AutoZone
(A National retail and service store)

Name: Naga Venkata Kanakalakshmi
UNT ID: 11725119
Project -Part 3 (Queries, updates, and deletes)

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.

Note: No changes were made to the description as part of project part 3, with a focus on resolving queries.

Assumptions:

- A product can be available both in store and in warehouse. If the product has automative_retailer_id same as address id, that means the product is in store. And if automative_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.

Along with the above assumptions we have added few more new assumptions as part of project part 3:

- 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.

Note: In Project Part-3, no modifications were made to the existing data or relationships between tables. To verify the correctness of the Entity-Relationship (ER) diagram, please refer to the Part 2 of the project.

Analysis

Table	Attributes	Domain Contraints	Contraints	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	

Analysis

Table	Attributes	Domain Contraints	Contraints	Constraint Name	Relations
	address_id	RAW(16)	NOT NULL, FOREIGN KEY (address_id) REFERENCES address(id)	NA	
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	
	address_id	RAW(16)	NOT NULL, FOREIGN KEY (address_id) REFERENCES address(id)	NA	
	hire_date	TIMESTAMP	NOT NULL	NA	
employee_payroll	id	RAW(16)	PRIMARY KEY	NA	employee_payroll has many to one relationship with employee
	hours_worked	NUMBER	NOT NULL	NA	
	start_date	TIMESTAMP	NOT NULL	NA	
	end_date	TIMESTAMP	NOT NULL	NA	
	pay	NUMBER	NOT NULL	NA	
	employee_id	RAW(16)	NOT NULL, FOREIGN KEY (employee_id) REFERENCES employee(id)	NA	
address	id	RAW(16)	PRIMARY KEY	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
	apartment_no	NUMBER	NOT NULL	NA	
	street	VARCHAR2(100)	NOT NULL	NA	
	city	VARCHAR2(100)	NOT NULL	NA	
	state	VARCHAR2(50)	NOT NULL	NA	
	country	VARCHAR2(50)	NOT NULL	NA	
	zip	CHAR(5)	NOT NULL	NA	
	id	RAW(16)	PRIMARY KEY	NA	

Analysis

Table	Attributes	Domain Contraints	Contraints	Constraint Name	Relations
bill	date	TIMESTAMP	NOT NULL	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
	mode_of_payment	CHAR(4)	NOT NULL, CHECK (mode_of_payment IN ('CARD', 'CASH'))	invalid_mode_of_payment	
	insurance_id	RAW(16)	NOT NULL, FOREIGN KEY (insurance_id) REFERENCES insurance(id)	NA	
	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	
	id	RAW(16)	PRIMARY KEY	NA	part_service has many to one relationship with job
	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	

Analysis

Table	Attributes	Domain Contraints	Contraints	Constraint Name	Relations
part_service	inventory_product_id	RAW(16)	NOT NULL, FOREIGN KEY (inventory_product_id) REFERENCES inventory_product(id)	NA	part_service has many to one relationship with job, part_service has one to one relationship with inventary_product
	type	CHAR(7)	NOT NULL, CHECK (mode_of_payment IN ('SERVICE', 'PART'))	invalid_part_service_type	
payment_plan	id	RAW(16)	PRIMARY KEY	NA	payment_plan has one to many relationship with bill
	name	VARCHAR2(100)	NOT NULL	NA	
	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_id	RAW(16)	PRIMARY KEY, NOT NULL, FOREIGN KEY (bill_id) REFERENCES bill(id)"	NA	

Analysis

Table	Attributes	Domain Contraints	Contraints	Constraint Name	Relations
bill_inventory_product	inventory_product_id	RAW(16)	PRIMARY KEY, NOT NULL, FOREIGN KEY (inventory_product_id) REFERENCES inventory_product(id)"	NA	NA
	quantity	INTEGER	NOT NULL	NA	
job_employee	job_id	RAW(16)	PRIMARY KEY, NOT NULL, FOREIGN KEY (job_id) REFERENCES job(id)"	NA	NA
	employee_id	RAW(16)	PRIMARY KEY, NOT NULL, FOREIGN KEY (employee_id) REFERENCES employee(id)"	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	

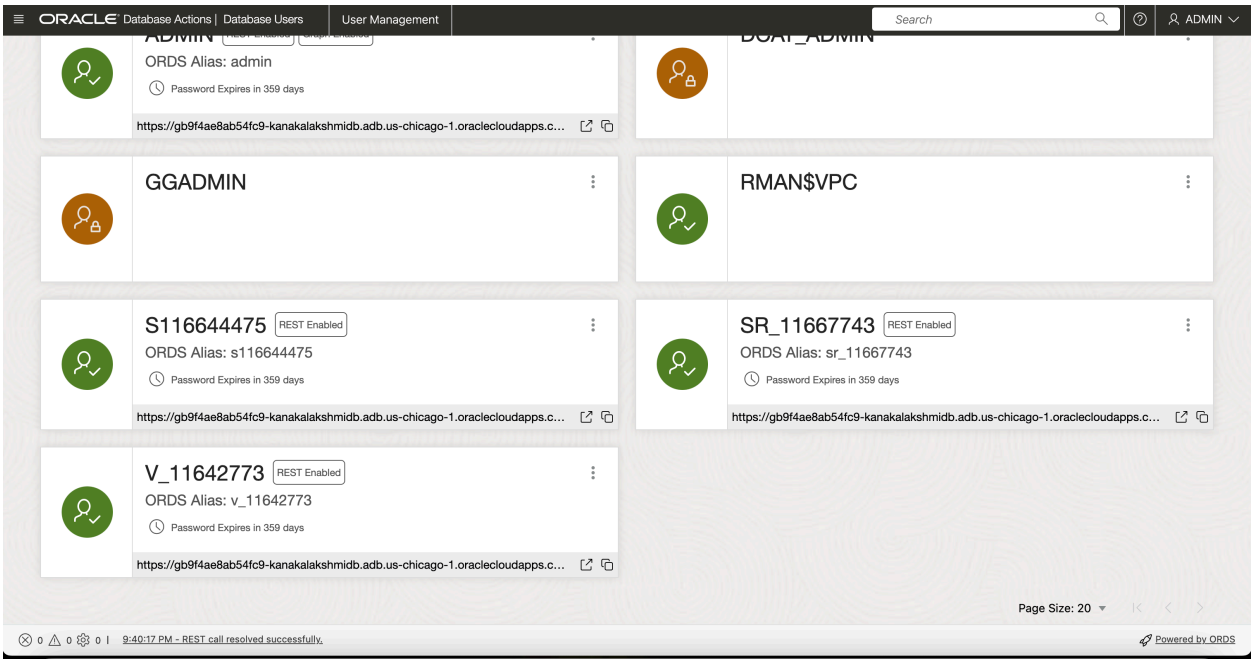
The cloud account users for Group-8 are as follows.

ADMIN – Kanakalakshmi Murikipudi

V_11642773 – Vishnu Suidreddy - https://gb9f4ae8ab54fc9-kanakalakshmidb.adb.us-chicago-1.oraclecloudapps.com/ords/v_11642773/_sdw/

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

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



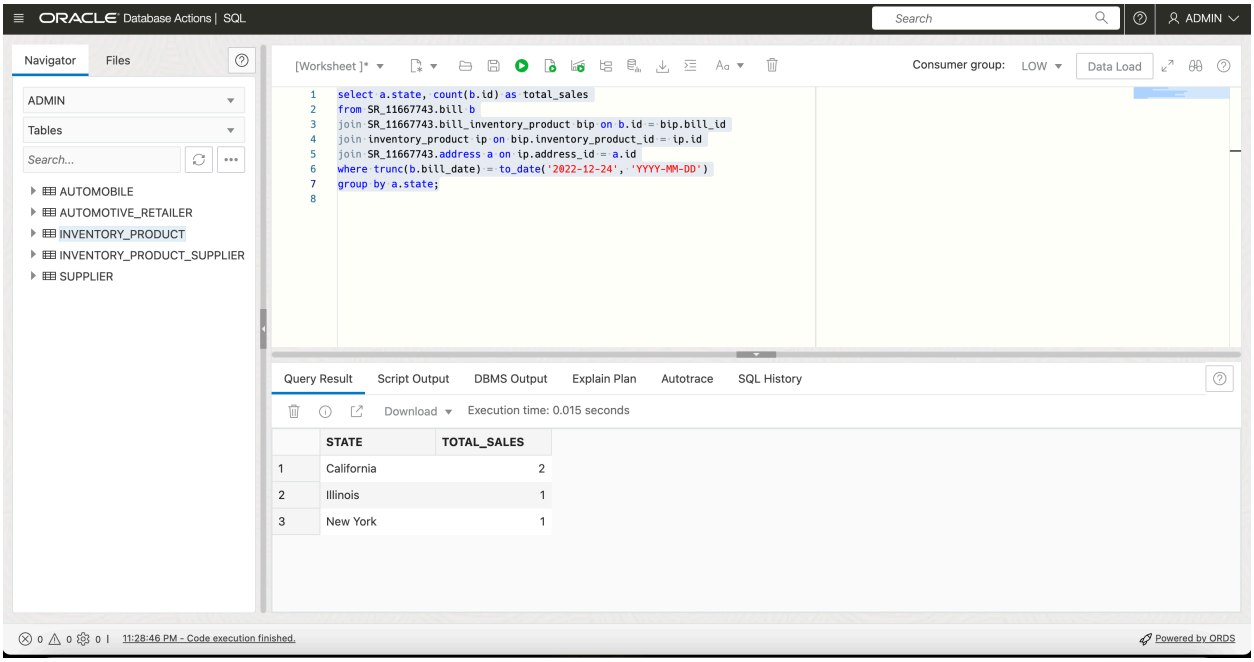
Queries of Project Part-3

Query-1: List the total number of sales on December 24, 2022 group by state.

SQL Query:

```
select a.state, count(b.id) as total_sales
from SR_11667743.bill b
join SR_11667743.bill_inventory_product bip on b.id = bip.bill_id
join inventory_product ip on bip.inventory_product_id = ip.id
join SR_11667743.address a on ip.address_id = a.id
where trunc(b.bill_date) = to_date('2022-12-24', 'YYYY-MM-DD')
group by a.state;
```

Screenshot:



Explanation: In the SQL query, we need to list the total number of sales on December 24, 2022 group by state. To achieve this, the query filters sales data by the bill_date column, specifically looking for sales on December 24, 2022. It then groups the results by the state in which the sales occurred, calculating the total number of sales for each state.

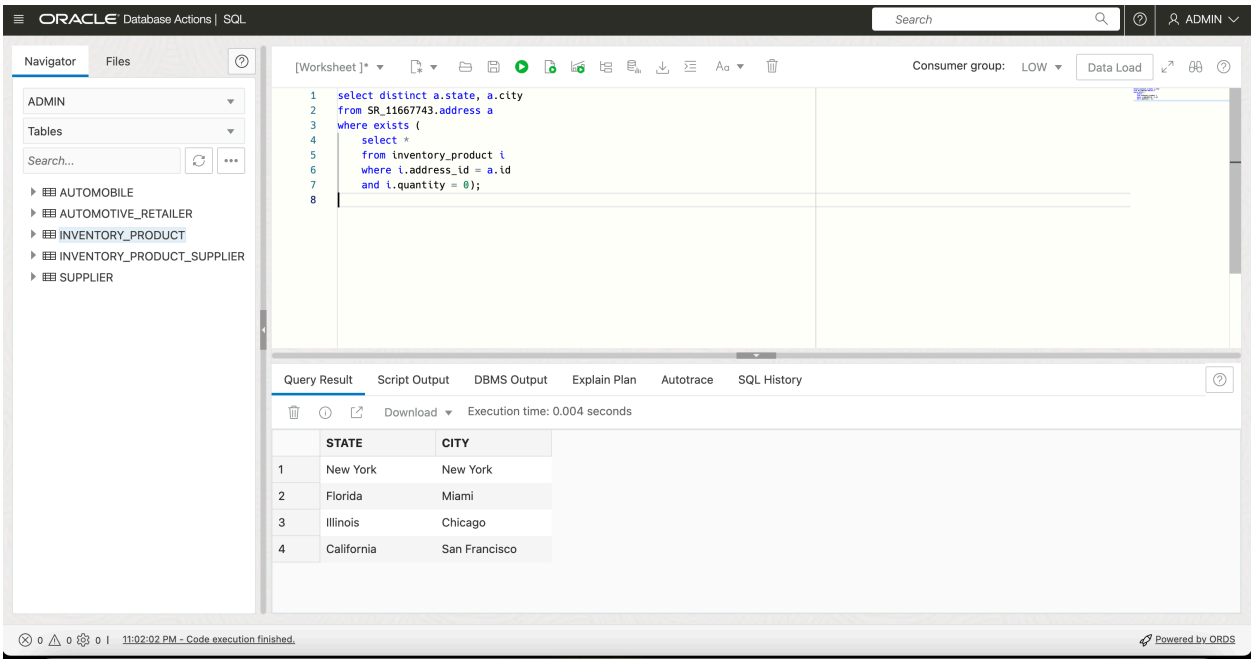
As a result of execution, we got three states in which sales happened along with the total number of sales on 24th December 2022. Those are California, Illinois and New York.

Query-2: Find locations with inventories that list at least one missing product (a product that has quantity of zero in the inventory)

SQL Query:

```
select distinct a.state, a.city
from SR_11667743.address a
where exists (
   select *
   from inventory_product i
   where i.address_id = a.id
   and i.quantity = 0);
```

Screenshot:



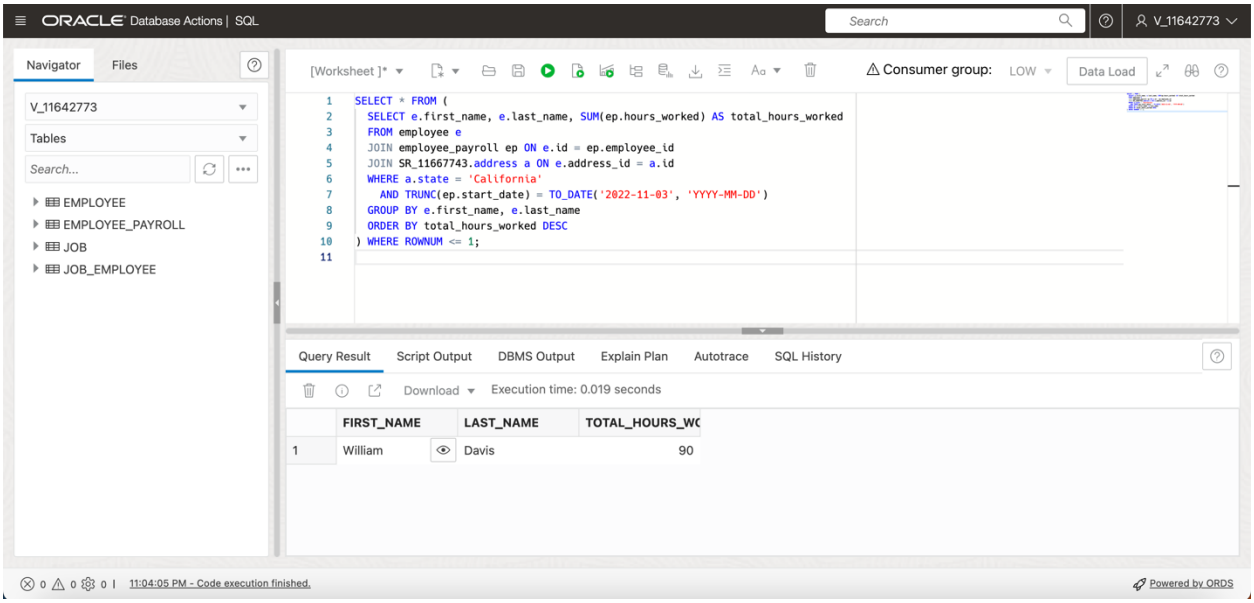
Explanation: We need to retrieve locations (which includes states and cities) with inventories that contain at least one missing product, which is defined as a product with a quantity of zero in the inventory. So, this query retrieves the unique state and city pairs from the "address" table for locations where there is at least one missing product. This is achieved by linking the "address" table with the "inventory_product" table and filtering for products with zero quantity at each location.
We got 4 records of locations with missing inventory product.

Query-3: Find the name of the employee(s) from California that had worked the most hours on November 3, 2022

SQL Query:

```
SELECT * FROM (
SELECT e.first_name, e.last_name, SUM(ep.hours_worked) AS total_hours_worked
FROM employee e
JOIN employee_payroll ep ON e.id = ep.employee_id
JOIN SR_11667743.address a ON e.address_id = a.id
WHERE a.state = 'California'
AND TRUNC(ep.start_date) = TO_DATE('2022-11-03', 'YYYY-MM-DD')
GROUP BY e.first_name, e.last_name
ORDER BY total_hours_worked DESC
) WHERE ROWNUM <= 1;
```

Screenshot:



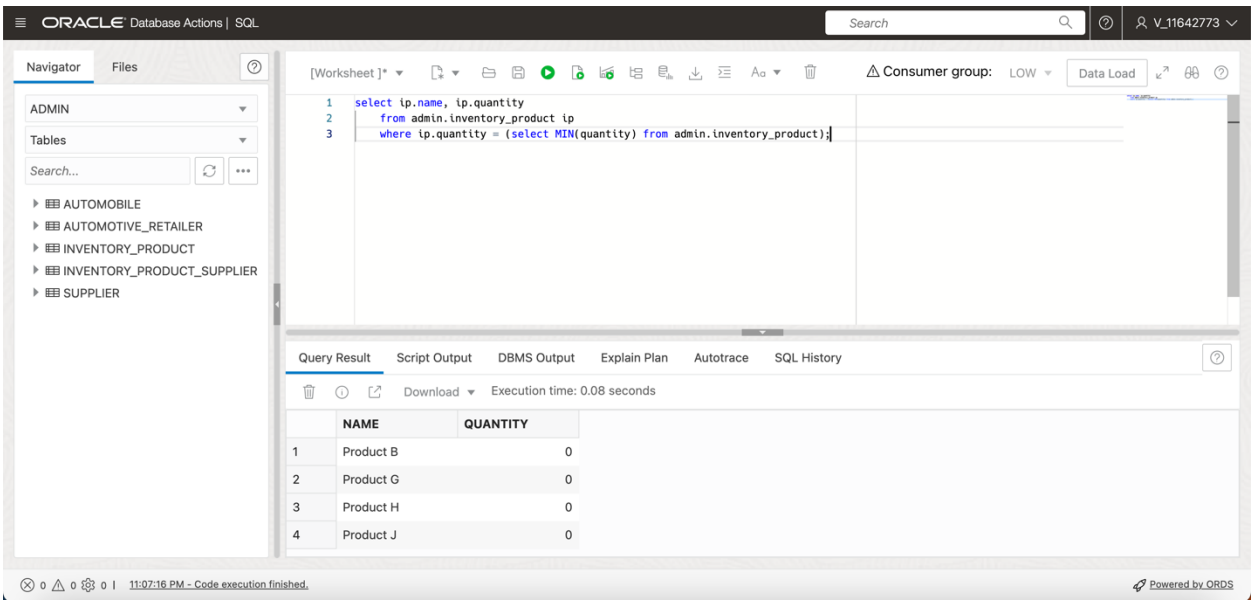
Explanation: We need to retrieve the names of all employees from California who worked the most hours on November 3, 2022. We retrieved this information by joining the employee table with the employee_payroll and address tables, filtering for the state ‘California’, and then grouping and ordering the results by total hours worked, returning the employee(s) with the highest hours worked on the specified date.

Query-4: List the items that currently have the least quantity on inventory.

SQL Query:

```
select ip.name, ip.quantity
from admin.inventory_product ip
where ip.quantity = (select MIN(quantity) from admin.inventory_product);
```

Screenshot:



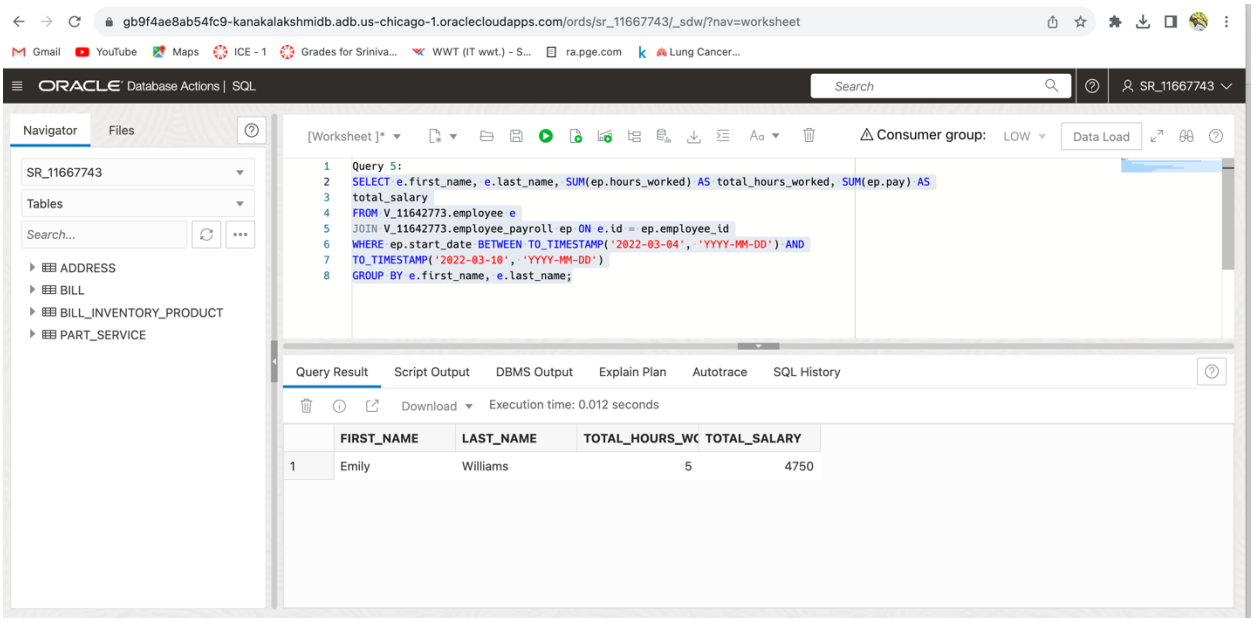
Explanation: We need to retrieves the items with the least quantity in the inventory_product table. This query finds and lists the items in the inventory with the smallest quantity. It first figures out what the minimum quantity is in the inventory, and then it selects and shows you the items that have exactly that minimum amount in quantity.

Query-5: Print the payroll from March 4, 2022 to March 10, 2022 displaying employee name, hours worked and total salary for all employees.

SQL Query:

```
SELECT e.first_name, e.last_name, SUM(ep.hours_worked) AS total_hours_worked, SUM(ep.pay) AS
total_salary
FROM V_11642773.employee e
JOIN V_11642773.employee_payroll ep ON e.id = ep.employee_id
WHERE ep.start_date BETWEEN TO_TIMESTAMP('2022-03-04', 'YYYY-MM-DD') AND
TO_TIMESTAMP('2022-03-10', 'YYYY-MM-DD')
GROUP BY e.first_name, e.last_name;
```

Screenshot:



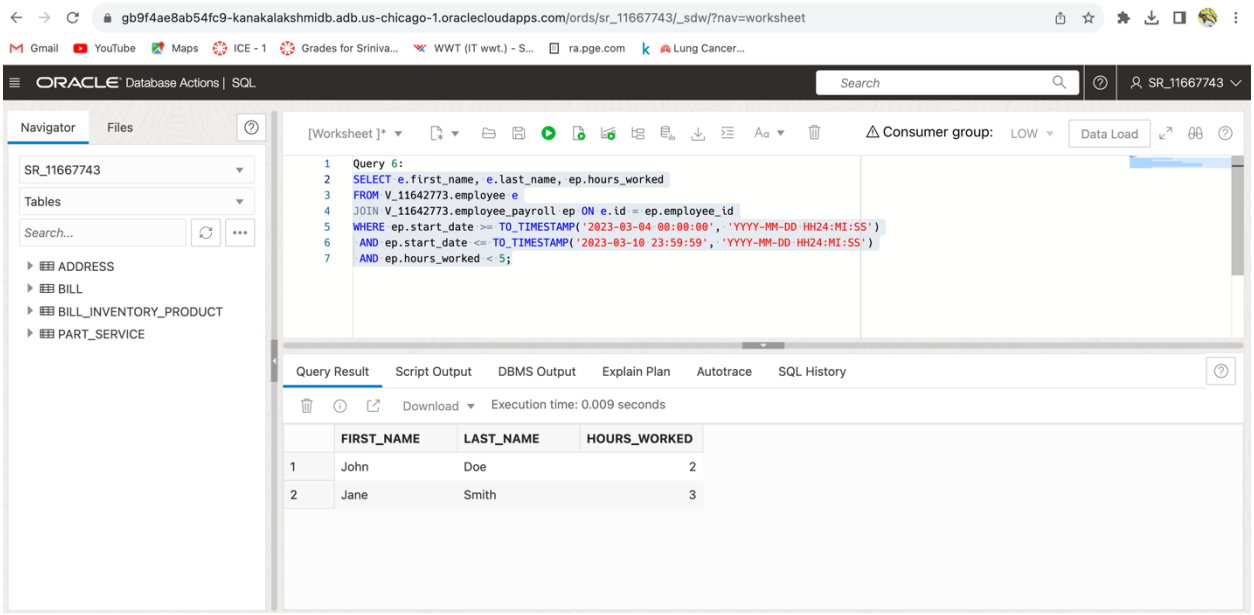
Explanation: In query, we need to show payroll data i.e total hours worked and the total salary they received for employees who worked from March 4, 2022, to March 10, 2022. It combines information from the "employee" and "employee_payroll" tables, showing each employee's first name, last name, total hours worked during that week, and their overall earnings. To achieve this, it connects the two tables by employee ID, filters the results for the specified date range, and groups the data by first name and last name to avoid duplicates.

Query-6: Design a delete statement to delete employees working less than 5 hours from March 4, 2023 to March 10, 2023.

SQL Query Before Delete:

```
SELECT e.first_name, e.last_name, ep.hours_worked
FROM V_11642773.employee e
JOIN V_11642773.employee_payroll ep ON e.id = ep.employee_id
WHERE ep.start_date >= TO_TIMESTAMP('2023-03-04 00:00:00', 'YYYY-MM-DD HH24:MI:SS')
AND ep.start_date <= TO_TIMESTAMP('2023-03-10 23:59:59', 'YYYY-MM-DD HH24:MI:SS')
AND ep.hours_worked < 5;
```

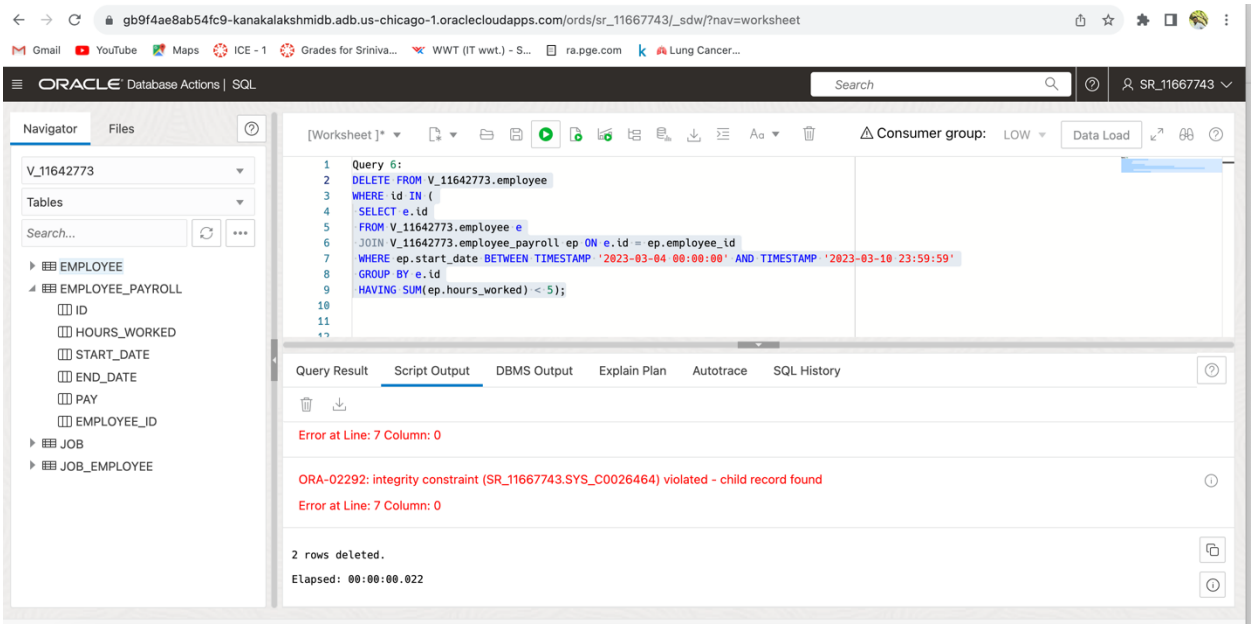
Screenshot-1:



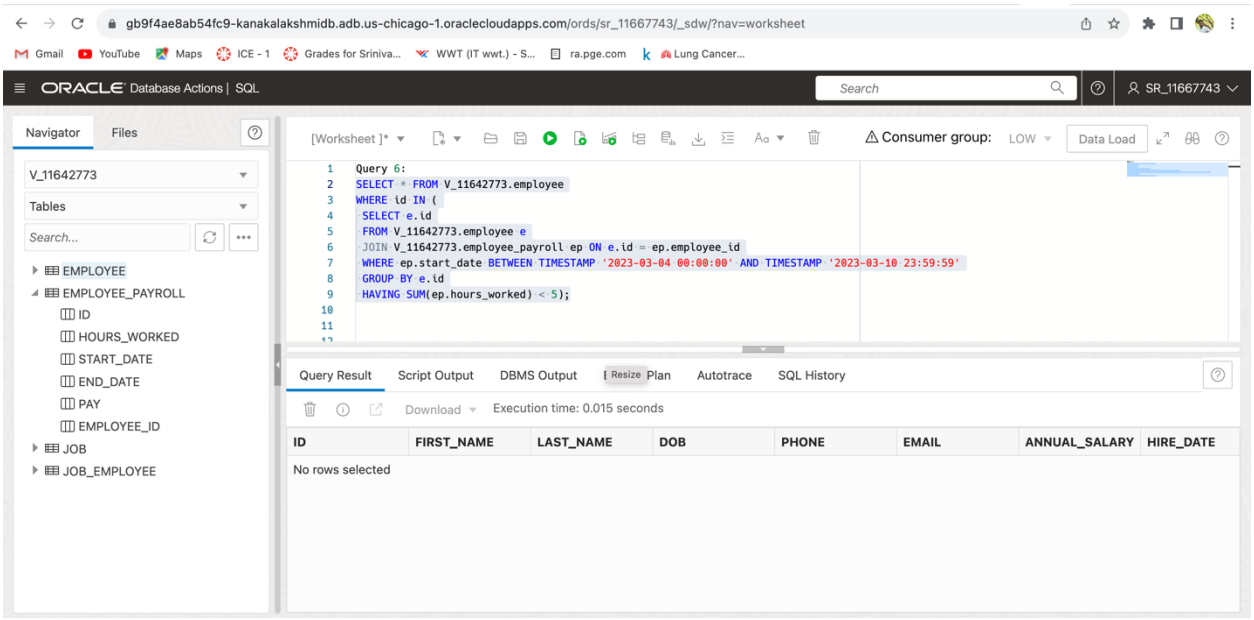
SQL Query to Delete:

```
DELETE FROM V_11642773.employee
WHERE id IN (
   SELECT e.id
   FROM V_11642773.employee e
   JOIN V_11642773.employee_payroll ep ON e.id = ep.employee_id
   WHERE ep.start_date BETWEEN TIMESTAMP ‘2023-03-04 00:00:00’ AND TIMESTAMP ‘2023-03-10 23:59:59’
   GROUP BY e.id
   HAVING SUM(ep.hours_worked) < 5);
```

Screenshot-2:



Screenshot-3: cross validation after deleting the records.



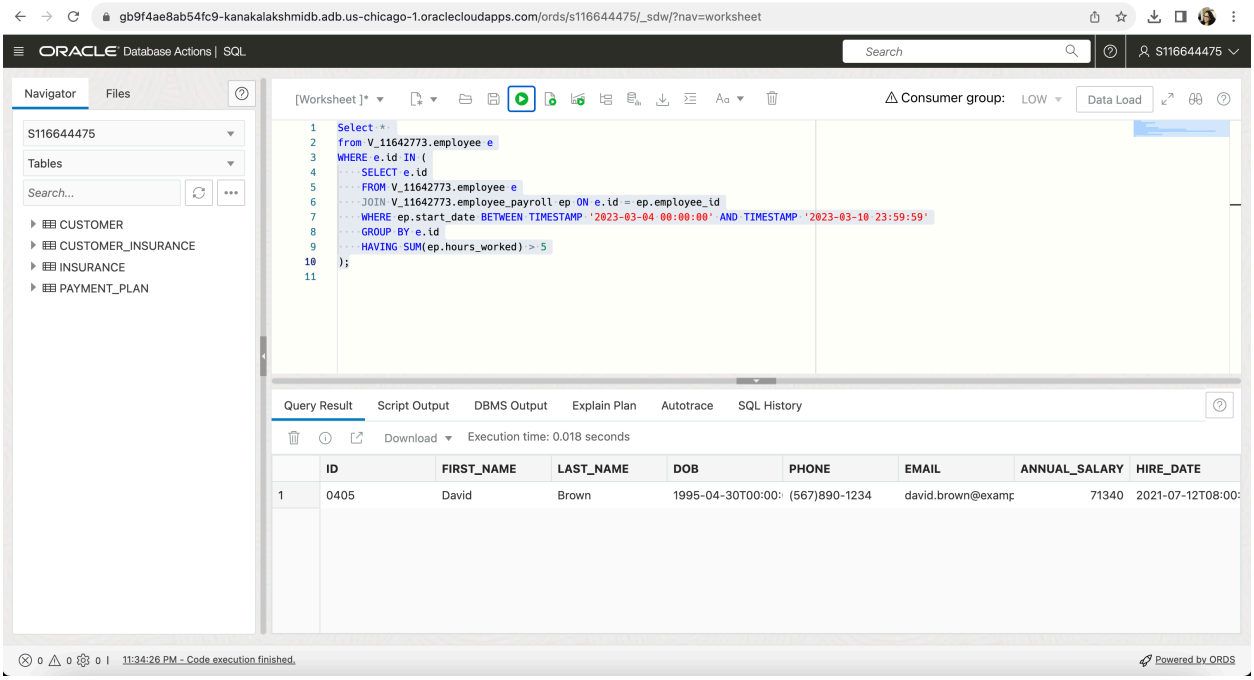
Explanation: This query is to find employees who worked less than 5 hours from March 4, 2023, to March 10, 2023. It provides with the employee details like their first names, last names, and the hours they worked during this period from table ‘employee’, joining it with another table employee_payroll using the employee ID as the common key. This information can be useful for monitoring and managing employee attendance and work hours.

Query-7: Design an update statement to give a 23% salary raise to employees working more than 5 hours from March 4, 2023 to March 10, 2023.

Query Before Update:

```
Select *
from V_11642773.employee e
WHERE e.id IN (
   SELECT e.id
   FROM V_11642773.employee e
   JOIN V_11642773.employee_payroll ep ON e.id = ep.employee_id
   WHERE ep.start_date BETWEEN TIMESTAMP '2023-03-04 00:00:00' AND TIMESTAMP '2023-03-10
23:59:59'
   GROUP BY e.id
   HAVING SUM(ep.hours_worked) > 5
);
```

Screenshot-1:

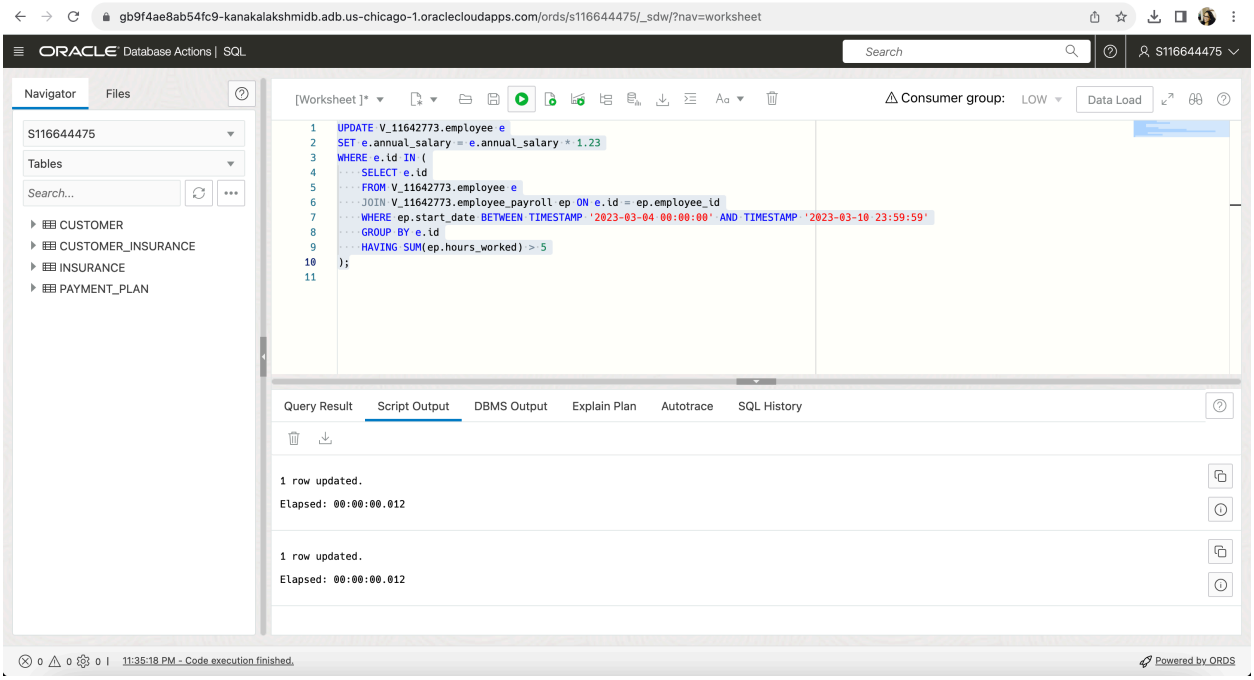


Explanation: As we have to design an update statement to give a 23% salary raise to employees working more than 5 hours from March 4, 2023 to March 10, 2023, first we are selecting all the employees who worked more than 5 hours from March 4, 2023, to March 10, 2023 by analyzing employee records and their corresponding payroll data, filtering out those who meet the specified working hours criteria.

SQL Query to update:

```
UPDATE V_11642773.employee e
SET e.annual_salary = e.annual_salary * 1.23
WHERE e.id IN (
   SELECT e.id
   FROM V_11642773.employee e
   JOIN V_11642773.employee_payroll ep ON e.id = ep.employee_id
   WHERE ep.start_date BETWEEN TIMESTAMP '2023-03-04 00:00:00' AND TIMESTAMP '2023-03-10
23:59:59'
   GROUP BY e.id
   HAVING SUM(ep.hours_worked) > 5
);
```

Screenshot-2:

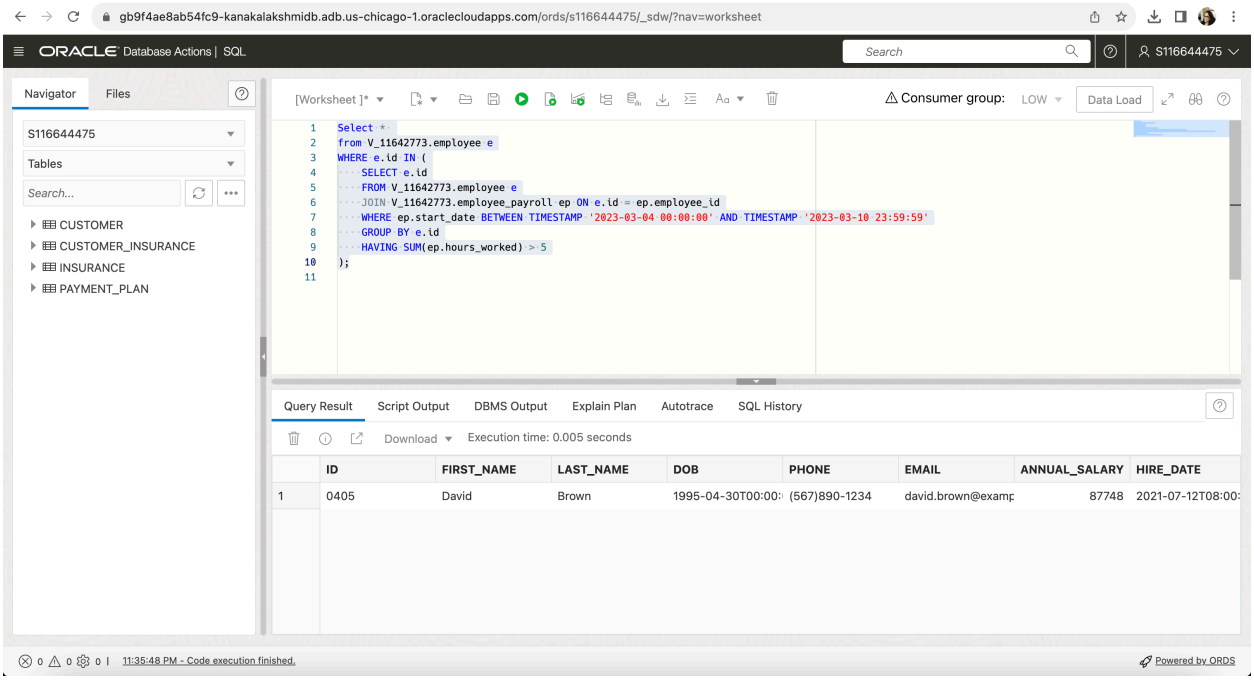


Explanation: This query is to update 23% salary raise to employees who worked more than 5 hours between March 4, 2023, and March 10, 2023. It operates on a database with two tables: "employee" and "employee_payroll." It first checks if an employee worked more than 5 hours between March 4, 2023, and March 10, 2023, by examining records in the "employee_payroll" table. Then, for each eligible employee, it multiplies their annual salary by 1.23 to grant them a 23% salary raise.

SQL Query After update:

```
Select *
from V_11642773.employee e
WHERE e.id IN (
   SELECT e.id
   FROM V_11642773.employee e
   JOIN V_11642773.employee_payroll ep ON e.id = ep.employee_id
   WHERE ep.start_date BETWEEN TIMESTAMP '2023-03-04 00:00:00' AND TIMESTAMP '2023-03-10
23:59:59'
   GROUP BY e.id
   HAVING SUM(ep.hours_worked) > 5
);
```

Screenshot-3:



Explanation: Here we are testing the previous update. This query fetches details about employees who have worked for more than 5 hours between March 4, 2023, and March 10, 2023. It's designed to provide a list of employees who have put in significant hours during that period. When compared the before update data and after update data, the salary got increased.

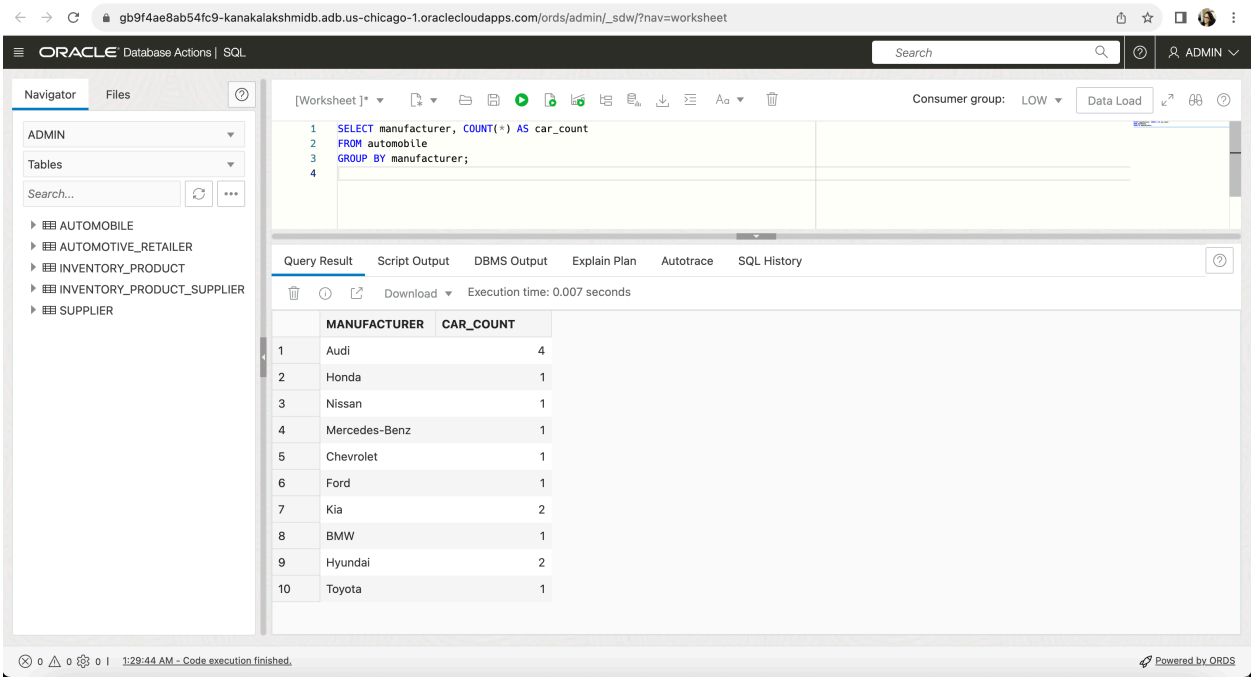
10 Select Queries:

Query-1: List manufacturer, count of each manufacturer produce.

SQL Query:

```
SELECT manufacturer, COUNT(*) AS car_count
FROM automobile
GROUP BY manufacturer;
```

Screenshot:

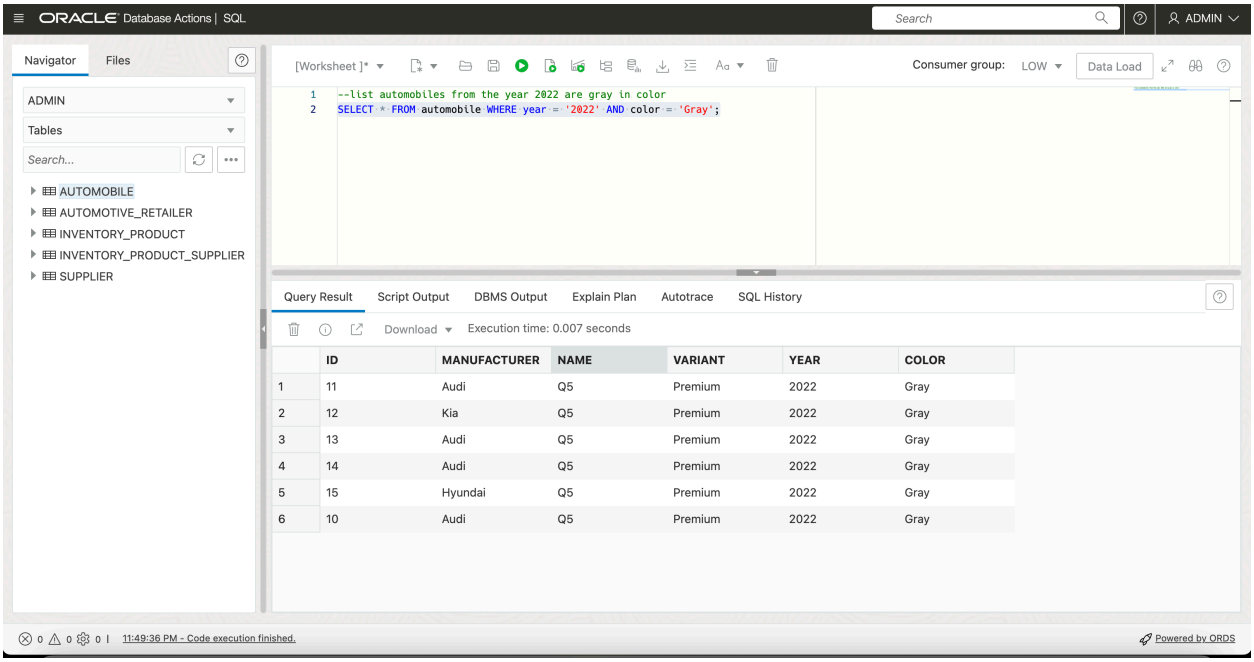


Explanation: The query retrieves data from a table called "automobile." It calculates the number of cars produced by each manufacturer and groups the results by the "manufacturer" column. The query uses the COUNT(*) function to count the number of rows (representing cars) for each distinct manufacturer. The results are presented with two columns: "manufacturer" and "car_count," showing the name of the manufacturer and the total count of cars they have produced, respectively.

Query-2: list automobiles from the year 2022 are gray in color

SQL Query: SQL Query - SELECT * FROM automobile WHERE year = '2022' AND color = 'Gray';

Screenshot:



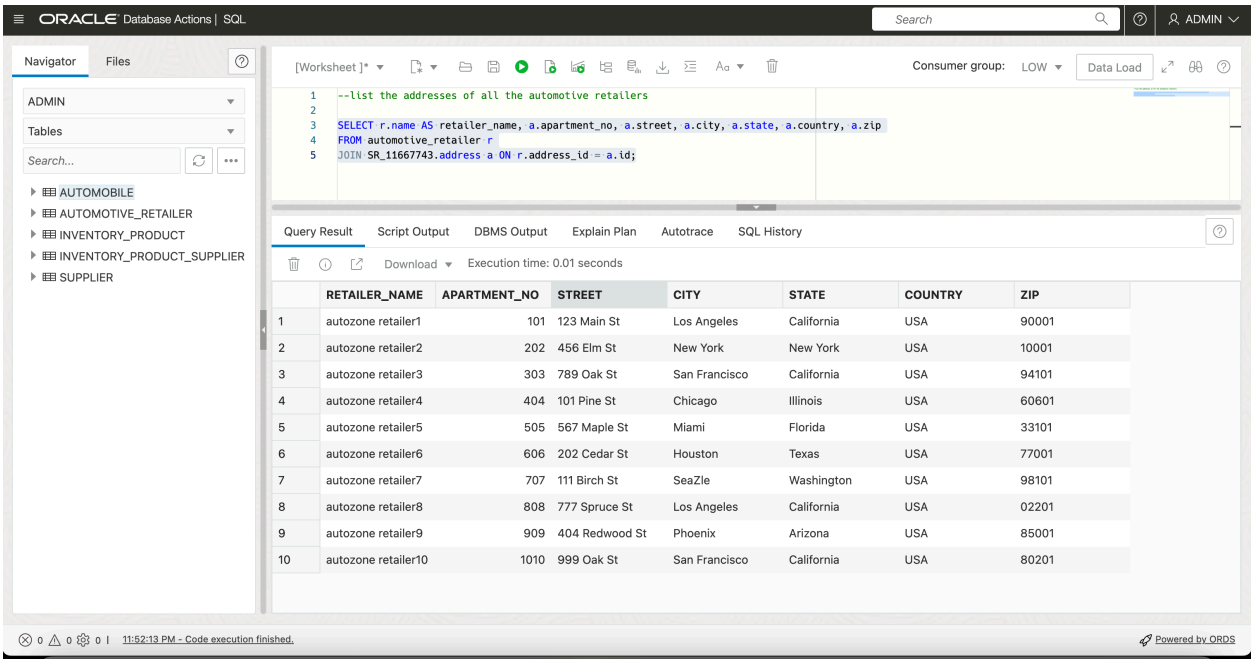
Explanation: The SQL query provided is designed to retrieve information about automobiles from the year 2022 that have a gray color. It uses the "SELECT" statement to specify the columns to be retrieved, the "FROM" clause to specify the table named "automobile," and the "WHERE" clause to filter the results based on two conditions: the year being 2022 and the color being gray.

Query-3: list the addresses of all the automotive retailers.

SQL Query:

```
SELECT r.name AS retailer_name, a.apartment_no, a.street, a.city, a.state, a.country, a.zip
FROM automotive_retailer r
JOIN SR_11667743.address a ON r.address_id = a.id;
```

Screenshot:



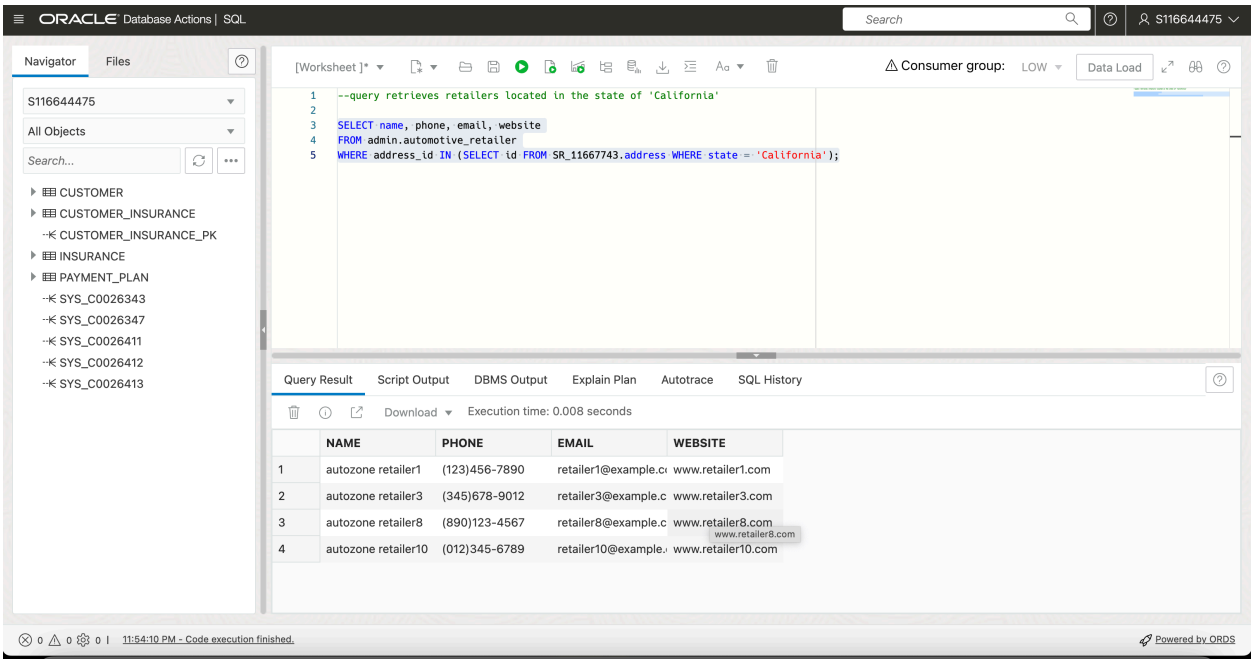
Explanation: The query is to obtain a list of automotive retailers and their respective addresses from the database. It accomplishes this by connecting the "automotive_retailer" table with the "address" table through a shared address ID. This way, we can compile a comprehensive list of automotive retailer names along with address details such as apartment number, street, city, state, country, and ZIP code for each retailer's location.

Query-4: Write a Query to retrieves retailers located in the state of 'California'.

SQL Query:

```
SELECT name, phone, email, website
FROM admin.automotive_retailer
WHERE address_id IN (SELECT id FROM SR_11667743.address WHERE state = 'California');
```

Screenshot:



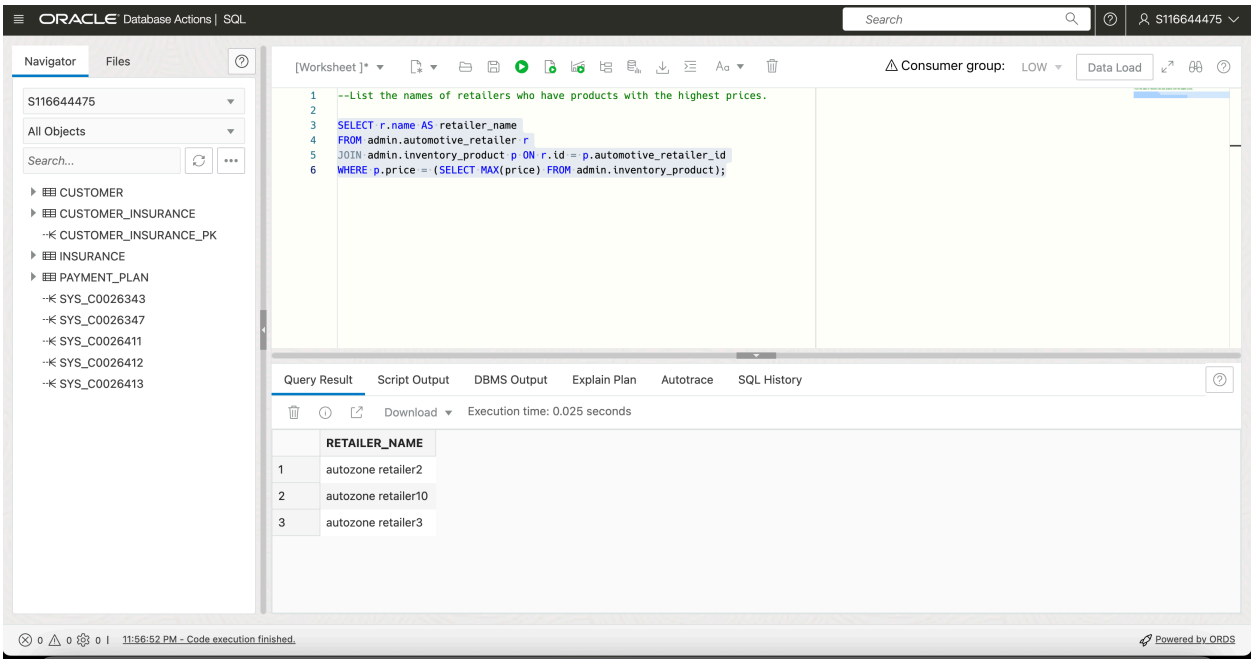
Explanation: This query fetches details of automotive retailers in California. It retrieves their names, phone numbers, emails, and websites. To identify retailers in California, it first finds the address IDs of those in the state from the 'address' table, and then it retrieves the relevant information from the 'automotive_retailer' table.

Query-5: List the names of retailers who have products with the highest prices.

SQL Query:

```
SELECT r.name AS retailer_name
FROM admin.automotive_retailer r
JOIN admin.inventory_product p ON r.id = p.automotive_retailer_id
WHERE p.price = (SELECT MAX(price) FROM admin.inventory_product);
```

Screenshot:



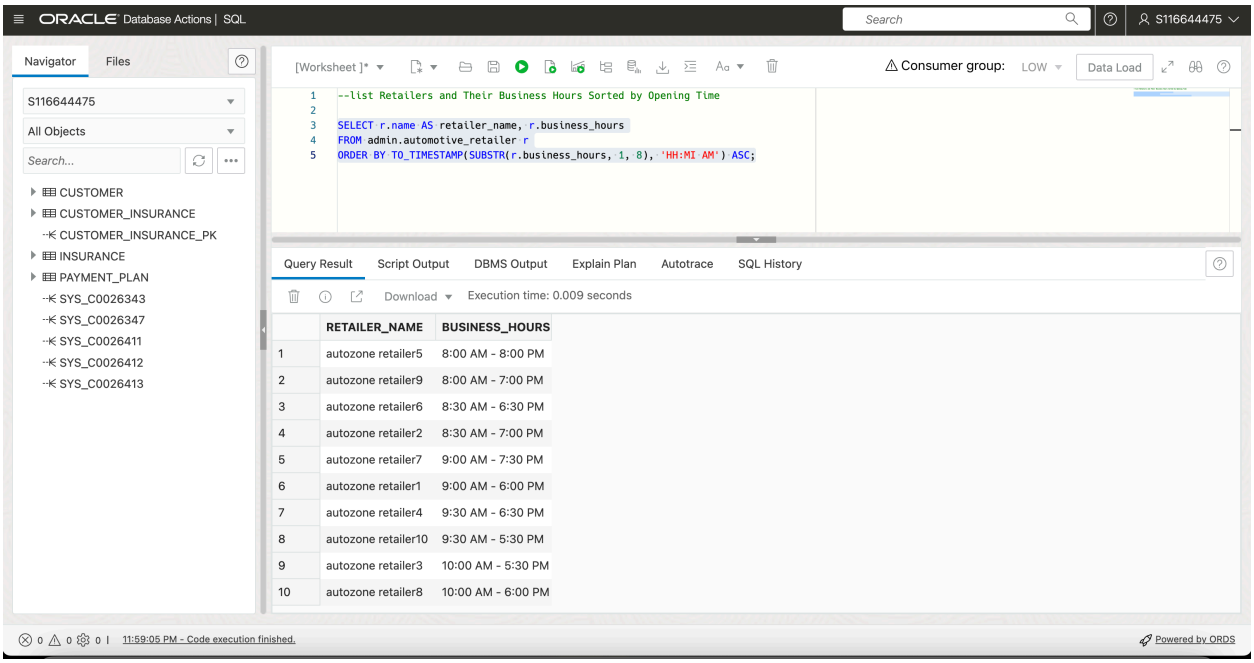
Explanation: The query identifies and lists the names of retailers who sell products with the highest prices within the "automotive" category. It achieves this by linking retailer and product information and then selecting the retailers whose product prices are equal to the maximum price found in the inventory. This helps to identify which retailers offer the most expensive products in the automotive category.

Query-6: list Retailers and Their Business Hours Sorted.

SQL Query:

```
SELECT r.name AS retailer_name, r.business_hours
FROM admin.automotive_retailer r
ORDER BY TO_TIMESTAMP(SUBSTR(r.business_hours, 1, 8), 'HH:MI AM') ASC;
```

Screenshot:



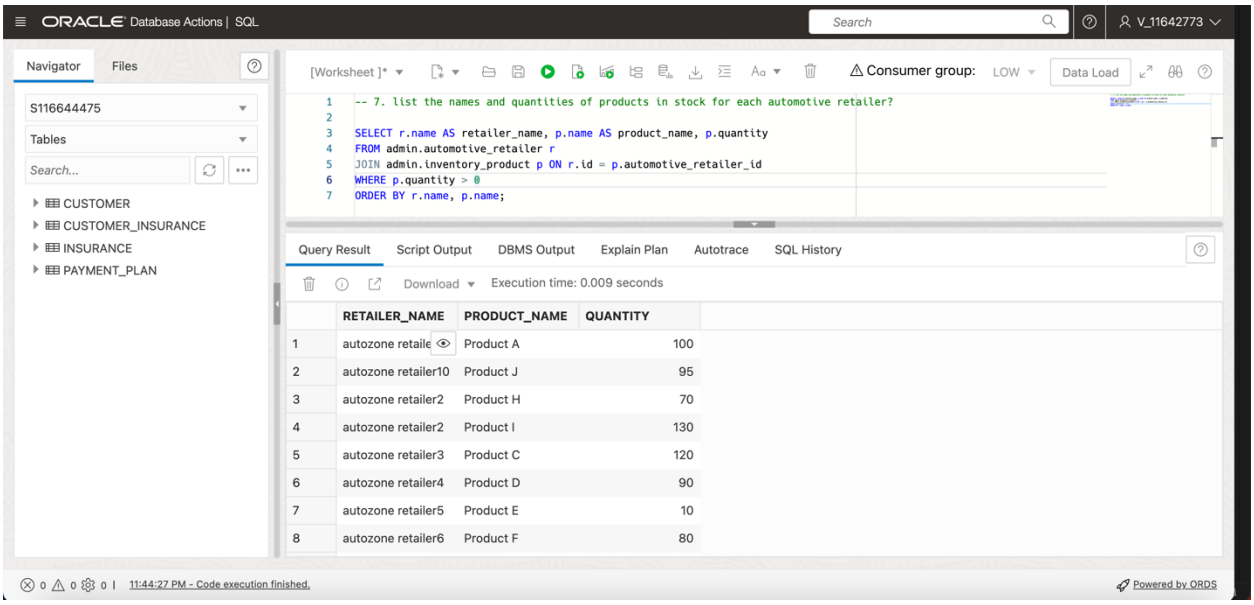
Explanation: The SQL query retrieves data from a table named "automotive_retailer" in a database schema named "admin." It selects two columns, "name" (aliased as "retailer_name") and "business_hours" from the table. The query then orders the results in ascending order based on the business hours after converting them to a timestamp format.

Query-7: list the names and quantities of products in stock for each automotive retailer

SQL Query:

```
SELECT r.name AS retailer_name, p.name AS product_name, p.quantity
FROM admin.automotive_retailer r
JOIN admin.inventory_product p ON r.id = p.automotive_retailer_id
WHERE p.quantity > 0
ORDER BY r.name, p.name;
```

Screenshot:



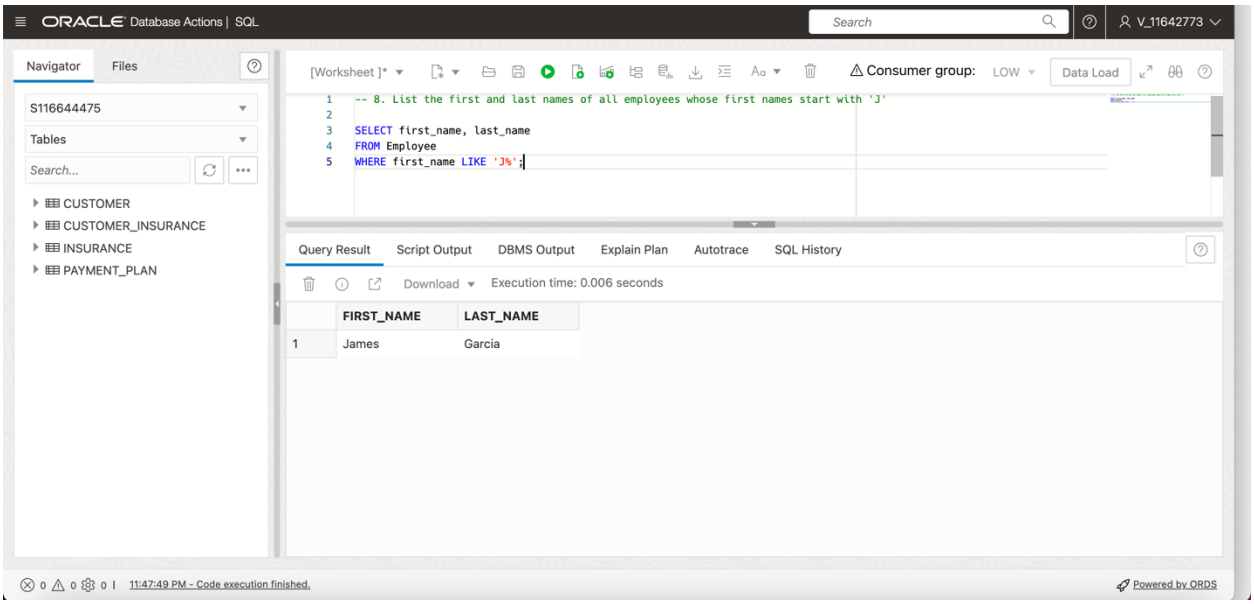
Explanation: This query provides a clear overview of the available products for each automotive retailer. It lists the retailer names along with the names of products they have in stock, along with the respective quantities. So, It filters the results to only include rows where the product quantity is greater than 0 and orders the results first by retailer name and then by product name. This information is helpful for inventory management and allows retailers to monitor their stock levels effectively.

Query-8: List the first and last names of all employees whose first names start with 'J'

SQL Query:

```
SELECT first_name, last_name
FROM Employee
WHERE first_name LIKE 'J%';
```

Screenshot:



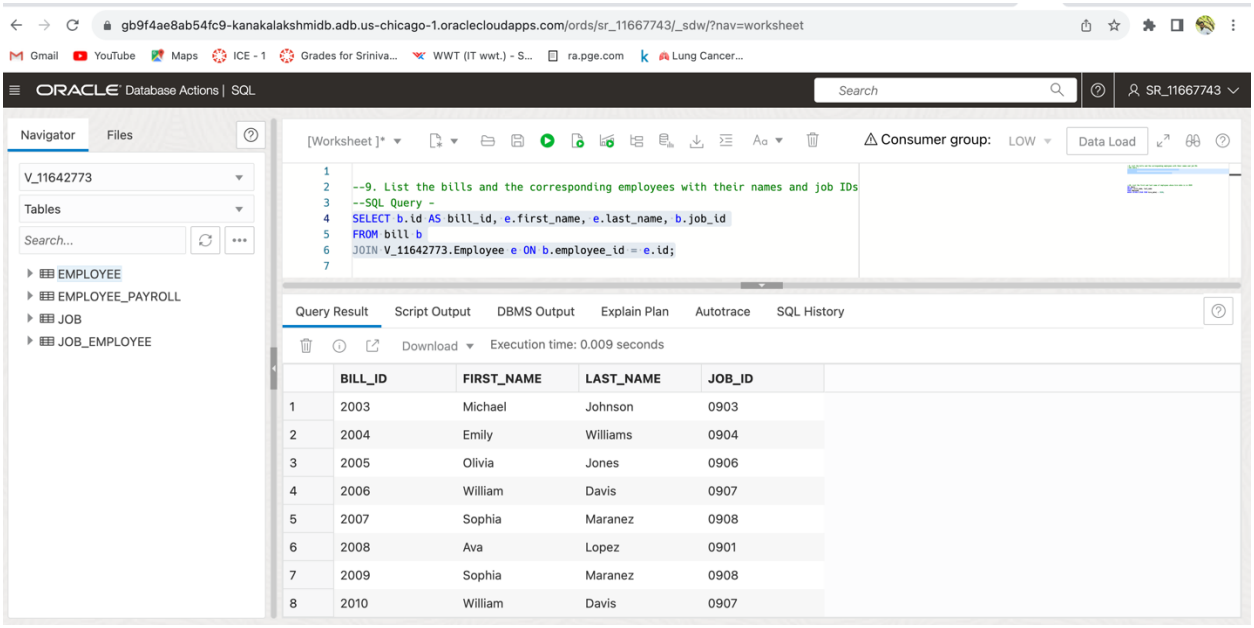
Explanation: This query is like a filter for employees with names that start with 'J.' It finds and lists their first and last names from the Employee table. It's a handy way to quickly identify all the employees whose first names begin with 'J'. We can identify all employees alphabetically by changing the character.

Query-9: List the bills and the corresponding employees with their names and job IDs

SQL Query:

```
SELECT b.id AS bill_id, e.first_name, e.last_name, b.job_id
FROM bill b
JOIN V_11642773.Employee e ON b.employee_id = e.id;
```

Screenshot:



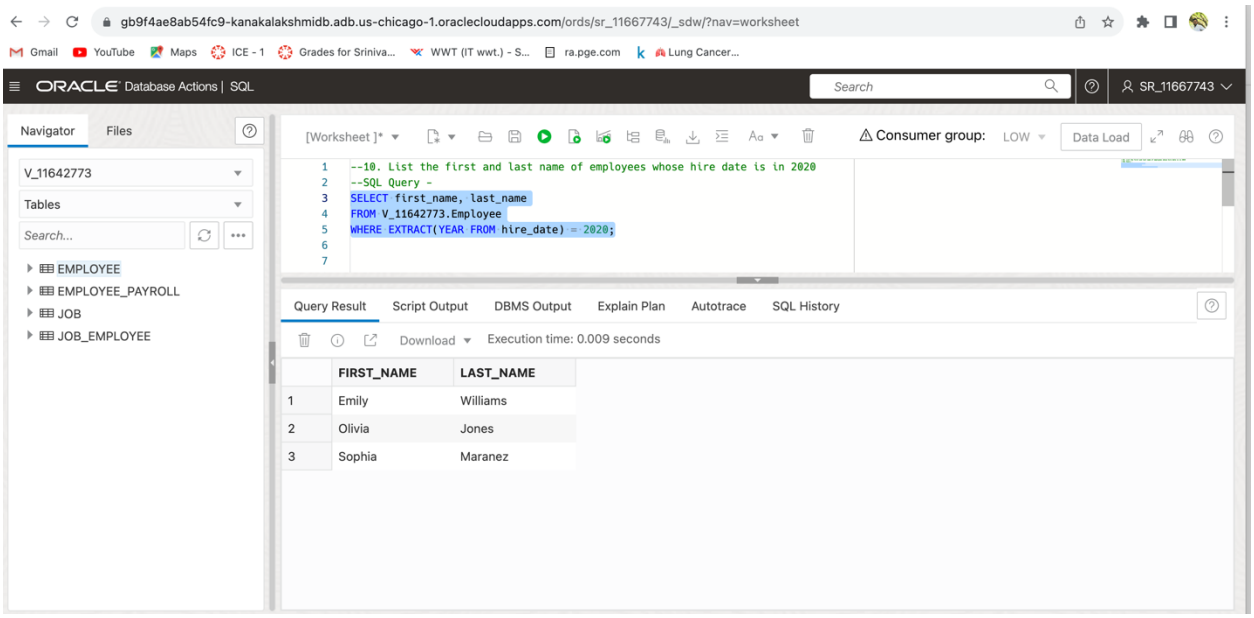
Explanation: The query combines information from two tables, bills, and employees. It links bills to the employees responsible for them by matching their employee IDs. The result includes a list of bills with their IDs, the first and last names of the employees handling them, and their respective job IDs. This query helps provide insight into which employees are associated with specific bills.

Query-10: List the first and last name of employees whose hire date is in 2020

SQL Query:

```
SELECT first_name, last_name
FROM V_11642773.Employee
WHERE EXTRACT(YEAR FROM hire_date) = 2020;
```

Screenshot:



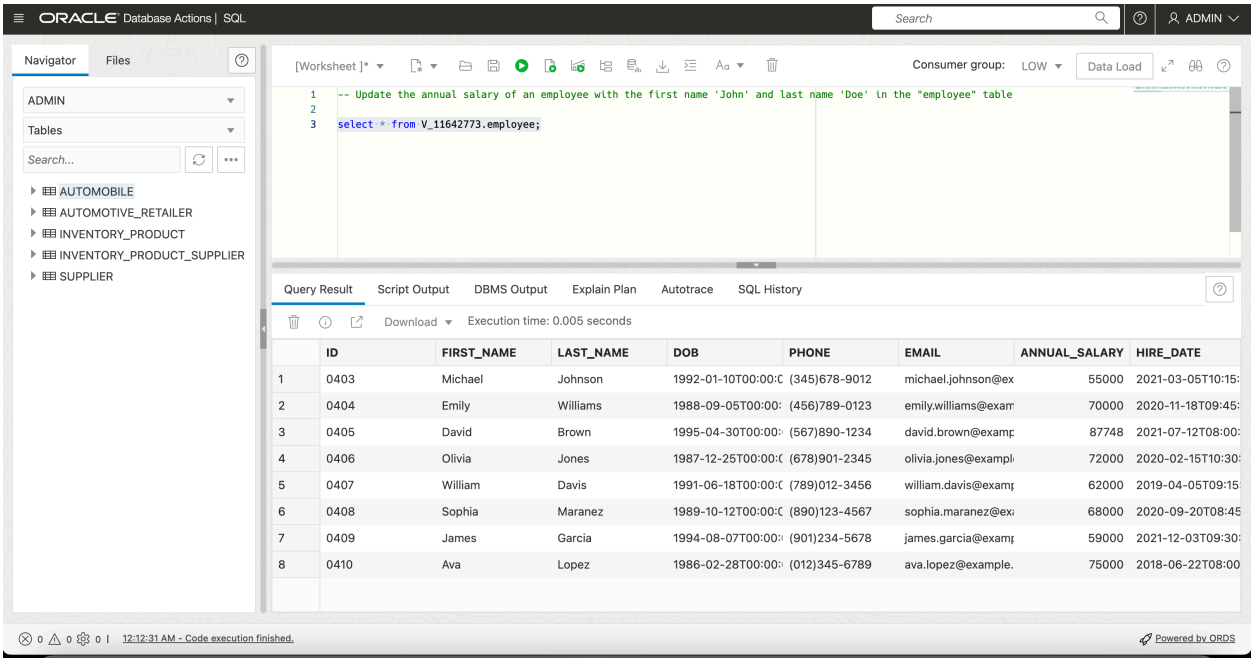
Explanation: This query selects the first name and last name of employees from a table named "Employee." It filters the results based on the hire date of employees. Specifically, it retrieves employees whose hire date is in the year 2020. It does this by looking at the hire date for each employee and filtering out those who joined the company during that specific year. The query utilizes the EXTRACT function to extract the year from the hire date and then compares it to the value 2020.

7 Update Queries:

Query-1: Update the annual salary of an employee to 75000 with the first name 'John' and last name 'Doe' in the "employee" table.

SQL Query Before update: Select Query Before Update: select * from V_11642773.employee;

Screenshot:

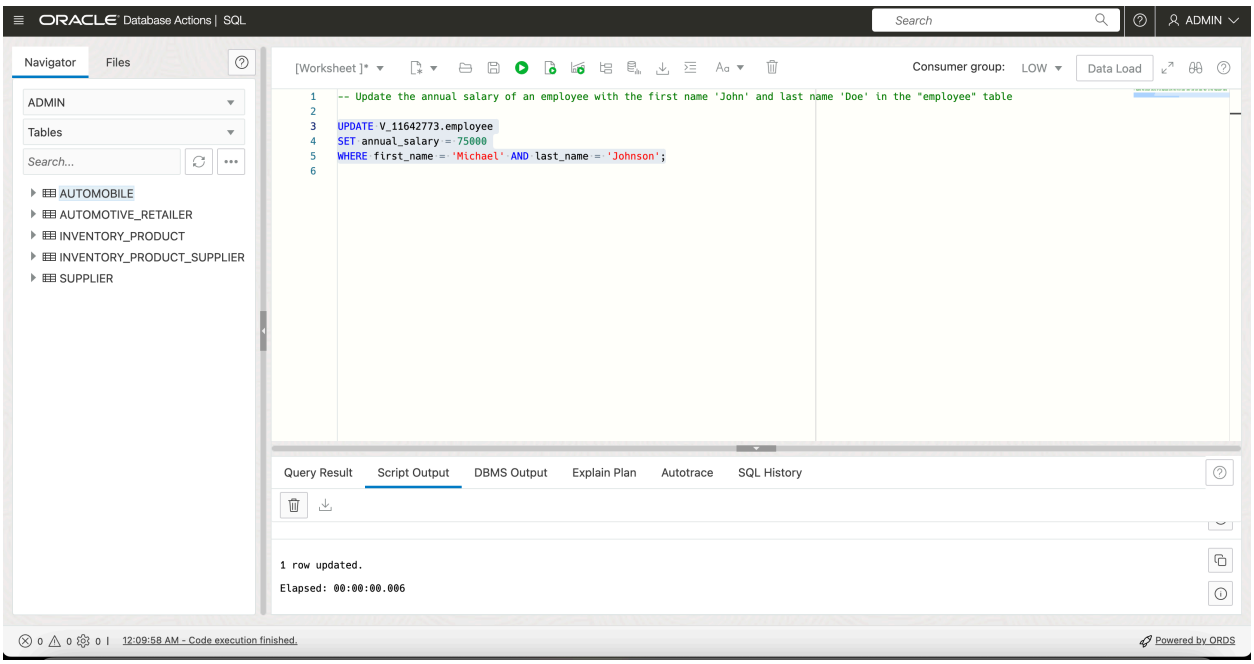


Explanation: Before updating the table, this is the data in employees table.

SQL Query to Update:

```
UPDATE V_11642773.employee
SET annual_salary = 75000
WHERE first_name = 'Michael' AND last_name = 'Johnson';
```

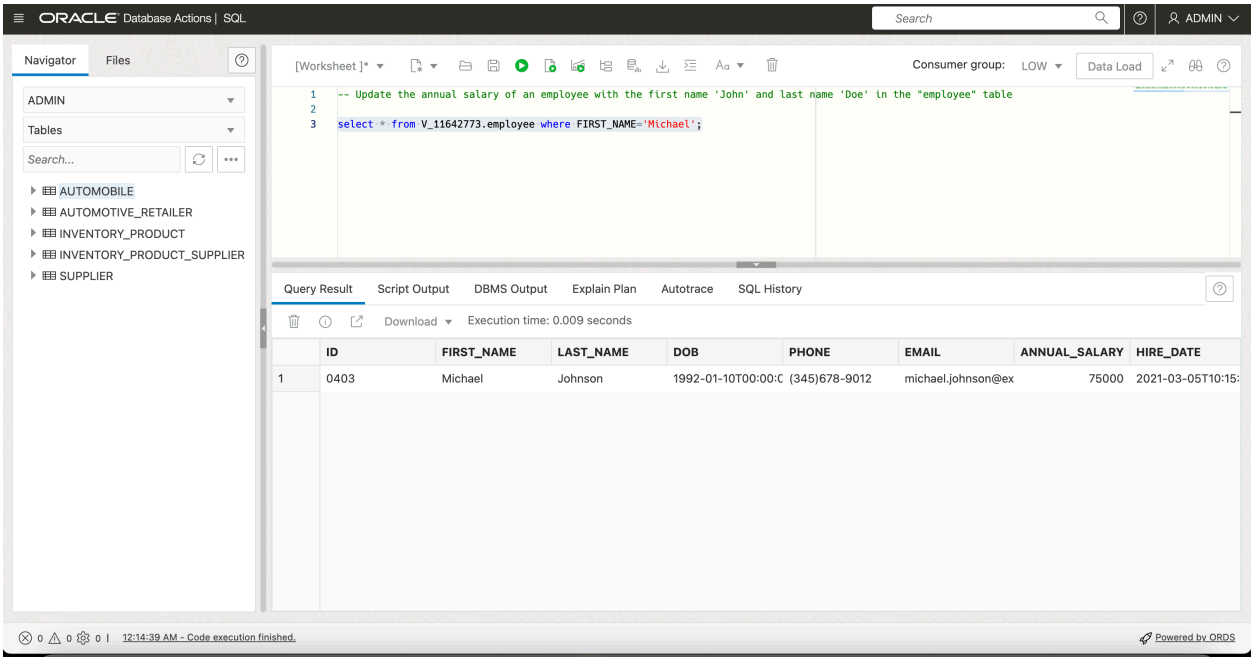
Screenshot:



Explanation: Through this SQL query, we are changing the annual salary of an employee named Michael Johnson to $75,000 in a database called "V_11642773." It's a way to update the salary information for a specific person in the employee database.

SQL Query After Update: select * from V_11642773.employee where FIRST_NAME=’Michael’;

Screenshot:

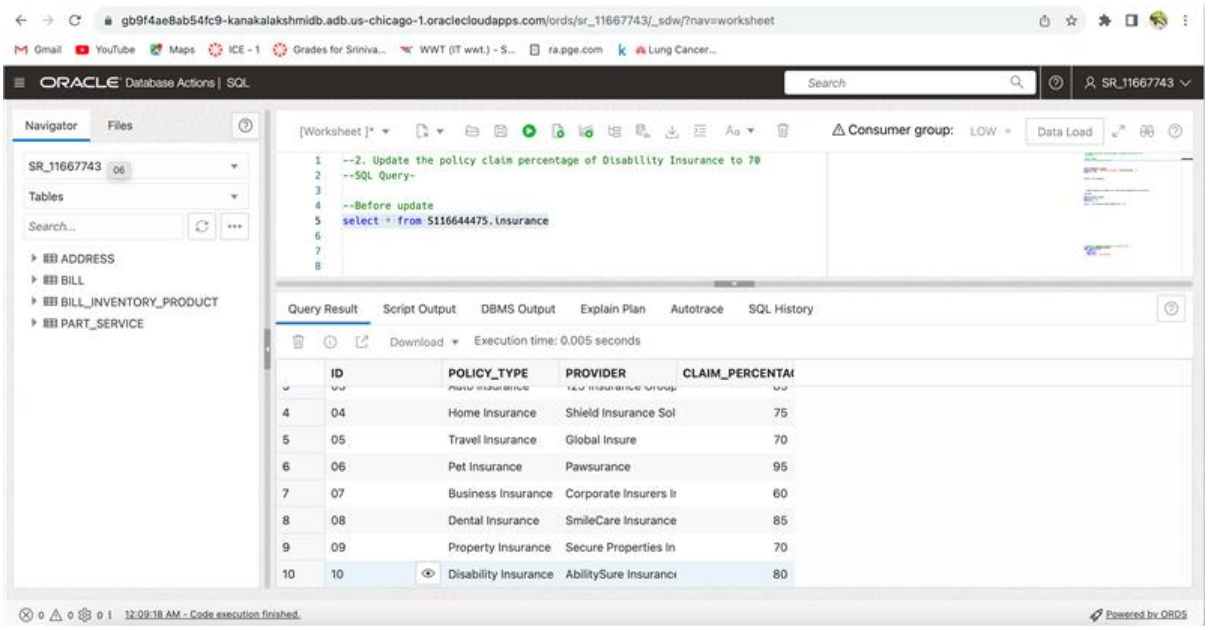


Explanation: Checking whether the data is updated or not after performing update.

Query-2: Update the policy claim percentage of Disability Insurance to 70

SQL Query Before Update: select * from S116644475.insurance

Screenshot:

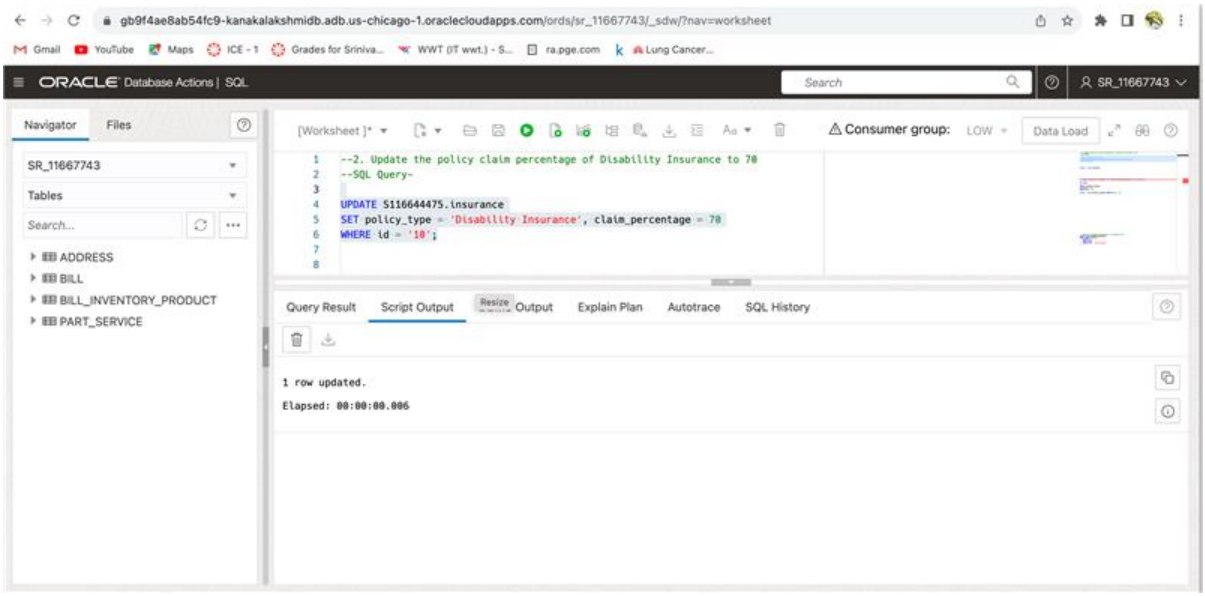


Explanation: This query is to display the information in insurance table. This data is before updating the table.

SQL Query to Update:

```
UPDATE S116644475.insurance
SET policy_type = 'Disability Insurance', claim_percentage = 70
WHERE id = '10';
```

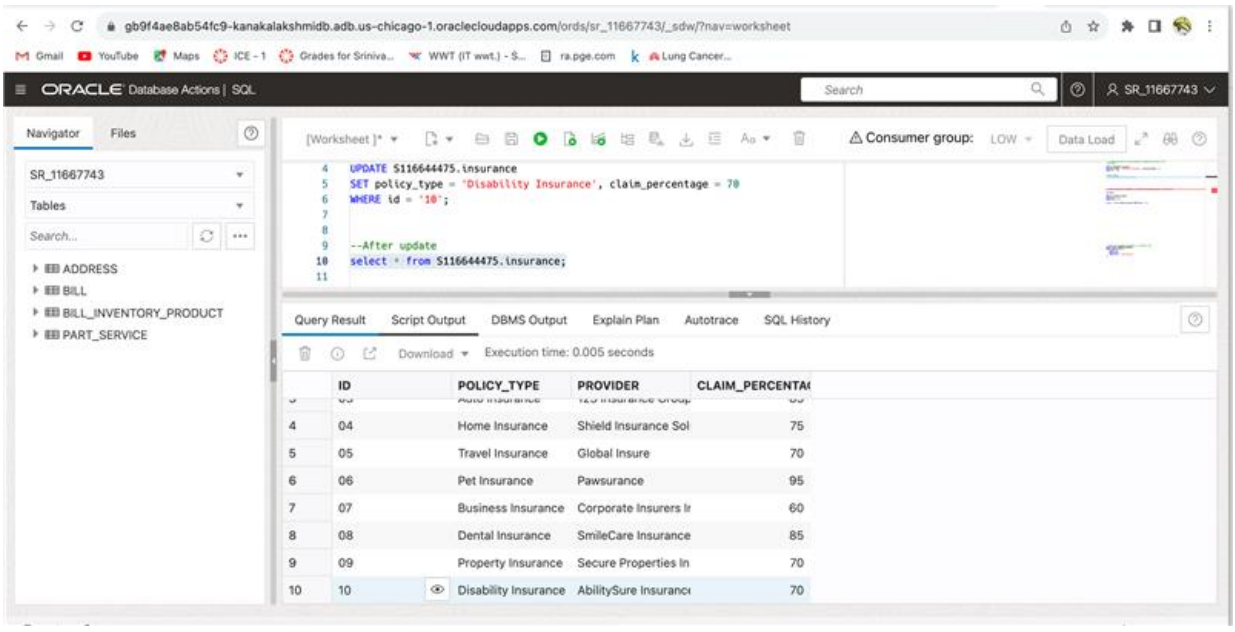
Screenshot:



Explanation: This query updates an entry in the insurance database. It changes the insurance policy type to 'Disability Insurance' and adjusts the claim percentage to 70% for the policy with ID 10. This could be important for managing and customizing insurance policy details in a database.

SQL Query After Update: select * from S116644475.insurance;

Screenshot:

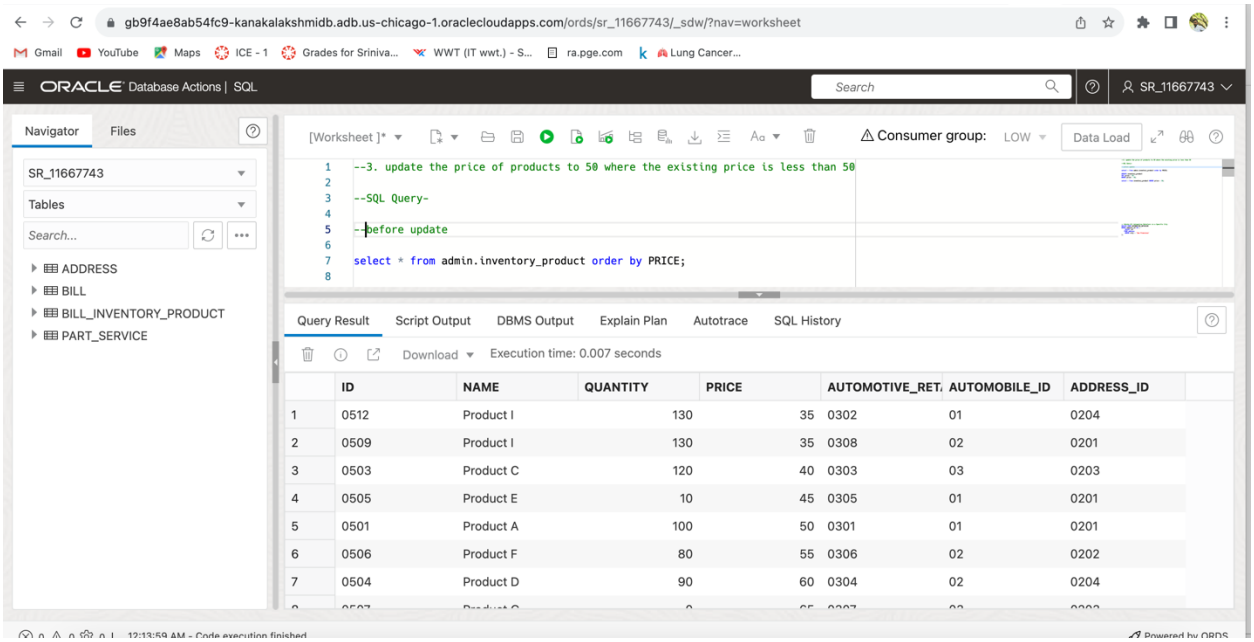


Explanation: The data in insurance table after update.

Query-3: update the price of product to 50 where the existing price of products is less than 50.

SQL Query Before Update: select * from admin.inventory_product order by PRICE;

Screenshot:

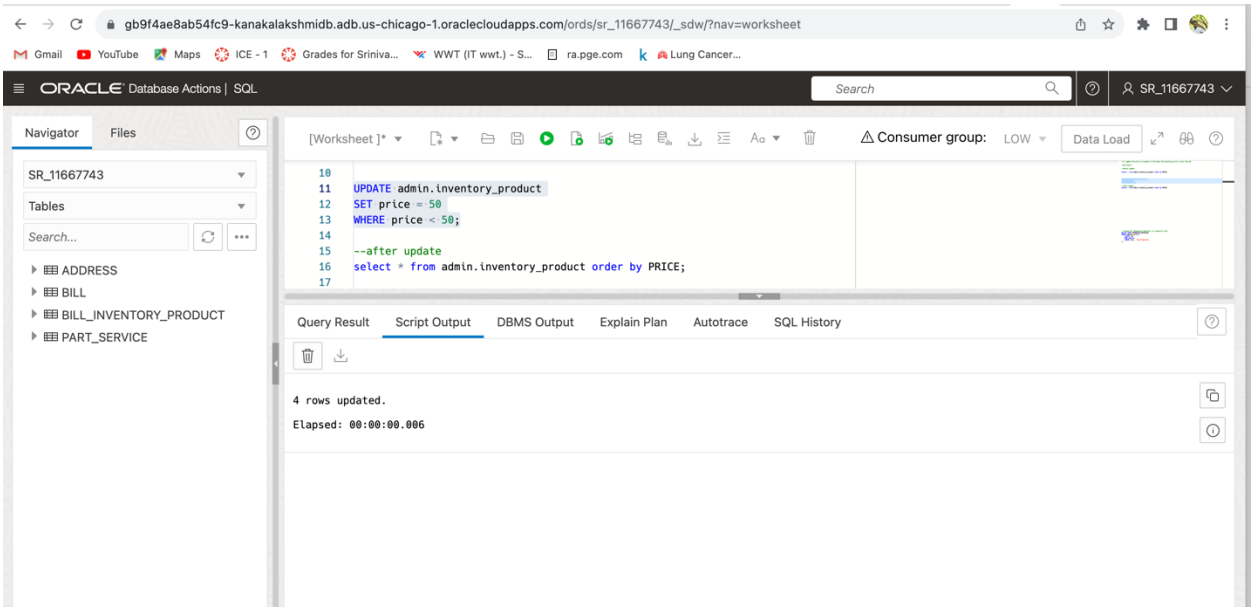


Explanation: The data in inventory product table and price table before updating.

SQL Query to Update:

```
UPDATE admin.inventory_product
SET price = 50
WHERE price < 50;
```

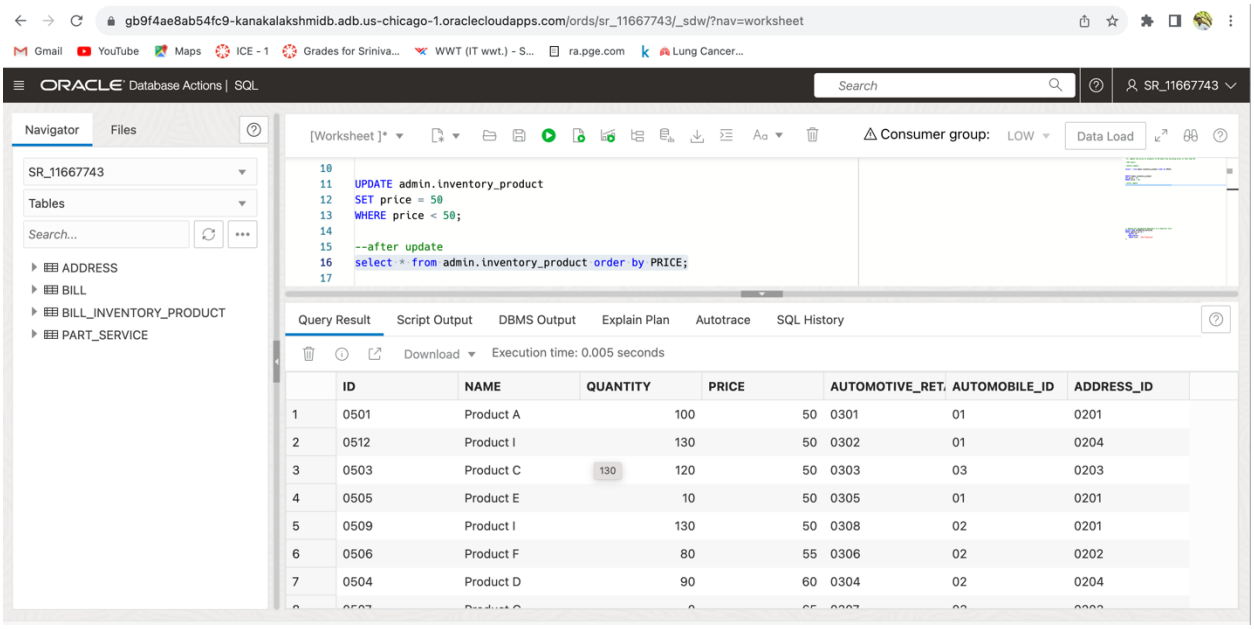
Screenshot:



Explanation: This SQL query is like instructing a database to go through a table “inventory_product" and find all the products with a price less than 50. For each of those products, it changes their price to 50. In simple terms, it's updating the prices of products that cost less than 50 to make them all cost 50.

SQL Query After update: select * from admin.inventory_product order by PRICE;

Screenshot:

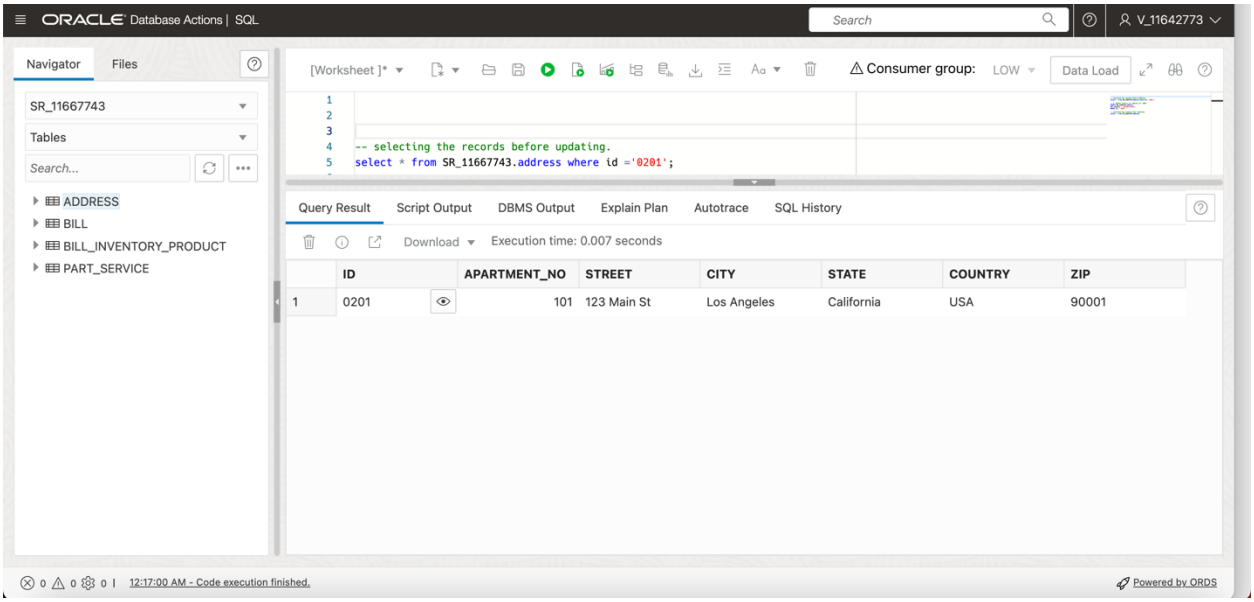


Explanation: After updating, no product price is less than 50.

Query-4: Update Country for a specific Address Id

SQL Query Before update: select * from SR_11667743.address where COUNTRY='USA';

Screenshot:

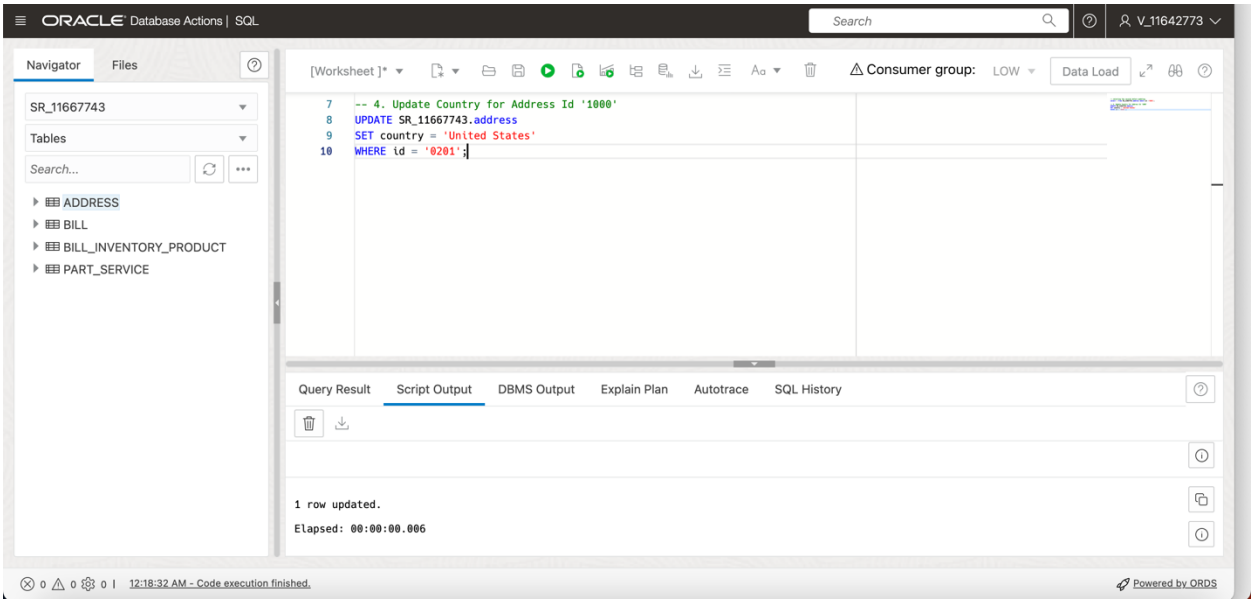


Explanation: This query is to display the rows where the country is listed as 'USA.' It's like asking for a list of addresses specifically in the United States.

SQL Query to Update:

```
UPDATE SR_11667743.address
SET country = 'United States'
WHERE id = '0201';
```

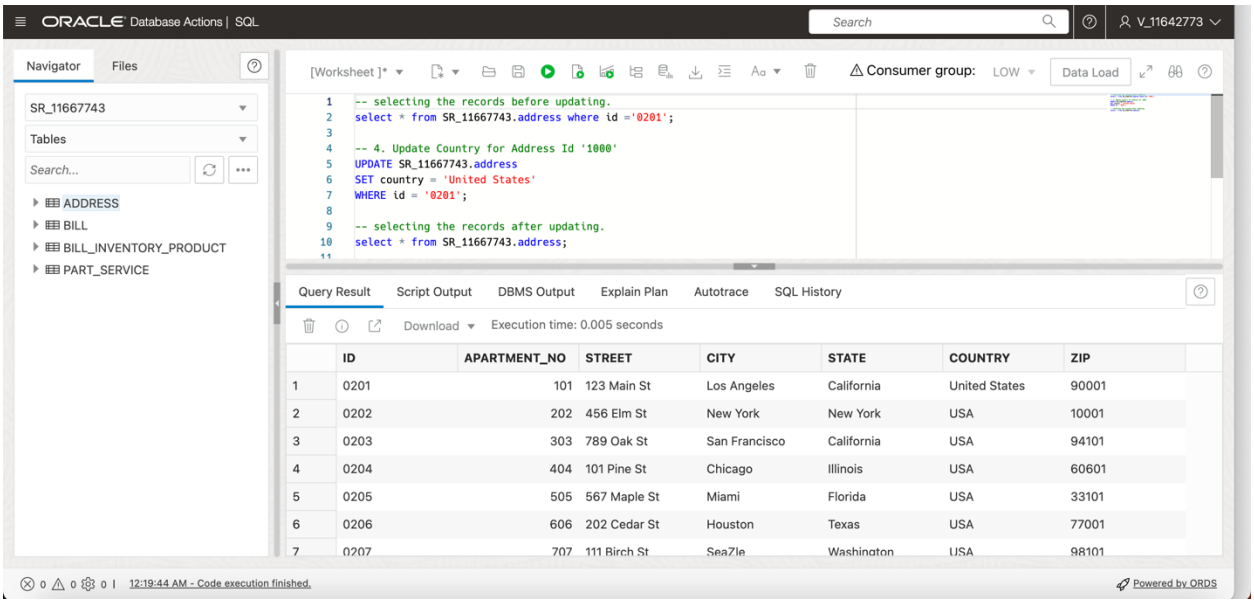
Screenshot:



Explanation: This SQL query updates a database table called "address" with a specific ID of '0201' and changes the country value to 'United States.' It essentially edits the country information for a particular address entry in the table.

SQL Query After update: select * from SR_11667743.address;

Screenshot:



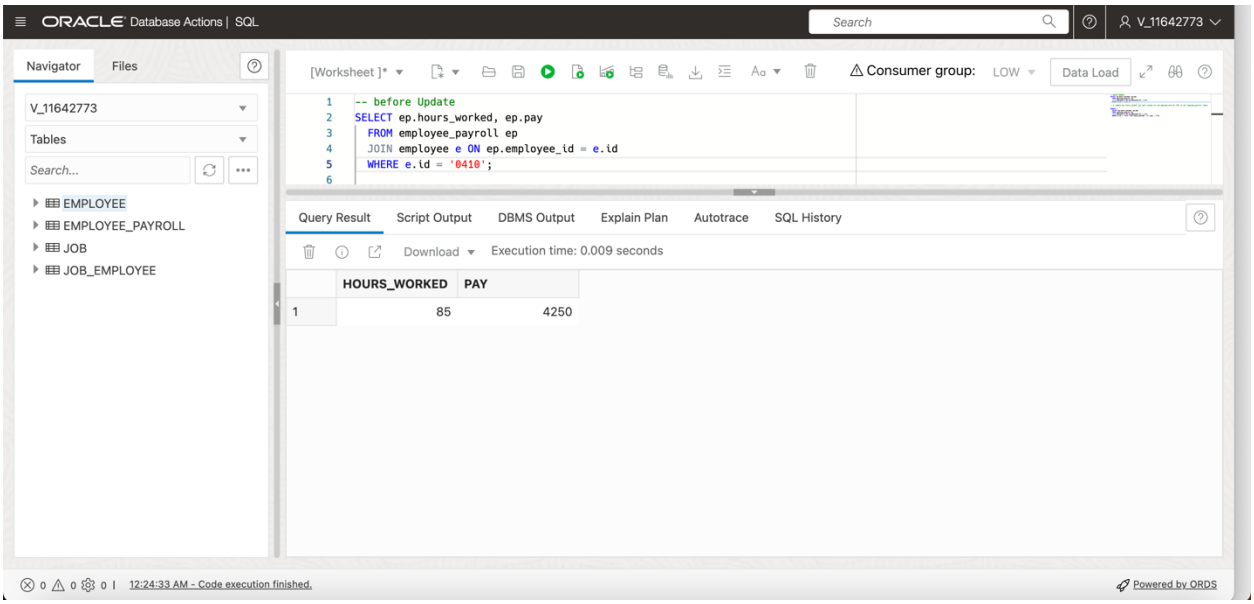
Explanation: The country of id 201 is updated to United States.

Query-5: update the "hours_worked" and "pay" columns for the employee for a specific ID in the "employee_payroll" table.

SQL Query Before update:

```
SELECT ep.hours_worked, ep.pay
  FROM employee_payroll ep
  JOIN employee e ON ep.employee_id = e.id
  WHERE e.id = '0410';
```

Screenshot:

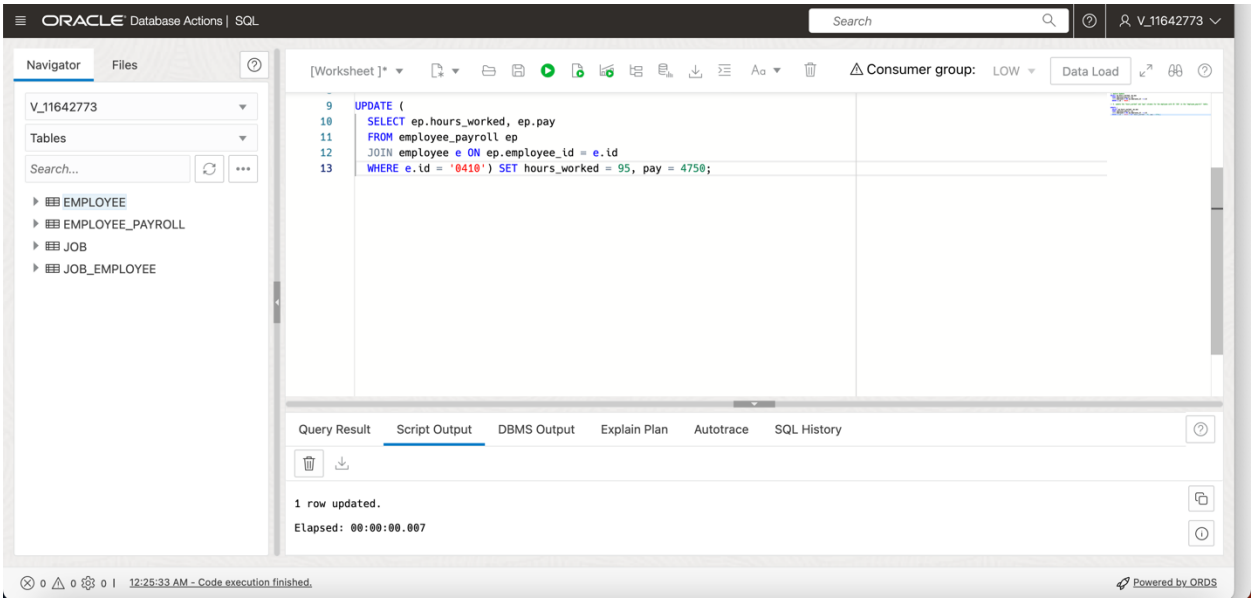


Explanation: This SQL query fetches the hours worked and pay details for an employee identified by the ID '0410' from a database that manages employee payroll. It combines data from two tables, 'employee_payroll' and 'employee,' linking them by the employee ID, and then narrows down the results to show information about the specific employee with that ID. This is performed to check the data before updating.

SQL Query to Update:

```
UPDATE(
SELECT ep.hours_worked, ep.pay
  FROM employee_payroll ep
  JOIN employee e ON ep.employee_id = e.id
  WHERE e.id = '0410') SET hours_worked = 95, pay = 4750;
```

Screenshot:

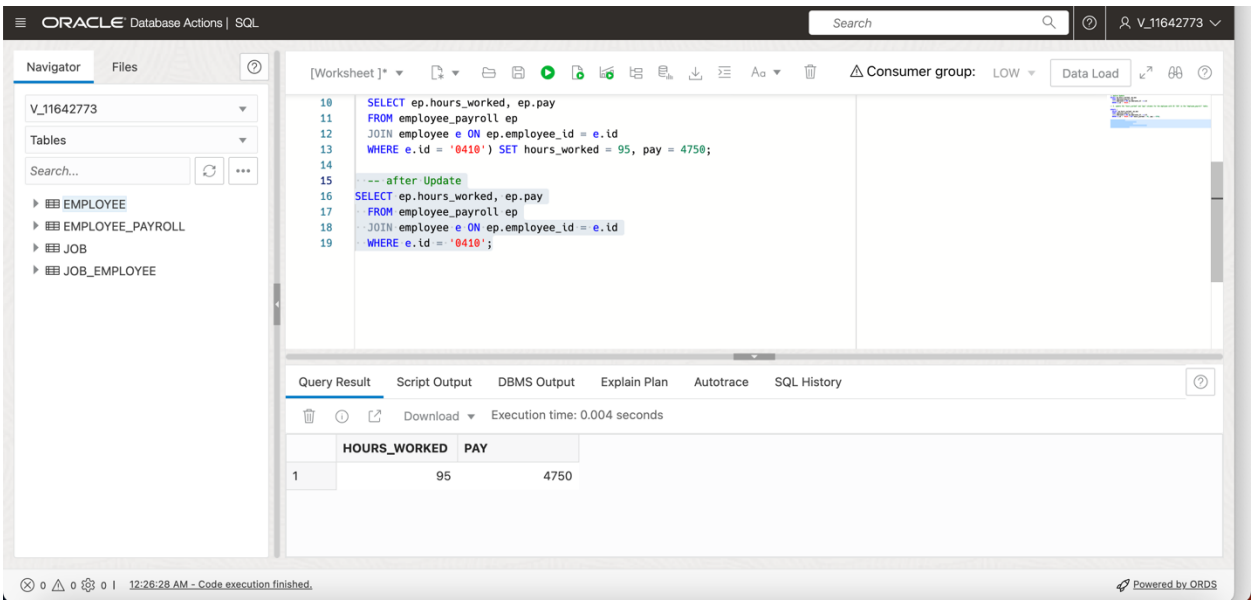


Explanation: The query is meant to change the recorded working hours and pay for an employee with the employee ID '0410' in the database. It finds this employee's data in the "employee_payroll" table, and then updates their "hours_worked" to 95 and "pay" to 4750. Essentially, it's adjusting the payroll information for that specific employee.

SQL Query After update:

```
SELECT ep.hours_worked, ep.pay
FROM employee_payroll ep
JOIN employee e ON ep.employee_id = e.id
WHERE e.id = '0410';
```

Screenshot:

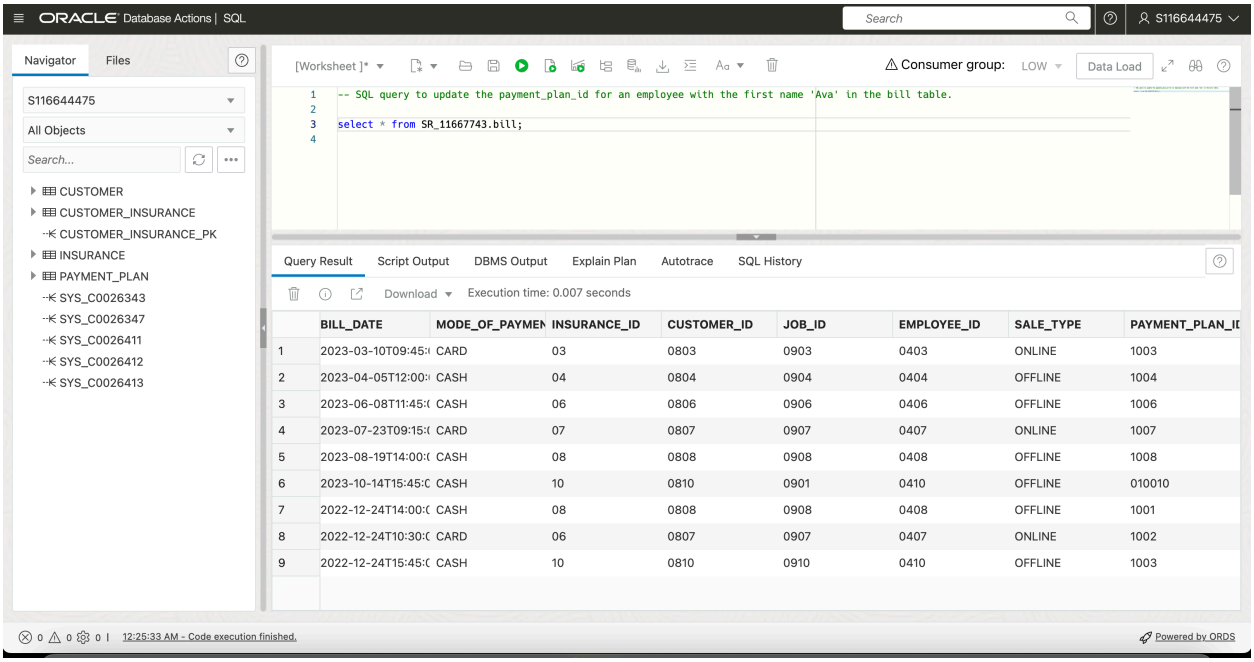


Explanation: This is the updated data.

Query-6: SQL query to update the payment_plan_id for an employee with the first name 'Ava' in the bill table.

SQL Query Before update: select * from SR_11667743.bill;

Screenshot:

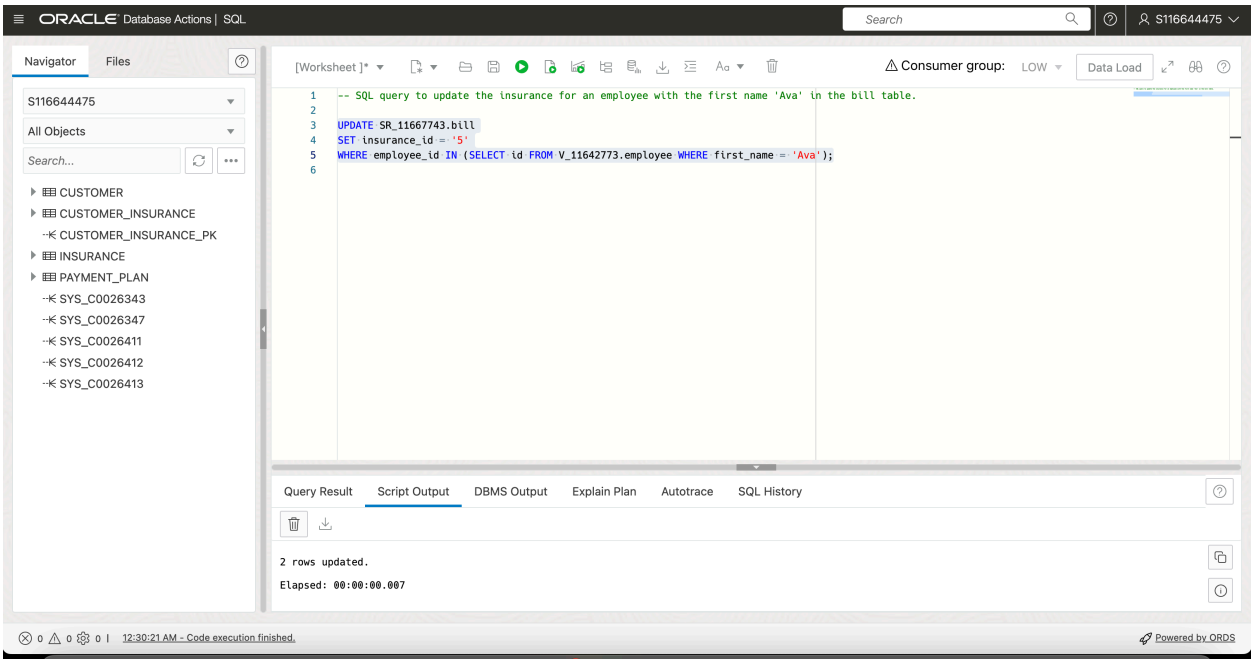


Explanation: The information of the “bill” table before updating.

SQL Query to Update:

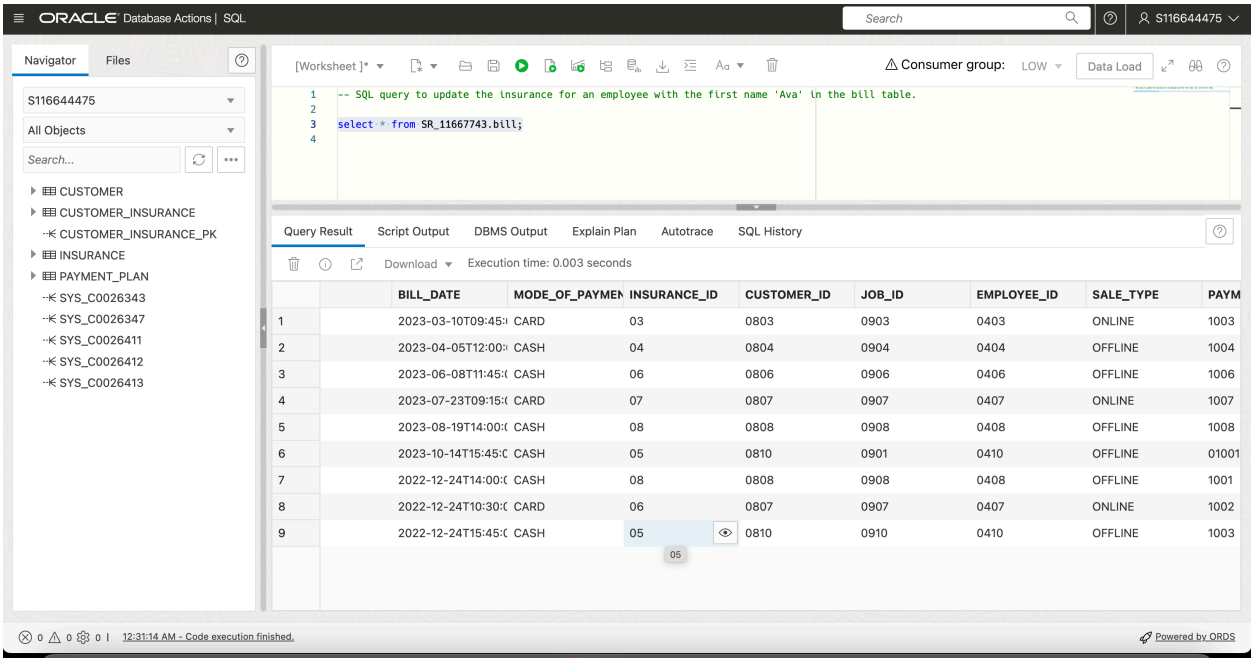
UPDATE SR_11667743.bill
SET insurance_id=’5’
WHERE employee_id IN(SELECT id FROM V_11642773.employee WHERE first_name = ‘Ava’);

Screenshot:



Explanation: This query is trying to change the "insurance_id" for certain entries in the "SR_11667743.bill" table. It's supposed to update the insurance ID to '5' for employees with the first name 'Ava.' However, it has some syntax errors and might not work as intended due to missing operators in the WHERE clause.

Screenshot After update:

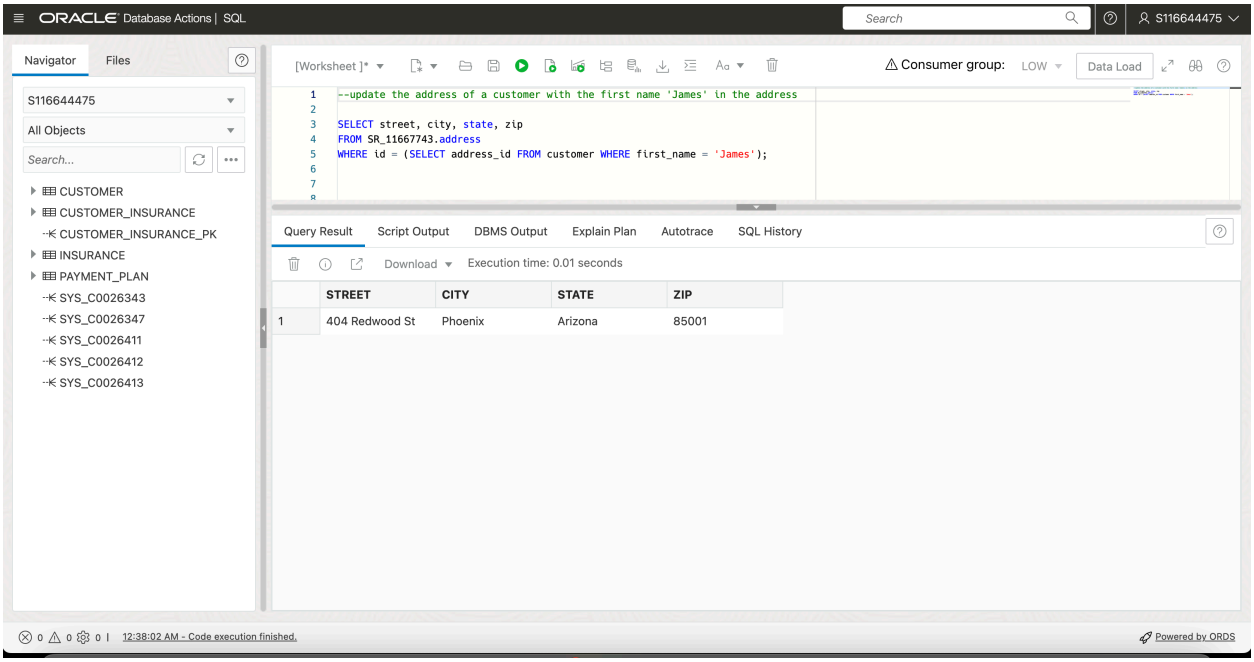


Query-7: update the address of a customer with the first name ‘James’ in the address

SQL Query Before update:

```
SELECT street, city, state, zip
FROM SR_11667743.address
WHERE id = (SELECT address_id FROM customer WHERE first_name = 'James');
```

Screenshot:

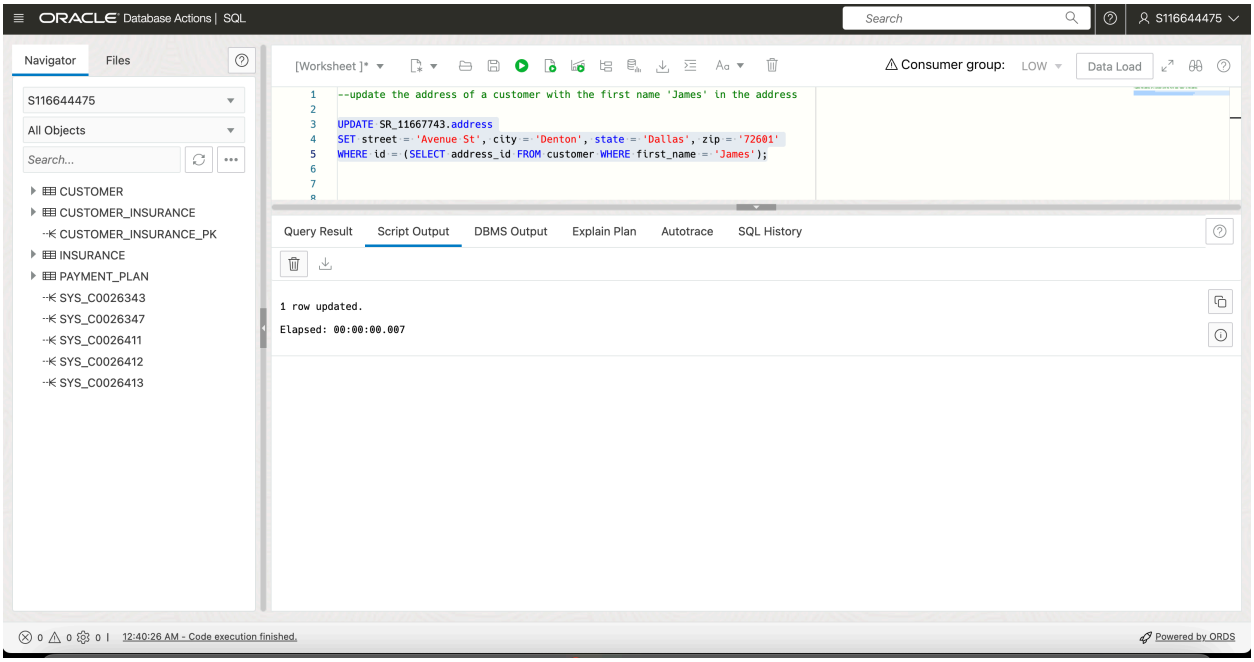


Explanation: This SQL query fetches the street, city, state, and zip code of the address for a customer named James. It first finds James's customer ID and then uses that ID to get his address information from the database. This is performed before updation.

SQL Query to Update:

```
UPDATE SR_11667743.address
SET street = 'Avenue St', city = 'Denton', state = 'Dallas', zip = '72601'
WHERE id = (SELECT address_id FROM customer WHERE first_name = 'James');
```

Screenshot:

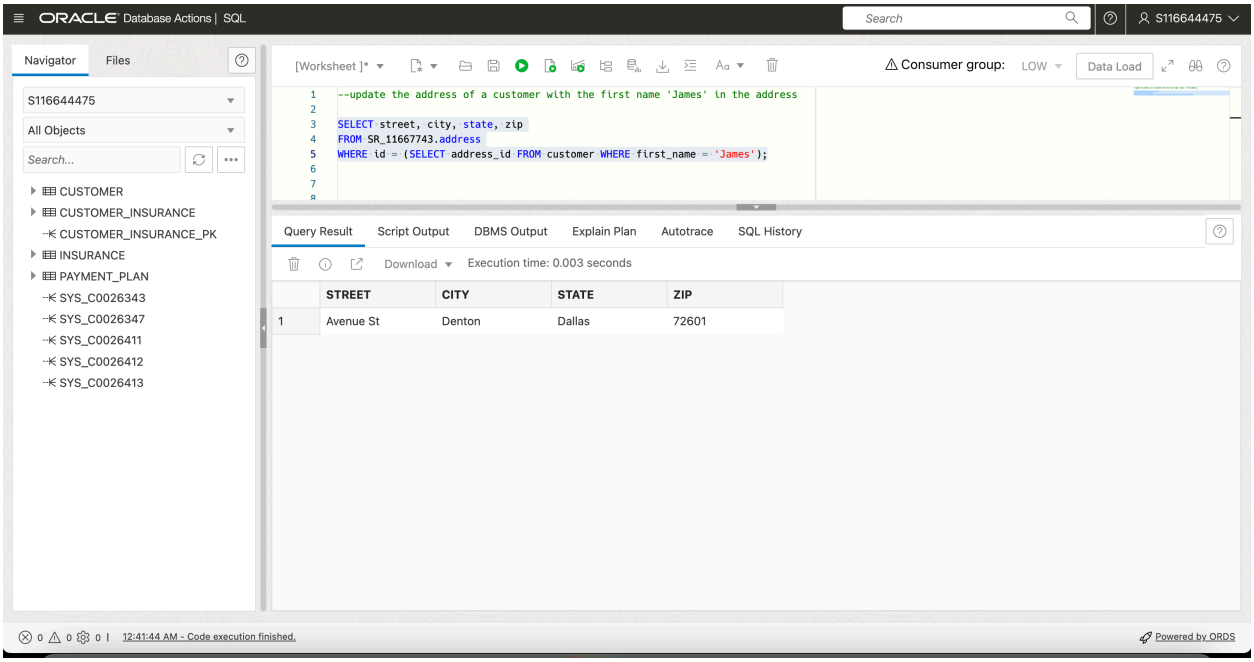


Explanation: This SQL query updates the address information for a customer named James. It changes the street to 'Avenue St', the city to 'Denton', the state to 'Dallas', and the ZIP code to '72601' for James' address.

SQL Query After Update:

```
SELECT street, city, state, zip
FROM SR_11667743.address
WHERE id = (SELECT address_id FROM customer WHERE first_name = 'James');
```

Screenshot:



Explanation: The same select statement is performed as before updating to check if the information is updated or not.

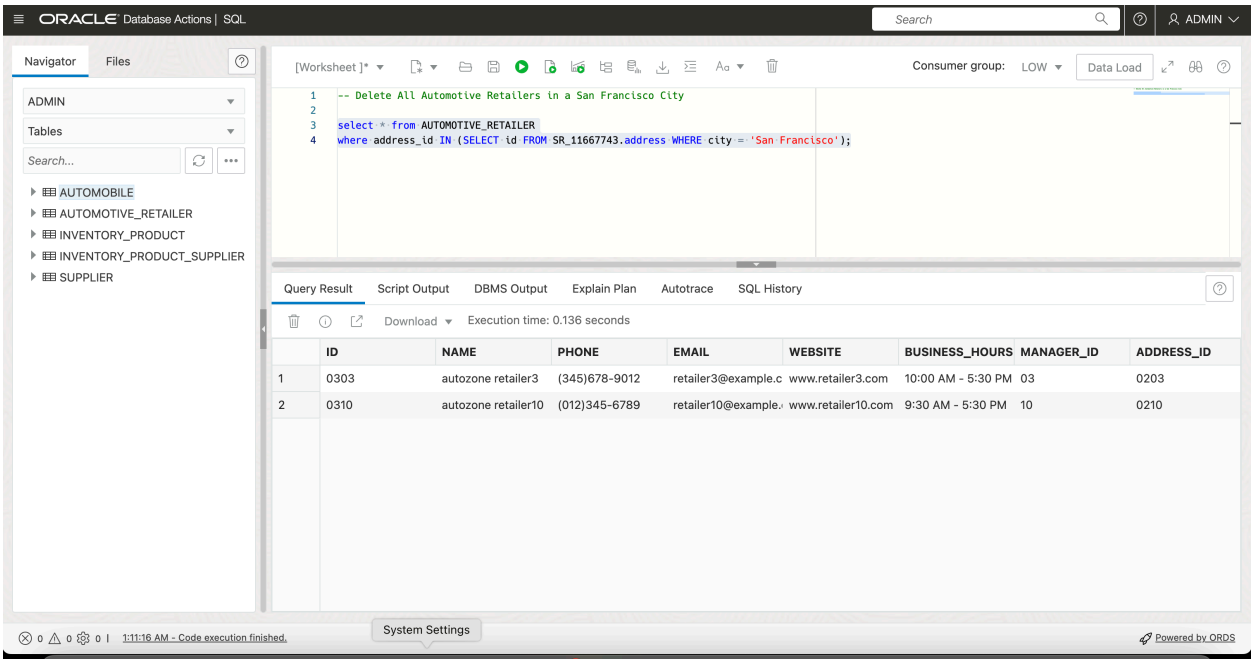
6 DELETE QUERIES:

Query-1: Delete All Automotive Retailers in a San Francisco City.

SQL Query Before Delete:

```
select * from AUTOMOTIVE_RETAILER
where address_id IN (SELECT id FROM SR_11667743.address WHERE city = 'San Francisco');
```

Screenshot:

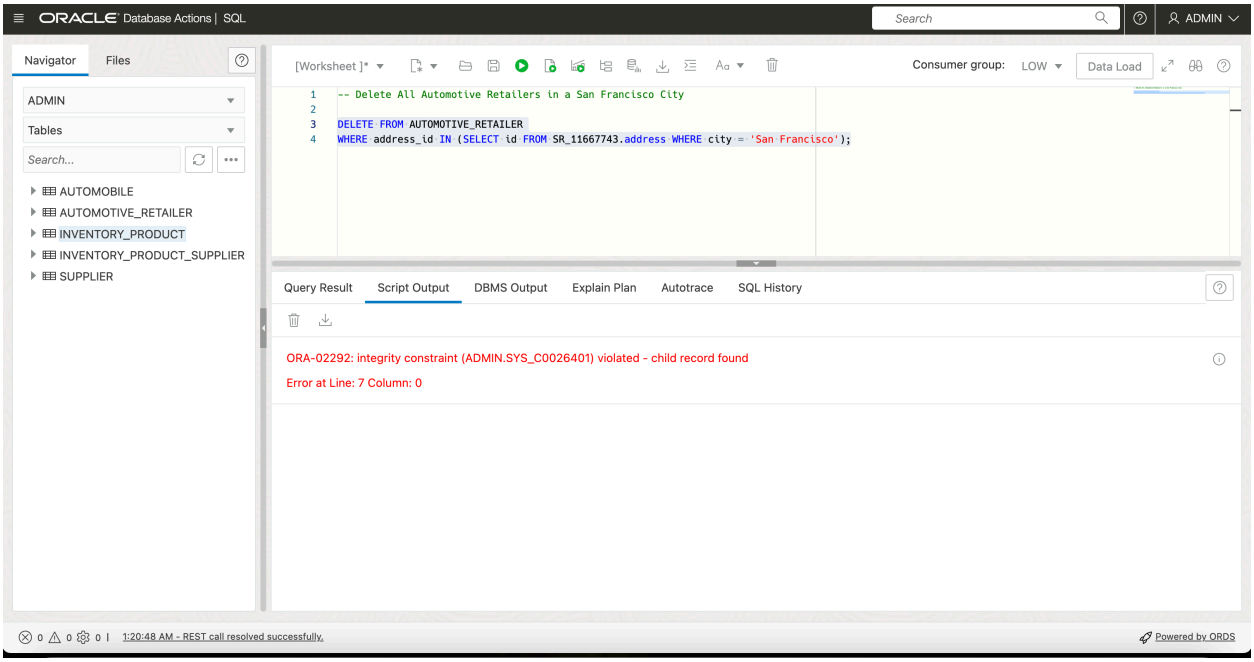


Explanation: As we are performing delete operation, the information present before deleting is displayed through this query. it first finds the unique identifiers (IDs) of San Francisco addresses in the "address" table and then matches those IDs with the "address_id" field in the "AUTOMOTIVE_RETAILER" table. This helps identify and select all the automotive retailers located in San Francisco.

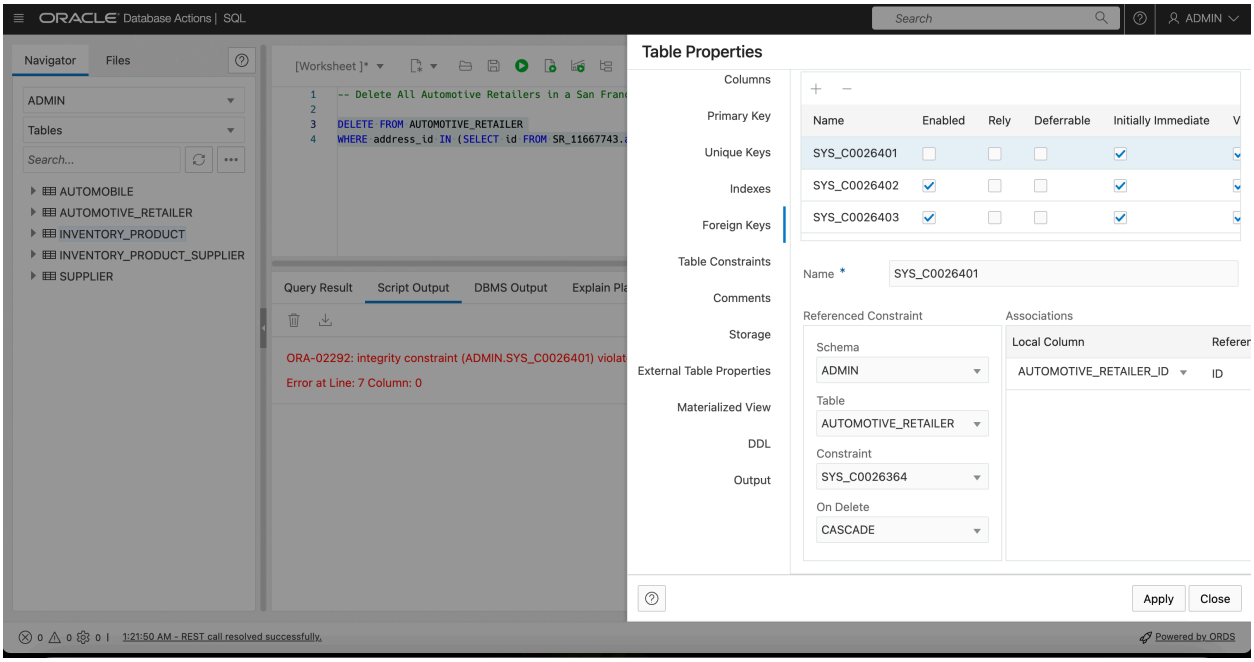
SQL Query to Delete:

DELETE FROM AUTOMOTIVE_RETAILER
WHERE address_id IN (SELECT id FROM SR_11667743.address WHERE city = 'San Francisco');

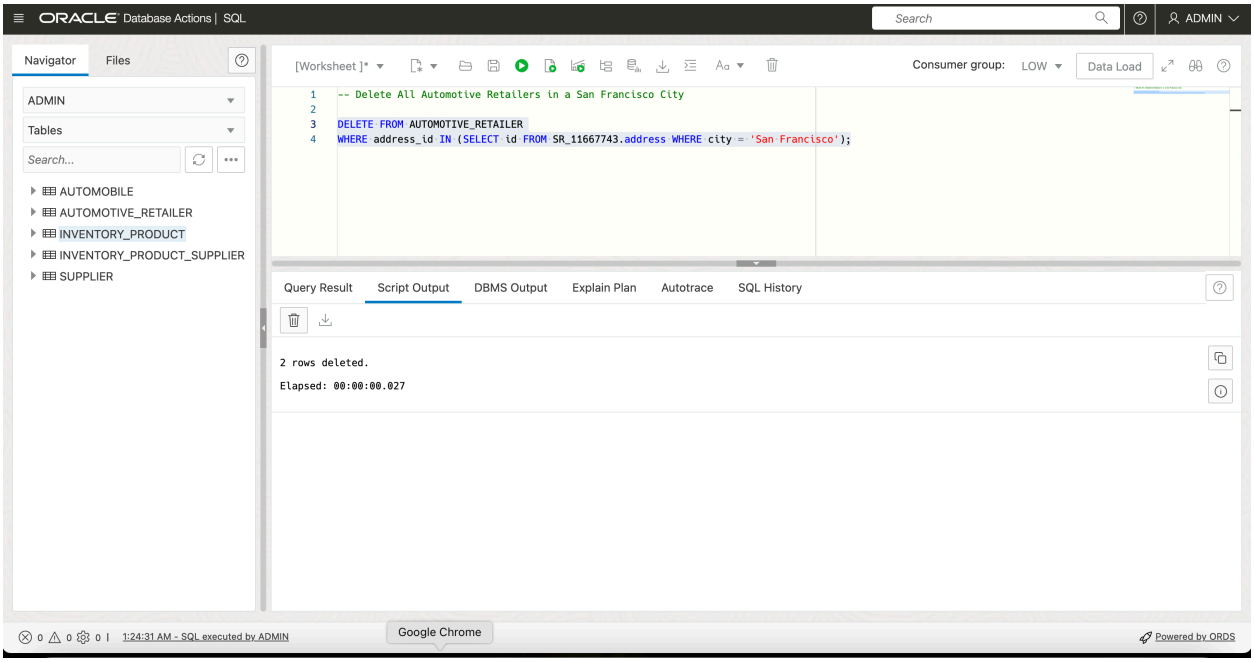
Screenshot:



Screenshot:

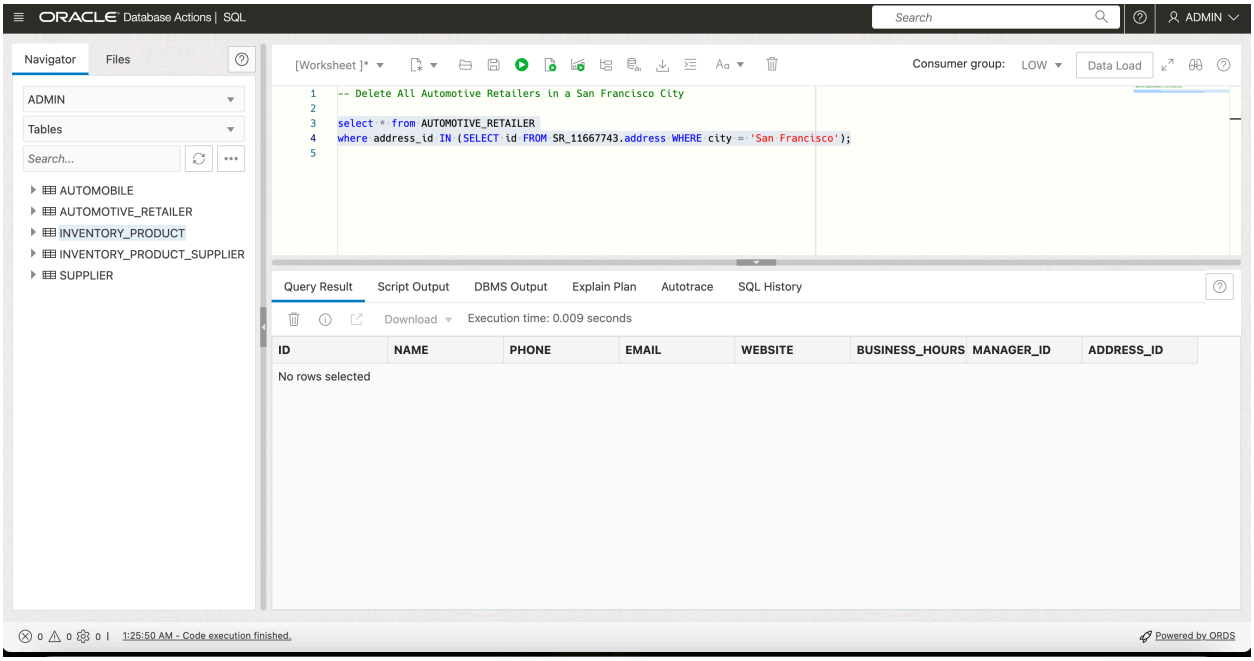


Explanation: Unable to delete the data due to integrity constraint violation, updating on Delete cascade to respective tables. *The same process is applied to below, wherever it is necessary.*



SQL Query After Delete: select * from AUTOMOTIVE_RETAILER
where address_id IN (SELECT id FROM SR_11667743.address WHERE city = 'San Francisco');

Screenshot:

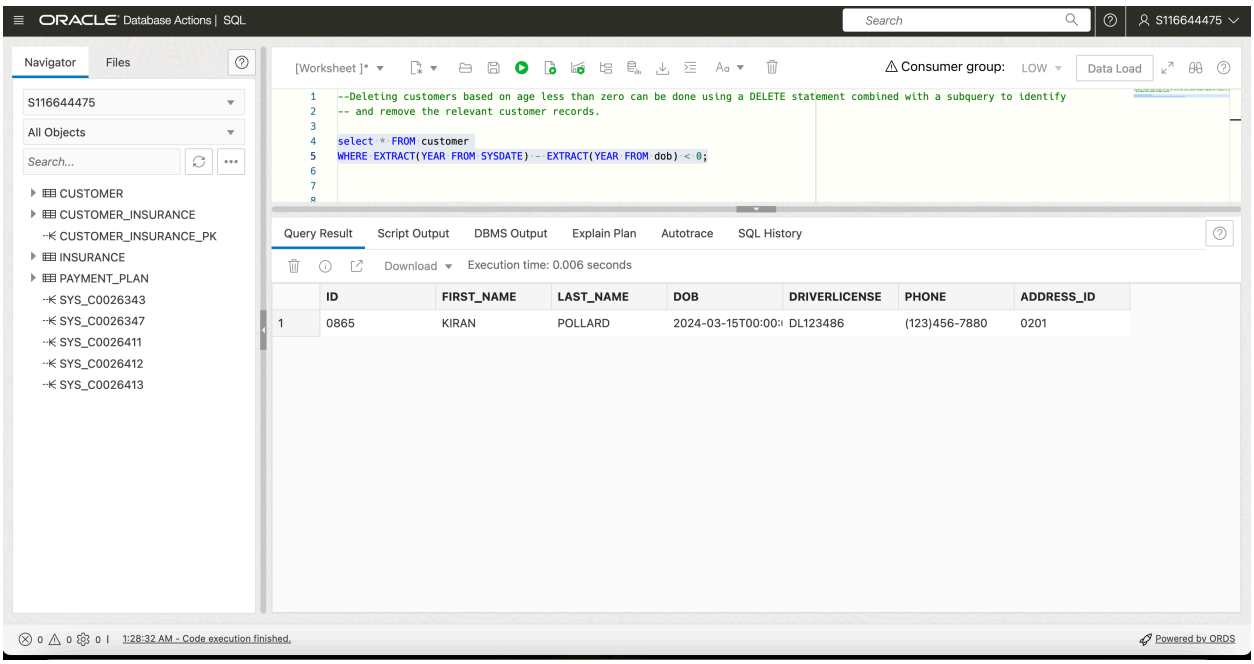


Explanation: The provided SQL query is to remove records from a table called "AUTOMOTIVE_RETAILER." The deletion is based on a condition that specifies removing rows where the "address_id" matches any ID found in a subquery result. The subquery retrieves IDs from the "address" table, but only those associated with the city 'San Francisco.' It specifically deletes entries that have an "address_id" linked to addresses in San Francisco. In simpler terms, it deletes information about automotive retailers in San Francisco from the database

Query-2: Deleting customers based on age less than zero can be done using a DELETE statement combined with a subquery to identify and remove the relevant customer records.

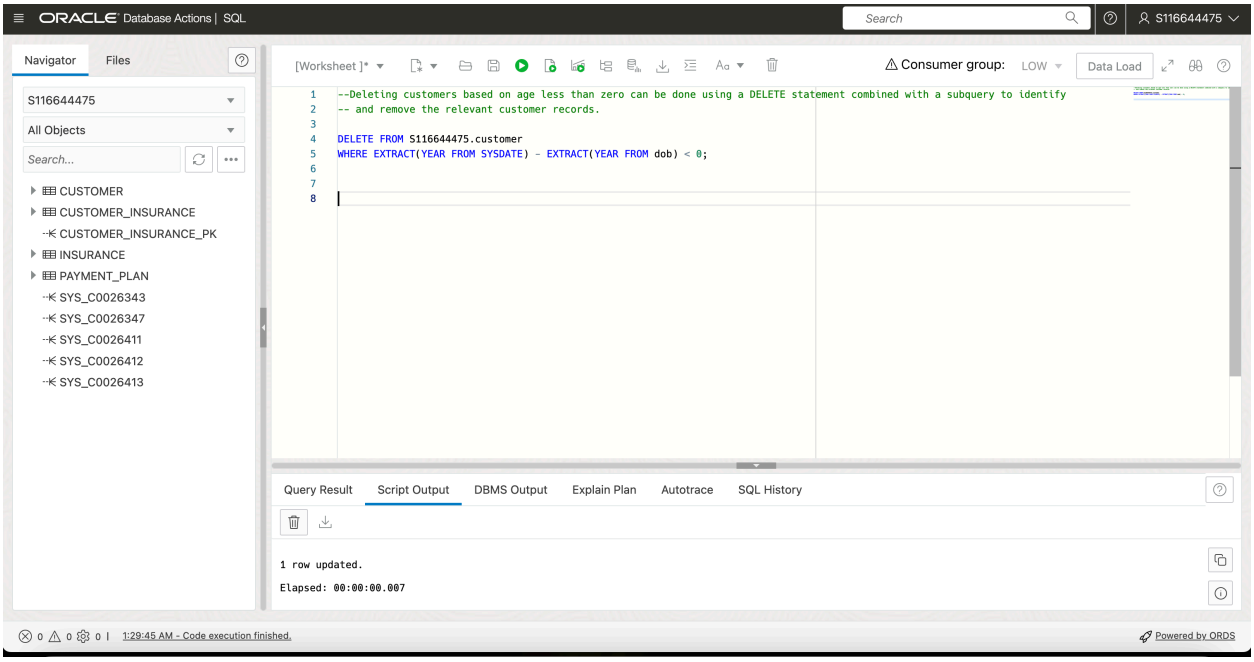
Select Query Before Delete: select * FROM customer
WHERE EXTRACT(YEAR FROM SYSDATE) - EXTRACT(YEAR FROM dob) < 0;

Screenshot:



SQL Query to Delete: DELETE FROM S116644475.customer WHERE EXTRACT(YEAR FROM SYSDATE) - EXTRACT(YEAR FROM dob) < 0;

Screenshot:

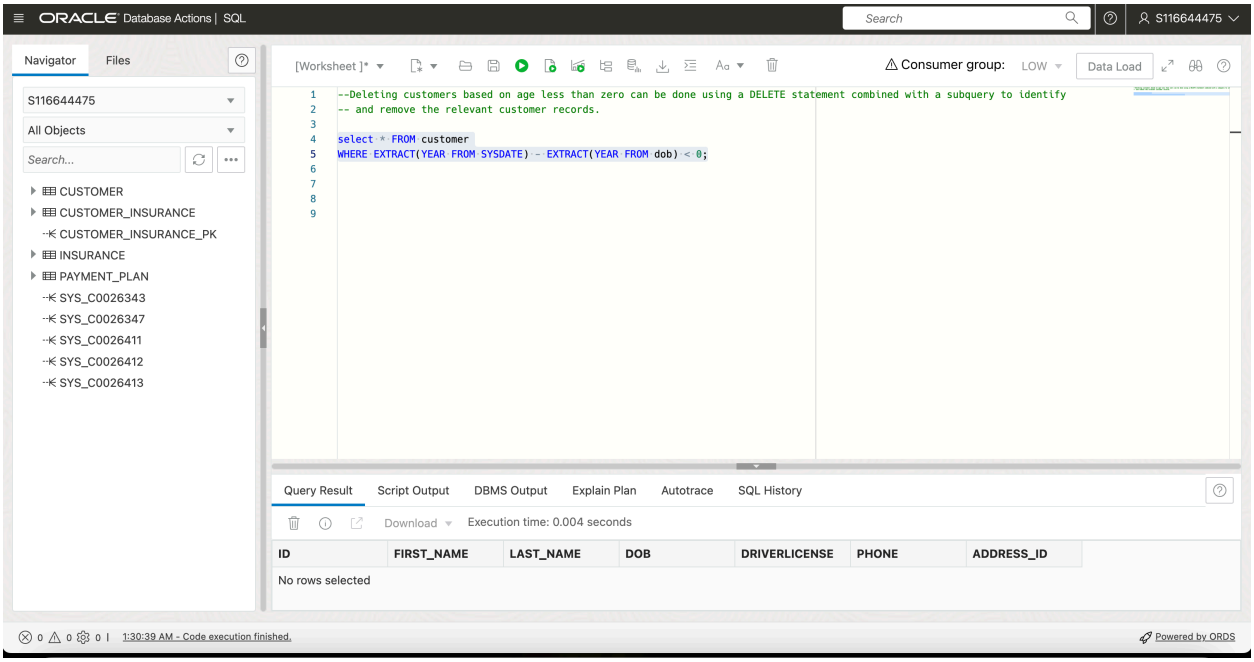


Explanation: The SQL query identifies and deletes customer records with an age less than zero. It does this by calculating the age of each customer using their date of birth and the current date. If it finds any customers whose calculated age is less than zero, it means there's a data problem, and these erroneous customer records will be removed from the database. This query helps maintain data accuracy by eliminating entries with impossible negative ages.

SQL Query After Delete:

select * FROM customer
WHERE EXTRACT(YEAR FROM SYSDATE) - EXTRACT(YEAR FROM dob) < 0;

Screenshot:



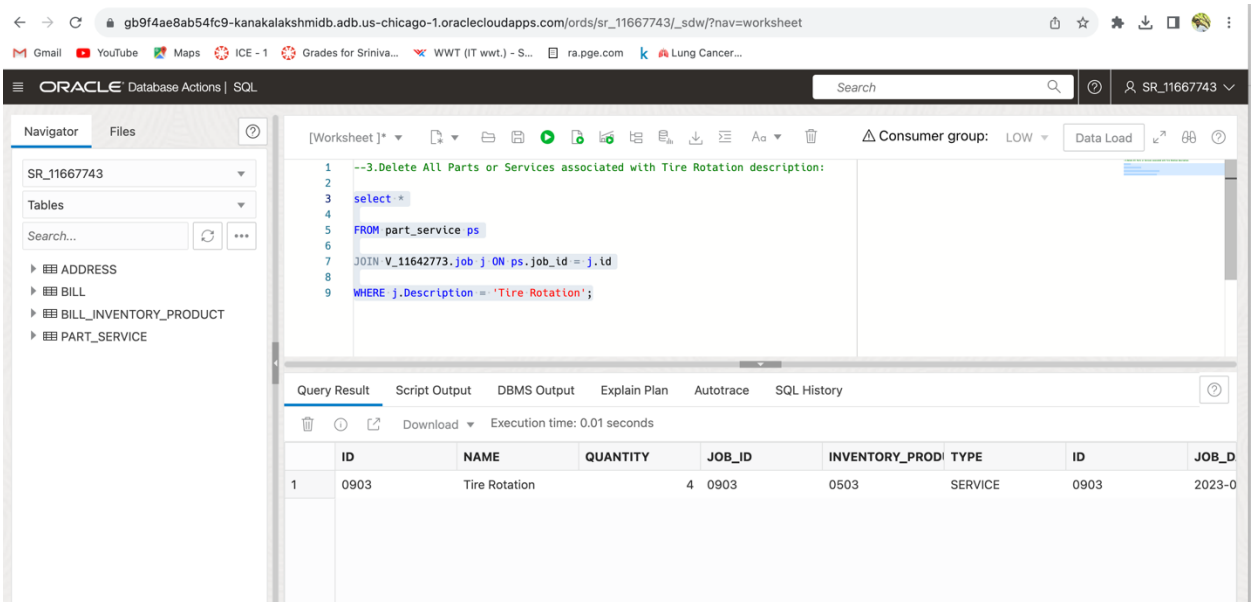
Explanation: This query fetches data from the "customer" table for individuals who, according to the current year, have not been born yet. It's essentially trying to find customers with birthdates in the future, based on the system's date. This is just to test whether the data got deleted or not.

Query-3: Delete All Parts or Services associated with Tire Rotation description.

SQL Query Before Delete:

```
select *
FROM part_service ps
JOIN V_11642773.job j ON ps.job_id = j.id
WHERE j.Description = 'Tire Rotation';
```

Screenshot:

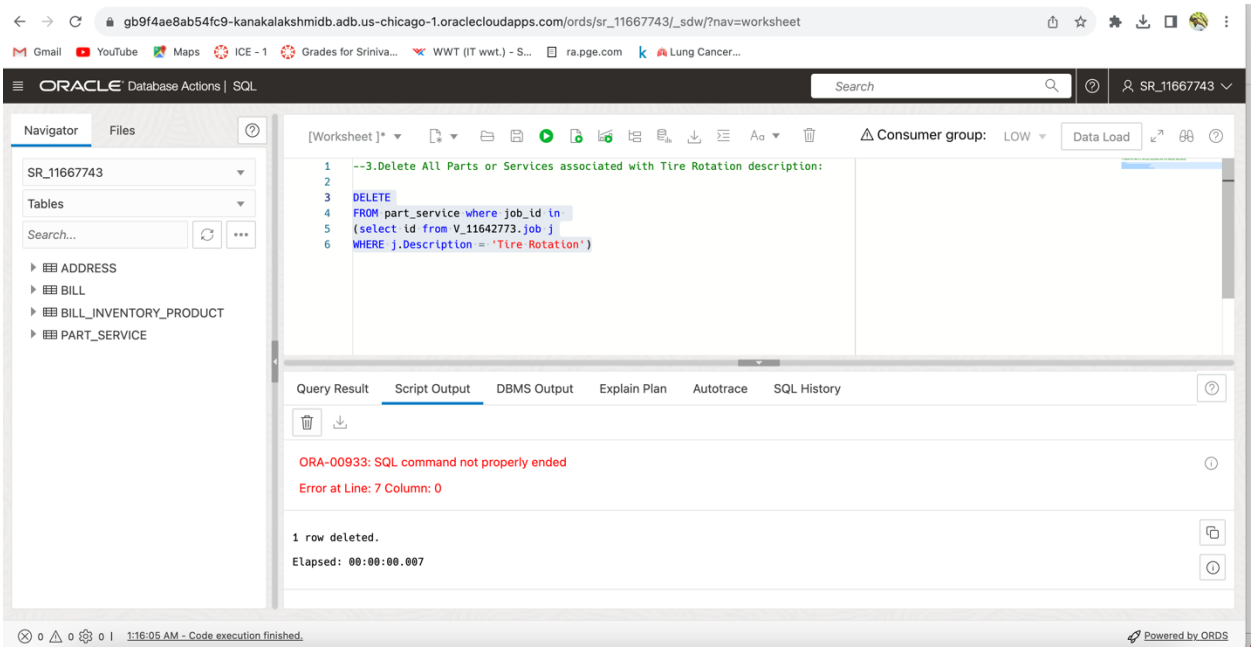


Explanation: This query identifies all the parts and services that are connected to the job with the description "Tire Rotation." It retrieves all the details about these parts and services. This is the data before the delete is performed.

SQL Query to Delete:

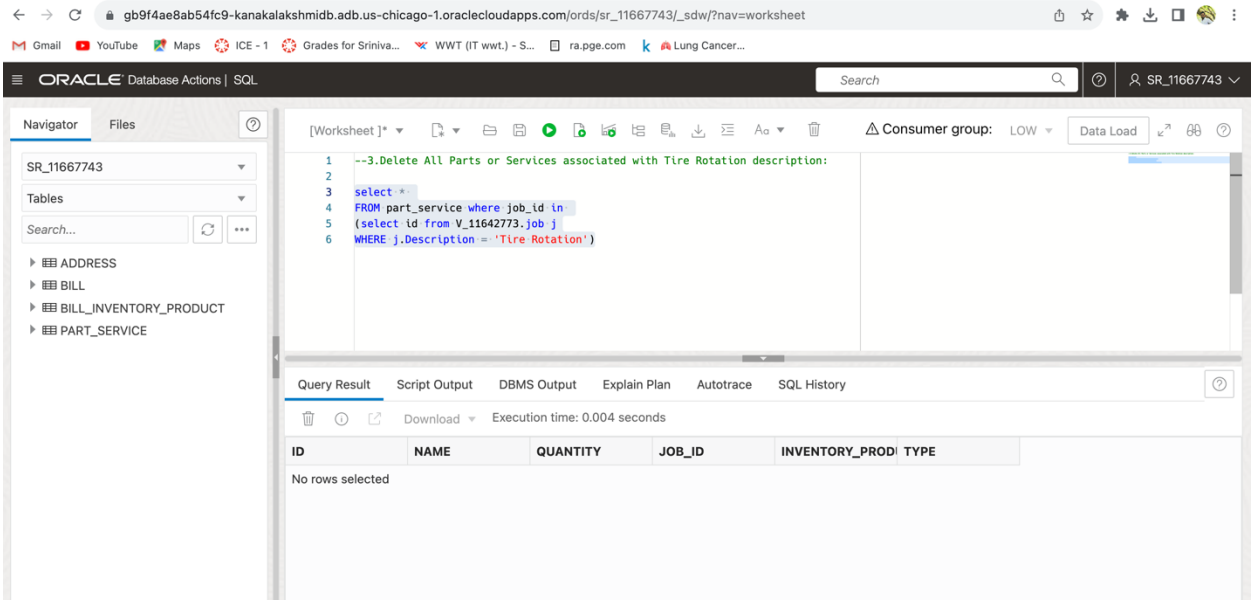
```
DELETE
FROM part_service where job_id in
(select id from V_11642773.job j
WHERE j.Description = 'Tire Rotation')
```

Screenshot:



Explanation: This query is designed to delete data from a table called "part_service." It targets records related to a specific job, which has the description "Tire Rotation." So, any services associated with this particular job will be removed from the "part_service" table.

Screenshot After Delete:



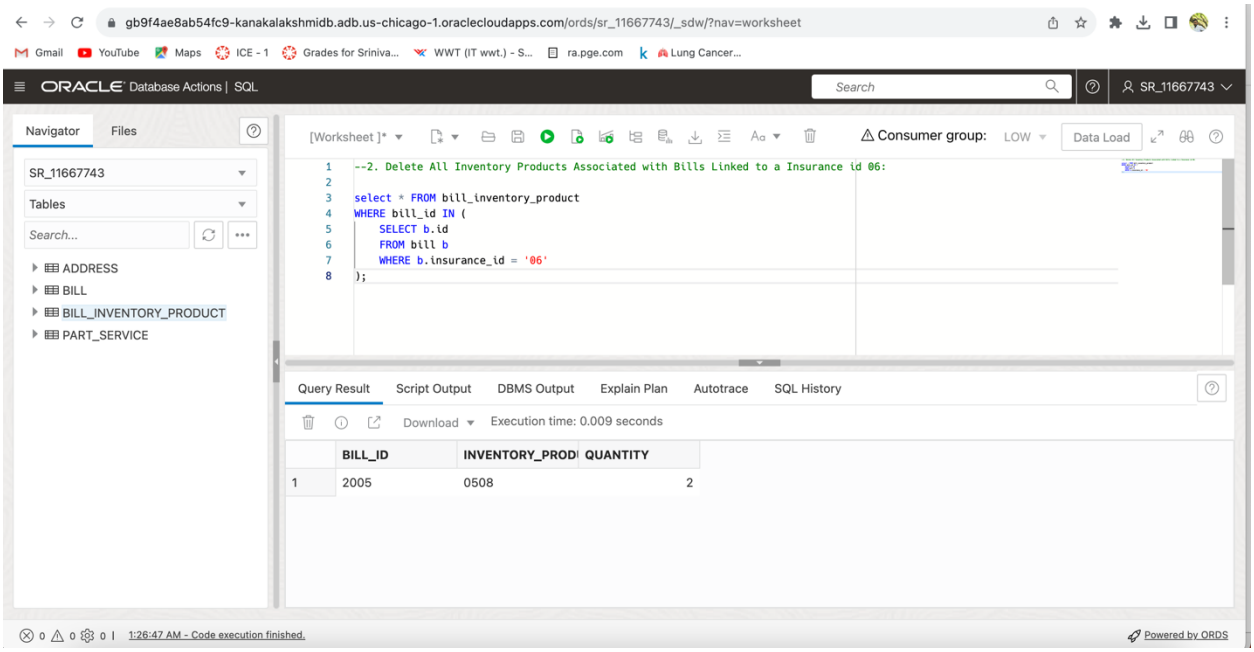
Explanation: This is the data after deleting.

Query-4: Delete All Inventory Products Associated with Bills Linked to a specific Insurance id.

Select Query Before Delete:

```
select * FROM bill_inventory_product
WHERE bill_id IN (SELECT b.id FROM bill b WHERE b.insurance_id = '06’);
```

Screenshot:

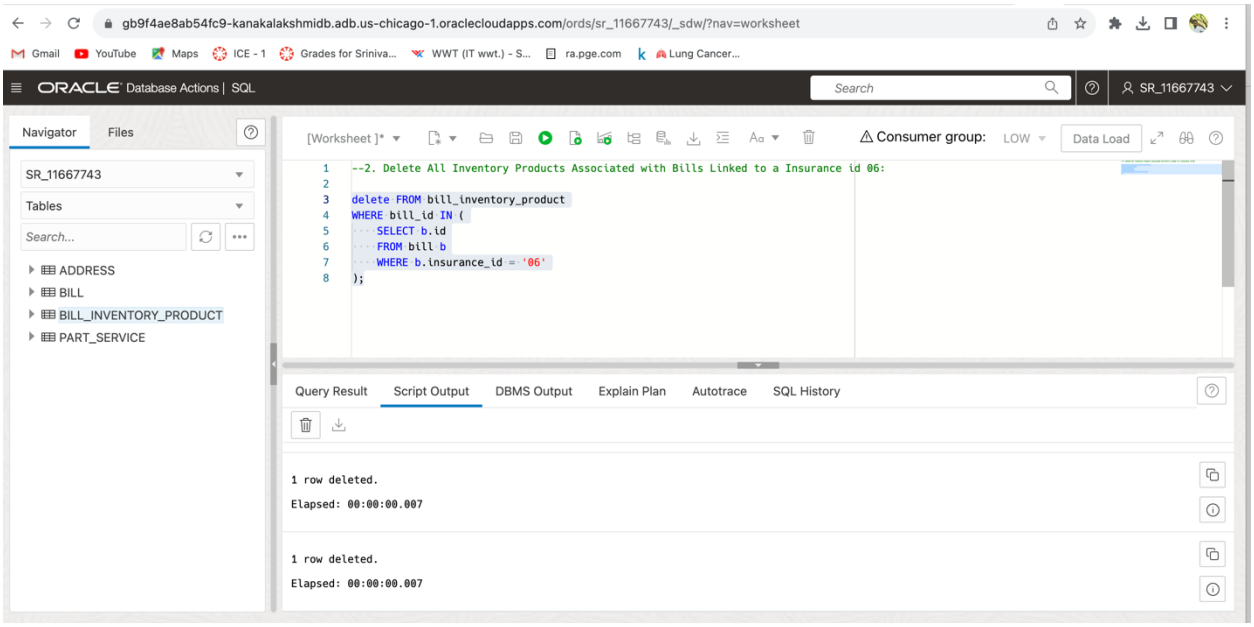


Explanation: This query fetches information about inventory products that are associated with bills having an insurance ID of '06'. It looks for these connections in the "bill_inventory_product" table and retrieves the relevant data. This is performed before performing delete on the table.

SQL Query to Delete:

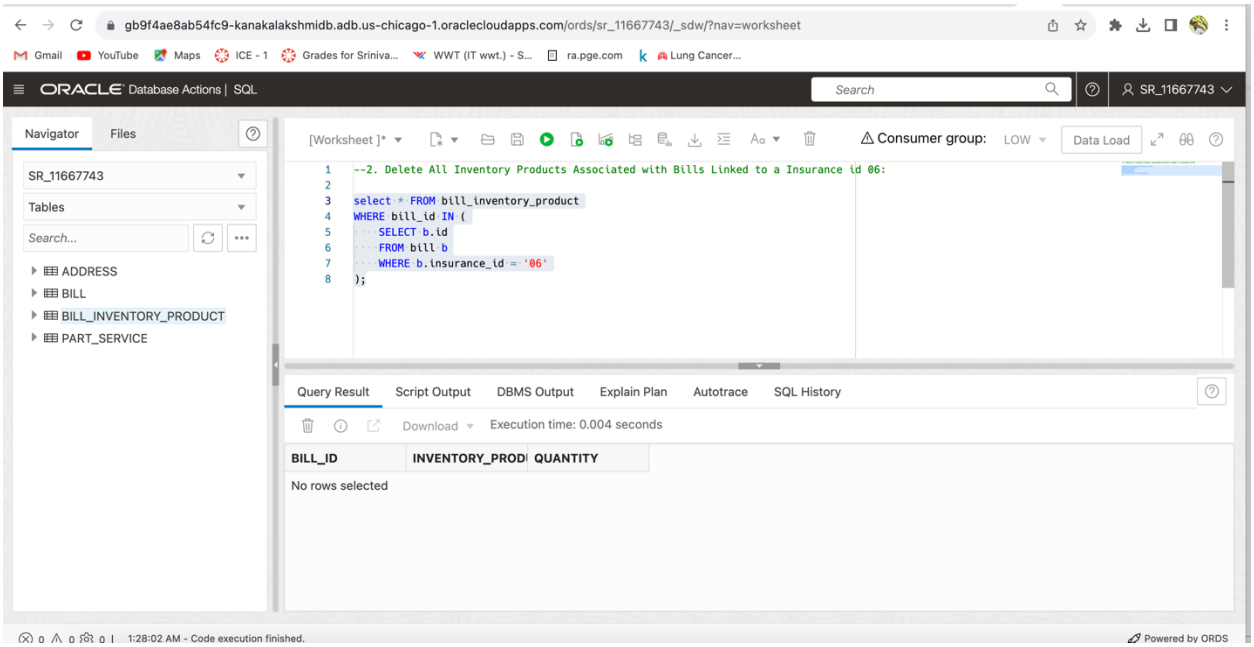
```
Delete FROM bill_inventory_product
WHERE bill_id IN (SELECT b.id FROM bill b WHERE b.insurance_id = '06');
```

Screenshot:



Explanation: It deletes data from the "bill_inventory_product" table. The deletion is based on a condition: it removes rows where the "bill_id" matches the result of a subquery. The subquery finds "bill" records with the condition that their "insurance_id" is equal to '06'.So, it deletes records in "bill_inventory_product" associated with bills having an "insurance_id" of '06'.

Screenshot After Delete:



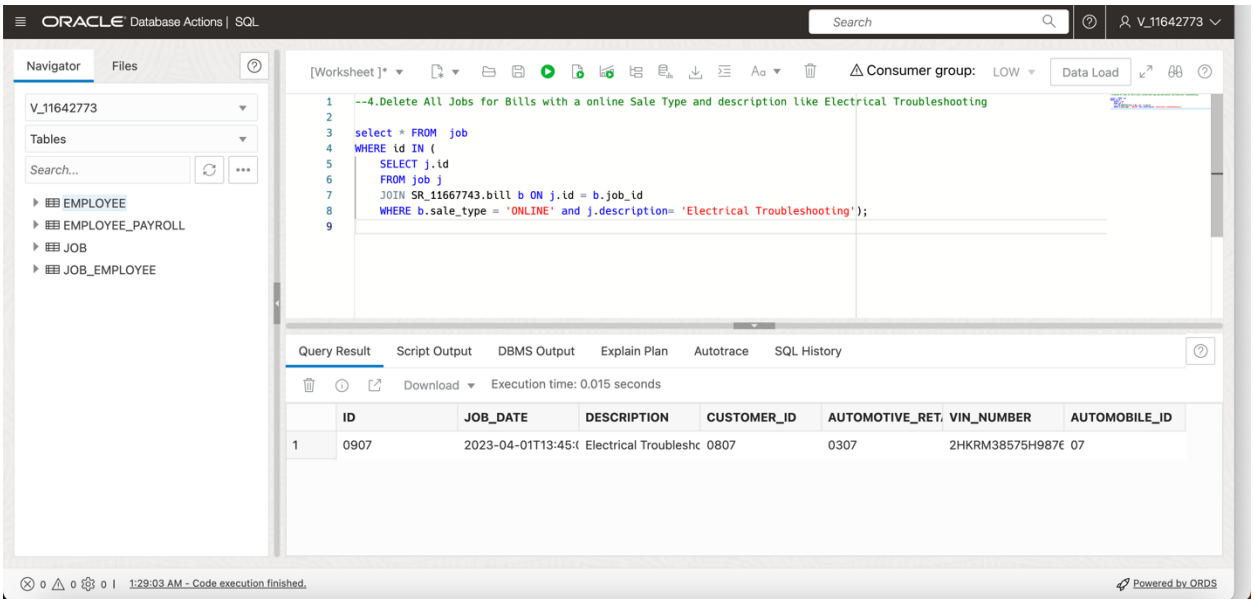
Explanation: Data after performing delete.

Query-5: Delete All Jobs for Bills with an online Sale Type and description like Electrical Troubleshooting

SQL Query Before Delete:

```
select * FROM  job
WHERE id IN (
   SELECT j.id
   FROM job j
   JOIN SR_11667743.bill b ON j.id = b.job_id
   WHERE b.sale_type = 'ONLINE' and j.description= 'Electrical Troubleshooting');
```

Screenshot:

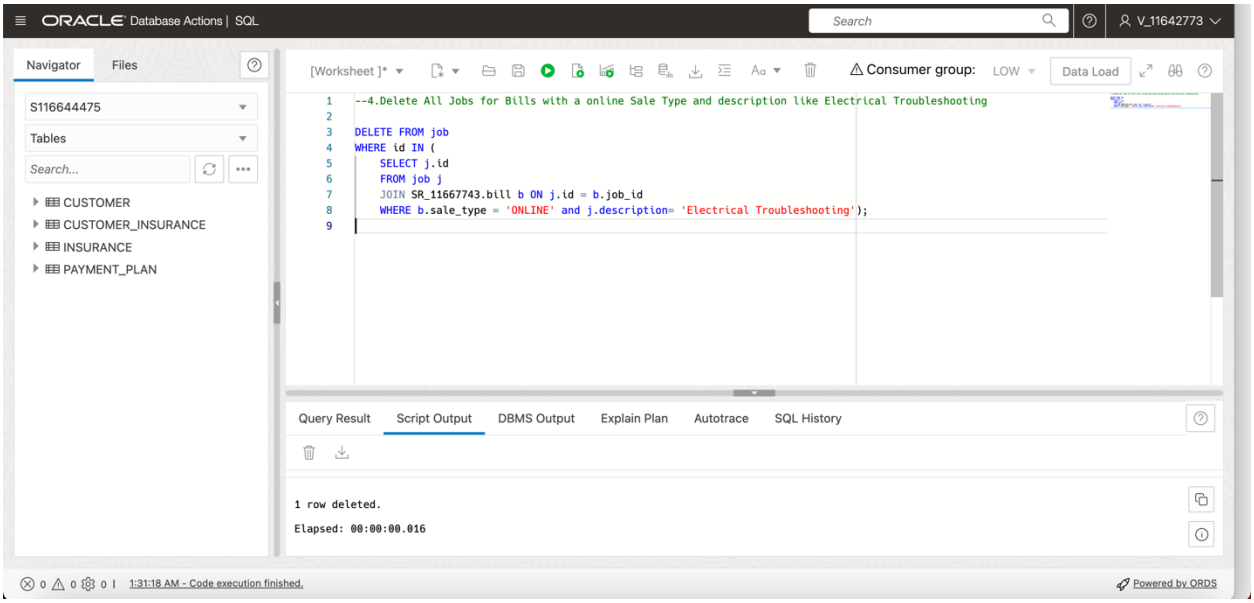


Explanation: This query retrieves all the job records connected to bills with an online sale type and a description containing the term 'Electrical Troubleshooting.' It's a way to find and identify specific jobs related to online sales with electrical troubleshooting needs.

SQL Query to Delete:

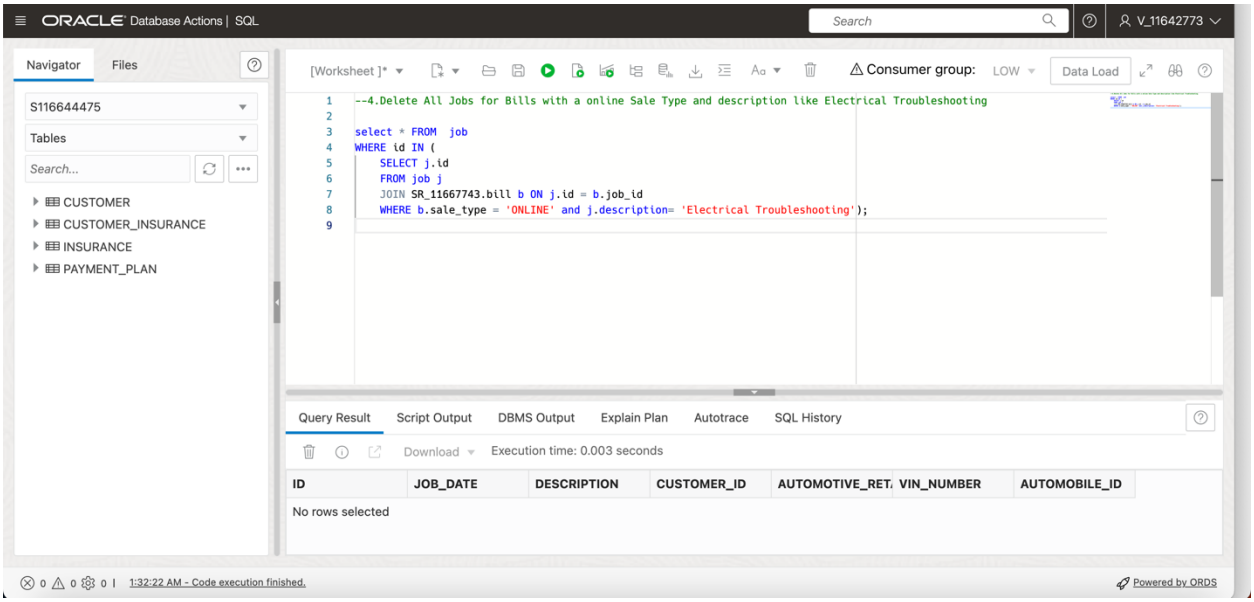
```
DELETE FROM job
WHERE id IN (
   SELECT j.id
   FROM job j
   JOIN SR_11667743.bill b ON j.id = b.job_id
   WHERE b.sale_type = 'ONLINE' and j.description= 'Electrical Troubleshooting');
```

Screenshot:



Explanation: This query will delete rows from the "job" table if they meet the conditions specified in the subquery. It specifically targets jobs with a description of 'Electrical Troubleshooting' that are associated with bills of the 'ONLINE' sale type. It joins the "job" table with another table named "bill" (aliased as "b") on the "job_id" column. It includes only rows where the "sale_type" in the "bill" table is 'ONLINE' and the "description" in the "job" table is 'Electrical Troubleshooting.'

Screenshot After Delete:

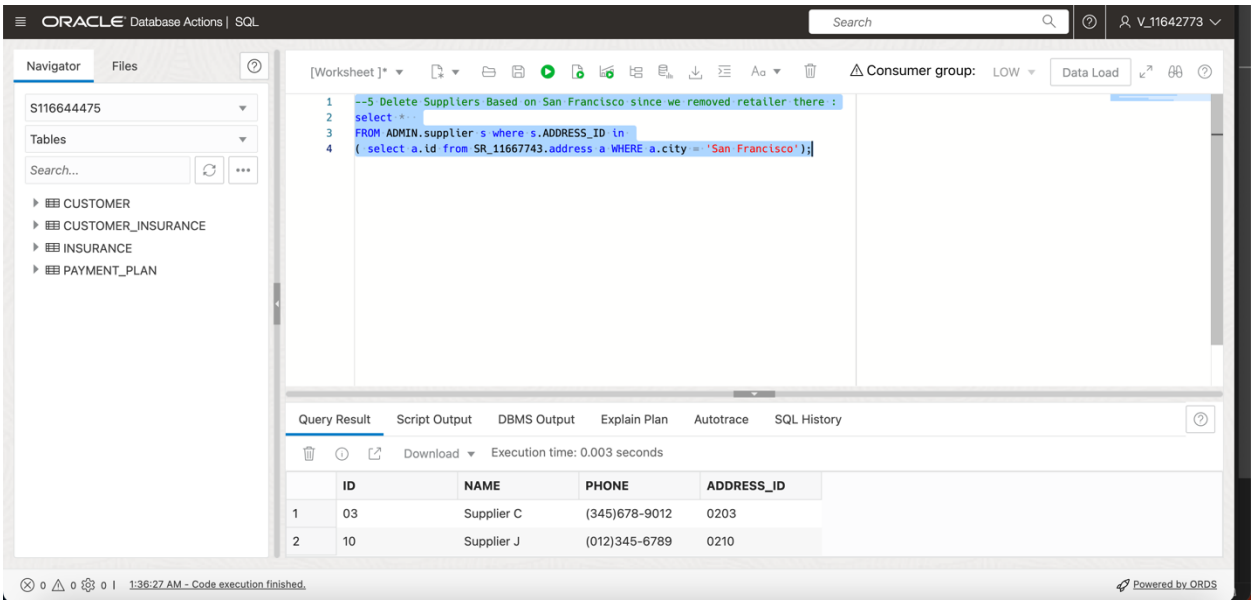


Query-6: Delete Suppliers Based on San Francisco as there is no retailers at San Francisco.

Select Query Before Delete:

```
select *
FROM ADMIN.supplier s where s.ADDRESS_ID in
( select a.id from SR_11667743.address a WHERE a.city = 'San Francisco');
```

Screenshot:

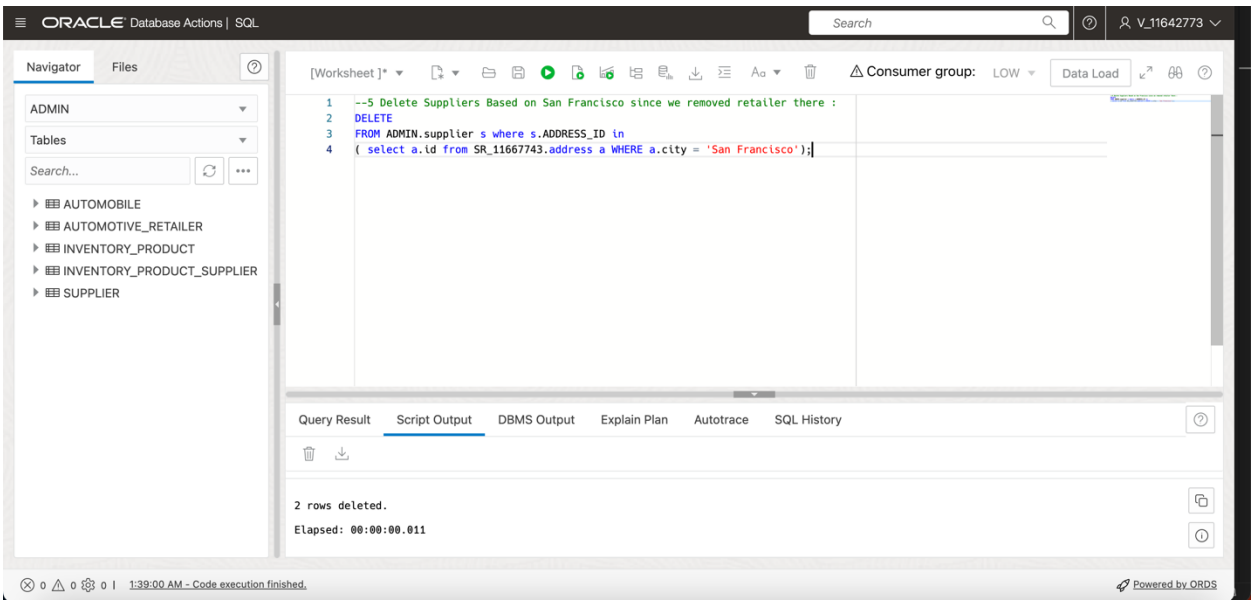


Explanation: This query fetches information about suppliers with addresses in San Francisco but does not perform any deletion action. The purpose of this query is to identify these suppliers, in preparation for a later deletion process.

SQL Query to Delete:

```
DELETE
FROM ADMIN.supplier s where s.ADDRESS_ID in
( select a.id from SR_11667743.address a WHERE a.city = 'San Francisco');
```

Screenshot:



Explanation: This SQL query is designed to delete some records in the "supplier" table in the "ADMIN" schema. It removes records that have an "ADDRESS_ID" matching specific conditions. In particular, it deletes supplier records associated with addresses located in 'San Francisco.' This action helps to manage the supplier data in the database by removing entries tied to San Francisco addresses.

Screenshot After Delete: