Vincent Nguyen
Class: CS 669 – Fall 1

Overview of the Lab

In this lab 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. This lab teaches you the mechanics crafting SQL queries that harness the power of subqueries to handle more complex use cases.

From a technical perspective, together, we will learn:

- what correlated and uncorrelated subqueries are and the theory supporting both.
- to use subqueries that return a single value, a list of values, and a table of values.
- to use subqueries that use aggregation.
- to address use cases by using uncorrelated subqueries in the column select list, the where clause, and the from clause.
- to address use cases by using correlated subqueries and an EXIST clause in the WHERE clause.
- how transaction schedules, locks, and multiversioning works with transaction concurrency.

Lab 5 Explanations Reminder

As a reminder, it is important to read through the Lab 5 Explanation document to successfully complete this lab, available in the assignment inbox alongside this lab. The explanation document illustrates how to correctly execute each SQL construct step-by-step, and explains important theoretical and practical details.

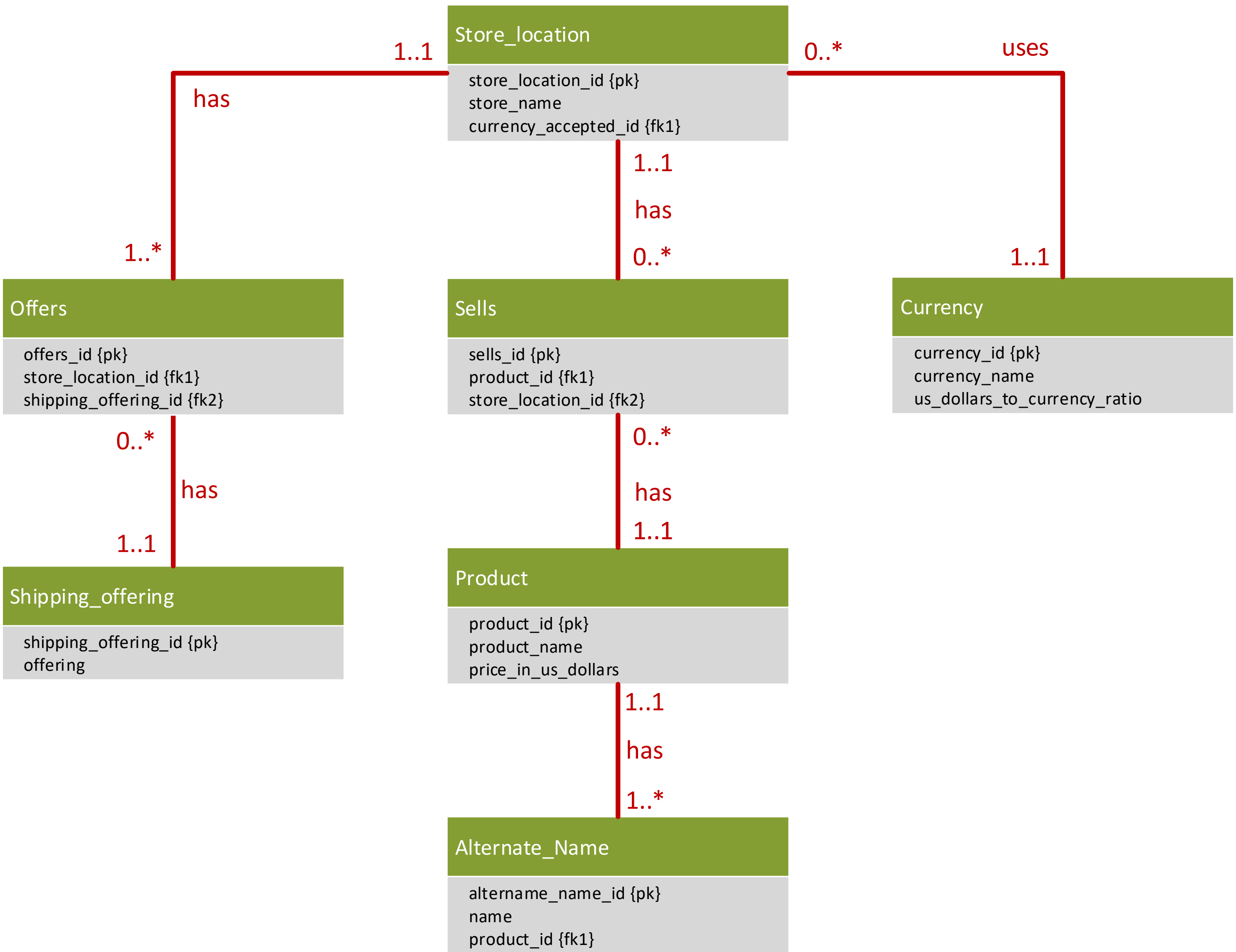
Other Reminders

- The examples in this lab will execute in modern versions of Oracle, Microsoft SQL Server, and PostgreSQL as is.
- The screenshots in this lab display execution of SQL in the default SQL clients supported in the course – Oracle SQL Developer, SQL Server Management Studio, and pgAdmin – but your screenshots may vary somewhat as different version of these clients are released.
- Don’t forget to commit your changes if you work on the lab in different sittings, using the “COMMIT” command, so that you do not lose your work.

Section One – Subqueries

Section Background

In this section, you will practice crafting subqueries for the schema illustrated below.



This schema's structure supports basic medical product and currency information for an international medical supplier, including store locations, the products they sell, shipping offerings, the currency each location accepts, as well as conversion factors for converting from U.S. dollars into the accepted currency. Due to the specific and technical nature of the names of medical products, the supplier also keeps a list of alternative names for each product that may help customers identify them. 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	0.67
Canadian Dollar	1.34
US Dollar	1.00
Euro	0.92
Mexican Peso	16.76

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
Glucometer	$50
Bag Valve Mask	$25
Digital Thermometer	$250
Electronic Stethoscope	$350
Handheld Pulse Oximeter	$450

Sells

Store Location	Product
Berlin Extension	Glucometer
Berlin Extension	Bag Valve Mask
Berlin Extension	Digital Thermometer
Berlin Extension	Handheld Pulse Oximeter
Cancun Extension	Bag Valve Mask
Cancun Extension	Digital Thermometer
Cancun Extension	Handheld Pulse Oximeter

London Extension	Glucometer
London Extension	Bag Valve Mask
London Extension	Digital Thermometer
London Extension	Electronic Stethoscope
London Extension	Handheld Pulse Oximeter
New York Extension	Glucometer
New York Extension	Bag Valve Mask
New York Extension	Digital Thermometer
New York Extension	Electronic Stethoscope
New York Extension	Handheld Pulse Oximeter
Toronto Extension	Glucometer
Toronto Extension	Bag Valve Mask
Toronto Extension	Digital Thermometer
Toronto Extension	Electronic Stethoscope
Toronto Extension	Handheld Pulse Oximeter

Shipping_offering

Offering
Same Day
Overnight
Two Day

Offers

Store Location	Shipping Offering
Berlin Extension	Two Day
Cancun Extension	Two Day
London Extension	Same Day
London Extension	Overnight
London Extension	Two Day
New York Extension	Overnight
New York Extension	Two Day
Toronto Extension	Two Day

Alternate Names

Name	Product
Glucose Meter	Glucometer
Blood Glucose Meter	Glucometer
Glucose Monitoring System	Glucometer
Thermometer	Digital Thermometer

Ambu Bag	Bag Valve Mask
Oxygen Bag Valve Mask	Oxygen Bag Valve Mask
Cardiology Stethoscope	Electronic Stethoscope
Portable Pulse Oximeter	Handheld Pulse Oximeter
Handheld Pulse Oximeter System	Handheld Pulse Oximeter

The DDL and DML to create and populate the tables in the schema are listed below. You can copy and paste this into your SQL client to create and populate the tables.

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

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

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

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

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

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

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

CREATE TABLE Alternate_name (
alternate_name_id DECIMAL(12) NOT NULL PRIMARY KEY,
name VARCHAR(255) NOT NULL,
product_id DECIMAL(12) NOT NULL);

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

```sql
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(shipping_offering_id)
REFERENCES Shipping_offering(shipping_offering_id);

ALTER TABLE Alternate_name
ADD CONSTRAINT fk_name_to_product FOREIGN KEY(product_id)
REFERENCES Product(product_id);

INSERT INTO Currency(currency_id, currency_name, us_dollars_to_currency_ratio)
VALUES(1, 'Britsh Pound', 0.67);
INSERT INTO Currency(currency_id, currency_name, us_dollars_to_currency_ratio)
VALUES(2, 'Canadian Dollar', 1.34);
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', 0.92);
INSERT INTO Currency(currency_id, currency_name, us_dollars_to_currency_ratio)
VALUES(5, 'Mexican Peso', 16.76);

INSERT INTO Shipping_offering(shipping_offering_id, offering)
VALUES (50, 'Same Day');
INSERT INTO Shipping_offering(shipping_offering_id, offering)
VALUES (51, 'Overnight');
INSERT INTO Shipping_offering(shipping_offering_id, offering)
VALUES (52, 'Two Day');

--Glucometer
INSERT INTO Product(product_id, product_name, price_in_us_dollars)
VALUES(100, 'Glucometer', 50);
INSERT INTO Alternate_name(alternate_name_id, name, product_id)
VALUES(10000, 'Glucose Meter', 100);
INSERT INTO Alternate_name(alternate_name_id, name, product_id)
VALUES(10001, 'Blood Glucose Meter', 100);
INSERT INTO Alternate_name(alternate_name_id, name, product_id)
VALUES(10002, 'Glucose Monitoring System', 100);

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

--Digital Thermometer
INSERT INTO Product(product_id, product_name, price_in_us_dollars)
```

```sql
VALUES(102, 'Digital Thermometer', 250);
INSERT INTO Alternate_name(alternate_name_id, name, product_id)
VALUES(10005, 'Thermometer', 102);

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

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

--Berlin Extension
INSERT INTO Store_location(store_location_id, store_name, currency_accepted_id)
VALUES(10, 'Berlin Extension', 4);
INSERT INTO Sells(sells_id, store_location_id, product_id)
VALUES(1000, 10, 100);
INSERT INTO Sells(sells_id, store_location_id, product_id)
VALUES(1001, 10, 101);
INSERT INTO Sells(sells_id, store_location_id, product_id)
VALUES(1002, 10, 102);
INSERT INTO Sells(sells_id, store_location_id, product_id)
VALUES(1003, 10, 104);
INSERT INTO Offers(offers_id, store_location_id, shipping_offering_id)
VALUES(150, 10, 52);

--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, 104);
INSERT INTO Offers(offers_id, store_location_id, shipping_offering_id)
VALUES(151, 11, 52);

--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, shipping_offering_id)
```

```sql
VALUES(152, 12, 50);
INSERT INTO Offers(offers_id, store_location_id, shipping_offering_id)
VALUES(153, 12, 51);
INSERT INTO Offers(offers_id, store_location_id, shipping_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, shipping_offering_id)
VALUES(155, 13, 51);
INSERT INTO Offers(offers_id, store_location_id, shipping_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, shipping_offering_id)
VALUES(157, 14, 52);
```

As a reminder, for each step that requires SQL, make sure to capture a screenshot of the command and the results of its execution. *Further, make sure to eliminate unneeded columns from the result set, to name your columns something user-friendly and human readable, and to format any prices as currencies.*

Section Steps

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

Query Query History

```sql
shipping_offering_id DECIMAL(12) NOT NULL);

CREATE TABLE Alternate_name (
alternate_name_id DECIMAL(12) NOT NULL PRIMARY KEY,
name VARCHAR(255) NOT NULL,
product_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(shipping_offering_id)
REFERENCES Shipping_offering(shipping_offering_id);

ALTER TABLE Alternate_name
ADD CONSTRAINT fk_name_to_product FOREIGN KEY(product_id)
REFERENCES Product(product_id);
```

Data output Messages Notifications

```
ALTER TABLE

Query returned successfully in 140 msec.
```

- Insert data into tables

Query Query History

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

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

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

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

Data output Messages Notifications

```
INSERT 0 1

Query returned successfully in 106 msec.
```

Query Query History

```sql
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, shipping_offering_id)
VALUES(155, 13, 51);
INSERT INTO Offers(offers_id, store_location_id, shipping_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, shipping_offering_id)
VALUES(157, 14, 52);
```

Data output Messages Notifications

```
INSERT 0 1

Query returned successfully in 132 msec.
```

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

```sql
SELECT Store_name, Product_name,
       to_char(price_in_us_dollars * (SELECT us_dollars_to_currency_ratio
                                      FROM Currency
                                      JOIN Store_location ON store_location.currency_accepted_id = Currency.currency_id
                                      WHERE store_location.store_name = 'London Extension'), 'FM£999D00')
                                      AS Price_in_British_Pound
FROM Product
JOIN Sells ON Sells.product_id = Product.product_id
JOIN store_location ON Sells.store_location_id = store_location.store_location_id
WHERE Product.product_name = 'Digital Thermometer'
      AND store_location.store_name = 'London Extension';
```

Data output Messages Notifications

	store_name character varying (255)	product_name character varying (255)	price_in_british_pound text
1	London Extension	Digital Thermometer	£167.50

Briefly explain how your solution makes use of the uncorrelated subquery to help retrieve the result.

An uncorrelated subquery can be run independently. In this step query, I need to retrieve the price of product in local accepted currency which is Pound in this case (London). But the product table lists price in US dollar price. So, I use the subquery to get the US dollar to Pound ratio, When I run whole query, the subquery will run first and get the value of us_dollars_to_currency_ratio = 0.67 (US dollar to Pound ratio) (Running subquery demonstrated screenshot below), And then use the result (0,67) multiply to the price_in_us_dollars to get price in British pound. The uncorrelated subquery (getting the ratio) would not change if the price of the product changes, And we always get the right value when they change the currencies table data because is dynamic and not hardcoded.

```sql
SELECT Store_name, Product_name,
       to_char(price_in_us_dollars * (SELECT us_dollars_to_currency_ratio
                                      FROM Currency
                                      JOIN Store_location ON store_location.currency_accepted_id = Currency.currency_id
                                      WHERE store_location.store_name = 'London Extension'), 'FM£999D00')
                                      AS Price_in_British_Pound
FROM Product
JOIN Sells ON Sells.product_id = Product.product_id
JOIN store_location ON Sells.store_location_id = store_location.store_location_id
WHERE Product.product_name = 'Digital Thermometer'
      AND store_location.store_name = 'London Extension';
```

Data output Messages Notifications

	us_dollars_to_currency_ratio numeric (12,2)
1	0.67

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

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

```sql
SELECT product_name,
       to_char(price_in_us_dollars * (SELECT us_dollars_to_currency_ratio
                                      FROM Currency
                                      WHERE currency_name = 'Euro'), 'FM€9999D00')
                                      AS price_in_Euro
FROM Product
JOIN sells ON sells.product_id = Product.product_id
JOIN store_location ON store_location.store_location_id = sells.store_location_id
WHERE store_location.store_name = 'London Extension'
      AND price_in_us_dollars * (SELECT us_dollars_to_currency_ratio
                                 FROM Currency
                                 WHERE currency_name = 'Euro') < 26
      OR price_in_us_dollars * (SELECT us_dollars_to_currency_ratio
                                 FROM Currency
                                 WHERE currency_name = 'Euro') > 299
GROUP BY Product.product_name, price_in_Euro;
```

Data output | Messages | Notifications

	product_name character varying (255)	price_in_euro text
1	Electronic Stethoscope	€322.00
2	Handheld Pulse Oximeter	€414.00
3	Bag Valve Mask	€23.00

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

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

Data output | Messages | Notifications

	us_dollars_to_currency_ratio numeric (12,2)
1	0.92

Overall, the subquery retrieves the US dollar to Euro currency ratio from the Currency table Where currency name Euro (0.92) and covert product price from US dollar to Euro, filter the results. More detail below.

- The code line from 225-229, select the name of product and the price, The first subquery (from line 225-229), the price of product is covert from US dollar to Euro based on the US dollar price multiplies to the US Dollar to Euro Currency ratio which is retrieved from the subquery above.

- The second subquery (234 – 236) is also convert the price from US dollar to Euro but use to get limited rows by filters the price to less than 26.

- The third subquery (237 – 239) is also convert the price from US dollar to Euro but use to get limited rows by filters the price to greater than 299.

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

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

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

```sql
SELECT Product.product_name, Alternate_name.name AS alternate_name
FROM Product
JOIN Alternate_Name ON Alternate_Name.product_id = Product.product_id
WHERE Product.product_id IN (SELECT Product.product_id
                             FROM Store_location
                             JOIN Sells ON Store_location.Store_location_id = Sells.Store_location_id
                             JOIN Product ON Product.product_id = Sells.product_id
                             GROUP BY Product.product_id
                             HAVING Count(Product.product_id) = (SELECT COUNT(Store_location.store_location_id)
                                                                 FROM Store_location));
```

Data output Messages Notifications

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

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

There are 2 subqueries in the overall query.

The first one is screenshot below.

```sql
SELECT COUNT(Store_location.store_location_id)
FROM Store_location;
```

Data output Messages Notifications

	count bigint
1	5

This first subquery's role is retrieved number of Store location which is 5 in this case. It will be always giving the correct even if we add more or change stores in it.

The second subquery is screenshot below.

```sql
251 SELECT Product.product_id, Product_name
252 FROM Store_location
253 JOIN Sells ON Store_location.Store_location_id = Sells.Store_location_id
254 JOIN Product ON Product.product_id = Sells.product_id
255 GROUP BY Product.product_id
256 HAVING Count(Product.product_id) = (SELECT COUNT(Store_location.store_location_id)
257                                     FROM Store_location);
258
259
260
```

Data output Messages Notifications

	product_id [PK] numeric (12)	product_name character varying (255)
1	104	Handheld Pulse Oximeter
2	102	Digital Thermometer
3	101	Bag Valve Mask

With this subquery I add one more Product_name in it to get better visualization (I only SELECT Product_ID in the overall query).
The main role of this subquery is retrieved all products which available in all locations (which is 5 locations gets from the first subquery). To get the result, I join Store_location to Sells, then to Product to get all products and locations sell them.
And then I group all product by using Group By function for Product.product_id, I limit the result by using condition Having Count(Product.product_id) = 5 (From the first subquery), this also mean, only retrieve the products which available in all stores (5 stores in this case).
With the overall query, I applied condition Where Clause with the IN operator in line 242, In Operator uses to test whether the value is found in a list of values (product_id = 101, 102, 104), I can limit the result to the products which are available in all store location.

```sql
246 WHERE Product.product_id IN (SELECT Product.product_id
247                              FROM Store_location
248                              JOIN Sells ON Store_location.Store_location_id = Sells.Store_location_id
249                              JOIN Product ON Product.product_id = Sells.product_id
250                              GROUP BY Product.product_id
251                              HAVING Count(Product.product_id) = (SELECT COUNT(Store_location.store_location_id)
252                                                                  FROM Store_location));
253
```

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

```sql
SELECT Product_in_all_stores.Product_name, Alternate_name.name AS alternate_name
FROM (SELECT Product.product_id, Product.product_name
      FROM Store_location
      JOIN Sells ON Store_location.Store_location_id = Sells.Store_location_id
      JOIN Product ON Product.product_id = Sells.product_id
      GROUP BY Product.product_id
      HAVING Count(Product.product_id) = (SELECT COUNT(Store_location.store_location_id)
                                          FROM Store_location)) Product_in_all_stores
JOIN Alternate_Name ON Alternate_Name.product_id = Product_in_all_stores.product_id;
```

Data output Messages Notifications

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

Explain how you integrated the subquery into the FROM clause to derive the same results as step 4.

The subquery is screenshot below.

```sql
SELECT Product.product_id, Product.product_name
FROM Store_location
JOIN Sells ON Store_location.Store_location_id = Sells.Store_location_id
JOIN Product ON Product.product_id = Sells.product_id
GROUP BY Product.product_id
HAVING Count(Product.product_id) = (SELECT COUNT(Store_location.store_location_id)
                                    FROM Store_location)
```

Data output Messages Notifications

	product_id [PK] numeric (12)	product_name character varying (255)
1	104	Handheld Pulse Oximeter
2	102	Digital Thermometer
3	101	Bag Valve Mask

Notice that the WHERE clause subquery in the step 4 is moved to the FROM clause, to lines 252 - 258 in the overall query. The subquery retrieves the product_id and product_name which available in all store. The words ‘’Product_in_all_store’’ on line 258 is an alias which provides a name for the subquery’s results. Once defined, the alias can be used as if it were table, table has product_id and product_name columns and 3 rows. On line 259, the alias ‘’Product_in_all_store’’ is used as a part of the join condition, to join the results from the subquery into the Alternate_name table.

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

```
267 SELECT Product.product_name, Alternate_name.name AS alternate_name
268 FROM Product
269 JOIN Alternate_Name ON Alternate_Name.product_id = Product.product_id
270 WHERE EXISTS (SELECT Product_in_all_stores.product_id
271               FROM Product Product_in_all_stores
272               JOIN Sells ON Sells.product_id = Product_in_all_stores.product_id
273               JOIN Store_location ON Store_location.Store_location_id = Sells.Store_location_id
274               GROUP BY Product_in_all_stores.product_id
275               HAVING Count(Product_in_all_stores.product_id) = (SELECT COUNT(Store_location.store_location_id)
276                                                                 FROM Store_location)
277               AND Product_in_all_stores.product_id = Product.product_id);
278
279 -- Check
```

Data output   Messages   Notifications

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

Explain:

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

From the line 270 – 277. On line 271, I created alias “Product_in_all_stores” for Product table in the subquery, The main purpose for this alias is to eliminate any ambiguity between Product table in the outer query (on line 268) and Product table in subquery (line 271).

```
267 SELECT Product.product_name, Alternate_name.name AS alternate_name
268 FROM Product
269 JOIN Alternate_Name ON Alternate_Name.product_id = Product.product_id
270 WHERE EXISTS (SELECT Product_in_all_stores.product_id
271               FROM Product Product_in_all_stores
272               JOIN Sells ON Sells.product_id = Product_in_all_stores.product_id
273               JOIN Store_location ON Store_location.Store_location_id = Sells.Store_location_id
274               GROUP BY Product_in_all_stores.product_id
275               HAVING Count(Product_in_all_stores.product_id) = (SELECT COUNT(Store_location.store_location_id)
276                                                                 FROM Store_location)
277               AND Product_in_all_stores.product_id = Product.product_id);
278
```

On line 277, I added one more condition "AND Product_in_all_stores.product_id = Product.product_id" into Having Clause. It is this line correlates the subquery with the outer query. Notice that the product_id of Product_in_all_stores must equal with product_id of Product, and it is this equality that forces the subquery into correlation. We can simply say "Retrieve the Product found in current row of the outer query only if that Product is available in all stores". This logic, coupled with the EXIST keyword, means that if the current row in the outer query does not contain a product that available in all locations, it is excluded from the result set.

b. how and when the correlated subquery is executed in the context of the outer query.
Correlated subquery can't be executed on their own, correlated subqueries only make sense in the context of the outer query into which they are embedded, In this step I use "AND Product_in_all_stores.product_id = Product.product_id" to correlate subquery and outter query. Correlate subqueries are executed once for each row in the outer query and therefore retrieve one result set for each row in the outer query.

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

```sql
CREATE OR REPLACE VIEW Product_in_all_stores AS
SELECT Product.product_id, Product.product_name
       FROM Store_location
       JOIN Sells ON Store_location.Store_location_id = Sells.Store_location_id
       JOIN Product ON Product.product_id = Sells.product_id
       GROUP BY Product.product_id
       HAVING Count(Product.product_id) = (SELECT COUNT(Store_location.store_location_id)
                                           FROM Store_location);
-- Using View in Query
SELECT Product_in_all_stores.Product_name, Alternate_name.name AS alternate_name
FROM Product_in_all_stores
JOIN Alternate_Name ON Alternate_Name.product_id = Product_in_all_stores.product_id;
```

Data output   Messages   Notifications

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

Section Two – Concurrency

Section Background

Modern information systems run transactions in parallel. Running hundreds or even thousands of transactions at the same time is commonplace for information systems today. Transactions running at the same run into many issues, including lost updates, uncommitted dependencies, inconsistent analysis, and others. To eliminate and manage these issues, modern relational databases use a scheduler which controls the schedule and timing of transaction execution, in addition to other mechanisms.

You have a chance to demonstrate understanding of concurrency control in this section.

In this section, the questions refer to the following data table, as well the following transactions and steps.

Data Table
1
2
3
4
5

Transaction 1
Read the value from row 4.
Multiply that value times 3.
Write the result to row 3.
Write the literal value “8” to row 2.
Write the literal value “20” to row 5.
Commit.

Transaction 2
Read the value from row 2.
Write that value to row 4.
Write the literal value “15” to row 3.
Commit.

Section Steps

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

Inconsistent Analysis		
T1	Read the value from row 4.	Transaction 1 has read value row 4 as "4"
T2	Read the value from row 2.	Transaction 2 has read value row 2 as "2"
T1	Multiply that value times 3.	Transaction 1: "4" multiply 3 = “12”
T2	Write that value to row 4.	Transaction 2 writes "2" to row 4: Row 4 becomes "2"
T2	Write the literal value “15” to row 3.	Transaction 2 writes "15" to row 3: Row 3 becomes "15"
T1	Write the result to row 3.	Transaction 1 writes result as 12 to row 3: Row 3 becomes "12"
T2	Commit.	Transaction 2's changes made permanent in the database
T1	Write the literal value “8” to row 2.	Transaction 1 writes "8" to row 2. Row 2 becomes "8"
T1	Write the literal value “20” to row 5.	Transaction 1 writes "20" to row 5. Row 5 becomes "20"
T1	Commit.	Transaction 1's changes made permanent in the database

The contents of table after the transaction complete using the schedule

Data Table
1
8
12
2
20

9. *Issues with Locking and Multiversioning* – Imagine the database has both locking and multiversioning in place for concurrency control.
    a. Starting with the same schedule in the prior step, show and explain step-by-step how the use of locking and multiversioning modifies the schedule. Also list out the contents of the table after the transactions complete using the new schedule. Make sure to explain specifically whether and how locking and multiversioning modifies the schedule and affects the final resulting table.

When Multiversioning is used, shared locks are no longer necessary. Because it is an advanced concurrency control technique whereby the database stores a history of each value rather than only the current values.

Locking and Multiversioning			
T1	Read the value from row 4.	Transaction 1 has read value row 4 as "4"	
T2	Read the value from row 2.	Transaction 2 has read value row 2 as "2"	
T1	Multiply that value times 3.	Transaction 1: "4" multiply 3 = 12	
T2	Write that value to row 4.	Transaction 2 writes "2" to row 4: Row 4 becomes "2"	T2 Locks row 4
T2	Write the literal value “15” to row 3.	Transaction 2 writes "15" to row 3: Row 3 becomes "15"	T2 Locks Lock row 3
T1	Write the result to row 3.	Transaction 1 writes result as 12 to row 3: Row 3 becomes "12"	T1 Needs to wait because row 3 is already locked by T2
T2	Commit.	Transaction 2's changes made permanent in the database	T2 commit and release lock on row 4 and row 3
T1	Write the literal value “8” to row 2.	Transaction 1 writes "8" to row 2. Row 2 becomes "8"	
T1	Write the literal value “20” to row 5.	Transaction 1 writes "20" to row 5. Row 5 becomes "20"	
T1	Commit.	Transaction 1's changes made permanent in the database	T1 needs to restart the transaction again.

Content of table after the transactions complete.

Data Table
1
2
15
2
5

b. Could a schedule of these transactions result in a deadlock? If not, explain why. If so, show a step-by-step schedule that results in a deadlock.

I am using Postgres, it always combines multiversioning with locking. With multiversioning, only updating or deleting the same row in a different order can cause deadlocks between concurrent transactions. Based on the transaction above, a deadlock is not occurring since there is no updating and deleting the same row at the same time during the concurrent transactions. With multiversioning, share locks are not no longer necessary, so read row does not cause deadlock. Two transactions try to update same row number 3 in the above schedule, but transaction 2 updates first and already locked it, transaction 1 needs to wait for transaction 2 to commit to release the lock. And also after update row 3 there is no row got locked after updating row in Transaction 2. So, there is no deadlock in this schedule

Evaluation

Your lab will be reviewed by your facilitator or instructor with the criteria outlined in the table below. Note that the grading process:

- involves the grader assigning an appropriate letter grade to each criterion.
- uses the following letter-to-number grade mapping – A+=100,A=96,A-=92,B+=88,B=85,B-=82,C+=88,C=85,C-=82,D=67,F=0.
- provides an overall grade for the submission based upon the grade and weight assigned to each criterion.
- allows the grader to apply additional deductions or adjustments as appropriate for the submission.
- applies equally to every student in the course.

5 points per day will be subtracted for late submissions. Submissions beyond 5 days late will not be accepted. Please contact your facilitator for any exceptions.

Criterion	A	B	C	D	F
Section 1: Results (20%)	The results for all steps in Section 1 are complete and accurate. No extra rows or columns are present, and no rows or columns are missing.	The results for most steps in Section 1 are complete and accurate. Few extra rows or columns are present, and few rows or columns are missing.	The results for some steps in Section 1 are accurate. Some extra rows or columns may be present, and some rows or columns may be missing.	The results for most steps in Section 1 are incomplete and inaccurate. Extra rows or columns may be present, and extra rows or columns may be missing.	The results for all of section 1 may be missing. All steps' results are incomplete and inaccurate.
Section 1: Construction (30%)	Appropriate subqueries have been defined and integrated well for all steps. For all steps possible, dynamic lookups have been used in lieu of hardcoded values.	Appropriate subqueries have been defined and integrated well for most steps. For most steps possible, dynamic lookups have been used in lieu of hardcoded values.	Appropriate subqueries have been defined and integrated well for some steps. For some steps possible, dynamic lookups have been used in lieu of hardcoded values.	Inappropriate subqueries have been defined or integrated poorly for most steps. Hardcoded values have been used for most steps, instead of dynamic lookups.	Answers for all of section 1 may be missing. Inappropriate subqueries have been defined or integrated poorly for all steps. Hardcoded values have been used for all steps, instead of dynamic lookups.
Section 2: #8 Quality (15%)	The explanation, schedule, and resulting table contents for #8 are entirely accurate, thorough, and clear.	The explanation, schedule, and resulting table contents for #8 are mostly accurate, thorough, and clear.	The explanation, schedule, and resulting table contents for 8 are somewhat accurate and clear.	The explanation, schedule, and resulting table contents for #8 are mostly inaccurate and unclear.	The answers for #8 could be missing. The explanation, schedule, and resulting table contents for #8 are entirely inaccurate and unclear.

Section 2: #9 Quality (15%)	The explanations, schedules, and resulting table contents for 9a and 9b are entirely accurate, thorough, and clear.	The explanations, schedules, and resulting table contents for 9a and 9b are mostly accurate, thorough, and clear.	The explanations, schedules, and resulting table contents for 9a and 9b are somewhat accurate and clear.	The explanations, schedules, and resulting table contents for 9a and 9b are mostly inaccurate and unclear.	The answer for #9 could be missing. The explanations, schedules, and resulting table contents for 9a and 9b are entirely inaccurate and unclear.
Overall Presentation (20%)	The explanations supporting all answers are excellent. Queries or other SQL commands have been executed to demonstrate correctness for all answers that need them.	The explanations supporting the answers are good. Queries or other SQL commands have been executed to demonstrate correctness for most answers that need them.	The explanations supporting the answers are satisfactory. Queries or other SQL commands have been executed to demonstrate correctness for some answers that need them.	The explanations supporting the answers are minimal. Queries or other SQL commands have not been executed to demonstrate correctness for most answers that need them.	Explanations for all answers are missing. Queries or other SQL commands have not been executed to demonstrate correctness.

Use the **Ask the Teaching Team Discussion Forum** if you have any questions regarding how to approach this lab. Make sure to include your name in the filename and submit it in the *Assignments* section of the course.