MET CS 669 Database Design and Implementation for Business
SQL Lab 5 Instructions: Subqueries, Expressions, and Value Manipulation

Objective

This lab teaches you the power of subqueries, expressions, and value manipulation, and the mechanics crafting SQL queries that harness the power of these three to handle more complex use cases.

Prerequisites

Before attempting this lab, it is best to read the textbook and lecture material covering the objectives listed above. While this lab shows you how to create and use these constructs in SQL, the lab does not explain in full the theory behind the constructs, as does the lecture and textbook.

Required Software

The examples in this lab will execute in modern versions of Oracle and Microsoft SQL Server as is. If you are using a different RDBMS, you may need to modify the SQL for successful execution.

Saving Your Data

If you choose to perform portions of the assignment in different sittings, it is important to *commit* your data at the end of each session. This way, you will be sure to make permanent any data changes you have made in your curent session, so that you can resume working without issue in your next session. To do so, simply issue this command:

```
COMMIT;
```

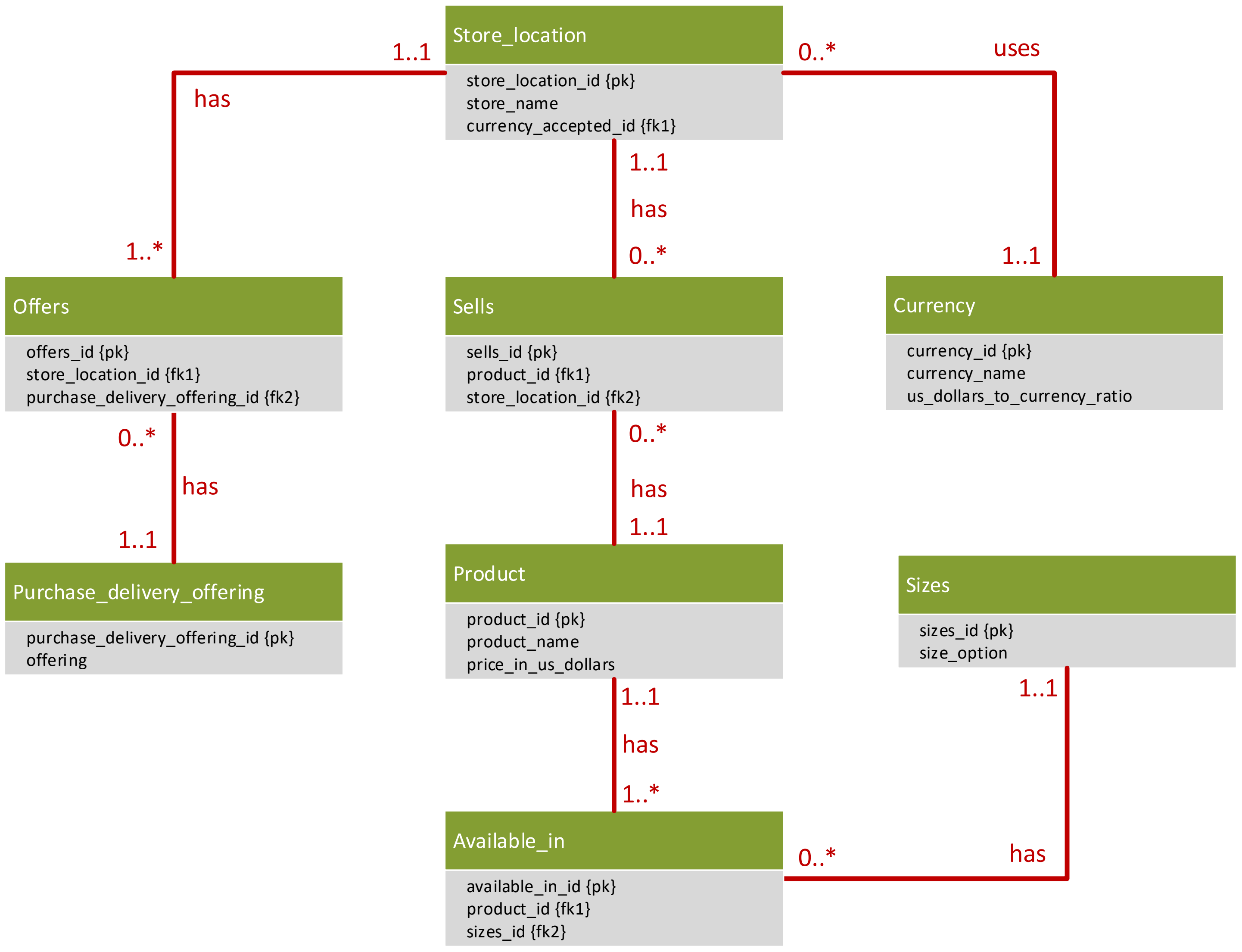
Data changes in one session will only be visible only in that session, unless they are committed, at which time the changes are made permanent in the database.

Lab Completion

Use the submission template provided in the assignment inbox to complete this lab.

Lab Overview

In this lab, we practice value manipulation and subqueries on the schema illustrated below.



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

DDL and DML to create and populate the tables in the 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);

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);
```

```sql
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)
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);
```

```sql
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)
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);
```

Section One – Expressions and Value Manipulation

Overview

While it is certainly useful to directly extract values as they are stored in a database, it is more useful in some contexts to manipulate these values to derive a different result. In this section we practice using value manipulation techniques to transform data values in useful ways. For example, what if we want to tell a customer exactly how much money they need to give for a purchase? We could extract a price and sales tax from the database, but it would be more useful to compute a price with tax as a single value by multiplying the two together and rounding appropriately, and formatting it as a currency, as illustrated in the figure below.

Less Useful to Customer	
price	**tax_percent**
7.99	8.5

More Useful to Customer
price_with_tax
$8.67

We do not need to store the price with tax, because we can derive it when we need it. As another example, what if we need to send an email communication to a customer by name? We could extract the prefix, first name, and last name of the customer, but it would be more useful to properly format the name by combining them in proper order, as illustrated below.

Less Useful to Customer		
prefix	**first_name**	**last_name**
Mr.	Seth	Nemes

More Useful to Customer
name
Mr. Seth Nemes

Again, we do not need to store the formatted name, because we can derive it when we need it from its constituent parts. Manipulating raw data values stored in database tables can yield a variety of useful results we need without adding the burden of storing every such result.

Steps

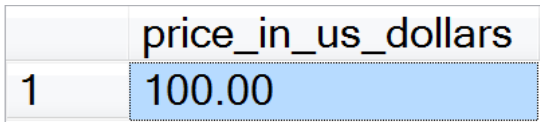
1. Execute the DDL and DML listed in the lab overview section in order to create and populate the currency schema in your database. There is no need to provide screenshots of executing this DDL and DML.

2. If we were asked to give the price of the Cashmere sweater in Euros, how would we do it using SQL? In looking in the Product table, we see that we easily pull out the

price of the sweater in U.S. Dollars with a simple query:

```sql
SELECT price_in_us_dollars
FROM   Product
WHERE  product_name = 'Cashmere Sweater'
```

The results of this query tell us that the price is $100:

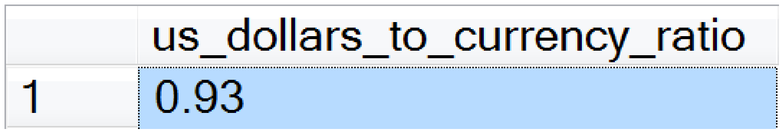


	price_in_us_dollars
1	100.00

This is a good first step, but still does not give us the price in Euros, only in U.S. Dollars. Taking a look at the Currency table, we see that there is a column named **`us_dollars_to_currency_ratio`** that has the conversion factor needed convert from U.S. Dollars into the target currency. We can again write a simply query to list out the conversion factor for Euros:

```sql
SELECT us_dollars_to_currency_ratio
FROM   Currency
WHERE  currency_name = 'Euro'
```

This tells us that the conversion factor is 0.93:

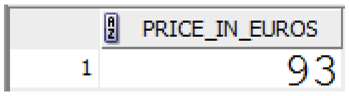


	us_dollars_to_currency_ratio
1	0.93

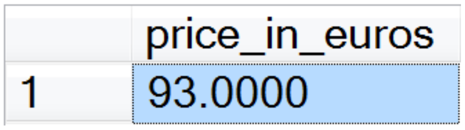
We then manually hardcode 0.93 into our first query in order to obtain the price in Euros as follows:

```sql
SELECT price_in_us_dollars * 0.93 AS price_in_euros
FROM   Product
WHERE  product_name = 'Cashmere Sweater'
```

In the query, we have taken the price in U.S. Dollars and multiplied it times the conversion factor of 0.93, then aliased the result as price_in_euros, which gives us the result of €93, as shown in the side-by-side screenshots of Oracle then SQL Server below.



```
  PRICE_IN_EUROS
1             93
```



```
   price_in_euros
1  93.0000
```

You could have quickly answered the question of the price of a Cashmere sweater in Euros by eyeballing the values in the tables and doing some basic math, so why did we use SQL? Perhaps the most significant reason is that when we are developing a database and surrounding I.T. system, *we are not actually asking for a single answer, but are asking for logic that is capable of answering that same question repeatedly for the entire life of the database,* whether it is years or decades. Answering the question just once is not useful given our goal. We want to know the price of the Cashmere sweater today, tomorrow, and next year, even if the prices or exchange rates change. A second reason is that *our goal is to give an I.T. system the ability to answer this question, not a human being.* I.T. systems and surrounding databases are all about automation, performing repeated tasks much more quickly than human beings, freeing us from the responsibility of doing tedious tasks ourselves. Obviously, I.T. systems are not capable of eyeballing, and need formal SQL logic in order to access relational databases. Lastly, *many real-world relational database schemas contain too many tables, relationships, and values for us to practically keep track them ourselves,* so even if we want to answer a question ourselves, we still need to use SQL to obtain the values we need from the database. We develop SQL queries to answer our data questions in today's world.

3. What if you are asked to give the price of a flowing skirt in Cancun? Just like in step 2, you need two queries. Your first query retrieves the currency ratio for the currency accepted in Cancun. Your second query hardcodes the currency ratio retrieved in the first query, in order to determine the price in Cancun. Note that unlike the use case in step 2, which asks for a currency by name, this use case asks for the store location by name in order to retrieve whatever currency is accepted by that location. This requires slightly different logic, so your first query will be similar to the first query in step 2, but will need to include the store location in order to retrieve the correct currency ratio.

   Capture a screenshots of both queries and the results of their execution.

4. Later we'll explore directly embedding the second query inside the first. After all, you may have already insightfully objected to hardcoding the value of 0.93 in step 2, noticing that the resulting query will yield incorrect results as soon as the ratio changes, and likewise for step 3. You may be eager to fix this problem, but let's first talk about money. How can we learn about subqueries when we are talking about values such as 93.0000 Euros as indicated by SQL Server? This does not look right! Each step is important in the learning process, and let us not rush this complex subject of subqueries.

SQL clients are sophisticated applications, but are not so sophisticated that they always display values in the format we expect. You may notice that the result from step 2 lists out no decimal points for Oracle and 4 decimal points for SQL Server, both without the Euro symbol (€). This is not what we are accustomed to seeing for monetary amounts in Euros. We would expect to see the monetary result as €93.00 in the United States or United Kingdom, or €93,00 in other European countries. Listing out 4 decimal points is nothing more than the default for the SQL Server Management Studio (SSMS) client when displaying numbers that are not whole numbers; SSMS does not understand that we are displaying a currency. The default for the Oracle SQL Developer client is to remove trailing 0 digits that occur to the right of the decimal point, hence the result of 93 with no decimal points. Other SQL clients may have different defaults for SQL Server and other databases. SQL clients oftentimes display a value from a basic SQL query in a nonconventional format.

The discrepancy between the value displayed and the value we conventionally expect shows us that there is something more involved. There are actually four significant components that determine how a value is displayed -- the raw value stored in a database table, manipulations on the value performed by the SQL query, formatting constructs applied in the SQL query, and how the particular SQL client displays the value. These components all collectively determine how a value will be displayed to us when we execute SQL in a SQL client. There is a tremendous amount of depth for each of these components, and while we will not be able to cover every detail, it is important that we explore each in more depth. Doing so will help give you the ability to craft queries that display values in whatever format you deem appropriate. Controlling how a value is displayed is an intricate subject.

Different kinds of data have different limits that present a challenge for database designers as they consider how to store the data. Some kinds of values have no theoretical limit, for example, fractions that result in infinitely repeating decimals. How do we store these infinitely long values? Some kinds of values have theoretical limits, but we cannot determine them. For example, how would we determine how much storage we need for the text of the lengthiest book in the world? Even if we determine the lengthiest book known, we could always discover a new, lengthier book, or someone could write a lengthier one in the future. Some kinds of values have known limits, but their limits are too big for practical storage. For example, a business may know all websites visited by its employees while at work in the prior year, but would it practical for the business to store the full content of every website every time it is visited? We need to think about these kinds of limits before we store the data in our database.

All values stored in a relational database column have size limits, and interestingly datatypes, which we learned in prior labs determine the set of legal values for database columns, also establish size limits. All exact numeric datatypes have a precision, which is the maximum number of digits allowed in the number, and a scale, which is the maximum number of digits allowed to the right of the decimal point. For example, if we want to store the number 12.34, we need a precision of at least 4 since there are 4 digits in total, and a scale of at least 2 since there are 2 digits to the right of the decimal point. All inexact numeric datatypes used for storing fractional numbers are constrained by a maximum number of bytes. All text datatypes have a maximum limit of either characters or bytes. Date and time datatypes have limits on the number of digits used to store fractions of a second. Data stored in a relational database is limited in size in various ways as specified in the datatype for each table column.

5. The legal form and size limitation of a value is one significant component that determines how a value will be displayed in the result of a SQL query, and you get a chance to explore it for yourself in this step. We now know it is the datatype that constrains to a set of legal values and a size limitation.

   Identify any one of the datatypes referred to in step 4, or even one of another of your own choosing, that has a size limitation that you can control. Then do the following:

   a. Explain the datatype in a few sentences – the set of legal values allowed by the datatype, the syntax of specifying the maximum size allowed for the column, and what the size limitation means for that particular datatype.

   b. Create a table with a name of your choosing that has just one column, and give the column the datatype you identified along with a size limitation of your choosing.

   c. Attempt to insert a value that exceeds the size limitation on the column and see what error message your DBMS gives you.

   Make sure to capture screenshots for parts b and c.

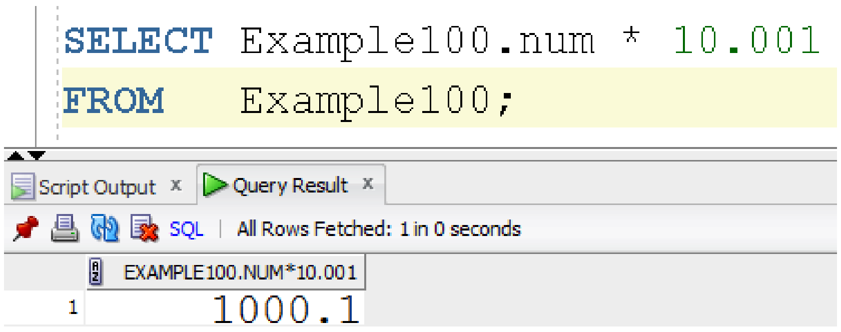
6. A SQL query always gives us a value in the same form as it is stored in the database, right? Wrong! Many SQL queries manipulate values, and the results have changes in the datatypes or size limits. For example, imagine we start with the number “100” in a column like so:

   ```
   CREATE TABLE Example100 (num DECIMAL(3));
   INSERT INTO Example100 (num) VALUES (100);
   ```

   Further imagine that we multiply the value times 10 and 1 thousandth like so:

   ```
   SELECT Example100.num * 10.001
   FROM   Example100;
   ```

The number "100" is stored as a DECIMAL(3) datatype, which means that it supports 3 digits total with no digits allowed to the right of the decimal point. So should we expect that the result is also of that same form? Let's find out! The screenshots below show the query executed in Oracle then SQL Server, respectively.

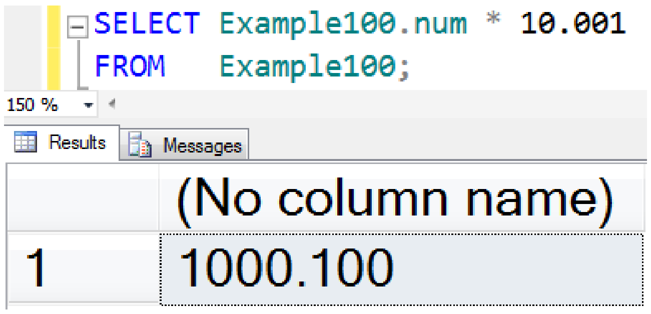


```
SELECT Example100.num * 10.001
FROM   Example100;
```

Script Output × | Query Result ×

SQL | All Rows Fetched: 1 in 0 seconds

```
   EXAMPLE100.NUM*10.001
1               1000.1
```



```
SELECT Example100.num * 10.001
FROM   Example100;
```

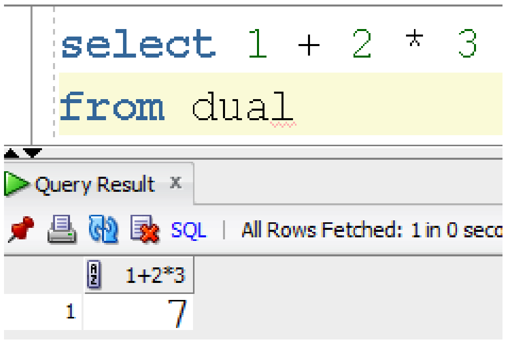
150 %

Results | Messages

```
   (No column name)
1  1000.100
```

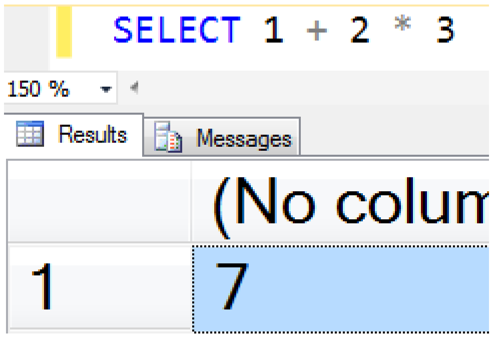
Whoa! The result in Oracle has 5 digits total, with four the left of the decimal point and one to the right, and the result in SQL Server has seven digits total, with four to the left of the decimal point and three to the right of the decimal point. This looks more like a DECIMAL(5,1) or DECIMAL(7,3) than a DECIMAL(4). As demonstrated, a SQL query can yield a result in a form different from that of the raw stored value.

Manipulations on a value in a SQL query affect how the results are displayed, and these manipulations are defined more technically as *expressions* in a SQL query. Expressions consist of *operands*, which are values from the database or hard-coded values, and *operators*, which are SQL keywords or symbols that derive results from one or more operands in a predefined way. For example, the simple expression "Example100.num * 10.001" we used earlier has two operands – Example100.num and 10.001 – and makes use of the "*" operator, which derives a new result by multiplying the values of two operands. The act of deriving the new result is termed an *operation,* so we say that operators perform operations on operands. Expressions give us the ability to transform raw database values in a variety of ways.
We expect the expression "Example100.num * 10.001" to multiply the values together, but we may not expect that the expression "1 + 2 * 3" gives us a result of 7, instead of a result of 9. Look at the side-by-side screenshots from Oracle and SQL Server, respectively, below.



```
select 1 + 2 * 3
from dual
```

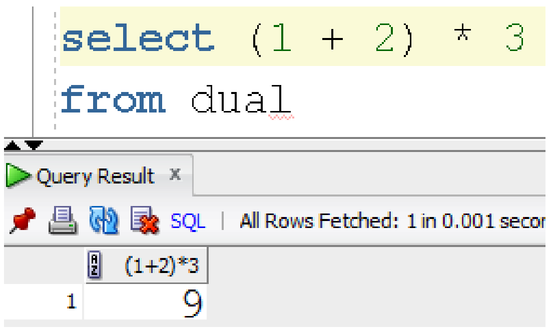
```
Query Result
All Rows Fetched: 1 in 0 seco
   1+2*3
1      7
```



```
SELECT 1 + 2 * 3
```

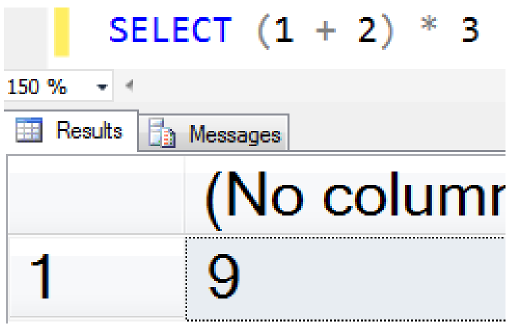
```
150 %
Results  Messages
   (No colum
1  7
```

By a casual glance, we would think that 1 + 2 = 3, then 3 * 3 = 9, so why is the result 7? It is important to note the order in which the operations occur is strictly defined by the DBMS' *operator precedence*, which is set of rules that indicates which operations will be evaluated before other operations. Both Oracle and SQL Server evaluate multiplication and division operators before addition and subtraction operators (Microsoft, 2016a; Oracle, 2015), so the expression "1 + 2 * 3" yields 7 in both DBMS, rather than 9, because the multiplication operation occurs before the addition operation. In this example, 2 * 3 = 6, then 6 + 1 = 7. Parentheses can be used in an expression to override operator precedence. If we use parentheses judiciously in the expression, changing it to (1 + 2) * 3, we can change the result to 9, as shown in the results below:



```
select (1 + 2) * 3
from dual
```

```
Query Result
All Rows Fetched: 1 in 0.001 secor
   (1+2)*3
1        9
```



```
SELECT (1 + 2) * 3
```

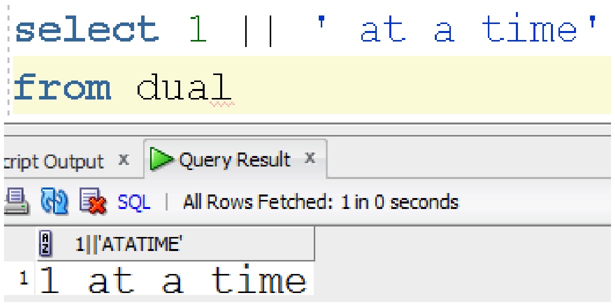
```
150 %
Results  Messages
   (No columr
1  9
```

Operator precedence is applied *to all expressions in all SQL queries without variation*, so within the context of a particular DBMS and version, we know that the same set of rules applies everywhere. Each DBMS follows a strict set of rules to determine the results of an expression, and we can use these rules judiciously to ensure we obtain the results we expect.

7. In your own words, explain what an expression is, and how operator precedence plays a role in determining what the result of an expression is.

8. While operator precedence determines the order of operations in an expression, *datatype precedence* determines the datatype that results from an expression. The result of each expression has a datatype, such as a VARCHAR, DATE, and so on, and this helps determine how the SQL client will display the result. For example, if we add

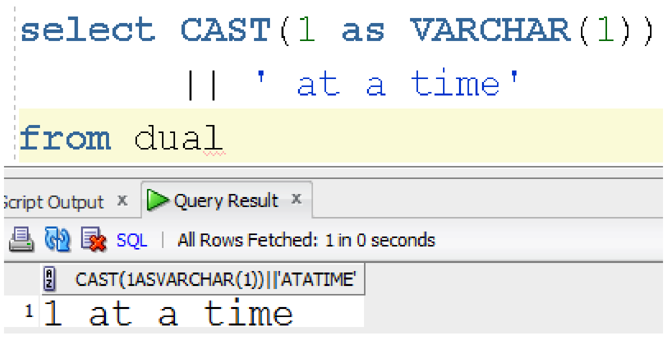
an integer value and a floating-point value, such as “1 + 2.33”, the datatype precedence of many modern relational DBMS will cause the datatype of that expression to be a floating point number rather than an integer. This makes sense, because if the result were an integer, the result could only be 3 if rounded down, or 4 if rounded up. But a floating point number datatype supports the expected result of 3.33. Many modern relational DBMS also support conversion to character values. For example, the expression “1 || ‘ at a time’” in oracle will produce a result of “1 at a time”, as shown below.



```
select 1 || ' at a time'
from dual
```

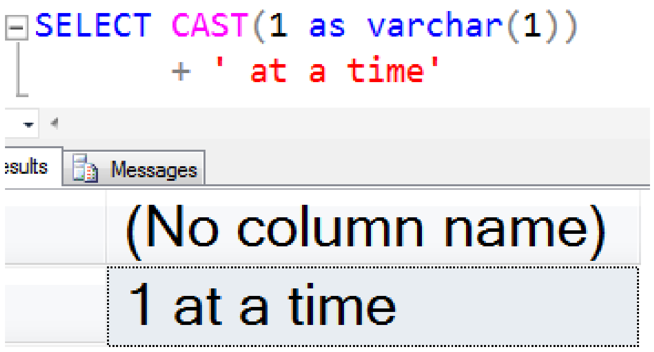
```
cript Output x   Query Result x
SQL | All Rows Fetched: 1 in 0 seconds
  1||'ATATIME'
1 1 at a time
```

Even though 1 is an integer, ‘ at a time’ is a character string, and Oracle’s datatype precedence rules dictate the result to be a character string. Modern relational DBMS have strict sets of rules that determine both the result and the datatype of the result from expressions.
Carefully combining operands with different datatypes in an expression will get you the resulting datatype you want, but using *explicit* datatype conversions is more production-worthy and maintainable. When we simply use operands and operators, we are relying on *implicit* datatype conversion. Even when that works, it is less clear to the reader how it is working, and implicit conversions are subject to change when the database is upgraded. Thankfully we can also use specific constructs to explicitly convert the datatypes. For example, if we want to concatenate an integer with a character sequence, we can explicitly cast the integer to a character sequence using the CAST function. Look at the equivalent examples below in Oracle and SQL Server, respectively.



```
select CAST(1 as VARCHAR(1))
       || ' at a time'
from dual
```

```
Script Output x   Query Result x
SQL | All Rows Fetched: 1 in 0 seconds
  CAST(1ASVARCHAR(1))||'ATATIME'
1 1 at a time
```



```
SELECT CAST(1 as varchar(1))
       + ' at a time'
```

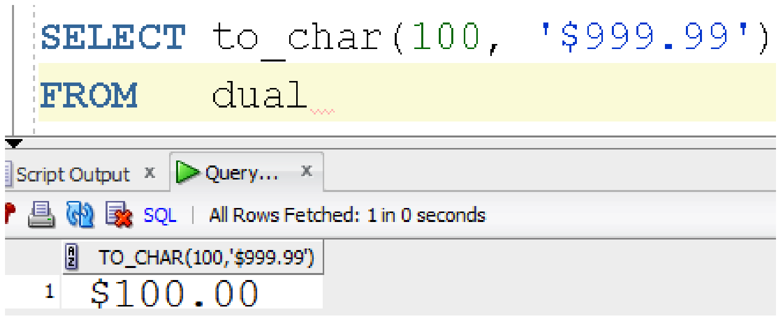
sults | Messages

```
(No column name)
1 at a time
```

Notice that the integer 1 was explicitly cast to a Varchar datatype, and that Varchar is then concatenated to the character sequence " at a time". Oracle uses the characters || as a concatenation operator, while SQL Server uses + as a concatenation operator. With this expression, we do not rely not on implicit datatype conversion. There is much detail involved with datatype conversion, so much so that we could go on exploring it with this entire lab, or even an entire book, but so that we do not lose the focus of this lab, let us move on. Suffice it to say, expressions in modern relational DBMS support operands that have different datatypes, and we explicitly or implicitly convert between datatypes to arrive at a final datatype for the expression's result.

9. In your own words, explain how datatype precedence plays a role in determining what the result of an expression is.

10. While we can use expressions to transform data values into other values, we can use formatting constructs to explicitly form the structure and appearance of the results. These constructs typically come in the form of formatting functions, which tend to be vendor specific; formatting functions for Oracle tend not to be available in SQL Server and vice versa. For example, suppose we want to display the raw number "100" as a currency, indicating it is 100 dollars. In Oracle, we can use the **`to_char`** function, as shown in the screenshot below.



```
SELECT to_char(100, '$999.99')
FROM   dual
```

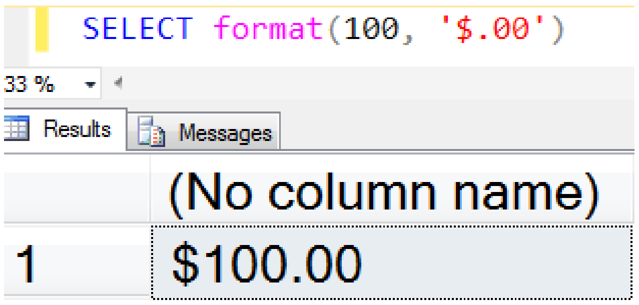
Script Output × | Query... ×

SQL | All Rows Fetched: 1 in 0 seconds

```
  TO_CHAR(100,'$999.99')
1 $100.00
```

The first argument to **`to_char`** is the number we want to format, in this case, 100, and the second argument is a series of characters that describes exactly how we want to format the number, which Oracle documentation refers to as a *format model*

(Oracle, 2016). In this case, the “$” symbol indicates we want to prefix the number with our currency symbol, the “.” symbol indicates we want to make use of a decimal point, the initial three “9” digits indicate we want to display up to three digits to the left of the decimal point, and last two “9” digits indicate we want to display two digits to the right of the decimal point. In SQL Server, we can use the **`format`** function, as shown in the screenshot below.



```
SELECT format(100, '$.00')
```

```
   (No column name)
1  $100.00
```

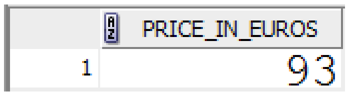
Similar to Oracle’s **`to_char`** function, the first argument to **`format`** is the number and the second argument is a series of characters that describes how we want to format the number (Microsoft, 2016b). The “$” symbol indicates the dollar sign symbol should prefix the result, the “.” symbol indicates we want to make use of a decimal point, and the two “0” digits indicate we want to display two digits to the right of the decimal point. The results for both Oracle and SQL server are conventionally, what we expect to see for a currency – $100.00. Note that your database’s region settings determine what currency symbol, digit group separator symbol, and decimal-point indicator symbol your database will use when formatting a currency, so your results may vary if your region is not the United States. Formatting functions are useful for displaying results in patterns human beings conventionally expect.

11. Following the examples in step 9, format the raw number “10500.37” as a currency using your database, so that the result is “$10,500.37”. Notice that the digits to the left of the decimal point are grouped in threes (termed a *period* in mathematics), so you will need to research how to do this in your database. Capture a screenshot of the SQL command and the results of its execution.

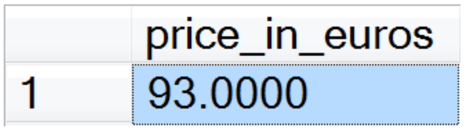
12. Let us end our exploration of the significant components that determine how a value is displayed by reviewing SQL client differences. In step 2 you observe Oracle SQL Developer and SSMS displaying the result of the same query differently. The query:

```
SELECT price_in_us_dollars * 0.93 AS price_in_euros
FROM   Product
WHERE  product_name = 'Cashmere Sweater'
```

has results in Oracle and SQL Server, respectively, as follows:



	PRICE_IN_EUROS
1	93



	price_in_euros
1	93.0000

Additionally in step 8, you learn that a whole number operand and a floating point number operand results in a floating point number, due to datatype precedence. Oracle SQL Developer is displaying the result “93” as if it is a whole number. Why? Because Oracle SQL Developer’s default is to automatically truncate 0 digits after the decimal points since they have no numerical value. So if Oracle SQL Developer receives a value of “93.0” or “93.00”, or presumably any number with multiple 0 digits to the right of the decimal point, it will remove the 0 digits. However, you notice that SSMS displays the results as “93.0000”, with 4 digits to the right of the decimal point. Clearly, SSMS does *not* truncate 0 digits to the right of the decimal point, at least not the first 4. This small example highlights the important point that *different SQL clients may display the same results differently*.

Even more interesting is the reason *why* different clients display results differently. Primarily, these discrepancies occur because each team or vendor creating its SQL client decides independently how results should be displayed, and different teams make different choices. There is no requirement or enforcement that different teams must make the same choices. This holds true even for SQL clients for the *same* database. For example, Oracle SQL Developer may display a result one way, the Toad client another, the PL/SQL Developer client another, all for the same Oracle instance! PL/SQL Developer, for example, gives you the option to “zoom in” on any particular value, where you can view the value as text, in binary as hexadecimal, in XML, and even as a picture (assuming the result is a picture). The same goes for SQL Server, where Toad and SSMS, for example, may display results differently. The key observation here is that *each SQL client is an application*, and is not bound to mechanically display a value in its raw form, but the client displays it in a way more useful to us. If we were to author our own application, such as a series of web pages that display values from our database, we would need to make our own choices as to how these values would be displayed. SQL clients are merely applications developed by application teams, and these teams decide how results are displayed.

13. Explain in your own words why different SQL clients may display the same results differently.

Section Two – Subqueries

Overview

In this section we learn to work with subqueries, which significantly extend the expressional power of queries. Through the use of subqueries, a single query can extract result sets that could not be extracted without subqueries. Subqueries enable the query creator to ask the database for many complex structures in a single query.

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 $\text{PROJECT}_{\text{last\_name}}$(Person).

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

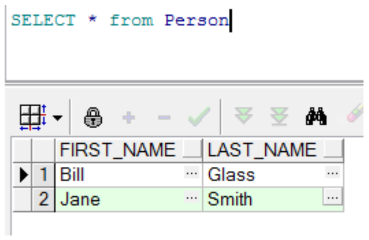
$\textbf{SELECT}_{\textbf{last\_name=Smith}}\textbf{(PROJECT}_{\textbf{last\_name}}\textbf{(Person))}$ =

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.

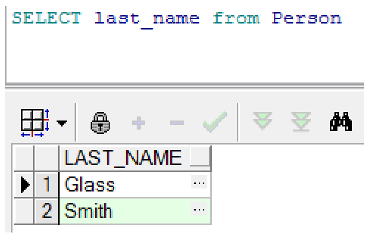
Let us look at a concrete example of a subquery using a Person table with the same rows as the Person relation just described. When we issue a **`SELECT * FROM Person`** command, we see the initial Person table with Bill Glass and Jane Smith as row values.



```
SELECT * from Person
```

	FIRST_NAME	LAST_NAME
1	Bill	Glass
2	Jane	Smith

To perform the $\textbf{PROJECT}_{\textbf{last\_name}}\textbf{(Person)}$ 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
```

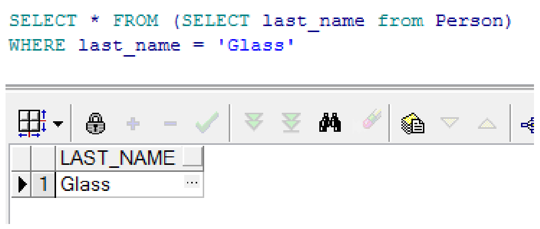
	LAST_NAME
1	Glass
2	Smith

Now if we would like to perform the equivalent of the **SELECT$_{\text{last\_name=Smith}}$(PROJECT$_{\text{last\_name}}$(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

Steps

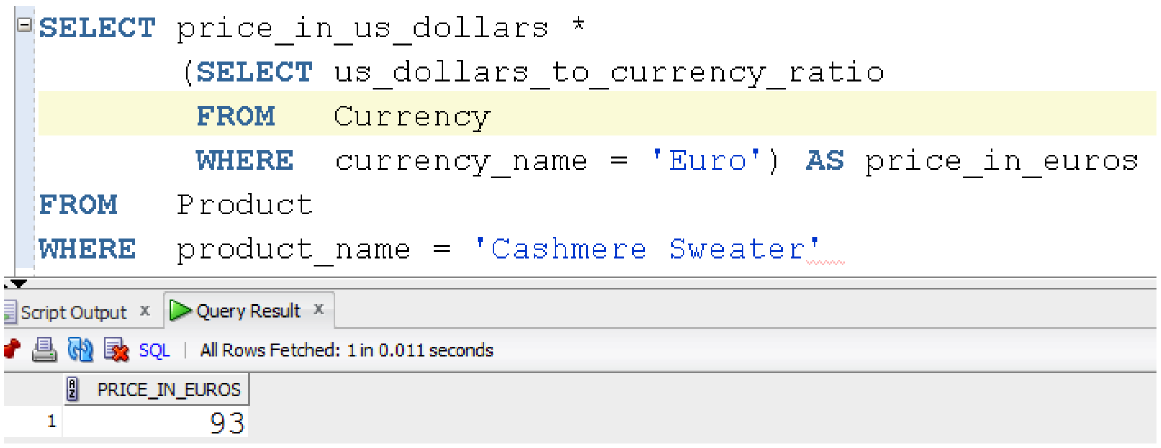
14. In step 2 we determine the price of a Cashmere sweater in Euros by using two independent queries, but in this step we learn to use the superior method of combining both queries into one. In step 2 we use this query to find out that the ratio is 0.93:

```
SELECT us_dollars_to_currency_ratio
FROM   Currency
WHERE  currency_name = 'Euro'
```

Then we manually hardcode that value into the next query to obtain the price in Euros, €93.00:

```
SELECT price_in_us_dollars * 0.93 AS price_in_euros
FROM   Product
WHERE  product_name = 'Cashmere Sweater'
```

Executing two queries independently gives us the results we want, but there is a better way. Embedding one query inside the other is more concise and production-worthy. Let us try it out in Oracle then SQL Server to see if we still get the same result.

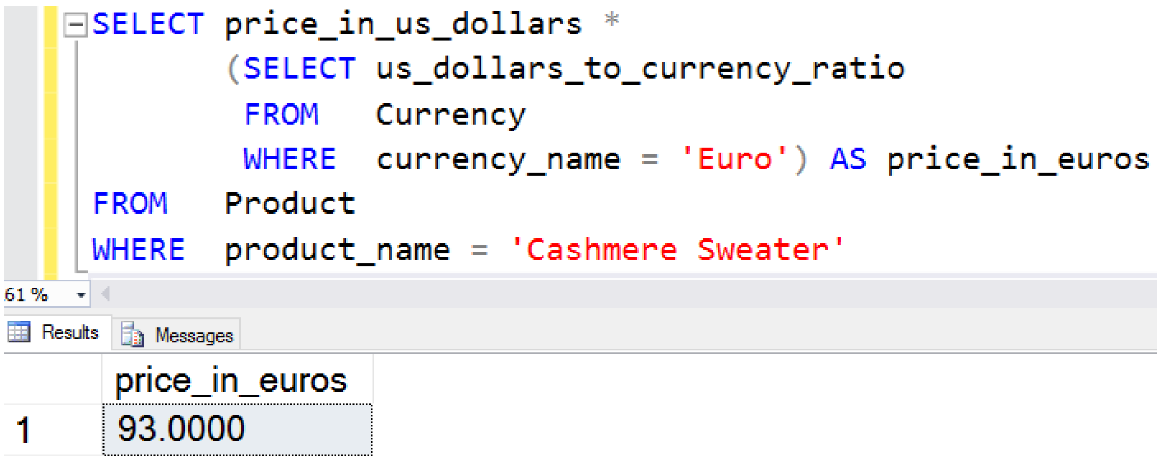


```
SELECT price_in_us_dollars *
       (SELECT us_dollars_to_currency_ratio
        FROM   Currency
        WHERE  currency_name = 'Euro') AS price_in_euros
FROM   Product
WHERE  product_name = 'Cashmere Sweater'
```

Script Output | Query Result

SQL | All Rows Fetched: 1 in 0.011 seconds

```
   PRICE_IN_EUROS
1              93
```



```
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'
```

61 %

Results | Messages

```
   price_in_euros
1  93.0000
```

The result is the same in both databases (except for formatting). 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.

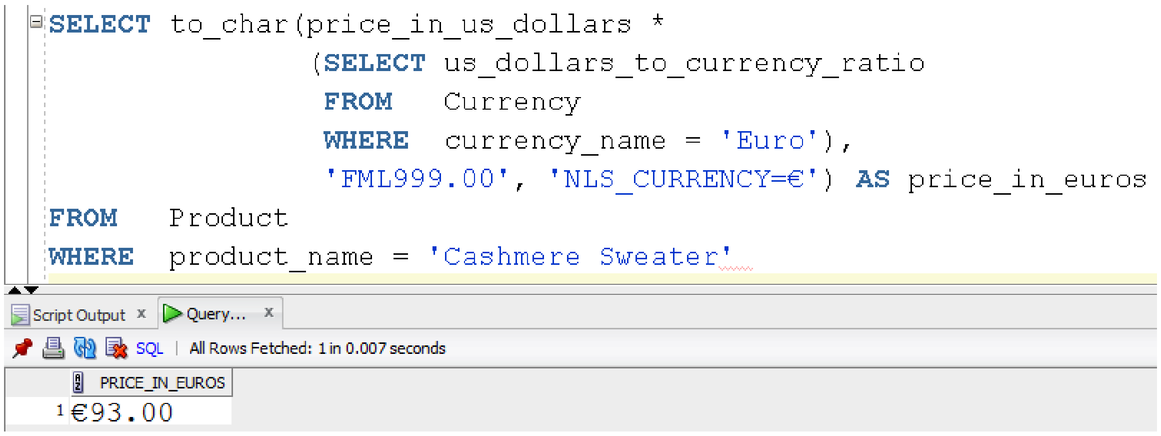
```
1: SELECT price_in_us_dollars *
2:        (SELECT us_dollars_to_currency_ratio
3:         FROM   Currency
4:         WHERE  currency_name = 'Euro') AS price_in_euros
5: FROM   Product
6: WHERE  product_name = 'Cashmere Sweater'
```

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.

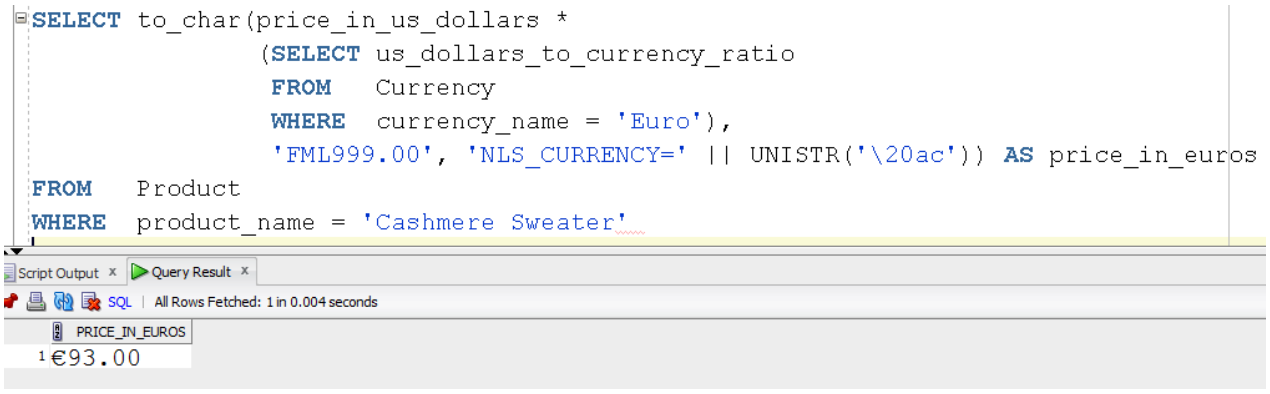
For completeness, let us format the result so that we see it as a monetary amount. In Oracle, we would modify the query as the following screenshot illustrates:



```
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'
```

```
  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.



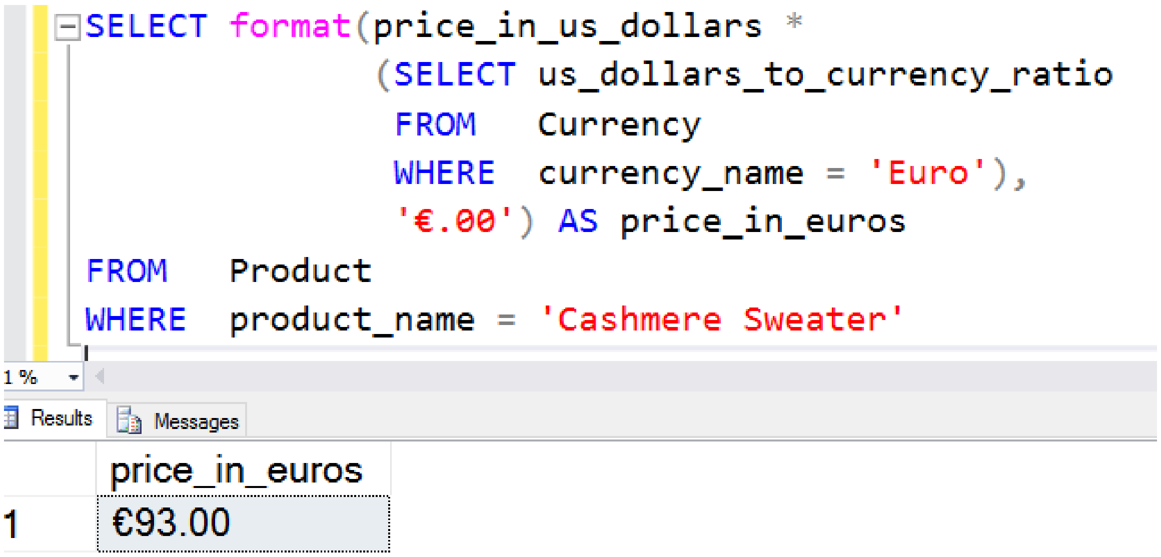
```
SELECT to_char(price_in_us_dollars *
                (SELECT us_dollars_to_currency_ratio
                 FROM   Currency
                 WHERE  currency_name = 'Euro'),
                 'FML999.00', 'NLS_CURRENCY=' || UNISTR('\20ac')) AS price_in_euros
FROM   Product
WHERE  product_name = 'Cashmere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

Script Output × | Query Result ×

SQL | 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:

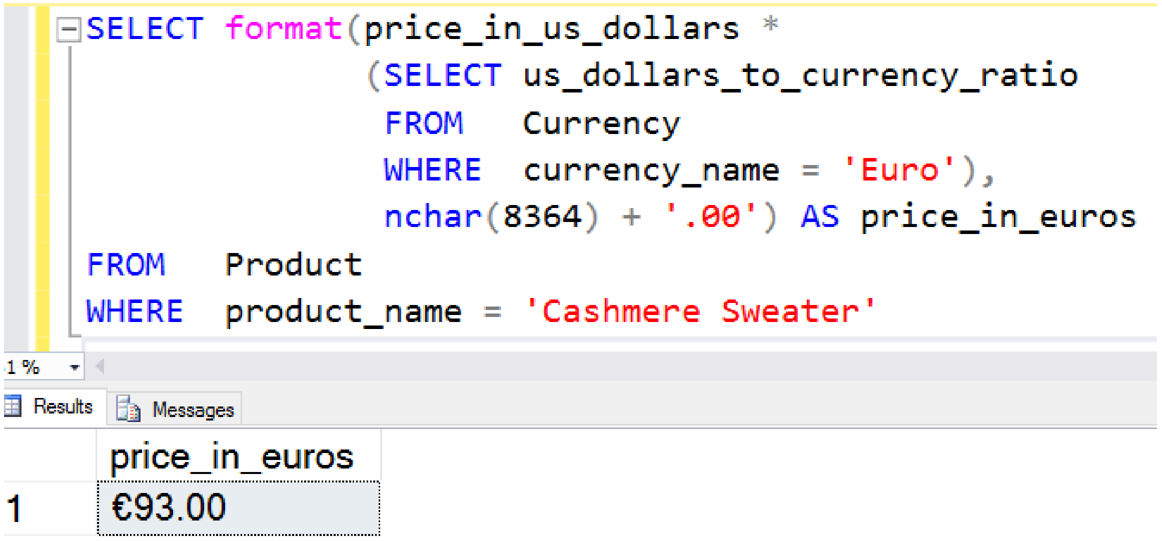


```
SELECT format(price_in_us_dollars *
              (SELECT us_dollars_to_currency_ratio
               FROM   Currency
               WHERE  currency_name = 'Euro'),
               '€.00') AS price_in_euros
FROM   Product
WHERE  product_name = 'Cashmere Sweater'
```

Results | Messages

```
  price_in_euros
1 €93.00
```

Just like with Oracle, we held 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.



```
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.

15. Explain in your own words:

    a. why using a subquery is superior to executing two independent queries and hardcoding the results of the first in the second.

    b. what it means for a subquery to be uncorrelated, including how and when an uncorrelated subquery is executed in the context of the outer query.

16. In step 3, you are asked to give the price of a flowing skirt in Cancun by writing two queries. You now have the skills to retrieve this in a single query! Do so now, making sure to format the result as a currency. It is convention to use either "$" or "Mex$" as the currency symbol for Mexican Pesos.

    a. Capture the results of the query and its execution.

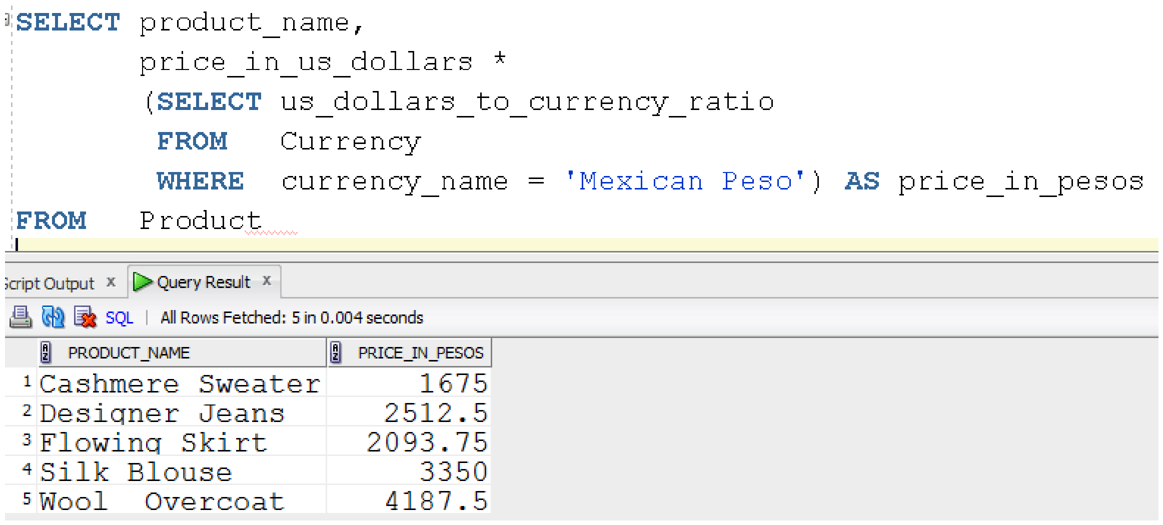
    b. Explain how your solution makes use of a subquery to help retrieve the result, and the advantages of your new solution over your solution in step 3.

17. We know how convert currencies in our example schema for display purposes, but what about making a decision based off the conversion? Imagine that a Mexican

customer wants to know the names and prices of all products that cost less than 2,750 Mexican Pesos. Following the same methodology as in step 14, we can start by calculating that for all products with the following query.

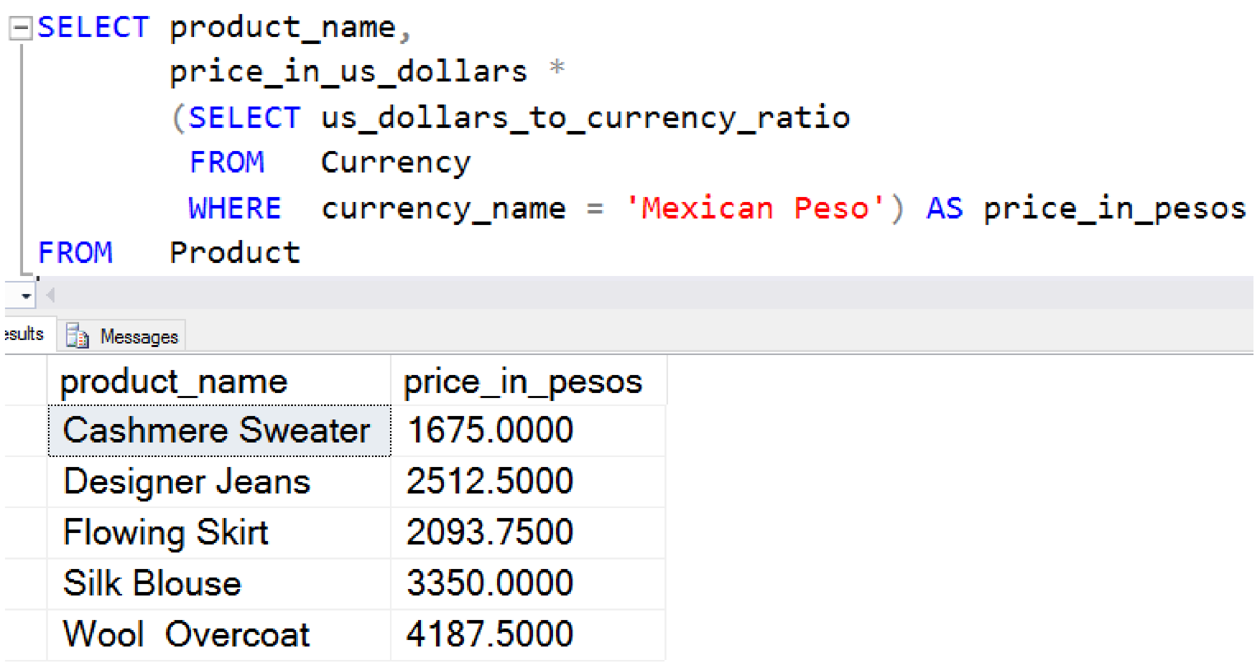
```
SELECT product_name,
       price_in_us_dollars *
       (SELECT us_dollars_to_currency_ratio
        FROM   Currency
        WHERE  currency_name = 'Mexican Peso') AS price_in_pesos
FROM   Product
```

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 and SQL Server, respectively, are below.



```
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       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
```



```
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
```

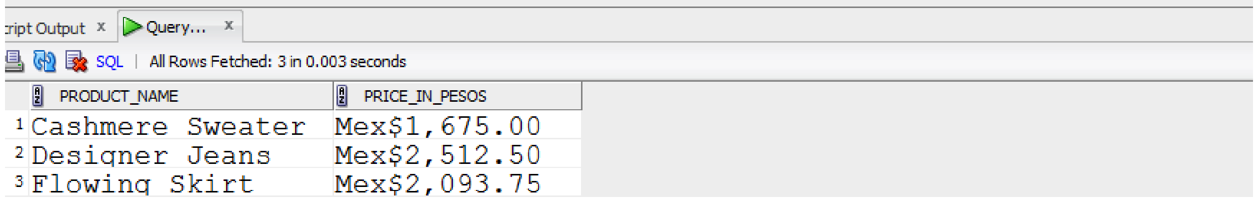
product_name	price_in_pesos
Cashmere Sweater	1675.0000
Designer Jeans	2512.5000
Flowing Skirt	2093.7500
Silk Blouse	3350.0000
Wool Overcoat	4187.5000

We have obtained the price of *all* products, and now we 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.

```
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
WHERE  price_in_us_dollars *
       (SELECT us_dollars_to_currency_ratio
        FROM   Currency
        WHERE  currency_name = 'Mexican Peso') < 2750
```

The formatted results are shown for Oracle below.

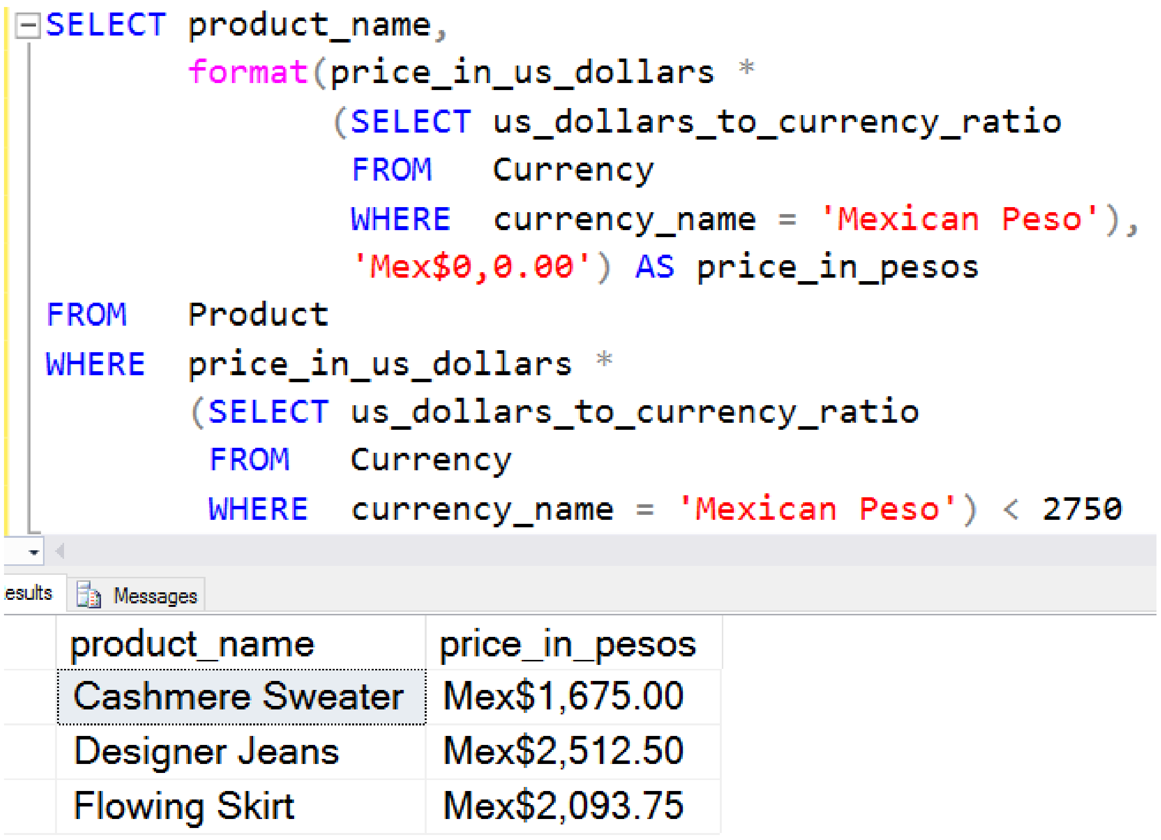
```
SELECT product_name,
       to_char(price_in_us_dollars *
               (SELECT us_dollars_to_currency_ratio
                FROM   Currency
                WHERE  currency_name = 'Mexican Peso'),
                'FML999,999.00', 'NLS_CURRENCY=Mex$') AS price_in_pesos
FROM   Product
WHERE  price_in_us_dollars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



```
ript Output ×   Query... ×
SQL | 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.



```
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.

Notice that in the results, each product in the list cost less then Mex$2,750. It is possible to add a subquery to a WHERE clause in order to make decisions based on the result of the subquery.

This example 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.

18. Now that you know how to use subqueries in the WHERE clause, let us use your skills to address a slightly more involved use case. Imagine that a Canadian store manager wants to know which products are on the cheaper end and which products are on the more expensive end. The manager asks you to retrieve the names and prices of all products that are less than 150 Canadian Dollars or more than 300 Canadian Dollars. Retrieve the results needed for this use case using a single query, and capture a screenshot of the query and its execution. Make sure to format the results in Canadian Dollars. The currency symbol for Canadian Dollars is the same as for U.S. Dollars, the "$" symbol.

19. 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 one such example.

> 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.

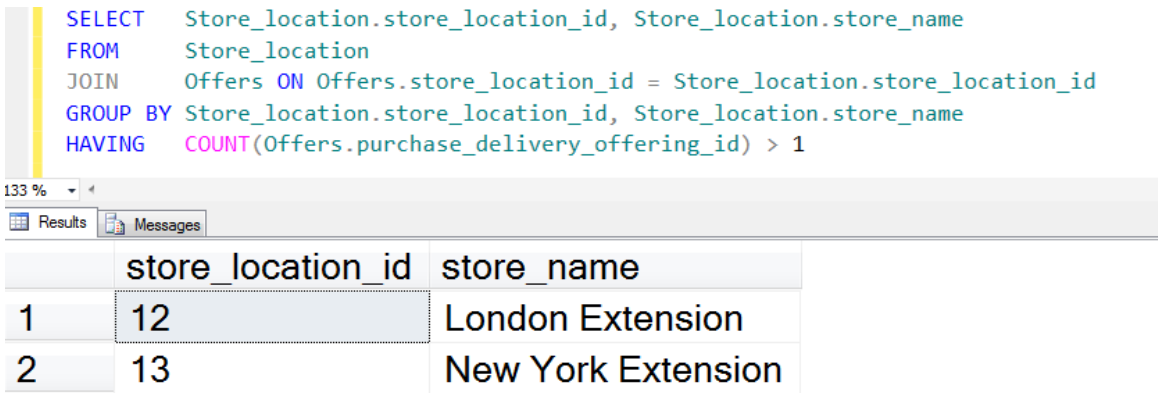
This use case is a challenge because it has two parts that require different SQL strategies. In order to determine which locations are suitable for Jill, we need to count the number of purchase and delivery options offered by each location, which means we need to aggregate results. However, because aggregation 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.

```
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
```

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 and SQL Server, illustrating that the London Extension and the New York Extension both have more than one purchase and delivery offering.



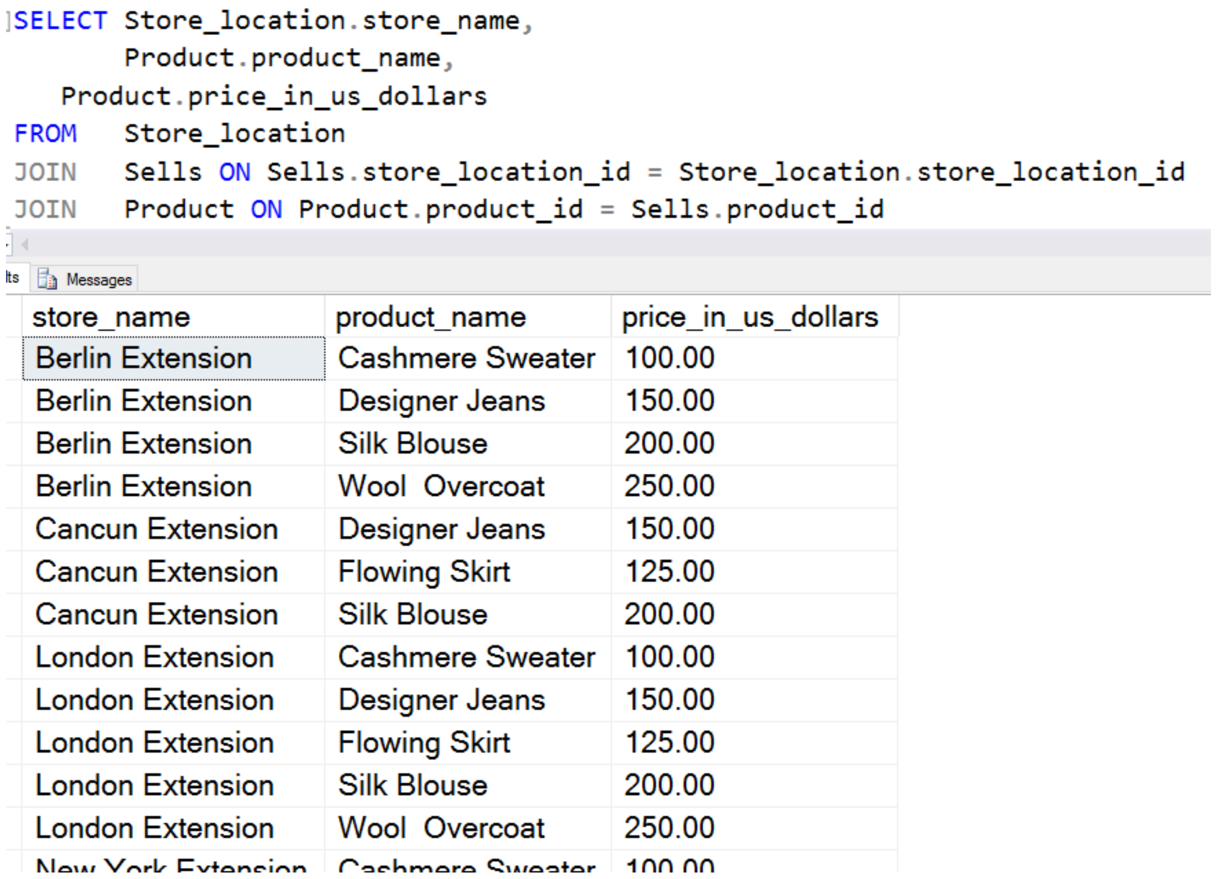
```
SELECT   Store_location.store_location_id, Store_location.store_name
FROM     Store_location
JOIN     Offers ON Offers.store_location_id = Store_location.store_location_id
GROUP BY Store_location.store_location_id, Store_location.store_name
HAVING   COUNT(Offers.purchase_delivery_offering_id) > 1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

	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.

```
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
```

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, and the results are truncated for brevity.



```
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
```

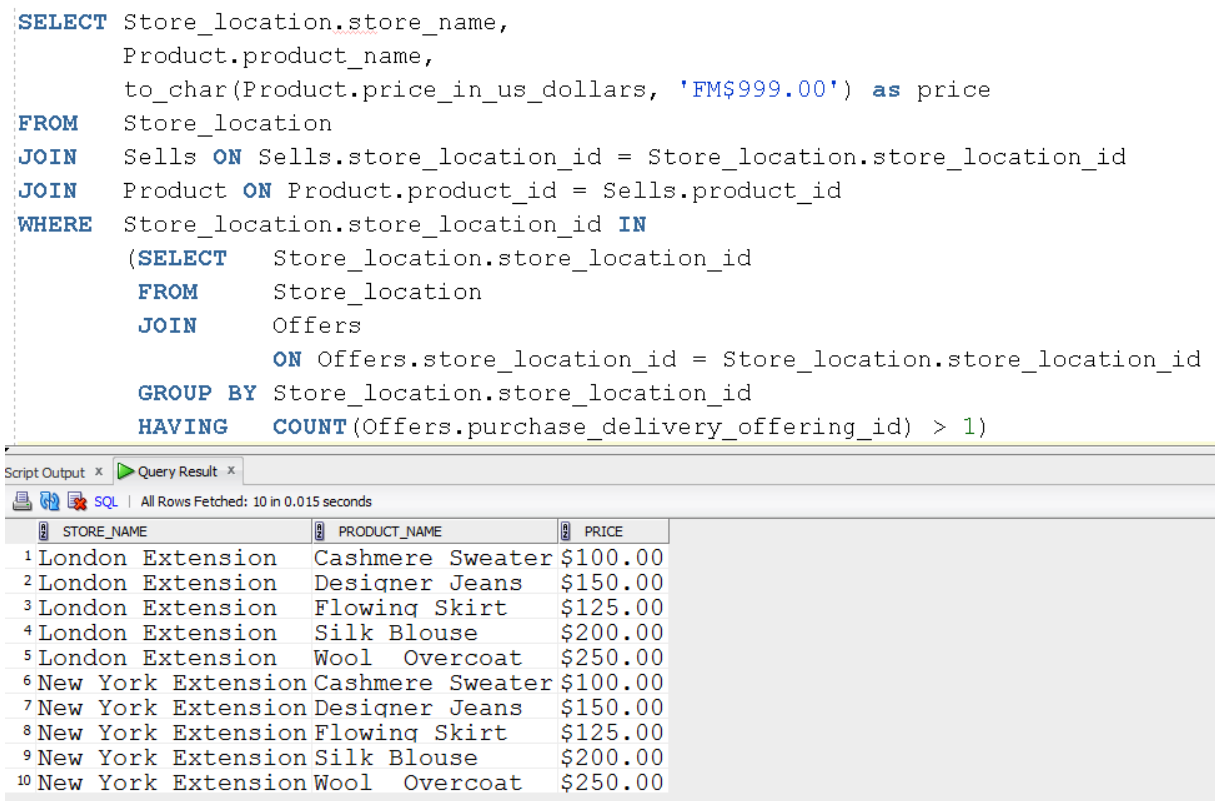
store_name	product_name	price_in_us_dollars
Berlin Extension	Cashmere Sweater	100.00
Berlin Extension	Designer Jeans	150.00
Berlin Extension	Silk Blouse	200.00
Berlin Extension	Wool Overcoat	250.00
Cancun Extension	Designer Jeans	150.00
Cancun Extension	Flowing Skirt	125.00
Cancun Extension	Silk Blouse	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 embed the first query into the second another to retrieve the results we need, as illustrated below.

```
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.

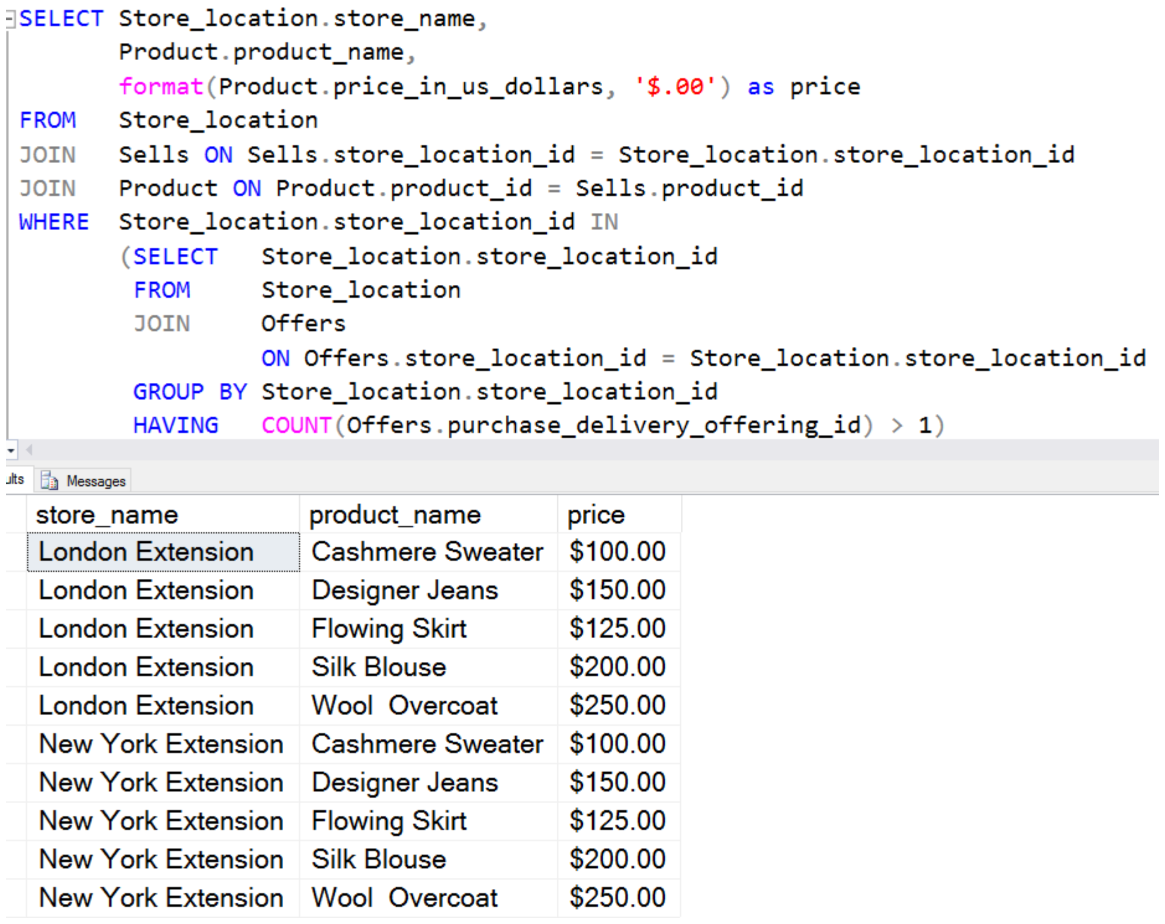


```
SELECT Store_location.store_name,
       Product.product_name,
       to_char(Product.price_in_us_dollars, 'FM$999.00') as price
FROM   Store_location
JOIN   Sells ON Sells.store_location_id = Store_location.store_location_id
JOIN   Product ON Product.product_id = Sells.product_id
WHERE  Store_location.store_location_id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



```
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

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.

20. Now it is time for another, more complex use case.

> Like Jill, Marcus also travels internationally, and he is interested in considering some products to purchase. Because his employer sends him to various locations throughout the world with little notice, he only wants to consider a product if it is available in all store locations, and is not interested in products that are available in some but not all store locations. This way, should he decide to purchase a product, he has the assurance that can purchase it at any of the locations. Lastly, he is interested in viewing the sizing options for each product that meets his criteria.

This use case can be solved using a similar methodology as in step 19. You can create different queries to solve different parts, then put them together to address the entire use case. For example, one of the subqueries can be put into the WHERE clause to limit what is returned. 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, we can see that there are 5 locations. Retrieving products that are sold at exactly 5 locations addresses Marcus’ request *at this present time*, but even better is to dynamically determine the total number of locations in the query itself so that it returns correct results over time, even if the number of locations changes. For example, if one store location closes, there would be 4 locations instead of 5. Likewise, if a new store location is opens, there would be 6 locations. Dynamically retrieving the number of locations is a superior solution.

a. Think about which parts can be solved with independent queries, then identify and briefly explain them here.

b. Write an independent query for each part you identified, and capture a screenshot of the execution and results of each query. Explain the role each result will play in the final query.

c. Write the full query to address the use case by combining your independent queries into a single, larger query. Capture a screenshot of its execution and results.

d. Explain how you combined the independent queries to get your results, in terms of what SQL constructs you used, and the mechanics of how they work together.

21. Some SQL constructs work with the single values, some with lists of values, and some with tables of values. 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 use case provided in step 19.

> 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 19 which makes use of a subquery in the WHERE clause.

```
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)
```

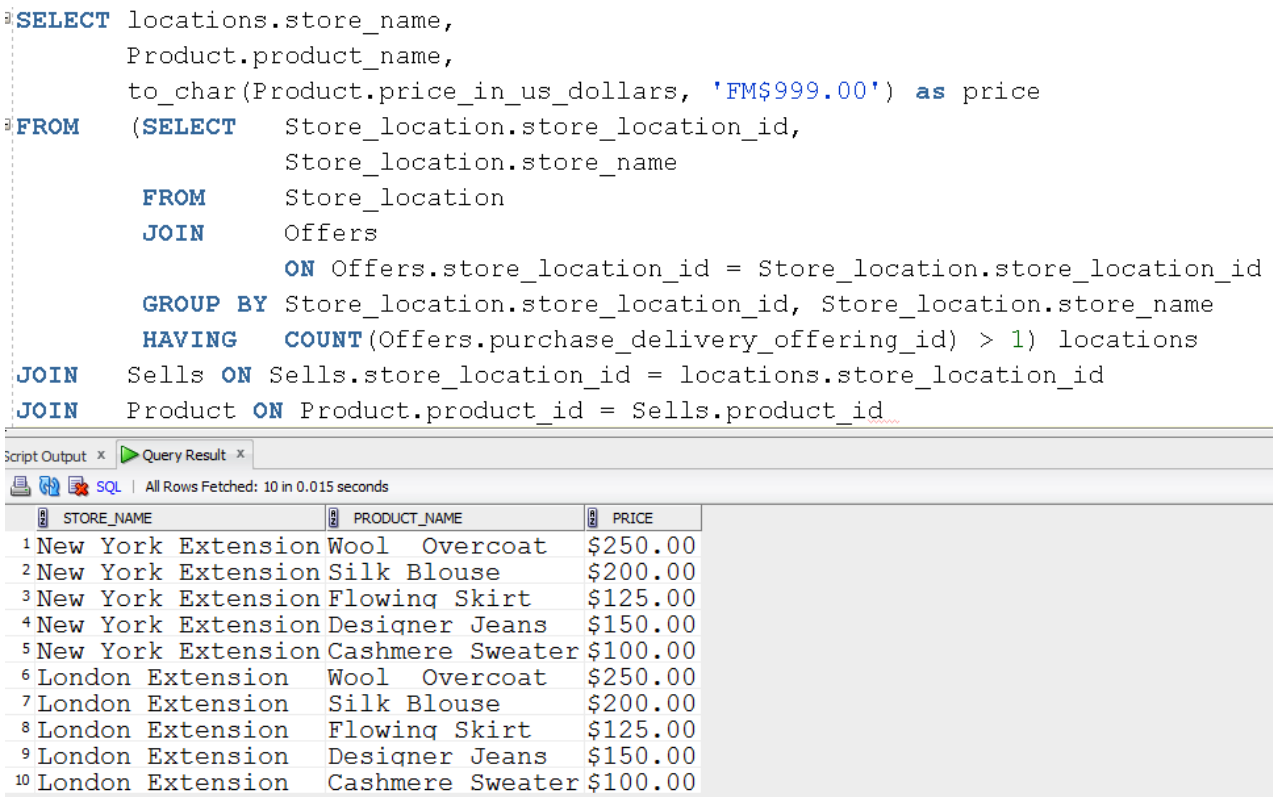
We include the Store_location table on line 4 so that we can retrieve the store name and also match up the locations to the products they sell. We restrict the store locations retrieved based upon the results in the subquery on lines 8-13, thus employing the subquery as a filtering mechanism. The combined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

```
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.



```
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
```

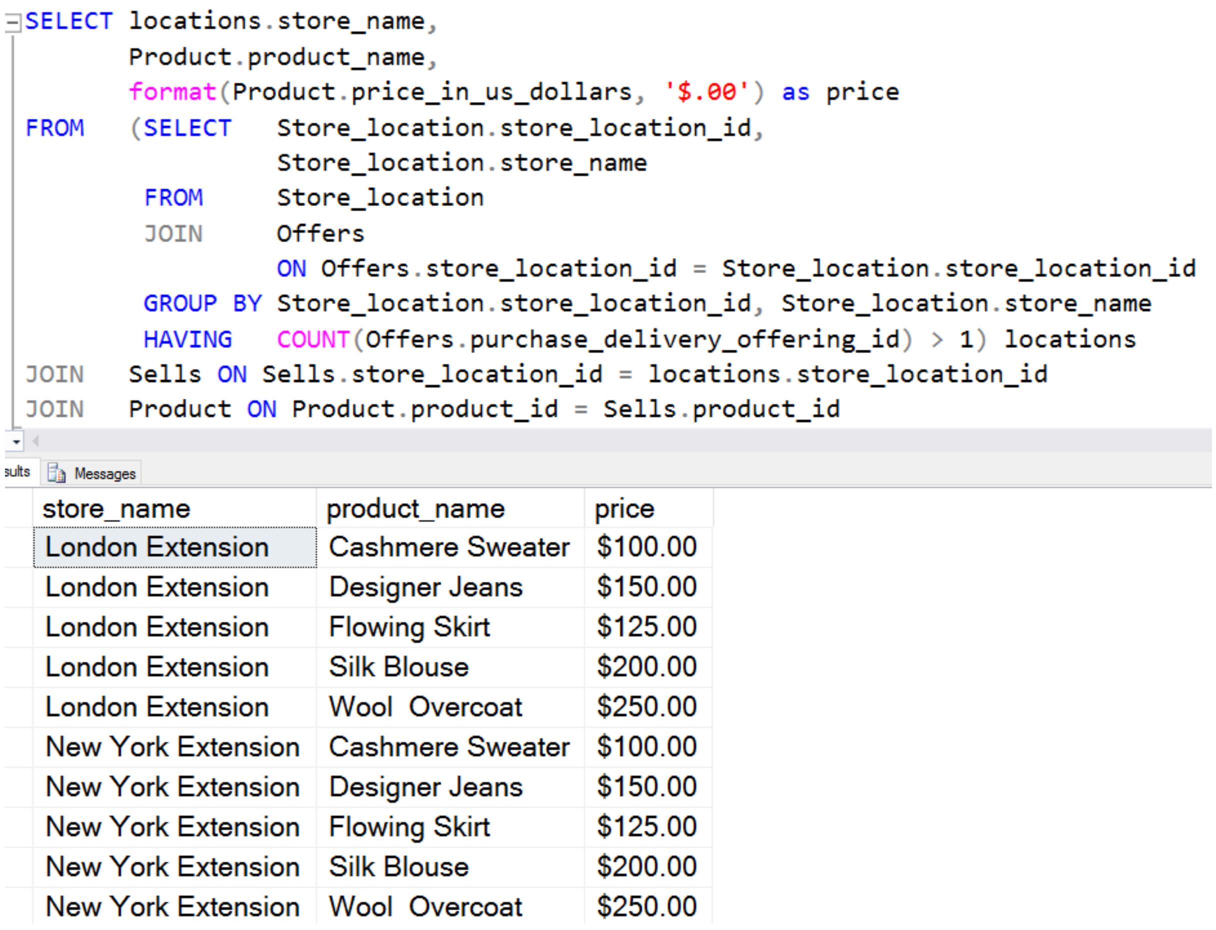
Script Output × Query Result ×

SQL | 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.



```
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 results are the same as in the original solution. Both screenshots illustrate that we can successfully filter rows by using a subquery in the FROM clause.

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 19 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 19 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.

22. You provided a solution to Marcus' use case in step 20 by using a subquery in the WHERE clause. Using a similar methodology as in the prior step, step 21, create an alternative solution by using a subquery in the FROM clause.

    a. Capture a screenshot of your new query and the results of its execution.

    b. Explain in your own words the mechanics of how your new query works to address Marcus' use case.

23. Some use cases have a distinct part that requires the existence of a particular item. Below is one such 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.

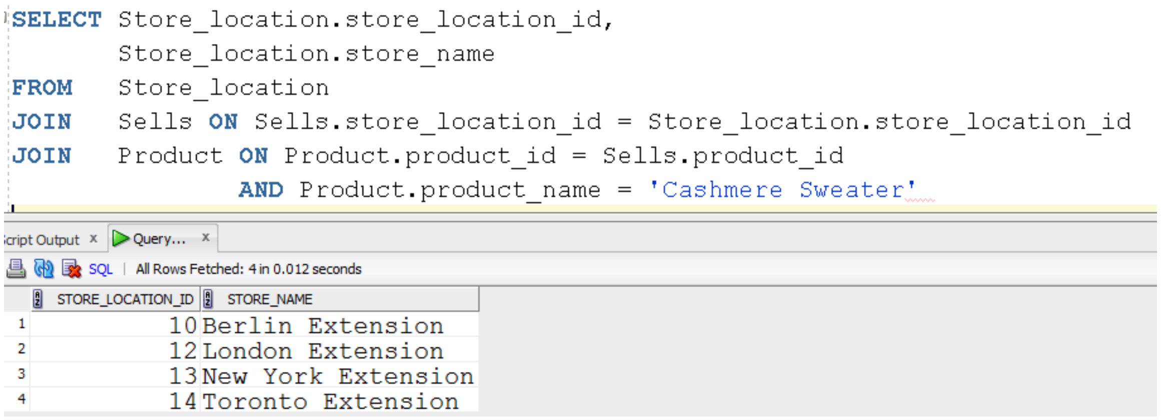
    Just as in prior steps, we can identify the parts, write queries for them, and then put them together. One part comes from this sentence fragment in the use case, "she wants to see all products and their prices in U.S. Dollars". This is straightforward and we can steal a query from step 19 to address this part, shown below.

```sql
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.

```
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 screenshot from Oracle below.



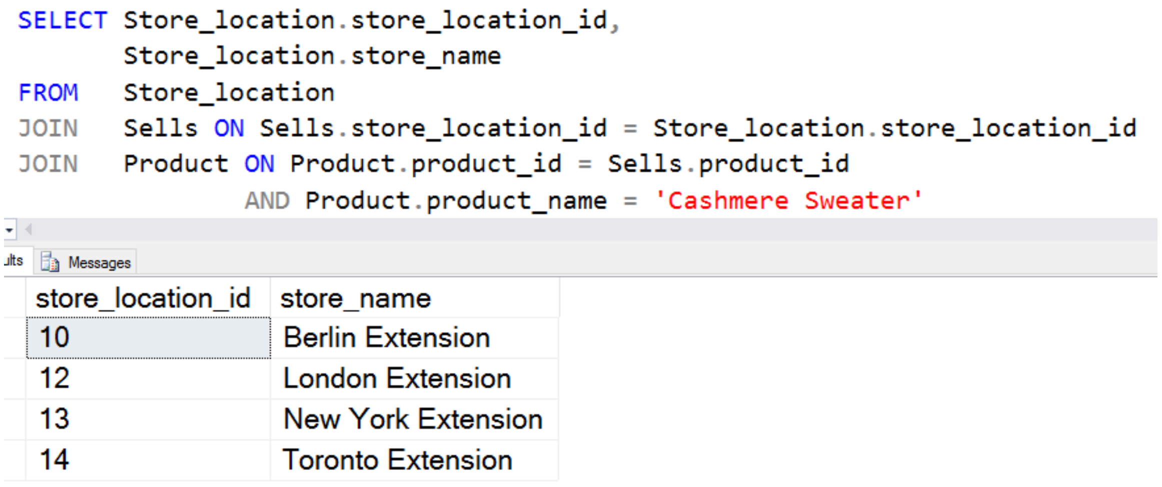
```
SELECT Store_location.store_location_id,
       Store_location.store_name
FROM   Store_location
JOIN   Sells ON Sells.store_location_id = Store_location.store_location_id
JOIN   Product ON Product.product_id = Sells.product_id
               AND Product.product_name = 'Cashmere Sweater'
```

Script Output × | Query... ×

SQL | 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
```

Notice that the Cancun Extension is excluded because it does not sell Cashmere sweaters. This is also illustrated in the SQL Server screenshot below.



```
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'
```

Results | Messages

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

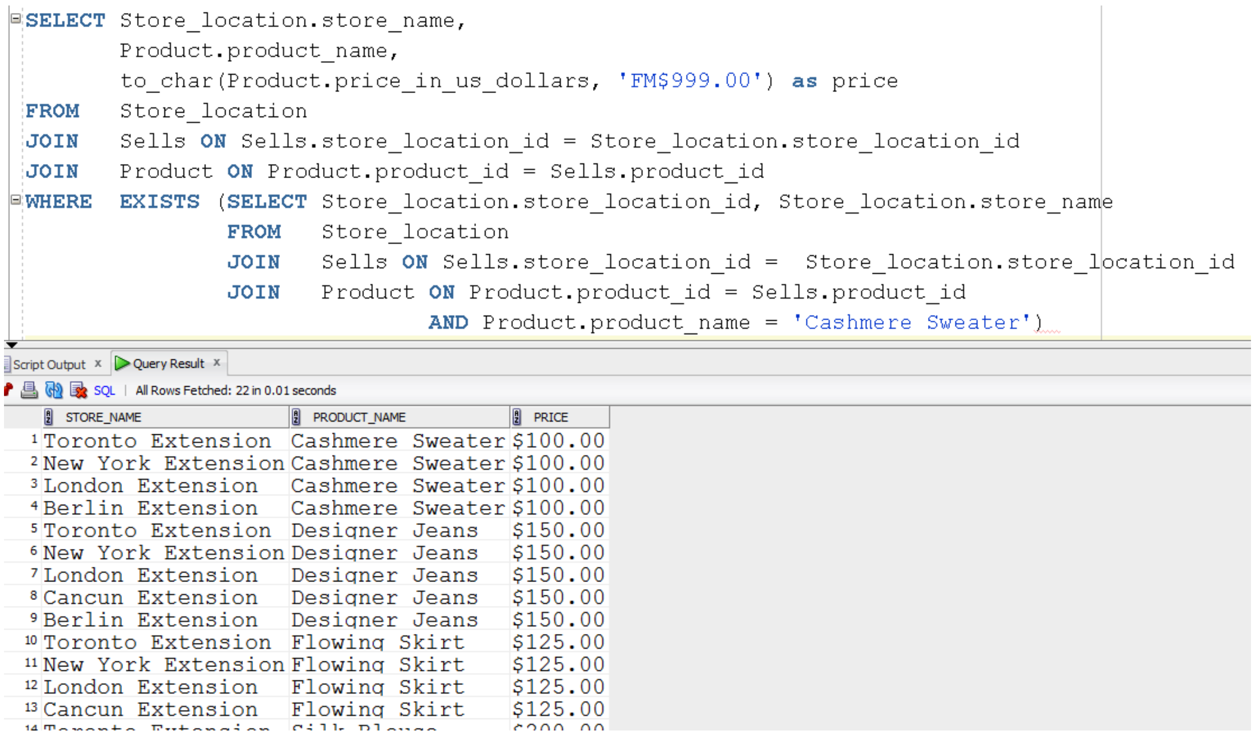
The initial strategy to solve this use case is similar to the initial strategies used to solve use cases in other steps.

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. Let us try it so we see what happens with the query below.

```
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 screenshot below shows part of the items returned in Oracle (there are too many rows to show them all).

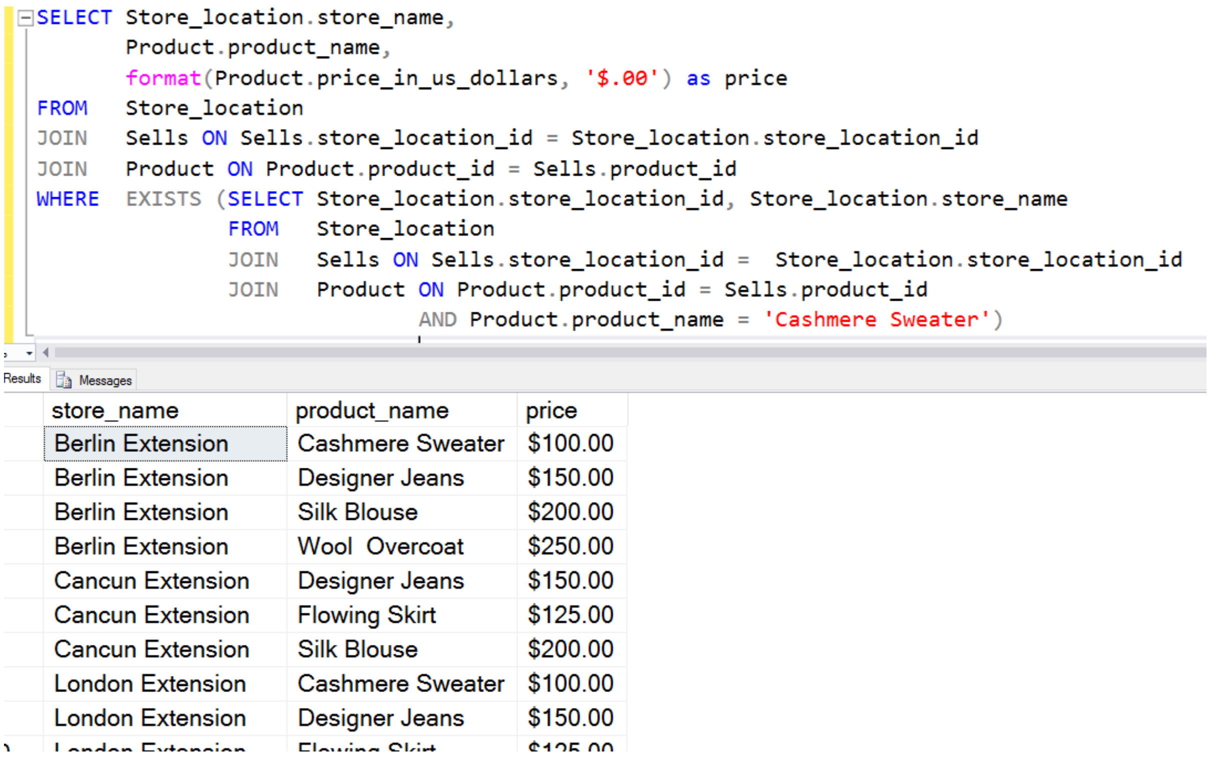


```
SELECT Store_location.store_name,
       Product.product_name,
       to_char(Product.price_in_us_dollars, 'FM$999.00') as price
FROM   Store_location
JOIN   Sells ON Sells.store_location_id = Store_location.store_location_id
JOIN   Product ON Product.product_id = Sells.product_id
WHERE  EXISTS (SELECT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

```
SQL | 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
```

And the screenshot below is for 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
```

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. 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.

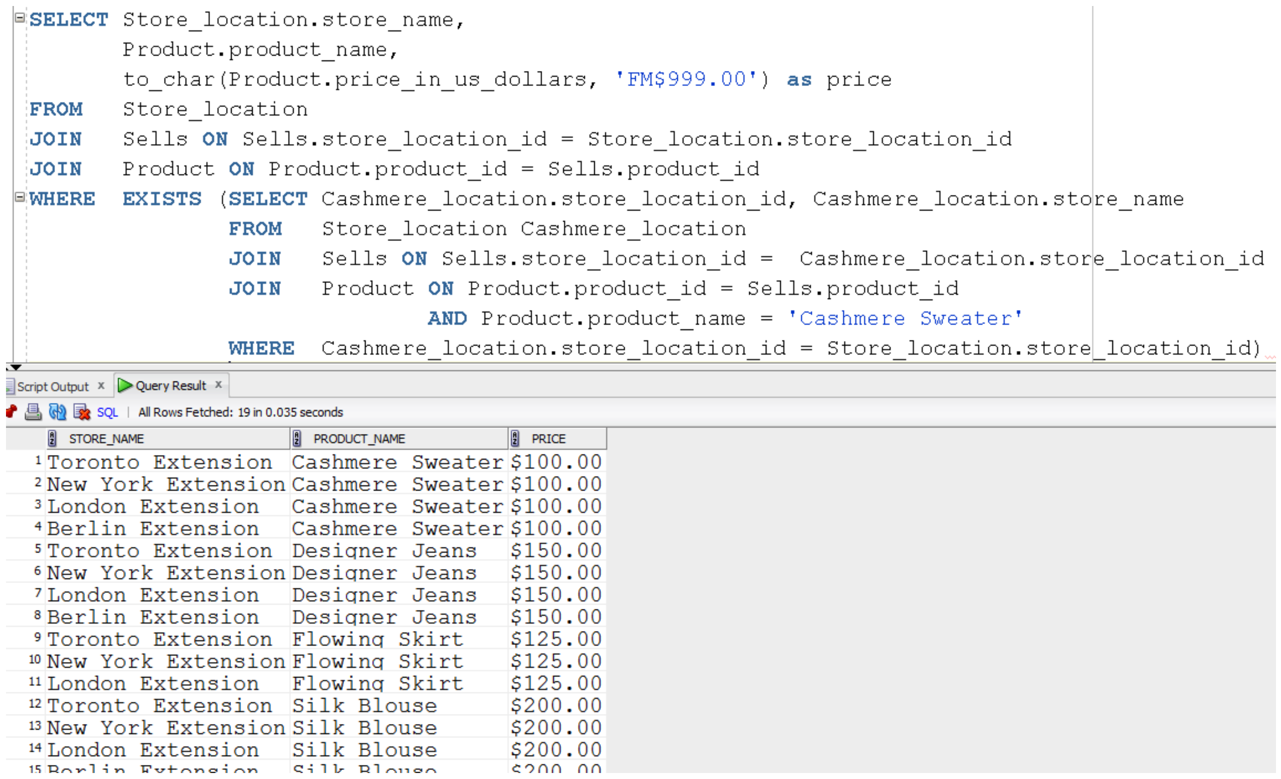
EXISTS usually demands a more complex method of combining the queries for each part. The subquery usually must be *correlated* with the outer query to get the results we want. A correlated subquery references at least one table from the outer query, which means that conceptually, the subquery is not 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.

```
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!

The modified solution containing a correlated subquery gives us the results we want. Below is the screenshot for Oracle, which shows enough rows to see that our query works, but excludes some rows for brevity.



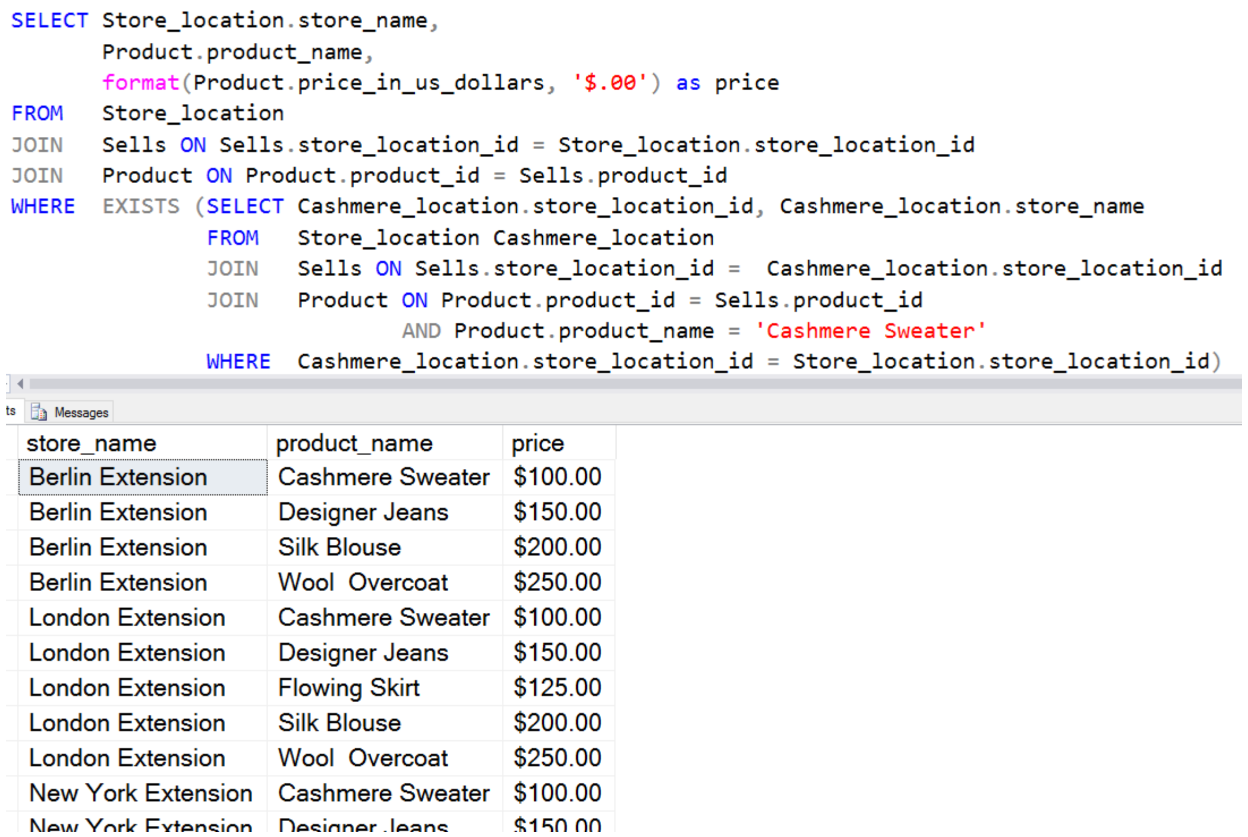
```sql
SELECT Store_location.store_name,
       Product.product_name,
       to_char(Product.price_in_us_dollars, 'FM$999.00') as price
FROM   Store_location
JOIN   Sells ON Sells.store_location_id = Store_location.store_location_id
JOIN   Product ON Product.product_id = Sells.product_id
WHERE  EXISTS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

Script Output × Query Result ×

SQL | 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

And below is the screenshot for SQL Server for the same.



```
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

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.

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.

24. Explain in your own words the differences between a correlated subquery and an uncorrelated subquery.

25. In step 20, you created a solution to Marcus' use case, which is reproduced again below.

    > Like Jill, Marcus also travels internationally, and he is interested in considering some products to purchase. Because his employer sends him to various locations throughout the world with little notice, he only wants to consider a product if it is available in all store locations, and is not interested in products that are only available in some but not all store locations. This way, should he decide to purchase a product, he has the assurance that can purchase it at any of the locations. Lastly, he is interested in viewing the sizing options for each product that meets his criteria.

    You identified the distinct parts, wrote queries for them, and then combined them to solve the use case. For this step, modify your solution to make use of the EXISTS clause and a correlated subquery.

    a. Capture a screenshot of the query and the results of its execution.

    b. Explain what changes you needed to make to your solution to make use of a correlated subquery.

    c. Explain which solution, the one in step 20 or the one in this step, you believe to be more appropriate to solve Marcus' use case, and why. There is no one correct answer, but it is useful to know why you would choose one solution over the other.

Lab Summary

Congratulations! You have learned how to transform values from their original form in the database by using expressions and other techniques. You have also learned how to solve more complex use cases by using subqueries. You are now better equipped to address the wide variety of demanding use cases presented to database designers and developers.

Works Cited

Microsoft (2016a). Operator Precedence (Transact-SQL). *Microsoft SQL Server Language Reference.* Retrieved January 8, 2016, from ttps://msdn.microsoft.com/en-us/library/dn198336.aspx

Microsoft (2016b). FORMAT (Transact-SQL). *Transact-SQL Reference (Database Engine).* Retrieved January 25, 2016, from https://msdn.microsoft.com/en-us/library/hh213505.aspx

Oracle (2015). About SQL Operators. *Database SQL Language Reference*. Retrieved January 8, 2016, from https://docs.oracle.com/database/121/SQLRF/operators001.htm#SQLRF51151

Oracle (2016). Format Models. *Database SQL Language Reference*. Retrieved January 25, 2016, from http://docs.oracle.com/database/121/SQLRF/sql_elements004.htm#SQLRF00210