# CECS 323 LAB VIEWS FROM SUB QUERIES

**OBJECTIVE:** Have a little fun with converting from a query using inline queries to one using a view.

**INTRODUCTION:** Generally speaking, the longer a SQL query gets, the harder it is to maintain, and the less likely it is that needed changes will get made until the become unavoidable. One technique that allows us to modularize our SQL is to create views that do part of the work of a query, and then select from those views. Once such views are created, they can be tested out individually, which makes testing of the overall query more incremental. Obviously, views can reference other views to create a layers of views. Some years ago, the conventional wisdom was to avoid creating views built on views because the database optimizer would not do a good job with layers of views. But the database management systems have gotten more sophisticated, and it is no longer a performance problem to build views on top of views.

**An Example:**

Note that this example is covered in my video that you can find at BeachBoard | Content | Lab Videos | Creating Views From Subqueries | Creating Views From Subqueries. The following is offered for those of you who prefer to see this in print.

Find customers that share the same state and country. The country must be one of the following: UK, Australia, Italy, or Canada. Remember that not all countries have states at all, so you need to substitute a character sting like 'N/A' for the state in those cases so that you can compare the states. Returns 15 rows. One solution to this looks like:

```
SELECT  cust1.customername as name1,
        cust2.customername as name2,
        cust1.stateNull as "State",
        cust1.country
FROM    (SELECT customerNumber, customerName,
                coalesce(state, 'N/A') stateNull, country
        FROM    customers) cust1 inner join
        (SELECT customerNumber, customerName,
                coalesce(state, 'N/A') stateNull, country
        FROM    customers) cust2 using (stateNull, country)
WHERE   cust1.CUSTOMERNUMBER < cust2.CUSTOMERNUMBER and
        cust1.country IN ('UK',    'Australia',
                          'Italy', 'Canada')
ORDER BY name1, name2;
```

Notice that there are two identical subqueries here. If we define a view, call it custNA (short for customers with N/A for a null state) like this:

```
CREATE  VIEW custNA as
SELECT  customerNumber, customerName,
        coalesce(state, 'N/A') stateNull, country
FROM    customers;
```

Now, refactor the original query to use custNA twice:

```
SELECT  cust1.customername as name1,
        cust2.customername as name2,
        cust1.stateNull as "State",
        cust1.country
FROM    custNA cust1 inner join
        custNA cust2 using (stateNull, country)
WHERE   cust1.CUSTOMERNUMBER < cust2.CUSTOMERNUMBER and
        cust1.country IN ('UK',    'Australia',
            'Italy', 'Canada')
ORDER BY name1, name2;
```

We see that the query is visibly shorter. We also see more clearly the parallelism between the two uses of the custNA view in the from clause.

Then, to see whether the new query produces the same exact output as the old one, I am going to create two new views: one that implements the original solution, and the second which implements the new, refactored solution.

```
CREATE VIEW old as
SELECT  cust1.customername as name1,
        cust2.customername as name2,
        cust1.stateNull as "State",
        cust1.country
FROM    (SELECT customerNumber, customerName,
                coalesce(state, 'N/A') stateNull, country
        FROM    customers) cust1 INNER JOIN
        (SELECT customerNumber, customerName,
                coalesce(state, 'N/A') stateNull, country
        FROM    customers) cust2 USING (stateNull, country)
WHERE   cust1.CUSTOMERNUMBER < cust2.CUSTOMERNUMBER and
        cust1.country IN ('UK',    'Australia',
                        'Italy', 'Canada')
ORDER BY name1, name2;
```

```
CREATE VIEW new as
SELECT  cust1.customername as name1,
        cust2.customername as name2,
        cust1.stateNull as "State",
        cust1.country
FROM    custNA cust1 inner join
        custNA cust2 using (stateNull, country)
WHERE   cust1.CUSTOMERNUMBER < cust2.CUSTOMERNUMBER and
        cust1.country IN ('UK',    'Australia',
            'Italy', 'Canada')
ORDER BY name1, name2;
```

Now, we can compactly run a test to make sure that there are no results coming out of the old solution that are not in the new solution, and vice versa:

```
SELECT * FROM new
EXCEPT
SELECT * FROM old
UNION
SELECT * from old
EXCEPT
SELECT * FROM new;
```

Without the union and the second half of this query, we might have some rows coming out from the old solution that are not in the new solution. Of course, if both the old and the new version of the solution return the same number of rows, then you technically do not need the second have of this check query.

Recursive queries often lend themselves to refactoring one or more of the subqueries into a view. Of course, you often have several ways of tackling the original problem, so you might not initially arrive at a solution that suggests creating a view to replace one or more subqueries. In the two problems below,

1. You will get some hints about how to structure your original solution.
2. You will come up with a solution to the stated problem.
3. You will identify where in your solution you might use views to make the solution easier to follow.
4. You will write the code to declare the view(s).
5. You will write a new version of your original solution that uses that/those views.
6. You will check that the new solution produces the same results as the original one.
    a. Remember that you can use the minus & the union:

```
select … (using the original query)
except … (using your refactored query)
union
select … (using your refactored query)
except … (using the original query)
```

    b. If the result is **zero** rows, you can be confident that the two queries produced the same results. Not 100% iron-clad guarantee, but it is pretty good.

**PROCEDURE:**

**Query #1**

Marketing wants to find out which products get ordered together, particularly in **threes**. Report out the sets of three products that get ordered together in the same customer order and the total number of orders that show all three of those products. Be sure that the same three products do not show up more than once. For instance, there should not be one row in your output with product A,

B, and C and then another row with the products B, C, and A. Order the output by the number of orders that the three products occur together, in descending order. Only show those combination of products that appear in more than 20 orders. Returns 56 rows.

There are (as always) several ways to approach this. For this exercise, please do this by joining three subqueries together. Each of the inline queries will just require a join between orderDetails and Products

**Query #2**

Find each pair of customers and the products that they have **both** ordered for all pairs of customers who have ordered the same product during the **first two months of 2015**. Do not repeat customer pairs. You will want to create two subqueries that join customers to orders to order details to products. Order by the first customer name, then the second customer name, then the name of the product that they have ordered in common. Returns 84 rows.

**WHAT TO TURN IN:**

- For each of the problems above:
  - The original query
  - The view(s) that you have created.
  - The refactored version of the query, that uses the view.
  - The output from running the original query.
  - The output from running the refactored query.
  - The code that you write to test that the output from the refactored query is identical to the original query that you wrote.
- Your team's filled out collaboration document. You can find a template for that here.

**PARTING THOUGHTS:**

- A smart developer will structure their views to be reusable. The good news there is that those views eventually form a library of views that the team can utilize with confidence. The bad news is that changes to those views or the underlying tables can be disruptive.
- Most database management systems provide a data dictionary table that has a row for each dependency between two objects of the database. That dependencies table can then be used to report all the objects in the database which will be impacted if a given view is modified. Unfortunately, I have not been able to find that table in Derby, but I will keep looking.