

AutoZone
(A National retail and service store)

Name: Naga Venkata Kanakalakshmi Murikipudi

UNT ID: 11725119

Project – Part 4: (PL/SQL Code)

Group Number: 8

Group Members:

Name	UNT ID	Mail ID
Naga Venkata kanakalakshmi	11725119	nagavenkatakanakalmurikipudi@my.unt.edu
Vishnu Vardhan Reddy Sudireddy	11642773	vishnusudireddy@my.unt.edu
Srinivas Sankula	11667743	srinivassankula@my.unt.edu
Sai Sindhu Rudraraju	11644475	saisindhurudraraju@my.unt.edu

Introduction:

AutoZone, a national retailer of automotive parts and accessories Company has requirement to store the information of the inventory, employees, insurance, servicing, payment plans, supplier details and customer details. AutoZone also want to store the information about locations, jobs done by the employees and automotive retailer branch details.

Description:

- Each retailer in the AutoZone chain needs to be identified based on the retailer id. It can have the basic information like contact, business hour and each AutoZone retailer can have a manager and the website details related to the that location. Multiple employees can work in the one automotive retailer. Each retailer can have in house inventory and external inventory. Automotive retailer address/location needs to be stored. Each automotive retailer can afford multiple jobs or services.
- Inventory product entity serves as a major component and uniquely identified in this system. It manages the various automotive products. It includes the attributes like name, quantity, price, automotive_retailer_id, automobile_id, address_id. It holds the details of the automotive retailer, automobile data. It establishes the relation with suppliers, automotive retailers, part service, automobile, address and bill. It serves as a central component for efficient inventory management, procurement, and tracking within the system.
- Employee entity holds the data of the employees working in AutoZone. Each employee is uniquely identified with their ID. Along with the ID this entity will also have the attributes first name, last name, date of birth, phone number, email address, annual salary, ssn, address, hire date and the automotive retailer ID. An employee can create many bills for the customers and so the employee entity will form a one-to-many

relation with the bill entity. More than one employee can have the same address as there is a chance of 2 employees living in the same location. So, employee will form a many to one relationship with the address entity. Many employees can be part of a job and one employee can be part of many jobs. So, employee entity will form many to many relationships with the job entity. As Many employees work in one location, employee entity will form a many to one relationship with the automotive retailer entity. Employees may receive multiple paychecks over a period. So, Employee entity will form a one-to-many relationship with employee payroll entity.

- Employee payroll entity has the record of paychecks given to employees. Each pay given to an employee is identified by a unique id. This entity has the attributes hours worked (number of hours worked by employee during the pay cycle), start date (start date of the pay cycle), end date (end date of the pay cycle), pay (amount paid to the employee) and employee id (id of the employee that is used to uniquely identify an employee). The employee payroll entity will form a many to one relationship with the employee as one employee may receive many pays over a period.
- AutoZone can have suppliers to inventory who supply the products that stored or used during the services. An inventory can have multiple suppliers and vice versa. Suppliers' information like contact and address needs to be stored. Suppliers can uniquely identified by their id.
- Address entity serves as a global component within the system, defines location information. It is uniquely identified and includes the attributes like apartment number, street, city, state, country, and ZIP code, all of them are mandatory. This entity forms relationships with other entities in various ways: many employees may have a single address, inventory products are associated with specific addresses, automotive retailers and suppliers have distinct addresses, and multiple customers can be linked to one address.
- Bill entity is crucial for recording financial transactions. It is uniquely defined and includes attributes like date, payment mode, insurance, customer, job, employee, sale type (can be an online or offline), and payment plan. It forms relationships with various entities: multiple bills can be linked to one employee, have many-to-many connections with inventory products, and maintain multiple bills of each association with insurance, customers, and payment plans. Additionally, every job has a specific bill.
- Job entity is essential for managing automotive service tasks. It is uniquely defining and includes attributes like date, description, customers, automotive retailers, VIN numbers, and automobiles. All of which are mandatory. "Job" forms key relationships: It establishes a direct link with billing records, multiple jobs are associated with automobiles and automotive retailers, Customers. Various jobs can be done by different employees. Whereas Single job can be performed using multiple part services. This entity serves as a central hub for tracking and managing automotive service jobs.
- Part service like during the service which are automotive parts we are using to get the job done for a vehicle. It may have the information like job details, name/description of the service and it can have quantity of the parts which are using for that service. Part service can be identified by their id. Part service can have type like is it only service or

any parts used to get that service done. In this many parts service can be related to one job and it have information related to the inventory products to know the parts related to the which inventory.

- Customer entity represents individuals in the system and is identified uniquely. It includes essential attributes like first name, last name, birthdate, driver's license, phone, and address. Multiple customers may opt for multiple insurance policies. A Customer can be associated with multiple jobs which has multiple bills. Many customers may have the single address. This entity enables efficient management of customer data, service history, and financial transactions.
- Insurance is like if customer wants to utilize his insurance plan for the payment, we need to store the information of that insurance. It can have policy type, provider and claim percentage. The particular insurance plan can be identified by plan id. A customer can have multiple insurance and vice versa. we would also like to include insurance id in the bill, it will be like one insurance plan can be included in many bills.
- Payment plan like if customer interested in the paying in instalments, he can use this option. This payment plan will have the information like plan name, number of instalments and the interest rate related to the payment plan. Each payment can be identified by plan id and multiple bills can have single payment plan.
- Would like to store the information about automobile. So that when the job is performed we can use the parts related to the specific automobile model. It can have the information like manufacture, name, variant, year of the build and color of the vehicle. Each automobile model can be identified by automobile id. An automobile can have multiple products in the inventory and would like to keep the automobile information in the jobs performed. It will be like multiple jobs can be performed to an automobile.
- Inventory-product-supplier is a relation table. Which has a primary and foreign key as inventory_product_id, supplier_id. It has a single attribute quantity. Multiple suppliers will provide multiple products. In this we maintain the product_id, supplier_id and the product quantity provided by the supplier.
- Bill_inventory_product is a relation table. Which has a primary and foreign key as bill_id, inventory_product_id. It has a single attribute quantity. Multiple products will have the multiple bills. In this we maintain the bill_id, product_id and the product quantity in each generated bill.
- Job_employee is a relation table. Which has the primary and foreign key as job_id and employee_id. There are multiple employees, working for the on the different jobs to maintains the those details this table helps to track the respective details. It's a join of employee and job and the job table.
- Cunstomer_insurance is a relation table. Which has the primary and foreign key as cutomer_id and insurance_id. It has a single attribute status. Many customers can opt

for multiple insurances. This table helps to track the insurance opted by the customers and their status.

- In the automotive_retailer a name attribute is newly added. The name identifier helps to track the name of specific automotive_retailer.

Along with the above description we have added one more attribute in the Job table called job_status which describes about the status of the job such as 'completed', 'inprogress', 'unassigned'.

Assumptions:

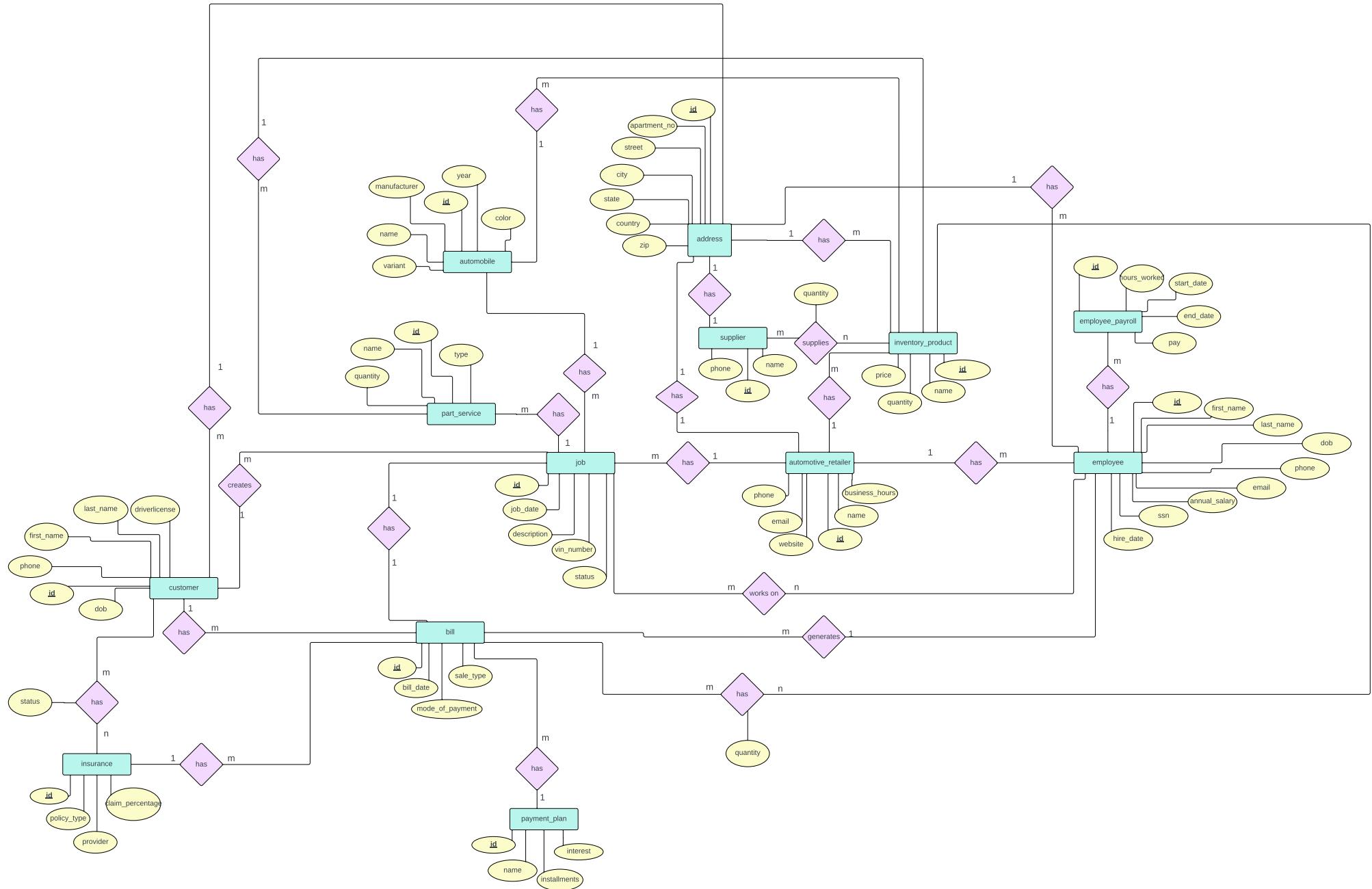
- A product can be available both in store and in warehouse. If the product has automotive_retailer_id same as address id, that means the product is in store. And if automotive_retailer_id is null, then the inventory is not in store.
- Automobile entity will have the generic information about the model, manufacturer and make of the automobile available in the market.
- A job will be created if a customer wants to get his automobile serviced in the store and the bill will be generated for the job.
- A bill can be generated without a job when a customer purchases products from the store without getting the service done in store where this data is stored in bill_inventory table.
- Employee can receive multiple payments over a period. This is maintained in employee_payroll table.
- Employee can be part of the job and he can generate bill for the job. And employee can sell the products to the customer who is not seeking service (not part of the job) from the retailer.
- Address is global table where the addresses of suppliers, customers, employees, inventory and retailer are stored.
- customer can pay the bill directly or he can opt for payment plan. Customer can select the payment plan form the payment_plan table.
- Job will be created if a customer opts services from the retailer. services are work done by the employees on the automobile to fix issues or install accessories.
- Part_services have the information about the products used in the job. One Job can have multiple products.
- Insurance table has different insurance policies that are available in the market.
- more than one person (customer or employee) can have same address. but no 2 suppliers and retailers can have the same address.
- product id in part service can be null as labor chargers of a job can also be clocked in part service where no inventory will be tagged to that service.

- The total amount of the bill of a customer who opted for a payment plan will be recorded as a transaction in bill table. And the installments are also saved as transactions in the same table.
- A phone number must consist of 10 digits and should only contain numerical characters, with no special symbols or spaces.
- Each email address must be unique, meaning no two users can share the same email ID.
- In the bill table, the sale type can be categorized as either online or offline.
- The vin_number in the job table must have a fixed length of 17 characters.
- The quantity of one item in a bill cannot exceed 10, as customers are limited to purchasing a maximum of 10 identical items.
- The status of insurance in the customer_insurance table can be either active or inactive.
- Zip codes are required to have a fixed length of 5 characters.
- Employees cannot work for more than 100 hours within a two-week period.
- The mode of payment in the bill table can be either cash or card.
- Sales cannot occur across different AutoZone locations. It is limited to few locations.
- When an automotive retailer is deleted from a location, the suppliers associated with that location are also removed.

Along with the above assumption we have added one more assumption as part of project part 4.

- In our autozone, in-progress jobs play the vital role until the job_status changed to completed. so, any changes on the customer table are closely monitored.

Entity Relationship Diagram



List of PL/SQL Queries for Project-Part 4

The below screenshot displays the list of cloud users using as part of project part 4.

The screenshot shows the Oracle Database Actions | Database Users interface. The top navigation bar includes 'Database Actions', 'Database Users', 'User Management', a search bar, and a user dropdown for 'ADMIN'. Below the header, there is a table listing five users:

User	Details
ORDS Alias: admin	Password Expires in 359 days https://gb9f4ae8ab54fc9-group8fdbproject.adb.us-chicago-1.oraclecloudapps.c...
GGADMIN	
RMAN\$VPC	
S116644475	REST Enabled ORDS Alias: s116644475 Password Expires in 359 days https://gb9f4ae8ab54fc9-group8fdbproject.adb.us-chicago-1.oraclecloudapps.c...
V_11642773	REST Enabled ORDS Alias: v_11642773 Password Expires in 359 days https://gb9f4ae8ab54fc9-group8fdbproject.adb.us-chicago-1.oraclecloudapps.c...
SR_11667743	REST Enabled ORDS Alias: sr_11667743 Password Expires in 359 days https://gb9f4ae8ab54fc9-group8fdbproject.adb.us-chicago-1.oraclecloudapps.c...

At the bottom of the interface, there are pagination controls ('Page Size: 20') and a note: '11:28:11 PM - REST call resolved successfully.' and 'Powered by ORDS'.

Figure-1: Displayed all the users.

Database Users Details:

V_11642773: https://gb9f4ae8ab54fc9-group8fdbproject.adb.us-chicago-1.oraclecloudapps.com/ords/v_11642773/_sdw/

SR_11667743: https://gb9f4ae8ab54fc9-group8fdbproject.adb.us-chicago-1.oraclecloudapps.com/ords/sr_11667743/_sdw/

S116644475: https://gb9f4ae8ab54fc9-group8fdbproject.adb.us-chicago-1.oraclecloudapps.com/ords/s116644475/_sdw/

Stored Procedures:

1. write a stored procedure when bill is generated for a new customer, the customer address should be recorded in the address table.

PL/SQL Code:

```
CREATE OR REPLACE PROCEDURE InsertNewCustomerAddress(
    v_first_name IN VARCHAR2,
    v_last_name VARCHAR2,
    v_dob VARCHAR2,
    v_driverlicense varchar2,
    v_phone varchar2,
    p_apartment_no IN NUMBER,
    p_street IN VARCHAR2,
    p_city IN VARCHAR2,
    p_state IN VARCHAR2,
    p_country IN VARCHAR2,
    p_zip IN CHAR,
    p_address_id OUT RAW,
    v_customer_id OUT RAW
)
IS
BEGIN
    -- Check if an address with the same details already exists
    SELECT id INTO p_address_id
    FROM SR_11667743.address
    WHERE apartment_no = p_apartment_no
        AND street = p_street
        AND city = p_city
        AND state = p_state
        AND country = p_country
        AND zip = p_zip;

    -- If an existing address is found, return its ID
    IF p_address_id IS NOT NULL THEN

        -- If no matching address is found, generate a new address ID
        dbms_output.put_line('Found address_id = ' || p_address_id);

        SELECT SYS_GUID() INTO v_customer_id FROM DUAL;

        INSERT INTO S116644475.customer(id, first_name, last_name, dob, driverlicense,
        phone, address_id)
```

```

    VALUES (v_customer_id, v_first_name, v_last_name, TO_TIMESTAMP(v_dob, 'DD-
MM-YYYY'), v_driverlicense, v_phone, p_address_id);
END IF;

EXCEPTION
WHEN NO_DATA_FOUND THEN
    SELECT SYS_GUID() INTO p_address_id FROM DUAL;
    dbms_output.put_line('New address_id = ' || p_address_id);

-- Insert the new address record
INSERT INTO SR_11667743.address(id, apartment_no, street, city, state, country,
zip) VALUES (p_address_id, p_apartment_no, p_street, p_city, p_state, p_country,
p_zip);

SELECT SYS_GUID() INTO v_customer_id FROM DUAL;

INSERT INTO S116644475.customer (id, first_name, last_name, dob, driverlicense,
phone, address_id) VALUES (v_customer_id, v_first_name, v_last_name,
TO_TIMESTAMP(v_dob, 'DD-MM-YYYY'), v_driverlicense, v_phone, p_address_id);
END;
/

```

Execution Code:

```

DECLARE
    v_first_name VARCHAR2(100) := 'vishnu';
    v_last_name VARCHAR2(100) := 'reddy';
    v_dob VARCHAR2(100) := '23-02-1999';
    v_driverlicense VARCHAR2(50) := '111111';
    v_phone VARCHAR2(50) := '(123)123-1238';
    v_apartment_no NUMBER := 101;
    v_street VARCHAR2(100) := '123 Main St';
    v_city VARCHAR2(100) := 'Anytown';
    v_state VARCHAR2(50) := 'CA';
    v_country VARCHAR2(50) := 'USA';
    v_zip CHAR(5) := '12345';
    v_address_id RAW(16);
    v_customer_id RAW(16);

```

```

BEGIN
    InsertNewCustomerAddress(
        v_first_name,
        v_last_name,
        v_dob,
        v_driverlicense,
        v_phone,
        v_apartment_no,
        v_street,
        v_city,
        v_state,
        v_country,
        v_zip,
        v_address_id,
        v_customer_id
    );
    DBMS_OUTPUT.PUT_LINE('Address ID: ' || v_address_id);
    DBMS_OUTPUT.PUT_LINE('customer ID: ' || v_customer_id);
END;
/

```

Code Execution in Cloud:

Screenshot-1:

The screenshot shows the Oracle Database Actions interface with a SQL worksheet. The code in the worksheet is a PL/SQL procedure named 'InsertNewCustomerAddress'. It takes several parameters (v_first_name, v_last_name, v_dob, v_driverlicense, v_phone, v_apartment_no, v_street, v_city, v_state, v_country, v_zip) and returns two raw values (v_address_id, v_customer_id). It also includes a check to see if an address with the same details already exists in the SR_11667743.address table.

```

CREATE OR REPLACE PROCEDURE InsertNewCustomerAddress(
    v_first_name IN VARCHAR2,
    v_last_name VARCHAR2,
    v_dob VARCHAR2,
    v_driverlicense varchar2,
    v_phone varchar2,
    p_apartment_no IN NUMBER,
    p_street IN VARCHAR2,
    p_city IN VARCHAR2,
    p_state IN VARCHAR2,
    p_country IN VARCHAR2,
    p_zip IN CHAR,
    p_address_id OUT RAW,
    v_customer_id OUT RAW
)
IS
BEGIN
    -- Check if an address with the same details already exists
    SELECT id INTO p_address_id
    FROM SR_11667743.address
    WHERE apartment_no = p_apartment_no
    AND street = p_street
    AND city = p_city
    AND state = p_state
    ...

```

The 'Script Output' tab at the bottom of the worksheet shows the message 'Procedure INSERTNEWCUSTOMERADDRESS compiled' and an elapsed time of 'Elapsed: 00:00:00.218'.

Screenshot-2:

```

1 CREATE OR REPLACE PROCEDURE InsertNewCustomerAddress(
2   v_first_name IN VARCHAR2,
3   v_last_name IN VARCHAR2,
4   v_dob VARCHAR2,
5   v_driverlicense varchar2,
6   v_phone varchar2,
7   v_apartment_no IN NUMBER,
8   p_street IN VARCHAR2,
9   p_city IN VARCHAR2,
10  p_state IN CHAR,
11  p_country IN VARCHAR2,
12  p_zip IN CHAR,
13  p_address_id OUT RAW,
14  v_customer_id OUT RAW
15 )
16 IS
17 BEGIN
18   -- Check if an address with the same details already exists
19   SELECT id INTO p_address_id
20   FROM SR_1166743.address
21   WHERE apartment_no = p_apartment_no
22   AND street = p_street
23   AND city = p_city
24   AND state = p_state
25   AND country = p_country
26   AND zip = p_zip;
27
28
29
30   -- If an existing address is found, return its ID
31   IF p_address_id IS NOT NULL THEN
32     -- If no matching address is found, generate a new address ID
33     dbms_output.put_line('Found address_id = ' || p_address_id);
34
35     SELECT SYS_GUID() INTO v_customer_id FROM DUAL;
36
37     INSERT INTO S116644475.customer(id, first_name, last_name, dob, driverlicense, phone, address_id)
38     VALUES (v_customer_id, v_first_name, v_last_name, TO_TIMESTAMP(v_dob, 'DD-MM-YYYY'), v_driverlicense, v_phone, p_address_id);
39   END IF;
40
41   EXCEPTION
42   WHEN NO_DATA_FOUND THEN
43     SELECT SYS_GUID() INTO p_address_id FROM DUAL;
44     dbms_output.put_line('New address_id = ' || p_address_id);
45
46   -- Insert the new address record
47   INSERT INTO SR_1166743.address(id, apartment_no, street, city, state, country, zip)
48   VALUES (p_address_id, p_apartment_no, p_street, p_city, p_state, p_country, p_zip);
49
50   SELECT SYS_GUID() INTO v_customer_id FROM DUAL;
51
52   INSERT INTO S116644475.customer(id, first_name, last_name, dob, driverlicense, phone, address_id)
53   VALUES (v_customer_id, v_first_name, v_last_name, TO_TIMESTAMP(v_dob, 'DD-MM-YYYY'), v_driverlicense, v_phone, p_address_id);
54
55 END;
56 /

```

Powered by ORDS

Screenshot-3:

```

57 DECLARE
58   v_first_name VARCHAR2(100) := 'Vishnu';
59   v_last_name VARCHAR2(100) := 'reddy';
60   v_dob VARCHAR2(100) := '23-02-1990';
61   v_driverlicense VARCHAR2(100) := '111111';
62   v_phone VARCHAR2(100) := '(123)123-1230';
63   v_apartment_no NUMBER;
64   p_street VARCHAR2(100) := '123 Main St';
65   v_city VARCHAR2(100) := 'Anytown';
66   v_state VARCHAR2(100) := 'CA';
67   v_country VARCHAR2(100) := 'USA';
68   v_zip CHAR(5) := '12345';
69   v_address_id RAW(10);
70   v_customer_id RAW(10);
71
72 BEGIN
73   InsertNewCustomerAddress(
74     v_first_name,
75     v_last_name,
76     v_dob,
77     v_driverlicense,
78     v_phone,
79     v_apartment_no,
80     v_street,
81     v_city,
82     v_state,
83     v_zip,
84     v_address_id,
85     v_customer_id
86   );
87   DBMS_OUTPUT.PUT_LINE('Address ID: ' || v_address_id);
88   DBMS_OUTPUT.PUT_LINE('Customer ID: ' || v_customer_id);
89 END;
90
91

```

Query Result Script Output DBMS Output Explain Plan Autotrace SQL History

Procedure INSERTNEWCUSTOMERADDRESS compiled
Elapsed: 00:00:00.218

Found address_id = 8201
Address ID: 6999F9A5D0E5436E8632810000A8343
customer ID: 6999F9A5D0E5436E8632810000A8343

PL/SQL procedure successfully completed.
Elapsed: 00:00:00.613

Powered by ORDS

Explanation: The procedure, `InsertNewCustomerAddress`, is designed to insert a new customer's address into the database and associate it with a customer record. This procedure inserts a new customer's address into the database, either by reusing an existing address or creating a new one and associates it with a new customer record. It ensures that the address is uniquely identified and linked to the customer.

2. **create a stored procedure that prints the list of inventory items belonging to a automotive retailer having more than 1 employees working in it**

PL/SQL Code:

```
CREATE OR REPLACE PROCEDURE GetInventoryItemsForRetailerWithMoreThanOneEmployees
AS
BEGIN
    FOR retailer_rec IN (SELECT a.Name AS retailer_name
                          FROM automotive_retailer a
                          JOIN V_11642773.Employee e ON a.id = e.automotive_retailer_id
                          GROUP BY a.Name
                          HAVING COUNT(e.id) > 1) LOOP
        DBMS_OUTPUT.PUT_LINE('Inventory items for Automotive Retailer: ' || 
retailer_rec.retailer_name);

        FOR inventory_rec IN (SELECT ip.name
                              FROM inventory_product ip
                              WHERE ip.automotive_retailer_id = (SELECT id FROM automotive_retailer
WHERE Name = retailer_rec.retailer_name)) LOOP
            DBMS_OUTPUT.PUT_LINE(' - ' || inventory_rec.name);
        END LOOP;
    END LOOP;
END;
/
```

Execution Code:

```
BEGIN
    GetInventoryItemsForRetailerWithMoreThanOneEmployees;
END;
```

Code Execution in Cloud:

Screenshot-1:

The screenshot shows the Oracle Database Actions interface with a SQL worksheet. The code is a PL/SQL procedure named GetInventoryItemsForRetailerWithMoreThanOneEmployees. It uses a cursor to find retailers with more than one employee, then loops through their inventory items. The code includes several commented-out sections and placeholder values.

```
1 --2. create a stored procedure that prints the list of inventory items belonging to a automotive retailer having more than 1 employees working in it
2
3 INSERT INTO V_11642773.Employee (id, first_name, last_name, dob, phone, email, annual_salary, hire_date, ssn, automotive_retailer_id, address_id)
4 VALUES ('615', 'J', 'Doe', TIMESTAMP '1990-05-15 00:00:00', '(155)555-5555', 'jdoe@example.com', 60000, TIMESTAMP '2022-03-20 08:00:00', '123456781', '501', '201');
5
6 CREATE OR REPLACE PROCEDURE GetInventoryItemsForRetailerWithMoreThanOneEmployees AS
7 BEGIN
8     FOR retailer_rec IN (SELECT a.Name AS retailer_name
9                           FROM automotive_retailer a
10                          JOIN V_11642773.Employee e ON a.id = e.automotive_retailer_id
11                         GROUP BY a.Name
12                         HAVING COUNT(e.id) > 1) LOOP
13         DBMS_OUTPUT.PUT_LINE('Inventory items for Automotive Retailer: ' || retailer_rec.retailer_name);
14
15         FOR inventory_rec IN (SELECT ip.name
16                                FROM inventory_product ip
17                               WHERE ip.automotive_retailer_id = (SELECT id FROM automotive_retailer WHERE Name = retailer_rec.retailer_name)
18                             ) LOOP
19             DBMS_OUTPUT.PUT_LINE(' - ' || inventory_rec.name);
20         END LOOP;
21     END LOOP;
22 END;
23 /
24
25
26
27 BEGIN
28     GetInventoryItemsForRetailerWithMoreThanOneEmployees;
29 END;
```

At the bottom of the screen, a message indicates "12:38:09 AM - Code execution finished."

Screenshot-2

The screenshot shows the Oracle Database Actions interface with a SQL worksheet. The same PL/SQL procedure is run, and the results are displayed in the "Query Result" tab. The output shows the procedure was compiled and took 0.00:00:00.065 seconds. The results show inventory items for a retailer named "Car Dealership 1".

Procedure GETINVENTORYITEMSFORRETAILERWITHMORETHANONEEMPLOYEES compiled
Elapsed: 00:00:00.065

Inventory items for Automotive Retailer: Car Dealership 1
- Product 1
- Product 4
- Product 7
- Product 10

PL/SQL procedure successfully completed.
Elapsed: 00:00:00.024

At the bottom of the screen, a message indicates "12:38:09 AM - Code execution finished."

Explanation: This procedure, named GetInventoryItemsForRetailerWithMoreThan5Employees, is designed to retrieve and display the inventory items for automotive retailers that have more

than 5 employees. The procedure identifies automotive retailers with more than 5 employees, and for each of these retailers, it lists the names of the inventory items they have for sale.

3. Write a stored procedure to retrieve the top suppliers' information including their names and the total quantity supplied using PL/SQL.

PL/SQL Code:

```
CREATE OR REPLACE PROCEDURE GenerateTopSuppliersReport AS
BEGIN
    FOR supplier_rank_rec IN (SELECT s.name, SUM(ips.quantity) AS total_quantity_supplied
                               FROM ADMIN.supplier s
                               JOIN ADMIN.inventory_product_supplier ips ON s.id = ips.supplier_id
                               GROUP BY s.name
                               ORDER BY total_quantity_supplied DESC) LOOP
        DBMS_OUTPUT.PUT_LINE('Supplier: ' || supplier_rank_rec.name);
        DBMS_OUTPUT.PUT_LINE('Total Quantity Supplied: ' ||
                             supplier_rank_rec.total_quantity_supplied);
        DBMS_OUTPUT.PUT_LINE('-----');
    END LOOP;
END;
/
```

Execution Code:

```
BEGIN
    GenerateTopSuppliersReport;
END;
/
```

Code Execution in Cloud:

Screenshot-1:

The screenshot shows the Oracle Database Actions SQL Worksheet interface. The code in the worksheet is:

```
1 CREATE OR REPLACE PROCEDURE GenerateTopSuppliersReport AS
2 BEGIN
3     FOR supplier_rank_rec IN (SELECT s.name, SUM(ips.quantity) AS total_quantity_supplied
4         FROM ADMIN.supplier s
5             JOIN ADMIN.inventory_product_supplier ips ON s.id = ips.supplier_id
6             GROUP BY s.name
7             ORDER BY total_quantity_supplied DESC) LOOP
8         DBMS_OUTPUT.PUT_LINE('Supplier: ' || supplier_rank_rec.name);
9         DBMS_OUTPUT.PUT_LINE('Total Quantity Supplied: ' || supplier_rank_rec.total_quantity_supplied);
10        DBMS_OUTPUT.PUT_LINE('-----');
11    END LOOP;
12
13 /
14
15 BEGIN
16     GenerateTopSuppliersReport;
17 END;
18
19 /
```

The results pane shows the output of the procedure execution:

```
Procedure GENERATETOPSUPPLIERSREPORT compiled
Elapsed: 00:00:00.009

Supplier: Supplier 7
Total Quantity Supplied: 30
-----
Supplier: Supplier 6
Total Quantity Supplied: 25
-----
Supplier: Supplier 9
Total Quantity Supplied: 22
-----
Supplier: Supplier 3
Total Quantity Supplied: 20
-----
Supplier: Supplier 8
Total Quantity Supplied: 18
-----
Supplier: Supplier 10
Total Quantity Supplied: 16
-----
Supplier: Supplier 2
Total Quantity Supplied: 15
-----
Supplier: Supplier 5
Total Quantity Supplied: 12
-----
Supplier: Supplier 1
Total Quantity Supplied: 10
```

At the bottom left, it says "124708 AM - Code execution finished."

Screenshot-2:

The screenshot shows the Oracle Database Actions SQL Worksheet interface. The code in the worksheet is identical to Screenshot-1:

```
1 CREATE OR REPLACE PROCEDURE GenerateTopSuppliersReport AS
2 BEGIN
3     FOR supplier_rank_rec IN (SELECT s.name, SUM(ips.quantity) AS total_quantity_supplied
4         FROM ADMIN.supplier s
5             JOIN ADMIN.inventory_product_supplier ips ON s.id = ips.supplier_id
6             GROUP BY s.name
7             ORDER BY total_quantity_supplied DESC) LOOP
8         DBMS_OUTPUT.PUT_LINE('Supplier: ' || supplier_rank_rec.name);
9         DBMS_OUTPUT.PUT_LINE('Total Quantity Supplied: ' || supplier_rank_rec.total_quantity_supplied);
10        DBMS_OUTPUT.PUT_LINE('-----');
11    END LOOP;
12
13 /
14
15 BEGIN
16     GenerateTopSuppliersReport;
17 END;
18
19 /
```

The results pane shows the output of the procedure execution:

```
Procedure GENERATETOPSUPPLIERSREPORT compiled
Elapsed: 00:00:00.009

Supplier: Supplier 7
Total Quantity Supplied: 30
-----
Supplier: Supplier 6
Total Quantity Supplied: 25
-----
Supplier: Supplier 9
Total Quantity Supplied: 22
-----
Supplier: Supplier 3
Total Quantity Supplied: 20
-----
Supplier: Supplier 8
Total Quantity Supplied: 18
-----
Supplier: Supplier 10
Total Quantity Supplied: 16
-----
Supplier: Supplier 2
Total Quantity Supplied: 15
-----
Supplier: Supplier 5
Total Quantity Supplied: 12
-----
Supplier: Supplier 1
Total Quantity Supplied: 10
```

At the bottom left, it says "124708 AM - Code execution finished."

Screenshot-3:

The screenshot shows the Oracle Database Actions | SQL interface in Oracle SQL Developer. The left sidebar displays a tree view of database objects under the 'ADMIN' schema, including AUTOMOBILE, AUTOMOTIVE_RETAILER, INVENTORY_PRODUCT, INVENTORY_PRODUCT_SUPPLIER, and SUPPLIER. The central workspace contains a code editor with the following PL/SQL procedure:

```
1 CREATE OR REPLACE PROCEDURE GenerateTopSuppliersReport AS
2 BEGIN
3     FOR supplier_rank_rec IN (SELECT s.name, SUM(ips.quantity) AS total_quantity_supplied
4                                 FROM ADMIN.supplier s
5                                 JOIN ADMIN.inventory_product_supplier ips ON s.id = ips.supplier_id
6                                 GROUP BY s.name)
7     LOOP
8         DBMS_OUTPUT.PUT_LINE('Supplier: ' || supplier_rank_rec.name);
9         DBMS_OUTPUT.PUT_LINE('Total Quantity Supplied: ' || supplier_rank_rec.total_quantity_supplied);
10    END LOOP;
11 END;
```

The 'Query Result' tab is selected, showing the output of the procedure:

```
Supplier: Supplier 6
Total Quantity Supplied: 25
-----
Supplier: Supplier 9
Total Quantity Supplied: 22
-----
Supplier: Supplier 3
Total Quantity Supplied: 20
-----
Supplier: Supplier 8
Total Quantity Supplied: 18
-----
Supplier: Supplier 10
Total Quantity Supplied: 16
-----
Supplier: Supplier 2
Total Quantity Supplied: 15
-----
Supplier: Supplier 5
Total Quantity Supplied: 12
-----
Supplier: Supplier 1
Total Quantity Supplied: 10
-----
Supplier: Supplier 4
Total Quantity Supplied: 8
-----
```

Below the results, a message indicates the procedure was successfully completed, and the elapsed time is shown as 00:00:00.015.

Explanation: This procedure, "GenerateTopSuppliersReport," is designed to calculate and display a report that lists the top suppliers in the database based on the total quantity of products they have supplied, with the highest-supplying suppliers appearing at the top of the report. It retrieves data by joining the supplier and inventory_product_supplier tables, grouping by supplier and calculating the total quantity supplied, and then ranks the suppliers in descending order.

- 4. Write a stored procedure to retrieve a list of customers who have purchased specific inventory products and provides details of their purchases.**

PL/SQL Code:

```
CREATE OR REPLACE PROCEDURE GetCustomersWithSpecificPurchases(
    product_name IN VARCHAR2
) AS
BEGIN
    FOR customer_rec IN (
        SELECT c.first_name, c.last_name, ip.name AS product_name, b.bill_date
        FROM S116644475.customer c
        JOIN bill b ON c.id = b.customer_id
        JOIN bill_inventory_product bp ON b.id = bp.bill_id
        JOIN admin.inventory_product ip ON bp.inventory_product_id = ip.id
        WHERE ip.name = product_name
    ) LOOP
        DBMS_OUTPUT.PUT_LINE('Customer: ' || customer_rec.first_name || ' ' ||
customer_rec.last_name);
        DBMS_OUTPUT.PUT_LINE('Purchased Product: ' || customer_rec.product_name);
        DBMS_OUTPUT.PUT_LINE('Purchase Date: ' || TO_CHAR(customer_rec.bill_date, 'DD-
MON-YYYY'));
        DBMS_OUTPUT.NEW_LINE;
    END LOOP;
END;
/
```

Execution Code:

```
BEGIN
    GetCustomersWithSpecificPurchases('Product 1');
END;
/
```

Code Execution in Cloud:

Screenshot-1:

The screenshot shows the Oracle SQL Developer interface. In the top-left, the Navigator pane displays a schema named SR_11667743 with tables ADDRESS, BILL, BILL_INVENTORY_PRODUCT, and PART_SERVICE. The central workspace contains the following PL/SQL code:

```
1 -- 4. This procedure retrieves a list of customers who have purchased specific inventory products and provides details of t
2
3 CREATE OR REPLACE PROCEDURE GetCustomersWithSpecificPurchases(
4     product_name IN VARCHAR2
5 ) AS
6 BEGIN
7     FOR customer_rec IN (
8         SELECT c.first_name, c.last_name, ip.name AS product_name, b.bill_date
9             FROM S116644475.customer c
10            JOIN bill b ON c.id = b.customer_id
11            JOIN bill_inventory_product bp ON b.id = bp.bill_id
12            JOIN admin.inventory_product ip ON bp.inventory_product_id = ip.id
13            WHERE ip.name = product_name
14    ) LOOP
15        DBMS_OUTPUT.PUT_LINE('Customer: ' || customer_rec.first_name || ' ' || customer_rec.last_name);
16        DBMS_OUTPUT.PUT_LINE('Purchased Product: ' || customer_rec.product_name);
17        DBMS_OUTPUT.PUT_LINE('Purchase Date: ' || TO_CHAR(customer_rec.bill_date, 'DD-MON-YYYY'));
18        DBMS_OUTPUT.NEW_LINE;
19    END LOOP;
20 END;
21 /
```

The bottom pane shows the results of the execution:

Procedure GETCUSTOMERSWITHSPECIFICPURCHASES compiled
Elapsed: 00:00:00.008

Powered by ORDS

Screenshot-2:

The screenshot shows the Oracle SQL Developer interface. The Navigator pane displays the same schema as in Screenshot-1. The central workspace contains the following PL/SQL code:

```
9      FROM S116644475.customer c
10     JOIN bill b ON c.id = b.customer_id
11     JOIN bill_inventory_product bp ON b.id = bp.bill_id
12     JOIN admin.inventory_product ip ON bp.inventory_product_id = ip.id
13     WHERE ip.name = product_name
14  ) LOOP
15      DBMS_OUTPUT.PUT_LINE('Customer: ' || customer_rec.first_name || ' ' || customer_rec.last_name);
16      DBMS_OUTPUT.PUT_LINE('Purchased Product: ' || customer_rec.product_name);
17      DBMS_OUTPUT.PUT_LINE('Purchase Date: ' || TO_CHAR(customer_rec.bill_date, 'DD-MON-YYYY'));
18      DBMS_OUTPUT.NEW_LINE;
19  END LOOP;
20 END;
21 /
23
24 BEGIN
25     GetCustomersWithSpecificPurchases('Product 1');
26 END;
27 /
28
29 /
```

The bottom pane shows the results of the execution:

Customer: John Doe
Purchased Product: Product 1
Purchase Date: 31-OCT-2023

PL/SQL procedure successfully completed.
Elapsed: 00:00:00.008

Powered by ORDS

Explanation: This procedure named "GetCustomersWithSpecificPurchases" is designed to retrieve and display information about customers who have made specific purchases of a product. It is to find and list customers who have purchased a specific product, providing their names, the product's name, and the date of purchase as output. The procedure can be called by passing the desired product name as an argument, and it will output information about customers who purchased that specific product.

5. Write a stored procedure to get the list of employee who can train the employees on specific automobile(Input param - automobile)

PL/SQL Code:

```
CREATE OR REPLACE PROCEDURE GetTrainersForAutomobile(automobile_id_in IN RAW) AS
BEGIN
    FOR trainer_rec IN (
        SELECT DISTINCT e.id, e.first_name, e.last_name
        FROM v_11642773.Employee e
        JOIN v_11642773.job_employee je ON e.id = je.employee_id
        JOIN v_11642773.job j ON je.job_id = j.id
        WHERE j.automobile_id = automobile_id_in
    ) LOOP
        DBMS_OUTPUT.PUT_LINE('_____');
        DBMS_OUTPUT.PUT_LINE('Employee ID: ' || trainer_rec.id);
        DBMS_OUTPUT.PUT_LINE('Name: ' || trainer_rec.first_name || ' ' ||
trainer_rec.last_name);
        DBMS_OUTPUT.PUT_LINE('_____');
    END LOOP;
END;
/
```

Execution Code:

```
BEGIN
    GetTrainersForAutomobile('1');
END;
/
```

Code Execution in Cloud:

Screenshot-1:

The screenshot shows the Oracle SQL Developer interface. The code editor displays a PL/SQL procedure named "GETTRAINERSFORAUTOMOBILE". The procedure retrieves distinct employee IDs, first names, and last names from the "Employee" table, joining it with the "Job_Employee" and "Job" tables to find employees associated with a specific automobile. The output is formatted using DBMS_OUTPUT.PUT_LINE statements. The code is highlighted in blue and black, indicating syntax and comments. Below the code, the message "Procedure GETTRAINERSFORAUTOMOBILE compiled" is displayed, along with the execution time "Elapsed: 00:00:00.009".

```
-- 5. Get the list of employee who can train the employees on specific automobile(Input param - automobile)
CREATE OR REPLACE PROCEDURE GetTrainersForAutomobile(automobile_id_in IN RAW) AS
BEGIN
    FOR trainer_rec IN (
        SELECT DISTINCT e.id, e.first_name, e.last_name
        FROM v_11642773.Employee e
        JOIN v_11642773.job_employee je ON e.id = je.employee_id
        JOIN v_11642773.job j ON je.job_id = j.id
        WHERE j.automobile_id = automobile_id_in
    ) LOOP
        DBMS_OUTPUT.PUT_LINE('-----');
        DBMS_OUTPUT.PUT_LINE('Employee ID: ' || trainer_rec.id);
        DBMS_OUTPUT.PUT_LINE('Name: ' || trainer_rec.first_name || ' ' || trainer_rec.last_name);
        DBMS_OUTPUT.PUT_LINE('-----');
    END LOOP;
END;
/
BEGIN
GetTrainersForAutomobile('1');
END;
/

```

Screenshot-2:

The screenshot shows the Oracle SQL Developer interface after executing the procedure. The "Script Output" tab is selected, displaying the output of the DBMS_OUTPUT.PUT_LINE statements. The output lists two employees: Jane Smith (Employee ID 6602) and John Doe (Employee ID 6601), each followed by a separator line. Below the output, the message "PL/SQL procedure successfully completed." is shown, along with the execution time "Elapsed: 00:00:00.020".

```
Employee ID: 6602
Name: Jane Smith
-----
Employee ID: 6601
Name: John Doe
-----
PL/SQL procedure successfully completed.
Elapsed: 00:00:00.020
```

Explanation: This procedure "GetTrainersForAutomobile," retrieves and displays the names of trainers who have been involved in jobs associated with a given automobile, helping to identify the individuals responsible for servicing or working on that vehicle. It takes an automobile ID as

input and retrieves data by joining the Employee, job_employee, and job tables to find employees who have worked on the specific automobile and can be considered trainers.

Stored Functions:

- 1. when a function to update the address of the employee with new address.**

PL/SQL Code:

```
CREATE OR REPLACE FUNCTION update_employee_address(
    p_employee_id IN RAW,
    p_apartment_no IN NUMBER,
    p_street IN VARCHAR2,
    p_city IN VARCHAR2,
    p_state IN VARCHAR2,
    p_country IN VARCHAR2,
    p_zip IN CHAR
) RETURN RAW IS
    v_address_id RAW(16);

    -- Declare a cursor to fetch the address data
    CURSOR address_cursor IS
        SELECT id
        FROM SR_11667743.address
        WHERE
            apartment_no = p_apartment_no
            AND street = p_street
            AND city = p_city
            AND state = p_state
            AND country = p_country
            AND zip = p_zip;
    BEGIN
        -- Initialize the address_id to NULL
        v_address_id := NULL;

        -- Fetch data using the cursor
        OPEN address_cursor;
        FETCH address_cursor INTO v_address_id;
        CLOSE address_cursor;

        -- If no data found, insert a new address
        IF v_address_id IS NULL THEN
            INSERT INTO SR_11667743.address (id, apartment_no, street, city, state, country, zip)
            VALUES (SYS_GUID(), p_apartment_no, p_street, p_city, p_state, p_country, p_zip)
        END IF;
    END;
```

```

    RETURNING id INTO v_address_id;
END IF;

-- Update the employee's address_id with the new or existing address_id
UPDATE V_11642773.Employee
SET address_id = v_address_id
WHERE id = p_employee_id;

COMMIT; -- Commit the transaction

-- Return the address ID
RETURN v_address_id;
END update_employee_address;
/

```

Execution Code:

```

DECLARE
  v_employee_id RAW(16) := '612';
  v_apartment_no NUMBER := 123;
  v_street VARCHAR2(100) := '122 Main St';
  v_city VARCHAR2(100) := 'Denton';
  v_state VARCHAR2(50) := 'TX';
  v_country VARCHAR2(50) := 'USA';
  v_zip CHAR(5) := '12345';
  v_address_id RAW(16);
BEGIN
  v_address_id := update_employee_address(
    p_employee_id => v_employee_id,
    p_apartment_no => v_apartment_no,
    p_street => v_street,
    p_city => v_city,
    p_state => v_state,
    p_country => v_country,
    p_zip => v_zip
  );
  DBMS_OUTPUT.PUT_LINE('Address ID: ' || v_address_id);
END;
/

```

Code Execution in Cloud:

Screenshot-1:

The screenshot shows the Oracle Database Actions interface. The top navigation bar includes 'ORACLE Database Actions | SQL', a search bar, and user information for 'ADMIN'. The left sidebar has sections for 'Navigator' (ADMIN), 'Tables' (AUTOMOBILE, AUTOMOTIVE_RETAILER, INVENTORY_PRODUCT, INVENTORY_PRODUCT_SUPPLIER, SUPPLIER), and a 'Search...' field. The main workspace is titled '[Worksheet]'. It contains a PL/SQL script to create a function named 'update_employee_address'. The script defines parameters for address components (p_employee_id, p_apartment_no, p_street, p_city, p_state, p_country, p_zip) and returns a RAW(16) value representing an address ID. It also declares a cursor 'address_cursor' to select from the SR_11667743.address table where address components match the input parameters. The script ends with a 'RFCTN' command. Below the script, the 'Query Result' tab is selected, showing the output of the compilation and execution of the function. The output indicates the function was compiled successfully in 0.000 seconds, and the PL/SQL procedure completed in 0.013 seconds. The address ID generated is 094B0EA736F50658E0632818000AF5FF.

```
1 CREATE OR REPLACE FUNCTION update_employee_address(
2   p_employee_id IN RAW,
3   p_apartment_no IN NUMBER,
4   p_street IN VARCHAR2,
5   p_city IN VARCHAR2,
6   p_state IN VARCHAR2,
7   p_country IN VARCHAR2,
8   p_zip IN CHAR
9 ) RETURN RAW IS
10   v_address_id RAW(16);
11
12   -- Declare a cursor to fetch the address data
13   CURSOR address_cursor IS
14     SELECT id
15       FROM SR_11667743.address
16      WHERE
17        apartment_no = p_apartment_no
18        AND street = p_street
19        AND city = p_city
20        AND state = p_state
21        AND country = p_country
22        AND zip = p_zip;
23
24 RFCTN
```

Query Result Script Output DBMS Output Explain Plan Autotrace SQL History

Function UPDATE_EMPLOYEE_ADDRESS compiled
Elapsed: 00:00:00.009

Address ID: 094B0EA736F50658E0632818000AF5FF

PL/SQL procedure successfully completed.
Elapsed: 00:00:00.013

Screenshot-2:

The screenshot shows the Oracle Database Actions | SQL interface. The left sidebar has 'Navigator' selected, showing 'ADMIN' as the consumer group. Under 'Tables', there are several entries: AUTOMOBILE, AUTOMOTIVE_RETAILER, INVENTORY_PRODUCT, INVENTORY_PRODUCT_SUPPLIER, and SUPPLIER. A search bar 'Search...' is also present. The main workspace is titled '[Worksheet]'. It contains the following PL/SQL code:

```
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75
```

```
DECLARE
  v_employee_id RAW(16) := '610';
  v_apartment_no NUMBER := 123;
  v_street VARCHAR2(100) := '122 Main St';
  v_city VARCHAR2(100) := 'Denton';
  v_state VARCHAR2(50) := 'TX';
  v_country VARCHAR2(50) := 'USA';
  v_zip CHAR(5) := '12345';
  v_address_id RAW(16);
BEGIN
  v.address_id := update_employee_address(
    p_employee_id => v.employee_id,
    p_apartment_no => v.apartment_no,
    p_street => v.street,
    p_city => v.city,
    p_state => v.state,
    p_country => v.country,
    p_zip => v.zip
  );
  -- You can use v_address_id as needed here.
END;
DBMS_OUTPUT.PUT_LINE('Address ID: ' || v_address_id);
```

The 'Query Result' tab is selected, showing the output:

```
Function UPDATE_EMPLOYEE_ADDRESS compiled
Elapsed: 00:00:00.009

Address ID: 09480EA136F58658E0632810800A05FF

PL/SQL procedure successfully completed.
Elapsed: 00:00:00.013
```

Explanation: This function updates an employee(s) address. It checks if the new address already exists in the database.

If it does, it links the employee to that address. If not, it creates a new address entry and links the employee to it. Finally, it returns the address ID.

2. write a stored function to retrieve eligible bonus employees. Eligible bonus employees are who have held more than 3 jobs within a defined date range.

PL/SQL Code:

```
CREATE OR REPLACE FUNCTION get_bonus_eligible_employees(
    p_start_date TIMESTAMP,
    p_end_date TIMESTAMP
) RETURN SYS_REFCURSOR IS
    bonus_employees_cursor SYS_REFCURSOR;
BEGIN
    OPEN bonus_employees_cursor FOR
        SELECT je.employee_id
        FROM V_11642773.job_employee je
        JOIN V_11642773.job j ON je.job_id = j.id
        WHERE j.job_date >= p_start_date
        AND j.job_date <= p_end_date
        GROUP BY je.employee_id
        HAVING COUNT(*) >= 3;

    RETURN bonus_employees_cursor;
END;
/
```

Execution Code:

```
DECLARE
    bonus_employee_cursor SYS_REFCURSOR;
    start_date TIMESTAMP;
    end_date TIMESTAMP;
    v_employee_id RAW(16);
BEGIN
    start_date := TO_TIMESTAMP('2023-10-22 00:00:00','YYYY-MM-DD HH24:MI:SS');
    end_date := TO_TIMESTAMP('2023-10-30 23:59:59','YYYY-MM-DD HH24:MI:SS');

    bonus_employee_cursor := get_bonus_eligible_employees(start_date, end_date);

    LOOP
        FETCH bonus_employee_cursor INTO v_employee_id;
        EXIT WHEN bonus_employee_cursor%NOTFOUND;
```

```

DBMS_OUTPUT.PUT_LINE('Bonus Eligible Employee ID: ' || v_employee_id);
END LOOP;

CLOSE bonus_employee_cursor;
END;
/

```

Code Execution in Cloud:

Screenshot-1:

The screenshot shows the Oracle Database Actions SQL Worksheet interface. The code area contains the following PL/SQL function definition:

```

CREATE OR REPLACE FUNCTION get_bonus_eligible_employees(
    p_start_date TIMESTAMP,
    p_end_date TIMESTAMP)
RETURN SYS_REFCURSOR
IS
    bonus_employees_cursor SYS_REFCURSOR;
BEGIN
    OPEN bonus_employees_cursor FOR
        SELECT je.employee_id
        FROM V_1164273.job_employee je
        JOIN V_1164273.job j ON je.job_id = j.id
        WHERE j.job_date >= p_start_date
        AND j.job_date <= p_end_date
        GROUP BY je.employee_id
        HAVING COUNT(*) >= 3;
    RETURN bonus_employees_cursor;
END;

```

The "Script Output" tab is selected, showing the results of the compilation:

```

Function GET_BONUS_ELIGIBLE_EMPLOYEES compiled
Elapsed: 00:00:00.009

```

The status bar at the bottom indicates: "Powered by ORDS".

Screenshot-2:

The screenshot shows the Oracle Database Actions SQL Worksheet interface. The top navigation bar includes 'ORACLE Database Actions | SQL', 'Search' (with a magnifying glass icon), and 'ADMIN'. Below the navigation is a toolbar with various icons. The main workspace displays a PL/SQL script in the 'Script Output' tab. The script declares variables, sets date ranges, opens a cursor, loops through it, and outputs results using DBMS_OUTPUT.PUT_LINE. The bottom section shows the 'Query Result' tab with the output of the script, which lists two employee IDs: 0633 and 0631. The status bar at the bottom indicates the URL as https://gb9f4ae8ab54fc9-group8fdbproject.adb.us-chicago-1.oraclecloudapps.com/ords/admin/_sdw/?nav=worksheet#.

```

1  DECLARE
2      bonus_employee_cursor SYS_REFCURSOR;
3      start_date TIMESTAMP;
4      end_date TIMESTAMP;
5      v_employee_id RAW(16);
6  BEGIN
7      start_date := TO_TIMESTAMP('2023-10-22 00:00:00','YYYY-MM-DD HH24:MI:SS');
8      end_date := TO_TIMESTAMP('2023-10-30 23:59:59','YYYY-MM-DD HH24:MI:SS');
9
10     bonus_employee_cursor := get_bonus_eligible_employees(start_date, end_date);
11
12     LOOP
13         FETCH bonus_employee_cursor INTO v_employee_id;
14         EXIT WHEN bonus_employee_cursor%NOTFOUND;
15
16         DBMS_OUTPUT.PUT_LINE('Bonus Eligible Employee ID: ' || v_employee_id);
17     END LOOP;
18
19     CLOSE bonus_employee_cursor;
20
21 END;
22

```

Elapsed: 00:00:00.013
Bonus Eligible Employee ID: 0633
Bonus Eligible Employee ID: 0631
PL/SQL procedure successfully completed.
Elapsed: 00:00:00.010

Explanation: Explanation: "update_employee_salary" that figures out which employees have done more than 3 jobs during a specific time frame and gives their Employee IDs in a list. In the execution part, we set a date range (start_date and end_date), and then we use the function to find the Employee IDs who meet the job criteria during that time.

3 write a stored function to return the list of customer ID's whose insurance status is 'active'.

PL/SQL Code:

```

CREATE OR REPLACE FUNCTION get_active_insured_customers
RETURN SYS_REFCURSOR IS
    v_cursor SYS_REFCURSOR;
BEGIN
    OPEN v_cursor FOR
    SELECT c.id AS customer_ids
    FROM s116644475.customer c
    JOIN s116644475.customer_insurance ci ON c.id = ci.customer_id
    WHERE ci.status = 'active';

    RETURN v_cursor;
END get_active_insured_customers;
/

```

Execution Code:

```
DECLARE
    v_customer_id RAW(16);
    v_cursor SYS_REFCURSOR;
BEGIN
    v_cursor := get_active_insured_customers;

    LOOP
        FETCH v_cursor INTO v_customer_id;
        EXIT WHEN v_cursor%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE('Customer ID: ' || v_customer_id );
    END LOOP;

    CLOSE v_cursor;
END;
/
```

Code Execution in Cloud:

Screenshot-1:

The screenshot shows the Oracle Database Actions interface with a SQL worksheet open. The code in the worksheet is a PL/SQL function named GET_ACTIVE_INSURED_CUSTOMERS. It declares a cursor variable v_cursor, opens it, and loops until it is not found. Inside the loop, it fetches the value of v_customer_id and prints it to the DBMS_OUTPUT. After the loop, it closes the cursor. The function ends with a RETURN statement. The code is annotated with comments explaining its purpose and structure.

```
-- 3. write a function to return the list of customer's whose insurance status is 'active'
CREATE OR REPLACE FUNCTION get_active_insured_customers
RETURN SYS_REFCURSOR IS
    v_cursor SYS_REFCURSOR;
BEGIN
    OPEN v_cursor FOR
        SELECT c.id AS customer_ids
        FROM s11664475.customer c
        JOIN s11664475.customer_insurance ci ON c.id = ci.customer_id
        WHERE ci.status = 'active';
    RETURN v_cursor;
END get_active_insured_customers;
/
-- Execution:
DECLARE
    v_customer_id RAW(16);
    v_cursor SYS_REFCURSOR;
BEGIN
    v_cursor := get_active_insured_customers;
    LOOP
        FETCH v_cursor INTO v_customer_id;
        EXIT WHEN v_cursor%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE('Customer ID: ' || v_customer_id );
    END LOOP;
    CLOSE v_cursor;
END;
```

Below the code, the message "Function GET_ACTIVE_INSURED_CUSTOMERS compiled" is displayed, along with the elapsed time "Elapsed: 00:00:00.170". The interface includes a Navigator pane on the left showing tables like EMPLOYEE, EMPLOYEE_PAYROLL, JOB, and JOB_EMPLOYEE, and various tabs at the bottom like Query Result, Script Output, DBMS Output, Explain Plan, Autotrace, and SQL History.

Screenshot-2:

```

10    OPEN v_cursor FOR
11      SELECT DISTINCT b.customer_id
12        FROM SR_11667743.bill b
13          WHERE b.bill_date = p_target_date
14            AND b.sale_type = 'OFFLINE'
15
16  RETURN v_cursor;
17  END get_active_insured_customers;
18 /

```

Customer ID: 0991
Customer ID: 0992
Customer ID: 0994
Customer ID: 0996
Customer ID: 0997
Customer ID: 0999

PL/SQL procedure successfully completed.

Explanation: This function, named 'get_active_insured_customers,' does the job of finding and listing customer IDs with active insurance status. It uses a database cursor to hold these customer IDs and a SQL query to fetch the relevant information from the 'customer' and 'customer_insurance' tables. When executed, it prints out these customer IDs one by one. Essentially, it helps to identify and display the customers who currently have active insurance policies.

4. write a stored function to return the list of customerId's, who did payment using CASH in autozone on particular date.

PL/SQL Code:

```

CREATE OR REPLACE FUNCTION get_offline_payment_customers(p_target_date
TIMESTAMP)
RETURN SYS_REFCURSOR IS
  v_cursor SYS_REFCURSOR;
BEGIN
  OPEN v_cursor FOR
  SELECT DISTINCT b.customer_id
  FROM SR_11667743.bill b
  WHERE b.bill_date = p_target_date
  AND b.sale_type = 'OFFLINE'

```

```
        AND b.mode_of_payment = 'CASH';

        RETURN v_cursor;
END get_offline_payment_customers;
/
```

Execution Code:

```
DECLARE
v_target_date TIMESTAMP;
v_customer_id RAW(16);
v_result_cursor SYS_REFCURSOR;
BEGIN
v_target_date := TO_TIMESTAMP('2023-10-24 09:55:00', 'YYYY-MM-DD HH24:MI:SS');
v_result_cursor := get_offline_payment_customers(v_target_date);

LOOP
    FETCH v_result_cursor INTO v_customer_id;
    EXIT WHEN v_result_cursor%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE('Customer ID: ' || v_customer_id);
END LOOP;

CLOSE v_result_cursor;
END;
/
```

Code Execution in Cloud:

Screenshot-1:

The screenshot shows the Oracle Database Actions interface with a SQL worksheet. The code is as follows:

```
1 -- 4. write a function to return the list of customerID's, who did payment using CASH in autozone on particular date.
2 -- Function:
3
4 CREATE OR REPLACE FUNCTION get_offline_payment_customers(p_target_date TIMESTAMP)
5   RETURN SYS_REFCURSOR IS
6   v_cursor SYS_REFCURSOR;
7
8   BEGIN
9     OPEN v_cursor FOR
10       SELECT DISTINCT b.customer_id
11         FROM SR_1166743.bill b
12        WHERE b.bill_date = p_target_date
13          AND b.sale_type = 'OFFLINE'
14          AND b.mode_of_payment = 'CASH';
15
16     RETURN v_cursor;
17   END get_offline_payment_customers;
18
19 -- Execution:
20
21 DECLARE
22   v_target_date TIMESTAMP;
23   v_customer_id RAW(16);
24   v_result_cursor SYS_REFCURSOR;
25
26 BEGIN
27   v_target_date := TO_TIMESTAMP('2023-10-24 09:55:00', 'YYYY-MM-DD HH24:MI:SS');
28   v_result_cursor := get_offline_payment_customers(v_target_date);
29
30   LOOP
31     FETCH v_result_cursor INTO v_customer_id;
32     EXIT WHEN v_result_cursor NOTFOUND;
33     DBMS_OUTPUT.PUT_LINE('Customer ID: ' || v_customer_id);
34   END LOOP;
35
36   CLOSE v_result_cursor;
37 END;
```

Execution results:

```
Function GET_OFFLINE_PAYMENT_CUSTOMERS compiled
Elapsed: 00:00:00.008
```

Screenshot-2:

The screenshot shows the Oracle Database Actions interface with a SQL worksheet. The code is identical to Screenshot-1, but the execution results show the output of the function.

```
17 /
18
19 -- Execution:
20
21 DECLARE
22   v_target_date TIMESTAMP;
23   v_customer_id RAW(16);
24   v_result_cursor SYS_REFCURSOR;
25
26 BEGIN
27   v_target_date := TO_TIMESTAMP('2023-10-24 09:55:00', 'YYYY-MM-DD HH24:MI:SS');
28   v_result_cursor := get_offline_payment_customers(v_target_date);
29
30   LOOP
31     FETCH v_result_cursor INTO v_customer_id;
32     EXIT WHEN v_result_cursor NOTFOUND;
33     DBMS_OUTPUT.PUT_LINE('Customer ID: ' || v_customer_id);
34   END LOOP;
35
36   CLOSE v_result_cursor;
37 END;
```

Execution results:

```
Customer ID: 0002
PL/SQL procedure successfully completed.
Elapsed: 00:00:00.017
```

Explanation: this function, 'get_offline_payment_customers,' does the job of identifying and listing customer IDs who paid with cash for their purchases at AutoZone on a specific date, which you can specify. It employs a cursor to store the results and a SQL query that picks out unique customer IDs from the 'bill' table where the purchase date matches your chosen date, the purchase was made 'OFFLINE,' and the payment method used was 'CASH.' When executed, it iterates through the results and prints out these customer IDs one by one. In essence, it helps to find and display the customers who paid in cash for their purchases at AutoZone on a particular date.

5. Write the function, which returns the inventory_product Id's whose automobile year before 2000.

PL/SQL Code:

```
CREATE OR REPLACE FUNCTION get_old_inventory_products RETURN SYS_REFCURSOR IS
    v_cursor SYS_REFCURSOR;
BEGIN
    OPEN v_cursor FOR
        SELECT ip.id
        FROM inventory_product ip
        JOIN automobile a ON ip.automobile_id = a.id
        WHERE a.year<2000;
    RETURN v_cursor;
END get_old_inventory_products;
/
```

Code Execution:

```
DECLARE
    old_products_cursor SYS_REFCURSOR;
    product_id RAW(16);
BEGIN
    old_products_cursor := get_old_inventory_products;

    LOOP
        FETCH old_products_cursor INTO product_id;
        EXIT WHEN old_products_cursor%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE('Deleting product with ID: ' || product_id);

        DELETE FROM admin.inventory_product WHERE id = product_id;
    END LOOP;
    CLOSE old_products_cursor;
END;
/
```

Code Execution in Cloud:

Screenshot-1:

The screenshot shows the Oracle Database Actions interface. In the Navigator pane, under the 'ADMIN' schema, several tables are listed: AUTOMOBILE, AUTOMOTIVE_RETAILER, INVENTORY_PRODUCT, INVENTORY_PRODUCT_SUPPLIER, and SUPPLIER. The 'Tables' section is expanded, showing the structure of the INVENTORY_PRODUCT table. The main workspace displays the following PL/SQL code:

```
1 CREATE OR REPLACE FUNCTION get_old_inventory_products RETURN SYS_REFCURSOR IS
2   v_cursor SYS_REFCURSOR;
3 BEGIN
4   OPEN v_cursor FOR
5   SELECT ip_id
6   FROM admin.inventory_product ip
7   JOIN admin.automobile a ON ip.automobile_id = a.id
8   WHERE a.year <= 2000;
9
10  RETURN v_cursor;
11 END get_old_inventory_products;
12 /
```

The 'Script Output' tab is selected, showing the output of the compilation:

```
Function GET_OLD_INVENTORY_PRODUCTS compiled
Elapsed: 00:00:00.009
```

The status bar at the bottom indicates: 0 0 0 0 | 12:53:20 AM - Code execution finished.

Screenshot-2:

The screenshot shows the Oracle Database Actions interface with the session identifier S11664475. The Navigator pane lists tables: CUSTOMER, CUSTOMER_INSURANCE, INSURANCE, and PAYMENT_PLAN. The main workspace contains the following PL/SQL code:

```
1 DECLARE
2   old_products_cursor SYS_REFCURSOR;
3   product_id RAW(16);
4 BEGIN
5   old_products_cursor := get_old_inventory_products;
6
7   LOOP
8     FETCH old_products_cursor INTO product_id;
9     EXIT WHEN old_products_cursor%NOTFOUND;
10    DBMS_OUTPUT.PUT_LINE('Deleting product with ID: ' || product_id);
11
12    DELETE FROM admin.inventory_product WHERE id = product_id;
13  END LOOP;
14  CLOSE old_products_cursor;
15 END;
16 /
```

The 'Script Output' tab shows the execution results:

```
Deleting product with ID: 0722
Deleting product with ID: 0723
Deleting product with ID: 0724
Deleting product with ID: 0725

PL/SQL procedure successfully completed.
Elapsed: 00:00:00.038
```

The status bar at the bottom indicates: 0 0 0 0 | 1:57:41 AM - SQL executed by S11664475.

Explanation: This function, 'get_old_inventory_products,' is designed to identify and list the IDs of inventory products associated with automobiles manufactured before the year 2000. It utilizes a cursor to hold these product IDs and a SQL query that fetches IDs from the 'inventory_product' table, where the related automobiles have a manufacturing year earlier than 2000. When executed, the code loops through these IDs, prints a message for each, and subsequently deletes the corresponding products from the inventory. It helps us find and remove products linked to older automobiles from the inventory.

6. write a stored function calculates the total sales made to a specific customer based on their customer ID.

PL/SQL Code:

```
CREATE OR REPLACE FUNCTION calculate_sales_on_customer(p_customer_id RAW) RETURN NUMBER IS
v_total_sales NUMBER;
BEGIN
    SELECT SUM(bill_total) INTO v_total_sales
    FROM (
        SELECT b.id, SUM(ip.price * bip.quantity) AS bill_total
        FROM SR_11667743.bill b
        JOIN SR_11667743.bill_inventory_product bip ON b.id = bip.bill_id
        JOIN admin.inventory_product ip ON bip.inventory_product_id = ip.id
        WHERE b.customer_id = p_customer_id
        GROUP BY b.id
    );
    RETURN v_total_sales;
END;
/
```

Execution Code:

```
DECLARE
    customer_id RAW(16);
    total_sales NUMBER;
BEGIN
    customer_id := '901';
    total_sales := calculate_sales_on_customer(customer_id);
    DBMS_OUTPUT.PUT_LINE('Total Sales for Customer ' || customer_id || ':' || total_sales);
END;
/
```

Code Execution in Cloud:

Screenshot-1:

The screenshot shows the Oracle Database Actions SQL Worksheet interface. The code in the worksheet is:

```
1 CREATE OR REPLACE FUNCTION calculate_sales_on_customer(p_customer_id RAW) RETURN NUMBER IS
2   v_total_sales NUMBER;
3   BEGIN
4     SELECT SUM(bill_total) INTO v_total_sales
5     FROM (
6       SELECT b.id, SUM(ip.price * bip.quantity) AS bill_total
7         FROM SR_11667743.bill b
8        JOIN SR_11667743.bill_inventory_product bip ON b.id = bip.bill_id
9        JOIN admin.inventory_product ip ON bip.inventory_product_id = ip.id
10       WHERE b.customer_id = p_customer_id
11      GROUP BY b.id
12    );
13   RETURN v_total_sales;
14 END;
15 /
16
```

The "Script Output" tab is selected, showing the message "Function CALCULATE_SALES_ON_CUSTOMER compiled". The elapsed time is 00:00:00.008.

Screenshot-2:

The screenshot shows the Oracle Database Actions SQL Worksheet interface. The code in the worksheet is:

```
1 DECLARE
2   customer_id RAW(16);
3   total_sales NUMBER;
4   BEGIN
5     customer_id := '0901';
6     total_sales := calculate_sales_on_customer(customer_id);
7     DBMS_OUTPUT.PUT_LINE('Total Sales for Customer ' || customer_id || ':' || total_sales);
8   END;
9 /
10
```

The "Script Output" tab is selected, showing the output "Total Sales for Customer 0901: 5650" and the message "PL/SQL procedure successfully completed.". The elapsed time is 00:00:00.023.

Explanation: This function, 'calculate_sales_on_customer,' serves the purpose of determining the total sales amount for a specific customer, whose ID is provided as 'p_customer_id.' It does so by summing up the bill totals, which are calculated from the prices and quantities of inventory products listed on each bill associated with the customer. When executed, it takes a customer's ID (in this case, '901'), calculates their total sales, and prints out the result, helping you find the total sales amount for a particular customer.

7 write a stored function, identifies employees who have not been assigned to any job.

PL/SQL Code:

```
CREATE OR REPLACE FUNCTION get_employees_without_jobs RETURN SYS_REFCURSOR AS
    result_cursor SYS_REFCURSOR;
BEGIN
    OPEN result_cursor FOR
        SELECT e.id, e.first_name, e.last_name
        FROM v_11642773.employee e
        WHERE NOT EXISTS (
            SELECT *
            FROM v_11642773.job_employee je
            WHERE je.employee_id = e.id
        );
    RETURN result_cursor;
END;
/
```

Execution Code:

```
DECLARE
    new_employees_cursor SYS_REFCURSOR;
    emp_id RAW(16);
    first_name VARCHAR2(100);
    last_name VARCHAR2(100);
BEGIN
    new_employees_cursor := get_employees_without_jobs;
    LOOP
        FETCH new_employees_cursor INTO emp_id, first_name, last_name;
        EXIT WHEN new_employees_cursor%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE('Employee ID: ' || emp_id);
        DBMS_OUTPUT.PUT_LINE('First Name: ' || first_name);
        DBMS_OUTPUT.PUT_LINE('Last Name: ' || last_name);
        DBMS_OUTPUT.PUT_LINE('-----');
    END LOOP;
```

```
CLOSE new_employees_cursor;
END;
/
```

Code Execution in Cloud:

Screenshot-1:

The screenshot shows the Oracle Database Actions SQL Worksheet interface. The top navigation bar includes 'Database Actions | SQL', 'Search' (with a magnifying glass icon), and a consumer group dropdown set to 'LOW'. The main workspace displays the following PL/SQL code:

```
2  -- Functions:
3
4  CREATE OR REPLACE FUNCTION get_employees_without_jobs RETURN SYS_REFCURSOR AS
5      result_cursor SYS_REFCURSOR;
6  BEGIN
7      OPEN result_cursor FOR
8          SELECT e.id, e.first_name, e.last_name
9          FROM v_11642773.employee e
10         WHERE NOT EXISTS (
11             SELECT *
12             FROM v_11642773.job_employee je
13            WHERE je.employee_id = e.id
14        );
15
16      RETURN result_cursor;
17  END;
18
19
20  -- Execution:
21
22  DECLARE
23      new_employees_cursor SYS_REFCURSOR;
24      emp_id RAW(16);
25      first_name VARCHAR2(100);
```

The 'Query Result' tab is selected, showing the output of the function compilation and execution:

```
Function GET_EMPLOYEES_WITHOUT_JOBS compiled
Elapsed: 00:00:00.024
```

The bottom status bar indicates: '0 0 0 0 0 1 129:38 AM - SQL executed by SR_11667743'.

Screenshot-2:

The screenshot shows the Oracle Database Actions SQL Worksheet interface. The code in the worksheet is a PL/SQL procedure named GET_EMPLOYEES_WITHOUT_JOBS. It declares a cursor for employees without jobs, loops through the results, and prints employee details (Employee ID, First Name, Last Name) to the DBMS_OUTPUT. The execution output shows one record: Employee ID: 8615, First Name: J, Last Name: Doe. The procedure is successfully compiled and executed.

```

18  /
19
20  -- Execution:
21
22  DECLARE
23      new_employees_cursor SYS_REFCURSOR;
24      emp_id RAW(16);
25      first_name VARCHAR2(100);
26      last_name VARCHAR2(100);
27
28  BEGIN
29      new_employees_cursor := get_employees_without_jobs;
30      LOOP
31          FETCH new_employees_cursor INTO emp_id, first_name, last_name;
32          EXIT WHEN new_employees_cursor%NOTFOUND;
33          DBMS_OUTPUT.PUT_LINE('Employee ID: ' || emp_id);
34          DBMS_OUTPUT.PUT_LINE('First Name: ' || first_name);
35          DBMS_OUTPUT.PUT_LINE('Last Name: ' || last_name);
36      END LOOP;
37      CLOSE new_employees_cursor;
38  END;
39

```

Function GET_EMPLOYEES_WITHOUT_JOBS compiled
Elapsed: 00:00:00.024

Employee ID: 8615
First Name: J
Last Name: Doe

PL/SQL procedure successfully completed.
Elapsed: 00:00:00.816

Explanation: This function, 'get_employees_without_jobs,' is designed to identify and list employees who currently don't have any job assignments. It uses a cursor to store the results and a SQL query that selects employee IDs, first names, and last names from the 'employee' table for those employees where no job assignments exist in the 'job_employee' table. When executed, the code loops through the results, prints out the employee details, and helps you easily find and display employees who are currently without job assignments within your organization.

Triggers:

1. Write a trigger to calculate the job cost, using total number of parts used and respective cost for them when a new job is created.

PL/SQL Code:

```

CREATE OR REPLACE TRIGGER calculate_job_cost
BEFORE INSERT ON SR_11667743.part_service
FOR EACH ROW
DECLARE
    job_cost NUMBER;
BEGIN

```

```

SELECT SUM(ps.quantity * ip.price) into job_cost
FROM SR_11667743.part_service ps
JOIN admin.inventory_product ip ON ps.inventory_product_id = ip.id
WHERE ps.job_id = :NEW.job_id;
dbms_output.put_line('The total cost of the job is ' || job_cost);
END;
/

```

Execution Code:

```

INSERT INTO SR_11667743.part_service (id, name, quantity, job_id, inventory_product_id,
type)
VALUES ('1127', 'Oil Change', 2, '1001', '0701', 'SERVICE');

```

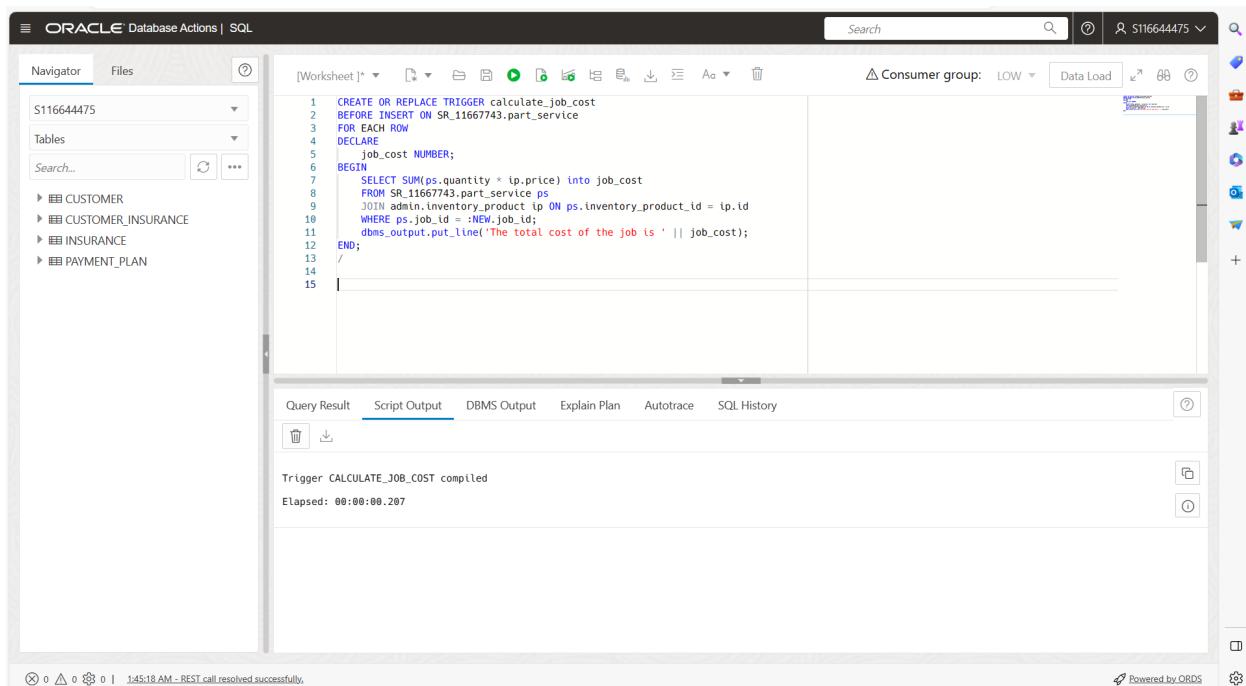
```

INSERT INTO SR_11667743.part_service (id, name, quantity, job_id, inventory_product_id,
type)
VALUES ('1128', 'Brake Replacement', 4, '1002', '702', 'PART');

```

Code Execution in Cloud:

Screenshot-1:



Screenshot-2:

The screenshot shows the Oracle Database Actions SQL Worksheet interface. In the top right, there's a search bar and a consumer group dropdown set to 'LOW'. Below it is a toolbar with icons for search, refresh, and data load. The main area has tabs for 'Worksheet' (selected), 'Script Output', 'DBMS Output', 'Explain Plan', 'Autotrace', and 'SQL History'. The 'Worksheet' tab contains the following SQL code:

```
1 INSERT INTO SR_11667743.part_service (id, name, quantity, job_id, inventory_product_id, type)
2 VALUES ('1127', 'Oil Change', 2, '1001', '0701', 'SERVICE');
3
4 INSERT INTO SR_11667743.part_service (id, name, quantity, job_id, inventory_product_id, type)
5 VALUES ('1128', 'Brake Replacement', 4, '1002', '702', 'PART');
```

Below the code, the 'Query Result' tab is active, showing the output of the first insert:

The total cost of the job is 2500
1 row inserted.
Elapsed: 00:00:00.013

Below that, the output of the second insert is shown:

The total cost of the job is 7500
1 row inserted.
Elapsed: 00:00:00.003

At the bottom left, there are status icons for rows, columns, and rows selected. At the bottom right, it says 'Powered by ORDS'.

Explanation: The trigger calculate_job_cost is designed to automatically calculate and output the total cost of a job. This trigger enhances the system by providing a real-time cost calculation for each job, making it easier to manage and track job expenses. It simplifies cost tracking for services and parts utilized in each job.

2. Write a trigger, to update the product quantity in the inventory_product table upon completion of job.

PL/SQL Code:

```
CREATE OR REPLACE TRIGGER update_inventory_product_quantity_on_job_completion
AFTER UPDATE ON V_11642773.job
FOR EACH ROW
declare quantity number;
BEGIN
    -- Update the inventory product quantity on job completion
    UPDATE admin.inventory_product ip
    SET ip.quantity = ip.quantity -
        (SELECT SUM(ps.quantity)
        FROM part_service ps
        WHERE ps.job_id = :NEW.id)
```

```

)
WHERE EXISTS (
    SELECT 1
    FROM part_service ps
    WHERE ps.job_id = :NEW.id
    AND ps.inventory_product_id = ip.id
);
select sum(quantity) into quantity from admin.inventory_product;
dbms_output.put_line('The total number parts remain in inventory is '||quantity);
END;
/

```

Execution Code:

```
update V_11642773.job set Job_status='Completed' where id='1001';
```

Code Execution in Cloud:

Screenshot-1:

The screenshot shows the Oracle Database Actions SQL Worksheet interface. The left sidebar displays the Navigator and Tables sections, with 'SR_11667743' selected. The main workspace contains the following PL/SQL code:

```

1 CREATE OR REPLACE TRIGGER update_inventory_product_quantity_on_job_completion
2 AFTER UPDATE ON v_11642773.job
3 FOR EACH ROW
4 BEGIN
5     declare quantity number;
6     -- Update the inventory product quantity on job completion
7     UPDATE admin.inventory_product ip
8     SET ip.quantity = ip.quantity - (
9         SELECT SUM(ps.quantity)
10        FROM part_service ps
11       WHERE ps.job_id = :NEW.id
12     );
13     WHERE EXISTS (
14         SELECT 1
15         FROM part_service ps
16        WHERE ps.job_id = :NEW.id
17        AND ps.inventory_product_id = ip.id
18     );
19 select sum(quantity) into quantity from admin.inventory_product;
20 dbms_output.put_line('The total number parts remain in inventory is '||quantity);
21 END;
22 /
23
24

```

The bottom pane shows the 'Script Output' tab with the message: 'Trigger UPDATE_INVENTORY_PRODUCT_QUANTITY_ON_JOB_COMPLETION compiled'. The 'Elapsed' time is listed as '00:00:00.026'.

Screenshot-2:

The screenshot shows the Oracle Database Actions SQL Worksheet interface. The code in the worksheet is a PL/SQL trigger named 'update_inventory_product_quantity_on_job_completion'. It uses a FOR EACH ROW loop to calculate the total quantity of parts used in a completed job by summing quantities from the 'part_service' table where job_id matches the new job_id. It then updates the 'admin.inventory_product' table by subtracting this quantity from the current quantity. Finally, it uses DBMS_OUTPUT.PUT_LINE to display the total number of parts remaining in inventory.

```

1  --PL/SQL BLOCK V_11642773.job
2
3  FOR EACH ROW
4      declare quantity number;
5      BEGIN
6          -- Update the inventory product quantity on job completion
7          UPDATE admin.inventory_product ip
8              SET ip.quantity = ip.quantity - (
9                  SELECT SUM(ps.quantity)
10                 FROM part_service ps
11                   WHERE ps.job_id = :NEW.id
12               )
13          WHERE EXISTS (
14              SELECT 1
15                 FROM part_service ps
16                   WHERE ps.job_id = :NEW.id
17                     AND ps.inventory_product_id = ip.id
18           );
19          select sum(quantity) into quantity from admin.inventory_product;
20          dbms_output.put_line('The total number parts remain in inventory is '||quantity);
21      END;
22  /
23
24
25  update v_11642773.job set Job_Status='Completed' where id='1001';
26

```

The 'Query Result' tab shows the execution details:

- Trigger 'UPDATE_INVENTORY_PRODUCT_QUANTITY_ON_JOB_COMPLETION' compiled
- Elapsed: 00:00:00.026
- The total number parts remain in inventory is 143
- 1 row updated.
- Elapsed: 00:00:00.015

Explanation: The 'update_inventory_product_quantity_on_job_completion' trigger is designed to automatically update inventory product quantities when a job is marked as 'Completed' in the 'V_11642773.job' table. It calculates the total quantity of parts used in the completed job by summing the quantities from the 'part_service' table. It then subtracts this calculated quantity from the corresponding inventory product in the 'admin.inventory_product' table, ensuring accurate inventory management. After the update, the trigger displays the total quantity of parts remaining in the inventory using 'DBMS_OUTPUT.PUT_LINE.' This trigger helps maintain inventory accuracy and streamline parts management in the automotive context.

3. Write a trigger message with employee name reminding his next target after completing 10 bills and creating 11th bill.

PL/SQL Code:

```

CREATE OR REPLACE TRIGGER check_successful_bills_trigger
BEFORE INSERT ON sr_11667743.bill
FOR EACH ROW
DECLARE
    first_name varchar(100);
    last_name varchar(100);
    bill_counts number;

```

```

BEGIN
  SELECT e.first_name , e.last_name, b.bill_count into first_name, last_name, bill_counts
  FROM Employee e
  JOIN (
    SELECT employee_id, COUNT(*) as bill_count
    FROM sr_11667743.bill
    GROUP BY employee_id
  ) b ON e.id = b.employee_id where b.employee_id = :NEW.employee_id;

  if bill_counts = 10 then
    DBMS_OUTPUT.PUT_LINE('Congratulations ' || first_name || ' ' || last_name || ' on
successfully creating 10 bills.');
    DBMS_OUTPUT.PUT_LINE('Your next milestone to finish 100 bills will began on successful
creation of 11th bill which is inprogress now');
  END if;
END;
/

```

Execution Code:

```

INSERT INTO sr_11667743.bill (id, bill_date, mode_of_payment, insurance_id, customer_id,
job_id, employee_id, sale_type, payment_plan_id)
VALUES ('1224', TIMESTAMP '2023-10-31 14:30:00', 'CARD', '301', '901', '1001', '601', 'ONLINE',
'101');

```

Code Execution in Cloud:

Screenshot-1:

The screenshot shows the Oracle Database Actions SQL Worksheet interface. The code in the worksheet is a PL/SQL trigger named CHECK_SUCCESSFUL_BILLS_TRIGGER. It triggers a message to remind the employee of their next target after completing 10 bills and creating 11th bill. The trigger uses DBMS_OUTPUT.PUT_LINE to display messages. The code includes a SELECT statement to get employee details and a COUNT(*) operation to check bill counts. The trigger is successfully compiled.

```
1 -- trigger a message with employee name reminding his next target after completing 10 bills and creating 11th bill
2 CREATE OR REPLACE TRIGGER check_successful_bills_trigger
3 BEFORE INSERT ON sr_11667743.bill
4 FOR EACH ROW
5 DECLARE
6   first_name varchar(100);
7   last_name varchar(100);
8   bill_counts number;
9 BEGIN
10   SELECT e.first_name , e.last_name, b.bill_count into first_name, last_name, bill_counts
11   FROM Employee e
12   JOIN
13     (SELECT employee_id, COUNT(*) as bill_count
14      FROM sr_11667743.bill
15     GROUP BY employee_id
16    ) b ON e.id = b.employee_id where b.employee_id = :NEW.employee_id;
17
18   if bill_counts = 10 then
19     DBMS_OUTPUT.PUT_LINE('Congratulations ' || first_name || ' ' || last_name || ' on successfully creating 10 bills.');
20     DBMS_OUTPUT.PUT_LINE('Your next milestone to finish 100 bills will begin on successful creation of 11th bill which is inprogress now');
21   END if;
22
23 END;
24 /
25
26
27
28
29
30
31
```

Trigger CHECK_SUCCESSFUL_BILLS_TRIGGER compiled
Elapsed: 00:00:00.019

Screenshot-2:

The screenshot shows the Oracle Database Actions SQL Worksheet interface. The code in the worksheet is a PL/SQL trigger named CHECK_SUCCESSFUL_BILLS_TRIGGER. It triggers a message to remind the employee of their next target after completing 10 bills and creating 11th bill. The trigger uses DBMS_OUTPUT.PUT_LINE to display messages. The code includes a SELECT statement to get employee details and a COUNT(*) operation to check bill counts. The trigger is successfully compiled. Following the compilation, an INSERT statement is executed into the sr_11667743.bill table, inserting a single row with specific values for each column.

```
10   SELECT e.first_name , e.last_name, b.bill_count into first_name, last_name, bill_counts
11   FROM Employee e
12   JOIN
13     (SELECT employee_id, COUNT(*) as bill_count
14      FROM sr_11667743.bill
15     GROUP BY employee_id
16    ) b ON e.id = b.employee_id where b.employee_id = :NEW.employee_id;
17
18   if bill_counts = 10 then
19     DBMS_OUTPUT.PUT_LINE('Congratulations ' || first_name || ' ' || last_name || ' on successfully creating 10 bills.');
20     DBMS_OUTPUT.PUT_LINE('Your next milestone to finish 100 bills will begin on successful creation of 11th bill which is inprogress now');
21   END if;
22
23 END;
24 /
25
26
27
28
29
30
31
```

```
INSERT INTO sr_11667743.bill (id, bill_date, mode_of_payment, insurance_id, customer_id, job_id, employee_id, sale_type, payment_plan_id)
VALUES ('1215', TIMESTAMP '2023-10-31 14:30:00', 'CARD', '301', '1001', '601', 'ONLINE', '101');
```

Trigger CHECK_SUCCESSFUL_BILLS_TRIGGER compiled
Elapsed: 00:00:00.019
insurance_id 301 is valid
1 row inserted.
Elapsed: 00:00:00.011

Screenshot-3:

The screenshot shows the Oracle Database Actions | SQL interface. The workspace contains a worksheet with the following PL/SQL code:

```

9  BEGIN
10   SELECT e.first_name , e.last_name, b.bill_count into first_name, last_name, bill_counts
11   FROM Employee e
12   JOIN (
13      SELECT employee_id, COUNT(*) as bill_count
14      FROM sr_11667743.bill
15      GROUP BY employee_id
16    ) b ON e.id = b.employee_id where b.employee_id = :NEW.employee_id;
17
18   if bill_counts = 10 then
19      DBMS_OUTPUT.PUT_LINE('Congratulations ' || first_name || ' ' || last_name || ' on successfully creating 10 bills.');
20      DBMS_OUTPUT.PUT_LINE('Your next milestone to finish 100 bills will begin on successful creation of 11th bill which is inprogress now');
21   END if;
22
23   END;
24 /
25
26
27   INSERT INTO sr_11667743.bill (id, bill_date, mode_of_payment, insurance_id, customer_id, job_id, employee_id, sale_type, payment_plan_id)
28   VALUES ('1224', TIMESTAMP '2023-10-31 14:30:00', 'CARD', '301', '991', '1001', '601', 'ONLINE', '101');

```

The code inserts a new bill record into the 'sr_11667743.bill' table. The output window shows the following message:

Congratulations John Doe on successfully creating 10 bills.
Your next milestone to finish 100 bills will begin on successful creation of
11th bill which is inprogress now
insurance_id #301 is valid

1 row inserted.

Elapsed: 00:00:00.004

Explanation: The 'check_successful_bills_trigger' trigger in the 'sr_11667743' schema monitors the creation of bills by employees. When an employee inserts a new bill, the trigger checks if they have successfully created 10 bills. If they have, it congratulates them and mentions that the next milestone is to complete 100 bills, which will start upon the successful creation of the 11th bill (currently in progress). This trigger provides recognition and motivation for employees based on their bill creation milestones.

4. Write a trigger to log customer details because it ensures that changes to customer data are carefully monitored and logged when in-progress jobs are associated with the customer to communicate this vital information.

PL/SQL Code:

```

CREATE OR REPLACE TRIGGER LogCustomerChanges
BEFORE UPDATE ON S116644475.customer
FOR EACH ROW
DECLARE
  v_job_id VARCHAR2(4000);
  employee_id RAW(15);
  first_name VARCHAR2(100);
  last_name VARCHAR2(100);
BEGIN

```

```

-- Query job information
SELECT j.id AS job_id, emp.ID, emp.FIRST_NAME, emp.LAST_NAME
INTO v_job_id, employee_id, first_name, last_name
FROM V_11642773.job j
JOIN V_11642773.JOB_EMPLOYEE je ON je.JOB_ID = j.ID
JOIN V_11642773.EMPLOYEE emp ON je.EMPLOYEE_ID = emp.ID
WHERE j.customer_id = :OLD.id AND j.job_status = 'inprogress';

IF v_job_id IS NOT NULL THEN
    -- Display a message to the console
    DBMS_OUTPUT.PUT_LINE('Change in customer data for Customer ID: ' || :OLD.id ||
        ' is not allowed because there are in-progress jobs associated with this customer.');

    -- You can add more details to the message if needed
    DBMS_OUTPUT.PUT_LINE('Related Employee: ' || first_name || ' ' || last_name || '
        (Employee ID: ' || employee_id || ')');
    DBMS_OUTPUT.PUT_LINE('Related Job IDs: ' || v_job_id);
END IF;
END;
/

```

Execution Code:

```
UPDATE S116644475.customer SET first_name = 'jones' WHERE id = '0904';
```

Code Execution in Cloud:

Screenshot -1:

```

CREATE OR REPLACE TRIGGER LogCustomerChanges
BEFORE UPDATE ON S116644475.customer
FOR EACH ROW
DECLARE
  v_job_id VARCHAR2(4000);
  employee_id RAW(15);
  first_name VARCHAR(100);
  last_name VARCHAR2(100);
BEGIN
  -- Query job information
  SELECT j.id AS job_id, emp.id, emp.FIRST_NAME, emp.LAST_NAME
  INTO v_job_id, employee_id, first_name, last_name
  FROM V_11642773.job j
  JOIN V_11642773.JOB_EMPLOYEE je ON je.JOB_ID = j.ID
  JOIN V_11642773.EMPLOYEE emp ON je.EMPLOYEE_ID = emp.ID
  WHERE j.customer_id = :OLD.id AND j.job_status = 'inprogress';

  IF v_job_id IS NOT NULL THEN
    -- Display a message to the console
    DBMS_OUTPUT.PUT_LINE('Change in customer data for Customer ID: ' || :OLD.id ||
      ' is not allowed because there are in-progress jobs associated with this customer.');
  END IF;
END;
/

```

Trigger LOGCUSTOMERCHANGES compiled
Elapsed: 00:00:00.019

Screenshot-2:

```

-- Query job information
SELECT j.id AS job_id, emp.id, emp.FIRST_NAME, emp.LAST_NAME
INTO v_job_id, employee_id, first_name, last_name
FROM V_11642773.job j
JOIN V_11642773.JOB_EMPLOYEE je ON je.JOB_ID = j.ID
JOIN V_11642773.EMPLOYEE emp ON je.EMPLOYEE_ID = emp.ID
WHERE j.customer_id = :OLD.id AND j.job_status = 'inprogress';

IF v_job_id IS NOT NULL THEN
  -- Display a message to the console
  DBMS_OUTPUT.PUT_LINE('Change in customer data for Customer ID: ' || :OLD.id ||
    ' is not allowed because there are in-progress jobs associated with this customer.');

  -- You can add more details to the message if needed
  DBMS_OUTPUT.PUT_LINE('Related Employee: ' || first_name || ' ' || last_name || ' (Employee ID: ' || employee_id || ')');
  DBMS_OUTPUT.PUT_LINE('Related Job IDs: ' || v_job_id);
END IF;
/

```

UPDATE S116644475.customer SET first_name = 'jones' WHERE id = '0994';

1 row updated.
Elapsed: 00:00:00.006

Explanation: The LogCustomerChanges trigger is designed to monitor and log changes to customer data in the S116644475.customer table. This trigger executes before an UPDATE operation on the customer table and checks for the presence of in-progress jobs associated with the customer. If such jobs are found, the trigger displays a console message via

`DBMS_OUTPUT.PUT_LINE`, providing a notification that the change in customer data is not allowed due to the active job associations. The message includes pertinent details such as the customer's ID, the related employee associated with in-progress jobs, and the job IDs currently in progress.

By implementing this trigger, you can maintain data integrity and provide real-time notifications when updates to customer data are restricted due to ongoing job commitments, ensuring that data modifications are appropriately tracked and communicated to relevant stakeholders.

Package:

- 1. Write a package and its corresponding package body to retrieve information about customers and their automobiles, as well as to print the list of automobiles serviced by employees in an automotive retailer?**

PL/SQL Code:

Package:

```
CREATE OR REPLACE PACKAGE automotive_package AS
    -- Stored Function 1: Get the list of customers and their purchased automobiles
    FUNCTION get_customers_with_purchased_automobiles RETURN SYS_REFCURSOR;

    -- Stored Procedure 1: Print the list of automobiles serviced in an automotive retailer
    PROCEDURE get_employee_jobs;

END automotive_package;
```

Package Body:

```
CREATE OR REPLACE PACKAGE BODY automotive_package AS
    -- Stored Function 1: Get the list of customers and their purchased automobiles
    FUNCTION get_customers_with_purchased_automobiles RETURN SYS_REFCURSOR IS
        customers_cursor SYS_REFCURSOR;
    BEGIN
        OPEN customers_cursor FOR
            SELECT c.id, c.first_name, c.last_name, a.manufacturer, a.name, a.variant
            FROM S116644475.customer c
            JOIN V_11642773.job j ON c.id = j.customer_id
            JOIN automobile a ON j.automobile_id = a.id;

        RETURN customers_cursor;
    END get_customers_with_purchased_automobiles;

    -- Stored Procedure 1: procedure fetches data related to employees and their associated jobs
```

```

PROCEDURE get_employee_jobs IS
BEGIN
  DBMS_OUTPUT.PUT_LINE('Employee Jobs');
  DBMS_OUTPUT.PUT_LINE('-----');

  FOR emp_rec IN (
    SELECT e.first_name, e.last_name, j.description, a.name AS automobile_name
    FROM V_11642773.Employee e
    JOIN V_11642773.job_employee je ON je.employee_id = e.id
    JOIN V_11642773.job j ON j.id = je.job_id
    JOIN Automobile a ON j.automobile_id = a.id
  ) LOOP
    DBMS_OUTPUT.PUT_LINE('Employee: ' || emp_rec.first_name || ' ' || emp_rec.last_name
    || ', ' || 'Job: ' || emp_rec.description || ', ' || 'Automobile: ' || emp_rec.automobile_name);
  END LOOP;
END get_employee_jobs;

END automotive_package;

```

Execution code for function:

```

DECLARE
  customers_cursor SYS_REFCURSOR;
  customer_id varchar(100);
  first_name VARCHAR2(100);
  last_name VARCHAR2(100);
  manufacturer VARCHAR(100);
  name VARCHAR(100);
  variant VARCHAR(100);
  -- Adjust data type as needed
BEGIN
  customers_cursor := automotive_package.get_customers_with_purchased_automobiles;
  DBMS_OUTPUT.PUT_LINE('-----');
LOOP
  FETCH customers_cursor INTO customer_id, first_name, last_name, manufacturer, name,
variant;
  EXIT WHEN customers_cursor%NOTFOUND;

  -- Process and print the data
  DBMS_OUTPUT.PUT_LINE('Customer ID: ' || customer_id || ', Customer: ' || first_name || '
  || last_name || ', Automobile: ' || manufacturer || ' ' || name || ' ' || variant);
END LOOP;
  DBMS_OUTPUT.PUT_LINE('-----');
CLOSE customers_cursor;

```

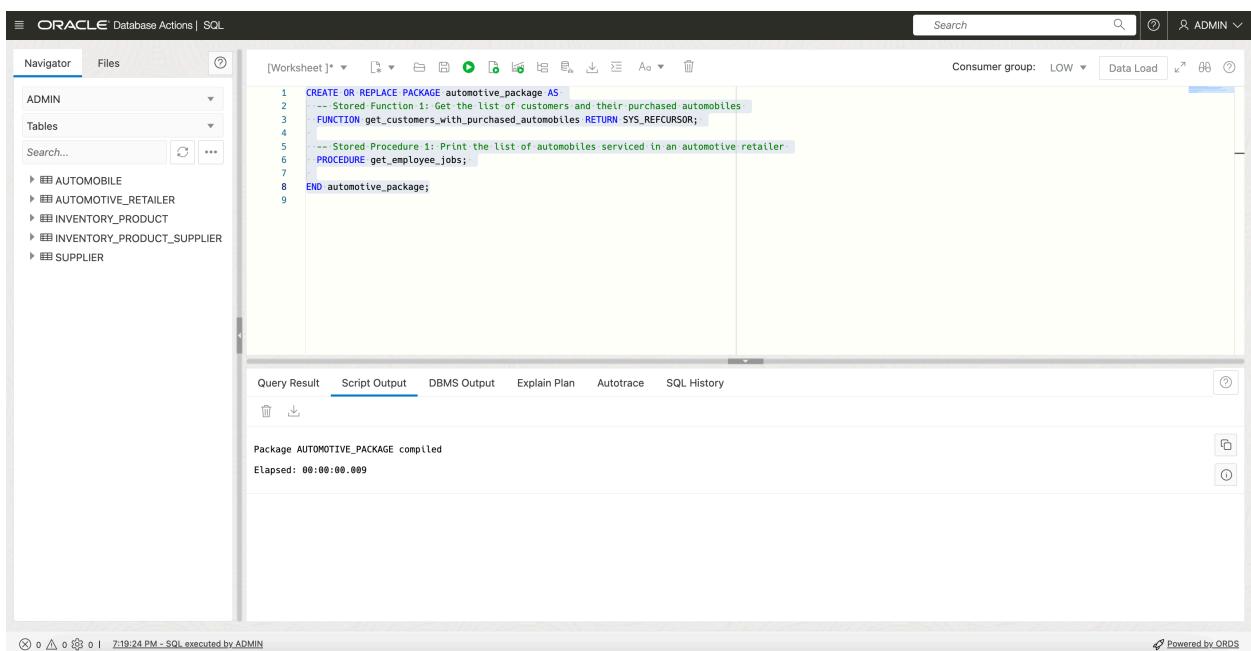
```
END;
```

Execution Code for Procedure:

```
BEGIN  
    automotive_package.get_employee_jobs;  
END;
```

Code Execution in Cloud:

Screenshot-1:



Screenshot-2:

The screenshot shows the Oracle Database Actions interface with the following details:

- Navigator** and **Files** tabs are selected.
- Consumer group:** LOW
- Data Load** button.
- Search** bar.
- ADMIN** user is logged in.
- Tables** section shows **AUTOMOBILE**, **AUTOMOTIVE_RETAILER**, **INVENTORY_PRODUCT**, **INVENTORY_PRODUCT_SUPPLIER**, and **SUPPLIER**.
- Search...** input field.
- Script Content:**

```
1 CREATE OR REPLACE PACKAGE BODY automotive_package AS
2   -- Stored Function 1: Get the list of customers and their purchased automobiles
3   FUNCTION get_customers_with_purchased_automobiles RETURN SYS_REFCURSOR IS
4     customers_cursor SYS_REFCURSOR;
5   BEGIN
6     OPEN customers_cursor FOR
7       SELECT c.id, c.first_name, c.last_name, a.manufacturer, a.name, a.variant
8         FROM S116644475.customer c
9        JOIN V_11642773.job j ON c.id = j.customer_id
10       JOIN automobile a ON j.automobile_id = a.id;
11
12     RETURN customers_cursor;
13   END get_customers_with_purchased_automobiles;
14
15   -- Stored Procedure 1: procedure fetches data related to employees and their associated jobs
16   PROCEDURE get_employee_jobs IS
17   BEGIN
18     DBMS_OUTPUT.PUT_LINE('Employee Jobs');
19     DBMS_OUTPUT.PUT_LINE('-----');
20
21
22
23
24
25
26
27
28
```
- Query Result** tab is selected.
- Script Output** tab is also visible.
- Elapsed: 00:00:00.009** for the package compilation.
- Elapsed: 00:00:00.008** for the package body compilation.
- Self Service** button at the bottom.

Screenshot-3:

The screenshot shows the Oracle Database Actions interface with the SQL tab selected. The code in the worksheet is as follows:

```
1  DECLARE
2      customers_cursor SYS_REFCURSOR;
3      customer_id varchar(100);
4      first_name VARCHAR2(100);
5      last_name VARCHAR2(100);
6      manufacturer VARCHAR2(100);
7      name VARCHAR2(100);
8      variant VARCHAR2(100);
9      -- Adjust data type as needed
10 BEGIN
11     OPEN customers_cursor : automotive_package.get_customers_with_purchased_automobiles;
12     DBMS_OUTPUT.PUT_LINE('Customer ID: ' || customer_id || ', Customer: ' || first_name || ' ' || last_name || ', Automobile: ' || manufacturer || ' ' || name || ' ' || variant);
13     LOOP
14         FETCH customers_cursor INTO customer_id, first_name, last_name, manufacturer, name, variant;
15         EXIT WHEN customers_cursor%NOTFOUND;
16     END LOOP;
17     -- Process and print the data
18     DBMS_OUTPUT.PUT_LINE('Customer ID: ' || customer_id || ', Customer: ' || first_name || ' ' || last_name || ', Automobile: ' || manufacturer || ' ' || name || ' ' || variant);
19     END LOOP;
20     DBMS_OUTPUT.PUT_LINE('-----');
21     CLOSE customers_cursor;
22 END;
23
```

The Query Result pane displays the output of the script, listing ten customer records with their details and the automobile they purchased. The results are as follows:

Customer ID	Customer	Automobile
0006	Sarah Wilson	Volkswagen Passat SE
0001	John Doe	BMW 3 Series 328i
0002	Jane Smith	Mercedes-Benz C-Class C300
0009	James White	Audi A4 Premium
0001	Mia Martinez	Lexus ES 350
0002	Jane Smith	Mercedes-Benz C-Class C300
0009	James White	Audi A4 Premium
0001	Mia Martinez	Lexus ES 350
0002	Jane Smith	Mercedes-Benz C-Class C300
0009	James White	Audi A4 Premium
0010	Mia Martinez	Lexus ES 350

Screenshot-4:

The screenshot shows the Oracle Database Actions SQL Worksheet interface. The code in the worksheet is a PL/SQL procedure that retrieves customers with purchased automobiles and prints their details. The output shows records for various customers and their purchased vehicles.

```

1  DECLARE
2      customers_cursor SYS_REFCURSOR;
3      customer_id varchar(100);
4      first_name VARCHAR2(100);
5      last_name VARCHAR2(100);
6      manufacturer VARCHAR(100);
7      name VARCHAR(100);
8      variant VARCHAR(100);
9      -- Adjust data type as needed
10 BEGIN
11     customers_cursor := automotive_package.get_customers_with_purchased_automobiles;
12     DBMS_OUTPUT.PUT_LINE('-----');
13     LOOP

```

Customer ID: 0902, Customer: Jane Smith, Automobile: Toyota Camry LE
Customer ID: 0901, Customer: John Doe, Automobile: Toyota Camry LE
Customer ID: 0903, Customer: Michael Johnson, Automobile: Ford Mustang GT
Customer ID: 0904, Customer: Emily Brown, Automobile: Chevrolet Malibu LT
Customer ID: 0905, Customer: David Davis, Automobile: Nissan Altima SV
Customer ID: 0906, Customer: Sarah Martinez, Automobile: Volkswagen Passat SE
Customer ID: 0907, Customer: John Doe, Automobile: BMW 3 Series 320i
Customer ID: 0902, Customer: Jane Smith, Automobile: Mercedes-Benz C-Class C300
Customer ID: 0909, Customer: James White, Automobile: Audi A4 Premium
Customer ID: 0910, Customer: Michael Johnson, Automobile: Lexus ES 350
Customer ID: 0902, Customer: Jane Smith, Automobile: Mercedes-Benz C-Class C300
Customer ID: 0909, Customer: James White, Automobile: Audi A4 Premium
Customer ID: 0910, Customer: Mia Martinez, Automobile: Lexus ES 350
Customer ID: 0908, Customer: James White, Automobile: Mercedes-Benz C-Class C300
Customer ID: 0909, Customer: James White, Automobile: Audi A4 Premium
Customer ID: 0910, Customer: Mia Martinez, Automobile: Lexus ES 350

PL/SQL procedure successfully completed.
Elapsed: 00:00:00.005

Powered by ORDS

Screenshot-5:

The screenshot shows the Oracle Database Actions SQL Worksheet interface. The code in the worksheet is a PL/SQL procedure that retrieves employee jobs. The output shows records for various employees and their assigned jobs.

```

1  BEGIN
2      automotive_package.get_employee_jobs;
3  END;
4

```

Employee Jobs

Employee: anu Deepthi, Job: Air Conditioning Repair, Automobile: C-Class
Employee: anu Deepthi, Job: Air Conditioning Repair, Automobile: C-Class
Employee: anu Deepthi, Job: Suspension Inspection, Automobile: A4
Employee: nik Smith, Job: Air Conditioning Repair, Automobile: C-Class
Employee: nik Smith, Job: Suspension Inspection, Automobile: A4
Employee: kar John, Job: Air Conditioning Repair, Automobile: C-Premium
Employee: kar John, Job: Suspension Inspection, Automobile: A4
Employee: jain Willy, Job: Battery Replacement, Automobile: ES
Employee: jain Willy, Job: Air Conditioning Repair, Automobile: C-Class
Employee: jain Willy, Job: Battery Replacement, Automobile: ES
Employee: John Doe, Job: Routine Maintenance, Automobile: Camry
Employee: John Doe, Job: Oil Change, Automobile: Civic
Employee: Jane Smith, Job: Oil Change, Automobile: Civic
Employee: Michael Johnson, Job: Brake Repair, Automobile: Mustang
Employee: Sarah Williams, Job: Tire Rotation, Automobile: Matador
Employee: Emily Miller, Job: Check Engine Light, Automobile: Altima
Employee: Emily Miller, Job: Transmission Service, Automobile: Passat
Employee: James Anderson, Job: Alignment, Automobile: 3 Series
Employee: Olivia Garcia, Job: Air Conditioning Repair, Automobile: C-Class
Employee: Robert Martinez, Job: Suspension Inspection, Automobile: A4
Employee: Ava Taylor, Job: Battery Replacement, Automobile: ES

PL/SQL procedure successfully completed.
Elapsed: 00:00:00.008

Powered by ORDS

Output for Function:

```
-----  
Customer ID: 0902, Customer: Jane Smith, Automobile: Toyota Camry LE  
Customer ID: 0901, Customer: John Doe, Automobile: Toyota Camry LE  
Customer ID: 0902, Customer: Jane Smith, Automobile: Honda Civic EX  
Customer ID: 0903, Customer: Michael Johnson, Automobile: Ford Mustang GT  
Customer ID: 0904, Customer: Emily Brown, Automobile: Chevrolet Malibu LT  
Customer ID: 0905, Customer: David Davis, Automobile: Nissan Altima SV  
Customer ID: 0906, Customer: Sarah Wilson, Automobile: Volkswagen Passat SE  
Customer ID: 0901, Customer: John Doe, Automobile: BMW 3 Series 320i  
Customer ID: 0902, Customer: Jane Smith, Automobile: Mercedes-Benz C-Class C300  
Customer ID: 0909, Customer: James White, Automobile: Audi A4 Premium  
Customer ID: 0910, Customer: Mia Martinez, Automobile: Lexus ES 350  
Customer ID: 0902, Customer: Jane Smith, Automobile: Mercedes-Benz C-Class C300  
Customer ID: 0909, Customer: James White, Automobile: Audi A4 Premium  
Customer ID: 0910, Customer: Mia Martinez, Automobile: Lexus ES 350  
Customer ID: 0902, Customer: Jane Smith, Automobile: Mercedes-Benz C-Class C300  
Customer ID: 0909, Customer: James White, Automobile: Audi A4 Premium  
Customer ID: 0910, Customer: Mia Martinez, Automobile: Lexus ES 350  
-----
```

Output for Procedure:

Employee Jobs

```
-----  
Employee: anu Doeph, Job: Air Conditioning Repair, Automobile: C-Class  
Employee: anu Doeph, Job: Air Conditioning Repair, Automobile: C-Class  
Employee: anu Doeph, Job: Suspension Inspection, Automobile: A4  
Employee: nik Smit, Job: Air Conditioning Repair, Automobile: C-Class  
Employee: nik Smit, Job: Suspension Inspection, Automobile: A4  
Employee: kar John, Job: Air Conditioning Repair, Automobile: C-Class  
Employee: kar John, Job: Suspension Inspection, Automobile: A4  
Employee: kar John, Job: Battery Replacement, Automobile: ES  
Employee: jain Willy, Job: Air Conditioning Repair, Automobile: C-Class  
Employee: jain Willy, Job: Battery Replacement, Automobile: ES  
Employee: John Doe, Job: Routine Maintenance, Automobile: Camry  
Employee: Jane Smith, Job: Oil Change, Automobile: Camry  
Employee: Jane Smith, Job: Oil Change, Automobile: Civic  
Employee: Michael Johnson, Job: Brake Repair, Automobile: Mustang  
Employee: Sarah Williams, Job: Tire Rotation, Automobile: Malibu  
Employee: David Brown, Job: Check Engine Light, Automobile: Altima  
Employee: Emily Miller, Job: Transmission Service, Automobile: Passat  
Employee: James Anderson, Job: Alignment, Automobile: 3 Series  
Employee: Olivia Garcia, Job: Air Conditioning Repair, Automobile: C-Class  
Employee: Robert Martinez, Job: Suspension Inspection, Automobile: A4  
Employee: Ava Taylor, Job: Battery Replacement, Automobile: ES
```

PL/SQL procedure successfully completed.

Explanation: The provided PL/SQL package, 'automotive_package,' and its body serve two main purposes:

The 'get_customers_with_purchased_automobiles' function retrieves information about customers and the automobiles they have purchased. It returns a list of customer IDs, first and last names, as well as details about the automobiles they own.

The 'get_employee_jobs' procedure prints a list of automobiles serviced by employees in an automotive retailer. It retrieves data about employees, their associated jobs, and the automobiles they've serviced, and then prints this information.

To use these functionalities, you can execute the function to retrieve customer and automobile data and execute the procedure to display employee job details. This package provides a convenient way to access and display essential information in an automotive context.

Analysis					
Table	Attributes	Domain Contraints	Constraints	Constraint Name	Relations
automotive_retailer	id	RAW(16)	PRIMARY KEY	NA	automotive_retailer has one to many relationship with employee, automotive_retailer has one to many relationship with inventory_product, automotive_retailer has one to one relationship with address, automotive_retailer has one to many relationship with job
	phone	VARCHAR2(20)	NOT NULL, UNIQUE, CHECK (REGEXP_LIKE (phone, '^(\d{3})\ \d{3}-\d {4}\$'))	invalid_automotive_retailer_phone	
	email	VARCHAR2(255)	NOT NULL, UNIQUE, CHECK (REGEXP_LIKE (email, '[A-Za-z0-9._%+-]+@[A- Za-z0-9.-]+\.[A-Za-z]{2,}'))	invalid_automotive_retailer_email	
	website	VARCHAR2(255)	NOT NULL, UNIQUE	NA	
	business_hours	VARCHAR2(20)	NOT NULL	NA	
	manager_id	RAW(16)	NOT NULL, FOREIGN KEY (manager_id) REFERENCES employee(id)	NA	
	address_id	RAW(16)	NOT NULL, FOREIGN KEY (address_id) REFERENCES address(id)	NA	
inventory_product	id	RAW(16)	PRIMARY KEY	NA	inventory_product has many to many relationship with supplier, inventory_product has many to one realationship with automotive_retailer, inventory_product has many to many relationship with bill, inventory_product has one to many relationship with part_service, inventory_product has many to one relationship with automobile, inventory_product has many to one relationship with address
	name	VARCHAR2(100)	NOT NULL	NA	
	quantity	INTEGER	NOT NULL	NA	
	price	INTEGER	NOT NULL	NA	
	automotive_retailer_id	RAW(16)	NOT NULL, FOREIGN KEY (automotive_retailer_id) REFERENCES automotive_retailer (id)	NA	
	automobile_id	RAW(16)	"NOT NULL, FOREIGN KEY (automobile_id) REFERENCES automobile(id)"	NA	
	address_id	RAW(16)	NOT NULL, FOREIGN KEY (address_id) REFERENCES address(id)	NA	
automobile	id	RAW(16)	PRIMARY KEY	NA	automobile has one to many relationship with inventory_product, automobile has one to many relationship with job
	manufacturer	VARCHAR2(100)	NOT NULL	NA	
	name	VARCHAR2(100)	NOT NULL	NA	
	variant	VARCHAR2(100)	NOT NULL	NA	
	year	VARCHAR2(10)	NOT NULL	NA	
	color	VARCHAR2(100)	NOT NULL	NA	
supplier	id	RAW(16)	PRIMARY KEY	NA	supplier has many to many relationship with inventory_product, supplier has one to one relationship with address
	name	VARCHAR2(100)	NOT NULL	NA	
	phone	VARCHAR2(20)	NOT NULL, UNIQUE, CHECK (REGEXP_LIKE (phone, '^(\d{3})\ \d{3}-\d {4}\$'))	invalid_supplier_phone	
	address_id	RAW(16)	NOT NULL, FOREIGN KEY (address_id) REFERENCES address(id)	NA	

Analysis					
Table	Attributes	Domain Contraints	Constraints	Constraint Name	Relations
Employee	id	RAW(16)	PRIMARY KEY	NA	employee has one to many relationship with employee_payroll, employee has many to one relationship with automotive_retailer, employee has one to many relationship with bill, employee has many to one relationship with address, employee has many to many relationship with job
	first_name	VARCHAR2(100)	NOT NULL	NA	
	last_name	VARCHAR2(100)	NOT NULL	NA	
	dob	TIMESTAMP	NOT NULL	NA	
	phone	VARCHAR2(20)	NOT NULL, UNIQUE, CHECK (REGEXP_LIKE (phone, '^(\d{3})\.\d{3}-\d{4}\$'))	invalid_employee_phone	
	email	VARCHAR2(255)	NOT NULL, UNIQUE, CHECK (REGEXP_LIKE (email, '^[A-Za-z0-9._%+-]+@[A- Za-z0-9.-]+\.[A-Za-z]{2,}\$'))	invalid_employee_email	
	annual_salary	INTEGER	NOT NULL	NA	
	ssn	CHAR(9)	NOT NULL, UNIQUE	NA	
	automotive_retailer_id	RAW(16)	NOT NULL, FOREIGN KEY (automotive_retailer_id) REFERENCES automotive_retailer (id)	NA	
employee_payroll	address_id	RAW(16)	NOT NULL, FOREIGN KEY (address_id) REFERENCES address(id)	NA	employee_payroll has many to one relationship with employee
	hire_date	TIMESTAMP	NOT NULL	NA	
	id	RAW(16)	PRIMARY KEY	NA	
	hours_worked	NUMBER	NOT NULL	NA	
	start_date	TIMESTAMP	NOT NULL	NA	
	end_date	TIMESTAMP	NOT NULL	NA	
address	pay	NUMBER	NOT NULL	NA	address has one to many relationship with employee, address has one to many relationship with inventory_product, address has one to one relationship with automotive_retailer, address has one to one relationship with supplier, address has one to many relationship with customer
	employee_id	RAW(16)	NOT NULL, FOREIGN KEY (employee_id) REFERENCES employee(id)	NA	
	id	RAW(16)	PRIMARY KEY	NA	
	apartment_no	NUMBER	NOT NULL	NA	
	street	VARCHAR2(100)	NOT NULL	NA	
	city	VARCHAR2(100)	NOT NULL	NA	
	state	VARCHAR2(50)	NOT NULL	NA	
bill	country	VARCHAR2(50)	NOT NULL	NA	
	zip	CHAR(5)	NOT NULL	NA	
	id	RAW(16)	PRIMARY KEY	NA	
mode_of_payment	date	TIMESTAMP	NOT NULL	NA	
	mode_of_payment	CHAR(4)	NOT NULL, CHECK (mode_of_payment IN ('CARD', 'CASH'))	invalid_mode_of_payment	

Analysis					
Table	Attributes	Domain Contraints	Constraints	Constraint Name	Relations
bill	insurance_id	RAW(16)	NOT NULL, FOREIGN KEY (insurance_id) REFERENCES insurance(id)	NA	bill has many to one relationship with employee, bill has many to many relationship with inventory_product, bill has many to one relationship with insurance, bill has many to one relationship with customer, bill has one to one relationship with job, bill has many to one relationship with payment_plan
	customer_id	RAW(16)	NOT NULL, FOREIGN KEY (customer_id) REFERENCES customer(id)	NA	
	job_id	RAW(16)	NOT NULL, FOREIGN KEY (job_id) REFERENCES job(id)	NA	
	employee_id	RAW(16)	NOT NULL, FOREIGN KEY (employee_id) REFERENCES employee(id)	NA	
	sale_type	CHAR(7)	NOT NULL, CHECK (mode_of_payment IN ('ONLINE', 'OFFLINE'))	invalid_sale_type	
	payment_plan_id	RAW(16)	NOT NULL, FOREIGN KEY (payment_plan_id) REFERENCES payment_plan(id)	NA	
job	id	RAW(16)	PRIMARY KEY	NA	job has one to one realtionship with bill, job has many to one realtionship with automobile, job has many to one realtionship with automotive_retailer, job has many to many relationship with employee, job has one to many realtionship with part_service, job has many to one realtionship with customer
	date	TIMESTAMP	NOT NULL	NA	
	description	VARCHAR2(255)	NOT NULL	NA	
	customer_id	RAW(16)	NOT NULL, FOREIGN KEY (customer_id) REFERENCES customer(id)	NA	
	automotive_retailer_id	RAW(16)	NOT NULL, FOREIGN KEY (automotive_retailer_id) REFERENCES automotive_retailer (id)	NA	
	vin_number	VARCHAR2(100)	NOT NULL, CHECK()	NA	
	automobile_id	RAW(16)	NOT NULL, FOREIGN KEY (automobile_id) REFERENCES automobile(id)	NA	
	job_status	VARCHAR2(20)		NA	
part_service	id	RAW(16)	PRIMARY KEY	NA	part_service has many to one relationship with job, part_service has one to one relationship with inventory_product
	name	VARCHAR2(100)	NOT NULL	NA	
	quantity	INTEGER	NOT NULL	NA	
	job_id	RAW(16)	NOT NULL, FOREIGN KEY (job_id) REFERENCES job(id)	NA	
	inventory_product_id	RAW(16)	NOT NULL, FOREIGN KEY (inventory_product_id) REFERENCES inventory_product (id)	NA	
	type	CHAR(7)	NOT NULL, CHECK (mode_of_payment IN ('SERVICE', 'PART'))	invalid_part_service_type	
	id	RAW(16)	PRIMARY KEY	NA	

Analysis					
Table	Attributes	Domain Constraints	Constraints	Constraint Name	Relations
payment_plan	name	VARCHAR2(100)	NOT NULL	NA	payment_plan has one to many relationship with bill
	installments	INTEGER	NOT NULL	NA	
	interest	INTEGER	NOT NULL	NA	
customer	id	RAW(16)	PRIMARY KEY	NA	customer has many to many relationship with insurance, customer has one to many relationship with job, customer has many to one relationship with address, customer has one to many relationship with bills
	first_name	VARCHAR2(100)	NOT NULL	NA	
	last_name	VARCHAR2(100)	NOT NULL	NA	
	dob	TIMESTAMP	NOT NULL	NA	
	driverlicense	VARCHAR2(50)	UNIQUE, NOT NULL	NA	
	phone	VARCHAR2(20)	NOT NULL, UNIQUE, CHECK (REGEXP_LIKE (phone, '^(\d{3})\.\d{3}-\d{4}\$'))	invalid_customer_phone	
	address_id	RAW(16)	NOT NULL, FOREIGN KEY (address_id) REFERENCES address(id)	NA	
insurance	id	RAW(16)	PRIMARY KEY	NA	insurance has many to many relationship with customer, insurance has one to many relationship with bill
	policy_type	VARCHAR2(100)	NOT NULL	NA	
	provider	VARCHAR2(100)	NOT NULL	NA	
	claim_percentage	INTEGER	NOT NULL	NA	
inventory_product_supplier	inventory_product_id	RAW(16)	PRIMARY KEY, NOT NULL, FOREIGN KEY (inventory_product_id) REFERENCES inventory_product (id)"	NA	NA
	supplier_id	RAW(16)	PRIMARY KEY, NOT NULL, FOREIGN KEY (supplier_id) REFERENCES supplier(id)	NA	
	quantity	INTEGER	NOT NULL	NA	
bill_inventory_product	bill_id	RAW(16)	PRIMARY KEY, NOT NULL, FOREIGN KEY (bill_id) REFERENCES bill(id)"	NA	NA
	inventory_product_id	RAW(16)	PRIMARY KEY, NOT NULL, FOREIGN KEY (inventory_product_id) REFERENCES inventory_product (id)"	NA	
	quantity	INTEGER	NOT NULL	NA	
	job_id	RAW(16)	PRIMARY KEY, NOT NULL, FOREIGN KEY (job_id) REFERENCES job(id)"	NA	

<u>Analysis</u>					
Table	Attributes	Domain Contraints	Constraints	Constraint Name	Relations
job_employee	employee_id	RAW(16)	PRIMARY KEY, NOT NULL, FOREIGN KEY (employee_id) REFERENCES employee(id)"	NA	NA
customer_insurance	customer_id	RAW(16)	PRIMARY KEY, NOT NULL, FOREIGN KEY (customer_id) REFERENCES customer(id)"	NA	NA
	insurance_id	RAW(16)	PRIMARY KEY, NOT NULL, FOREIGN KEY (insurance_id) REFERENCES insurance(id)"	NA	
	status	CHAR(8)	NOT NULL	NA	