PROGRAMMER WHERE PROF1 IN ('C','PASCAL') OR PROF2 IN ('C','PASCAL');

**Q21)** SELECT \* FROM PROGRAMMER WHERE PROF1 NOT IN ('C','C++') AND PROF2 NOT IN ('C','C++');

**Q22)** SELECT TRUNC(MAX(MONTHS\_BETWEEN(SYSDATE,DOB)/12)) FROM PROGRAMMER WHERE SEX = 'M';

**Q23)** SELECT TRUNC(AVG(MONTHS\_BETWEEN(SYSDATE,DOB)/12)) FROM PROGRAMMER WHERE SEX = 'F';

**Q24)** SELECT PNAME, TRUNC(MONTHS\_BETWEEN(SYSDATE,DOJ)/12) FROM PROGRAMMER ORDER BY PNAME DESC;

**Q25)** SELECT PNAME FROM PROGRAMMER WHERE TO\_CHAR(DOB,'MON')

= TO\_CHAR(SYSDATE,'MON');

**Q26)** SELECT COUNT(\*) FROM PROGRAMMER WHERE SEX = 'F';

**Q27)** SELECT DISTINCT(PROF1) FROM PROGRAMMER WHERE SEX = 'M';

**Q28)** SELECT AVG(SAL) FROM PROGRAMMER;

**Q29)** SELECT COUNT(\*) FROM PROGRAMMER WHERE SAL BETWEEN 5000 AND 7500;

**Q30)** SELECT \* FROM PROGRAMMER WHERE PROF1 NOT IN ('C','C++','PASCAL') AND PROF2 NOT IN ('C','C++','PASCAL');

**Q31)** SELECT PNAME,TITLE,SCOST FROM SOFTWARE WHERE SCOST IN (SELECT MAX(SCOST) FROM SOFTWARE GROUP BY PNAME);

**Q32)** SELECT 'Mr.' || PNAME || ' - has ' || TRUNC(MONTHS\_BETWEEN(SYSDATE,DOJ)/12) || ' years of experience' “Programmer” FROM PROGRAMMER WHERE SEX = 'M' UNION SELECT 'Ms.' || PNAME || ' - has ' || TRUNC (MONTHS\_BETWEEN

(SYSDATE,DOJ)/12) || ' years of experience' “Programmer” FROM PROGRAMMER WHERE SEX = 'F';

**Q33)** SELECT DISTINCT(A.ENAME) FROM EMP A, EMP B WHERE A.EMPNO = B.MGR; or SELECT ENAME FROM EMP WHERE EMPNO IN (SELECT MGR FROM EMP);

**Q34)** SELECT \* FROM EMP WHERE DEPTNO IN (SELECT DEPTNO FROM EMP GROUP BY DEPTNO HAVING COUNT(EMPNO)>10 AND DEPTNO=10);

**Q35)** SELECT A.ENAME "EMPLOYEE", B.ENAME "REPORTS TO" FROM EMP A, EMP B WHERE A.MGR=B.EMPNO;

**Q36)** SELECT \* FROM EMP WHERE EMPNO IN ( SELECT EMPNO FROM EMP MINUS SELECT MGR FROM EMP);

**Q37)** SELECT \* FROM EMP WHERE SAL > ( SELECT MIN(SAL) FROM EMP GROUP BY DEPTNO HAVING DEPTNO=20);

**Q38)** SELECT \* FROM EMP WHERE SAL > ( SELECT MAX(SAL) FROM EMP GROUP BY JOB HAVING JOB = 'MANAGER' );

**Q057)** SELECT JOB, MAX(SAL) FROM EMP GROUP BY JOB;

**Q40)** SELECT \* FROM EMP WHERE (DEPTNO, HIREDATE) IN (SELECT DEPTNO, MAX(HIREDATE) FROM EMP GROUP BY DEPTNO);

**Q41)** SELECT TO\_CHAR(HIREDATE,'YYYY') "YEAR", COUNT(EMPNO) "NO. OF EMPLOYEES" FROM EMP GROUP BY TO\_CHAR(HIREDATE,'YYYY') HAVING COUNT(EMPNO) = (SELECT MAX(COUNT(EMPNO)) FROM EMP GROUP BY TO\_CHAR(HIREDATE,'YYYY'));

**Q42)** SELECT DEPTNO, LPAD(SUM(12\*(SAL+NVL(COMM,0))),15) "COMPENSATION" FROM EMP GROUP BY DEPTNO HAVING SUM( 12\*(SAL+NVL(COMM,0))) = (SELECT MAX(SUM(12\*(SAL+NVL(COMM,0)))) FROM EMP GROUP BY DEPTNO);

**Q43)** SELECT ENAME, HIREDATE, LPAD('\*',8) "RECENTLY HIRED" FROM EMP WHERE HIREDATE = (SELECT MAX(HIREDATE) FROM EMP) UNION SELECT ENAME NAME, HIREDATE, LPAD(' ',15) "RECENTLY HIRED" FROM EMP WHERE HIREDATE != (SELECT MAX(HIREDATE) FROM EMP);

**Q44)** SELECT ENAME,SAL FROM EMP E WHERE SAL > (SELECT AVG(SAL) FROM EMP F WHERE E.DEPTNO = F.DEPTNO);

**Q45)** SELECT ENAME, SAL FROM EMP A WHERE &N = (SELECT COUNT (DISTINCT(SAL)) FROM EMP B WHERE A.SAL<=B.SAL);

**Q46)** SELECT \* FROM EMP A WHERE A.EMPNO IN (SELECT EMPNO FROM EMP GROUP BY EMPNO HAVING COUNT(EMPNO)>1) AND A.ROWID!=MIN (ROWID));

**Q47)** SELECT ENAME "EMPLOYEE",TO\_CHAR(TRUNC(MONTHS\_BETWEEN(SYSDATE,HIRED ATE)/12))||' YEARS '|| TO\_CHAR(TRUNC(MOD(MONTHS\_BETWEEN (SYSDATE, HIREDATE),12)))||' MONTHS ' "LENGTH OF SERVICE" FROM EMP;

**Q48)** REFER EXERCISES 1 AND 3 FOR TABLE CREATION.

**Q49)** SELECT CUST\_NAME FROM CUSTOMER WHERE CUST\_ID IN (SELECT CUST\_ID FROM SHIPMENT

WHERE DESTINATION='SIOUX CITY');

**Q50)** SELECT DESTINATION FROM SHIPMENT WHERE 1000000> (SELECT SUM(ANNUAL\_REVENUE) FROM CUSTOMER WHERE CUSTOMER.CUST\_ID = SHIPMENT.CUST\_ID GROUP BY DESTINATION);

**Q51)** SELECT DESTINATION FROM

(SELECT SUM(WEIGHT)AS SUM\_WEIGHT,DESTINATION FROM SHIPMENT GROUP BY DESTINATION) WHERE SUM\_WEIGHT>100;

**Q52)** SELECT CUST\_NAME FROM CUSTOMER,SHIPMENT WHERE CUSTOMER.CUST\_ID=SHIPMENT.CUST\_ID AND ANNUAL\_REVENUE>5000000 AND WEIGHT<1;

**Q53)** SELECT CUST\_NAME FROM CUSTOMER,SHIPMENT WHERE CUSTOMER.CUST\_ID=SHIPMENT.CUST\_ID AND ANNUAL\_REVENUE>5000000 AND

( DESTINATION='SAN FRANCISCO'OR WEIGHT<1 );

**Q54)** SELECT DRIVER\_NAME FROM TRUCK WHERE TRUCK\_# IN (SELECT TRUCK\_# FROM SHIPMENT,CUSTOMER,CITY WHERE SHIPMENT.CUST\_ID=SHIPMENT.CUST\_ID AND ANNUAL\_REVENUE>20000000 AND CITY.CITY\_NAME = SHIPMENT.DESTINATION AND POPULATION>1000000 );

**Q55)** SELECT DESTINATION FROM SHIPMENT,CUSTOMER WHERE CUSTOMER.CUST\_ID=SHIPMENT.CUST\_ID AND ANNUAL\_REVENUE>15000000;

**Q057)** SELECT DRIVER\_NAME FROM TRUCK WHERE TRUCK\_# IN (SELECT TRUCK\_# FROM SHIPMENT WHERE WEIGHT>100);

**Q57)** SELECT DISTINCT CUST\_NAME,ANNUAL\_REVENUE FROM CUSTOMER,SHIPMENT

WHERE CUSTOMER.CUST\_ID=SHIPMENT.CUST\_ID AND WEIGHT>100;

**Q58)** SELECT DISTINCT CUST\_NAME,ANNUAL\_REVENUE FROM CUSTOMER,SHIPMENT,TRUCK

WHERE CUSTOMER.CUST\_ID=SHIPMENT.CUST\_ID AND SHIPMENT.TRUCK\_#=TRUCK.TRUCK\_# AND DRIVER\_NAME='JENSEN';

**Q59)** SELECT CUST\_NAME FROM CUSTOMER

WHERE NOT EXISTS (SELECT \* FROM SHIPMENT,TRUCK WHERE SHIPMENT.CUST\_ID=CUSTOMER.CUST\_ID GROUP BY SHIPMENT.CUST\_ID

HAVING COUNT(DISTINCT SHIPMENT.TRUCK\_#)<COUNT(DISTINCT TRUCK.TRUCK\_#));

**Q60)** SELECT CITY\_NAME FROM CITY

WHERE NOT EXISTS (SELECT \* FROM SHIPMENT,CUSTOMER WHERE SHIPMENT.DESTINATION=CITY.CITY\_NAME GROUP BY SHIPMENT.DESTINATION

HAVING COUNT(DISTINCT SHIPMENT.CUST\_ID)<COUNT(DISTINCT CUSTOMER.CUST\_ID));

**Q61)** SELECT DRIVER\_NAME FROM TRUCK

WHERE NOT EXISTS (SELECT \* FROM SHIPMENT,CITY WHERE SHIPMENT.TRUCK\_#=TRUCK.TRUCK\_# GROUP BY SHIPMENT.TRUCK\_#

HAVING COUNT(DISTINCT SHIPMENT.DESTINATION)<COUNT(DISTINCT CITY.CITY\_NAME));

**Q62)** SELECT DISTINCT CUST\_NAME FROM CUSTOMER,SHIPMENT WHERE CUST\_TYPE='MANUFACTURER'OR DESTINATION='ST.LOUIS';

(or)

SELECT DISTINCT CUST\_NAME FROM CUSTOMER WHERE CUST\_TYPE='MANUFACTURER' OR CUST\_ID IN (SELECT CUST\_ID FROM SHIPMENT

WHERE DESTINATION='ST.LOUIS');

**Q63)** SELECT CITY\_NAME FROM CITY

WHERE POPULATION>1000000 AND CITY\_NAME IN (SELECT DESTINATION FROM SHIPMENT

WHERE WEIGHT=100 AND CUST\_ID=311);

**Q64)** SELECT TRUCK\_# FROM TRUCK

WHERE DRIVER\_NAME='JAKE STINSON' AND TRUCK\_# NOT IN (SELECT TRUCK\_# FROM SHIPMENT WHERE DESTINATION='DENVER');

**Q65)** SELECT CUST\_NAME FROM CUSTOMER

WHERE ANNUAL\_REVENUE>10000000 AND CUST\_ID IN

(SELECT CUST\_ID FROM SHIPMENT,CITY WHERE WEIGHT<1 AND POPULATION<10000 AND CITY\_NAME=DESTINATION);

**Q66)** a) CREATE VIEW CUST\_VIEW1 AS (SELECT \* FROM CUSTOMER

WHERE ANNUAL\_REVENUE<1000000);

b) CREATE VIEW CUST\_VIEW2 AS (SELECT \* FROM CUSTOMER

WHERE ANNUAL\_REVENUE BETWEEN 1000000 AND 5000000);

1. CREATE VIEW CUST\_VIEW3 AS (SELECT \* FROM CUSTOMER

WHERE ANNUAL\_REVENUE >5000000);

**Q67)** a) SELECT DRIVER\_NAME FROM TRUCK,CUST\_VIEW3,SHIPMENT WHERE SHIPMENT.CUST\_ID=CUST\_VIEW3.CUST\_ID

AND TRUCK.TRUCK\_#=SHIPMENT.TRUCK\_# AND DESTINATION='LOS ANGELES ';

1. SELECT CITY\_NAME,POPULATION FROM CITY WHERE CITY\_NAME IN

(SELECT DESTINATION FROM SHIPMENT,CUST\_VIEW2 WHERE SHIPMENT.CUST\_ID=CUST\_VIEW2.CUST\_ID);

1. SELECT DISTINCT DRIVER\_NAME,POPULATION,CITY\_NAME FROM TRUCK,CITY,SHIPMENT

WHERE SHIPMENT.TRUCK\_#=TRUCK.TRUCK\_#

AND SHIPMENT.CUST\_ID IN (SELECT CUST\_ID FROM CUST\_VIEW1);

# B I B L I O G R A P H Y

The following books and manuals were referred during the preparation of this work book and suggested for further reading

* SQL \* Plus User’s Guide and Reference – Oracle Corporation – 2015.
* Introduction to Oracle 9i – Instructors Guide – Oracle corporation – Nancy Greenberg, Priya Nathan – 2015.
* PL/SQL User Guide and Reference – Oracle Corporation – 2007.
* Oracle/SQL Tutorial – Michael Gertz – University of California, Davis – 2010.
* ORACLE SQL\*Plus - Prof. Richard Holowczak - City University of New York, USA.
* Database system concepts – Silberschatz, korth and Sundarshan – McGraw Hill Publishers – 6th Edition 2010.
* Peter rob, Carlos Coronel, “Database Systems – Design, Implementation, and Management”, 9th Edition, 2009, Thomson Learning, ISBN: 978-05738469685
* Date C.J, “An Introduction to Database”, 8th Edition , 2003, Addison-Wesley Pub Co, ISBN: 978-0321197849
* Raghu Ramakrishnan, Johannes Gehrke, “Database Management System”, 3rd Edition, 2007, McGraw Hill, ISBN: 978-00724605731

### Normalization

* + 1. **Unnormalized Table (UNF)**

We'll define a single table that stores multiple values in some fields (repeating groups), which violates 1NF.

Table: Rental

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Rental\_ID | Customer\_Name | Customer\_Phone | Car\_IDs | Car\_Models | Rent\_Dates | Total\_Rent |
| R001 | Alice Smith | 1234567890 | C001, C002 | Honda, Toyota | 2025-04-01, 2025-04-03 | 200 |
| R002 | Bob Johnson | 9876543210 | C003 | BMW | 2025-04-02 | 150 |

* Problems:
  + Multiple Car IDs, Models, and Rent Dates in single fields (repeating groups).
  + Not in First Normal Form.

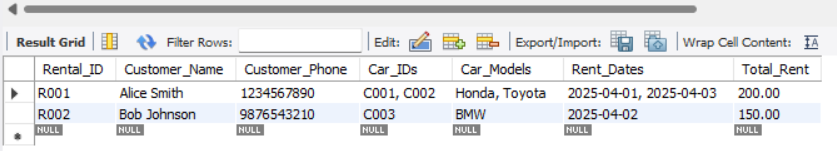


Fig 3.4.1. Unormalised Table

* + 1. **First Normal Form (1NF)**
* Eliminate repeating groups.
* Each field must have atomic (indivisible) values.

Rental Table (1NF)

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Rental\_ID | Customer\_Name | Customer\_Phone | Car\_ID | Car\_Model | Rent\_Date | Rent |
| R001 | Alice Smith | 1234567890 | C001 | Honda | 2025-04-01 | 100 |
| R001 | Alice Smith | 1234567890 | C002 | Toyota | 2025-04-03 | 100 |
| R002 | Bob Johnson | 9876543210 | C003 | BMW | 2025-04-02 | 150 |

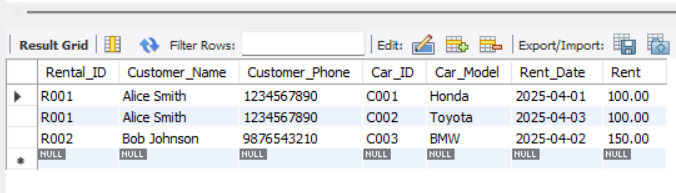
****

Fig 3.4.2 1NF Form

* + 1. **Second Normal Form (2NF)**
* Must be in 1NF.
* Remove partial dependencies (no non-key attribute should depend only on part of a composite key).

Here, Rental\_ID + Car\_ID is the composite primary key. But Customer\_Name and Customer\_Phone depend only on Rental\_ID, not Car\_ID.

So we split into:

➤ Rental Table

|  |  |  |
| --- | --- | --- |
| Rental\_ID | Customer\_Name | Customer\_Phone |
| R001 | Alice Smith | 1234567890 |
| R002 | Bob Johnson | 9876543210 |

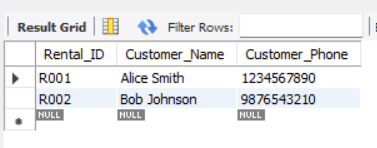
****

Fig 3.4.3. 2NF Rental Table

➤ Rental\_Detail Table

|  |  |  |  |
| --- | --- | --- | --- |
| Rental\_ID | Car\_ID | Rent\_Date | Rent |
| R001 | C001 | 2025-04-01 | 100 |
| R001 | C002 | 2025-04-03 | 100 |
| R002 | C003 | 2025-04-02 | 150 |

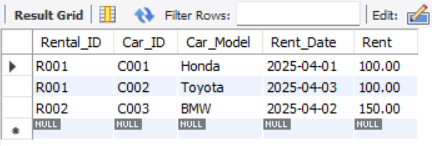


Fig 3.4.4. 2NF Rental Details Table

➤ Car Table

|  |  |
| --- | --- |
| Car\_ID | Car\_Model |
| C001 | Honda |
| C002 | Toyota |
| C003 | BMW |

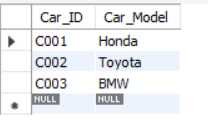


Fig 3.4.5. 2NF Car Table

* + 1. **Third Normal Form (3NF)**
* Must be in 2NF.
* Remove transitive dependencies (non-key attributes should depend only on the key).

In our case:

* All non-key attributes in every table depend only on the primary key.
* No further normalization needed.

Tables in 3NF:

➤ Rental Table

|  |  |  |
| --- | --- | --- |
| Rental\_ID | Customer\_Name | Customer\_Phone |

➤ Rental\_Detail Table

|  |  |  |
| --- | --- | --- |
| Rental\_ID | Car\_ID | Rent\_Date |

➤ Car Table

|  |  |
| --- | --- |
| Car\_ID | Car\_Model |

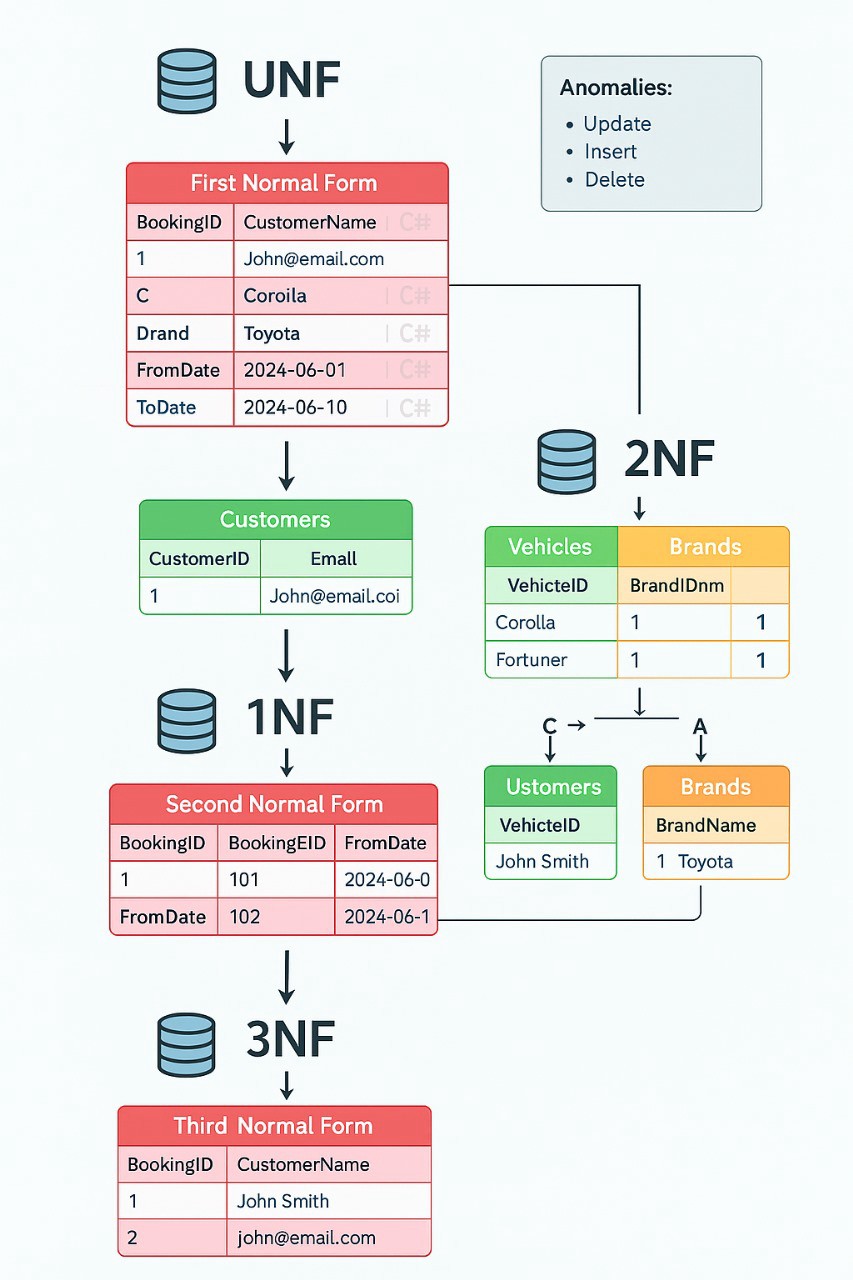


Fig 3.4.6. Normalization

### Transaction & Concurrency Control

Concurrency control ensures that multiple transactions can happen together without interfering with each other, preserving data accuracy and consistency.

It's essential to maintain:

* Atomicity (all-or-nothing)
* Consistency (valid states)
* Isolation (transactions don't interfere)
* Durability (results are permanent)

Types of Concurrency Control

* + 1. Lock-based Concurrency Control
* Transactions lock data before accessing it.
* Two main types of locks:
  + Shared Lock (S-lock): Multiple transactions can read the data.
  + Exclusive Lock (X-lock): Only one transaction can write the data.
* Two-Phase Locking (2PL):
  + Growing phase: Acquire all locks.
  + Shrinking phase: Release locks without acquiring any new ones.

Problem: Can lead to deadlocks (transactions waiting forever).

* + 1. Timestamp-based Concurrency Control
* Each transaction is given a unique timestamp when it starts.
* The DBMS uses these timestamps to order transactions.
* Older transactions get priority over newer ones.

Problem: May cause transaction rollbacks if conflicts are detected.

* + 1. Optimistic Concurrency Control
* Transactions execute freely without locking.
* Before committing, the system checks for conflicts:
  + If no conflict ➔ commit.
  + If conflict ➔ rollback and restart.

Best for: systems with low data conflict rates.

* + 1. Multiversion Concurrency Control (MVCC)
* Instead of locking, the system keeps multiple versions of data.
* Readers access an older snapshot; writers create a new version.
* Very common in databases like PostgreSQL, Oracle, MySQL (InnoDB).

Step 1: Create Tables

Car Table

CREATE TABLE Car (

Car\_ID VARCHAR(10) PRIMARY KEY,

Car\_Model VARCHAR(50),

Is\_Available BOOLEAN

);

Booking Table

CREATE TABLE Booking (

Booking\_ID SERIAL PRIMARY KEY,

Car\_ID VARCHAR(10),

Customer\_Name VARCHAR(100),

Booking\_Date DATE,

FOREIGN KEY (Car\_ID) REFERENCES Car(Car\_ID)

);

Output:

*Query OK, 0 rows affected* for both CREATE TABLE statements.  
(No actual result set, just success messages.)

Step 2: Insert Sample Data

INSERT INTO Car VALUES ('C001', 'Toyota', TRUE);

A screenshot of a computer

AI-generated content may be incorrect.

Output:

A screenshot of a computer

AI-generated content may be incorrect.  
*Query OK, 1 row affected*.

You now have one car in the database.

Step 3: See Car Table Initially

SELECT \* FROM Car;

Output:

|  |  |  |
| --- | --- | --- |
| Car\_ID | Car\_Model | Is\_Available |
| C001 | Toyota | 1 |

Step 4: Start Manual Transaction

START TRANSACTION;

Output:

*Query OK, 0 rows affected*  
Transaction begins (no visible result).

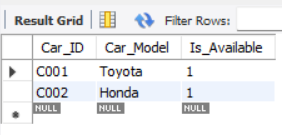
Step 5: Lock Car Row

SELECT \* FROM Car

WHERE Car\_ID = 'C001' AND Is\_Available = TRUE

FOR UPDATE;

Output:



|  |  |  |
| --- | --- | --- |
| Car\_ID | Car\_Model | Is\_Available |
| C001 | Toyota | 1 |

The system locks the C001 car row now (no one else can update it until you finish this transaction).

Step 6: Insert into Booking

INSERT INTO Booking (Car\_ID, Customer\_Name, Booking\_Date)

VALUES ('C001', 'Alice Smith', CURDATE());

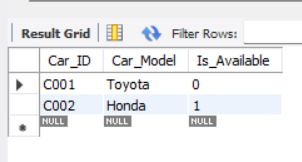
Output:

*Query OK, 1 row affected*  
New booking inserted into Booking table.

Step 7: Update Car Availability

UPDATE Car SET Is\_Available = FALSE WHERE Car\_ID = 'C001';

Output:

  
 *Query OK, 1 row affected*  
Car availability updated to not available.

Step 8: Commit Transaction

COMMIT;

Output:  
*Query OK, 0 rows affected*  
All changes saved permanently.

Step 9: Final Outputs – See Tables Now

Check Booking Table:

SELECT \* FROM Booking;

Output:

A screenshot of a computer

AI-generated content may be incorrect.

|  |  |  |  |
| --- | --- | --- | --- |
| Booking\_ID | Car\_ID | Customer\_Name | Booking\_Date |
| 1 | C001 | Alice Smith | 2025-04-27 |

Check Car Table:

SELECT \* FROM Car;

Output:

A screenshot of a computer

AI-generated content may be incorrect.

|  |  |  |
| --- | --- | --- |
| Car\_ID | Car\_Model | Is\_Available |
| C001 | Toyota | 0 |

(Is\_Available = 0 means the car is now booked and unavailable.)

# CHAPTER 4: RESULTS

The vehicle rental system has been designed and implemented to provide a comprehensive and efficient solution for managing vehicle rentals. The system facilitates seamless interaction between customers and the rental service by enabling secure customer registration, login, and profile management, including the ability to update personal and driver’s license details. Customers can search for available vehicles using multiple criteria such as vehicle category, availability dates, and pickup/drop-off locations. The search results are dynamically generated, displaying only those vehicles that are not reserved or under maintenance, ensuring accurate and real-time availability information. Once a booking is initiated, the system processes the reservation by updating the vehicle status to “Rented,” creating a reservation record, and notifying the customer through automated email or SMS confirmations. Integrated payment functionality allows customers to complete transactions securely, with the system recording payment details, generating invoices, and maintaining a full payment history for auditing and customer reference. On the administrative side, the system provides powerful fleet management tools, allowing staff to add new vehicles, update existing vehicle details, schedule and track maintenance activities, and manage vehicle insurance records. The system also includes incident reporting features to record any damages or accidents, thereby maintaining

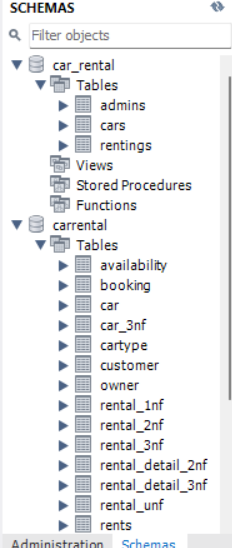
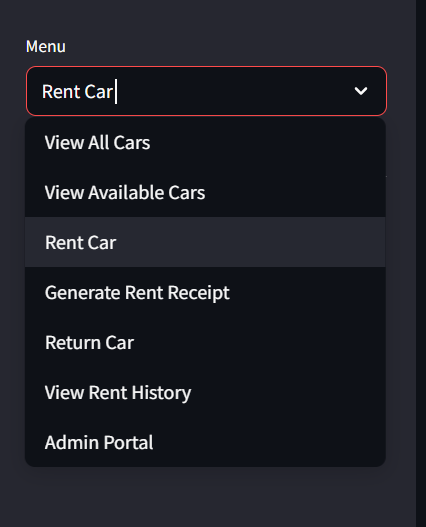
 

Fig.4.1 All Tables



Fig.4.2 All Cars in the Inventory



Fig.4.3 Available Cars for Rent

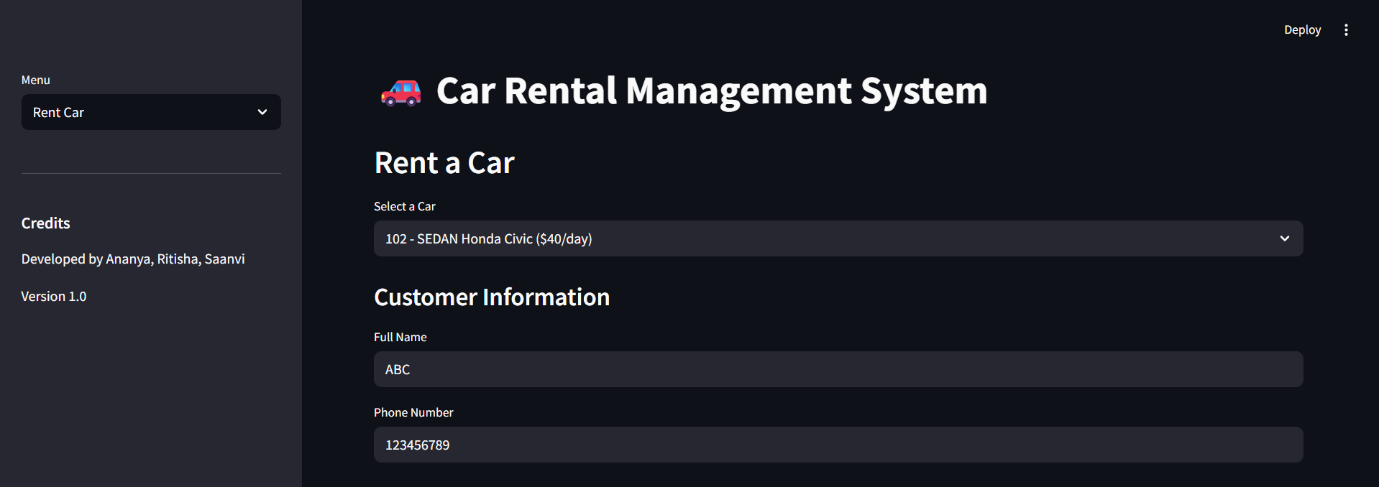
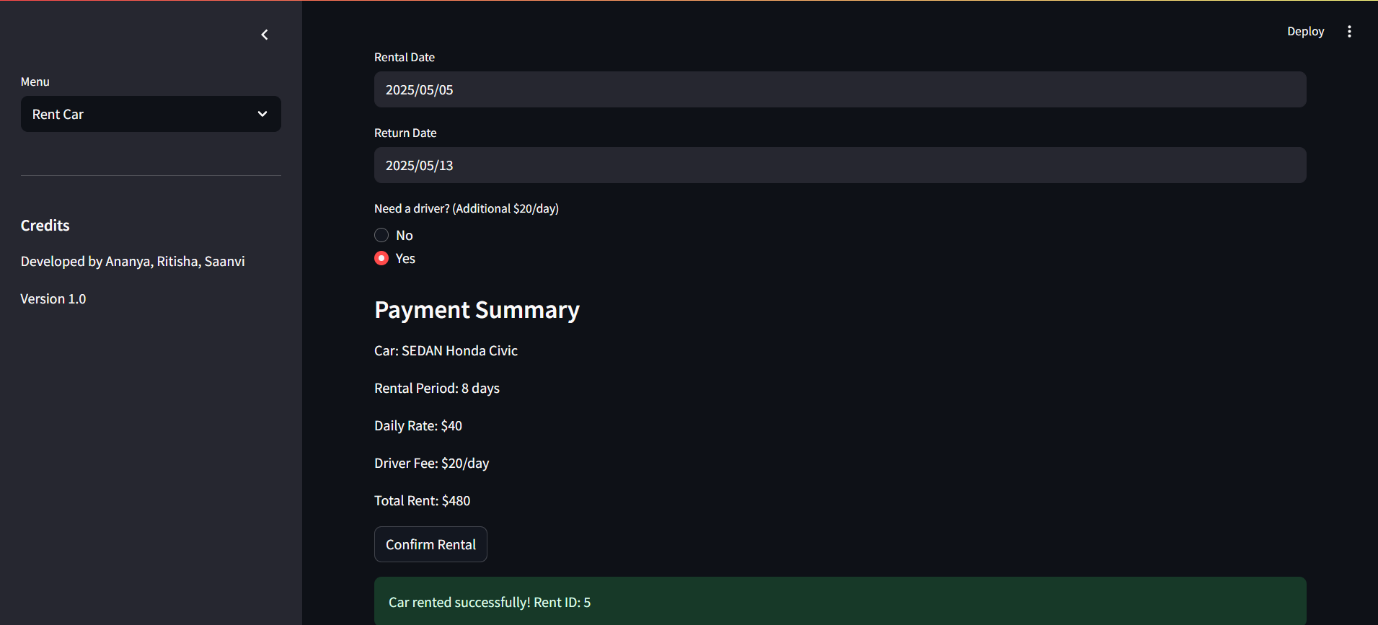


Fig.4.4. Rent a car



Fig.4.5. Rent Receipt



Fig.4.6. Returning a Rented Car

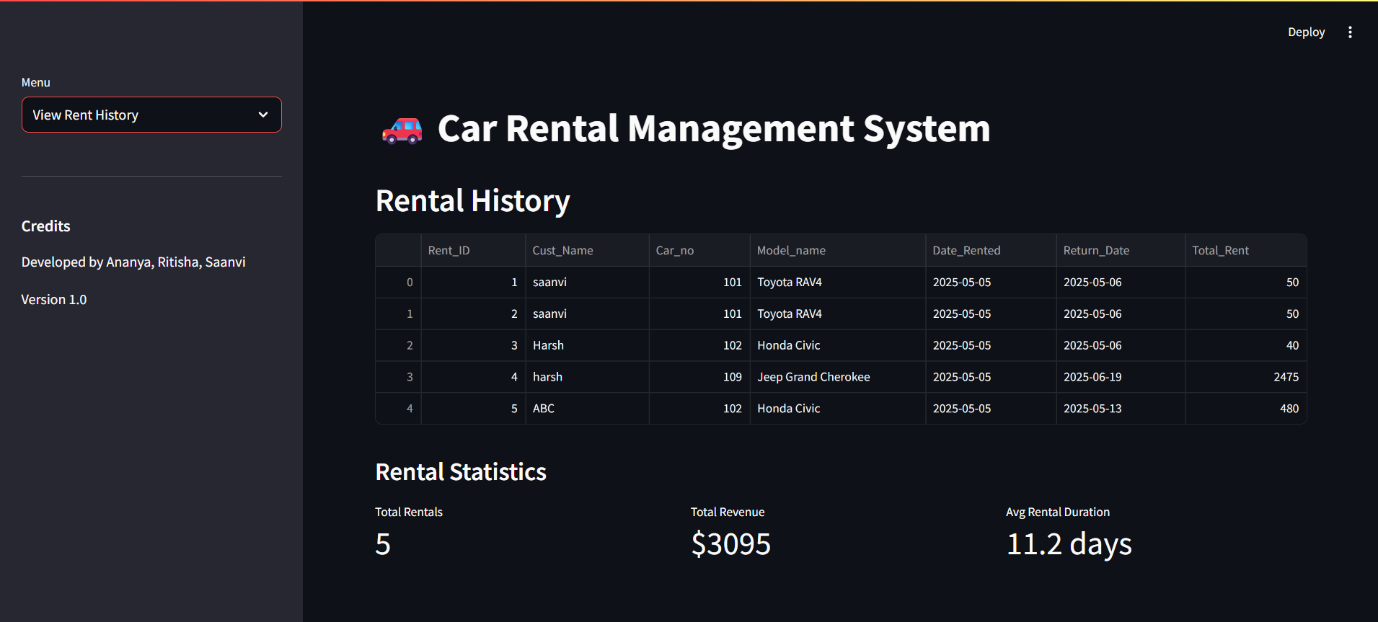


Fig.4.7. Rental Statistics

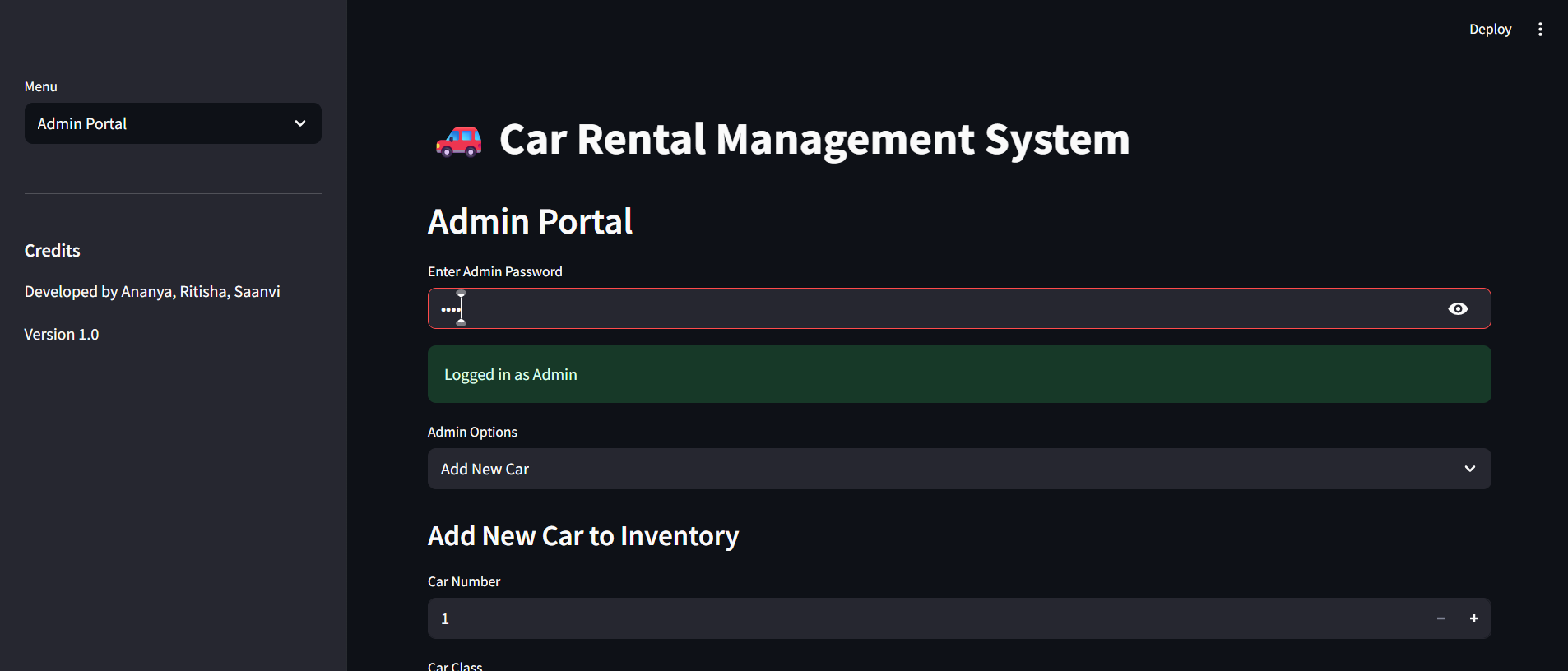


Fig.4.8. Admin Portal

# CHAPTER 5: CONCLUSION AND FUTURE ENHANCEMENTS

**CONCLUSION**

The vehicle rental system developed successfully integrates essential functionalities required for efficient vehicle rental management. It provides a secure and user-friendly interface for customers to register, search for available vehicles, make reservations, and complete payments with ease. The system ensures real-time accuracy by dynamically updating vehicle availability and maintaining comprehensive records of bookings and payments. On the administrative side, it offers robust fleet management capabilities, including maintenance scheduling, insurance tracking, and detailed reporting tools, which support effective oversight and decision-making. The incorporation of automated notifications and role-based access controls enhances both usability and security. Overall, the system achieves its objectives by streamlining operations, improving customer satisfaction, and providing valuable business insights, making it a reliable and scalable solution for vehicle rental companies.

# FUTURE ENHANCEMENTS

While the current system meets core requirements, several enhancements could further improve its functionality and adaptability. Future developments may include integrating a GPS tracking system to monitor vehicle locations in real-time, offering dynamic pricing models based on demand and seasonality, and implementing a mobile application to enhance customer convenience. Additionally, expanding the loyalty and rewards program could help retain frequent customers, while incorporating AI-driven analytics could provide deeper insights into customer preferences and fleet performance. Other potential improvements include multi-language support to cater to a broader audience, enhanced fraud detection mechanisms, and integration with third-party services such as insurance companies and traffic authorities. These enhancements will ensure the system remains competitive and responsive to evolving business needs and customer expectations.

.

# APPENDIX I: SAMPLE SOURCE CODE

import streamlit as st

import sqlite3

from datetime import datetime, timedelta

import pandas as pd

# Initialize database connection

def init\_db():

conn = sqlite3.connect('car\_rental.db')

c = conn.cursor()

# Create tables

c.execute('''CREATE TABLE IF NOT EXISTS cars

(Car\_no INTEGER PRIMARY KEY,

Car\_class TEXT,

Model\_name TEXT,

Car\_color TEXT,

Capacity INTEGER,

Daily\_Rent INTEGER,

Car\_Status TEXT DEFAULT 'AVAILABLE',

Rent\_ID INTEGER DEFAULT NULL)''')

c.execute('''CREATE TABLE IF NOT EXISTS admins

(password TEXT)''')

c.execute('''CREATE TABLE IF NOT EXISTS rentings

(Rent\_ID INTEGER PRIMARY KEY AUTOINCREMENT,

Cust\_Name TEXT,

Cust\_Phone\_no INTEGER,

Car\_no INTEGER,

Date\_Rented TEXT,

Return\_Date TEXT,

Driver TEXT,

Total\_Rent INTEGER,

FOREIGN KEY(Car\_no) REFERENCES cars(Car\_no))''')

# Insert default data if empty

c.execute("SELECT COUNT(\*) FROM admins")

if c.fetchone()[0] == 0:

c.execute("INSERT INTO admins VALUES ('1289')")

c.execute("SELECT COUNT(\*) FROM cars")

if c.fetchone()[0] == 0:

sample\_cars = [

(101, 'SUV', 'Toyota RAV4', 'Red', 5, 50, 'AVAILABLE', None),

(102, 'SEDAN', 'Honda Civic', 'Blue', 4, 40, 'AVAILABLE', None),

(103, 'SUV', 'Ford Explorer', 'Black', 7, 60, 'AVAILABLE', None),

(104, 'SEDAN', 'Toyota Camry', 'Silver', 4, 45, 'AVAILABLE', None),

(105, 'TRUCK', 'Ford F-150', 'White', 3, 70, 'AVAILABLE', None),

(106, 'SUV', 'Chevrolet Tahoe', 'Black', 8, 65, 'AVAILABLE', None),

(107, 'SEDAN', 'Nissan Altima', 'Gray', 4, 42, 'AVAILABLE', None),

(108, 'COMPACT', 'Toyota Corolla', 'Red', 4, 35, 'AVAILABLE', None),

(109, 'SUV', 'Jeep Grand Cherokee', 'White', 5, 55, 'AVAILABLE', None),

(110, 'LUXURY', 'BMW 5 Series', 'Black', 4, 90, 'AVAILABLE', None),

(111, 'LUXURY', 'Mercedes E-Class', 'Silver', 4, 95, 'AVAILABLE', None),

(112, 'SPORTS', 'Porsche 911', 'Red', 2, 150, 'AVAILABLE', None),

(113, 'COMPACT', 'Honda Fit', 'Blue', 4, 30, 'AVAILABLE', None),

(114, 'MINIVAN', 'Honda Odyssey', 'Gray', 7, 60, 'AVAILABLE', None),

(115, 'TRUCK', 'Chevrolet Silverado', 'Black', 3, 75, 'AVAILABLE', None),

(116, 'HYBRID', 'Toyota Prius', 'Green', 4, 48, 'AVAILABLE', None)

]

c.executemany("INSERT INTO cars VALUES (?, ?, ?, ?, ?, ?, ?, ?)", sample\_cars)

conn.commit()

return conn

def view\_all\_cars(conn):

st.header("All Cars in Inventory")

df = pd.read\_sql("SELECT \* FROM cars", conn)

st.dataframe(df)

def view\_available\_cars(conn):

st.header("Available Cars for Rent")

df = pd.read\_sql("SELECT \* FROM cars WHERE Car\_Status='AVAILABLE'", conn)

st.dataframe(df)

def rent\_car(conn):

st.header("Rent a Car")

c = conn.cursor()

available\_cars = pd.read\_sql("SELECT Car\_no, Car\_class, Model\_name, Daily\_Rent FROM cars WHERE Car\_Status='AVAILABLE'", conn)

if available\_cars.empty:

st.warning("No cars available for rent.")

return

car\_options = [f"{row['Car\_no']} - {row['Car\_class']} {row['Model\_name']} (${row['Daily\_Rent']}/day)"

for \_, row in available\_cars.iterrows()]

selected\_car = st.selectbox("Select a Car", car\_options)

car\_no = int(selected\_car.split(" - ")[0])

car\_details = pd.read\_sql(f"SELECT \* FROM cars WHERE Car\_no={car\_no}", conn).iloc[0]

st.subheader("Customer Information")

cust\_name = st.text\_input("Full Name")

cust\_phone = st.text\_input("Phone Number")

st.subheader("Rental Details")

today = datetime.now().date()

rent\_date = st.date\_input("Rental Date", today)

return\_date = st.date\_input("Return Date", today + timedelta(days=1))

if return\_date <= rent\_date:

st.error("Return date must be after rental date.")

return

duration = (return\_date - rent\_date).days

total\_rent = duration \* car\_details['Daily\_Rent']

driver = st.radio("Need a driver? (Additional $20/day)", ("No", "Yes"))

if driver == "Yes":

total\_rent += duration \* 20

st.subheader("Payment Summary")

st.write(f"Car: {car\_details['Car\_class']} {car\_details['Model\_name']}")

st.write(f"Rental Period: {duration} days")

st.write(f"Daily Rate: ${car\_details['Daily\_Rent']}")

if driver == "Yes":

st.write("Driver Fee: $20/day")

st.write(f"Total Rent: ${total\_rent}")

if st.button("Confirm Rental"):

if not cust\_name or not cust\_phone:

st.error("Please fill in all customer details.")

return

c.execute('''INSERT INTO rentings

(Cust\_Name, Cust\_Phone\_no, Car\_no, Date\_Rented, Return\_Date, Driver, Total\_Rent)

VALUES (?, ?, ?, ?, ?, ?, ?)''',

(cust\_name, cust\_phone, car\_no,

rent\_date.strftime("%Y-%m-%d"),

return\_date.strftime("%Y-%m-%d"),

driver, total\_rent))

rent\_id = c.lastrowid

c.execute("UPDATE cars SET Car\_Status='RENTED', Rent\_ID=? WHERE Car\_no=?", (rent\_id, car\_no))

conn.commit()

st.success(f"Car rented successfully! Rent ID: {rent\_id}")

def generate\_rent\_receipt(conn):

st.header("Generate Rent Receipt")

rent\_id = st.number\_input("Enter Rent ID", min\_value=1, step=1)

if st.button("Generate Receipt"):

df = pd.read\_sql(f'''SELECT r.Rent\_ID, r.Cust\_Name, r.Cust\_Phone\_no,

c.Car\_no, c.Car\_class, c.Model\_name, c.Car\_color,

r.Date\_Rented, r.Return\_Date, r.Driver, r.Total\_Rent

FROM rentings r

JOIN cars c ON r.Car\_no = c.Car\_no

WHERE r.Rent\_ID={rent\_id}''', conn)

if not df.empty:

rental = df.iloc[0]

st.subheader("Rental Receipt")

st.write(f"Receipt ID: {rental['Rent\_ID']}")

st.write(f"Customer Name: {rental['Cust\_Name']}")

st.write(f"Phone Number: {rental['Cust\_Phone\_no']}")

st.write(f"Car Details: {rental['Car\_class']} {rental['Model\_name']} ({rental['Car\_color']}) - #{rental['Car\_no']}")

st.write(f"Rental Period: {rental['Date\_Rented']} to {rental['Return\_Date']}")

st.write(f"Driver Option: {rental['Driver']}")

st.write(f"Total Amount: ${rental['Total\_Rent']}")

else:

st.error("Rent ID not found.")

def return\_car(conn):

st.header("Return a Rented Car")

c = conn.cursor()

rented\_cars = pd.read\_sql('''SELECT r.Rent\_ID, c.Car\_no, c.Car\_class, c.Model\_name, r.Cust\_Name

FROM rentings r

JOIN cars c ON r.Car\_no = c.Car\_no

WHERE c.Car\_Status='RENTED' ''', conn)

if rented\_cars.empty:

st.warning("No cars currently rented out.")

return

car\_options = [f"Rent ID: {row['Rent\_ID']} - Car #{row['Car\_no']}: {row['Car\_class']} {row['Model\_name']} (Rented by: {row['Cust\_Name']})"

for \_, row in rented\_cars.iterrows()]

selected\_car = st.selectbox("Select a Rental", car\_options)

rent\_id = int(selected\_car.split(" - ")[0].split(": ")[1])

rental = pd.read\_sql(f'''SELECT r.Rent\_ID, c.Car\_no, r.Total\_Rent

FROM rentings r

JOIN cars c ON r.Car\_no = c.Car\_no

WHERE r.Rent\_ID={rent\_id}''', conn).iloc[0]

st.write(f"Car to be returned: #{rental['Car\_no']}")

st.write(f"Total Rent Paid: ${rental['Total\_Rent']}")

damage\_fee = st.number\_input("Damage Fee (if any)", min\_value=0, value=0)

late\_fee = st.number\_input("Late Return Fee (if any)", min\_value=0, value=0)

total\_fees = damage\_fee + late\_fee

if total\_fees > 0:

st.write(f"Additional Fees: ${total\_fees}")

if st.button("Confirm Return"):

c.execute("UPDATE cars SET Car\_Status='AVAILABLE', Rent\_ID=NULL WHERE Car\_no=?", (rental['Car\_no'],))

if total\_fees > 0:

st.warning(f"Additional fees of ${total\_fees} collected")

conn.commit()

st.success(f"Car #{rental['Car\_no']} returned successfully!")

def view\_rent\_history(conn):

st.header("Rental History")

df = pd.read\_sql('''SELECT r.Rent\_ID, r.Cust\_Name, c.Car\_no, c.Model\_name,

r.Date\_Rented, r.Return\_Date, r.Total\_Rent

FROM rentings r

JOIN cars c ON r.Car\_no = c.Car\_no

ORDER BY r.Date\_Rented DESC''', conn)

if not df.empty:

st.dataframe(df)

st.subheader("Rental Statistics")

col1, col2, col3 = st.columns(3)

col1.metric("Total Rentals", len(df))

col2.metric("Total Revenue", f"${df['Total\_Rent'].sum()}")

avg\_duration = (pd.to\_datetime(df['Return\_Date']) - pd.to\_datetime(df['Date\_Rented'])).dt.days.mean()

col3.metric("Avg Rental Duration", f"{avg\_duration:.1f} days")

else:

st.warning("No rental history found.")

def admin\_portal(conn):

st.header("Admin Portal")

password = st.text\_input("Enter Admin Password", type="password")

c = conn.cursor()

c.execute("SELECT password FROM admins")

correct\_password = c.fetchone()[0]

if password != correct\_password:

st.error("Incorrect password")

return

st.success("Logged in as Admin")

admin\_menu = ["Add New Car", "Update Car Status", "View All Rentals", "Change Admin Password"]

admin\_choice = st.selectbox("Admin Options", admin\_menu)

if admin\_choice == "Add New Car":

add\_new\_car(conn)

elif admin\_choice == "Update Car Status":

update\_car\_status(conn)

elif admin\_choice == "View All Rentals":

view\_all\_rentals(conn)

elif admin\_choice == "Change Admin Password":

change\_admin\_password(conn)

def add\_new\_car(conn):

st.subheader("Add New Car to Inventory")

c = conn.cursor()

car\_no = st.number\_input("Car Number", min\_value=1, step=1)

car\_class = st.selectbox("Car Class", ["SUV", "SEDAN", "TRUCK", "COMPACT", "LUXURY", "SPORTS", "MINIVAN", "HYBRID"])

model\_name = st.text\_input("Model Name")

car\_color = st.text\_input("Color")

capacity = st.number\_input("Seating Capacity", min\_value=1, max\_value=10, step=1)

daily\_rent = st.number\_input("Daily Rent ($)", min\_value=1, step=1)

if st.button("Add Car"):

if pd.read\_sql(f"SELECT Car\_no FROM cars WHERE Car\_no={car\_no}", conn).empty:

c.execute('''INSERT INTO cars

(Car\_no, Car\_class, Model\_name, Car\_color, Capacity, Daily\_Rent)

VALUES (?, ?, ?, ?, ?, ?)''',

(car\_no, car\_class, model\_name, car\_color, capacity, daily\_rent))

conn.commit()

st.success(f"Car #{car\_no} added successfully!")

else:

st.error(f"Car #{car\_no} already exists.")

def update\_car\_status(conn):

st.subheader("Update Car Status")

cars\_df = pd.read\_sql("SELECT Car\_no, Model\_name, Car\_Status FROM cars", conn)

if cars\_df.empty:

st.warning("No cars in the database.")

return

car\_options = [f"#{row['Car\_no']} - {row['Model\_name']} (Current: {row['Car\_Status']})"

for \_, row in cars\_df.iterrows()]

selected\_car = st.selectbox("Select a Car", car\_options)

car\_no = int(selected\_car.split(" - ")[0][1:])

current\_status = cars\_df[cars\_df['Car\_no'] == car\_no]['Car\_Status'].values[0]

new\_status = st.selectbox("New Status", ["AVAILABLE", "RENTED", "MAINTENANCE"])

if st.button("Update Status") and new\_status != current\_status:

if new\_status == "AVAILABLE":

rent\_id = pd.read\_sql(f"SELECT Rent\_ID FROM cars WHERE Car\_no={car\_no}", conn)['Rent\_ID'].values[0]

if rent\_id is not None:

st.error("Cannot set to AVAILABLE - car has active rental.")

return

conn.execute(f"UPDATE cars SET Car\_Status='{new\_status}' WHERE Car\_no={car\_no}")

conn.commit()

st.success(f"Car #{car\_no} status updated to {new\_status}")

def view\_all\_rentals(conn):

st.subheader("All Rental Records")

df = pd.read\_sql('''SELECT r.Rent\_ID, r.Cust\_Name, r.Cust\_Phone\_no,

c.Car\_no, c.Model\_name,

r.Date\_Rented, r.Return\_Date, r.Total\_Rent

FROM rentings r

JOIN cars c ON r.Car\_no = c.Car\_no

ORDER BY r.Date\_Rented DESC''', conn)

if not df.empty:

st.dataframe(df)

if st.button("Export to CSV"):

csv = df.to\_csv(index=False)

st.download\_button("Download CSV", data=csv, file\_name="rental\_records.csv", mime="text/csv")

else:

st.warning("No rental records found.")

def change\_admin\_password(conn):

st.subheader("Change Admin Password")

new\_password = st.text\_input("New Password", type="password")

confirm\_password = st.text\_input("Confirm Password", type="password")

if st.button("Change Password"):

if not new\_password:

st.error("Password cannot be empty.")

elif new\_password != confirm\_password:

st.error("Passwords do not match.")

elif len(new\_password) > 10:

st.error("Password must be 10 characters or less.")

else:

conn.execute(f"UPDATE admins SET password='{new\_password}'")

conn.commit()

st.success("Password changed successfully!")

def main():

st.set\_page\_config(page\_title="Car Rental Management System", layout="wide")

conn = init\_db()

st.title("🚗 Car Rental Management System")

menu = ["View All Cars", "View Available Cars", "Rent Car", "Generate Rent Receipt",

"Return Car", "View Rent History", "Admin Portal"]

choice = st.sidebar.selectbox("Menu", menu)

if choice == "View All Cars":

view\_all\_cars(conn)

elif choice == "View Available Cars":

view\_available\_cars(conn)

elif choice == "Rent Car":

rent\_car(conn)

elif choice == "Generate Rent Receipt":

generate\_rent\_receipt(conn)

elif choice == "Return Car":

return\_car(conn)

elif choice == "View Rent History":

view\_rent\_history(conn)

elif choice == "Admin Portal":

admin\_portal(conn)

st.sidebar.markdown("---")

st.sidebar.markdown("### Credits")

st.sidebar.markdown("Developed by [Your Name]")

st.sidebar.markdown("Version 1.0")

conn.close()

if \_\_name\_\_ == "\_\_main\_\_":

main()