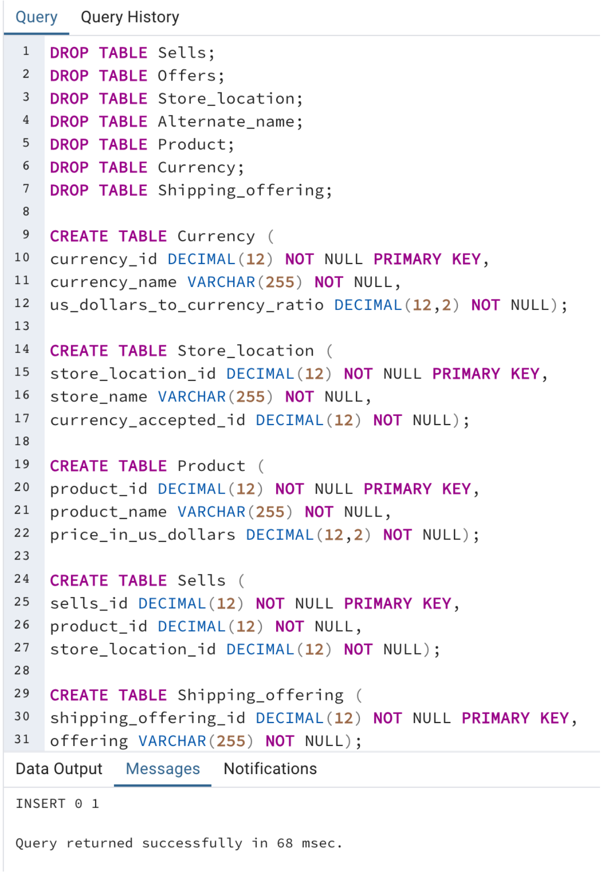
MET CS 669 Database Design and Implementation for Business
Lab 5: Subqueries and Distributed Databases

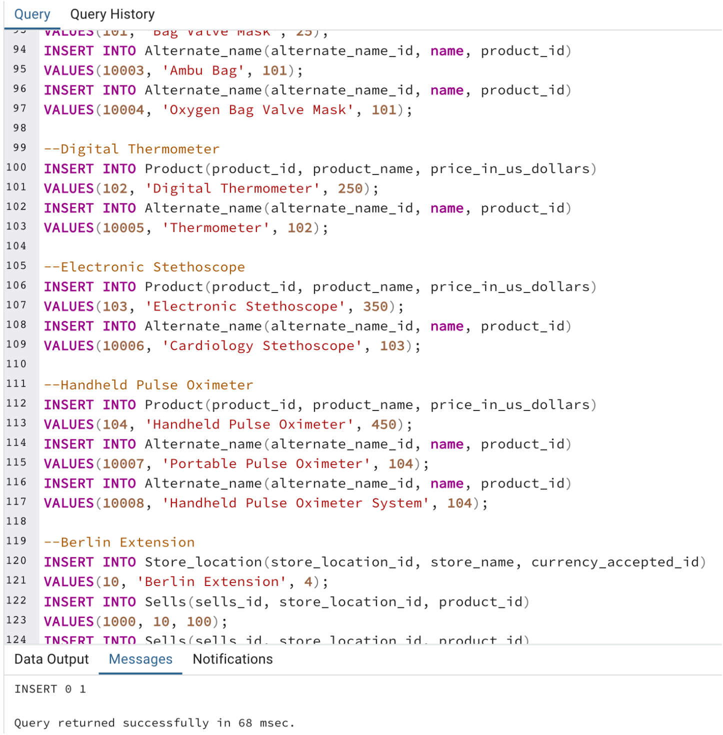
Section One – Subqueries

1. *Create Table Structure* – Create the tables in the schema, including all of their columns, datatypes, and constraints, and populate the tables with data. You can do so by executing the DDL and DML above in your SQL client. You only need to capture one or two demonstrative screenshots for this step. No need to screenshot execution of every line of code (that could require dozens of screenshots).



```sql
DROP TABLE Sells;
DROP TABLE Offers;
DROP TABLE Store_location;
DROP TABLE Alternate_name;
DROP TABLE Product;
DROP TABLE Currency;
DROP TABLE Shipping_offering;

CREATE TABLE Currency (
currency_id DECIMAL(12) NOT NULL PRIMARY KEY,
currency_name VARCHAR(255) NOT NULL,
us_dollars_to_currency_ratio DECIMAL(12,2) NOT NULL);

CREATE TABLE Store_location (
store_location_id DECIMAL(12) NOT NULL PRIMARY KEY,
store_name VARCHAR(255) NOT NULL,
currency_accepted_id DECIMAL(12) NOT NULL);

CREATE TABLE Product (
product_id DECIMAL(12) NOT NULL PRIMARY KEY,
product_name VARCHAR(255) NOT NULL,
price_in_us_dollars DECIMAL(12,2) NOT NULL);

CREATE TABLE Sells (
sells_id DECIMAL(12) NOT NULL PRIMARY KEY,
product_id DECIMAL(12) NOT NULL,
store_location_id DECIMAL(12) NOT NULL);

CREATE TABLE Shipping_offering (
shipping_offering_id DECIMAL(12) NOT NULL PRIMARY KEY,
offering VARCHAR(255) NOT NULL);
```

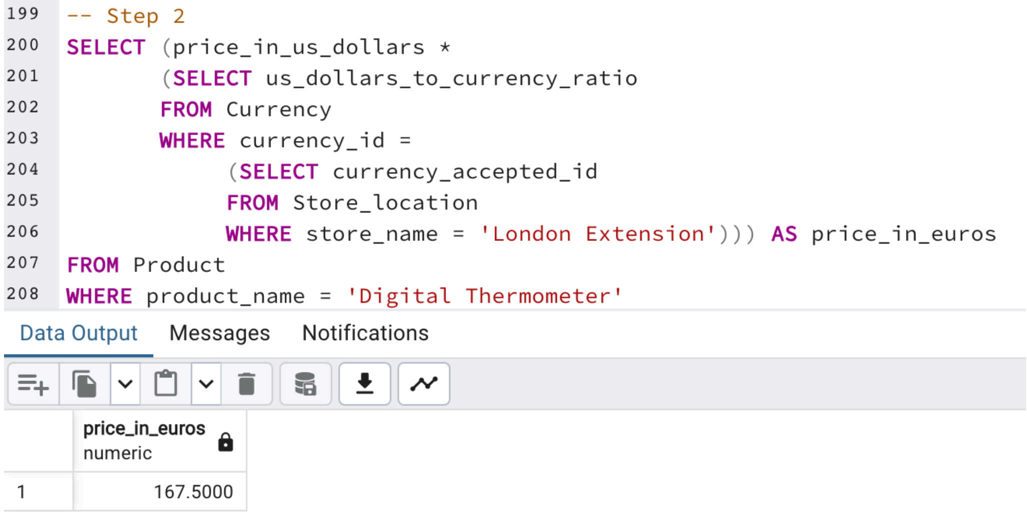
```
INSERT 0 1

Query returned successfully in 68 msec.
```



```
VALUES(101, 'Bag Valve Mask', 25);
INSERT INTO Alternate_name(alternate_name_id, name, product_id)
VALUES(10003, 'Ambu Bag', 101);
INSERT INTO Alternate_name(alternate_name_id, name, product_id)
VALUES(10004, 'Oxygen Bag Valve Mask', 101);

--Digital Thermometer
INSERT INTO Product(product_id, product_name, price_in_us_dollars)
VALUES(102, 'Digital Thermometer', 250);
INSERT INTO Alternate_name(alternate_name_id, name, product_id)
VALUES(10005, 'Thermometer', 102);

--Electronic Stethoscope
INSERT INTO Product(product_id, product_name, price_in_us_dollars)
VALUES(103, 'Electronic Stethoscope', 350);
INSERT INTO Alternate_name(alternate_name_id, name, product_id)
VALUES(10006, 'Cardiology Stethoscope', 103);

--Handheld Pulse Oximeter
INSERT INTO Product(product_id, product_name, price_in_us_dollars)
VALUES(104, 'Handheld Pulse Oximeter', 450);
INSERT INTO Alternate_name(alternate_name_id, name, product_id)
VALUES(10007, 'Portable Pulse Oximeter', 104);
INSERT INTO Alternate_name(alternate_name_id, name, product_id)
VALUES(10008, 'Handheld Pulse Oximeter System', 104);

--Berlin Extension
INSERT INTO Store_location(store_location_id, store_name, currency_accepted_id)
VALUES(10, 'Berlin Extension', 4);
INSERT INTO Sells(sells_id, store_location_id, product_id)
VALUES(1000, 10, 100);
INSERT INTO Sells(sells_id, store_location_id, product_id)
```

```
INSERT 0 1

Query returned successfully in 68 msec.
```

2. *Subquery in Column List* – Write a query that retrieves the price of a digital thermometer in London. A subquery will retrieve the currency ratio for the currency accepted in London. The outer query will use the results of the subquery (the currency ratio) in order to determine the price of the thermometer. The subquery should retrieve dynamic results by looking up the currency the store location accepts, not by hardcoding a specific value. Briefly explain how your solution makes use of the uncorrelated subquery to help retrieve the result.



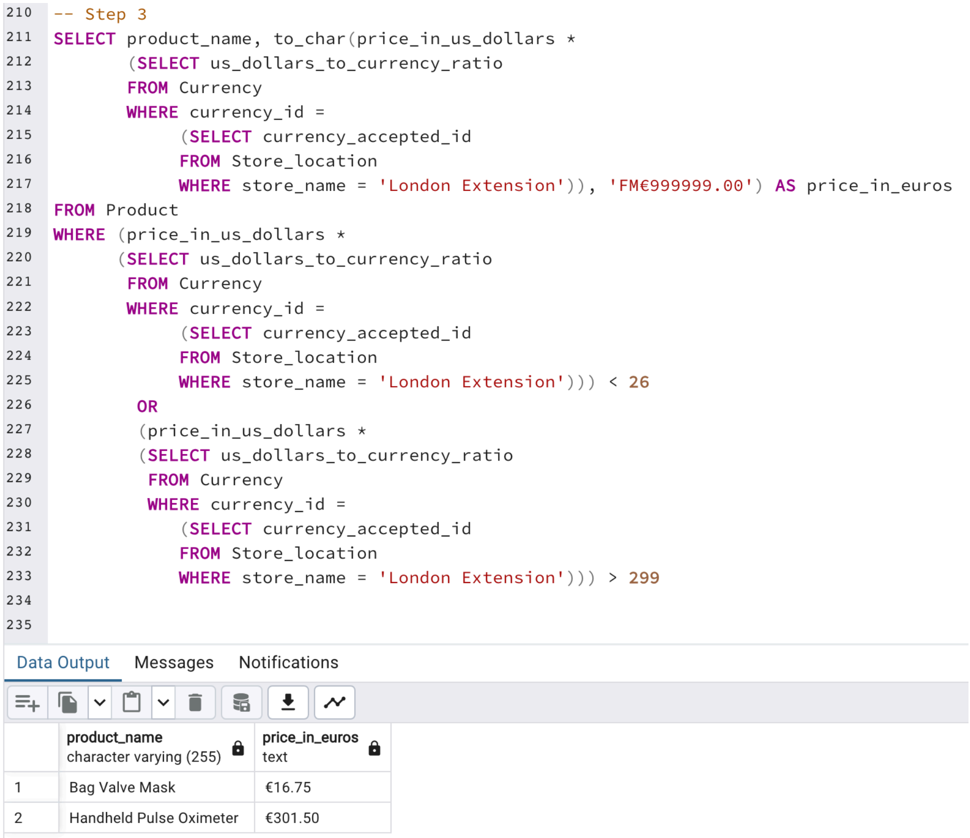
```sql
-- Step 2
SELECT (price_in_us_dollars *
       (SELECT us_dollars_to_currency_ratio
       FROM Currency
       WHERE currency_id =
            (SELECT currency_accepted_id
            FROM Store_location
            WHERE store_name = 'London Extension'))) AS price_in_euros
FROM Product
WHERE product_name = 'Digital Thermometer'
```

	price_in_euros numeric
1	167.5000

Instead of hard coding the US exchange rate with the London store, I linked the London store with its currency ratio and multiplied that by the price of the digital thermometer.

3. *Subquery in WHERE Clause* – Imagine a charity in London is hosting a fundraiser to purchase medical supplies for organizations that provide care to people in impoverished areas. The charity is targeting both people with average income as well a few wealthier people, and to this end asks for a selection of products both groups can contribute to purchase. Specifically, for the average income group, they would like to know what products cost less than 26 Euros, and for the wealthier group, they would like to know what products cost more than 299 Euros.

   a. Develop a single query to provide them this result, which should contain uncorrelated subqueries and should list the names of the products as well as their prices in Euros.



```sql
-- Step 3
SELECT product_name, to_char(price_in_us_dollars *
       (SELECT us_dollars_to_currency_ratio
       FROM Currency
       WHERE currency_id =
            (SELECT currency_accepted_id
            FROM Store_location
            WHERE store_name = 'London Extension')), 'FM€999999.00') AS price_in_euros
FROM Product
WHERE (price_in_us_dollars *
      (SELECT us_dollars_to_currency_ratio
       FROM Currency
       WHERE currency_id =
            (SELECT currency_accepted_id
            FROM Store_location
            WHERE store_name = 'London Extension'))) < 26
        OR
        (price_in_us_dollars *
        (SELECT us_dollars_to_currency_ratio
         FROM Currency
         WHERE currency_id =
            (SELECT currency_accepted_id
            FROM Store_location
            WHERE store_name = 'London Extension'))) > 299
```

	product_name character varying (255)	price_in_euros text
1	Bag Valve Mask	€16.75
2	Handheld Pulse Oximeter	€301.50

b. Explain how what each subquery does, its role in the overall query, and how the subqueries were integrated to give the correct results.

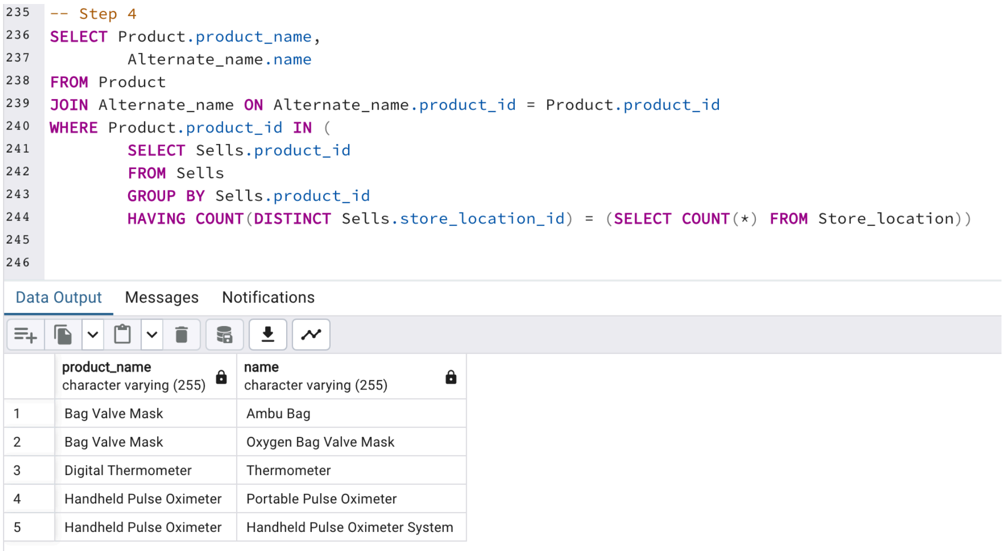
Note that the Euro monetary prefix is €.

Since the subquery is looking for the price in euros, we have to include the select also in the where part of the outer query. We use an or because we want both the ones less than 26 euros and the ones more than 299 euros.

4. *Using the IN Clause with a Subquery* – Imagine that Esther is a traveling doctor who works for an agency that sends her to various locations throughout the world with very little notice. As a result, she needs to know about medical supplies *that are available in all store locations (not just some locations)*. This way, regardless of where she is sent, she knows she can purchase those products. She is also interested in viewing the alternate names for these products, so she is absolutely certain what each product is.

   Note: It is important to Esther that she can purchase the product in any location; only products sold in all stores should be listed, that is, if a product is sold in some stores, but not all stores, it should not be listed.

   a. Develop a single query to list out these results, making sure to use uncorrelated subqueries where needed (one subquery will be put into the WHERE clause of the outer query).



```sql
-- Step 4
SELECT Product.product_name,
        Alternate_name.name
FROM Product
JOIN Alternate_name ON Alternate_name.product_id = Product.product_id
WHERE Product.product_id IN (
        SELECT Sells.product_id
        FROM Sells
        GROUP BY Sells.product_id
        HAVING COUNT(DISTINCT Sells.store_location_id) = (SELECT COUNT(*) FROM Store_location))
```

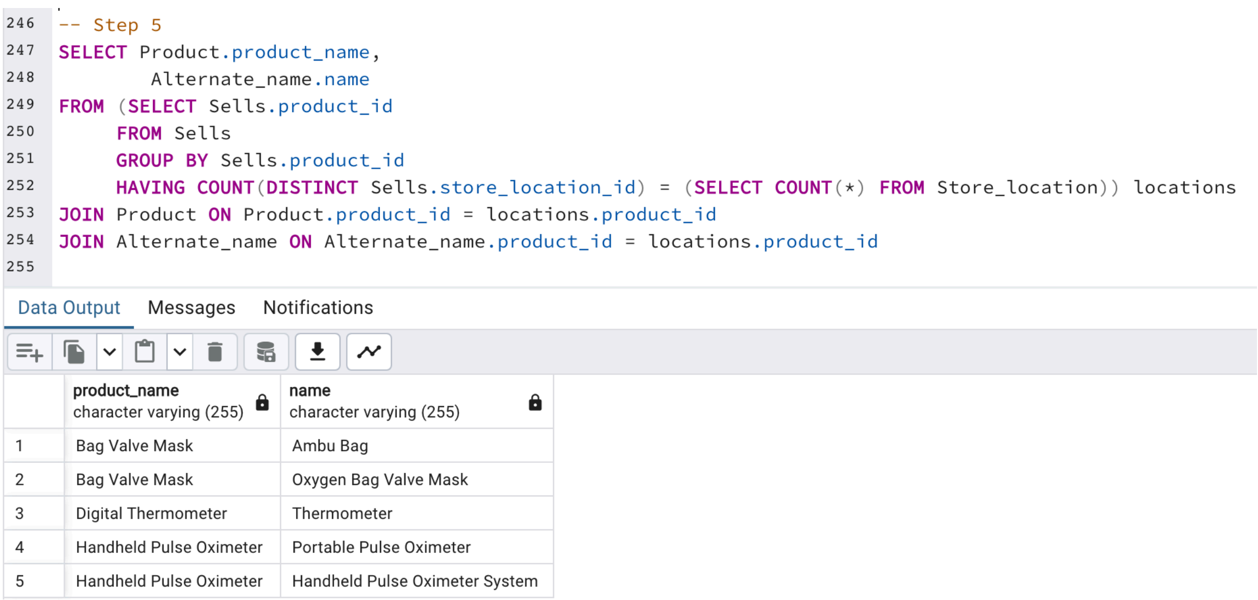
	product_name character varying (255)	name character varying (255)
1	Bag Valve Mask	Ambu Bag
2	Bag Valve Mask	Oxygen Bag Valve Mask
3	Digital Thermometer	Thermometer
4	Handheld Pulse Oximeter	Portable Pulse Oximeter
5	Handheld Pulse Oximeter	Handheld Pulse Oximeter System

b. Explain how what each subquery does, its role in the overall query, and how the subqueries were integrated to give the correct results.

In your thinking about how to address this use case, one item should be brought to your attention – the phrase “all store locations”. By eyeballing the data, you can determine the number of locations and hardcode that number, which will satisfy Esther’s request at this present time; however, as the number of locations change over time (with stores opening or closing), such hardcoding would fail. It’s better to dynamically determine the total number of locations in the query itself so that the results are correct over time.

The subquery creates a list of product id’s that have every store_location_id in the sells table. These are then used to pick out the product name and alternate name.

5. *Subquery in FROM Clause* – For this problem you will write a single query to address the same use case as in step 4, but change your query so that the main uncorrelated subquery is in the FROM clause rather than in the WHERE clause. The results should be the same as in step 4, except of course possibly row ordering which can vary. Explain how you integrated the subquery into the FROM clause to derive the same results as step 4.

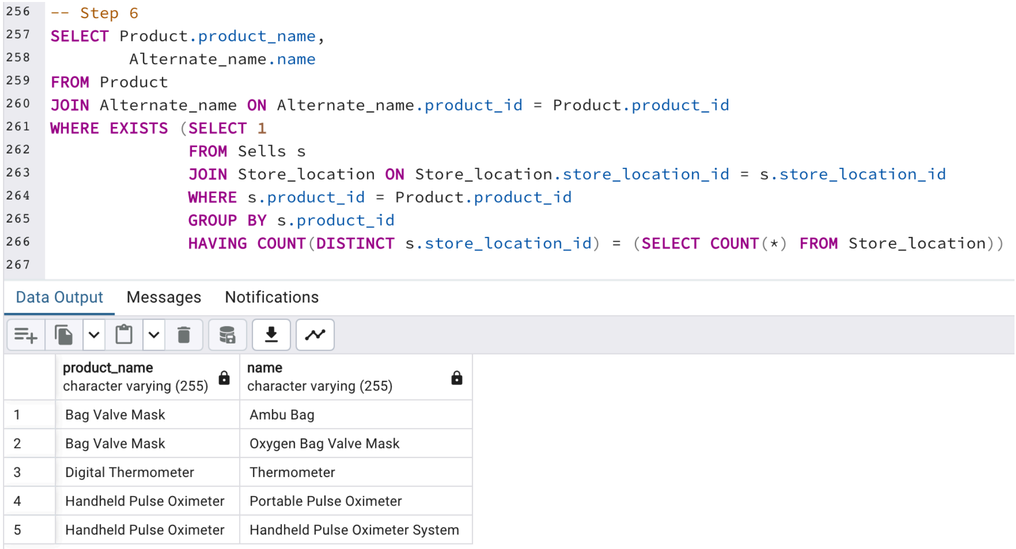


```
-- Step 5
SELECT Product.product_name,
        Alternate_name.name
FROM (SELECT Sells.product_id
     FROM Sells
     GROUP BY Sells.product_id
     HAVING COUNT(DISTINCT Sells.store_location_id) = (SELECT COUNT(*) FROM Store_location)) locations
JOIN Product ON Product.product_id = locations.product_id
JOIN Alternate_name ON Alternate_name.product_id = locations.product_id
```

	product_name character varying (255)	name character varying (255)
1	Bag Valve Mask	Ambu Bag
2	Bag Valve Mask	Oxygen Bag Valve Mask
3	Digital Thermometer	Thermometer
4	Handheld Pulse Oximeter	Portable Pulse Oximeter
5	Handheld Pulse Oximeter	Handheld Pulse Oximeter System

The subquery creates a new relation called locations that has the product id’s that have every store_location_id. The outer query then selects the product and alternate name from that table. To do this though, the 3 tables have to be joined together.

6. *Correlated Subquery* – For this problem you will write a single query to address the same use case as in step 4, but change your query to use a *correlated* query combined with an EXISTS clause. The results should be the same as in step 4, except of course possibly row ordering which can vary. Explain:



```
-- Step 6
SELECT Product.product_name,
        Alternate_name.name
FROM Product
JOIN Alternate_name ON Alternate_name.product_id = Product.product_id
WHERE EXISTS (SELECT 1
              FROM Sells s
              JOIN Store_location ON Store_location.store_location_id = s.store_location_id
              WHERE s.product_id = Product.product_id
              GROUP BY s.product_id
              HAVING COUNT(DISTINCT s.store_location_id) = (SELECT COUNT(*) FROM Store_location))
```

	product_name character varying (255)	name character varying (255)
1	Bag Valve Mask	Ambu Bag
2	Bag Valve Mask	Oxygen Bag Valve Mask
3	Digital Thermometer	Thermometer
4	Handheld Pulse Oximeter	Portable Pulse Oximeter
5	Handheld Pulse Oximeter	Handheld Pulse Oximeter System

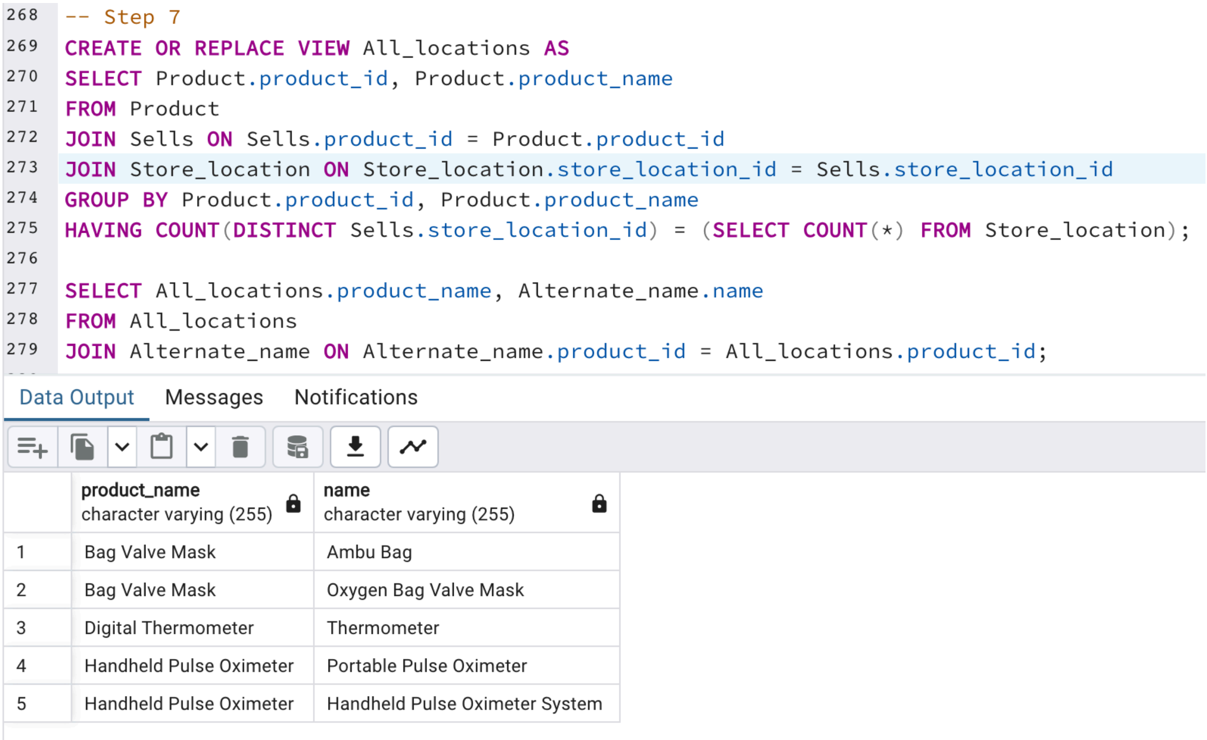
a. how your solution makes use of the correlated subquery and EXISTS clause to help retrieve the result

For this one the exists clause selects every row it this is true and then the select feature takes the product name and alternate name out.

b. how and when the correlated subquery is executed in the context of the outer query.

Correlated subquerry's are executed in tandem with the out query. This means that the subquery is executed once for each row. This means that the product name and alternate name are grabbed from each row that the exists clause is true for.

7. *Correlated Subquery Using View in Query* – For this problem you will write a query to address the same use case as in step 4, except you will create and use a *view* in the FROM clause in place of the subquery. The results should be the same as in step 4, except of course possibly row ordering which can vary.



```
-- Step 7
CREATE OR REPLACE VIEW All_locations AS
SELECT Product.product_id, Product.product_name
FROM Product
JOIN Sells ON Sells.product_id = Product.product_id
JOIN Store_location ON Store_location.store_location_id = Sells.store_location_id
GROUP BY Product.product_id, Product.product_name
HAVING COUNT(DISTINCT Sells.store_location_id) = (SELECT COUNT(*) FROM Store_location);

SELECT All_locations.product_name, Alternate_name.name
FROM All_locations
JOIN Alternate_name ON Alternate_name.product_id = All_locations.product_id;
```

	product_name character varying (255)	name character varying (255)
1	Bag Valve Mask	Ambu Bag
2	Bag Valve Mask	Oxygen Bag Valve Mask
3	Digital Thermometer	Thermometer
4	Handheld Pulse Oximeter	Portable Pulse Oximeter
5	Handheld Pulse Oximeter	Handheld Pulse Oximeter System

Section Two – Concurrency

Use the tables and transactions provided in the lab; do not create your own.

8. *Issues with No Concurrency Control* – Imagine the transactions for this section are presented to a modern relational database at the same time, and the database does *not* have concurrency control mechanisms in place. Show a step-by-step schedule that results in a lost update, inconsistent analysis, or uncommitted dependency. Also list out the contents of the table after the transactions complete using the schedule. You only need to show a schedule for one of the issues, not all three. You are not creating this table in SQL, so it is fine to show the table in Excel, Word, or another comparable application.

Schedule	
Step	Explanation
Transaction 2: Read the value from row 2	The database reads in the value 2
Transaction 2: Write that value to row 4	The database replaces the value 4 with the value 2
Transaction 1: Read the value from row 4.	The database reads in the value 2
Transaction 2: Write the literal value “15” to row 3.	The database updates row 3 with the value 15
Transaction 1: Multiply that value times 3.	The database multiplies 2 by 3 to get 6
Transaction 1: Write the result to row 3.	The database enters 6 in row 3
Transaction 2: Commit	The database commits the results
Transaction 1: Write the literal value “8” to row 2.	The database enters 8 in row 2
Transaction 1: Write the literal value “20” to row 5.	The database enters 20 in row 5
Transaction 1: Commit	The database commits the results

Data Table (inconsistent analysis)
1
8
6
2
20

Data Table (series)
1
8
15
8
20

If the “correct” way for this database to run is in series, then the schedule I have created is an example of inconsistent analysis. The database looses the value 15 and creates the value 6 and 2. Neither 6 or 2 should be present and 15 gets overwritten. If this were data that could be matching then for example the birthdate and name would be inconsistent with the individuals actual birthdate and name.

9. *Issues with Locking and Multiversioning* – Imagine the database has both locking and multiversioning in place for concurrency control.
    a. Starting with the same schedule in the prior step, show and explain step-by-step how the use of locking and multiversioning modifies the schedule. Also list out the contents of the table after the transactions complete using the new schedule. Make sure to explain specifically whether and how locking and multiversioning modifies the schedule and affects the final resulting table.
    b. Could a schedule of these transactions result in a deadlock? If not, explain why. If so, show a step-by-step schedule that results in a deadlock.

Schedule	
Step	Explanation
Transaction 2: Read the value from row 2	The database reads in the value 2 with a shared lock on row 2
Transaction 2: Write that value to row 4	The database replaces the value 4 with the value 2 with an exclusive lock on row 4
Transaction 1: Read the value from row 4.	The database is forced to wait at this step because transaction 2 has an exclusive lock on row 4
Transaction 2: Write the literal value “15” to row 3.	The database updates row 3 with the value 15 shared lock on row 3
Transaction 1: Multiply that value times 3.	The database multiplies reads the value from row 4 (2) and multiplies it by 3 with a shared lock on row 4.
Transaction 1: Write the result to row 3.	The database enters 6 in row 3 with an exclusive lock
Transaction 2: Commit	The database commits the results
Transaction 1: Write the literal value “8” to row 2.	The database enters 8 in row 2 with an exclusive lock
Transaction 1: Write the literal value “20” to row 5.	The database enters 20 in row 5 with an exclusive lock

Transaction 1: Commit	The database commits the results

Data Table
1
8
6
2
20

The use of locks doesn’t actually change the outcome of the data table. This is because transactions ‘15’ still gets overwritten. There is no way for a deadlock because Transaction 1 doesn’t have any shared locks while Transaction 2 is altering and using exclusive locks.