Data 225 Assignment 5
Submitted By: Anitha Balachandran (016684486)

1. Mini-Project on Oracle Database Analytics in the Cloud

Title: Analysis of E-commerce Sales Data

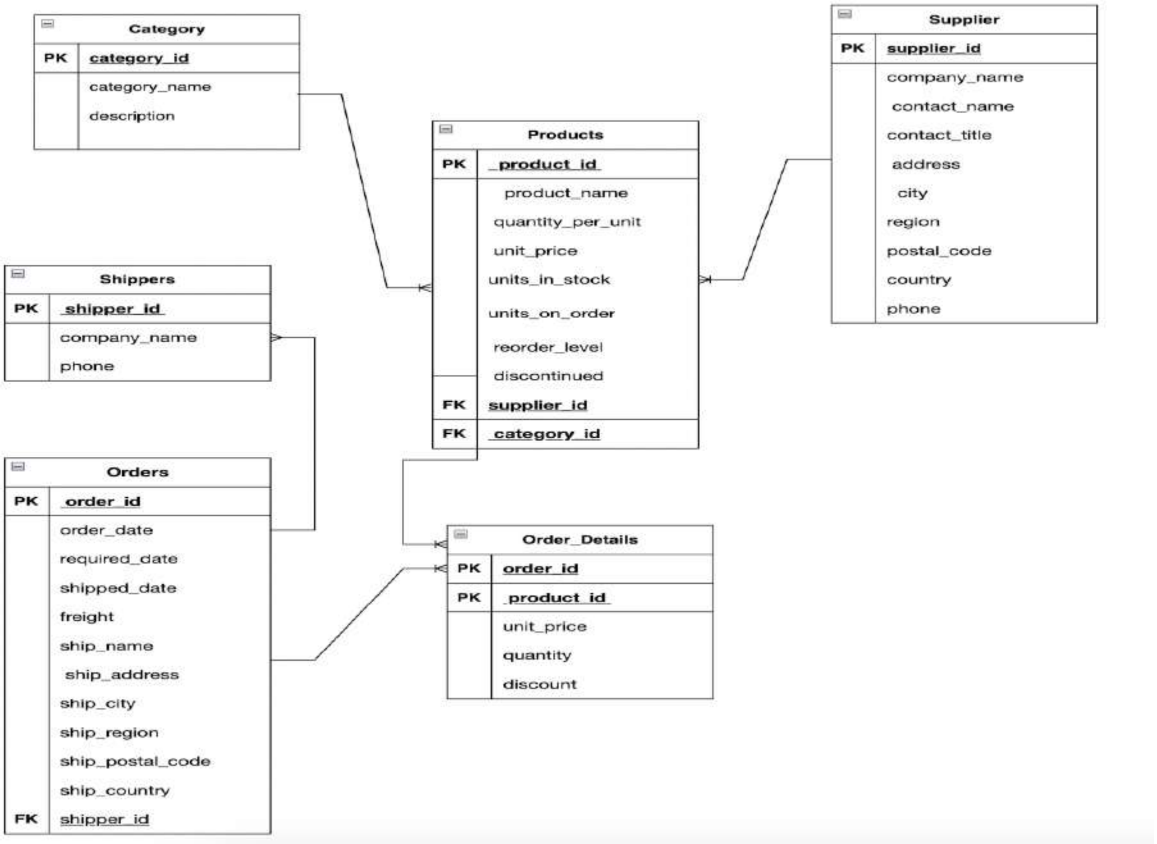
Introduction:

In today's data-driven world, businesses rely heavily on analytics to gain valuable insights that help them make informed decisions. One of the most widely used database management systems for data analytics is Oracle. This mini-project aims to leverage Oracle's powerful analytics capabilities to analyze e-commerce sales data. The project will focus on creating a database system that can efficiently store and manage data related to various aspects of e-commerce sales, including product details, order details, and more. By leveraging Oracle SQL and cloud technology, the project will explore the possibilities of data analytics in the cloud and provide insights that can drive business growth and success.

Project Goals:

- ➢ To use the Oracle database on Cloud to analyze e-commerce sales data, such as total sales revenue, top-selling products, most profitable categories, and customer buying patterns.
- ➢ To use the insights gained from the analysis to make data-driven decisions that can improve business operations, such as adjusting pricing strategies, optimizing inventory levels, and enhancing customer experiences.
- ➢ To develop an efficient and effective data management system that can store, organize, and retrieve data quickly and accurately which can lead to improved productivity, reduced costs, and better decision-making.

Data Modelling: Entity Relationship Diagram (ERD) for E-commerce Database



Methodology:

1. **Data Collection:** The first step of the project was to collect the e-commerce sales data from a Git-hub source. The data included information about product categories, suppliers, products, shippers, orders, and order details. The data was obtained from the following GitHub repository: https://github.com/ydchauh/yogeshchauhan.com-public.
2. **Data Modeling:** The next step was to create a database schema to store and manage the collected data. The schema was designed to include six tables, namely categories, suppliers, products, shippers, orders, order_details. The required relationships between these tables were established .
3. **Database implementation:** The database was implemented using Oracle SQL Developer on Cloud, which is a free tool provided by Oracle Corporation. The SQL statements were used to create the tables, insert the data, and define the constraints. A default schema was used to create the database.
4. **Data analysis:** Once the database was created and populated with data, various SQL queries were used to analyze the data. The Oracle SQL Developer provided a number of analytics functions such as AVG, SUM, COUNT, PARTITION BY, OVER, RANK, etc. These functions were used to perform statistical analysis on the data, such as finding the total sales amount, average order value, and top-selling products.

5. **Data aggregation:** The ROLLUP and CUBE operators were used to perform data aggregation and generate summary reports. These operators are used to group the data by one or more dimensions and provide grand totals. For example, the ROLLUP operator was used to generate a report that shows the total sales amount by product category and by year.

The following DDL statements were used to create the tables:

```
    1.  CREATE TABLE categories (
  category_id NUMBER(5) NOT NULL PRIMARY KEY,
  category_name VARCHAR2(15) NOT NULL,
  description VARCHAR2(15)
  );
```

```
    2.  CREATE TABLE suppliers (
  supplier_id NUMBER(5) NOT NULL PRIMARY KEY,
  company_name VARCHAR2(40) NOT NULL,
  contact_name VARCHAR2(30),
  contact_title VARCHAR2(30),
  address VARCHAR2(60),
  city VARCHAR2(15),
  region VARCHAR2(15),
  postal_code VARCHAR2(10),
  country VARCHAR2(15),
  phone VARCHAR2(24),
  fax VARCHAR2(24),
  homepage CLOB
);
```

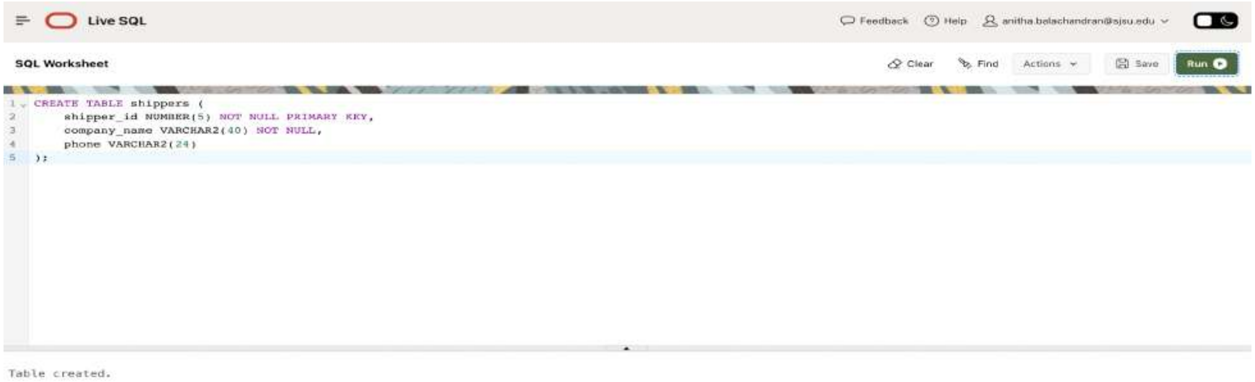
```
    3.  CREATE TABLE products (
  product_id NUMBER(5) NOT NULL PRIMARY KEY,
  product_name VARCHAR2(40) NOT NULL,
  supplier_id NUMBER(5),
  category_id NUMBER(5),
  quantity_per_unit VARCHAR2(20),
  unit_price FLOAT(126),
  units_in_stock NUMBER(5),
  units_on_order NUMBER(5),
  reorder_level NUMBER(5),
  discontinued NUMBER(10) NOT NULL,
  FOREIGN KEY (category_id) REFERENCES categories(category_id),
  FOREIGN KEY (supplier_id) REFERENCES suppliers(supplier_id)
);
```

```
    4.  CREATE TABLE shippers (
  shipper_id NUMBER(5) NOT NULL PRIMARY KEY,
  company_name VARCHAR2(40) NOT NULL,
    phone VARCHAR2(24)
);
```

```
    5. CREATE TABLE orders (
    order_id NUMBER(5) NOT NULL PRIMARY KEY,
    order_date DATE,
    required_date DATE,
    shipped_date DATE,
    ship_via NUMBER(5),
    freight FLOAT,
    ship_name VARCHAR2(40),
    ship_address VARCHAR2(60),
    ship_city VARCHAR2(15),
    ship_region VARCHAR2(15),
    ship_postal_code VARCHAR2(10),
    ship_country VARCHAR2(15),
    FOREIGN KEY (ship_via) REFERENCES shippers
);
```

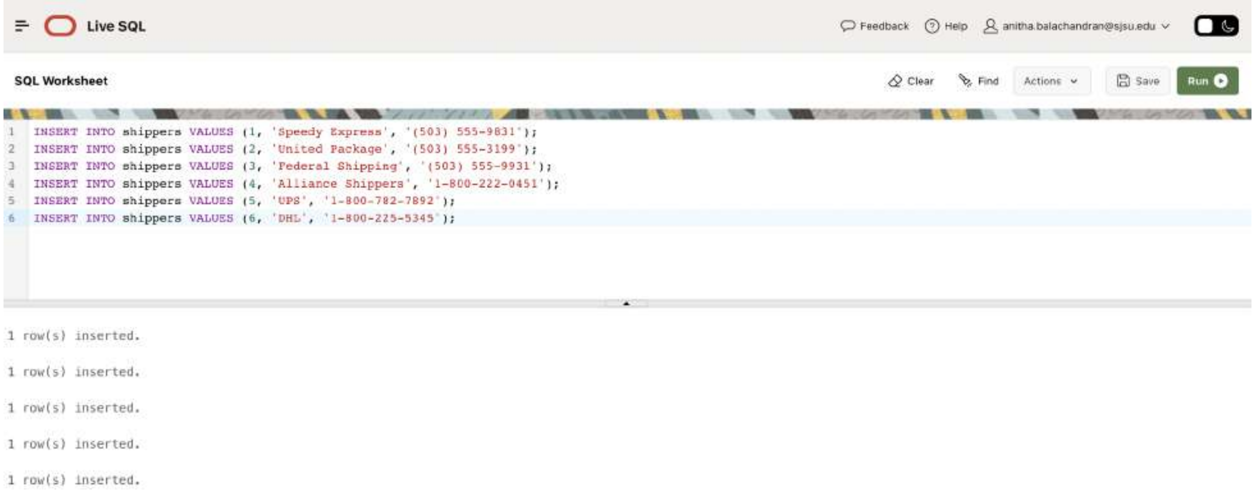
```
    6. CREATE TABLE order_details (
    order_id NUMBER(5) NOT NULL,
    product_id NUMBER(5) NOT NULL,
    unit_price FLOAT NOT NULL,
    quantity NUMBER(5) NOT NULL,
    discount FLOAT NOT NULL,
    PRIMARY KEY (order_id, product_id),
    FOREIGN KEY (product_id) REFERENCES products,
    FOREIGN KEY (order_id) REFERENCES orders
);
```

DDL Statements Snapshots:



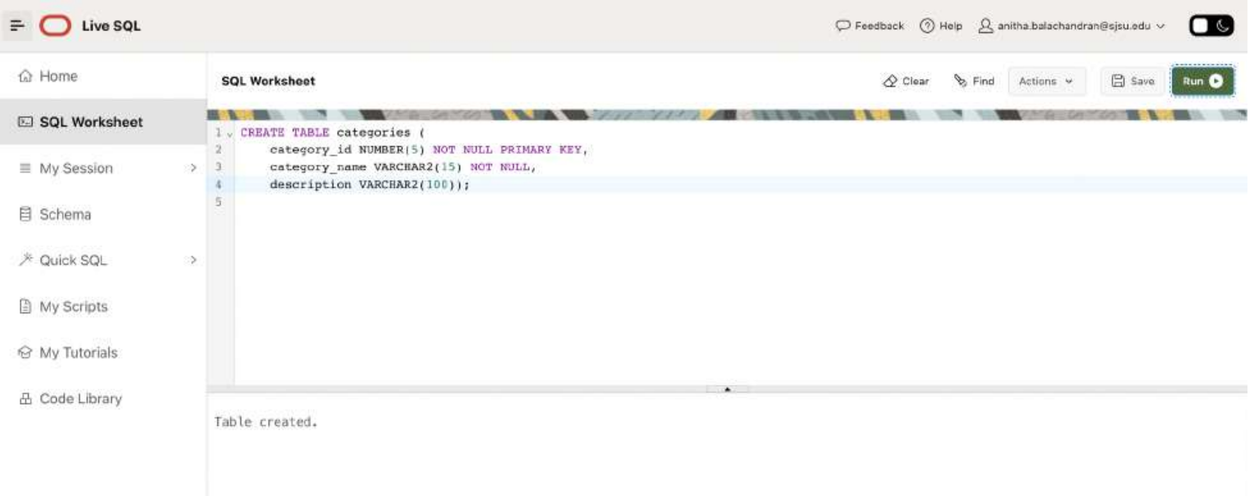
```
CREATE TABLE shippers (
    shipper_id NUMBER(5) NOT NULL PRIMARY KEY,
    company_name VARCHAR2(40) NOT NULL,
    phone VARCHAR2(24)
);
```

```
Table created.
```



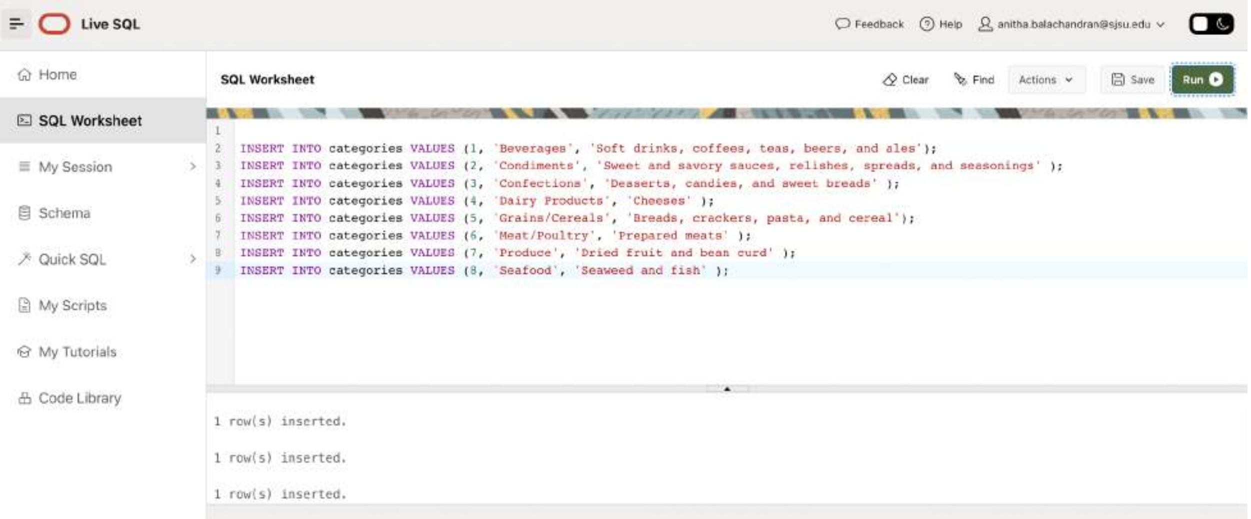
```sql
INSERT INTO shippers VALUES (1, 'Speedy Express', '(503) 555-9831');
INSERT INTO shippers VALUES (2, 'United Package', '(503) 555-3199');
INSERT INTO shippers VALUES (3, 'Federal Shipping', '(503) 555-9931');
INSERT INTO shippers VALUES (4, 'Alliance Shippers', '1-800-222-0451');
INSERT INTO shippers VALUES (5, 'UPS', '1-800-782-7892');
INSERT INTO shippers VALUES (6, 'DHL', '1-800-225-5345');
```

```
1 row(s) inserted.

1 row(s) inserted.

1 row(s) inserted.

1 row(s) inserted.

1 row(s) inserted.
```



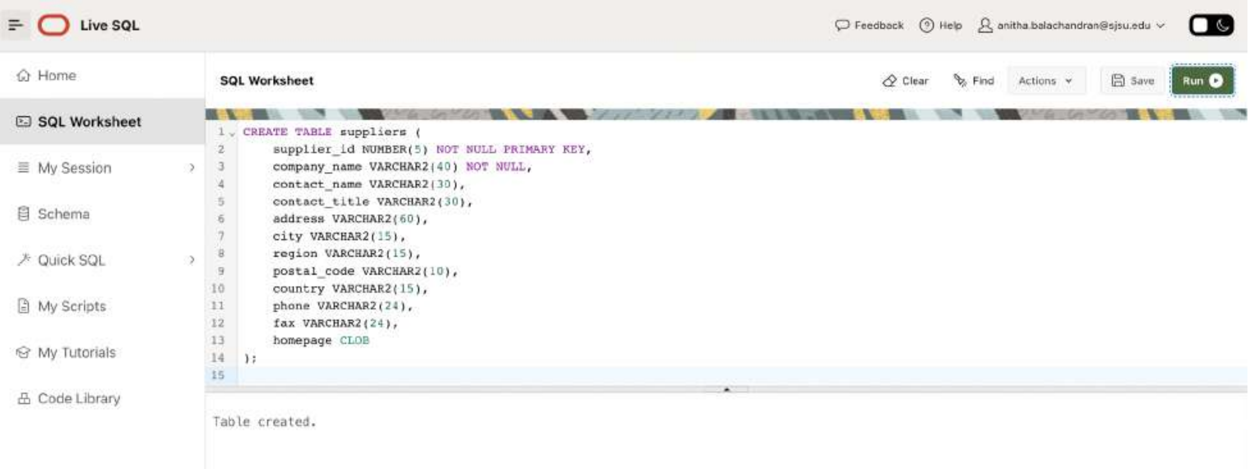
```sql
CREATE TABLE categories (
    category_id NUMBER(5) NOT NULL PRIMARY KEY,
    category_name VARCHAR2(15) NOT NULL,
    description VARCHAR2(100));
```

```
Table created.
```



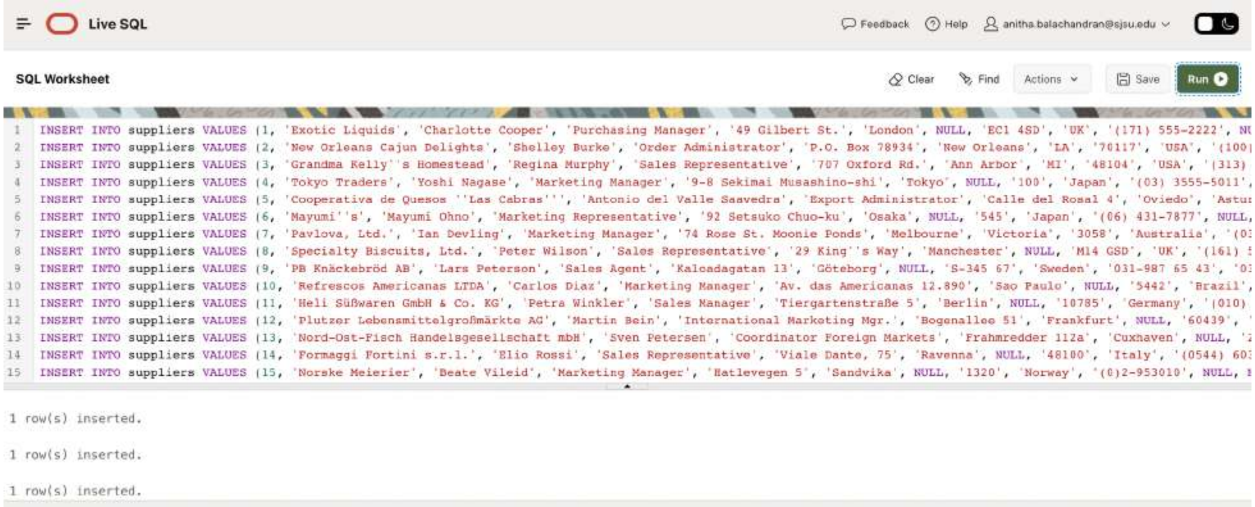
```sql

INSERT INTO categories VALUES (1, 'Beverages', 'Soft drinks, coffees, teas, beers, and ales');
INSERT INTO categories VALUES (2, 'Condiments', 'Sweet and savory sauces, relishes, spreads, and seasonings' );
INSERT INTO categories VALUES (3, 'Confections', 'Desserts, candies, and sweet breads' );
INSERT INTO categories VALUES (4, 'Dairy Products', 'Cheeses' );
INSERT INTO categories VALUES (5, 'Grains/Cereals', 'Breads, crackers, pasta, and cereal');
INSERT INTO categories VALUES (6, 'Meat/Poultry', 'Prepared meats' );
INSERT INTO categories VALUES (7, 'Produce', 'Dried fruit and bean curd' );
INSERT INTO categories VALUES (8, 'Seafood', 'Seaweed and fish' );
```

```
1 row(s) inserted.

1 row(s) inserted.

1 row(s) inserted.
```



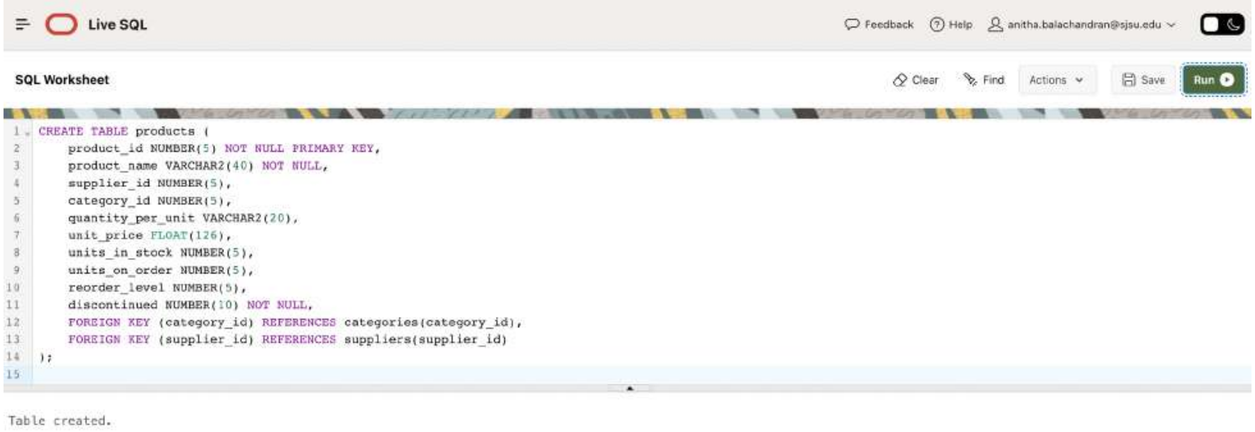
```sql
CREATE TABLE suppliers (
    supplier_id NUMBER(5) NOT NULL PRIMARY KEY,
    company_name VARCHAR2(40) NOT NULL,
    contact_name VARCHAR2(30),
    contact_title VARCHAR2(30),
    address VARCHAR2(60),
    city VARCHAR2(15),
    region VARCHAR2(15),
    postal_code VARCHAR2(10),
    country VARCHAR2(15),
    phone VARCHAR2(24),
    fax VARCHAR2(24),
    homepage CLOB
);
```

```
Table created.
```



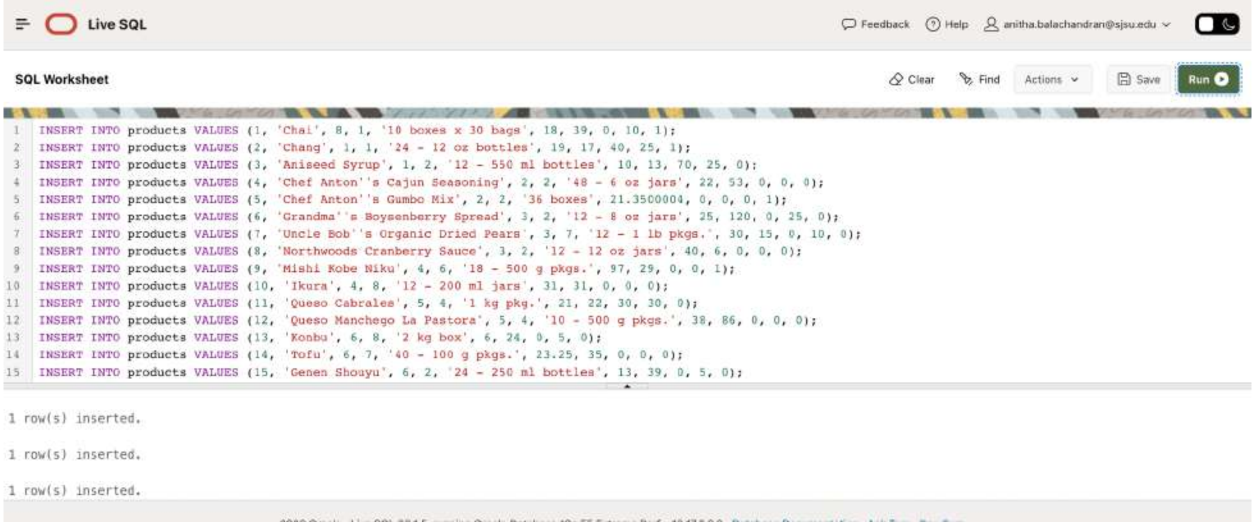
```sql
INSERT INTO suppliers VALUES (1, 'Exotic Liquids', 'Charlotte Cooper', 'Purchasing Manager', '49 Gilbert St.', 'London', NULL, 'EC1 4SD', 'UK', '(171) 555-2222', NU
INSERT INTO suppliers VALUES (2, 'New Orleans Cajun Delights', 'Shelley Burke', 'Order Administrator', 'P.O. Box 78934', 'New Orleans', 'LA', '70117', 'USA', '(100
INSERT INTO suppliers VALUES (3, 'Grandma Kelly''s Homestead', 'Regina Murphy', 'Sales Representative', '707 Oxford Rd.', 'Ann Arbor', 'MI', '48104', 'USA', '(313)
INSERT INTO suppliers VALUES (4, 'Tokyo Traders', 'Yoshi Nagase', 'Marketing Manager', '9-8 Sekimai Musashino-shi', 'Tokyo', NULL, '100', 'Japan', '(03) 3555-5011',
INSERT INTO suppliers VALUES (5, 'Cooperativa de Quesos ''Las Cabras''', 'Antonio del Valle Saavedra', 'Export Administrator', 'Calle del Rosal 4', 'Oviedo', 'Astu
INSERT INTO suppliers VALUES (6, 'Mayumi''s', 'Mayumi Ohno', 'Marketing Representative', '92 Setsuko Chuo-ku', 'Osaka', NULL, '545', 'Japan', '(06) 431-7877', NULL,
INSERT INTO suppliers VALUES (7, 'Pavlova, Ltd.', 'Ian Devling', 'Marketing Manager', '74 Rose St. Moonie Ponds', 'Melbourne', 'Victoria', '3058', 'Australia', '(0
INSERT INTO suppliers VALUES (8, 'Specialty Biscuits, Ltd.', 'Peter Wilson', 'Sales Representative', '29 King''s Way', 'Manchester', NULL, 'M14 GSD', 'UK', '(161) 
INSERT INTO suppliers VALUES (9, 'PB Knäckebröd AB', 'Lars Peterson', 'Sales Agent', 'Kaloadagatan 13', 'Göteborg', NULL, 'S-345 67', 'Sweden', '031-987 65 43', '0
INSERT INTO suppliers VALUES (10, 'Refrescos Americanas LTDA', 'Carlos Diaz', 'Marketing Manager', 'Av. das Americanas 12.890', 'Sao Paulo', NULL, '5442', 'Brazil',
INSERT INTO suppliers VALUES (11, 'Heli Süßwaren GmbH & Co. KG', 'Petra Winkler', 'Sales Manager', 'Tiergartenstraße 5', 'Berlin', NULL, '10785', 'Germany', '(010)
INSERT INTO suppliers VALUES (12, 'Plutzer Lebensmittelgroßmärkte AG', 'Martin Bein', 'International Markoting Mgr.', 'Bogenallee 51', 'Frankfurt', NULL, '60439', '
INSERT INTO suppliers VALUES (13, 'Nord-Ost-Fisch Handelsgesellschaft mbH', 'Sven Petersen', 'Coordinator Foreign Markets', 'Frahmredder 112a', 'Cuxhaven', NULL, '
INSERT INTO suppliers VALUES (14, 'Formaggi Fortini s.r.l.', 'Elio Rossi', 'Sales Representative', 'Viale Dante, 75', 'Ravenna', NULL, '48100', 'Italy', '(0544) 60
INSERT INTO suppliers VALUES (15, 'Norske Meierier', 'Beate Vileid', 'Marketing Manager', 'Hatlevegen 5', 'Sandvika', NULL, '1320', 'Norway', '(0)2-953010', NULL,
```

```
1 row(s) inserted.

1 row(s) inserted.

1 row(s) inserted.
```



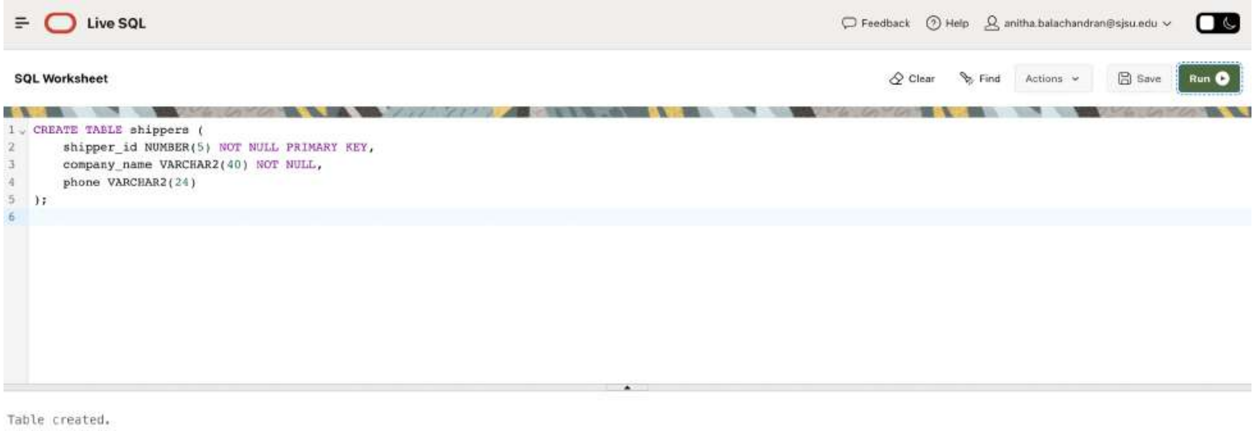
```
CREATE TABLE products (
    product_id NUMBER(5) NOT NULL PRIMARY KEY,
    product_name VARCHAR2(40) NOT NULL,
    supplier_id NUMBER(5),
    category_id NUMBER(5),
    quantity_per_unit VARCHAR2(20),
    unit_price FLOAT(126),
    units_in_stock NUMBER(5),
    units_on_order NUMBER(5),
    reorder_level NUMBER(5),
    discontinued NUMBER(10) NOT NULL,
    FOREIGN KEY (category_id) REFERENCES categories(category_id),
    FOREIGN KEY (supplier_id) REFERENCES suppliers(supplier_id)
);
```

```
Table created.
```



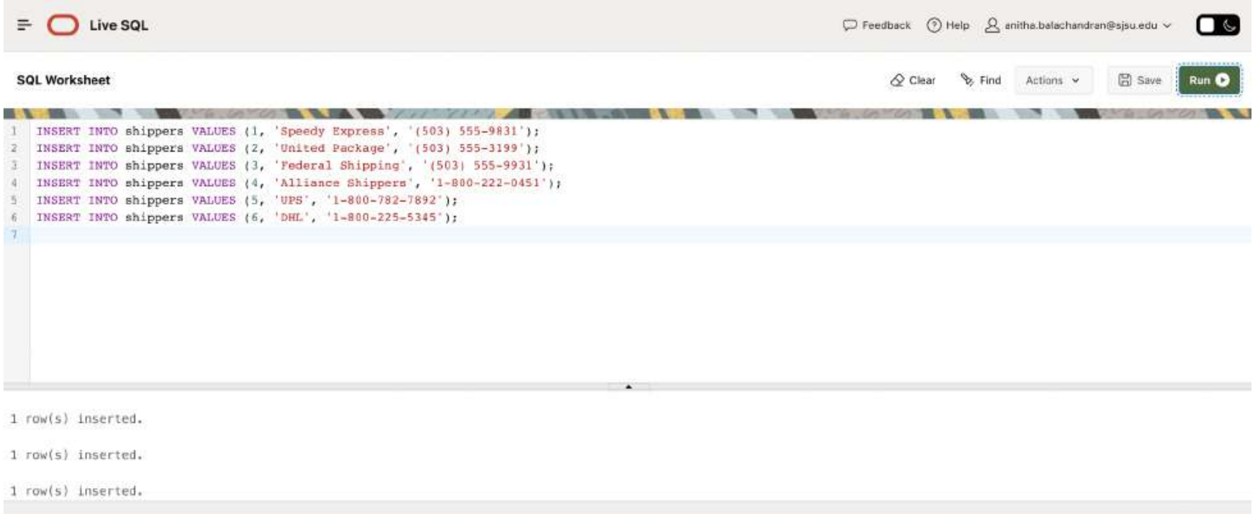
```
INSERT INTO products VALUES (1, 'Chai', 8, 1, '10 boxes x 30 bags', 18, 39, 0, 10, 1);
INSERT INTO products VALUES (2, 'Chang', 1, 1, '24 - 12 oz bottles', 19, 17, 40, 25, 1);
INSERT INTO products VALUES (3, 'Aniseed Syrup', 1, 2, '12 - 550 ml bottles', 10, 13, 70, 25, 0);
INSERT INTO products VALUES (4, 'Chef Anton''s Cajun Seasoning', 2, 2, '48 - 6 oz jars', 22, 53, 0, 0, 0);
INSERT INTO products VALUES (5, 'Chef Anton''s Gumbo Mix', 2, 2, '36 boxes', 21.3500004, 0, 0, 0, 1);
INSERT INTO products VALUES (6, 'Grandma''s Boysenberry Spread', 3, 2, '12 - 8 oz jars', 25, 120, 0, 25, 0);
INSERT INTO products VALUES (7, 'Uncle Bob''s Organic Dried Pears', 3, 7, '12 - 1 lb pkgs.', 30, 15, 0, 10, 0);
INSERT INTO products VALUES (8, 'Northwoods Cranberry Sauce', 3, 2, '12 - 12 oz jars', 40, 6, 0, 0, 0);
INSERT INTO products VALUES (9, 'Mishi Kobe Niku', 4, 6, '18 - 500 g pkgs.', 97, 29, 0, 0, 1);
INSERT INTO products VALUES (10, 'Ikura', 4, 8, '12 - 200 ml jars', 31, 31, 0, 0, 0);
INSERT INTO products VALUES (11, 'Queso Cabrales', 5, 4, '1 kg pkg.', 21, 22, 30, 30, 0);
INSERT INTO products VALUES (12, 'Queso Manchego La Pastora', 5, 4, '10 - 500 g pkgs.', 38, 86, 0, 0, 0);
INSERT INTO products VALUES (13, 'Konbu', 6, 8, '2 kg box', 6, 24, 0, 5, 0);
INSERT INTO products VALUES (14, 'Tofu', 6, 7, '40 - 100 g pkgs.', 23.25, 35, 0, 0, 0);
INSERT INTO products VALUES (15, 'Genen Shouyu', 6, 2, '24 - 250 ml bottles', 13, 39, 0, 5, 0);
```

```
1 row(s) inserted.

1 row(s) inserted.

1 row(s) inserted.
```



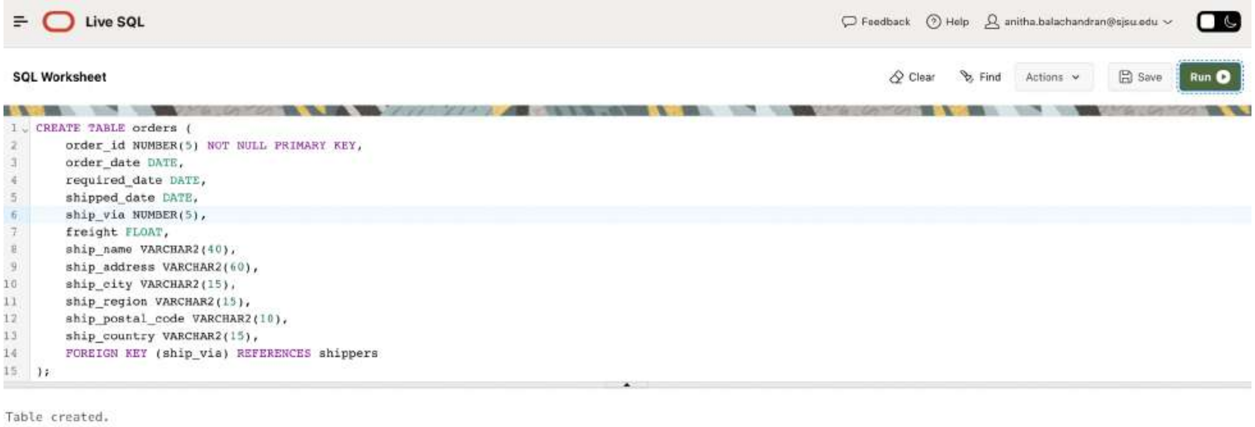
```
CREATE TABLE shippers (
    shipper_id NUMBER(5) NOT NULL PRIMARY KEY,
    company_name VARCHAR2(40) NOT NULL,
    phone VARCHAR2(24)
);
```

```
Table created.
```



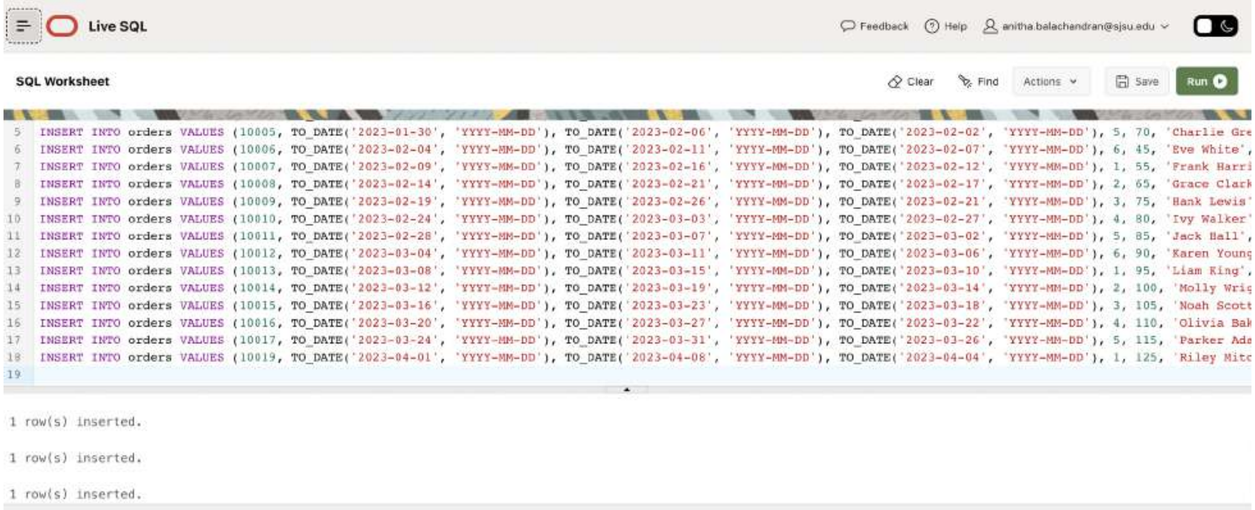
```
INSERT INTO shippers VALUES (1, 'Speedy Express', '(503) 555-9831');
INSERT INTO shippers VALUES (2, 'United Package', '(503) 555-3199');
INSERT INTO shippers VALUES (3, 'Federal Shipping', '(503) 555-9931');
INSERT INTO shippers VALUES (4, 'Alliance Shippers', '1-800-222-0451');
INSERT INTO shippers VALUES (5, 'UPS', '1-800-782-7892');
INSERT INTO shippers VALUES (6, 'DHL', '1-800-225-5345');
```

```
1 row(s) inserted.

1 row(s) inserted.

1 row(s) inserted.
```



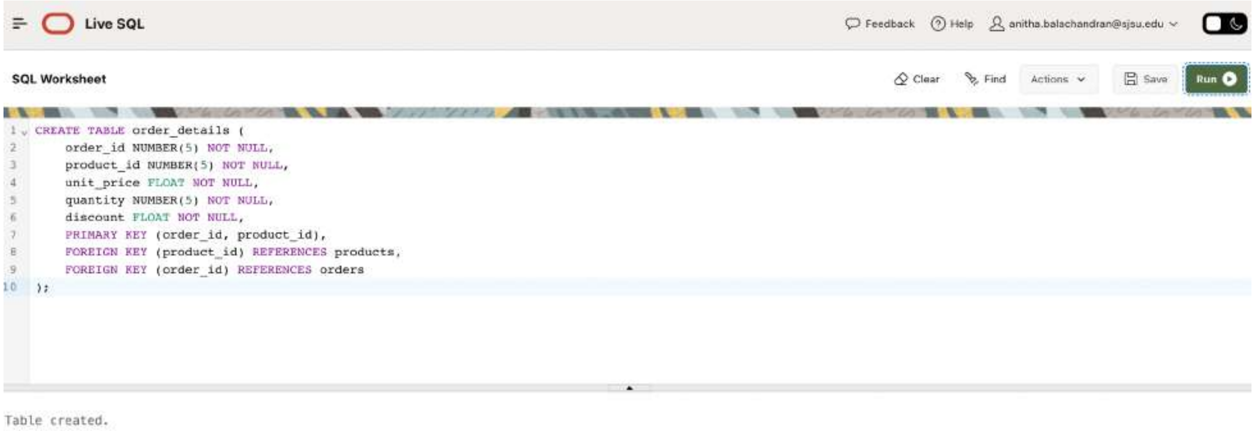
```
CREATE TABLE orders (
    order_id NUMBER(5) NOT NULL PRIMARY KEY,
    order_date DATE,
    required_date DATE,
    shipped_date DATE,
    ship_via NUMBER(5),
    freight FLOAT,
    ship_name VARCHAR2(40),
    ship_address VARCHAR2(60),
    ship_city VARCHAR2(15),
    ship_region VARCHAR2(15),
    ship_postal_code VARCHAR2(10),
    ship_country VARCHAR2(15),
    FOREIGN KEY (ship_via) REFERENCES shippers
);
```

```
Table created.
```



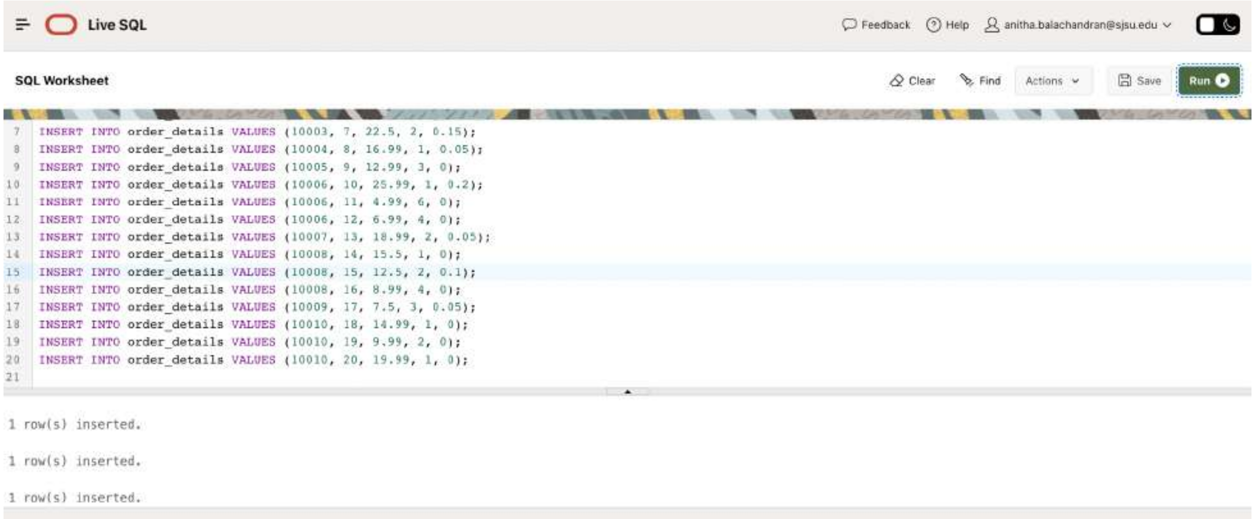
```
INSERT INTO orders VALUES (10005, TO_DATE('2023-01-30', 'YYYY-MM-DD'), TO_DATE('2023-02-06', 'YYYY-MM-DD'), TO_DATE('2023-02-02', 'YYYY-MM-DD'), 5, 70, 'Charlie Gre
INSERT INTO orders VALUES (10006, TO_DATE('2023-02-04', 'YYYY-MM-DD'), TO_DATE('2023-02-11', 'YYYY-MM-DD'), TO_DATE('2023-02-07', 'YYYY-MM-DD'), 6, 45, 'Eve White',
INSERT INTO orders VALUES (10007, TO_DATE('2023-02-09', 'YYYY-MM-DD'), TO_DATE('2023-02-16', 'YYYY-MM-DD'), TO_DATE('2023-02-12', 'YYYY-MM-DD'), 1, 55, 'Frank Harri
INSERT INTO orders VALUES (10008, TO_DATE('2023-02-14', 'YYYY-MM-DD'), TO_DATE('2023-02-21', 'YYYY-MM-DD'), TO_DATE('2023-02-17', 'YYYY-MM-DD'), 2, 65, 'Grace Clark
INSERT INTO orders VALUES (10009, TO_DATE('2023-02-19', 'YYYY-MM-DD'), TO_DATE('2023-02-26', 'YYYY-MM-DD'), TO_DATE('2023-02-21', 'YYYY-MM-DD'), 3, 75, 'Hank Lewis'
INSERT INTO orders VALUES (10010, TO_DATE('2023-02-24', 'YYYY-MM-DD'), TO_DATE('2023-03-03', 'YYYY-MM-DD'), TO_DATE('2023-02-27', 'YYYY-MM-DD'), 4, 80, 'Ivy Walker'
INSERT INTO orders VALUES (10011, TO_DATE('2023-02-28', 'YYYY-MM-DD'), TO_DATE('2023-03-07', 'YYYY-MM-DD'), TO_DATE('2023-03-02', 'YYYY-MM-DD'), 5, 85, 'Jack Hall',
INSERT INTO orders VALUES (10012, TO_DATE('2023-03-04', 'YYYY-MM-DD'), TO_DATE('2023-03-11', 'YYYY-MM-DD'), TO_DATE('2023-03-06', 'YYYY-MM-DD'), 6, 90, 'Karen Young
INSERT INTO orders VALUES (10013, TO_DATE('2023-03-08', 'YYYY-MM-DD'), TO_DATE('2023-03-15', 'YYYY-MM-DD'), TO_DATE('2023-03-10', 'YYYY-MM-DD'), 1, 95, 'Liam King',
INSERT INTO orders VALUES (10014, TO_DATE('2023-03-12', 'YYYY-MM-DD'), TO_DATE('2023-03-19', 'YYYY-MM-DD'), TO_DATE('2023-03-14', 'YYYY-MM-DD'), 2, 100, 'Molly Wrig
INSERT INTO orders VALUES (10015, TO_DATE('2023-03-16', 'YYYY-MM-DD'), TO_DATE('2023-03-23', 'YYYY-MM-DD'), TO_DATE('2023-03-18', 'YYYY-MM-DD'), 3, 105, 'Noah Scott
INSERT INTO orders VALUES (10016, TO_DATE('2023-03-20', 'YYYY-MM-DD'), TO_DATE('2023-03-27', 'YYYY-MM-DD'), TO_DATE('2023-03-22', 'YYYY-MM-DD'), 4, 110, 'Olivia Bak
INSERT INTO orders VALUES (10017, TO_DATE('2023-03-24', 'YYYY-MM-DD'), TO_DATE('2023-03-31', 'YYYY-MM-DD'), TO_DATE('2023-03-26', 'YYYY-MM-DD'), 5, 115, 'Parker Ada
INSERT INTO orders VALUES (10019, TO_DATE('2023-04-01', 'YYYY-MM-DD'), TO_DATE('2023-04-08', 'YYYY-MM-DD'), TO_DATE('2023-04-04', 'YYYY-MM-DD'), 1, 125, 'Riley Mitc
```

```
1 row(s) inserted.

1 row(s) inserted.

1 row(s) inserted.
```



```
CREATE TABLE order_details (
    order_id NUMBER(5) NOT NULL,
    product_id NUMBER(5) NOT NULL,
    unit_price FLOAT NOT NULL,
    quantity NUMBER(5) NOT NULL,
    discount FLOAT NOT NULL,
    PRIMARY KEY (order_id, product_id),
    FOREIGN KEY (product_id) REFERENCES products,
    FOREIGN KEY (order_id) REFERENCES orders
);
```

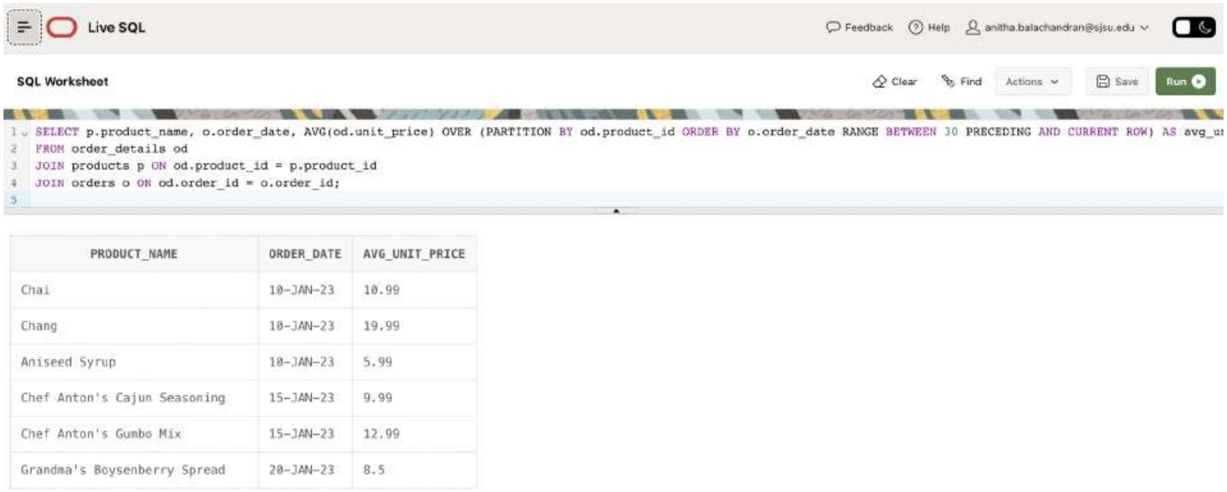
```
Table created.
```



```
INSERT INTO order_details VALUES (10003, 7, 22.5, 2, 0.15);
INSERT INTO order_details VALUES (10004, 8, 16.99, 1, 0.05);
INSERT INTO order_details VALUES (10005, 9, 12.99, 3, 0);
INSERT INTO order_details VALUES (10006, 10, 25.99, 1, 0.2);
INSERT INTO order_details VALUES (10006, 11, 4.99, 6, 0);
INSERT INTO order_details VALUES (10006, 12, 6.99, 4, 0);
INSERT INTO order_details VALUES (10007, 13, 18.99, 2, 0.05);
INSERT INTO order_details VALUES (10008, 14, 15.5, 1, 0);
INSERT INTO order_details VALUES (10008, 15, 12.5, 2, 0.1);
INSERT INTO order_details VALUES (10008, 16, 8.99, 4, 0);
INSERT INTO order_details VALUES (10009, 17, 7.5, 3, 0.05);
INSERT INTO order_details VALUES (10010, 18, 14.99, 1, 0);
INSERT INTO order_details VALUES (10010, 19, 9.99, 2, 0);
INSERT INTO order_details VALUES (10010, 20, 19.99, 1, 0);
```

```
1 row(s) inserted.

1 row(s) inserted.

1 row(s) inserted.
```

Data Analysis using SQL Queries and Analytics Functions:

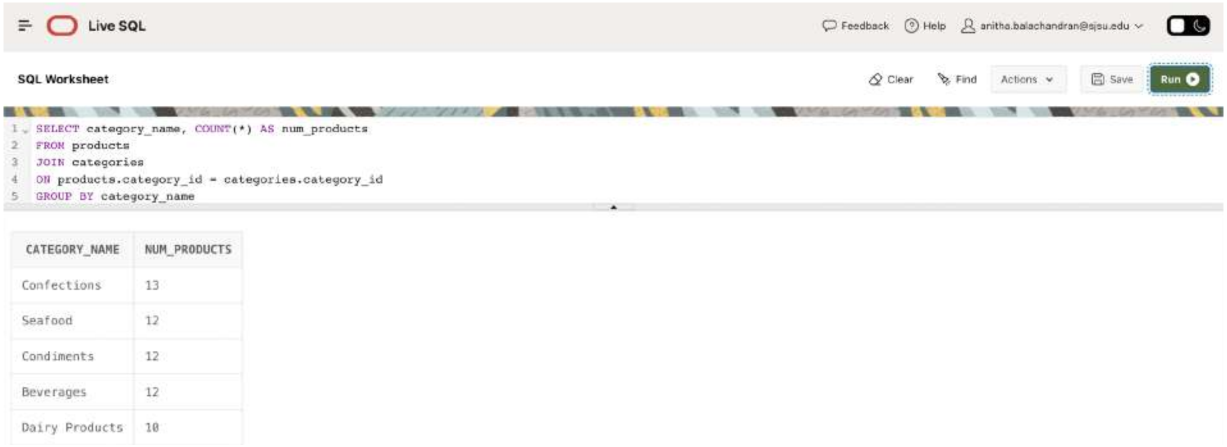


```
SELECT p.product_name, o.order_date, AVG(od.unit_price) OVER (PARTITION BY od.product_id ORDER BY o.order_date RANGE BETWEEN 30 PRECEDING AND CURRENT ROW) AS avg_u
FROM order_details od
JOIN products p ON od.product_id = p.product_id
JOIN orders o ON od.order_id = o.order_id;
```

PRODUCT_NAME	ORDER_DATE	AVG_UNIT_PRICE
Chai	10-JAN-23	10.99
Chang	10-JAN-23	19.99
Aniseed Syrup	10-JAN-23	5.99
Chef Anton's Cajun Seasoning	15-JAN-23	9.99
Chef Anton's Gumbo Mix	15-JAN-23	12.99
Grandma's Boysenberry Spread	20-JAN-23	8.5

SELECT p.product_name, o.order_date, AVG(od.unit_price) OVER (PARTITION BY od.product_id ORDER BY o.order_date RANGE BETWEEN 30 PRECEDING AND CURRENT ROW) AS avg_unit_price
FROM order_details od
JOIN products p ON od.product_id = p.product_id
JOIN orders o ON od.order_id = o.order_id;

- ➢ This query retrieves the average unit price for each product in a sliding window of 30 days.
- ➢ The query joins the order_details, products, and orders tables, and uses the PARTITION BY clause to group the data by product ID, and the ORDER BY clause to order the data by order date.
- ➢ The AVG function is then used with the OVER clause to calculate the average unit price for each row within the sliding window of 30 days.
- ➢ The result set includes the product name, order date, and the average unit price for each row. This query can help identify trends in unit prices over time for different products and can be useful for pricing strategies and inventory management.



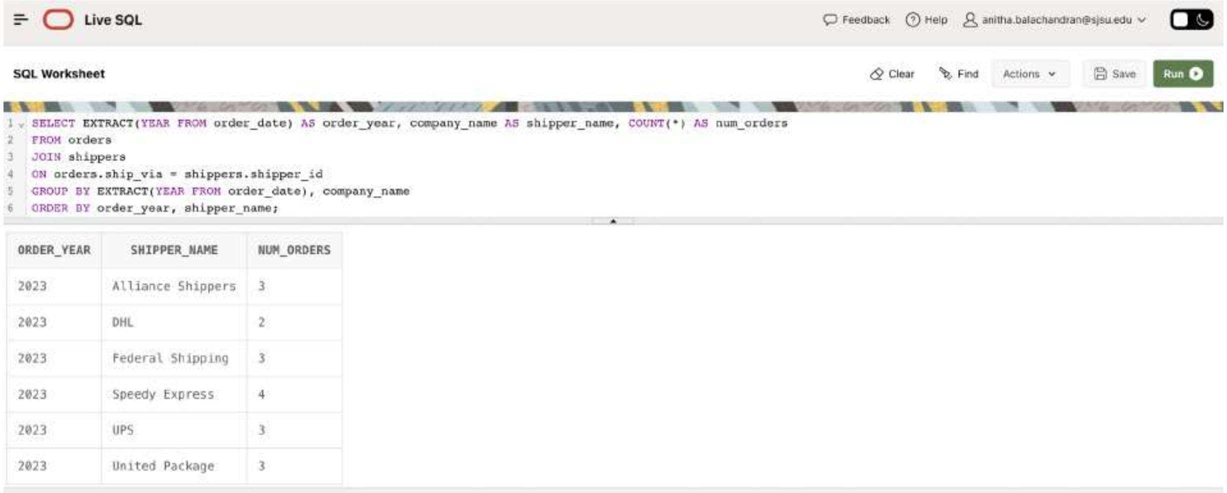
```
SELECT category_name, COUNT(*) AS num_products
FROM products
JOIN categories
ON products.category_id = categories.category_id
GROUP BY category_name
```

CATEGORY_NAME	NUM_PRODUCTS
Confections	13
Seafood	12
Condiments	12
Beverages	12
Dairy Products	10

SELECT category_name, COUNT(*) AS num_products

```
FROM products
JOIN categories
ON products.category_id = categories.category_id
GROUP BY category_name
```

- ➢ The query joins the "products" and "categories" tables on their "category_id" columns and groups the result by "category_name". The COUNT(*) function is used to count the number of products in each category. The results are then ordered by the number of products in descending order, and the FETCH FIRST clause limits the result to the first 5 rows.
- ➢ By running this query, we can gain insights on which categories have the most products, which may help in identifying popular product categories and make informed business decisions.



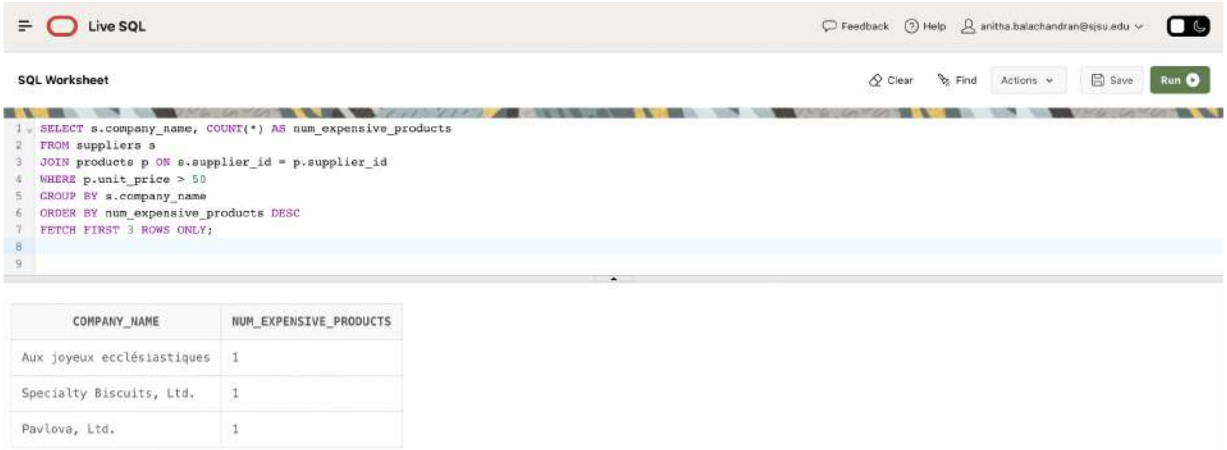
```
SELECT EXTRACT(YEAR FROM order_date) AS order_year, company_name AS shipper_name, COUNT(*) AS num_orders
FROM orders
JOIN shippers
ON orders.ship_via = shippers.shipper_id
GROUP BY EXTRACT(YEAR FROM order_date), company_name
ORDER BY order_year, shipper_name;
```

ORDER_YEAR	SHIPPER_NAME	NUM_ORDERS
2023	Alliance Shippers	3
2023	DHL	2
2023	Federal Shipping	3
2023	Speedy Express	4
2023	UPS	3
2023	United Package	3

```
SELECT EXTRACT(YEAR FROM order_date) AS order_year, company_name AS
shipper_name, COUNT(*) AS num_orders
FROM orders
JOIN shippers
ON orders.ship_via = shippers.shipper_id
GROUP BY EXTRACT(YEAR FROM order_date), company_name
ORDER BY order_year, shipper_name;
```

- ➢ This query returns the number of orders for each year and each shipper in the database.
- ➢ The query first uses the EXTRACT() function to extract the year from the order_date column and renames the result as order_year.
- ➢ It also selects the company_name column from the shippers table and renames it as shipper_name. Then, it counts the number of orders for each year and each shipper by grouping the results based on order_year and shipper_name using the GROUP BY clause.
- ➢ Finally, it sorts the result by order_year and shipper_name using the ORDER BY clause.
- ➢ By using this query, we can gain insights into the number of orders each shipper has received each year. This information can help businesses to make decisions regarding

which shippers to use for their orders based on their past performance.

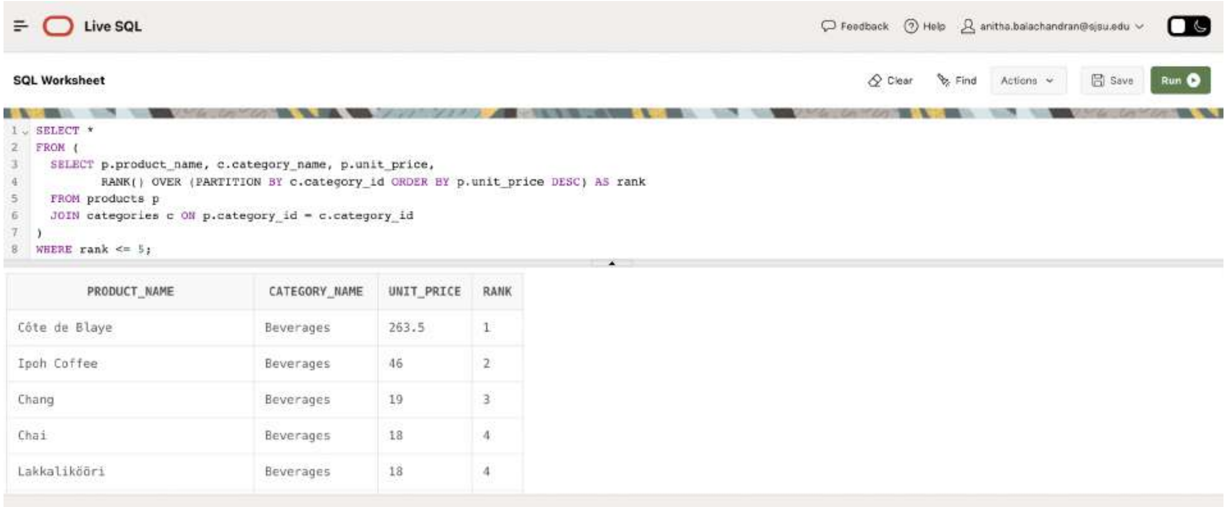


```
SELECT s.company_name, COUNT(*) AS num_expensive_products
FROM suppliers s
JOIN products p ON s.supplier_id = p.supplier_id
WHERE p.unit_price > 50
GROUP BY s.company_name
ORDER BY num_expensive_products DESC
FETCH FIRST 3 ROWS ONLY;
```

COMPANY_NAME	NUM_EXPENSIVE_PRODUCTS
Aux joyeux ecclésiastiques	1
Specialty Biscuits, Ltd.	1
Pavlova, Ltd.	1

SELECT s.company_name, COUNT(*) AS num_expensive_products
FROM suppliers s
JOIN products p ON s.supplier_id = p.supplier_id
WHERE p.unit_price > 50
GROUP BY s.company_name
ORDER BY num_expensive_products DESC
FETCH FIRST 3 ROWS ONLY;

- ➢ This query retrieves the names of the top 3 suppliers that provide the highest number of products with a unit price greater than 50.
- ➢ The query first joins the "suppliers" table with the "products" table on the "supplier_id" field. Then, it filters the products that have a "unit_price" greater than 50 using the "WHERE" clause.
- ➢ After that, the query groups the results by "company_name" from the "suppliers" table and uses the "COUNT" function to count the number of products with a unit price greater than 50 for each supplier.
- ➢ Finally, the query orders the results in descending order by the number of expensive products and fetches only the top 3 results using the "FETCH FIRST" clause.
- ➢ The insights we can gain from this query include identifying the top 3 suppliers that provide expensive products and the number of such products they offer which can be useful for businesses to make informed decisions about their suppliers and pricing strategies.

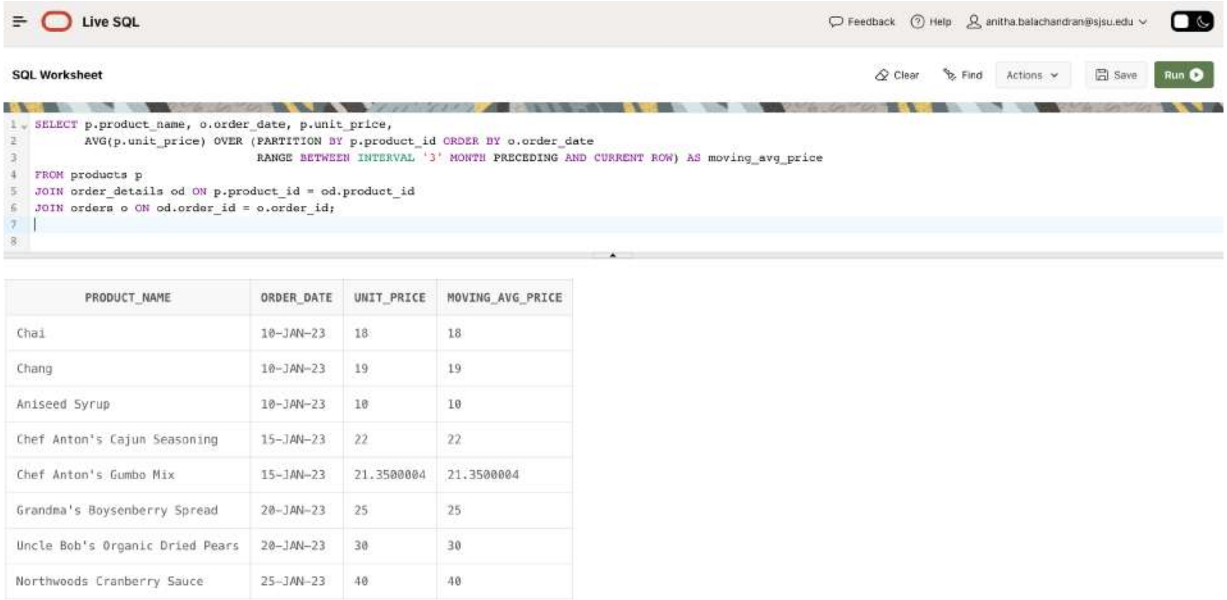


```
SELECT *
FROM (
  SELECT p.product_name, c.category_name, p.unit_price,
         RANK() OVER (PARTITION BY c.category_id ORDER BY p.unit_price DESC) AS rank
  FROM products p
  JOIN categories c ON p.category_id = c.category_id
)
WHERE rank <= 5;
```

PRODUCT_NAME	CATEGORY_NAME	UNIT_PRICE	RANK
Côte de Blaye	Beverages	263.5	1
Ipoh Coffee	Beverages	46	2
Chang	Beverages	19	3
Chai	Beverages	18	4
Lakkalikööri	Beverages	18	4

```
SELECT *
FROM (
  SELECT p.product_name, c.category_name, p.unit_price,
         RANK() OVER (PARTITION BY c.category_id ORDER BY p.unit_price DESC) AS
rank
  FROM products p
  JOIN categories c ON p.category_id = c.category_id
)
WHERE rank <= 5;
```

- This query retrieves the top 5 most expensive products in each category, along with their category names and product names.
- The query uses the RANK() analytic function to rank the products within each category based on their unit price, in descending order.
- The PARTITION BY clause is used to create partitions (or groups) within the data based on the category ID, and the ORDER BY clause is used to order the products within each partition by their unit price.
- The insights gained from this query could be useful in pricing strategies and product positioning.

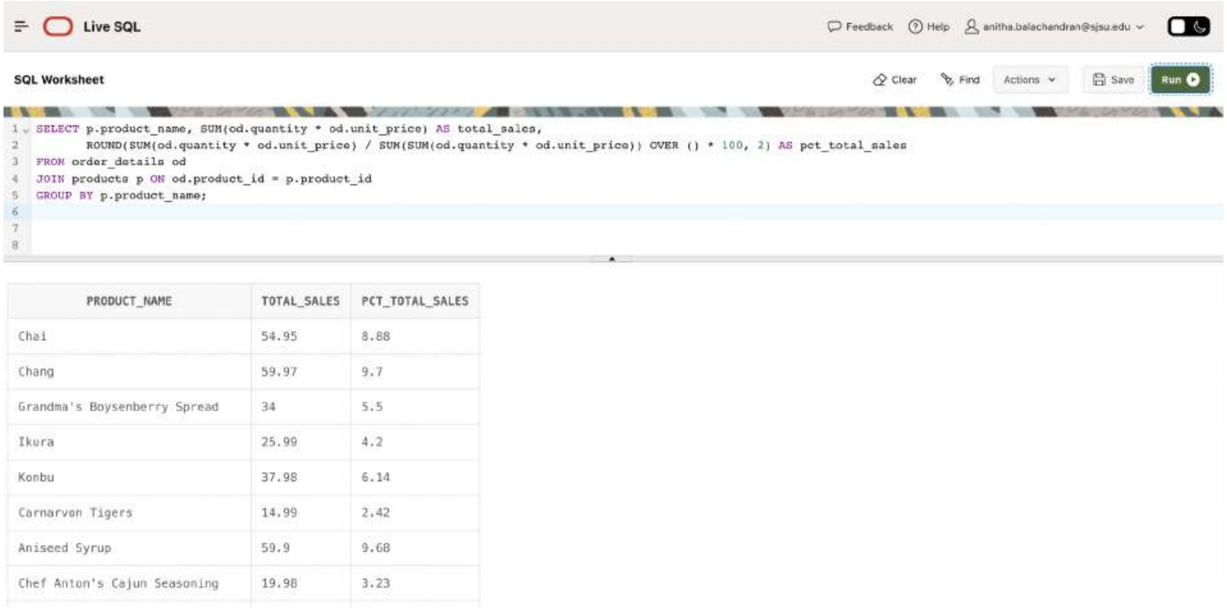


```
SELECT p.product_name, o.order_date, p.unit_price,
       AVG(p.unit_price) OVER (PARTITION BY p.product_id ORDER BY o.order_date
                               RANGE BETWEEN INTERVAL '3' MONTH PRECEDING AND CURRENT ROW) AS moving_avg_price
FROM products p
JOIN order_details od ON p.product_id = od.product_id
JOIN orders o ON od.order_id = o.order_id;
```

PRODUCT_NAME	ORDER_DATE	UNIT_PRICE	MOVING_AVG_PRICE
Chai	10-JAN-23	18	18
Chang	10-JAN-23	19	19
Aniseed Syrup	10-JAN-23	10	10
Chef Anton's Cajun Seasoning	15-JAN-23	22	22
Chef Anton's Gumbo Mix	15-JAN-23	21.3500004	21.3500004
Grandma's Boysenberry Spread	20-JAN-23	25	25
Uncle Bob's Organic Dried Pears	20-JAN-23	30	30
Northwoods Cranberry Sauce	25-JAN-23	40	40

```
SELECT p.product_name, o.order_date, p.unit_price,
    AVG(p.unit_price) OVER (PARTITION BY p.product_id ORDER BY o.order_date
                    RANGE BETWEEN INTERVAL '3' MONTH PRECEDING AND
CURRENT ROW) AS moving_avg_price
FROM products p
JOIN order_details od ON p.product_id = od.product_id
JOIN orders o ON od.order_id = o.order_id;
```

- ➢ This query retrieves the name of the product, the date of the order, and the unit price of the product. It also calculates the moving average price of the product for the last three months.
- ➢ The analytic function used in this query is AVG, which calculates the average of a set of values. It is used in combination with the OVER clause, which specifies the partitioning and ordering of the data. In this case, the data is partitioned by product_id and ordered by order_date.
- ➢ The RANGE BETWEEN INTERVAL '3' MONTH PRECEDING AND CURRENT ROW clause specifies the range of rows to include in the calculation of the moving average price, which includes the current row and the preceding three months.
- ➢ The query provides insights into the price trends of products over time. It can be used to identify products with fluctuating prices or products that have a steady price trend. The moving average price can also be used to forecast future prices and plan pricing strategies accordingly.



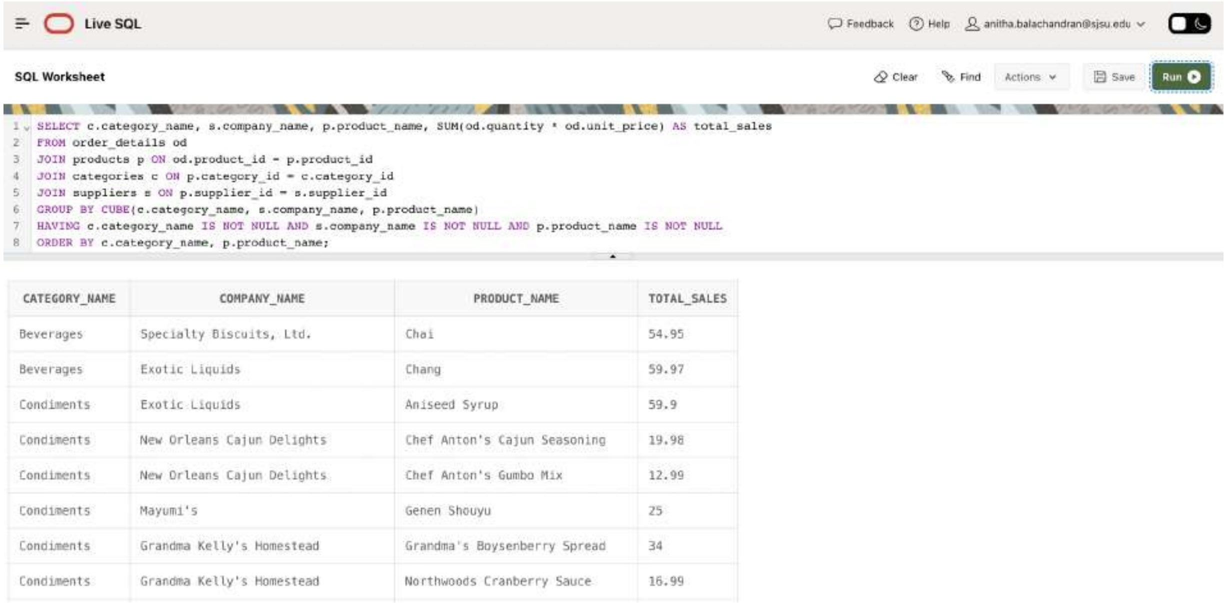
```sql
SELECT p.product_name, SUM(od.quantity * od.unit_price) AS total_sales,
       ROUND(SUM(od.quantity * od.unit_price) / SUM(SUM(od.quantity * od.unit_price)) OVER () * 100, 2) AS pct_total_sales
FROM order_details od
JOIN products p ON od.product_id = p.product_id
GROUP BY p.product_name;
```

PRODUCT_NAME	TOTAL_SALES	PCT_TOTAL_SALES
Chai	54.95	8.88
Chang	59.97	9.7
Grandma's Boysenberry Spread	34	5.5
Ikura	25.99	4.2
Konbu	37.98	6.14
Carnarvon Tigers	14.99	2.42
Aniseed Syrup	59.9	9.68
Chef Anton's Cajun Seasoning	19.98	3.23

```sql
SELECT p.product_name, SUM(od.quantity * od.unit_price) AS total_sales,
      ROUND(SUM(od.quantity * od.unit_price) / SUM(SUM(od.quantity * od.unit_price))
OVER () * 100, 2) AS pct_total_sales
FROM order_details od
JOIN products p ON od.product_id = p.product_id
GROUP BY p.product_name;
```

- ➢ This query calculates the total sales and percentage of total sales for each product in the order_details table, joined with the products table to get the product name.
- ➢ The main analytical function used in this query is the SUM() function, which calculates the total sales for each product by multiplying the quantity and unit price columns. The GROUP BY clause is used to group the results by product name.
- ➢ The other important analytical function used in this query is the SUM() function with the OVER() clause, which calculates the total sales for all products in the result set. This is then used to calculate the percentage of total sales for each product by dividing each product's total sales by the total sales for all products and multiplying by 100. The ROUND() function is used to round the percentage to two decimal places.
- ➢ Insights gained from this query could include identifying which products have the highest total sales and what percentage of the total sales each product represents. This information could be useful for product performance analysis and strategic decision-making regarding product offerings and promotions.

1. b) Using CUBE and ROLLUP clauses for Aggregation in Oracle Database Analytics:

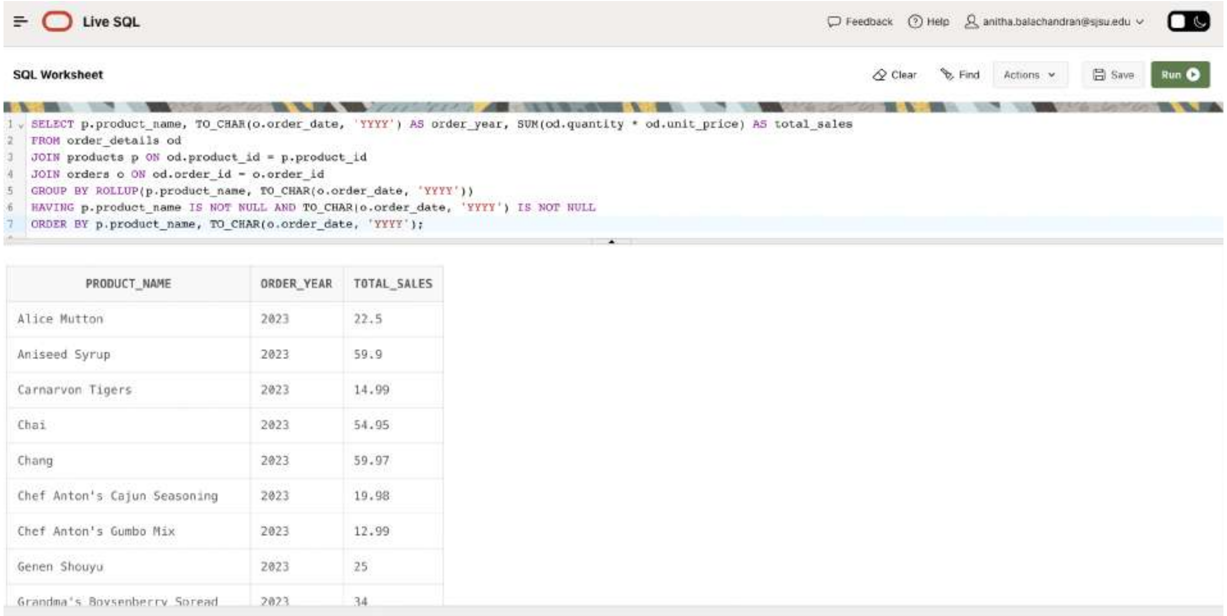


```
SELECT c.category_name, s.company_name, p.product_name, SUM(od.quantity * od.unit_price) AS total_sales
FROM order_details od
JOIN products p ON od.product_id = p.product_id
JOIN categories c ON p.category_id = c.category_id
JOIN suppliers s ON p.supplier_id = s.supplier_id
GROUP BY CUBE(c.category_name, s.company_name, p.product_name)
HAVING c.category_name IS NOT NULL AND s.company_name IS NOT NULL AND p.product_name IS NOT NULL
ORDER BY c.category_name, p.product_name;
```

CATEGORY_NAME	COMPANY_NAME	PRODUCT_NAME	TOTAL_SALES
Beverages	Specialty Biscuits, Ltd.	Chai	54.95
Beverages	Exotic Liquids	Chang	59.97
Condiments	Exotic Liquids	Aniseed Syrup	59.9
Condiments	New Orleans Cajun Delights	Chef Anton's Cajun Seasoning	19.98
Condiments	New Orleans Cajun Delights	Chef Anton's Gumbo Mix	12.99
Condiments	Mayumi's	Genen Shouyu	25
Condiments	Grandma Kelly's Homestead	Grandma's Boysenberry Spread	34
Condiments	Grandma Kelly's Homestead	Northwoods Cranberry Sauce	16.99

```
SELECT c.category_name, s.company_name, p.product_name, SUM(od.quantity *
od.unit_price) AS total_sales
FROM order_details od
JOIN products p ON od.product_id = p.product_id
JOIN categories c ON p.category_id = c.category_id
JOIN suppliers s ON p.supplier_id = s.supplier_id
GROUP BY CUBE(c.category_name, s.company_name, p.product_name)
HAVING c.category_name IS NOT NULL AND s.company_name IS NOT NULL AND
p.product_name IS NOT NULL
ORDER BY c.category_name, p.product_name;
```

- This query joins the order_details, products, categories, and suppliers tables and calculates the total sales for each combination of category, supplier, and product.
- It uses the GROUP BY clause with the CUBE function to generate all possible combinations of these three dimensions. The SUM function calculates the total sales for each combination of dimensions.
- The HAVING clause is used to filter out any rows that have a null value in any of the dimensions, as the CUBE function generates null values for all combinations that are not explicitly present in the data.
- The ORDER BY clause orders the results by category name and product name.
- The insights gained from this query include identifying the top-selling products by category and supplier, as well as analyzing the overall sales performance of each category and supplier.

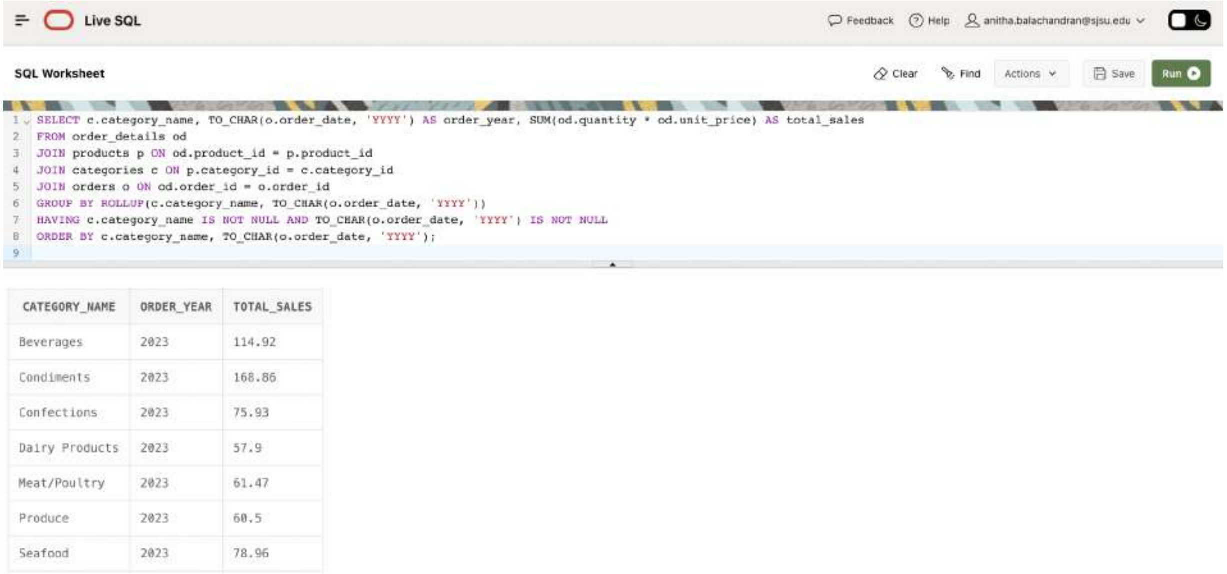


```sql
SELECT p.product_name, TO_CHAR(o.order_date, 'YYYY') AS order_year, SUM(od.quantity * od.unit_price) AS total_sales
FROM order_details od
JOIN products p ON od.product_id = p.product_id
JOIN orders o ON od.order_id = o.order_id
GROUP BY ROLLUP(p.product_name, TO_CHAR(o.order_date, 'YYYY'))
HAVING p.product_name IS NOT NULL AND TO_CHAR(o.order_date, 'YYYY') IS NOT NULL
ORDER BY p.product_name, TO_CHAR(o.order_date, 'YYYY');
```

PRODUCT_NAME	ORDER_YEAR	TOTAL_SALES
Alice Mutton	2023	22.5
Aniseed Syrup	2023	59.9
Carnarvon Tigers	2023	14.99
Chai	2023	54.95
Chang	2023	59.97
Chef Anton's Cajun Seasoning	2023	19.98
Chef Anton's Gumbo Mix	2023	12.99
Genen Shouyu	2023	25
Grandma's Boysenberry Spread	2023	34

```
SELECT p.product_name, TO_CHAR(o.order_date, 'YYYY') AS order_year,
SUM(od.quantity * od.unit_price) AS total_sales
FROM order_details od
JOIN products p ON od.product_id = p.product_id
JOIN orders o ON od.order_id = o.order_id
GROUP BY ROLLUP(p.product_name, TO_CHAR(o.order_date, 'YYYY'))
HAVING p.product_name IS NOT NULL AND TO_CHAR(o.order_date, 'YYYY') IS NOT
NULL
ORDER BY p.product_name, TO_CHAR(o.order_date, 'YYYY');
```

- The query is calculating the total sales for each product by year, using data from the "order_details", "products", and "orders" tables.
- It is using the ROLLUP function to generate subtotals for each level of the hierarchy (product and year).
- The HAVING clause filters out any rows with NULL values in the product_name or order_year columns. The results are ordered by product_name and order_year.
- Insights that can be gained from this query include identifying the top selling products by year, comparing sales trends between different products, and identifying any gaps in sales data (i.e. years where no sales were made for a particular product).



```sql
SELECT c.category_name, TO_CHAR(o.order_date, 'YYYY') AS order_year, SUM(od.quantity * od.unit_price) AS total_sales
FROM order_details od
JOIN products p ON od.product_id = p.product_id
JOIN categories c ON p.category_id = c.category_id
JOIN orders o ON od.order_id = o.order_id
GROUP BY ROLLUP(c.category_name, TO_CHAR(o.order_date, 'YYYY'))
HAVING c.category_name IS NOT NULL AND TO_CHAR(o.order_date, 'YYYY') IS NOT NULL
ORDER BY c.category_name, TO_CHAR(o.order_date, 'YYYY');
```

CATEGORY_NAME	ORDER_YEAR	TOTAL_SALES
Beverages	2023	114.92
Condiments	2023	168.86
Confections	2023	75.93
Dairy Products	2023	57.9
Meat/Poultry	2023	61.47
Produce	2023	60.5
Seafood	2023	78.96

```sql
SELECT c.category_name, TO_CHAR(o.order_date, 'YYYY') AS order_year,
SUM(od.quantity * od.unit_price) AS total_sales
FROM order_details od
JOIN products p ON od.product_id = p.product_id
JOIN categories c ON p.category_id = c.category_id
JOIN orders o ON od.order_id = o.order_id
GROUP BY ROLLUP(c.category_name, TO_CHAR(o.order_date, 'YYYY'))
HAVING c.category_name IS NOT NULL AND TO_CHAR(o.order_date, 'YYYY') IS NOT
NULL
ORDER BY c.category_name, TO_CHAR(o.order_date, 'YYYY');
```

- ➢ This query joins the tables order_details, products, categories, and orders, and groups the data by the category_name and order_date dimensions using the ROLLUP function in the GROUP BY clause.
- ➢ It then calculates the total sales for each combination of the category_name and order_date dimensions using the SUM function applied to the product of the quantity and unit_price columns from the order_details table.
- ➢ The HAVING clause is used to filter out rows where either category_name or order_date is null, and the results are sorted by category_name and order_date.
- ➢ By using the ROLLUP function, the query generates subtotals and grand totals for each combination of the category_name and order_date dimensions.
- ➢ The query provides insights on the total sales for each category in each year, as well as the total sales for each category and for all categories in each year. This can help identify trends and patterns in sales over time, as well as the relative performance of each category in terms of sales.

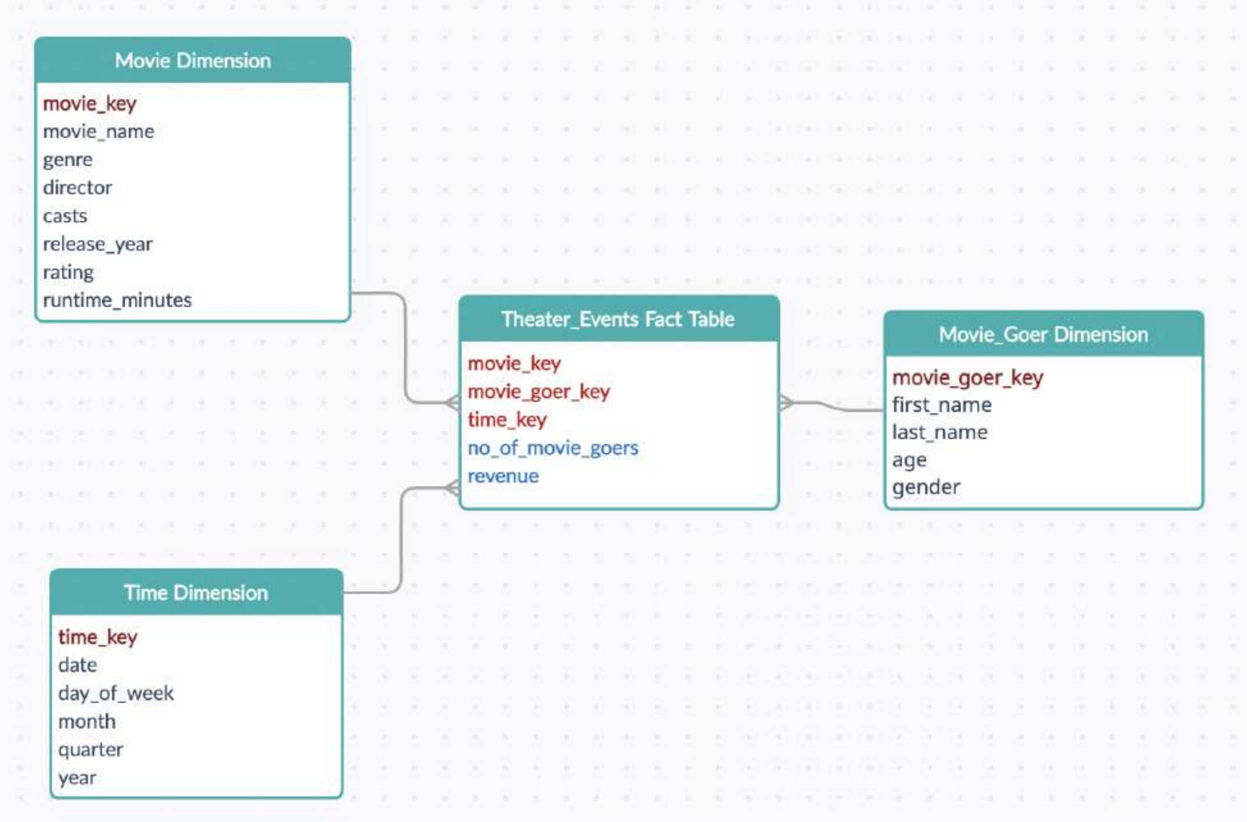
Conclusion:

In conclusion, this project successfully demonstrated the use of Oracle SQL to create a database system for storing and managing e-commerce sales data. The project also showcased the use of CUBE and ROLLUP clauses for generating useful aggregate information from the tables. Through data analysis using various analytics functions, valuable insights were gained, such as the top-selling products and average order value. Overall, the project achieved its goal of providing a comprehensive solution for e-commerce sales data management using Oracle Database Analytics in the Cloud.

===================xxxxxxx===================

2.

a) Simple Star Schema for ABC Theaters Datawarehouse



The **Theater Fact Table** in the ABC Theaters data warehouse contains two measures: "no_of_movie_goers" and "revenue", which are numerical values that can be aggregated or analyzed across the dimensions of the movie, movie-goer, and time. It also includes foreign keys that connect it to the surrounding Dimensions, enabling analysts to slice and dice the measures by different dimensions to gain insights and generate reports.

The **Movie dimension** contains information about the movies shown at the theater, such as the movie ID, name of the movie, genre, the director, release year, rating, and runtime.

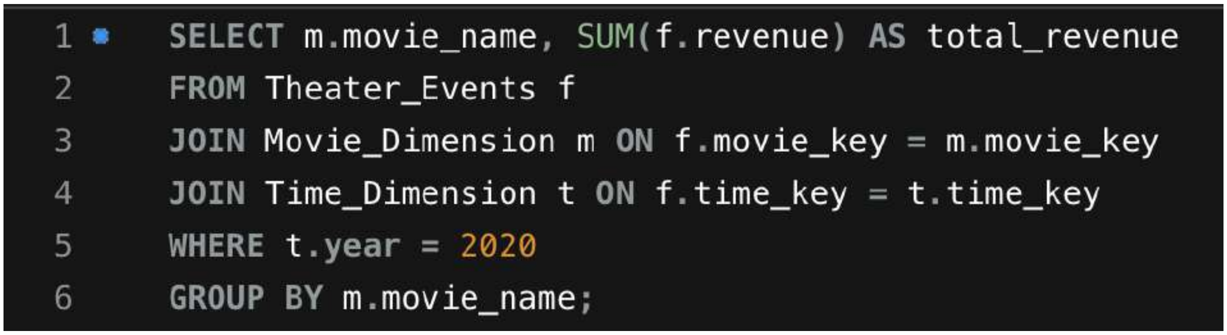
The **Time dimension** contains information about when the events took place, including time ID, the date, day of the week, month, quarter, and year.

The **Movie-goer dimension** contains information about the customers who attended the movies, including their ID, first and last names, age and gender.

By querying the above data warehouse, we can generate meaningful summaries that help us understand trends and patterns in the theater's activity.

b) The below SQL JOIN query can be used to retrieve the total revenue collected through ticket sales by each movie in the pandemic year, '2020'.

Assuming, The 'revenue' in the 'Theater_Event' Fact table represents the actual amount of money collected from ticket sales for a particular screening.
The 'no_of_movie_goers' in the 'Theater_Event' Fact table represents the total number of people who attended the screening regardless of the number of tickets sold (number of movie-goers may not necessarily be equal to the number of tickets sold, if say some people received free tickets or used coupons to enter the screening).
Hence using 'revenue' metric directly would give us an accurate representation of the total revenue generated from ticket sales, which is shown below.



```
SELECT m.movie_name, SUM(f.revenue) AS total_revenue
FROM Theater_Events f
JOIN Movie_Dimension m ON f.movie_key = m.movie_key
JOIN Time_Dimension t ON f.time_key = t.time_key
WHERE t.year = 2020
GROUP BY m.movie_name;
```

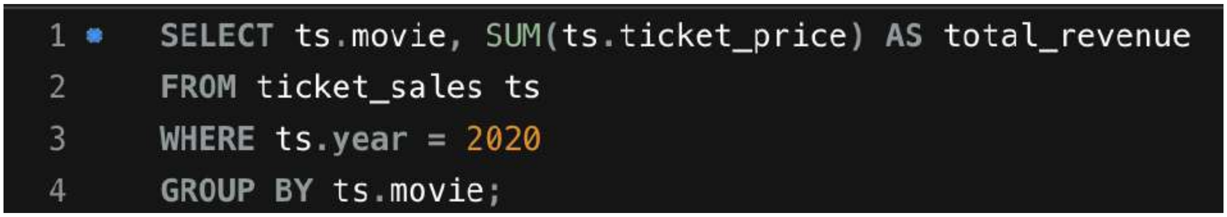
1) **SELECT m.movie_name, SUM(f.revenue) AS total_revenue:** It retrieves the movie name from the movie dimension table and the revenue from the fact table, and uses the SUM function to calculate the total revenue for each movie.

2) **FROM Theater_Event f JOIN Movie_Dimension m ON f.movie_key = m.movie_key JOIN Time_Dimension t ON f.time_key = t.time_key:** Specifies the Fact table 'Theater_Event' we are retrieving data from, while the two JOIN clauses specify the Dimension tables ('Movie_Dimension' and 'Time_Dimension') we are joining with the Fact table. The join condition for each JOIN clause matches the key column from the Fact table (movie_key or time_key) with the corresponding key column in the Dimension table (movie_key or time_key).

3) **WHERE t.year = 2020:** Specifies the conditions that the rows in the tables must meet in order to be included in the query result. This condition ensures that we only consider sales data from the year 2020 from Time_Dimension

4) **GROUP BY m.movie_name:** Groups the data by movie name so that we can calculate the total revenue for each movie separately.

c) **In RDBMS, The below SQL query can be to retrieve the total revenue collected through ticket sales by each movie in the pandemic year, 2020, based on the ticket_sales table.**



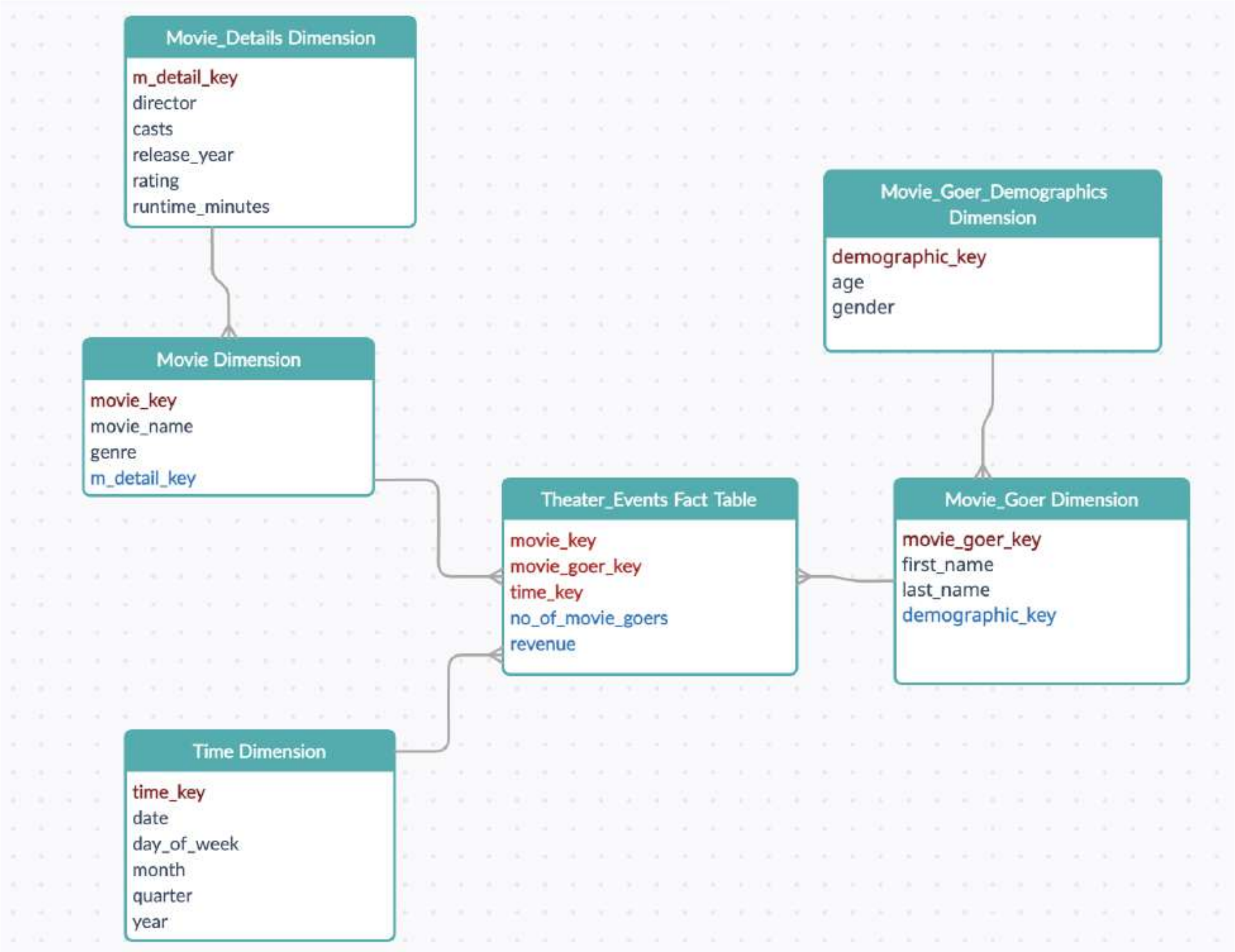
```sql
SELECT ts.movie, SUM(ts.ticket_price) AS total_revenue
FROM ticket_sales ts
WHERE ts.year = 2020
GROUP BY ts.movie;
```

1) **SELECT movie, SUM(ticket_price) AS total_revenue:** It retrieves the movie name from the movie column in the ticket_sales table and the total revenue from the ticket_price column in the same table and uses the SUM function to add up the ticket price for each movie.
2) **FROM ticket_sales ts:** Specifies the tables to be used in the query. The ticket_sales table is aliased as ‘ts’.
3) **WHERE ts.year = 2020:** Specifies the conditions that the rows in the tables must meet in order to be included in the query result. This condition ensures that we only consider sales data from the year 2020 from the ticket_sales table.
4) **GROUP BY ts.movie:** Groups the data by movie name so that we can calculate the total revenue for each movie separately.

d) **Simple Snowflake Schema for ABC Theaters Datawarehouse:**

- ➢ The Snowflake schema is a variant of the Star schema where one or more dimensions are normalized into multiple tables to improve performance and reduce redundancy.
- ➢ In the Star schema, we could normalize the **Movie_Goer** dimension into another table called **Movie_Goer_Demographics** with demographic_key, age, and gender.
- ➢ Additionally, we could normalize the **Movie** dimension and create a new table called Movie_Details with a m_detail_key attribute to hold movie information that is not frequently used, such as director, cast, release year, rating, and runtime minutes, to create a Snowflake schema.

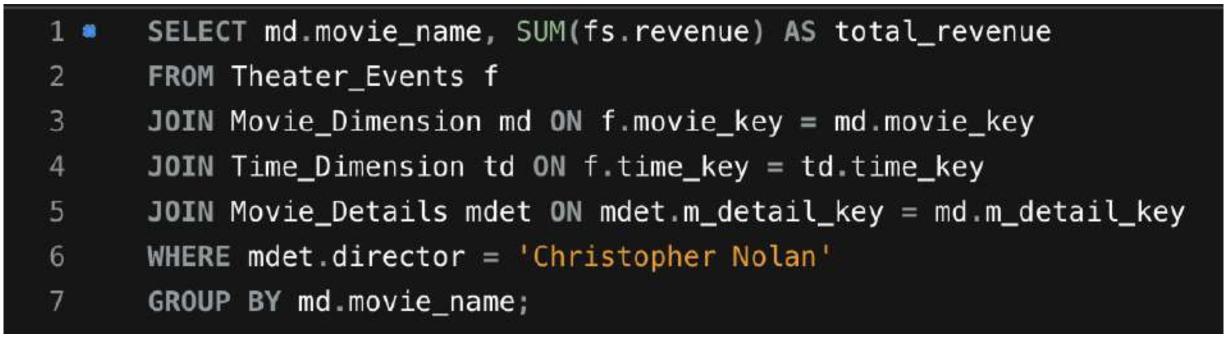
The resulting Snowflake schema is shown below:



Below are some of the SQL JOIN queries that can be used to output the additional summary information from the above Snowflake schema.

1) **The below SQL JOIN query can be used to retrieve the total revenue generated by movies directed by a specific director.**

The below query shows total revenue generated by each movie directed by Christopher Nolan, by joining the Theater_Events Fact table with the Movie_Dimension, Time_Dimension, and Movie_Details table.

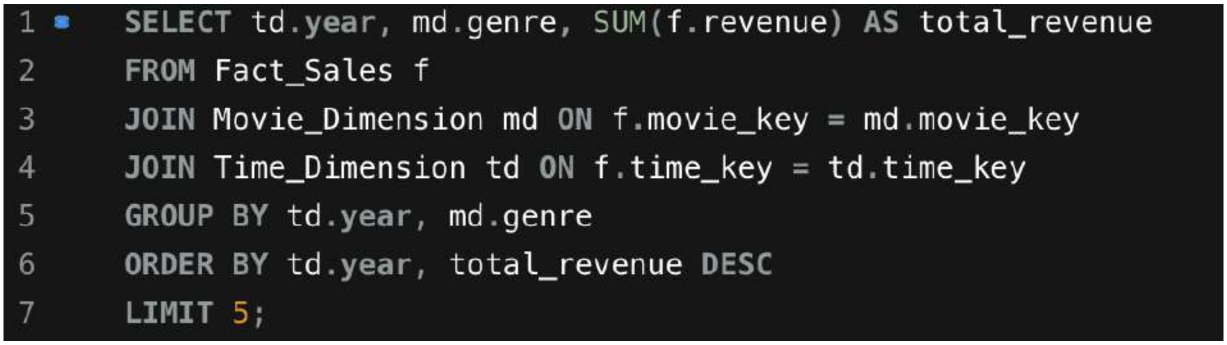


```sql
SELECT md.movie_name, SUM(fs.revenue) AS total_revenue
FROM Theater_Events f
JOIN Movie_Dimension md ON f.movie_key = md.movie_key
JOIN Time_Dimension td ON f.time_key = td.time_key
JOIN Movie_Details mdet ON mdet.m_detail_key = md.m_detail_key
WHERE mdet.director = 'Christopher Nolan'
GROUP BY md.movie_name;
```

1) **SELECT md.movie_name, SUM(fs.revenue) AS total_revenue:** It retrieves the movie name from the Movie_Dimension table and the total revenue generated from each movie from the Theater_Events Fact table and uses the SUM function to calculate the total revenue generated for each movie.
2) **FROM Theater_Events f JOIN Movie_Dimension md ON f.movie_key = md.movie_key JOIN Time_Dimension td ON f.time_key = td.time_key JOIN Movie_Details mdet ON mdet.m_detail_key = md.m_detail_key:** Specifies the tables we are retrieving data from and how they are joined. The Theater_Events Fact table is joined with the Movie_Dimension table on the movie_key foreign key, the Time_Dimension table is joined with Theater_Events Fact table on the time_key foreign key, and the Movie_Details table is joined with the Movie_Dimension table on the m_detail_key foreign key.
3) **WHERE mdet.director = 'Christopher Nolan':** Filters the results to only include movies directed by Christopher Nolan.
4) **GROUP BY md.movie_name:** Groups the results by movie name from the Movie_Dimension table. It calculates the total revenue generated by each movie, allowing us to see which of Christopher Nolan's movies generated the most revenue.

2) **The below SQL JOIN query can be used to retrieve the top 5 movie genres by total revenue for each year and the output can help ABC Theaters make informed decisions about which movies to show, which genres to prioritize, and how to allocate resources to maximize revenue and profitability.**

The below query retrieves data from the Theater_Events Fact table, Movie_Dimension, and Time_Dimension tables to find the top 5 movie genres by total revenue for each year.



```sql
SELECT td.year, md.genre, SUM(f.revenue) AS total_revenue
FROM Fact_Sales f
JOIN Movie_Dimension md ON f.movie_key = md.movie_key
JOIN Time_Dimension td ON f.time_key = td.time_key
GROUP BY td.year, md.genre
ORDER BY td.year, total_revenue DESC
LIMIT 5;
```

1. **SELECT td.year, md.genre, SUM(f.revenue) AS total_revenue:** It retrieves the year, genre, and total revenue columns from the tables, and uses the SUM function to aggregate the revenue for each genre in each year. ‘
2. **FROM Theater_Events f JOIN Movie_Dimension md ON f.movie_key = md.movie_key JOIN Time_Dimension td ON f.time_key = td.time_key:** Specifies the tables we are retrieving data from and joins them using their respective foreign keys. The join is performed using the movie_key column from both the Theater_Events and Movie_Dimension tables, and the time_key column from both the Theater_Events and Time_Dimension tables

3. **GROUP BY td.year, md.genre:** This clause groups the results by year and genre to calculate the total revenue for each combination of year and genre.
4. **ORDER BY td.year, total_revenue DESC:** This clause orders the results by year in ascending order and by total revenue in descending order.
5. **LIMIT 5:** It limits the results to the top 5 rows.

===================xxxxxxx===================

3. **Short notes on Columnar Databases for Data Warehousing:**

➢ Columnar databases store data in columns rather than rows.
➢ This results in faster query processing, particularly for complex analytical functions like aggregation and filtering.
➢ Columnar databases use specialized compression techniques for each data type in a column, which reduces storage requirements and further improves query performance.
➢ Columnar databases are designed for analytical workloads and can handle complex queries more efficiently than traditional row-based databases.
➢ They support schema-on-read, which allows for more flexible data modeling and easier data exploration.
➢ Popular columnar databases used for data warehouse solutions include Amazon Redshift, Google BigQuery, and Apache Cassandra.

Key Insights and Takeaways on Using Columnar Databases for Data Warehouses in industry:

1. **Improved Query Efficiency and Complex Query Handling:**

➢ By storing table data in a columnar format and allowing for efficient scanning of individual columns, complex queries and aggregations can be handled more efficiently than traditional row-based databases.

- ➢ This leads to improved data analysis and the extraction of valuable insights.
- ➢ For example, Spotify is successfully utilizing columnar databases, like Google BigQuery, due to their optimized design for analytical workloads.

  Reference links:

  https://cloud.google.com/bigquery/docs/storage_overview#:~:text=BigQuery%20stores%20table%20data%20in,columns%20over%20an%20entire%20dataset.

  https://cloud.google.com/customers/spotify#:~:text=Spotify's%20technology%20leaders%20point%20to,has%20also%20been%20particularly%20useful.

2. **Flexible data modeling:**

- ➢ Columnar databases support schema-on-read, which allows for flexible data modeling and easier data exploration.
- ➢ For example, Apache Cassandra has been used by companies like Netflix to handle large volumes of data and provide a flexible data model for their analytics.

  Reference links:

  https://cassandra.apache.org/doc/latest/cassandra/architecture/overview.html

  https://www.infoq.com/news/2023/02/netflix-annotations-cassandra/#:~:text=The%20team%20at%20Netflix%20decided,database%20that%20provides%20horizontal%20scalability.

3. I**mproved query performance:**

- ➢ Columnar databases provide faster query processing and reduced storage requirements due to their optimized compression techniques.
- ➢ For example, Amazon Redshift has been used by companies like Lyft to handle large amounts of data and improve query performance.

  Reference Links:

  https://docs.aws.amazon.com/redshift/latest/dg/c_columnar_storage_disk_mem_mgmnt.html

  https://aws.amazon.com/solutions/case-studies/lyft/#:~:text=Lyft%20launched%20on%20AWS%20and,%2Dride%20product%2C%20Lyft%20Line.

4. **Compression:**

- ➢ Columnar databases are designed to store data more efficiently than traditional row-based databases, which can lead to significant storage cost savings.

- ➢ This is because columnar databases can use compression techniques that are optimized for the data types and patterns found in analytical workloads.
- ➢ For example, Vertica has been used by Zynga (Online game company) to compress large datasets and reduce storage costs.

Reference Links:

https://www.vertica.com/secrets-behind-verticas-performance/

https://readwrite.com/vertica-the-analytics-behind-zynga/

5. **Real-time analytics:**

- ➢ Columnar databases are well-suited for real-time analytics and can deliver fast query response times even on large datasets.
- ➢ This makes them a popular choice for applications that require near-instantaneous insights, such as fraud detection, recommendation engines, and IoT sensor data analysis.
- ➢ For example, Kinetica has been used by companies like Dell to perform real-time analytics on streaming data.

Reference Links:

https://www.kinetica.com/blog/top-questions/#:~:text=Kinetica%20is%20a%20vectorized%20columnar,%2C%20and%20high%2Dcardinality%20data.

https://www.kinetica.com/partner/dell-emc/#:~:text=Dell%20EMC%20OEM%20Solutions%20and%20Kinetica%20have%20signed%20an%20OEM,PowerEdge%20servers%20with%20NVIDIA%20GPUs.

3. b) 5 Key Observation on different OLAP TOOLS used in the industry:

1. **Microsoft SSAS**
    - ➢ It is a powerful OLAP tool that allows for the creation and management of multi-dimensional databases for analysis and reporting purposes.
    - ➢ One of the key features of SSAS is its ability to create and manage hierarchies, which allows for easy analysis of data at different levels of aggregation. This feature has been used by companies such as The North Face to analyze sales data across various dimensions and make data-driven decisions.

2. **Oracle OLAP**
    - ➢ It is widely used in various industries for fast and efficient data analysis and reporting. Its ability to manage data and business rules securely and centrally makes it an ideal solution for enterprises.
    - ➢ Hundreds of analytic functions can be easily combined to solve any calculation requirement, and its standard star schema design allows for easy integration with various reporting and analysis tools, making it simple and productive to use.

3. **IBM Cognos**
    - ➢ It is a well-established business intelligence product, widely used in the financial sector, especially in banking, insurance, and securities. It provides various OLAP features to help businesses gain insights from their data, including multidimensional analysis((analyzing data across multiple dimensions), ad hoc reporting (creating custom reports on the fly), interactive dashboards (visualizing data in a dashboard format), predictive analytics (using statistical models to predict future outcomes), and data visualization.
    - ➢ Metlife insurance is an example of an industry that utilizes IBM Cognos for financial analysis and reporting. By leveraging its capabilities, they can track their performance, improve customer satisfaction, and make data-driven business decisions.

4. **Apache Kylin**:
    - ➢ It is being used by various industries, including e-commerce, finance, and telecommunications, to provide fast and accurate data analytics. For example, eBay uses Apache Kylin to process massive amounts of data from its e-commerce platform and provide near-real-time insights to its business users.
    - ➢ One of its key features is its ability to provide sub-second query response times even for complex queries on big data. This is achieved through its pre-calculation of cubes using a technology called "**Cube Building**" and its efficient use of Hadoop-based distributed processing. This makes it an ideal solution for industries that require fast and accurate data analytics, such as real-time marketing, fraud detection, and customer behavior analysis.

5. **MicroStrategy OLAP Services:**
    - ➢ They are used in Retail, Finance, HealthCare industries and offer a feature called Intelligent Cubes, which allows users to create pre-aggregated data sets that can be accessed and analyzed quickly, without the need to re-query the underlying data source. This feature can significantly improve query performance and speed up report generation, making it easier for users to get the information they need

in a timely manner. Additionally, Intelligent Cubes can be scheduled to refresh automatically, ensuring that the data is always up-to-date.

➢ Another key takeaway from using MicroStrategy OLAP Services is its ease of use. The tool has a user-friendly interface that allows non-technical users to create reports and perform ad-hoc analyses easily. Additionally, it offers a variety of visualization options, including charts and graphs, to help users better understand their data. This makes it a popular choice for businesses looking to empower their employees with self-service analytics capabilities

====================xxxxxxx=====================

Note: As I am not familiar with ML/k-means, I am unable to provide a solution for questions 4a and 4b.