

Centennial College
School of Engineering Technology and Applied Science (SETAS)
Information and Communication Engineering Technology (ICET)

COMP214 - Advanced Database Concepts
Section 4 - Summer 2024

Garage Store Oracle SQL Database on Streamlit

Group Members:

Babajide Aroyewun - 301320575

Eldin Bautista - 301310036

Jessica Marie Hernandez - 301250315

Mattia Carganico – 301278358

Submitted to:

Prof. Zoran Sarajlic

Table of Contents

Problem Domain Description	3
Overview.....	3
Objectives.....	3
Functional Requirements:.....	3
Non-Functional Requirements:	3
Key Components	4
Programming Units	4
Database Creation and Indexes	5
Sample Data and Tests.....	6
ER Diagram and Oracle SQL Model.....	6
Tables:	6
Relationships:	8
Detailed ERD Implementation	9
Web Application: Garage Database Interface	10
Overview.....	10
Features	10
Application Structure.....	10
Detailed Components	10
Error Handling	15
Dependencies.....	15
Running the Application	15
How to Use	16
1. View Data:.....	16
2. SQL Console:.....	17
3. Update Vehicle:.....	18
4. Insert Workorder:	19

Problem Domain Description

Overview

The automotive service management system is designed to efficiently manage the operations of an auto repair garage, focusing on handling customer information, vehicle details, service schedules, work orders, and payments. The system uses a relational database schema implemented in Oracle SQL to support its functionalities.

Objectives

- To manage customer and vehicle information effectively.
- To schedule and track vehicle services.
- To handle work orders and payments efficiently.
- To provide accurate and real-time information on garage operations.
- To automate routine tasks and ensure data integrity.

Functional Requirements:

- Track customer information and link it to their vehicles.
- Manage vehicle details and associate them with scheduled services.
- Schedule services for vehicles and generate work orders and its statuses.
- Record payments for work orders and update payment statuses.
- Enforce garage capacity constraints to 10 to prevent overbooking.
- Efficiently search and retrieve customer and vehicle information using indexes.
- Calculate the revenue of the garage based on specific date.
- Assign a loyalty tier to customers based on the number of services they have.
- Count the number of workorders created
- Provide the date for the latest service conducted to the vehicle

Non-Functional Requirements:

- Ensure data integrity through appropriate foreign key constraints.
- Improve query performance with strategically placed indexes.

- Automate payment and status updates using triggers to reduce manual intervention

Key Components

The system includes the following key components, which are represented by database tables and programming units:

1. Customer Management:

- **ad_customers** table: Stores customer information, including names, contact details, and addresses.

2. Vehicle Management:

- **ad_vehicles** table: Maintains details about vehicles owned by customers, such as VIN, year, mileage, model, maker, engine type, and license plate number.

3. Service Management:

- **ad_services** table: Lists the various services offered by the garage, with details like service name, description, and price.
- **service_schedule** table: Links vehicles to scheduled services, detailing which services are to be performed on which vehicles.

4. Work Order Management:

- **ad_workorders** table: Tracks work orders for vehicle services, including details such as start and end dates, VIN, service schedule, and status of the work order.

5. Payment Management:

- **ad_payments** table: Manages payment records associated with work orders, including payment dates and status.

Programming Units

The system utilizes various programming units (procedures, functions, and triggers) to automate and streamline operations:

1. Procedures:

- **update_vehicle_mileage**: Updates the mileage of a vehicle, ensuring the new mileage is greater than the existing one.

- `get_overdue_payments`: Retrieves and displays overdue payments by iterating over pending payments that are past due.
- `count_vehicles_in_garage`: Counts the number of vehicles currently in the garage based on work order statuses and payment statuses.
- `insert_workorder`: Inserts a new work order into the database and updates global work order count and last updated timestamp.

2. Functions:

- `calculate_revenue`: Calculates total revenue for a given date range by summing up service prices of completed and paid work orders.
- `get_customer_tier`: Determines the loyalty tier of a customer based on the number of services they have availed.

3. Triggers:

- `trg_create_payment`: Automatically creates a payment entry when a new work order is created.
- `trg_update_payment`: Updates the payment date when the payment status changes to 'PAID'.
- `trg_check_garage_capacity`: Checks the garage capacity before inserting a new work order, ensuring it does not exceed the maximum capacity.

4. Packages:

- `garage_pkg`: Encapsulates procedures and functions related to the garage operations, allowing for modular and organized code management.

5. Global Variables

- `g_workorder_count`: A counter for work orders.
- `g_last_updated`: Stores the last update date

Database Creation and Indexes

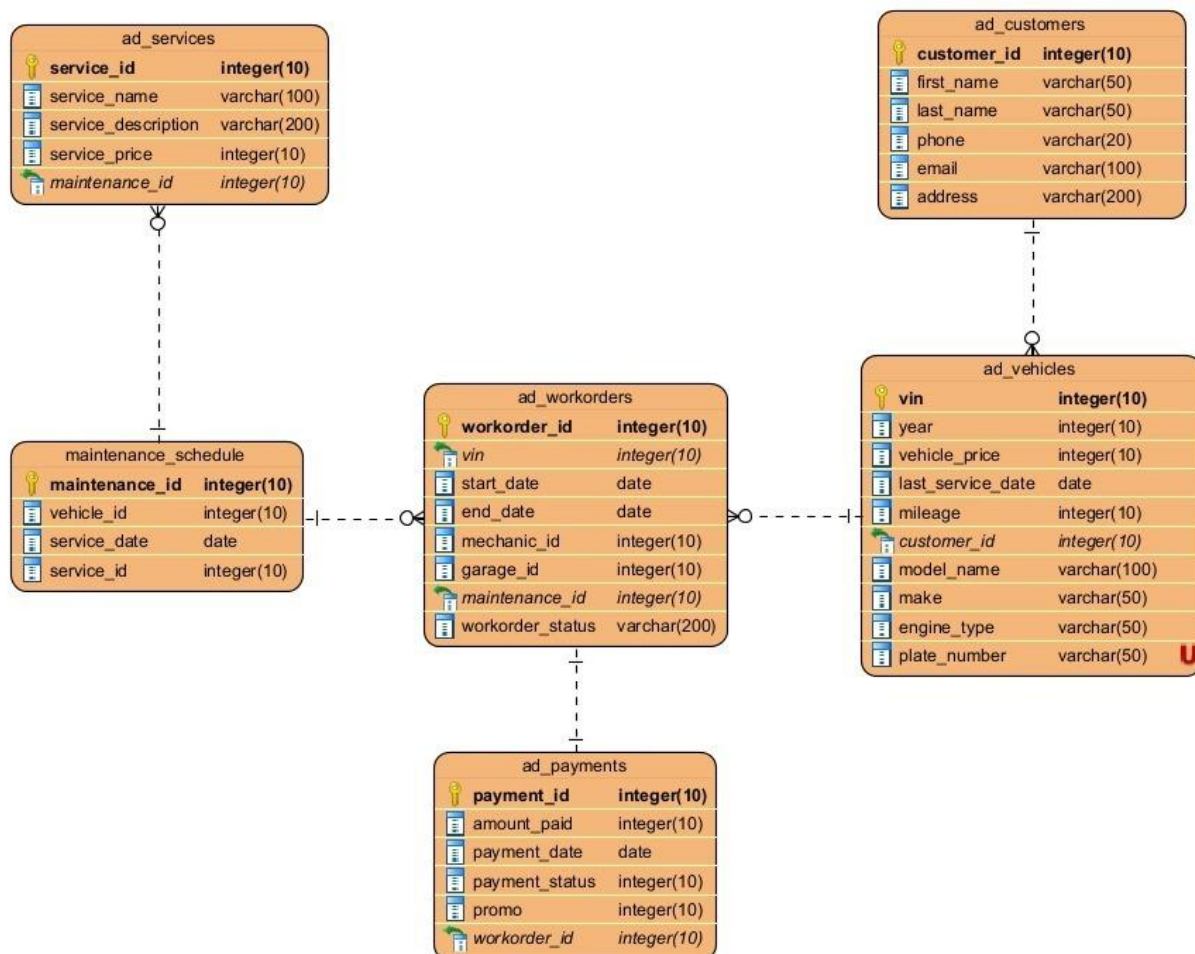
The system includes SQL scripts for creating and populating tables, resetting sequences, and defining indexes to optimize searches:

- **Indexes**: Created on columns such as `last_name` and `email` in `ad_customers`, and `plate_number` in `ad_vehicles` to enhance query performance.
- **Sequences**: Used to generate unique identifiers for work orders and payments.

Sample Data and Tests

The SQL scripts also include sample data inserts for customers, vehicles, services, schedules, work orders, and payments. Additionally, tests for procedures and triggers are provided to ensure the correct functionality of the system components.

ER Diagram and Oracle SQL Model



The Entity Relationship (ER) Diagram shows the relationships of the tables that are created for the garage shop's automotive service management system. The schema is composed of six tables: ad_services, maintenance_schedule, ad_workorders, ad_customers, ad_vehicles, and ad_payments.

Tables:

1. ad_services:

- **service_id**: Primary key, an integer uniquely identifying each service.
- **service_name**: Name of the service, varchar(100).

- **service_description**: Description of the service, varchar(200).
- **service_price**: Price of the service, integer.
- **maintenance_id**: Foreign key referencing maintenance_schedule.maintenance_id, integer.

2. maintenance_schedule:

- **maintenance_id**: Primary key, an integer uniquely identifying each maintenance schedule.
- **vehicle_id**: Foreign key referencing ad_vehicles.vin, integer.
- **service_date**: Date when the service is scheduled.
- **service_id**: Foreign key referencing ad_services.service_id, integer.

3. ad_workorders:

- **workorder_id**: Primary key, an integer uniquely identifying each work order.
- **vin**: Foreign key referencing ad_vehicles.vin, integer.
- **start_date**: Date when the work order starts.
- **end_date**: Date when the work order ends.
- **mechanic_id**: Integer, identifier for the mechanic.
- **garage_id**: Integer, identifier for the garage.
- **maintenance_id**: Foreign key referencing maintenance_schedule.maintenance_id, integer.
- **workorder_status**: Status of the work order, varchar(200).

4. ad_customers:

- **customer_id**: Primary key, an integer uniquely identifying each customer.
- **first_name**: Customer's first name, varchar(50).
- **last_name**: Customer's last name, varchar(50).
- **phone**: Customer's phone number, varchar(20).
- **email**: Customer's email address, varchar(100).
- **address**: Customer's address, varchar(200).

5. **ad_vehicles:**

- **vin:** Primary key, an integer uniquely identifying each vehicle.
- **year:** Year of the vehicle, integer.
- **vehicle_price:** Price of the vehicle, integer.
- **last_service_date:** Date of the last service, date.
- **mileage:** Vehicle mileage, integer.
- **customer_id:** Foreign key referencing ad_customers.customer_id, integer.
- **model_name:** Name of the vehicle model, varchar(100).
- **make:** Manufacturer of the vehicle, varchar(50).
- **engine_type:** Type of the vehicle engine, varchar(50).
- **plate_number:** License plate number, varchar(50).

6. **ad_payments:**

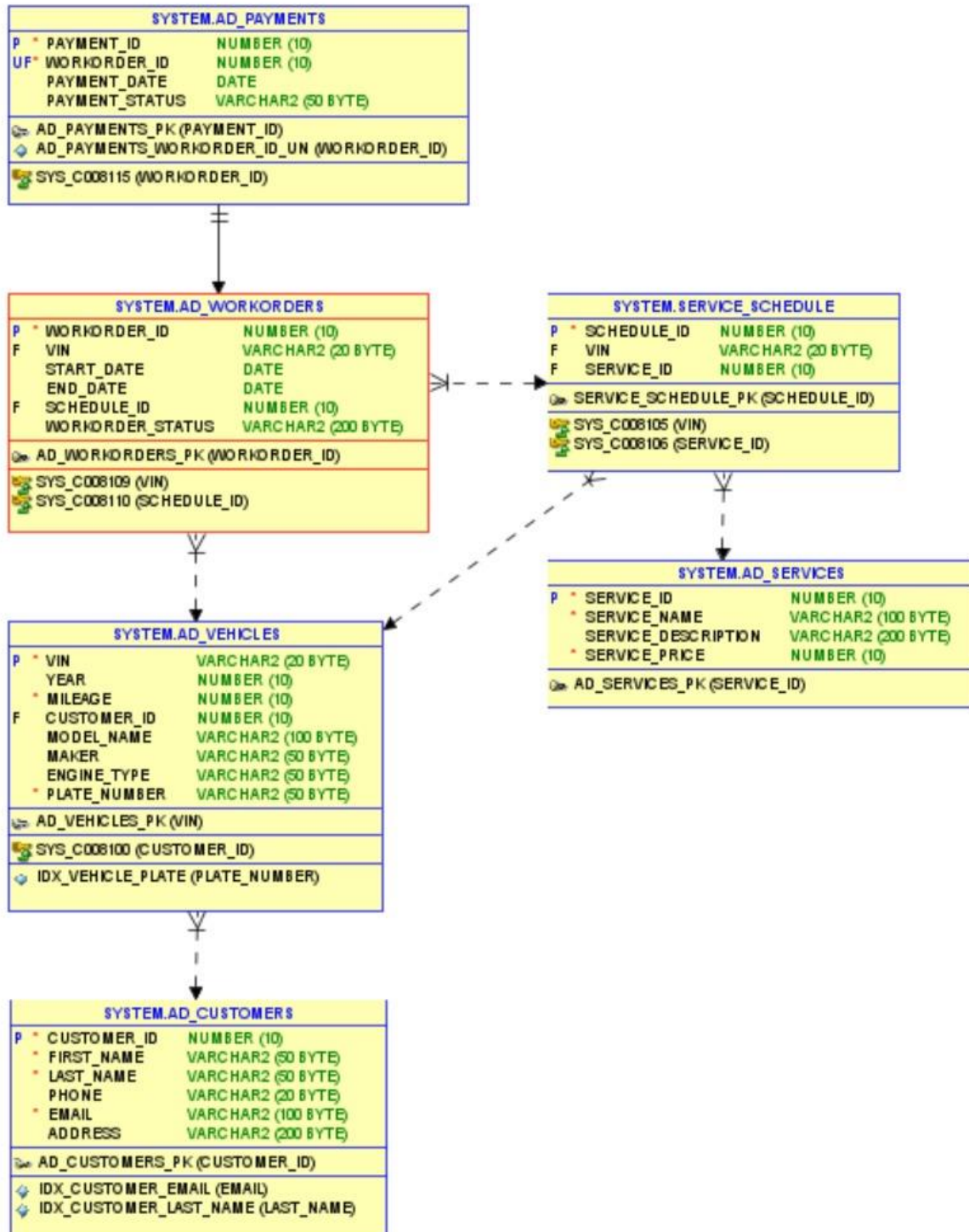
- **payment_id:** Primary key, an integer uniquely identifying each payment.
- **amount_paid:** Amount paid, integer.
- **payment_date:** Date when the payment was made.
- **payment_status:** Status of the payment, varchar(100).
- **promo:** Promotional code or details, varchar(100).
- **workorder_id:** Foreign key referencing ad_workorders.workorder_id, integer.

Relationships:

- The ad_services table is linked to the maintenance_schedule table through the maintenance_id foreign key.
- The maintenance_schedule table is linked to the ad_workorders table via the maintenance_id foreign key.
- The ad_workorders table has a foreign key relationship with the ad_vehicles table through the vin.
- The ad_vehicles table is related to the ad_customers table via the customer_id foreign key.
- The ad_payments table is linked to the ad_workorders table through the workorder_id foreign key.

Detailed ERD Implementation

The Oracle SQL model below represents the detailed implementation of the ER diagram.



Web Application: Garage Database Interface

Overview

The **Garage Database Interface** is a web application designed to manage and interact with an automotive service management system's Oracle database. This application allows users to view, update, and insert data related to customers, vehicles, services, work orders, and payments through an intuitive web interface.

Features

1. View Data:

- Users can select and view data from various tables in the database, such as ad_vehicles, ad_customers, ad_workorders, ad_payments, ad_services, and service_schedule.

2. SQL Console:

- Allows users to interact with the database by executing SQL queries.

3. Update Vehicle:

- Allows users to update the mileage and other details of vehicles.

4. Insert Workorder:

- Provides functionality to insert new work orders into the system.

Application Structure

The application is structured using Streamlit, a popular framework for creating web applications in Python. The key components include functions for database connection, SQL execution, procedure calls, and the Streamlit UI configuration.

Detailed Components

1. Database Connection

The function `get_db_connection()` establishes a connection to the Oracle database using `cx_Oracle`.

```
def get_db_connection():  
    return cx_Oracle.connect(user='SYSTEM', password='password',  
                             dsn='localhost:1521/xe')
```

2. SQL Execution

The function `execute_sql(query)` executes SQL queries and returns the results as a pandas DataFrame. It handles exceptions and displays errors using Streamlit.

```
def execute_sql(query):
    try:
        with get_db_connection() as conn:
            return pd.read_sql(query, conn)
    except cx_Oracle.Error as e:
        st.error(f"Error executing SQL: {e}")
        return None
```

3. Procedure Calls

The function `call_procedure(procedure_name, params=None)` calls stored procedures in the Oracle database, enabling `DBMS_OUTPUT` to fetch the procedure's output.

```
def call_procedure(procedure_name, params=None):
    try:
        with get_db_connection() as conn:
            cursor = conn.cursor()
            cursor.callproc("dbms_output.enable")
            cursor.callproc(procedure_name, params)
            chunk_size = 100
            lines = []
            numlines = cursor.var(int)
            linebuf = cursor.arrayvar(str, chunk_size)
            while True:
                cursor.callproc("dbms_output.get_lines", (linebuf, numlines))
                nlines = numlines.getvalue()
                lines.extend(linebuf.getvalue()[0:nlines])
                if nlines < chunk_size:
                    break
            conn.commit()
            return lines
    except cx_Oracle.Error as e:
        st.error(f"Error during procedure call: {e}")
        return None
```

4. Streamlit UI Configuration

Configures the Streamlit page with a title and layout.

```
st.set_page_config(page_title="Garage Database Interface", layout="wide")
st.title('Garage Database Interface')
```

5. Main Interface

Provides navigation options and functionality based on user selection.

```
st.sidebar.title("Options")
page = st.sidebar.radio("Go to", ["View Data", "Update Vehicle", "Insert Workorder"])
```

6. View Data

Allows users to select and view data from different tables in the database.

```
if page == "View Data":
    st.header('View Data')
    tables = ["ad_vehicles", "ad_customers", "ad_workorders", "ad_payments",
"ad_services", "service_schedule"]
    selected_table = st.selectbox("Select Table", tables)
    if selected_table:
        query = f"SELECT * FROM {selected_table}"
        data = execute_sql(query)
        if data is not None:
            st.dataframe(data)
```

Associated SQL Query:

```
SELECT * FROM ad_vehicles;
SELECT * FROM ad_customers;
SELECT * FROM ad_workorders;
SELECT * FROM ad_payments;
SELECT * FROM ad_services;
SELECT * FROM service_schedule;
```

7. SQL Console

Allows users to interact directly with the database by executing SQL queries.

```
if page == "SQL Console":
    st.header('SQL Console')

    # Create a text area for SQL input
    sql_query = st.text_area("Enter your SQL query here:", height=150)

    # Execute button
    if st.button('Execute Query'):
        if sql_query:
            # Remove any trailing semicolons as they can cause issues with some
            # database drivers
            sql_query = sql_query.strip().rstrip(';')

            try:
                # Execute the SQL query
                result = execute_sql(sql_query)
                if result is not None:
                    # Display the results
                    st.dataframe(result)
                else:
                    st.write("Query executed successfully, but returned no results.")
            except Exception as e:
```

```
        st.error(f"Error executing query: {str(e)}")
        st.write("Please check your SQL syntax and try again.")
    else:
        st.write("Please enter a SQL query.")
```

8. Update Vehicle

Allows users to update the mileage of a vehicle.

```
if page == "Update Vehicle":
    st.header('Update Vehicle Mileage')
    vin = st.text_input('Vehicle VIN')
    new_mileage = st.number_input('New Mileage', min_value=0)
    if st.button('Update Mileage'):
        procedure_name = "update_vehicle_mileage"
        params = [vin, new_mileage]
        output = call_procedure(procedure_name, params)
        if output:
            st.success(f'Mileage updated successfully: {output}')
```

Associated Procedure:

```
CREATE OR REPLACE PROCEDURE update_vehicle_mileage(
    p_vin IN ad_vehicles.vin%TYPE,
    p_new_mileage IN ad_vehicles.mileage%TYPE
)
IS
    v_old_mileage NUMBER;
    e_invalid_mileage EXCEPTION;
BEGIN
    SELECT mileage INTO v_old_mileage
    FROM ad_vehicles
    WHERE vin = p_vin;

    IF p_new_mileage <= v_old_mileage THEN
        RAISE e_invalid_mileage;
    END IF;

    UPDATE ad_vehicles
    SET mileage = p_new_mileage
    WHERE vin = p_vin;

    COMMIT;
    DBMS_OUTPUT.PUT_LINE('Mileage updated successfully.');
```

```
EXCEPTION
    WHEN NO_DATA_FOUND THEN
        DBMS_OUTPUT.PUT_LINE('Vehicle not found.');
```

```
    WHEN e_invalid_mileage THEN
        DBMS_OUTPUT.PUT_LINE('New mileage must be greater than current mileage.');
```

```
    WHEN OTHERS THEN
```

```
DBMS_OUTPUT.PUT_LINE('An error occurred: ' || SQLERRM);  
ROLLBACK;  
END;
```

9. Insert Workorder

Provides functionality to insert a new work order.

```
if page == "Insert Workorder":  
    st.header('Insert New Workorder')  
    vin = st.text_input('Vehicle VIN')  
    schedule_id = st.number_input('Schedule ID', min_value=1)  
    start_date = st.date_input('Start Date', value=datetime.today())  
    if st.button('Insert Workorder'):  
        procedure_name = "insert_workorder"  
        params = [vin, schedule_id, start_date]  
        output = call_procedure(procedure_name, params)  
        if output:  
            st.success(f'Workorder inserted successfully: {output}')
```

Associated Procedure:

```
CREATE OR REPLACE PROCEDURE insert_workorder(  
    p_vin IN ad_vehicles.vin%TYPE,  
    p_schedule_id IN service_schedule.schedule_id%TYPE,  
    p_start_date IN DATE DEFAULT SYSDATE  
)  
IS  
BEGIN  
    INSERT INTO ad_workorders (  
        workorder_id, vin, start_date, schedule_id, workorder_status  
    ) VALUES (  
        workorder_id_seq.NEXTVAL, p_vin, p_start_date, p_schedule_id, 'PENDING'  
    );  
  
    DBMS_OUTPUT.PUT_LINE('Workorder added successfully.');
```

```
EXCEPTION  
    WHEN NO_DATA_FOUND THEN  
        DBMS_OUTPUT.PUT_LINE('No vehicle found with the specified VIN.');
```

```
    WHEN OTHERS THEN  
        DBMS_OUTPUT.PUT_LINE('An error occurred: ' || SQLERRM);  
        ROLLBACK;  
END;
```

Error Handling

The application handles database connection errors, SQL execution errors, and procedure call errors gracefully by displaying error messages using Streamlit's `st.error` function.

Dependencies

- **Streamlit:** For creating the web interface.
- **Pandas:** For handling data frames.
- **cx_Oracle:** For connecting to the Oracle database.
- **datetime:** For handling date and time operations.

Running the Application

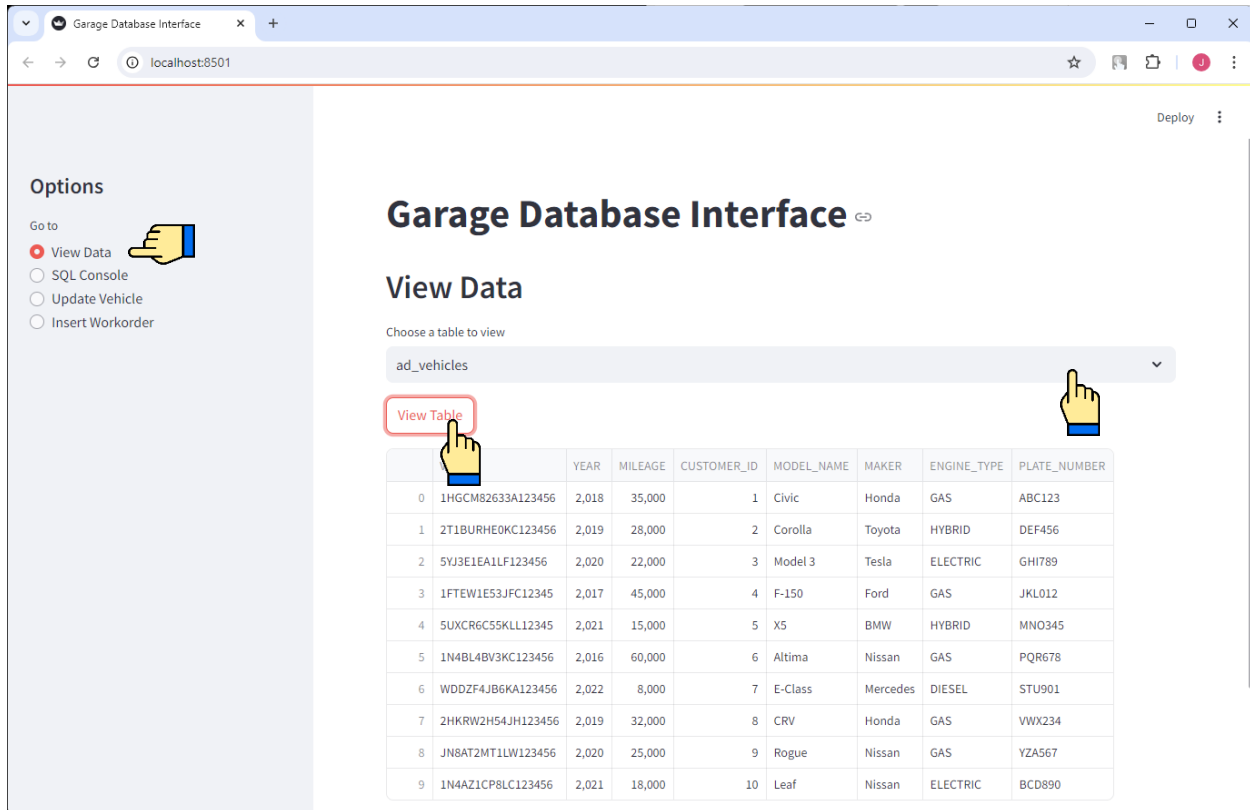
To run the application, execute the following command in your terminal:

```
streamlit run app.py
```

How to Use

1. View Data:

- Navigate to the "View Data" section from the sidebar.
- Select the desired table from the dropdown menu to view its contents.

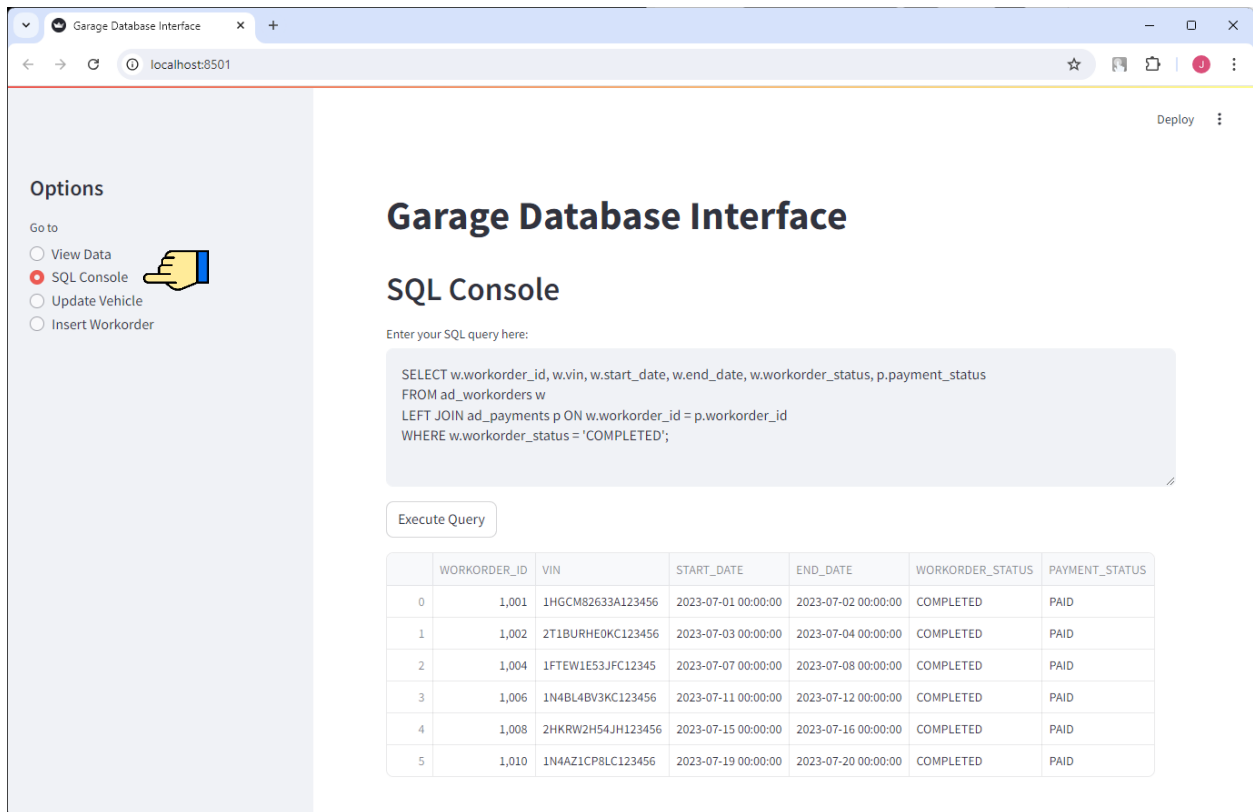


The screenshot displays the 'Garage Database Interface' web application. On the left sidebar, under 'Options', the 'View Data' option is selected. The main content area is titled 'Garage Database Interface' and 'View Data'. Below the title, there is a dropdown menu labeled 'Choose a table to view' with 'ad_vehicles' selected. A red box highlights the 'View Table' button. A table of vehicle data is displayed below the button.

		YEAR	MILEAGE	CUSTOMER_ID	MODEL_NAME	MAKER	ENGINE_TYPE	PLATE_NUMBER
0	1HGCM82633A123456	2,018	35,000	1	Civic	Honda	GAS	ABC123
1	2T1BURHE0KC123456	2,019	28,000	2	Corolla	Toyota	HYBRID	DEF456
2	5YJ3E1EA1LF123456	2,020	22,000	3	Model 3	Tesla	ELECTRIC	GHI789
3	1FTEW1E53JFC12345	2,017	45,000	4	F-150	Ford	GAS	JKL012
4	5UXCR6C55KLL12345	2,021	15,000	5	X5	BMW	HYBRID	MNO345
5	1N4BL4BV3KC123456	2,016	60,000	6	Altima	Nissan	GAS	PQR678
6	WDDZF4JB6KA123456	2,022	8,000	7	E-Class	Mercedes	DIESEL	STU901
7	2HKRW2H54JH123456	2,019	32,000	8	CRV	Honda	GAS	VWX234
8	JN8AT2MT1LW123456	2,020	25,000	9	Rogue	Nissan	GAS	YZA567
9	1N4AZ1CP8LC123456	2,021	18,000	10	Leaf	Nissan	ELECTRIC	BCD890

2. SQL Console:

- Navigate to the "SQL Console" section from the sidebar.
- Input the SQL commands and click Execute Query button.



The screenshot shows a web browser window titled "Garage Database Interface" at the URL "localhost:8501". The interface has a sidebar on the left with "Options" and a main area on the right. In the sidebar, "SQL Console" is selected with a red dot and a hand icon. The main area has a heading "Garage Database Interface" and "SQL Console". Below this is a text input field with the SQL query:

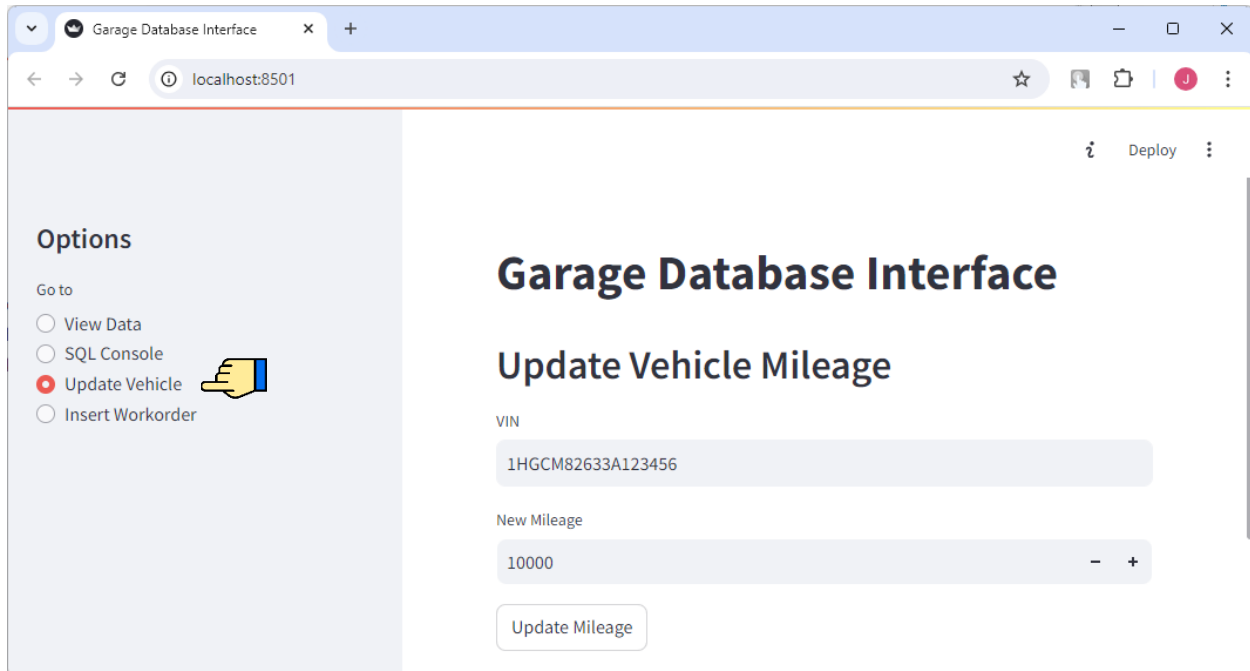
```
SELECT w.workorder_id, w.vin, w.start_date, w.end_date, w.workorder_status, p.payment_status
FROM ad_workorders w
LEFT JOIN ad_payments p ON w.workorder_id = p.workorder_id
WHERE w.workorder_status = 'COMPLETED';
```

 Below the query is an "Execute Query" button. The results are displayed in a table with 7 columns: WORKORDER_ID, VIN, START_DATE, END_DATE, WORKORDER_STATUS, and PAYMENT_STATUS. The table contains 6 rows of data.

	WORKORDER_ID	VIN	START_DATE	END_DATE	WORKORDER_STATUS	PAYMENT_STATUS
0	1,001	1HGCM82633A123456	2023-07-01 00:00:00	2023-07-02 00:00:00	COMPLETED	PAID
1	1,002	2T1BURHE0KC123456	2023-07-03 00:00:00	2023-07-04 00:00:00	COMPLETED	PAID
2	1,004	1FTEW1E53JFC12345	2023-07-07 00:00:00	2023-07-08 00:00:00	COMPLETED	PAID
3	1,006	1N4BL4BV3KC123456	2023-07-11 00:00:00	2023-07-12 00:00:00	COMPLETED	PAID
4	1,008	2HKRW2H54JH123456	2023-07-15 00:00:00	2023-07-16 00:00:00	COMPLETED	PAID
5	1,010	1N4AZ1CP8LC123456	2023-07-19 00:00:00	2023-07-20 00:00:00	COMPLETED	PAID

3. Update Vehicle:

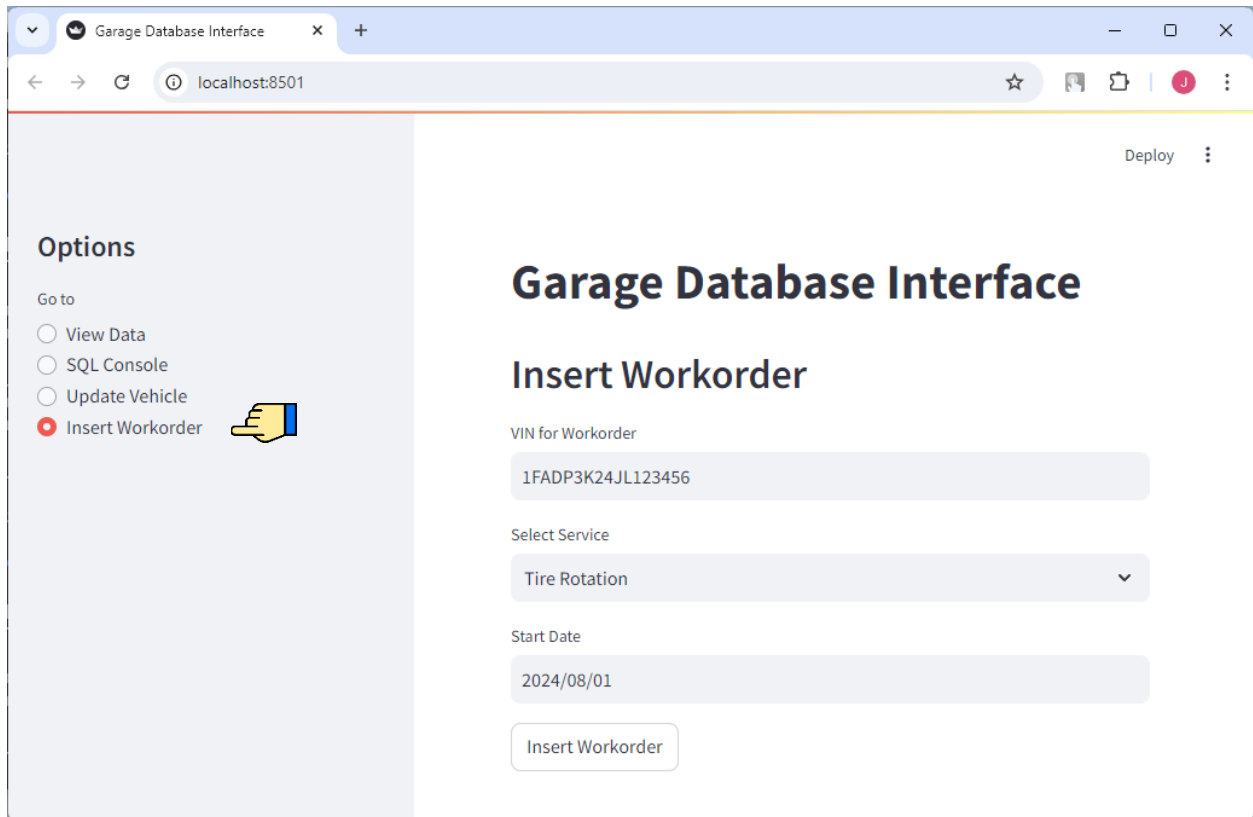
- Navigate to the "Update Vehicle" section from the sidebar.
- Input the vehicle details and submit the form to update the vehicle's information.



The screenshot shows a web browser window with the title "Garage Database Interface" and the address "localhost:8501". The interface has a sidebar on the left with the heading "Options" and a "Go to" section. The sidebar contains four radio button options: "View Data", "SQL Console", "Update Vehicle" (which is selected and has a yellow hand icon pointing to it), and "Insert Workorder". The main content area has the heading "Garage Database Interface" and a sub-heading "Update Vehicle Mileage". Below the sub-heading, there are two input fields: "VIN" with the value "1HGCM82633A123456" and "New Mileage" with the value "10000". There is a "Deploy" button in the top right corner and an "Update Mileage" button at the bottom of the form.

4. Insert Workorder:

- Navigate to the "Insert Workorder" section from the sidebar.
- Fill in the required details and submit the form to insert a new work order.



The screenshot shows a web browser window with the title "Garage Database Interface" and the URL "localhost:8501". The interface has a sidebar on the left with the heading "Options" and four radio button options: "View Data", "SQL Console", "Update Vehicle", and "Insert Workorder". The "Insert Workorder" option is selected, and a hand cursor icon is pointing at it. The main content area has the heading "Garage Database Interface" and "Insert Workorder". Below the heading, there are three input fields: "VIN for Workorder" with the value "1FADP3K24JL123456", "Select Service" with a dropdown menu showing "Tire Rotation", and "Start Date" with the value "2024/08/01". At the bottom of the form is a button labeled "Insert Workorder".