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

Lab 5 Explanations

This explanation document illustrates how to correctly execute each SQL construct step-by-step for Lab 5, and explains important theoretical and practical details. Before completing a step, read its explanation here first.

Table of Contents

Subqueries Overview

The foundation for learning how subqueries work lies in the expressional nature of the relational model. We learned previously that the operations in the relational model, such as SELECT, PROJECT, and UNION, perform operations on relations *and* yield a new relation as the result. That is, when one operation operates on a relation, and yields a new relation, we can use a second operation to operate on the result of the first operation.

We will take a look at these relational operations in a moment, but first let us look at a simple mathematical example. If we add two plus two to obtain a result of 4, **2 + 2 = 4**, we have applied the plus operation to two numbers. Interestingly, the result of the plus operation is another number. So we can say that the plus operator operates on two numbers, and results in a new number.

Because the result is a number, there is no reason why we cannot use that result in another operation. For example, if we wanted to subtract one from the result, we *nest* the operations like so: **(2 + 2) - 1 = 3**. That is, the plus operation adds two plus two to arrive at a result of 4, then the minus operation subtracts one from that result, to arrive at a final result of 3.

Now that we have seen a simple example of nesting operations, let us look at a concrete example in the relational model. Imagine that we have a Person relation which has two columns – first_name and last_name.

Person =

first_name	*last_name*
Bill	Glass
Jane	Smith

We can use the PROJECT operation to create a new relation consisting of only the last_name column from Person, denoted as PROJECT$_{last\_name}$(Person).

PROJECT$_{last\_name}$(Person) =

last_name
Glass
Smith

Because the PROJECT operation creates a new relation, there is no reason why we cannot apply a second operation to that new relation. In this example, we can apply the SELECT operation to the result of the PROJECT in order to retrieve only the rows where the last name is Smith, denoted as SELECT$_{last\_name=Smith}$(PROJECT$_{last\_name}$(Person)).

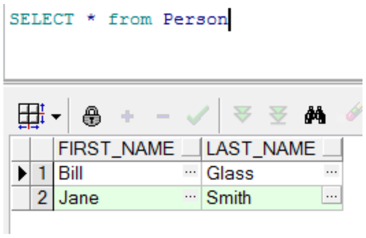
SELECT$_{last\_name=Smith}$(PROJECT$_{last\_name}$(Person)) =

last_name
Smith

In other words, wherever a particular operation expects a relation, we can give it an existing relation, or we can give it the result of another operation. When we have operations that operate on the results of other operations, we term this as *nesting* the operations. It is this nesting ability that gives the relational model, and therefore relational databases, their expressional power. Think about the concept of nesting operations until you are sure you understand it well, for it is the foundation of understanding subqueries.

Relational databases allow queries to be nested inside of other queries. Wherever a SQL statement expects a table, we can give it an existing table, or we can give it the result of another query. This is because all queries in relational databases yield result sets which are of the same form as a relational table -- a two-dimensional set of rows and columns.

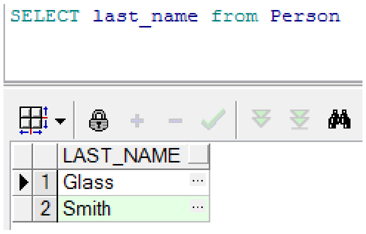
Let us look at a concrete example of a subquery using a Person table with the same rows as the Person relation just described. When we issue a **`SELECT * FROM Person`** command, we see the initial Person table with Bill Glass and Jane Smith as row values.



```
SELECT * from Person
```

	FIRST_NAME	LAST_NAME
1	Bill	Glass
2	Jane	Smith

To perform the **PROJECT$_{last\_name}$(Person)** operation in SQL, we specify the last_name column in the command -- **`SELECT last_name FROM Person --`** and the database returns for us the values in the last_name column.



```
SELECT last_name from Person
```

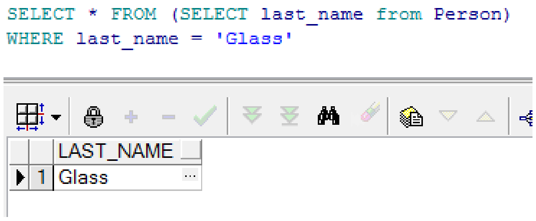
	LAST_NAME
1	Glass
2	Smith

Now if we would like to perform the equivalent of the **SELECT$_{last\_name=Smith}$(PROJECT$_{last\_name}$(Person))** operation in our database. We can do so through the use of a subquery, illustrated in the command below.

```
SELECT * FROM (SELECT last_name from Person)
WHERE last_name = 'Glass'
```

Did you see what we did in this command? The **`SELECT last_name FROM Person`** query has been put in place of an existing table, in order to perform the PROJECT operation. We usually see the name of a table in the **`FROM`** clause, but here we see the placement of the subquery. And then the additional **`WHERE last_name = 'Glass'`** restriction has been placed on the outer query, which performs the relational SELECT operation. Thus, we have used the results from one query in another query, nesting one query inside the other. This use of subqueries is powerful and allows us to obtain and manipulate data in a variety of ways.

In the screenshot below, you can see that we obtain the result we expect.

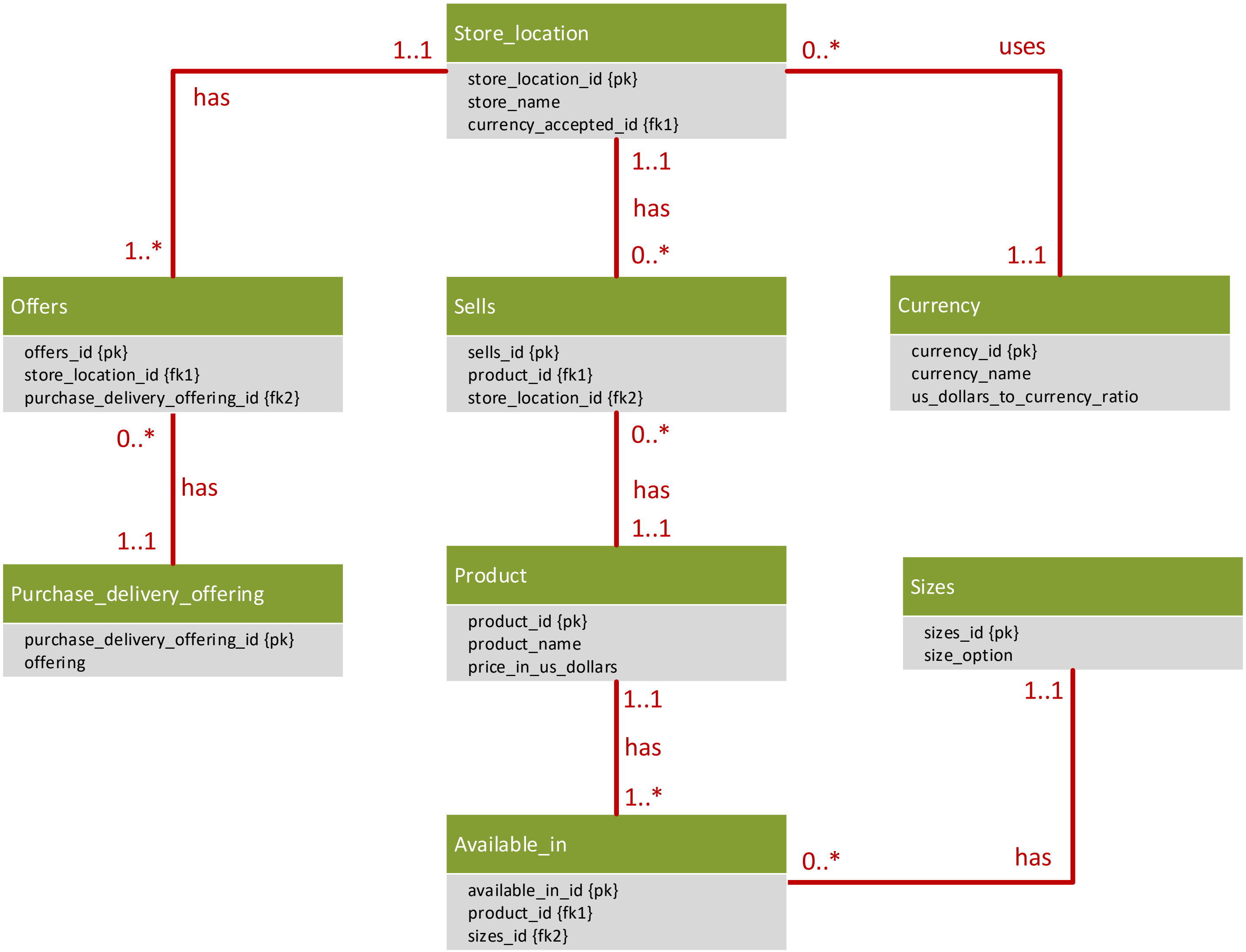


```
SELECT * FROM (SELECT last_name from Person)
WHERE last_name = 'Glass'
```

	LAST_NAME
1	Glass

Example Schema Overview

In this lab, we explain how to perform each step by practicing similar subqueries on the schema illustrated below. This example schema is similar to the schema you are working with in the lab, but does have a few differences, because it's designed for general products rather than medical products.



This schema's structure supports basic product and currency information for an international organization, including store locations, the products they sell and their sizes, purchase and delivery offerings, the currency each location accepts, as well as conversion factors for converting from U.S. dollars into the accepted currency. This schema models prices and exchange rates at a specific point in time. While a real-world schema would make provision for changes to prices and exchange rates over time, the tables needed to support this have been intentionally excluded from our schema, because their addition would add unneeded complexity on your journey of learning subqueries, expressions, and value manipulation. The schema has just the right amount of complexity for your learning.

The data for the tables is listed below.

Currencies

Name	Ratio
British Pound	0.66
Canadian Dollar	1.33
US Dollar	1.00
Euro	0.93
Mexican Peso	16.75

Store Locations

Name	Currency
Berlin Extension	Euro
Cancun Extension	Mexican Peso
London Extension	British Pound
New York Extension	US Dollar
Toronto Extension	Canadian Dollar

Product

Name	US Dollar Price
Cashmere Sweater	$100
Designer Jeans	$150
Flowing Skirt	$125
Silk Blouse	$200
Wool Overcoat	$250

Sells

Store Location	Product
Berlin Extension	Cashmere Sweater
Berlin Extension	Designer Jeans
Berlin Extension	Silk Blouse
Berlin Extension	Wool Overcoat
Cancun Extension	Designer Jeans
Cancun Extension	Flowing Skirt
Cancun Extension	Silk Blouse
London Extension	Cashmere Sweater
London Extension	Designer Jeans
London Extension	Flowing Skirt
London Extension	Silk Blouse
London Extension	Wool Overcoat
New York Extension	Cashmere Sweater
New York Extension	Designer Jeans
New York Extension	Flowing Skirt
New York Extension	Silk Blouse
New York Extension	Wool Overcoat
Toronto Extension	Cashmere Sweater
Toronto Extension	Designer Jeans
Toronto Extension	Flowing Skirt
Toronto Extension	Silk Blouse
Toronto Extension	Wool Overcoat

Purchase_delivery_offering

Offering
Purchase In Store
Purchase Online, Ship to Home
Purchase Online, Pickup in Store

Offers

Store Location	Purchase Delivery Offering
Berlin Extension	Purchase In Store
Cancun Extension	Purchase In Store
London Extension	Purchase In Store
London Extension	Purchase Online, Ship to Home
London Extension	Purchase Online, Pickup in Store
New York Extension	Purchase In Store
New York Extension	Purchase Online, Pickup in Store
Toronto Extension	Purchase In Store

Sizes

Size Option
Small
Medium
Large
Various
2
4
6
8
10
12
14
16

Available_in

Product	Size Option
Cashmere Sweater	Small
Cashmere Sweater	Medium
Cashmere Sweater	Large
Designer Jeans	Various
Flowing Skirt	2
Flowing Skirt	4
Flowing Skirt	6
Flowing Skirt	8
Flowing Skirt	10
Flowing Skirt	12
Flowing Skirt	14
Flowing Skirt	16
Silk Blouse	Small
Silk Blouse	Medium
Silk Blouse	Large
Wool Overcoat	Small
Wool Overcoat	Medium
Wool Overcoat	Large

Section One – Subqueries

STEP 1

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).

The DDL and DML we use to create and populate the tables in our example schema is listed below.

```
DROP TABLE Sells;
DROP TABLE Offers;
DROP TABLE Available_in;
DROP TABLE Store_location;
DROP TABLE Product;
DROP TABLE Currency;
DROP TABLE Purchase_delivery_offering;
DROP TABLE Sizes;

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 Purchase_delivery_offering (
purchase_delivery_offering_id DECIMAL(12) NOT NULL PRIMARY KEY,
offering VARCHAR(255) NOT NULL);

CREATE TABLE Offers (
offers_id DECIMAL(12) NOT NULL PRIMARY KEY,
store_location_id DECIMAL(12) NOT NULL,
purchase_delivery_offering_id DECIMAL(12) NOT NULL);

CREATE TABLE Sizes (
sizes_id DECIMAL(12) NOT NULL PRIMARY KEY,
size_option VARCHAR(255) NOT NULL);

CREATE TABLE Available_in (
available_in_id DECIMAL(12) NOT NULL PRIMARY KEY,
product_id DECIMAL(12) NOT NULL,
sizes_id DECIMAL(12) NOT NULL);
```

```sql
ALTER TABLE Store_location
ADD CONSTRAINT fk_location_to_currency FOREIGN KEY(currency_accepted_id) REFERENCES
Currency(currency_id);

ALTER TABLE Sells
ADD CONSTRAINT fk_sells_to_product FOREIGN KEY(product_id) REFERENCES Product(product_id);

ALTER TABLE Sells
ADD CONSTRAINT fk_sells_to_location FOREIGN KEY(store_location_id) REFERENCES
Store_location(store_location_id);

ALTER TABLE Offers
ADD CONSTRAINT fk_offers_to_location FOREIGN KEY(store_location_id) REFERENCES
Store_location(store_location_id);

ALTER TABLE Offers
ADD CONSTRAINT fk_offers_to_offering FOREIGN KEY(purchase_delivery_offering_id)
REFERENCES Purchase_delivery_offering(purchase_delivery_offering_id);

ALTER TABLE Available_in
ADD CONSTRAINT fk_available_to_product FOREIGN KEY(product_id)
REFERENCES Product(product_id);

ALTER TABLE Available_in
ADD CONSTRAINT fk_available_to_sizes FOREIGN KEY(sizes_id)
REFERENCES Sizes(sizes_id);

INSERT INTO Currency(currency_id, currency_name, us_dollars_to_currency_ratio)
VALUES(1, 'Britsh Pound', 0.66);
INSERT INTO Currency(currency_id, currency_name, us_dollars_to_currency_ratio)
VALUES(2, 'Canadian Dollar', 1.33);
INSERT INTO Currency(currency_id, currency_name, us_dollars_to_currency_ratio)
VALUES(3, 'US Dollar', 1.00);
INSERT INTO Currency(currency_id, currency_name, us_dollars_to_currency_ratio)
VALUES(4, 'Euro', 0.93);
INSERT INTO Currency(currency_id, currency_name, us_dollars_to_currency_ratio)
VALUES(5, 'Mexican Peso', 16.75);

INSERT INTO Purchase_delivery_offering(purchase_delivery_offering_id, offering)
VALUES (50, 'Purchase In Store');
INSERT INTO Purchase_delivery_offering(purchase_delivery_offering_id, offering)
VALUES (51, 'Purchase Online, Ship to Home');
INSERT INTO Purchase_delivery_offering(purchase_delivery_offering_id, offering)
VALUES (52, 'Purchase Online, Pickup in Store');

INSERT INTO Sizes(sizes_id, size_option)
VALUES(1, 'Small');
INSERT INTO Sizes(sizes_id, size_option)
VALUES(2, 'Medium');
INSERT INTO Sizes(sizes_id, size_option)
VALUES(3, 'Large');
INSERT INTO Sizes(sizes_id, size_option)
VALUES(4, 'Various');
INSERT INTO Sizes(sizes_id, size_option)
VALUES(5, '2');
INSERT INTO Sizes(sizes_id, size_option)
VALUES(6, '4');
INSERT INTO Sizes(sizes_id, size_option)
VALUES(7, '6');
INSERT INTO Sizes(sizes_id, size_option)
VALUES(8, '8');
INSERT INTO Sizes(sizes_id, size_option)
VALUES(9, '10');
INSERT INTO Sizes(sizes_id, size_option)
VALUES(10, '12');
INSERT INTO Sizes(sizes_id, size_option)
```

```sql
VALUES(11, '14');
INSERT INTO Sizes(sizes_id, size_option)
VALUES(12, '16');

--Cashmere Sweater
INSERT INTO Product(product_id, product_name, price_in_us_dollars)
VALUES(100, 'Cashmere Sweater', 100);
INSERT INTO Available_in(available_in_id, product_id, sizes_id)
VALUES(10000, 100, 1);
INSERT INTO Available_in(available_in_id, product_id, sizes_id)
VALUES(10001, 100, 2);
INSERT INTO Available_in(available_in_id, product_id, sizes_id)
VALUES(10002, 100, 3);

--Designer Jeans
INSERT INTO Product(product_id, product_name, price_in_us_dollars)
VALUES(101, 'Designer Jeans', 150);
INSERT INTO Available_in(available_in_id, product_id, sizes_id)
VALUES(10003, 101, 4);

--Flowing Skirt
INSERT INTO Product(product_id, product_name, price_in_us_dollars)
VALUES(102, 'Flowing Skirt', 125);
INSERT INTO Available_in(available_in_id, product_id, sizes_id)
VALUES(10004, 102, 5);
INSERT INTO Available_in(available_in_id, product_id, sizes_id)
VALUES(10005, 102, 6);
INSERT INTO Available_in(available_in_id, product_id, sizes_id)
VALUES(10006, 102, 7);
INSERT INTO Available_in(available_in_id, product_id, sizes_id)
VALUES(10007, 102, 8);
INSERT INTO Available_in(available_in_id, product_id, sizes_id)
VALUES(10008, 102, 9);
INSERT INTO Available_in(available_in_id, product_id, sizes_id)
VALUES(10009, 102, 10);
INSERT INTO Available_in(available_in_id, product_id, sizes_id)
VALUES(10010, 102, 11);
INSERT INTO Available_in(available_in_id, product_id, sizes_id)
VALUES(10011, 102, 12);

--Silk Blouse
INSERT INTO Product(product_id, product_name, price_in_us_dollars)
VALUES(103, 'Silk Blouse', 200);
INSERT INTO Available_in(available_in_id, product_id, sizes_id)
VALUES(10012, 103, 1);
INSERT INTO Available_in(available_in_id, product_id, sizes_id)
VALUES(10013, 103, 2);
INSERT INTO Available_in(available_in_id, product_id, sizes_id)
VALUES(10014, 103, 3);

--Wool Overcoat
INSERT INTO Product(product_id, product_name, price_in_us_dollars)
VALUES(104, 'Wool  Overcoat', 250);
INSERT INTO Available_in(available_in_id, product_id, sizes_id)
VALUES(10015, 104, 1);
INSERT INTO Available_in(available_in_id, product_id, sizes_id)
VALUES(10016, 104, 2);
INSERT INTO Available_in(available_in_id, product_id, sizes_id)
VALUES(10017, 104, 3);

--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)
```

```sql
VALUES(1001, 10, 101);
INSERT INTO Sells(sells_id, store_location_id, product_id)
VALUES(1002, 10, 103);
INSERT INTO Sells(sells_id, store_location_id, product_id)
VALUES(1003, 10, 104);
INSERT INTO Offers(offers_id, store_location_id, purchase_delivery_offering_id)
VALUES(150, 10, 50);

--Cancun Extension
INSERT INTO Store_location(store_location_id, store_name, currency_accepted_id)
VALUES(11, 'Cancun Extension', 5);
INSERT INTO Sells(sells_id, store_location_id, product_id)
VALUES(1004, 11, 101);
INSERT INTO Sells(sells_id, store_location_id, product_id)
VALUES(1005, 11, 102);
INSERT INTO Sells(sells_id, store_location_id, product_id)
VALUES(1006, 11, 103);
INSERT INTO Offers(offers_id, store_location_id, purchase_delivery_offering_id)
VALUES(151, 11, 50);

--London Extension
INSERT INTO Store_location(store_location_id, store_name, currency_accepted_id)
VALUES(12, 'London Extension', 1);
INSERT INTO Sells(sells_id, store_location_id, product_id)
VALUES(1007, 12, 100);
INSERT INTO Sells(sells_id, store_location_id, product_id)
VALUES(1008, 12, 101);
INSERT INTO Sells(sells_id, store_location_id, product_id)
VALUES(1009, 12, 102);
INSERT INTO Sells(sells_id, store_location_id, product_id)
VALUES(1010, 12, 103);
INSERT INTO Sells(sells_id, store_location_id, product_id)
VALUES(1011, 12, 104);
INSERT INTO Offers(offers_id, store_location_id, purchase_delivery_offering_id)
VALUES(152, 12, 50);
INSERT INTO Offers(offers_id, store_location_id, purchase_delivery_offering_id)
VALUES(153, 12, 51);
INSERT INTO Offers(offers_id, store_location_id, purchase_delivery_offering_id)
VALUES(154, 12, 52);

--New York Extension
INSERT INTO Store_location(store_location_id, store_name, currency_accepted_id)
VALUES(13, 'New York Extension', 3);
INSERT INTO Sells(sells_id, store_location_id, product_id)
VALUES(1012, 13, 100);
INSERT INTO Sells(sells_id, store_location_id, product_id)
VALUES(1013, 13, 101);
INSERT INTO Sells(sells_id, store_location_id, product_id)
VALUES(1014, 13, 102);
INSERT INTO Sells(sells_id, store_location_id, product_id)
VALUES(1015, 13, 103);
INSERT INTO Sells(sells_id, store_location_id, product_id)
VALUES(1016, 13, 104);
INSERT INTO Offers(offers_id, store_location_id, purchase_delivery_offering_id)
VALUES(155, 13, 50);
INSERT INTO Offers(offers_id, store_location_id, purchase_delivery_offering_id)
VALUES(156, 13, 52);

--Toronto Extension
INSERT INTO Store_location(store_location_id, store_name, currency_accepted_id)
VALUES(14, 'Toronto Extension', 2);
INSERT INTO Sells(sells_id, store_location_id, product_id)
VALUES(1017, 14, 100);
INSERT INTO Sells(sells_id, store_location_id, product_id)
VALUES(1018, 14, 101);
INSERT INTO Sells(sells_id, store_location_id, product_id)
```

```
VALUES(1019, 14, 102);
INSERT INTO Sells(sells_id, store_location_id, product_id)
VALUES(1020, 14, 103);
INSERT INTO Sells(sells_id, store_location_id, product_id)
VALUES(1021, 14, 104);
INSERT INTO Offers(offers_id, store_location_id, purchase_delivery_offering_id)
VALUES(157, 14, 50);ent.
```

All you need to do for this step is copy and paste the DML and DDL provided to you in the lab, and provide some demonstrative screenshots of doing so.

STEP 2

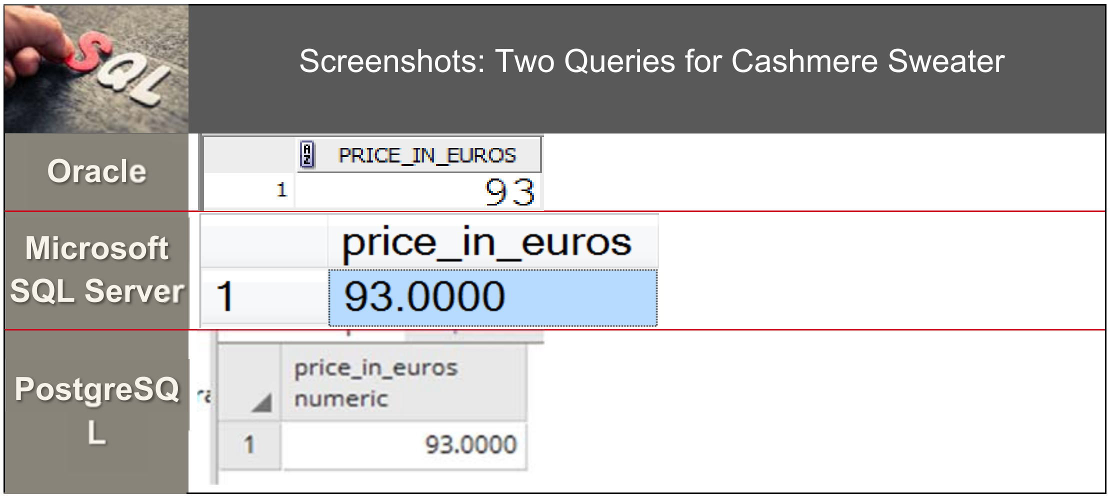
Write two queries which together retrieve the price of a digital thermometer in London. The first query will retrieve the currency ratio for the currency accepted in London. Your second query will hardcode the currency ratio retrieved in the first query, in order to determine the price of the thermometer in London. The first query should be dynamic in that the needed currency should be looked up rather than hardcoded. That is, the currency should be obtained by looking up the currency the store location accepts, not hardcoded by manually eyeballing the tables yourself.

To help you address this use case, we'll take a similar example. Imagine we were asked to give the price of the Cashmere sweater in Euros. We could write two queries. The first would find out the proper conversion ratio. The second would hardcode that value to obtain the price in Euros. Both queries are given below.

Code: Two Queries for Cashmere Sweater

```
--Obtain the Dollar to Euro ratio, which gives us 0.93.
SELECT us_dollars_to_currency_ratio
FROM   Currency
WHERE  currency_name = 'Euro';

--Hardcode 0.93 in another query to get the price.
SELECT price_in_us_dollars * 0.93 AS price_in_euros
FROM   Product
WHERE  product_name = 'Cashmere Sweater';
```

Using these queries together gives us the result of €93, as shown in the screenshots below.



Screenshots: Two Queries for Cashmere Sweater

Oracle	PRICE_IN_EUROS 1 93
Microsoft SQL Server	price_in_euros 1 93.0000
PostgreSQL	price_in_euros numeric 1 93.0000

You can do similar to address this step.

STEP 3

In step 2, you determined the price of a digital thermometer in London by writing two queries. For this step, determine the same by writing a single query that contains an uncorrelated subquery. Explain:

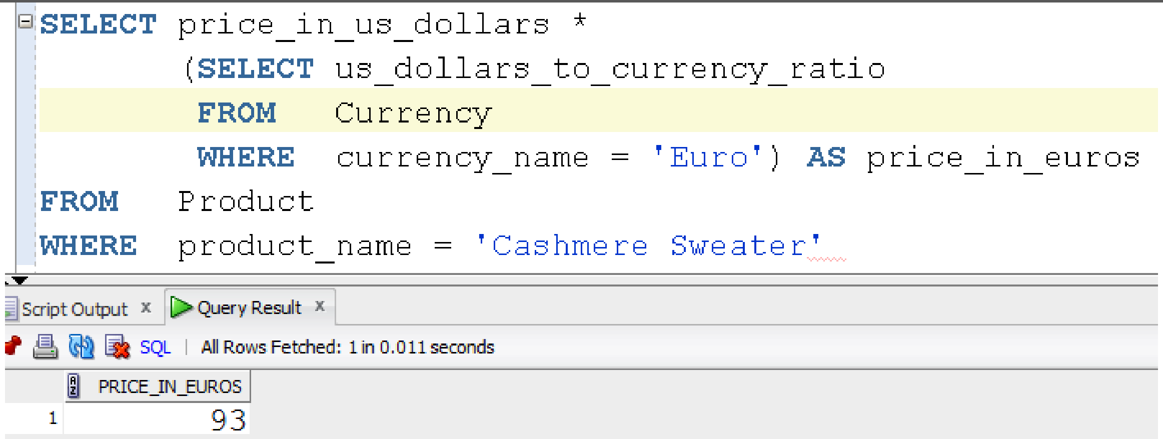
a. how your solution makes use of the uncorrelated subquery to help retrieve the result
b. how and when the uncorrelated subquery is executed in the context of the outer query, and
c. the advantages of this solution over your solution for step 2.

Continuing on with our similar example about the Cashmere sweater, executing two queries independently gives us the results we want, but forces us to manually hardcode one value into another. For subsequent executions of the query, if we do not execute the first query to reobtain the conversion ratio and resubstitute that value into the second query, we risk giving a wrong result since the conversion factor will change often.

There is a better way. Embedding one query inside the other is more concise and production-worthy, and gets us out of the business of manually hardcoding values from one query into another. Let us try it out in Oracle, SQL Server and PostgreSQL to see if we still get the same result.

Screenshots: Single Query for Cashmere Sweater

Oracle

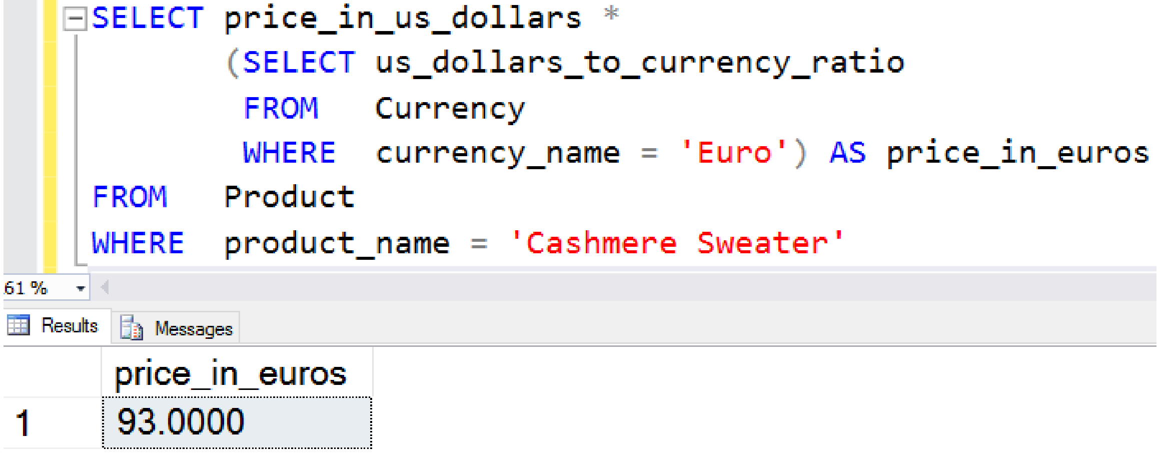


```
SELECT price_in_us_dollars *
       (SELECT us_dollars_to_currency_ratio
        FROM   Currency
        WHERE  currency_name = 'Euro') AS price_in_euros
FROM   Product
WHERE  product_name = 'Cashmere Sweater'
```

All Rows Fetched: 1 in 0.011 seconds

```
   PRICE_IN_EUROS
1              93
```

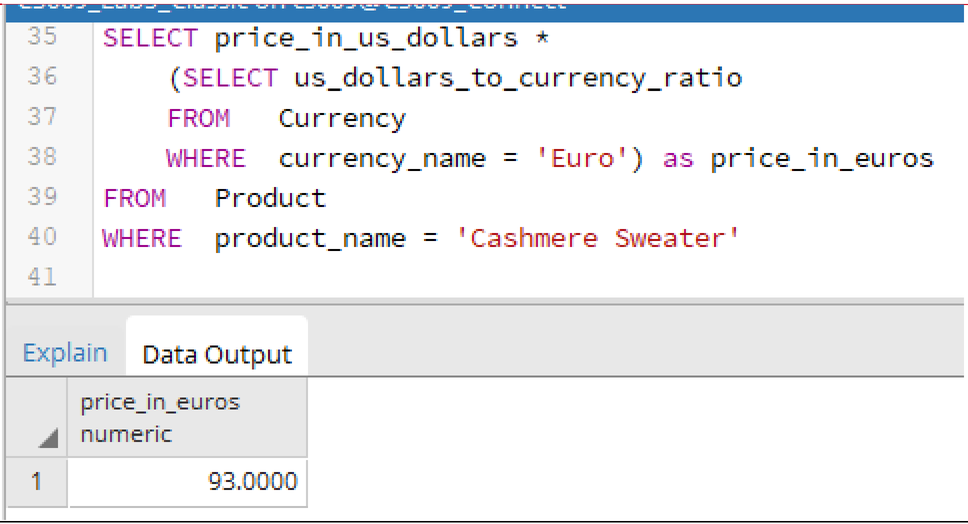
Microsoft SQL Server



```
SELECT price_in_us_dollars *
       (SELECT us_dollars_to_currency_ratio
        FROM   Currency
        WHERE  currency_name = 'Euro') AS price_in_euros
FROM   Product
WHERE  product_name = 'Cashmere Sweater'
```

```
   price_in_euros
1  93.0000
```

PostgreSQL



```
SELECT price_in_us_dollars *
    (SELECT us_dollars_to_currency_ratio
     FROM   Currency
     WHERE  currency_name = 'Euro') as price_in_euros
FROM   Product
WHERE  product_name = 'Cashmere Sweater'
```

```
   price_in_euros
   numeric
1         93.0000
```

The result is the same in these databases, except for the number of decimal points, of course. It is possible to embed one query in another! Let us explore how the two-in-one combination yields its results by examining the query line-by-line.

Code: Single Query for Cashmere Sweater

```
1: SELECT price_in_us_dollars *
2:        (SELECT us_dollars_to_currency_ratio
3:         FROM   Currency
4:         WHERE  currency_name = 'Euro') AS price_in_euros
5: FROM   Product
6: WHERE  product_name = 'Cashmere Sweater'
```

On lines 2-4, we embed the query that retrieves the ratio for Euros. The embedded query is termed a *subquery* because it resides inside of another query. The subquery has the advantage that should the ratio change over time, the overall query will always retrieve the correct results. A significant property of this particular subquery is that it retrieves one column from one row, so it retrieves *one single value* (in this case, the ratio). Not all subqueries retrieve a single value, but this one does, so we can use this subquery wherever a single value is expected. This two-in-one combination is superior because it is more concise and survives changes over time.

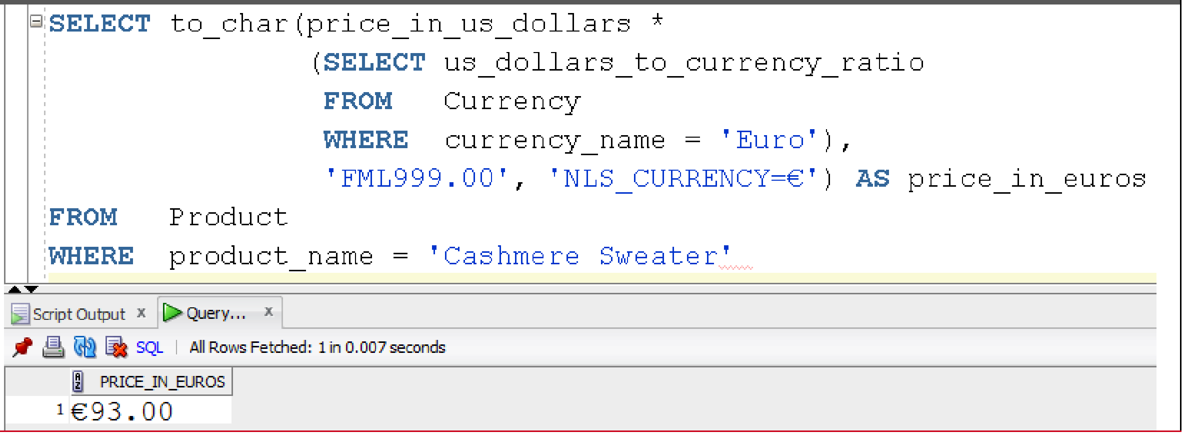
Knowing how to retrieve a single value from a subquery is useful, but knowing *where* to place the subquery is just as important. Placing a subquery in the column list of a SELECT statement gives us the ability to directly manipulate values from every row returned in the outer query. On line 5, we indicate we want to select from the Product table, on line 6, we indicate we only want the Product named “Cashmere Sweater”, and on line 1 we indicate we want the U.S. Dollar price of the sweater. The result of our subquery is thus used to manipulate the U.S. Dollar price for the one row returned from the outer query. However, if the WHERE clause on line 6 were to allow for multiple products, the result of the subquery would be used to manipulate the U.S. Dollar price of all products returned. The principle is that *the result of a subquery placed in the column list is used for every row returned from the outer query*.

Placing a subquery correctly is important, but understanding how the SQL engine executes a subquery empowers us to make the best use subqueries to solve a wider variety of problems. In our example, the result of the subquery does not depend on which product price is retrieved. That is, the ratio of the Euro is the ratio of the Euro; the ratio does not vary if the prices of the products vary. Therefore the SQL engine executes the subquery on its own to retrieve its result. We could state that a subquery will be executed before the outer query is executed, but that is not entirely correct. A subquery may be executed in parallel with the outer query depending upon the DBMS and its configuration, but we do know that a subquery’s result must be available before it is used in the expression in the column list. To be more specific, this type of subquery is termed an *uncorrelated subquery,* which means that the subquery does not reference a table or value in the outer query, and that its results can be retrieved with or without the existence of the outer query. An uncorrelated subquery can always be extracted and executed as a query in its own right. In fact, a simple test to determine whether a subquery is correlated or not is to try and execute it on its own outside of the outer query. The SQL engine executes an uncorrelated subquery independently of the outer query, before it needs the subquery’s results.

For completeness, let us format the result so that we see it as a monetary amount. In Oracle, we would modify the query as the following screenshot illustrates:

Screenshots: Formatted Cashmere Query for Oracle

Oracle



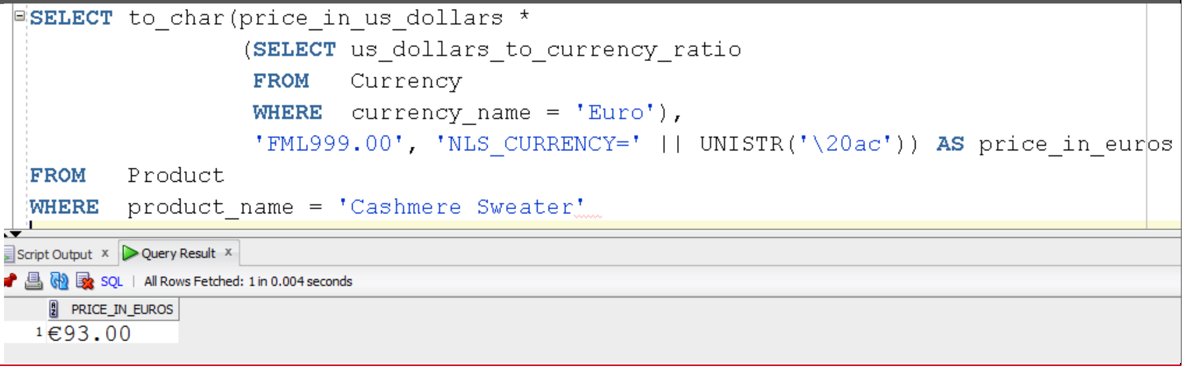
```
SELECT to_char(price_in_us_dollars *
               (SELECT us_dollars_to_currency_ratio
                FROM   Currency
                WHERE  currency_name = 'Euro'),
                'FML999.00', 'NLS_CURRENCY=€') AS price_in_euros
FROM   Product
WHERE  product_name = 'Cashmere Sweater'
```

```
All Rows Fetched: 1 in 0.007 seconds

  PRICE_IN_EUROS
1 €93.00
```

Notice that the result is now “€93.00” instead of “93”. Just as in step 10, we use the to_char function to format our result. What is different than step 10 is the format string, “FML999.00” and the fact that a parameter list follows. The “FM” in the format string instructs the SQL engine to display only as many digits are as necessary (in this example, displaying “93” instead of “093” or the number prefixed with spaces), the “L” indicates to use the local currency symbol (which is specified as a parameter in the next argument), the “999” indicates there may be up to three digits to the left of the decimal point, and the “.00” indicates that there must always be two digits to the right of the decimal point. The parameter list “NLS_CURRENCY=€” indicates the local currency symbol is the Euro symbol. In Windows, holding the Alt key followed by the numbers 0128 on the number pad inserts the Euro symbol. Alternatively, the function UNISTR('\20ac') can be used to insert the Euro symbol without the need to hardcode it by using the Alt key combination, as illustrated below.

Screenshots: Alternative Formatted Cashmere Query for Oracle

Oracle



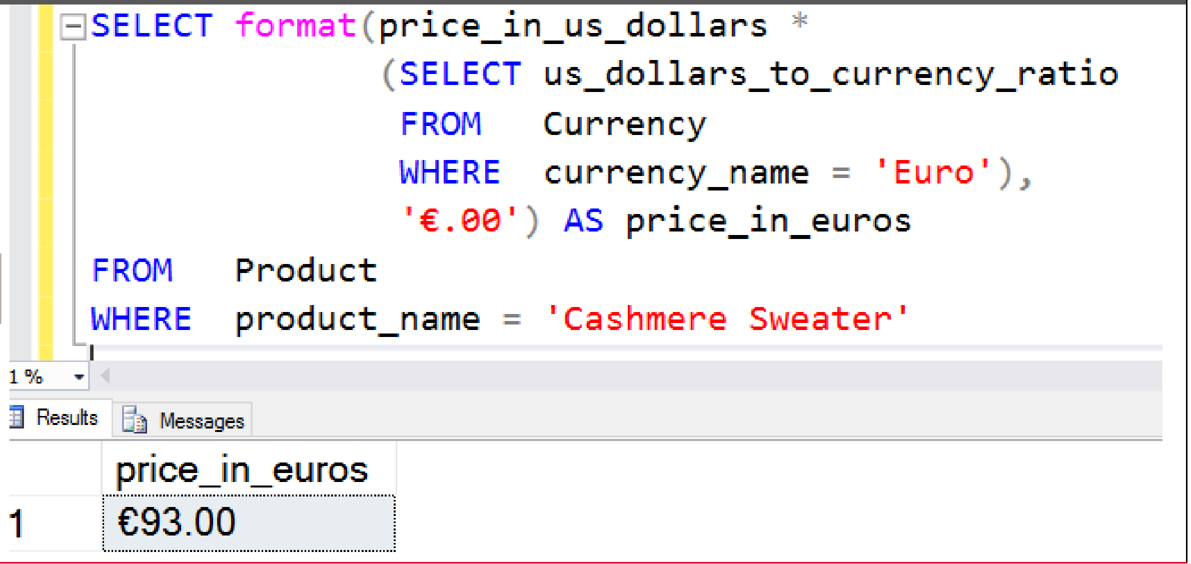
```
SELECT to_char(price_in_us_dollars *
               (SELECT us_dollars_to_currency_ratio
                FROM   Currency
                WHERE  currency_name = 'Euro'),
                'FML999.00', 'NLS_CURRENCY=' || UNISTR('\20ac')) AS price_in_euros
FROM   Product
WHERE  product_name = 'Cashmere Sweater'
```

```
All Rows Fetched: 1 in 0.004 seconds

  PRICE_IN_EUROS
1 €93.00
```

In SQL Server, we would modify the query as the following screenshot indicates:

Screenshots: Formatted Cashmere Query for SQL Server

SQL Server



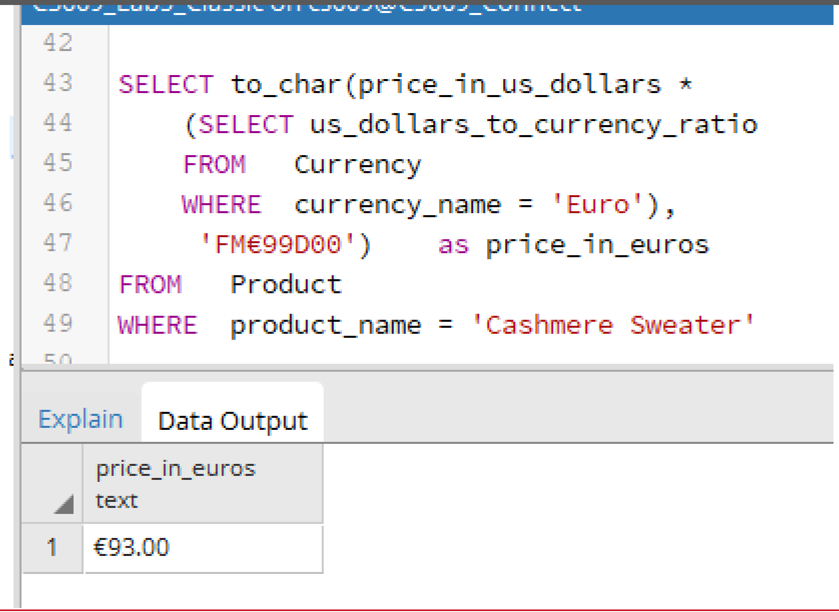
```
SELECT format(price_in_us_dollars *
              (SELECT us_dollars_to_currency_ratio
               FROM   Currency
               WHERE  currency_name = 'Euro'),
               '€.00') AS price_in_euros
FROM    Product
WHERE   product_name = 'Cashmere Sweater'
```

```
  price_in_euros
1 €93.00
```

In PostgreSQL we modify the query as so. FM suppresses padding so the symbol used is next to the numeric value and D is used as the decimal point and pulled from the local setting.

Screenshots: Formatted Cashmere Query for Postgres

Postgres



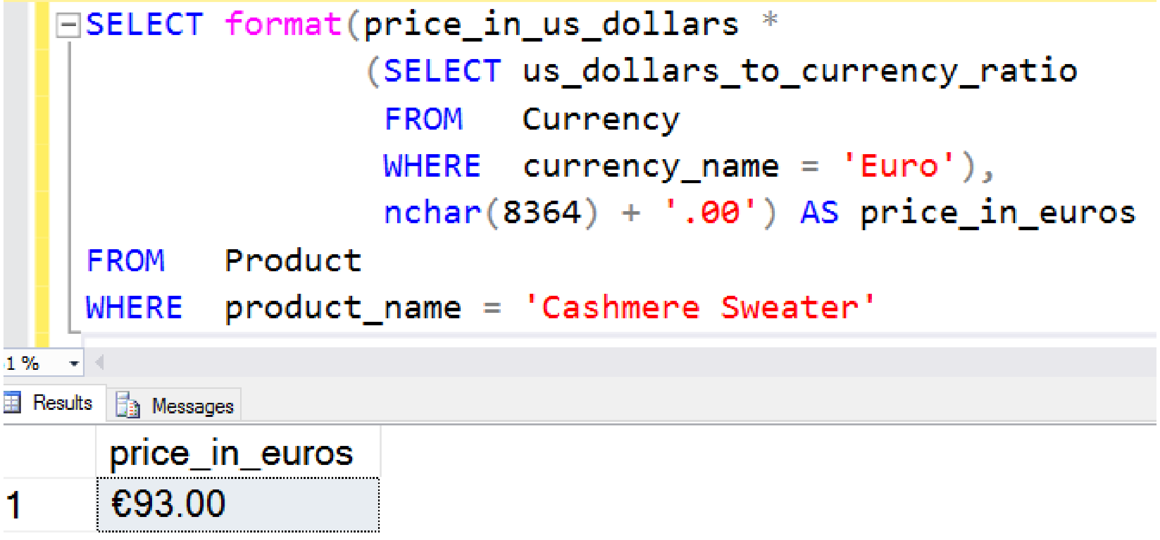
```
SELECT to_char(price_in_us_dollars *
    (SELECT us_dollars_to_currency_ratio
    FROM   Currency
    WHERE  currency_name = 'Euro'),
     'FM€99D00')     as price_in_euros
FROM   Product
WHERE  product_name = 'Cashmere Sweater'
```

```
  price_in_euros
  text
1 €93.00
```

Just like with Oracle, we hold the Alt key followed by the numbers 0128 on the number pad inserts the Euro symbol. The result is now €93.00 instead of 93.0000. We use the format function as illustrated in step 9 with one difference – we use the "€" symbol instead of the "$" symbol. To avoid hardcoding the Euro symbol as a character, we could also concatenate the result of the function nchar(8364), as shown below.

Screenshots: Alternative Formatted Cashmere Query for Postgres

Postgres



```
SELECT format(price_in_us_dollars *
              (SELECT us_dollars_to_currency_ratio
               FROM   Currency
               WHERE  currency_name = 'Euro'),
               nchar(8364) + '.00') AS price_in_euros
FROM   Product
WHERE  product_name = 'Cashmere Sweater'
```

```
     price_in_euros
1    €93.00
```

In this step, we place the subquery in the column list of the outer query, but there are other options for placement. If the situation were to merit it, we could also place the subquery in the WHERE clause, the FROM clause, the ORDER BY clause, and in several other locations in a SQL query. We could also place a subquery inside of another subquery! Where we place a subquery determines the role its results play in the outer query. We explore additional placements in other steps. Nevertheless, you already have a taste of the flexibility and power of that subqueries give you.

You now have enough information to address this step.

STEP 4

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 25 Euros, and for the wealthier group, they would like to know what products cost more than 300 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.
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 step 3, you used a subquery to alter values in the column select list. The new skill you need to address this step is to filter rows based off the results of the subquery. To help address how to do this, imagine that a Mexican customer wants to know the names and prices of all products that cost less than 2,750 Mexican Pesos. We can start by calculating that for all products with the following query.

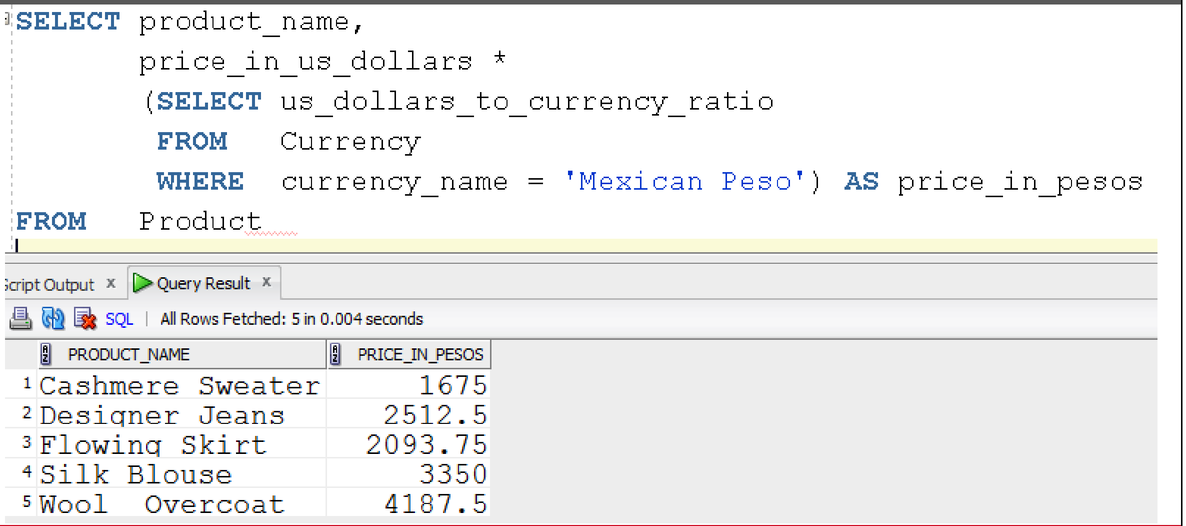
Code: All Products' Prices in Pesos

```
SELECT product_name,
       price_in_us_dollars *
       (SELECT us_dollars_to_currency_ratio
        FROM   Currency
        WHERE  currency_name = 'Mexican Peso') AS price_in_pesos
FROM   Product
```

Just as in the prior step, we use a subquery in the column list to obtain the ratio for Mexican Pesos in order to calculate the product price in Mexican Pesos. The results in Oracle, SQL Server and PostgreSQL respectively, are below.

Screenshots: All Products' Prices in Pesos

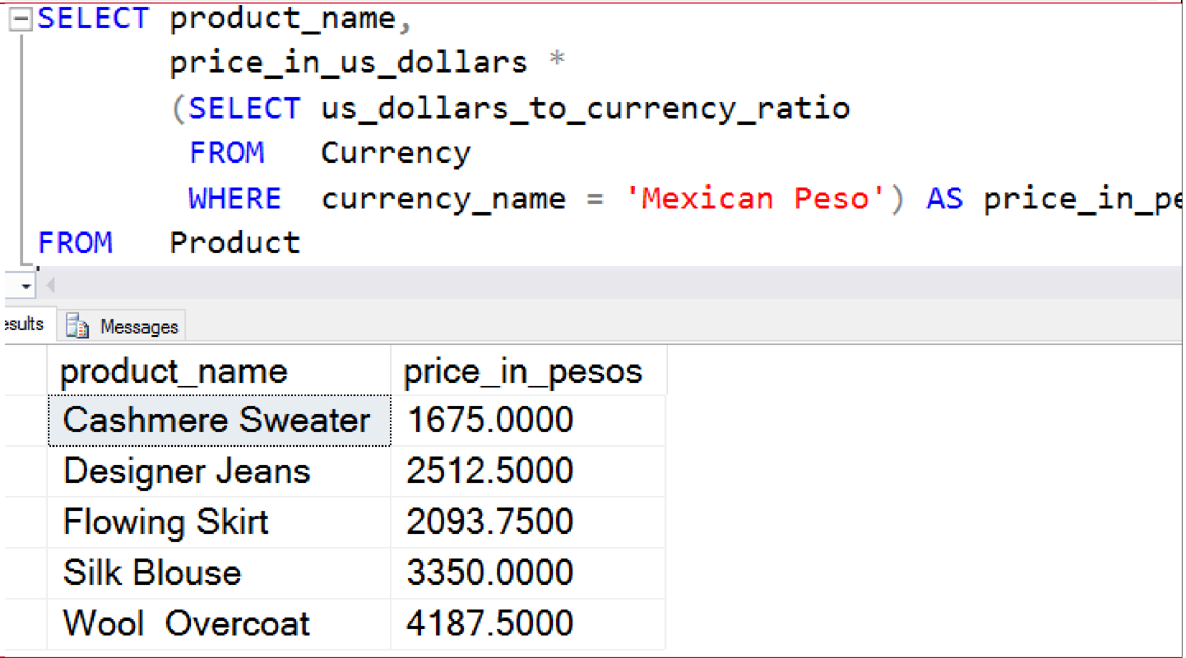
Oracle



```
SELECT product_name,
       price_in_us_dollars *
       (SELECT us_dollars_to_currency_ratio
        FROM   Currency
        WHERE  currency_name = 'Mexican Peso') AS price_in_pesos
FROM   Product
```

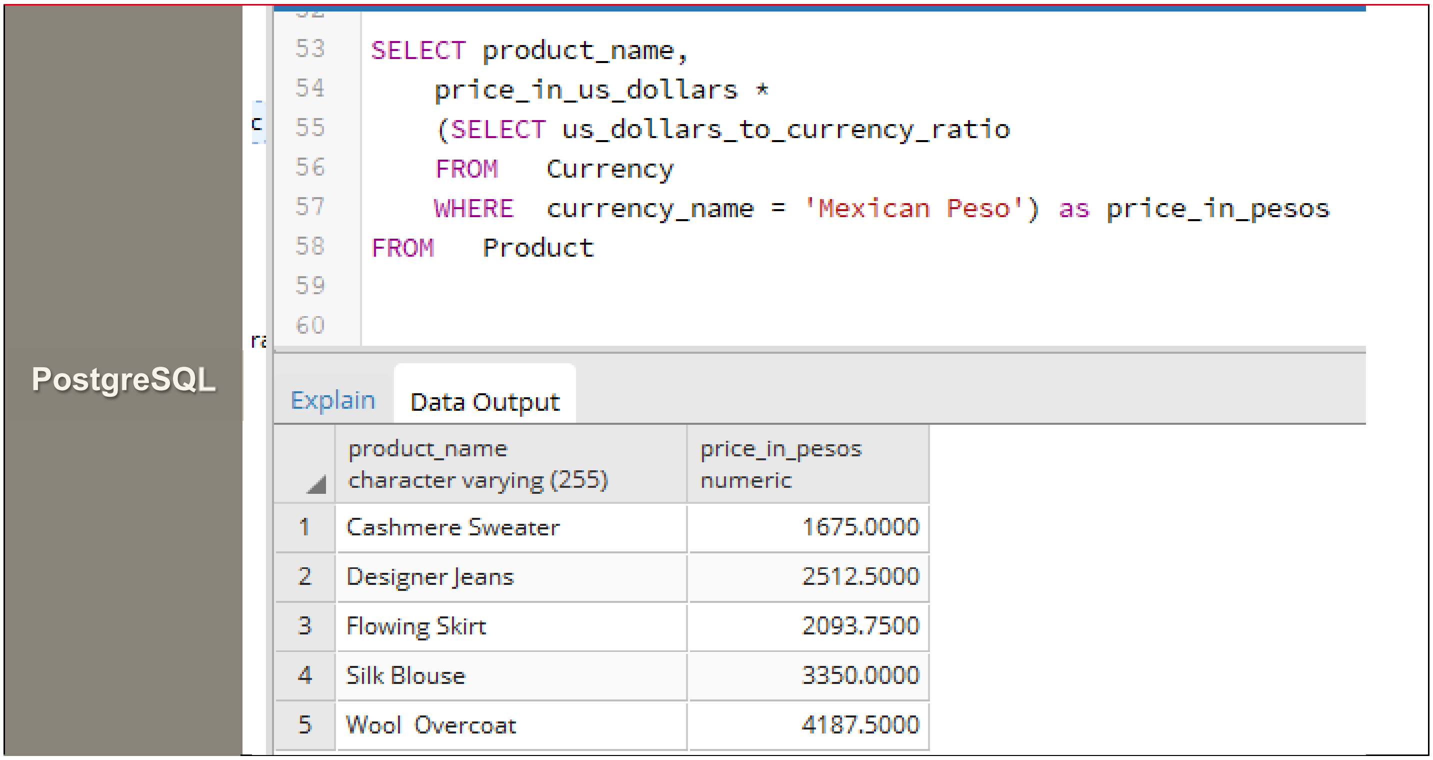
```
All Rows Fetched: 5 in 0.004 seconds

  PRODUCT_NAME      PRICE_IN_PESOS
1 Cashmere Sweater            1675
2 Designer Jeans            2512.5
3 Flowing Skirt            2093.75
4 Silk Blouse                 3350
5 Wool  Overcoat            4187.5
```

Microsoft SQL Server



```
SELECT product_name,
       price_in_us_dollars *
       (SELECT us_dollars_to_currency_ratio
        FROM   Currency
        WHERE  currency_name = 'Mexican Peso') AS price_in_pe
FROM   Product
```

```
product_name       price_in_pesos
Cashmere Sweater   1675.0000
Designer Jeans     2512.5000
Flowing Skirt      2093.7500
Silk Blouse        3350.0000
Wool  Overcoat     4187.5000
```



```
SELECT product_name,
    price_in_us_dollars *
    (SELECT us_dollars_to_currency_ratio
    FROM   Currency
    WHERE  currency_name = 'Mexican Peso') as price_in_pesos
FROM   Product
```

	product_name character varying (255)	price_in_pesos numeric
1	Cashmere Sweater	1675.0000
2	Designer Jeans	2512.5000
3	Flowing Skirt	2093.7500
4	Silk Blouse	3350.0000
5	Wool Overcoat	4187.5000

So far, we've only used the same skill you learned in the prior step to obtain the price of all products in Pesos. To address the use case, we now need to restrict the list to the products costing less than Mex$2,750, which we can do by adding a subquery to the WHERE clause, as shown below.

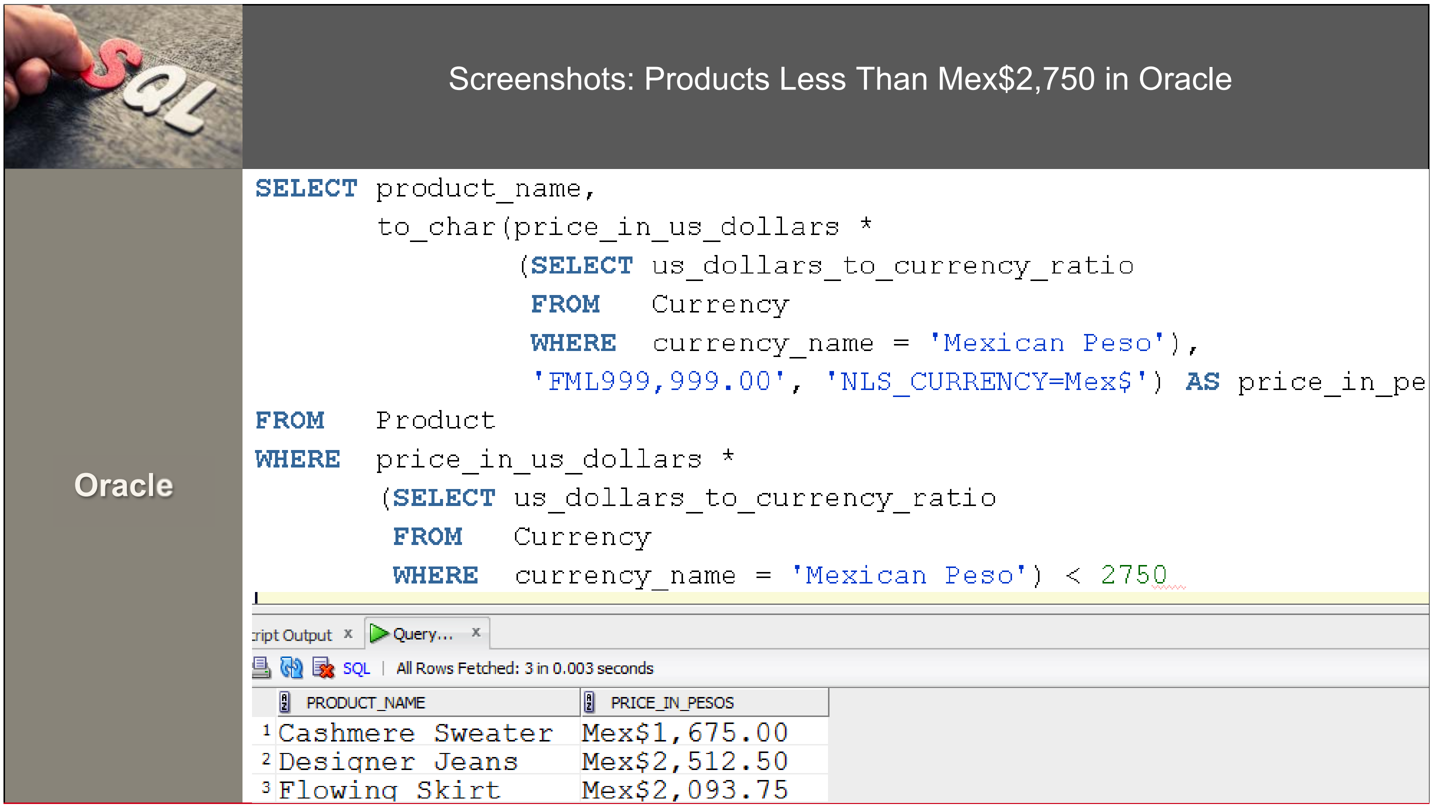
Code: Products Less Than Mex$2,750

```
1: SELECT product_name,
2:        price_in_us_dollars *
3:        (SELECT us_dollars_to_currency_ratio
4:         FROM   Currency
5:         WHERE  currency_name = 'Mexican Peso') AS price_in_pesos
6: FROM   Product
7: WHERE  price_in_us_dollars *
8:        (SELECT us_dollars_to_currency_ratio
9:         FROM   Currency
10:        WHERE  currency_name = 'Mexican Peso') < 2750
```

This query illustrates a construct we have seen – a subquery – used in a clause we have not seen – the WHERE clause. Let us examine the query line by line. Lines 1-6 need no additional explanation beyond the fact that a subquery is used in the column list to calculate the price in Mexican Pesos for each product. Lines 7-10 contain the same subquery used in lines 3-5. That subquery is executed before its result must be used in the WHERE clause, and the subquery's result takes the place of a literal value. The results of that subquery are used to restrict (filter) rows in the result, because it is located in the WHERE clause. Recall that the conditions specified in the WHERE clause are applied to each row, and rows that do not meet the conditions are excluded from the result set. In this specific example, products whose prices are *not* less than Mex$2,750 are excluded. Stated differently, products whose prices are greater than or equal to Mex$2,750 are excluded. Subqueries placed in different clauses are used for different purposes, but the methodology behind how and when they are executed does not change.

This example illustrates another important point, which is that *more than one subquery can be embedded in a single query*. In this example, the first subquery is used to retrieve the price in Mexican Pesos, and the second subquery is to restrict the products retrieved. Each subquery has a useful purpose, and the use of one does not preclude the use of another.

The formatted results are shown for Oracle below.



Screenshots: Products Less Than Mex$2,750 in Oracle

Oracle

```
SELECT product_name,
       to_char(price_in_us_dollars *
               (SELECT us_dollars_to_currency_ratio
                FROM   Currency
                WHERE  currency_name = 'Mexican Peso'),
                'FML999,999.00', 'NLS_CURRENCY=Mex$') AS price_in_pe
FROM   Product
WHERE  price_in_us_dollars *
       (SELECT us_dollars_to_currency_ratio
        FROM   Currency
        WHERE  currency_name = 'Mexican Peso') < 2750
```

All Rows Fetched: 3 in 0.003 seconds

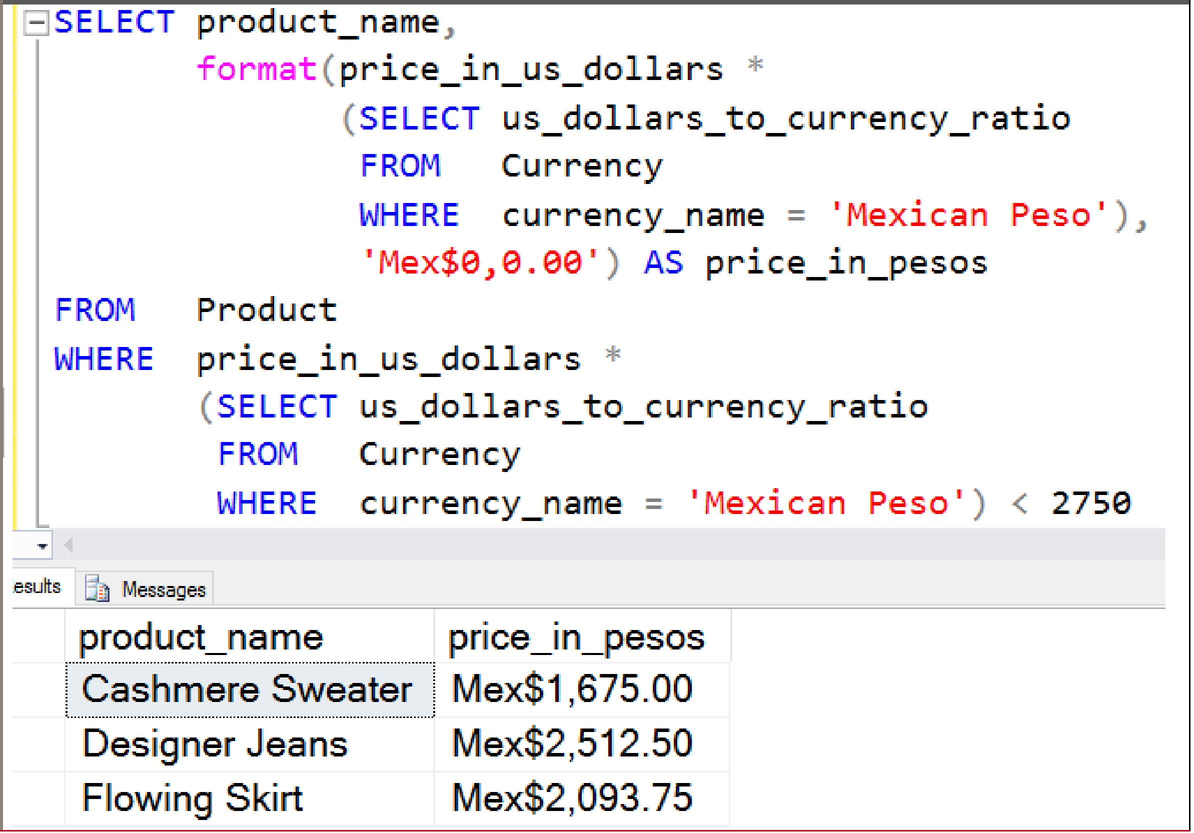
```
  PRODUCT_NAME      PRICE_IN_PESOS
1 Cashmere Sweater  Mex$1,675.00
2 Designer Jeans    Mex$2,512.50
3 Flowing Skirt     Mex$2,093.75
```

Notice that, because the results are in the thousands, we use three additional 9 digits with a comma separator in the format string, to properly format the result. We also use “Mex$” instead of the U.S. Dollar or Euro symbol.

The formatted results for SQL Server are shown below.

Screenshots: Products Less Than Mex$2,750 in SQL Server

SQL Server



```
SELECT product_name,
       format(price_in_us_dollars *
              (SELECT us_dollars_to_currency_ratio
               FROM   Currency
               WHERE  currency_name = 'Mexican Peso'),
               'Mex$0,0.00') AS price_in_pesos
FROM   Product
WHERE  price_in_us_dollars *
       (SELECT us_dollars_to_currency_ratio
        FROM   Currency
        WHERE  currency_name = 'Mexican Peso') < 2750
```

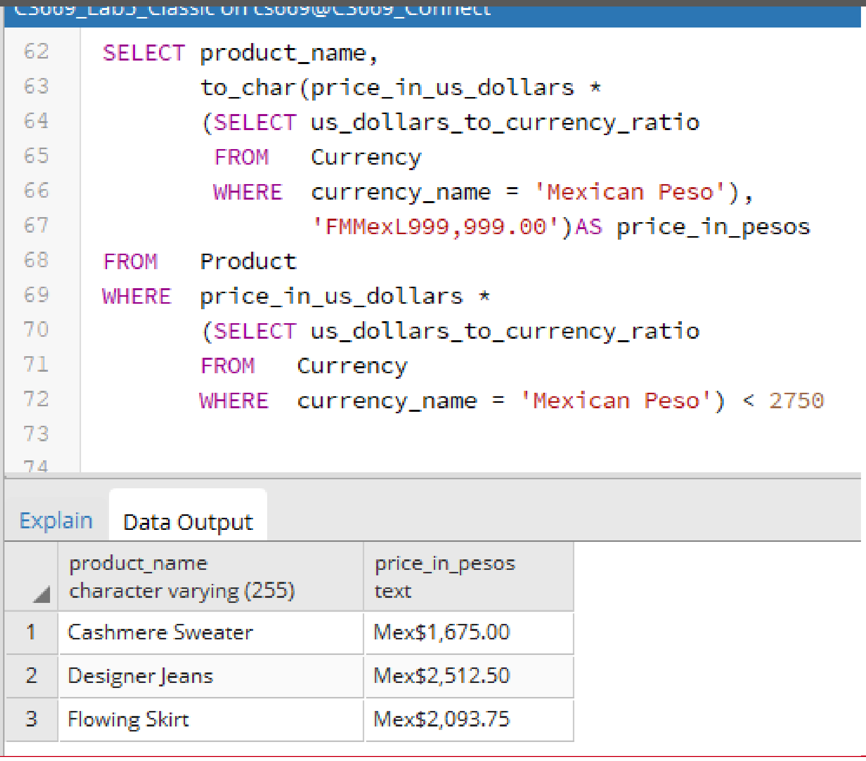
product_name	price_in_pesos
Cashmere Sweater	Mex$1,675.00
Designer Jeans	Mex$2,512.50
Flowing Skirt	Mex$2,093.75

The only formatting change for SQL Server compared to prior steps is that we use “Mex$” in the format string, and add “0,0” to the left of the decimal point so that each group of 3 numbers is separated by a comma.

The formatted results for PostgreSQL are shown below.

Screenshots: Products Less Than Mex$2,750 in Postgres

Postgres



```
SELECT product_name,
       to_char(price_in_us_dollars *
       (SELECT us_dollars_to_currency_ratio
        FROM   Currency
        WHERE  currency_name = 'Mexican Peso'),
               'FMMexL999,999.00')AS price_in_pesos
FROM   Product
WHERE  price_in_us_dollars *
       (SELECT us_dollars_to_currency_ratio
       FROM   Currency
       WHERE  currency_name = 'Mexican Peso') < 2750
```

	product_name character varying (255)	price_in_pesos text
1	Cashmere Sweater	Mex$1,675.00
2	Designer Jeans	Mex$2,512.50
3	Flowing Skirt	Mex$2,093.75

Notice that in each screenshot, each product in the list cost less then Mex$2,750, because of the subquery in the WHERE clause.

You can use similar logic to solve to address your use case for this step.

STEP 5

5. Imagine that Denisha 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 Denisha 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).
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 Denisha'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.

This use case is getting more complex! As you have already learned, deciding where to place a subquery is important, yet deciding *how many values* to retrieve in the subquery is equally important. In prior steps our subqueries always retrieve a single value, but single-valued subqueries cannot practically address all use cases. We need, and thankfully have, more options! In this step we examine retrieving a list of values, and in subsequent steps we examine retrieving results in tabular form. Using subqueries to address use cases requires skill and necessitates making many decisions.

Understanding the concept of a single value is straightforward, but what about a list of values? Simply put, a list of values in a relational database is a tabular construct that consists of exactly one column with the one or more rows. In contrast, a single value consists of exactly one column and exactly one row. When we create a subquery, we decide the maximum number of rows and columns it may retrieve. If a subquery retrieves one column, then we have the option to retrieve a single value by ensuring the subquery always retrieves exactly one row, and we have the option to retrieve a list of values by allowing it to retrieve as many rows as are needed. For example, the subquery

```
SELECT last_name FROM Person WHERE person_id = 5
```

would presumably retrieve a single value, because it restricts the number of rows retrieved by a single primary key value. But the subquery

```
SELECT last_name FROM Person WHERE weight_in_pounds < 130
```

would presumably retrieve a list of values, because there would be many people that weigh less than 130 pounds. We control whether a subquery retrieves a list of values or not by how we write it.

Whatever our decision, we must ensure that the outer query uses the correct construct to handle the result of the subquery based upon the number of values returned. The equality operator generally is only used to compare single values. For example, the test “Does the value in column X equal 5?” makes sense, but the test “Does the value in column X equal the list of numbers 5, 10, and 20?” does not make sense. Using SQL syntax, we would say that “**WHERE X = 5**” makes sense, but “**WHERE X = (1, 2, 3, 4)**” does not make sense. We can however use the **IN** operator instead, “**WHERE X IN (5, 10, 20)**”. The **IN** operator tests whether a single value is found in a list of values. If X is 5 or 10 or 20, then “**X IN (5, 10, 20)**” is true; otherwise, it is false. Some constructs in SQL only work with single values, and some work with lists of values, and we must use the correct class of constructs with each subquery we create.

Subqueries in conjunction with appropriate SQL constructs can oftentimes be used to address use cases that have distinct and dissimilar parts. Let’s look at a more complex use case and solve it, to help you address this step.

> Jill travels internationally and is considering purchasing some items from some of the store locations while on her travels. She wants flexibility in how she can order and receive the items, so she would like to see the list of products and prices (in U.S. Dollars) *only* for store locations that offer more than one purchase and delivery option.

This use case is a challenge because it has two parts that require different SQL strategies. In order to determine which locations are suitable for Jill, we need to count the number of purchase and delivery options offered by each location, which means we need to aggregate results. However, because aggregation hides line-by-line details in favor of summarized results, we need to avoid it in order to obtain line-by-line product information for each store. So what do we do about this apparent conflict? You guessed it. Use a subquery! More complex use cases require more complex SQL strategies.

A good way to craft a query to address a more complex use case is to create one independent query for each distinct part, then put them together. For this example, we first create a query that finds the right stores, then create a query that lists the names and prices of products of all stores, then combine the two.

Determining which store locations offer more than one purchase and delivery option is solvable by use of a GROUP BY coupled with a HAVING clause as shown below.

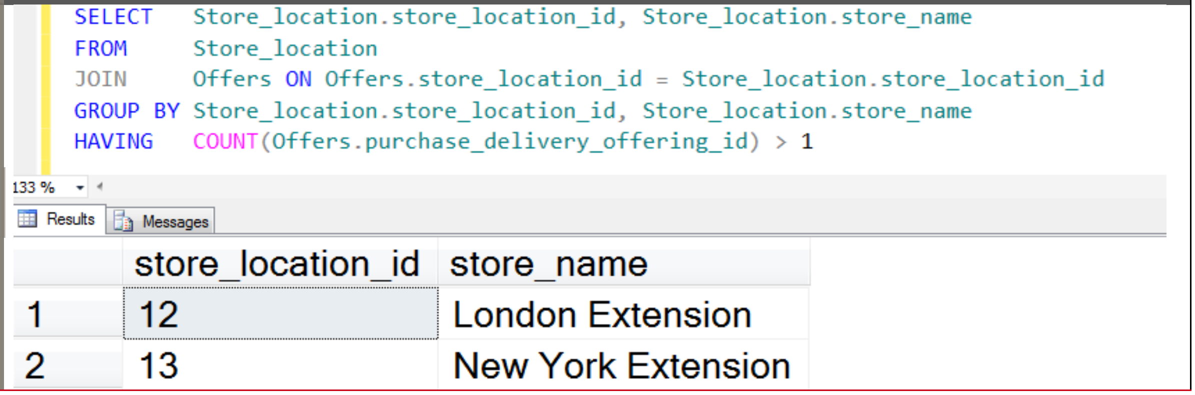
Code: Store Locations Query

```
SELECT   Store_location.store_location_id, Store_location.store_name
FROM     Store_location
JOIN     Offers ON Offers.store_location_id = Store_location.store_location_id
GROUP BY Store_location.store_location_id, Store_location.store_name
HAVING   COUNT(Offers.purchase_delivery_offering_id) > 1
```

On line 4, the GROUP BY groups the result set by the store_location_id and secondarily by the store_name, and the HAVING limits the results returned to those stores with more than one purchase and delivery option. Executing this query yields results similar to the screenshot below in both Oracle, SQL Server and PostgreSQL, illustrating that the London Extension and the New York Extension both have more than one purchase and delivery offering (only the SQL Server screenshot is illustrated for brevity).

Screenshots: Store Locations Query

SQL Server



```
SELECT    Store_location.store_location_id, Store_location.store_name
FROM      Store_location
JOIN      Offers ON Offers.store_location_id = Store_location.store_location_id
GROUP BY Store_location.store_location_id, Store_location.store_name
HAVING    COUNT(Offers.purchase_delivery_offering_id) > 1
```

	store_location_id	store_name
1	12	London Extension
2	13	New York Extension

Listing the products and prices for all stores is straightforward and only requires basic joins, as illustrated below.

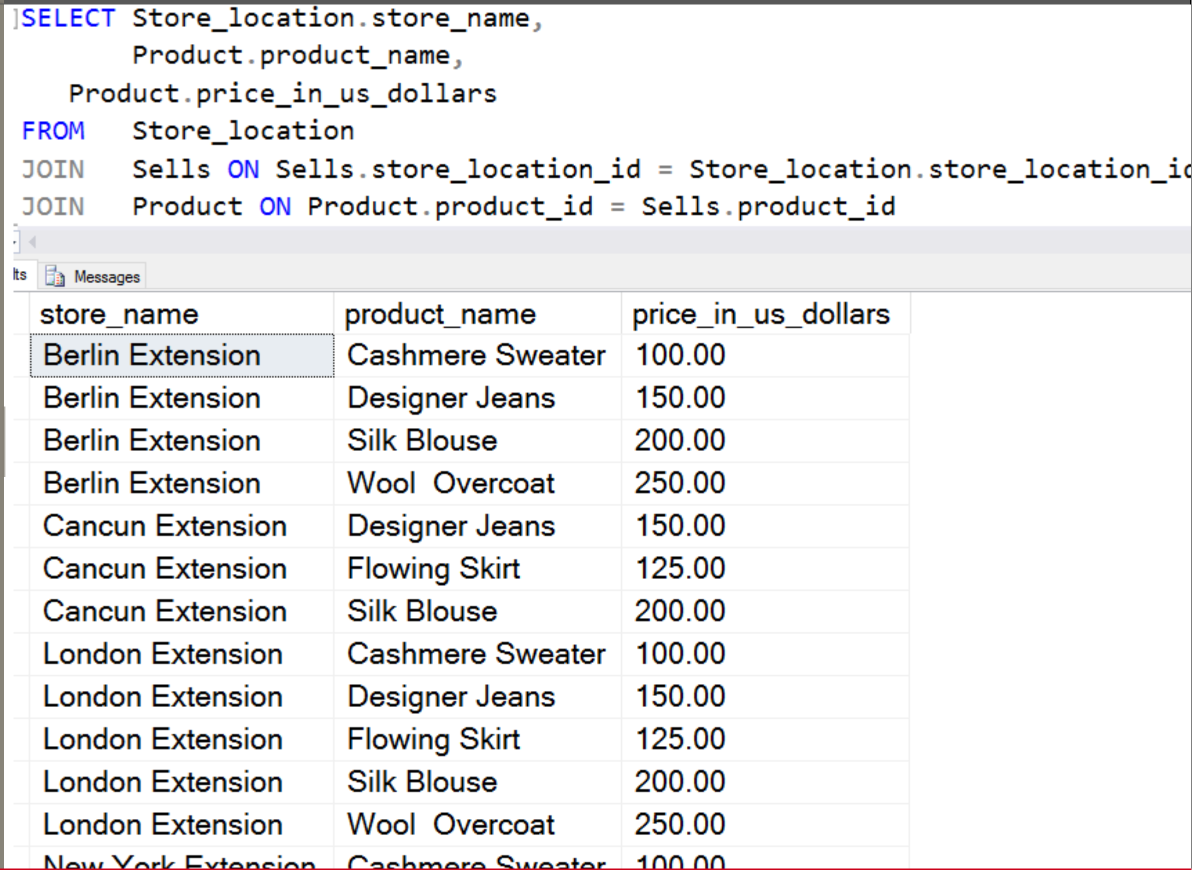
Code: Products and Prices Query

```
1: SELECT   Store_location.store_location_id, Store_location.store_name
2: FROM     Store_location
3: JOIN     Offers ON Offers.store_location_id = Store_location.store_location_id
4: GROUP BY Store_location.store_location_id, Store_location.store_name
5: HAVING   COUNT(Offers.purchase_delivery_offering_id) > 1
```

Notice that we join Store_location to Sells to Product, and list out each store's name, product name, and product price. The query's execution is illustrated below in SQL Server (Oracle and Postgres give the same results). The results are truncated for brevity.

Screenshots: Products and Prices Query

SQL Server



```
SELECT Store_location.store_name,
       Product.product_name,
    Product.price_in_us_dollars
FROM   Store_location
JOIN   Sells ON Sells.store_location_id = Store_location.store_location_i
JOIN   Product ON Product.product_id = Sells.product_id
```

store_name	product_name	price_in_us_dollars
Berlin Extension	Cashmere Sweater	100.00
Berlin Extension	Designer Jeans	150.00
Berlin Extension	Silk Blouse	200.00
Berlin Extension	Wool Overcoat	250.00
Cancun Extension	Designer Jeans	150.00
Cancun Extension	Flowing Skirt	125.00
Cancun Extension	Silk Blouse	200.00
London Extension	Cashmere Sweater	100.00
London Extension	Designer Jeans	150.00
London Extension	Flowing Skirt	125.00
London Extension	Silk Blouse	200.00
London Extension	Wool Overcoat	250.00
New York Extension	Cashmere Sweater	100.00

Finally, we combine both queries to retrieve the results we need, as illustrated below.

Code: Full Query

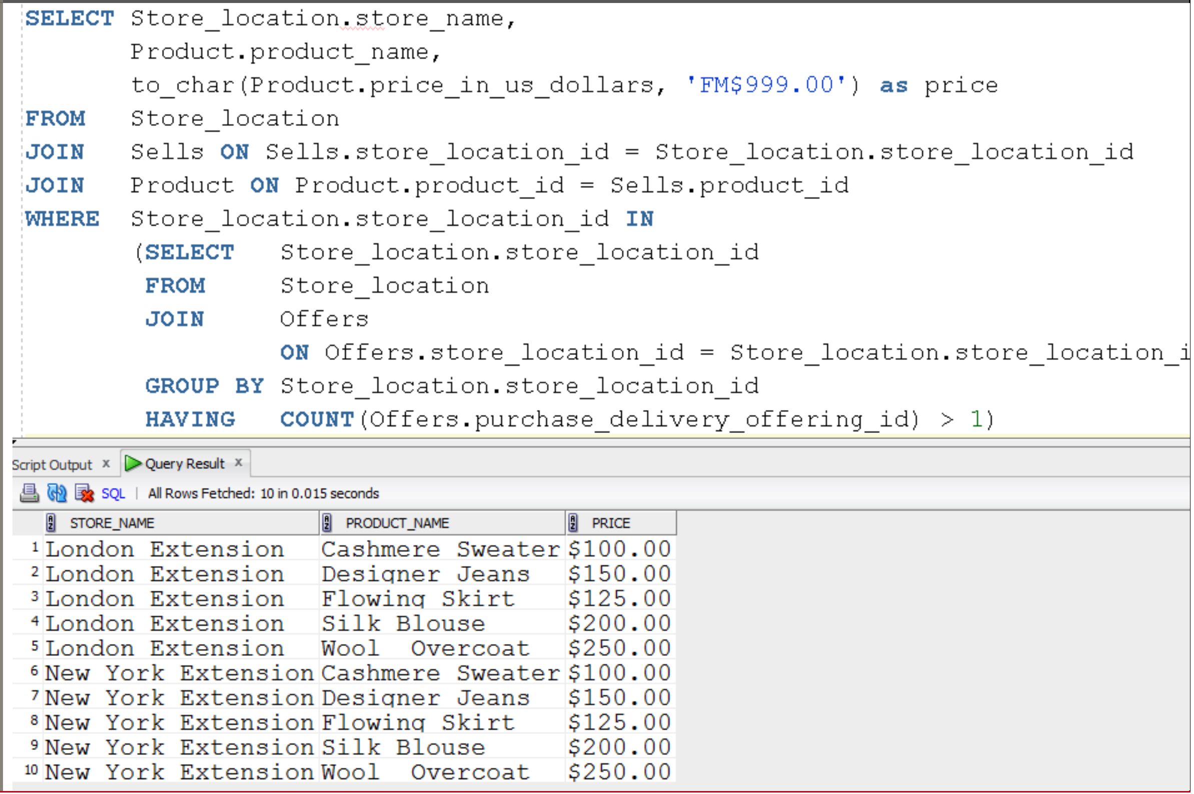
```
1: SELECT Store_location.store_name,
2:        Product.product_name,
3:       Product.price_in_us_dollars
4: FROM   Store_location
5: JOIN   Sells ON Sells.store_location_id = Store_location.store_location_id
6: JOIN   Product ON Product.product_id = Sells.product_id
7: WHERE  Store_location.store_location_id IN
8:        (SELECT   Store_location.store_location_id
9:         FROM     Store_location
10:        JOIN     Offers
11:                 ON Offers.store_location_id = Store_location.store_location_id
12:        GROUP BY Store_location.store_location_id
13:        HAVING   COUNT(Offers.purchase_delivery_offering_id) > 1)
```

On lines 8-13, the first query is embedded as a subquery with one change: the name of the store is not retrieved. This is because when we use the query in a standalone fashion, we want to see the name of the stores (seeing the store ID alone is not helpful), but when we embed the query, we want only to retrieve the store IDs so that the outer query can limit what it retrieves by those IDs. On lines 1-6 you see the second query, the one that lists all products, is present without any changes compared to the original. Line 7 is where this query gets interesting; it is the glue that causes the two queries to work together. The outer query only retrieves the products of stores returned by the subquery by ensuring that the outer query's store_location_id is in the list of the ids returned by the subquery. The "**`Store_location.store_location_id IN`**" part of line 7 sets up the condition that the store_location_id must be in the list of values that follow, and the list of values that follow are determined by the subquery. Think about the last two sentences until you are sure you understand how these queries work together; it is essential you understand how to glue two queries together as illustrated in this example. Combining queries to solve complex use cases is powerful, but also requires skilled knowledge of SQL constructs and mechanisms.

Execution of the combined query is shown for Oracle below, with the price formatted as a monetary amount.

Screenshots: Full Query Execution in Oracle

Oracle



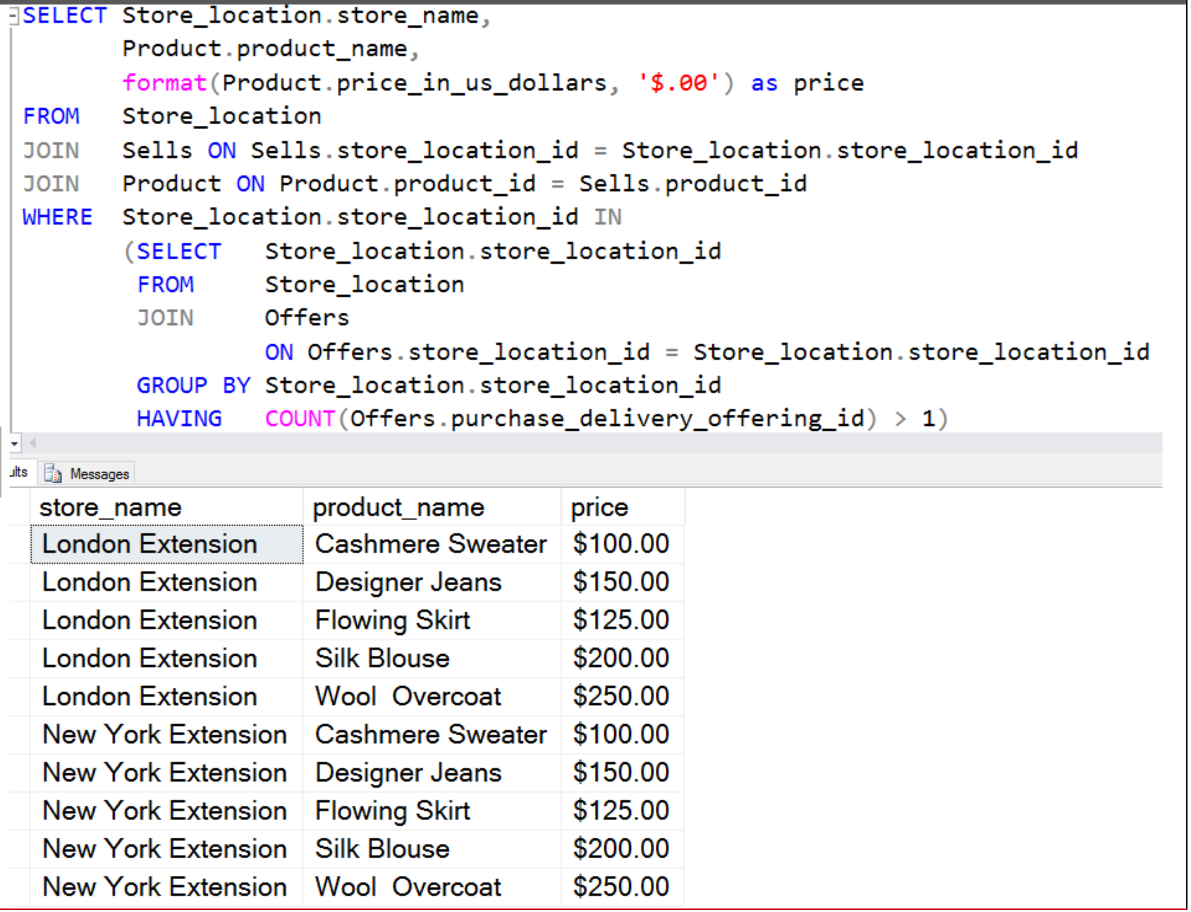
```
SELECT Store_location.store_name,
       Product.product_name,
       to_char(Product.price_in_us_dollars, 'FM$999.00') as price
FROM   Store_location
JOIN   Sells ON Sells.store_location_id = Store_location.store_location_id
JOIN   Product ON Product.product_id = Sells.product_id
WHERE  Store_location.store_location_id IN
       (SELECT    Store_location.store_location_id
        FROM      Store_location
        JOIN      Offers
                  ON Offers.store_location_id = Store_location.store_location_i
        GROUP BY Store_location.store_location_id
        HAVING    COUNT(Offers.purchase_delivery_offering_id) > 1)
```

```
All Rows Fetched: 10 in 0.015 seconds

   STORE_NAME         PRODUCT_NAME      PRICE
 1 London Extension   Cashmere Sweater $100.00
 2 London Extension   Designer Jeans   $150.00
 3 London Extension   Flowing Skirt    $125.00
 4 London Extension   Silk Blouse      $200.00
 5 London Extension   Wool  Overcoat   $250.00
 6 New York Extension Cashmere Sweater $100.00
 7 New York Extension Designer Jeans   $150.00
 8 New York Extension Flowing Skirt    $125.00
 9 New York Extension Silk Blouse      $200.00
10 New York Extension Wool  Overcoat   $250.00
```

Notice that only the products and prices for the London Extension and New York Extension are list in the results. Execution in SQL Server is shown below, and the results are the same.

Screenshots: Full Query Execution in SQL Server

SQL Server



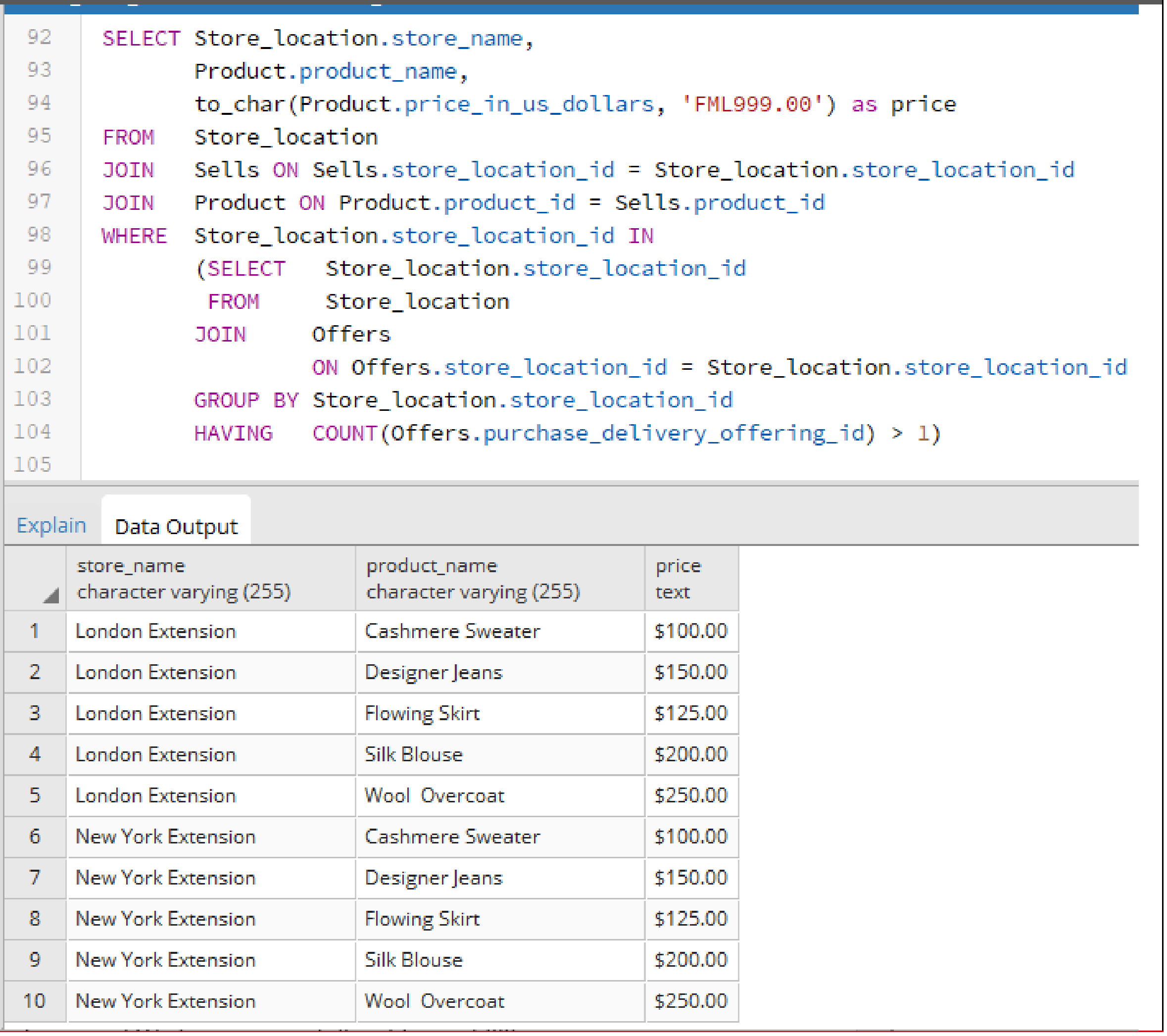
```sql
SELECT Store_location.store_name,
       Product.product_name,
       format(Product.price_in_us_dollars, '$.00') as price
FROM   Store_location
JOIN   Sells ON Sells.store_location_id = Store_location.store_location_id
JOIN   Product ON Product.product_id = Sells.product_id
WHERE  Store_location.store_location_id IN
       (SELECT   Store_location.store_location_id
        FROM     Store_location
        JOIN     Offers
                 ON Offers.store_location_id = Store_location.store_location_id
        GROUP BY Store_location.store_location_id
        HAVING   COUNT(Offers.purchase_delivery_offering_id) > 1)
```

store_name	product_name	price
London Extension	Cashmere Sweater	$100.00
London Extension	Designer Jeans	$150.00
London Extension	Flowing Skirt	$125.00
London Extension	Silk Blouse	$200.00
London Extension	Wool Overcoat	$250.00
New York Extension	Cashmere Sweater	$100.00
New York Extension	Designer Jeans	$150.00
New York Extension	Flowing Skirt	$125.00
New York Extension	Silk Blouse	$200.00
New York Extension	Wool Overcoat	$250.00

Execution in PostgreSQL is shown below, and the results are the same.

Screenshots: Full Query Execution in Postgres

Postgres



```
SELECT Store_location.store_name,
       Product.product_name,
       to_char(Product.price_in_us_dollars, 'FML999.00') as price
FROM   Store_location
JOIN   Sells ON Sells.store_location_id = Store_location.store_location_id
JOIN   Product ON Product.product_id = Sells.product_id
WHERE  Store_location.store_location_id IN
       (SELECT   Store_location.store_location_id
        FROM     Store_location
       JOIN     Offers
                ON Offers.store_location_id = Store_location.store_location_id
       GROUP BY Store_location.store_location_id
       HAVING   COUNT(Offers.purchase_delivery_offering_id) > 1)
```

	store_name character varying (255)	product_name character varying (255)	price text
1	London Extension	Cashmere Sweater	$100.00
2	London Extension	Designer Jeans	$150.00
3	London Extension	Flowing Skirt	$125.00
4	London Extension	Silk Blouse	$200.00
5	London Extension	Wool Overcoat	$250.00
6	New York Extension	Cashmere Sweater	$100.00
7	New York Extension	Designer Jeans	$150.00
8	New York Extension	Flowing Skirt	$125.00
9	New York Extension	Silk Blouse	$200.00
10	New York Extension	Wool Overcoat	$250.00

This query gives Jill what she wants! We have successfully addressed Jill's use case using a subquery embedded in an outer query, and she now knows about all products that are sold in locations that offer more than one purchase and delivery option.

How does all of this help you address this step? In our example, you saw that we tackled the problem by creating two different queries first, then combining them to get the results we want. This can be a helpful strategy to use when you need to make use of uncorrelated subqueries, to help break down the problem into manageable chunks. You also saw us make use of the IN clause to "glue" one query to another, when the subquery returns more than one value. You also saw make use of the GROUP BY and HAVING clauses within a subquery, illustrating that a subquery can aggregate the results if needed.

Armed with these skills, you can now address this step.

STEP 6

For this problem you will write a single query to address the same use case as in step 5, 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 5, 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 5.

You've worked with subqueries that retrieve single values and lists of values, but what about tables of values? That's what this step is about. Recall that all queries, subqueries included, return tabular results in the form of rows and columns. If a subquery returns one column and one row, the result can be treated as a single value. Likewise, results with one column and one or more rows can be treated as a list of values. However, there are no restrictions on the number of rows or columns when tabular results are expected, since the results are by definition tabular. In particular, the FROM clause always expects tabular elements, so a subquery can be used in the FROM clause without regard to the number of columns and rows it retrieves. Constructs that expect tables of values allow for flexible subquery creation.

Let us review how a subquery placed in the WHERE clause can filter rows in advanced ways by reviewing the example use case we solved in step 6.

> Jill travels internationally and is considering purchasing some items from some of the store locations while on her travels. She wants flexibility in how she can order and receive the items, so she would like to see the list of products and prices (in U.S. Dollars) *only* for store locations that offer more than one purchase and delivery option.

Next, let us review the solution from step 6 which makes use of a subquery in the WHERE clause.

Code: Full Query with WHERE Clause Subquery

```
1: SELECT Store_location.store_name,
2:        Product.product_name,
3:       Product.price_in_us_dollars
4: FROM   Store_location
5: JOIN   Sells ON Sells.store_location_id = Store_location.store_location_id
6: JOIN   Product ON Product.product_id = Sells.product_id
7: WHERE  Store_location.store_location_id IN
8:        (SELECT   Store_location.store_location_id
9:         FROM     Store_location
10:        JOIN     Offers
11:                ON Offers.store_location_id = Store_location.store_location_id
12:        GROUP BY Store_location.store_location_id
13:        HAVING   COUNT(Offers.purchase_delivery_offering_id) > 1)
```

We include the Store_location table on line 4 so that we can retrieve the store name and also match up the locations to the products they sell. We restrict the store locations retrieved based upon the results in the subquery on lines 8-13, thus employing the subquery as a filtering mechanism. The combined query correctly addresses the use case.

It makes sense that a subquery in the WHERE clause can be employed as an advanced filtering mechanism since filtering conditions in general are placed in the WHERE clause, but you may be surprised to know that a *subquery in the FROM clause can serve the same purpose*. This is more easily explained by example, so let us look at an alternative solution to Jill’s use case.

Code: Subquery in FROM Clause

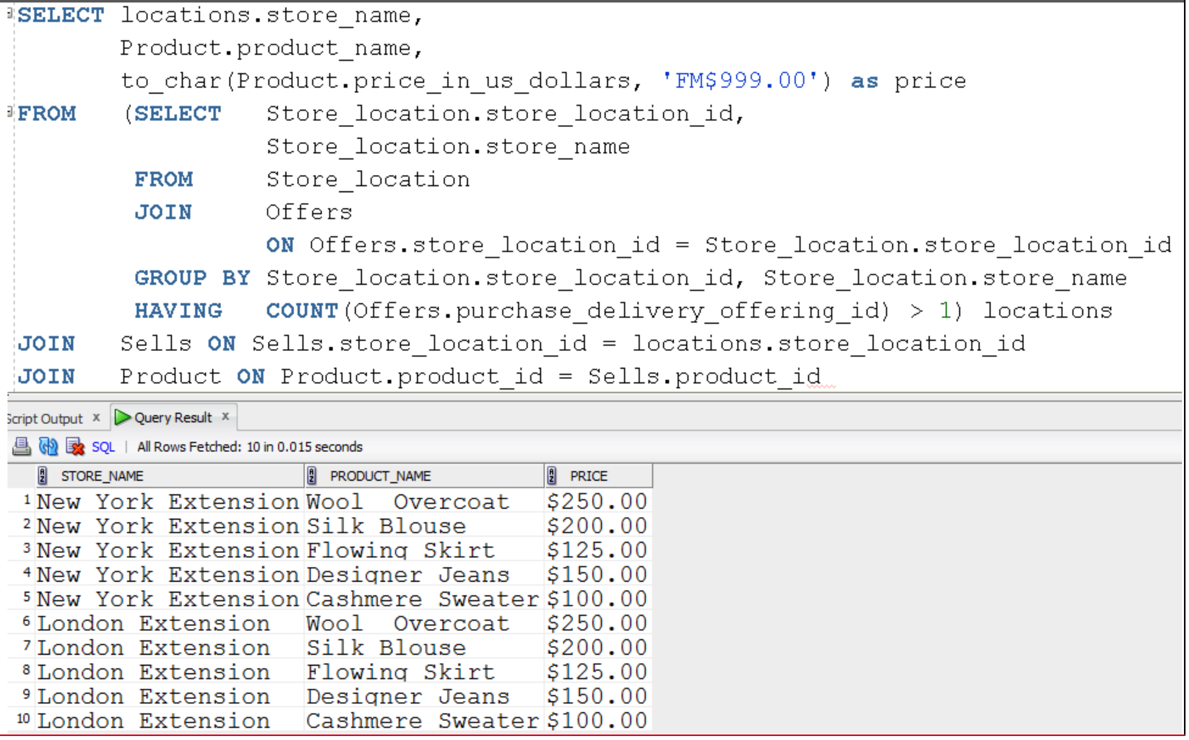
```
1:  SELECT locations.store_name,
2:         Product.product_name,
3:         Product.price_in_us_dollars
4:  FROM   (SELECT   Store_location.store_location_id,
5:                   Store_location.store_name
6:          FROM     Store_location
7:          JOIN     Offers
8:                 ON Offers.store_location_id = Store_location.store_location_id
9:          GROUP BY Store_location.store_location_id, Store_location.store_name
10:         HAVING   COUNT(Offers.purchase_delivery_offering_id) > 1) locations
11: JOIN   Sells ON Sells.store_location_id = locations.store_location_id
12: JOIN   Product ON Product.product_id = Sells.product_id
```

Notice that the WHERE clause subquery in the original solution is moved to the FROM clause, to lines 4-10. The subquery retrieves the id and name of each store location that matches Jill’s criteria, which are locations that have more than one purchase and delivery option. The word “locations” on line 10 is an *alias* which provides a name for the subquery’s results. Once defined, the alias can be used as if it were a table, and that table consists of whatever rows and columns are retrieved by the subquery. On line 11, the “locations” alias is used as a part of the join condition, to join the results from the subquery into the Sells table. Because the subquery only retrieves locations matching Jill’s criteria, the overall query does the same, and filtering has been achieved in the FROM clause rather than the WHERE clause. Subqueries placed in the FROM clause can be flexible and powerful constructs.

The screenshot below shows execution of the query in Oracle with a formatted price.

Screenshots: Subquery in FROM Clause in Oracle

Oracle



```
SELECT locations.store_name,
       Product.product_name,
       to_char(Product.price_in_us_dollars, 'FM$999.00') as price
FROM   (SELECT   Store_location.store_location_id,
                 Store_location.store_name
        FROM     Store_location
        JOIN     Offers
                 ON Offers.store_location_id = Store_location.store_location_id
        GROUP BY Store_location.store_location_id, Store_location.store_name
        HAVING   COUNT(Offers.purchase_delivery_offering_id) > 1) locations
JOIN   Sells ON Sells.store_location_id = locations.store_location_id
JOIN   Product ON Product.product_id = Sells.product_id
```

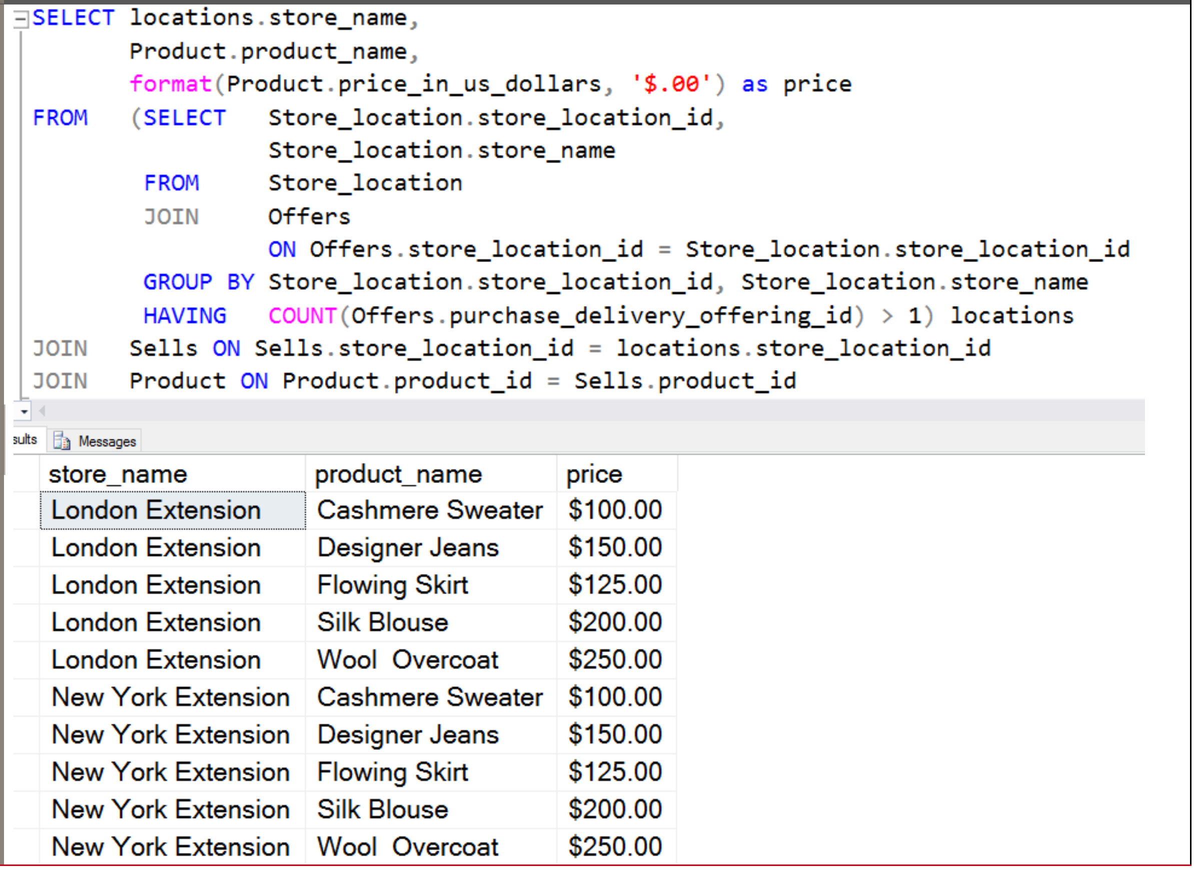
All Rows Fetched: 10 in 0.015 seconds

```
    STORE_NAME          PRODUCT_NAME       PRICE
 1  New York Extension  Wool  Overcoat     $250.00
 2  New York Extension  Silk Blouse        $200.00
 3  New York Extension  Flowing Skirt      $125.00
 4  New York Extension  Designer Jeans     $150.00
 5  New York Extension  Cashmere Sweater   $100.00
 6  London Extension    Wool  Overcoat     $250.00
 7  London Extension    Silk Blouse        $200.00
 8  London Extension    Flowing Skirt      $125.00
 9  London Extension    Designer Jeans     $150.00
10  London Extension    Cashmere Sweater   $100.00
```

The same results are retrieved as in the original solution with the exception of row ordering, which is insignificant. The screenshot below shows execution of the query in SQL Server.

Screenshots: Subquery in FROM Clause in SQL Server

SQL Server



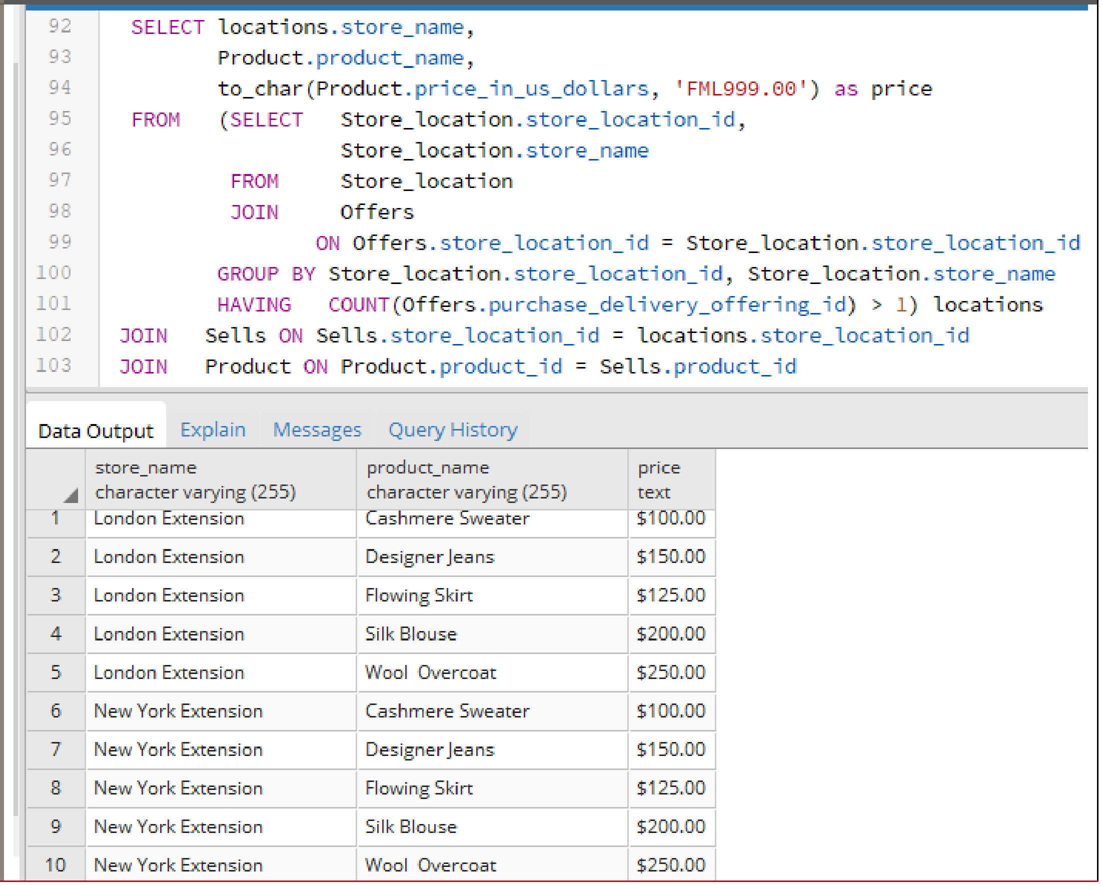
```sql
SELECT locations.store_name,
       Product.product_name,
       format(Product.price_in_us_dollars, '$.00') as price
FROM   (SELECT    Store_location.store_location_id,
                  Store_location.store_name
        FROM      Store_location
        JOIN      Offers
                  ON Offers.store_location_id = Store_location.store_location_id
        GROUP BY Store_location.store_location_id, Store_location.store_name
        HAVING    COUNT(Offers.purchase_delivery_offering_id) > 1) locations
JOIN   Sells ON Sells.store_location_id = locations.store_location_id
JOIN   Product ON Product.product_id = Sells.product_id
```

store_name	product_name	price
London Extension	Cashmere Sweater	$100.00
London Extension	Designer Jeans	$150.00
London Extension	Flowing Skirt	$125.00
London Extension	Silk Blouse	$200.00
London Extension	Wool Overcoat	$250.00
New York Extension	Cashmere Sweater	$100.00
New York Extension	Designer Jeans	$150.00
New York Extension	Flowing Skirt	$125.00
New York Extension	Silk Blouse	$200.00
New York Extension	Wool Overcoat	$250.00

The screenshot below shows execution of the query in PostgreSQL.

Screenshots: Subquery in FROM Clause in Postgres

Postgres



```
 SELECT locations.store_name,
        Product.product_name,
        to_char(Product.price_in_us_dollars, 'FML999.00') as price
 FROM   (SELECT   Store_location.store_location_id,
                  Store_location.store_name
         FROM     Store_location
         JOIN     Offers
               ON Offers.store_location_id = Store_location.store_location_id
        GROUP BY Store_location.store_location_id, Store_location.store_name
        HAVING   COUNT(Offers.purchase_delivery_offering_id) > 1) locations
JOIN    Sells ON Sells.store_location_id = locations.store_location_id
JOIN    Product ON Product.product_id = Sells.product_id
```

	store_name character varying (255)	product_name character varying (255)	price text
1	London Extension	Cashmere Sweater	$100.00
2	London Extension	Designer Jeans	$150.00
3	London Extension	Flowing Skirt	$125.00
4	London Extension	Silk Blouse	$200.00
5	London Extension	Wool Overcoat	$250.00
6	New York Extension	Cashmere Sweater	$100.00
7	New York Extension	Designer Jeans	$150.00
8	New York Extension	Flowing Skirt	$125.00
9	New York Extension	Silk Blouse	$200.00
10	New York Extension	Wool Overcoat	$250.00

Notice that the results are the same as in the solution that places the subquery in the WHERE clause.

The screenshots illustrate that we can successfully filter rows by using a subquery in the FROM clause, though subqueries in the FROM clause are useful for more than just filtering. The columns retrieved by the subquery can actually be returned in the outer query directly! Notice in the new solution, the Store_location table is no longer directly used in the FROM clause; the results from the subquery actually *take the place of* using the Store_location table. On line 1, the store_name column is used directly from the subquery through use of the "locations" alias, and there is no need to additionally join into the Store_location table in order to retrieve the name of the store. A subquery placed in the FROM clause sometimes take the place of a table.

The fact that two solutions address the same use case, one in step 5 and one in this step, demonstrates that *the same results can be retrieved from different queries*. Given that many queries address a particular use case correctly, is one better, and if so, which one? There is no universal answer. Typically, we choose the query that performs the best by selecting the one that outperforms the others. If several queries perform well, we typically choose the one that is the least complex. Sometimes two or more queries work equally as well, and we just need to select one of them. With enough experience, we discern one of the better strategies before we write the query, and only change strategies if the query we write has a problem we did not foresee. For Jill's use case, given the small data set in our schema, either solution – the one in step 5 with a subquery in the WHERE clause, and the one in this step with a subquery in the FROM clause – works fine. One solution could outperform the other if we add millions of products into the schema, but even that depends upon the particular DBMS used and the particular execution plan the DBMS selects. Some DBMS would discern that the two queries are functionally equivalent, and choose the same execution plan for both of them. Which solution is better for a particular use case depends upon many factors.

You can tackle this step in a fashion similar to what has been demonstrated here.

STEP 7

For this problem you will write a single query to address the same use case as in step 5, but change your query to use a correlated query combined with an EXISTS clause. The results should be the same as in step 5, except of course possibly row ordering which can vary. Explain:

a. how your solution makes use of the correlated subquery and EXISTS clause to help retrieve the result
b. how and when the correlated subquery is executed in the context of the outer query.

To help demonstrate how to write a correlated subquery with an EXISTS clause, we'll tackle another complex use case example.

> Adina travels regularly, has already decided she wants to purchase a Cashmere sweater from one of the store locations, and is considering purchasing other products as well. For each location that sells Cashmere sweaters, she wants to see all products and their prices in U.S. Dollars. Then she can make an informed decision of where the purchase the sweater in addition to any other products she may want.

Some use cases have a distinct part that requires the existence of a particular item. Just as in prior steps, we can identify the parts, write queries for them, and then put them together. One part comes from this sentence fragment in the use case, "she wants to see all products and their prices in U.S. Dollars". This is straightforward and we can steal a query from step 5 to address this part, shown below.

Code: Products and Prices Query

```
SELECT Store_location.store_name,
       Product.product_name,
      Product.price_in_us_dollars
FROM   Store_location
JOIN   Sells ON Sells.store_location_id = Store_location.store_location_id
JOIN   Product ON Product.product_id = Sells.product_id
```

The other part comes from this sentence fragment, "she wants to purchase a Cashmere sweater from one of the store locations"; this use case has a distinct part that requires a location to sell Cashmere sweaters in order to be considered. The SQL for this part is similar to the SQL for the prior part, differing by an extra condition that accepts only the "Cashmere Sweater" product, and retrieving only columns related to the Store_location table.

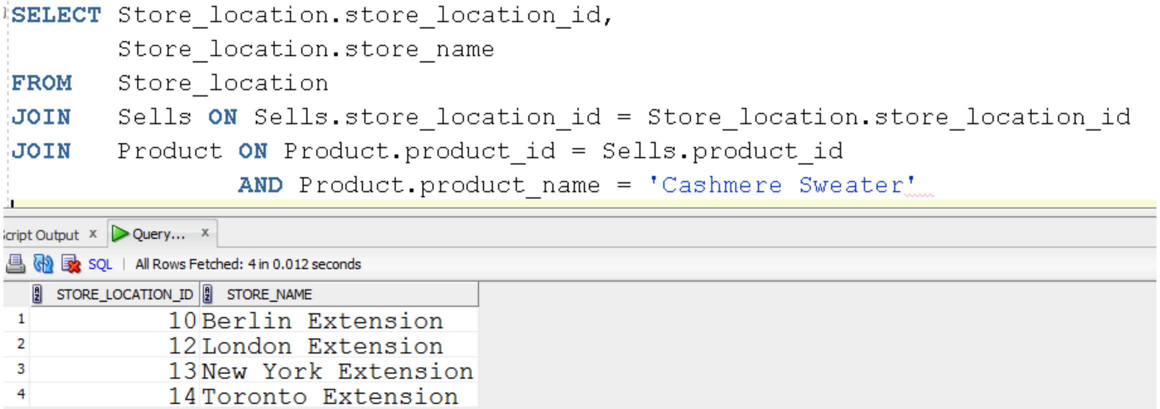
Code: Locations Selling Cashmere Sweaters

```
SELECT Store_location.store_location_id,
       Store_location.store_name
FROM   Store_location
JOIN   Sells ON Sells.store_location_id = Store_location.store_location_id
JOIN   Product ON Product.product_id = Sells.product_id
               AND Product.product_name = 'Cashmere Sweater'
```

Only store locations that sell Cashmere sweaters are retrieved by this query, as illustrated in the screenshots below.

Screenshots: Single Query for Cashmere Sweater

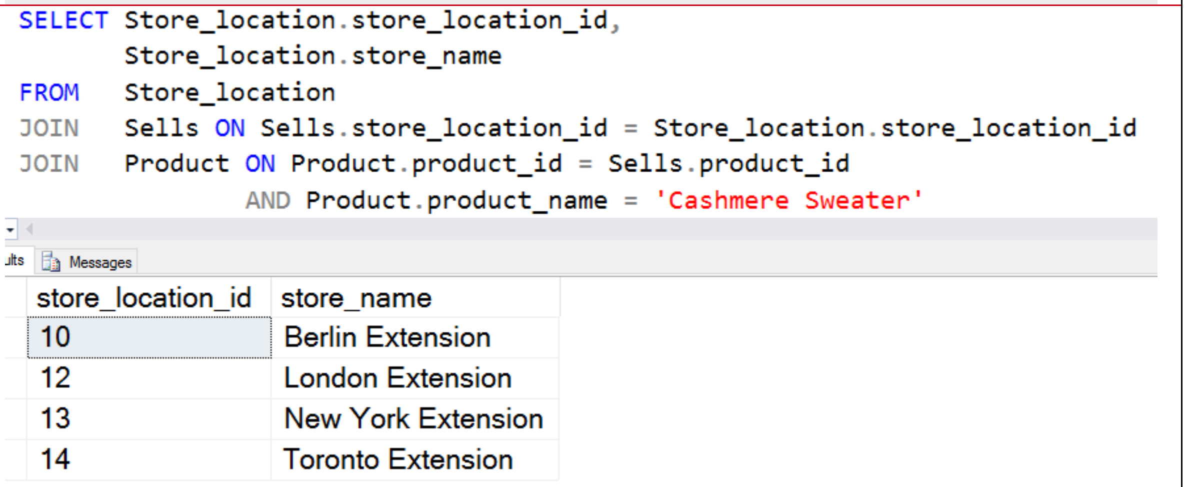
Oracle



```
SELECT Store_location.store_location_id,
       Store_location.store_name
FROM   Store_location
JOIN   Sells ON Sells.store_location_id = Store_location.store_location_id
JOIN   Product ON Product.product_id = Sells.product_id
               AND Product.product_name = 'Cashmere Sweater'
```

```
All Rows Fetched: 4 in 0.012 seconds

  STORE_LOCATION_ID  STORE_NAME
1                10  Berlin Extension
2                12  London Extension
3                13  New York Extension
4                14  Toronto Extension
```

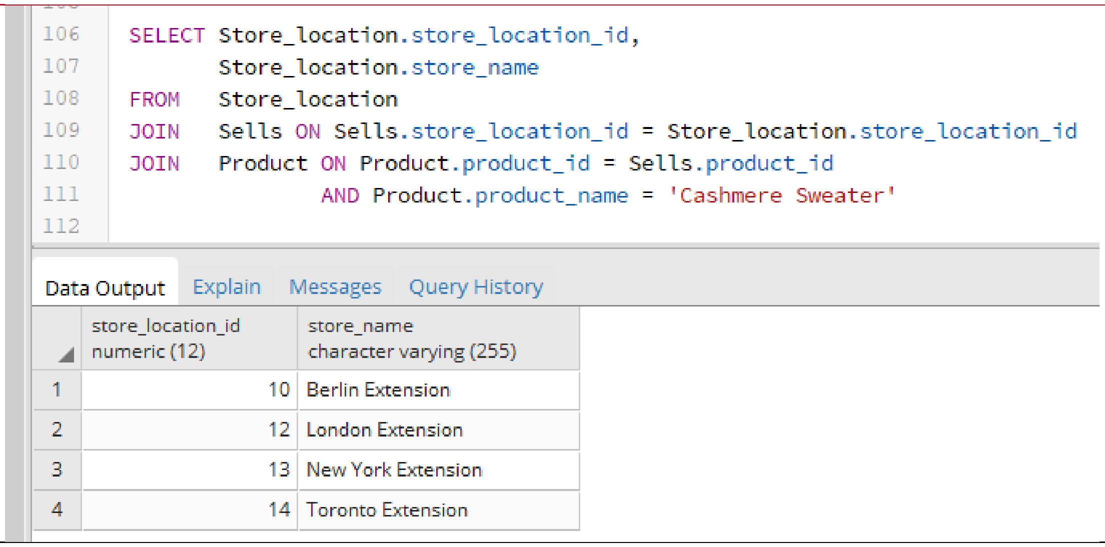
Microsoft SQL Server



```
SELECT Store_location.store_location_id,
       Store_location.store_name
FROM   Store_location
JOIN   Sells ON Sells.store_location_id = Store_location.store_location_id
JOIN   Product ON Product.product_id = Sells.product_id
               AND Product.product_name = 'Cashmere Sweater'
```

```
store_location_id  store_name
10                 Berlin Extension
12                 London Extension
13                 New York Extension
14                 Toronto Extension
```

PostgreSQL



```
SELECT Store_location.store_location_id,
       Store_location.store_name
FROM   Store_location
JOIN   Sells ON Sells.store_location_id = Store_location.store_location_id
JOIN   Product ON Product.product_id = Sells.product_id
               AND Product.product_name = 'Cashmere Sweater'
```

```
   store_location_id  store_name
   numeric (12)       character varying (255)
1                 10  Berlin Extension
2                 12  London Extension
3                 13  New York Extension
4                 14  Toronto Extension
```

Notice that in all screenshots the Cancun Extension is excluded because it does not sell Cashmere sweaters.

The initial strategy to solve this use case is similar to the initial strategies used to solve use cases in other steps. The SQL construct used when combining the queries for each part for this use case, however, is quite different than those used in other steps. The *EXISTS* clause is useful to address use cases such as this that test for the existence of a certain item. Unlike some SQL constructs that work with a single value, a list of values, or a table of values, the EXISTS clause only works with a subquery. An EXISTS clause is a Boolean expression that returns only true and false, true if the subquery returns any rows at all, and false if the subquery returns no rows. EXISTS does *not* consider the number of columns or the column's datatypes retrieved by the subquery, and even does not consider whether any values are NULL; rather, EXISTS only tests the existence of at least one row. So if we were to provide an English description of what EXISTS tests, it could be "Is any row retrieved from this subquery?" EXISTS is different than most other SQL constructs because it is designed specifically for subqueries.

At this point we would expect to simply combine the two independent queries with EXISTS, but this will not get us the results we need for Adina's use case. For illustrative purposes, let us try it so we see what happens with the query below.

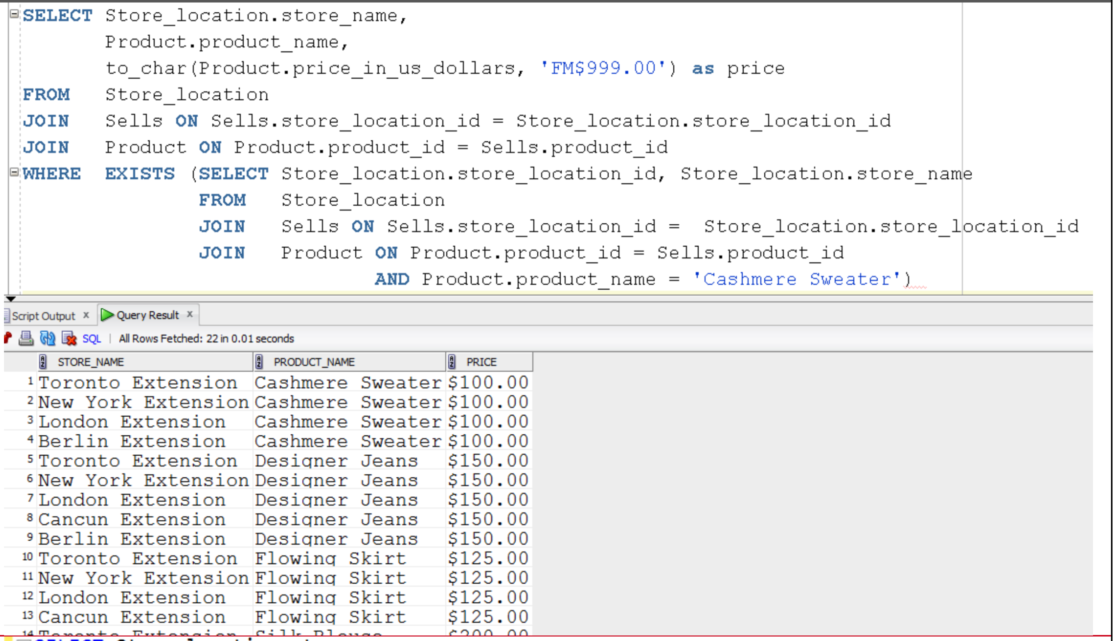
Code: Initial Combination Attempt

```
1:  SELECT Store_location.store_name,
2:         Product.product_name,
3:         Product.price_in_us_dollars
4:  FROM   Store_location
5:  JOIN   Sells ON Sells.store_location_id = Store_location.store_location_id
6:  JOIN   Product ON Product.product_id = Sells.product_id
7:  WHERE  EXISTS (SELECT Store_location.store_location_id, Store_location.store_name
8:                 FROM   Store_location
9:                 JOIN   Sells ON Sells.store_location_id =
Store_location.store_location_id
10:                JOIN   Product ON Product.product_id = Sells.product_id
11:                               AND Product.product_name = 'Cashmere Sweater')
```

You will notice that the first query is found on lines 1-6, and the second is found on lines 7-11 embedded inside of the EXISTS clause. The screenshots below shows part of the items returned (there are too many rows to show them all).

Screenshots: Initial Combination Attempt

Oracle

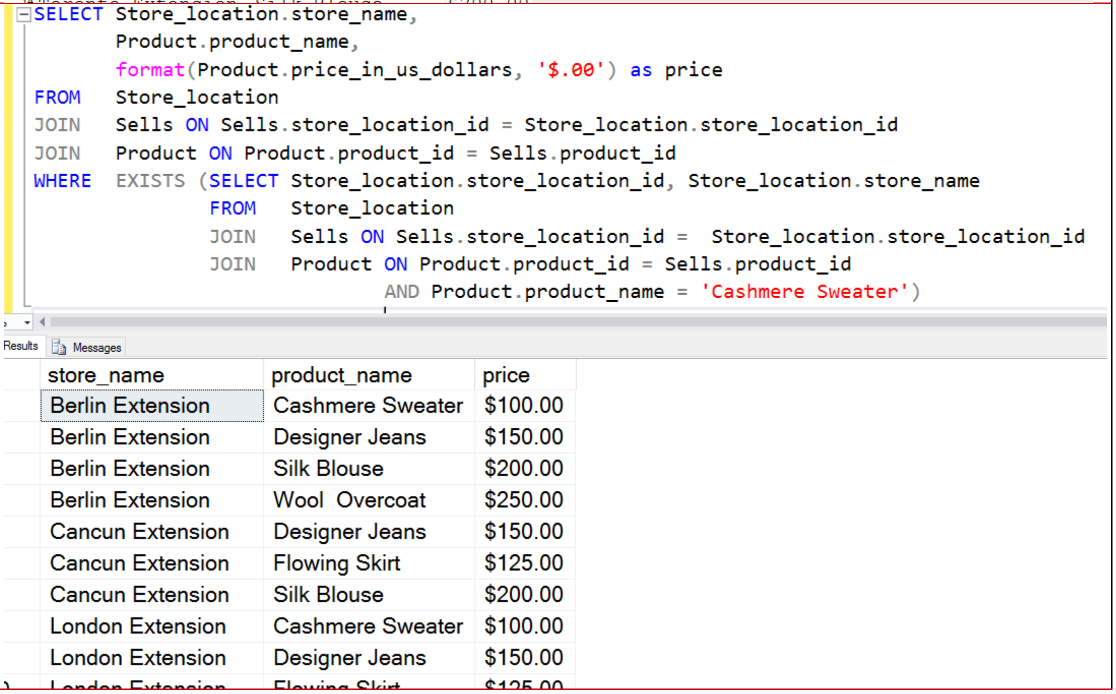


```
SELECT Store_location.store_name,
       Product.product_name,
       to_char(Product.price_in_us_dollars, 'FM$999.00') as price
FROM   Store_location
JOIN   Sells ON Sells.store_location_id = Store_location.store_location_id
JOIN   Product ON Product.product_id = Sells.product_id
WHERE  EXISTS (SELECT Store_location.store_location_id, Store_location.store_name
               FROM   Store_location
               JOIN   Sells ON Sells.store_location_id =  Store_location.store_location_id
               JOIN   Product ON Product.product_id = Sells.product_id
                      AND Product.product_name = 'Cashmere Sweater')
```

All Rows Fetched: 22 in 0.01 seconds

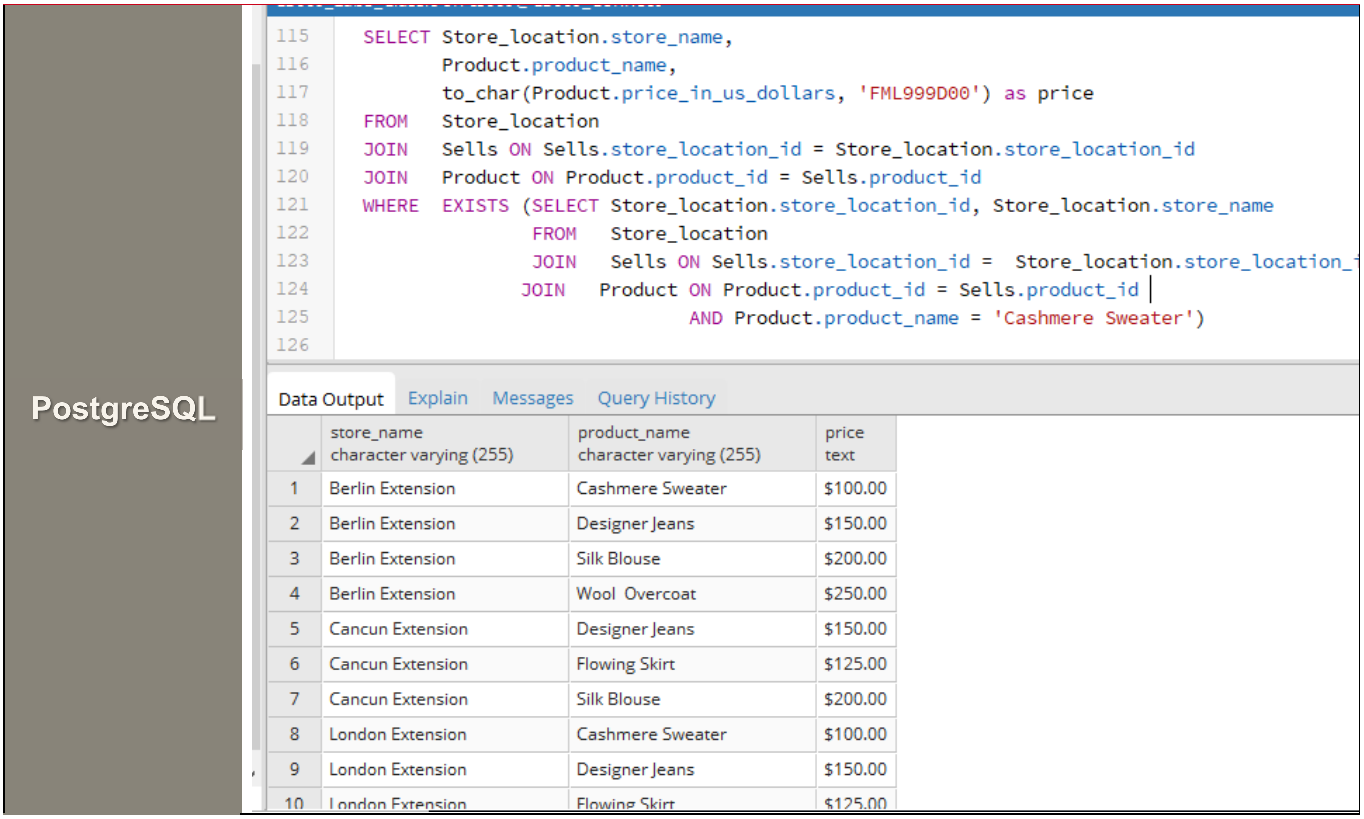
```
   STORE_NAME          PRODUCT_NAME      PRICE
 1 Toronto Extension   Cashmere Sweater  $100.00
 2 New York Extension  Cashmere Sweater  $100.00
 3 London Extension    Cashmere Sweater  $100.00
 4 Berlin Extension    Cashmere Sweater  $100.00
 5 Toronto Extension   Designer Jeans    $150.00
 6 New York Extension  Designer Jeans    $150.00
 7 London Extension    Designer Jeans    $150.00
 8 Cancun Extension    Designer Jeans    $150.00
 9 Berlin Extension    Designer Jeans    $150.00
10 Toronto Extension   Flowing Skirt     $125.00
11 New York Extension  Flowing Skirt     $125.00
12 London Extension    Flowing Skirt     $125.00
13 Cancun Extension    Flowing Skirt     $125.00
14 Toronto Extension   Silk Blouse       $200.00
```

Microsoft SQL Server



```
SELECT Store_location.store_name,
       Product.product_name,
       format(Product.price_in_us_dollars, '$.00') as price
FROM   Store_location
JOIN   Sells ON Sells.store_location_id = Store_location.store_location_id
JOIN   Product ON Product.product_id = Sells.product_id
WHERE  EXISTS (SELECT Store_location.store_location_id, Store_location.store_name
               FROM   Store_location
               JOIN   Sells ON Sells.store_location_id =  Store_location.store_location_id
               JOIN   Product ON Product.product_id = Sells.product_id
                      AND Product.product_name = 'Cashmere Sweater')
```

```
store_name          product_name      price
Berlin Extension    Cashmere Sweater  $100.00
Berlin Extension    Designer Jeans    $150.00
Berlin Extension    Silk Blouse       $200.00
Berlin Extension    Wool  Overcoat    $250.00
Cancun Extension    Designer Jeans    $150.00
Cancun Extension    Flowing Skirt     $125.00
Cancun Extension    Silk Blouse       $200.00
London Extension    Cashmere Sweater  $100.00
London Extension    Designer Jeans    $150.00
London Extension    Flowing Skirt     $125.00
```



```
SELECT Store_location.store_name,
       Product.product_name,
       to_char(Product.price_in_us_dollars, 'FML999D00') as price
FROM   Store_location
JOIN   Sells ON Sells.store_location_id = Store_location.store_location_id
JOIN   Product ON Product.product_id = Sells.product_id
WHERE  EXISTS (SELECT Store_location.store_location_id, Store_location.store_name
               FROM   Store_location
               JOIN   Sells ON Sells.store_location_id =  Store_location.store_location_
              JOIN   Product ON Product.product_id = Sells.product_id
                      AND Product.product_name = 'Cashmere Sweater')
```

	store_name character varying (255)	product_name character varying (255)	price text
1	Berlin Extension	Cashmere Sweater	$100.00
2	Berlin Extension	Designer Jeans	$150.00
3	Berlin Extension	Silk Blouse	$200.00
4	Berlin Extension	Wool Overcoat	$250.00
5	Cancun Extension	Designer Jeans	$150.00
6	Cancun Extension	Flowing Skirt	$125.00
7	Cancun Extension	Silk Blouse	$200.00
8	London Extension	Cashmere Sweater	$100.00
9	London Extension	Designer Jeans	$150.00
10	London Extension	Flowing Skirt	$125.00

You will immediately notice the problem with the results in either screenshot. The products for Cancun Extension are included in the results, but Adina only wants products for locations that sell Cashmere sweaters! We can see what is wrong with the query's logic by summarizing in English what the query does, "Retrieve all products and their prices for all locations if *any* location sells Cashmere sweaters." Therein lies the problem, "if *any* location sells Cashmere sweaters". EXISTS only checks for the existence of any row, so if *any* location sells a Cashmere sweater, EXISTS indicates a true value, and the products for all locations are retrieved. Simple combining does not work in this case.

EXISTS usually demands a more complex method of combining the queries for each part. The subquery usually must be *correlated* with the outer query to get the results we want. A correlated subquery references at least one table from the outer query, which means that conceptually, the subquery is not an independent query. Unlike subqueries in prior steps, which are termed *uncorrelated* subqueries, we cannot execute correlated subqueries on their own; correlated subqueries only make sense in the context of the outer query into which they are embedded. And unlike uncorrelated subqueries which are executed once and retrieve results once, correlated subqueries are executed *once for each row* in the outer query and therefore retrieve *one result set for each row* in the outer query. We can say that during the execution of an overall query, the results of an uncorrelated subquery are fixed, and the results of a correlated subquery are relative to each row in the outer query. EXISTS is typically coupled with a correlated subquery to achieve meaningful results.

Altering our subquery for Adina's use case by correlating the subquery gives us the logic we want.

Code: Correlated Subquery

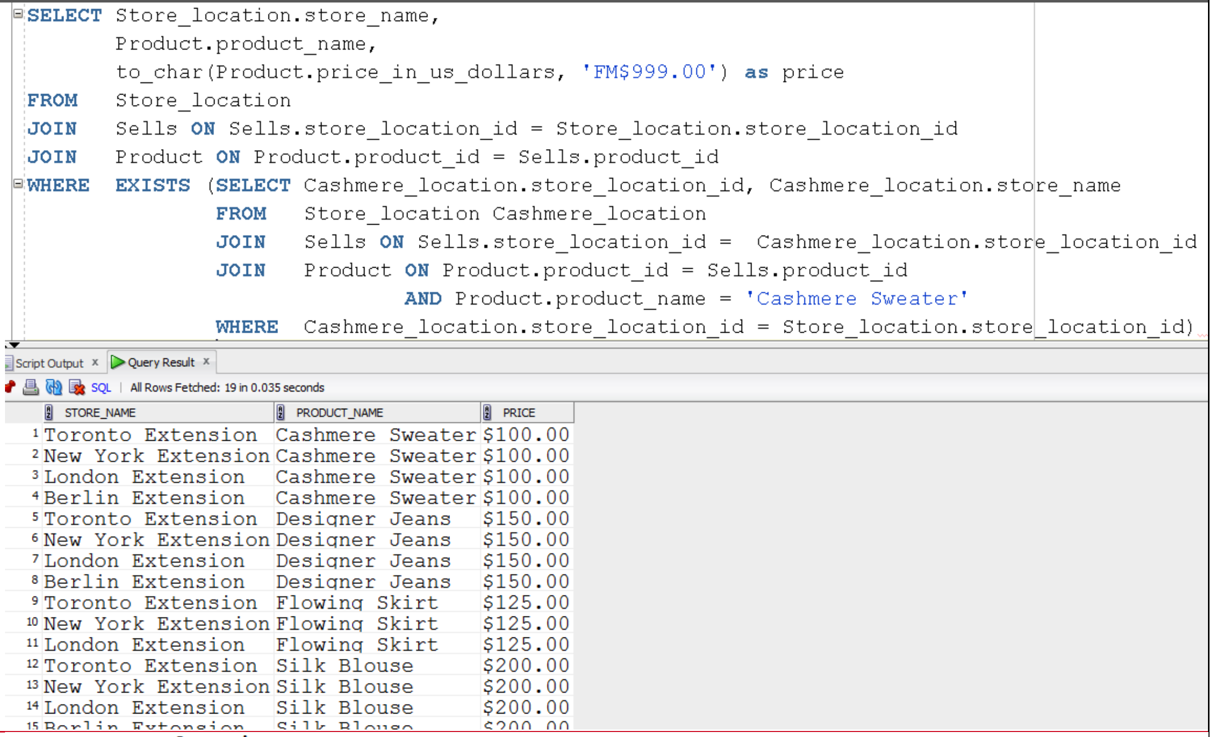
```
1:  SELECT Store_location.store_name,
2:         Product.product_name,
3:         Product.price_in_us_dollars
4:  FROM   Store_location
5:  JOIN   Sells ON Sells.store_location_id = Store_location.store_location_id
6:  JOIN   Product ON Product.product_id = Sells.product_id
7:  WHERE  EXISTS (SELECT Cashmere_location.store_location_id, Cashmere_location.store_name
8:                 FROM   Store_location Cashmere_location
9:                 JOIN   Sells ON Sells.store_location_id =  Cashmere_location.store_location_id
10:                JOIN   Product ON Product.product_id = Sells.product_id
11:                               AND Product.product_name = 'Cashmere Sweater'
12:                WHERE  Cashmere_location.store_location_id = Store_location.store_location_id)
```

Lines 1-6 are the same as in the first solution, but lines 7-12 are different. The first difference you may notice is that there is the alias **`Cashmere_location`** for the Store_location table introduced the subquery. This is necessary to eliminate any ambiguity between the Store_location table introduced in the outer query on line 4, and the Store_location introduced in the subquery on line 8. With the alias, it is clear that a reference to **`Store_location`** is a reference to the table introduced in the outer query, and a reference to **`Cashmere_location`** is a reference to the table introduced in the subquery. We use the identifier **`Cashmere_location`** to highlight the fact that locations in the subquery are only those that sell Cashmere sweaters. The second difference is found on line 12, **`WHERE Cashmere_location.store_location_id = Store_location.store_location_id`**. It is this line that correlates the subquery with the outer query! Notice that the ID of **`Cashmere_location`** must equal the ID of **`Store_location`**, and it is this equality that forces the subquery into correlation. In English, we could summarize the logic of the subquery as follows: “Retrieve the store location found in the current row of the outer query only if that store location sells Cashmere sweaters”. This logic, coupled with the EXISTS keyword, means that if the current row in the outer query does not contain a store location that sells Cashmere sweaters, it is excluded from the result set. This is exactly what we want!

Below are the screenshots, which show enough rows to see that our query works, but excludes some rows for brevity.

Screenshots: Correlated Subquery

Oracle

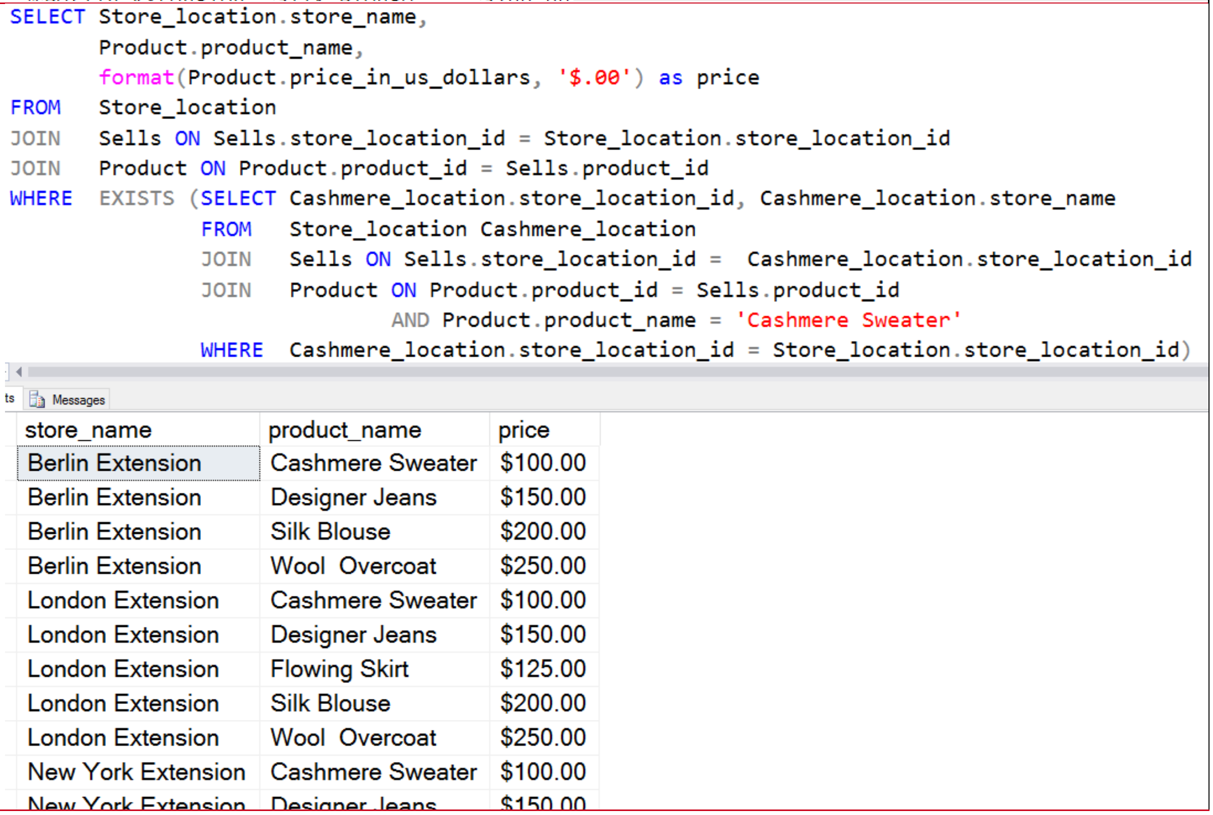


```
SELECT Store_location.store_name,
       Product.product_name,
       to_char(Product.price_in_us_dollars, 'FM$999.00') as price
FROM   Store_location
JOIN   Sells ON Sells.store_location_id = Store_location.store_location_id
JOIN   Product ON Product.product_id = Sells.product_id
WHERE  EXISTS (SELECT Cashmere_location.store_location_id, Cashmere_location.store_name
               FROM   Store_location Cashmere_location
               JOIN   Sells ON Sells.store_location_id =  Cashmere_location.store_location_id
               JOIN   Product ON Product.product_id = Sells.product_id
                           AND Product.product_name = 'Cashmere Sweater'
               WHERE  Cashmere_location.store_location_id = Store_location.store_location_id)
```

All Rows Fetched: 19 in 0.035 seconds

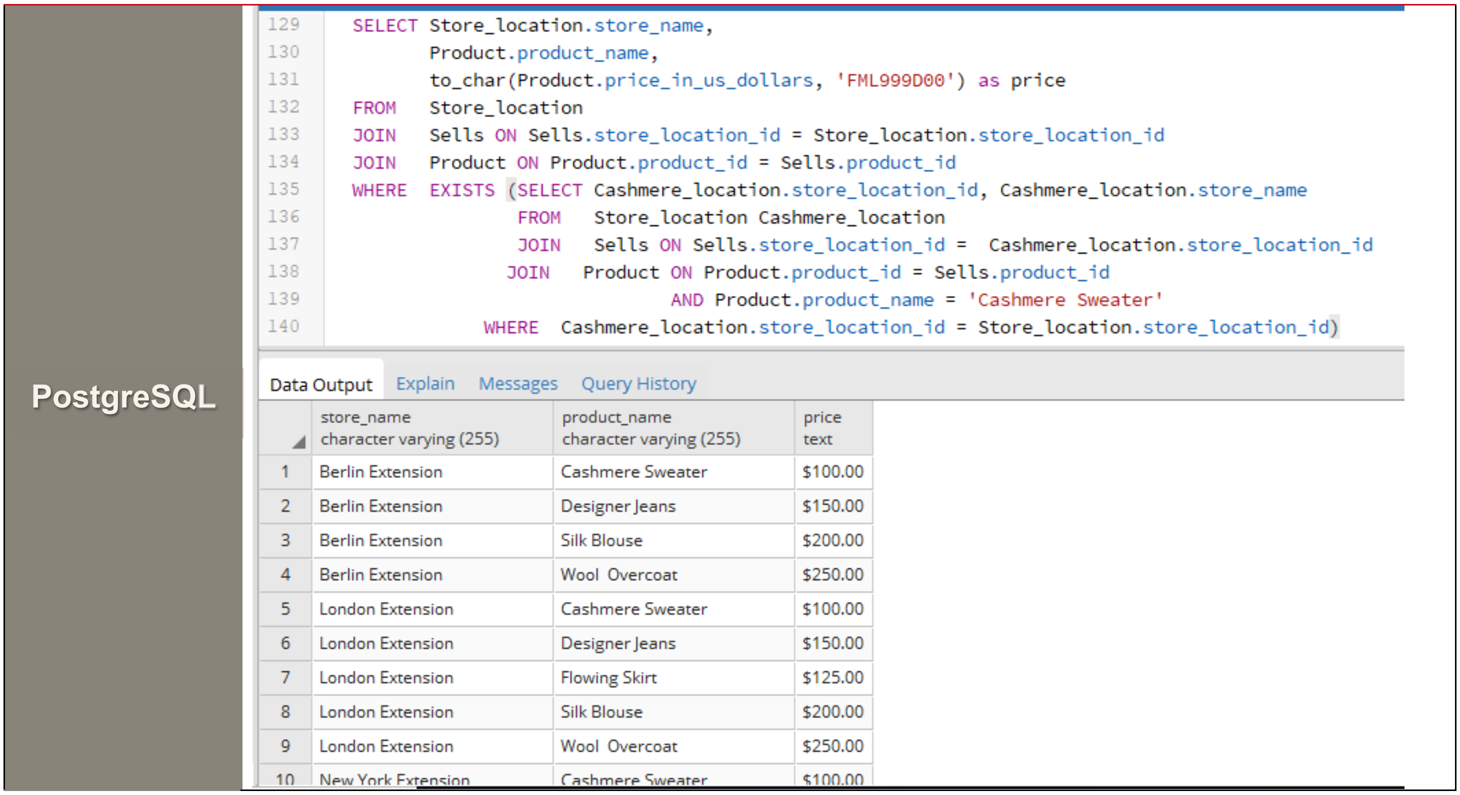
```
   STORE_NAME        PRODUCT_NAME      PRICE
 1 Toronto Extension Cashmere Sweater $100.00
 2 New York Extension Cashmere Sweater $100.00
 3 London Extension  Cashmere Sweater $100.00
 4 Berlin Extension  Cashmere Sweater $100.00
 5 Toronto Extension Designer Jeans   $150.00
 6 New York Extension Designer Jeans  $150.00
 7 London Extension  Designer Jeans   $150.00
 8 Berlin Extension  Designer Jeans   $150.00
 9 Toronto Extension Flowing Skirt    $125.00
10 New York Extension Flowing Skirt   $125.00
11 London Extension  Flowing Skirt    $125.00
12 Toronto Extension Silk Blouse      $200.00
13 New York Extension Silk Blouse     $200.00
14 London Extension  Silk Blouse      $200.00
15 Berlin Extension  Silk Blouse      $200.00
```

Microsoft SQL Server



```
SELECT Store_location.store_name,
       Product.product_name,
       format(Product.price_in_us_dollars, '$.00') as price
FROM   Store_location
JOIN   Sells ON Sells.store_location_id = Store_location.store_location_id
JOIN   Product ON Product.product_id = Sells.product_id
WHERE  EXISTS (SELECT Cashmere_location.store_location_id, Cashmere_location.store_name
               FROM   Store_location Cashmere_location
               JOIN   Sells ON Sells.store_location_id =  Cashmere_location.store_location_id
               JOIN   Product ON Product.product_id = Sells.product_id
                           AND Product.product_name = 'Cashmere Sweater'
               WHERE  Cashmere_location.store_location_id = Store_location.store_location_id)
```

```
store_name          product_name       price
Berlin Extension    Cashmere Sweater   $100.00
Berlin Extension    Designer Jeans     $150.00
Berlin Extension    Silk Blouse        $200.00
Berlin Extension    Wool  Overcoat     $250.00
London Extension    Cashmere Sweater   $100.00
London Extension    Designer Jeans     $150.00
London Extension    Flowing Skirt      $125.00
London Extension    Silk Blouse        $200.00
London Extension    Wool  Overcoat     $250.00
New York Extension  Cashmere Sweater   $100.00
New York Extension  Designer Jeans     $150.00
```



```
SELECT Store_location.store_name,
       Product.product_name,
       to_char(Product.price_in_us_dollars, 'FML999D00') as price
FROM   Store_location
JOIN   Sells ON Sells.store_location_id = Store_location.store_location_id
JOIN   Product ON Product.product_id = Sells.product_id
WHERE  EXISTS (SELECT Cashmere_location.store_location_id, Cashmere_location.store_name
               FROM   Store_location Cashmere_location
               JOIN   Sells ON Sells.store_location_id =  Cashmere_location.store_location_id
              JOIN   Product ON Product.product_id = Sells.product_id
                             AND Product.product_name = 'Cashmere Sweater'
            WHERE  Cashmere_location.store_location_id = Store_location.store_location_id)
```

	store_name character varying (255)	product_name character varying (255)	price text
1	Berlin Extension	Cashmere Sweater	$100.00
2	Berlin Extension	Designer Jeans	$150.00
3	Berlin Extension	Silk Blouse	$200.00
4	Berlin Extension	Wool Overcoat	$250.00
5	London Extension	Cashmere Sweater	$100.00
6	London Extension	Designer Jeans	$150.00
7	London Extension	Flowing Skirt	$125.00
8	London Extension	Silk Blouse	$200.00
9	London Extension	Wool Overcoat	$250.00
10	New York Extension	Cashmere Sweater	$100.00

Notice that Cancun Extension, which does not sell Cashmere sweaters, has been excluded from the result set. This is exactly what Adina wants! She now has a list of all products and their prices for store locations that sell Cashmere sweaters.

Since the topic of correlated subqueries is complex, let us summarize the steps we go through to solve Adina's use case, and any use case requiring a correlated subquery. First, we identify the distinct parts of the use case that require different SQL queries and constructs. Second, we write independent queries that address each part. Third, we combine the independent queries in such a way that the subquery is correlated to the outer query. Correlating the subquery involves changing it from an independent (uncorrelated) subquery to one that references at least one table introduced in the outer query, thus ensuring the subquery retrieves results based upon the current row of the outer query. The steps needed to solve all use cases requiring subqueries are similar, but those that require correlated subqueries necessitate a different method of combining the queries that make up the parts. To be sure, once you become very experienced with correlated subqueries, you will be comfortable writing the entire query, including the subquery, all at once without the need to first identify the distinct parts.

It should be noted that use of EXISTS combined with a correlated subquery is not limited to addressing use cases that test for the existence of a single item. This EXISTS combination can be used to test for the existence of mostly any set of conditions. In fact, many use cases that make use of uncorrelated subqueries in the WHERE clause can be rewritten to use a correlated subquery with EXISTS. EXISTS may perform better in some situations, and an uncorrelated subquery may perform better in others. Knowing how to use EXISTS gives us another tool we can use to help address more complex use cases.

Lastly, correlated subqueries can be used with other constructs, such as the IN construct. However, it is difficult to think of use cases where using correlated subqueries with other constructs makes for the best solution. Correlated subqueries commonly are coupled with EXISTS.

You can now use a similar strategy to address this step.

STEP 8

Compare and contrast the construction of the three different queries you developed in steps 5-7, which all address the same use case. What advantages and disadvantages do each construction have over the others? Which do you prefer and why?

To best answer this step, you may wish to review the construction of your SQL for steps 5-7, as well as the explanations of the concepts for each step. Try to look at your different solutions from an overall perspective, to better understand the advantages and disadvantages of each.