Chapter 3

Subqueries, Distinct & Case

Contents

1 Query Evaluation Order: SELECT and WHERE

- Assume that we wanted to look at the registrations per dollar of annualfee for 4 Ton Truck Tractors in Scott county. We could start with the following query:

```
select
    year, registrations, annualfee
from
    cls.cars
where
    countyname = 'Scott'
    and tonnage = '4 Tons'
    and vehicletype = 'Truck Tractor';


  year    registrations    annualfee
------  ---------------  -----------
  2010                1            0
  2009                1            0
  2007                1            5
  2008                2           85
```

- To determine the registrations per annual fee, we could change the select statement to the following:

```
select
    year, registrations::float/ annualfee as ratio
from
    cls.cars
where
    countyname = 'Scott'
    and tonnage = '4 Tons'
    and vehicletype = 'Truck Tractor';
```

  which yields an error:

```
ERROR:  division by zero
```

  Unsurprisingly, rows with an annualfee equal to zero are causing this query to fail.

- To handle this we can remove those rows that cause this query to fail:

```
select
    year, registrations::float/ annualfee as ratio
from
    cls.cars
where
    annualfee > 0
    and countyname = 'Scott'
    and tonnage = '4 Tons'
    and vehicletype = 'Truck Tractor';


  year      ratio
------  ---------
  2007  0.2
  2008  0.0235294
```

which will return only the two rows where the division by zero is not an issue. Notice about this query is that the `WHERE` clause is evaluated *before* the `SELECT` statement is evaluated.

- This allows the user to exclude observations that may generate problems *before* the `SELECT` statement operates on them.
- An implication of this is that since `SELECT` is done *after* `WHERE`, things defined in the `SELECT` are *not* available in the `WHERE`:

```
SELECT
    year, annualfee::float / registrations as avg_fee
from
    cls.cars
where avg_fee > 0;
```

```
ERROR:  column "avg_fee" does not exist
```

Why did this happen? It happened because the column `avg_fee` isn't defined at the time that the `WHERE` clause is executed.

- The same logic applies to the `FROM` clause, which is evaluated *first*. Consider the following query, which renames our table into something else.

```
select
        renamed_table.*
from
        cls.cars as renamed_table
limit 100;


  year  countyname    motorvehicle    vehiclecat    vehicletype     tonnage       [...]
------  ------------  --------------  ------------  --------------  ---------  -- [...]
  2008  Ida           Yes             Bus           Bus                        [...]
  2011  Jasper        Yes             Moped         Moped                      [...]
  2012  Harrison      Yes             Truck         Truck           3 Tons     [...]
  2015  Palo Alto     No              Trailer       Travel Trailer             [...]
  2016  Adair         Yes             Truck         Truck           3 Tons     [...]
[...]
```

In this query the table has been renamed in the FROM clause and that naming is passed through to the SELECT statement. If we were to instead try to reference cls.cars in the SELECT after the renaming, an error will occur:

```
select
        cars.*
from
        cls.cars as renamed_table
limit 100;
```

```
ERROR:  missing FROM-clause entry for table "cars"
LINE 2:  cars.*
```

Once again this confirms that the FROM clause is evaluated *before* SELECT.

2 Comparisons: BETWEEN, LIKE and ILIKE

- Another common comparison operator is BETWEEN:

```
select
    *
from
    cls.cars
where
    registrations between 2050 and 2100;
```

```
  year  countyname    motorvehicle    vehiclecat     vehicletype            tonna [...]
------  ------------  --------------  -------------  ---------------------  ----- [...]
  2011  Monroe        Yes             Multi-purpose  Multi-purpose                [...]
  2010  Shelby        No              Trailer        Small Regular Trailer        [...]
  2015  Ida           Yes             Truck          Truck                  3 Ton [...]
  2013  Dubuque       Yes             Truck          Truck                  6+ To [...]
  2010  Woodbury      No              Trailer        Semi Trailer                 [...]
[...]
```

will return all columns from the table cars where registrations are between 2050 and 2100. Note that this is equivalent to:

```
select
    *
from
    cls.cars
where
    registrations >= 2050 and 2100 >= registrations;
```

```
  year  countyname    motorvehicle    vehiclecat     vehicletype            tonna [...]
------  ------------  --------------  -------------  ---------------------  ----- [...]
  2011  Monroe        Yes             Multi-purpose  Multi-purpose                [...]
  2010  Shelby        No              Trailer        Small Regular Trailer        [...]
  2015  Ida           Yes             Truck          Truck                  3 Ton [...]
  2013  Dubuque       Yes             Truck          Truck                  6+ To [...]
  2010  Woodbury      No              Trailer        Semi Trailer                 [...]
[...]
```

In other words, BETWEEN is inclusive as it includes both end points.

- BETWEEN can also be used with strings, but be careful when doing so. In our cars database, for example, there is a single county that begins with the letter 'R' ("Ringgold"). If you run the following query:

```
select
    *
from
    cls.cars
where
    countyname between 'R' and 'R';


year    countyname    motorvehicle    vehiclecat    vehicletype    tonnage    reg [...]
------  ------------  --------------  ------------  -------------  ---------  --- [...]
```

will return zero rows! BETWEEN is computed using alphabetical order and, since "R" is before "Ringgold", alphabetically, this means that it won't be returned by this query. Instead, the following query will return all rows with a countyname which begins with the letter 'R':

```
select
    *
from
    cls.cars
where
    countyname between 'R' and 'S';


  year  countyname    motorvehicle    vehiclecat    vehicletype     tonnage       [...]
------  ------------  --------------  ------------  --------------  ------------- [...]
  2011  Ringgold      Yes             Bus           Bus                           [...]
  2014  Ringgold      Yes             Truck         Truck           6+ Tons Non-S [...]
  2016  Ringgold      Yes             Moped         Moped                         [...]
  2011  Ringgold      Yes             Motorcycle    Motorcycle                    [...]
  2005  Ringgold      Yes             Motor Home    Motor Home - B                [...]
[...]
```

- Second note: alphabetical order PostgreSQL is case insensitive. If you sort the following data:

```
A B D E c f g h
```

the result will be:

```
A B c D E f g h
```

- To further match strings we can use LIKE and ILIKE which searches for specified patterns within a string. Using LIKE without any special characters yields a simple equality comparison:

```
where countyname like 'Ringgold'
```

is equivalent to:

```
where countyname = 'Ringgold'
```

- ILIKE on the other hand is a case insensitive matching. In other words, the following where clauses

will return all rows from Ringgold county:

```
where countyname ilike 'ringgold'

where countyname ilike 'RINGgold'
```

- Both like and ilike allow for more complex pattern matching using percent sign ("%") and underscore ("_"). The percent sign is used to match any string while the underscore matches a single character. We call these types of characters "wildcards" and they allow users to create more complex matching criteria. Continuing with the example of the county of "Ringgold":

Clause	Will Match Ringgold?
like '%inggold'	Yes
like 'ring_old'	No
ilike 'ring_old'	Yes
like 'r%'	No
ilike 'r%'	Yes
ilike '%ringgold%'	Yes

- Remember that Null presents as False, even with wildcard characters. If there was a column in a table called "alwaysNull" which was Null in every row, the following:

```
where alwaysNull ilike '%'
```

would return zero rows.

- One difference between % and _ is that underscore *requires* a character to be there. For example, the string '_Ringgold' will **not** match Ringgold while '%Ringgold' will match.
- **Performance considerations:** Be mindful when using `LIKE` and `ILIKE` as they are expensive for the database to evaluate. When evaluating these expressions, the database moves from the first to last character within each string attempting to determine if each row matches the criteria. Whenever possible, minimize the use of wildcard characters.

3 CASE: Conditional Logic

- We have covered how to use a `SELECT` statement to manipulate columns. For example, we can easily add numbers together or transform a string. An extension of this is to change columns conditionally. To do this we use the `CASE` statement, which allows us to conditionally transform what the database returns.
- In the Iowa cars data we may be interested in doing analysis comparing those rows with more than 100 registrations against those with less than 100 registrations. As an example, consider the following query:

```
SELECT
    CASE
        WHEN registrations > 100 THEN 'BIG'
        ELSE 'SMALL'
    END as regSize
    , *
from
    cls.cars;


regsize      year  countyname    motorvehicle    vehiclecat    vehicletype     to [...]
---------  ------  ------------  --------------  ------------  --------------  -- [...]
SMALL        2008  Ida           Yes             Bus           Bus                [...]
BIG          2011  Jasper        Yes             Moped         Moped              [...]
BIG          2012  Harrison      Yes             Truck         Truck           3  [...]
BIG          2015  Palo Alto     No              Trailer       Travel Trailer     [...]
BIG          2016  Adair         Yes             Truck         Truck           3  [...]
[...]
```

This query will return all the columns in the database and one more column, with the name "regSize" that takes the value of "BIG" or "SMALL" depending on if the number of registrations is greater than 100.

- In the case of a Null value for registration it would fail the initial conditional and then be caught by the
- The `ELSE` clause is optional. The query below provides an example without an `ELSE` clause:

```
select
    case
        WHEN registrations > 100 then 'BIG'
    END as regsize
from
    cls.cars;


regsize
---------

BIG
BIG
BIG
BIG
[...]
```

In this case, the column regsize will have the value 'BIG' for registrations greater than 100. For values of registration less than 100, the value in the column will be Null.

- The `CASE` statement is evaluated row-by-row.
- We can add additional criteria by using multiple `WHEN` arguments. For example, we may want to do analysis on four different size criteria as can be seen in this query:

```
SELECT
    CASE
        WHEN registrations > 1000 THEN 'VERY VERY BIG'
        WHEN registrations > 500 THEN 'VERY BIG'
        WHEN registrations > 100 THEN 'BIG'
        ELSE 'SMALL'
    END as regSize
    , *
from
     cls.cars;


regsize          year  countyname    motorvehicle    vehiclecat    vehicletype    [...]
-------------  ------  ------------  --------------  ------------  -------------- [...]
SMALL            2008  Ida           Yes             Bus           Bus            [...]
BIG              2011  Jasper        Yes             Moped         Moped          [...]
VERY VERY BIG    2012  Harrison      Yes             Truck         Truck          [...]
BIG              2015  Palo Alto     No              Trailer       Travel Trailer [...]
VERY VERY BIG    2016  Adair         Yes             Truck         Truck          [...]
[...]
```

We only needed to include ">" signs because each of our inequalities excludes the previous. In other words, when the database evaluates the above it checks the `WHEN` statements in order: it first checks to determine if the number of registrations is greater than 1000, then if it is greater than 500, then if it is greater than 100 and finally, only if all 3 of those criteria fail, will it assign the value of "SMALL".

If the query was written this way:

```
SELECT
    CASE
        WHEN registrations > 500 THEN 'VERY BIG'
        WHEN registrations > 1000 THEN 'VERY VERY BIG'
        WHEN registrations > 100 THEN 'BIG'
    ELSE 'SMALL'
    END as regSize
    , *
from
    cls.cars;


regsize       year  countyname    motorvehicle    vehiclecat    vehicletype     to [...]
---------  ------  ------------  --------------  ------------  --------------  -- [...]
SMALL         2008  Ida           Yes             Bus           Bus                [...]
BIG           2011  Jasper        Yes             Moped         Moped              [...]
VERY BIG      2012  Harrison      Yes             Truck         Truck           3  [...]
BIG           2015  Palo Alto     No              Trailer       Travel Trailer     [...]
VERY BIG      2016  Adair         Yes             Truck         Truck           3  [...]
[...]
```

then *zero* observations would be classified as "VERY VERY BIG" since every row with registrations greater than 1000 are also greater than 500.

- When using case statements we can use any statement that we would use in a `WHERE` clause, including using `AND` and `OR` to create more complex Boolean statements:

```
select
    case
        when registrations > 500 and annualfee > 500 THEN 'Type 1'
        when registrations >= 500 and annualfee < 499 THEN 'Type 2'
        when registrations < 500 and annualfee > 500 THEN 'Type 3'
        when registrations >= 500 and annualfee < 499 THEN 'Type 4'
    else
        'hasNulls'
    END as regSize
    , *
from
    cls.cars
limit 1000;
```

```
regsize      year  countyname    motorvehicle    vehiclecat    vehicletype     to [...]
---------  ------  ------------  --------------  ------------  --------------  -- [...]
Type 3       2008  Ida           Yes             Bus           Bus                [...]
Type 3       2011  Jasper        Yes             Moped         Moped              [...]
Type 1       2012  Harrison      Yes             Truck         Truck           3  [...]
Type 3       2015  Palo Alto     No              Trailer       Travel Trailer     [...]
Type 1       2016  Adair         Yes             Truck         Truck           3  [...]
[...]
```

In the query above if there is a Null `registration` or `annualfee` then that row will fail the Boolean clauses on part of the `CASE` statement, resulting in those rows being caught in the `ELSE` condition.

- Note that you can use a `CASE` statement in a `WHERE` clause, though it uncommon to do so. What does the following do?

```
select * from cls.cars
where
    case
        when registrations < 100 then 1
        when registrations between 200 and 300 then 2
        when registrations > 500 then 3 end = 2;
```

```
  year  countyname    motorvehicle    vehiclecat    vehicletype      tonnage      [...]
------  ------------  --------------  ------------  ---------------  ---------  - [...]
  2016  Van Buren     Yes             Truck         Truck            4 Tons       [...]
  2009  Lucas         Yes             Truck         Truck            4 Tons       [...]
  2015  Keokuk        Yes             Truck         Truck            4 Tons       [...]
  2008  Decatur       No              Trailer       Regular Trailer               [...]
  2009  Lee           Yes             Truck         Truck            5 Tons       [...]
[...]
```

- There is a second syntax for the `CASE` statement, which is not used as frequently. This second syntax can only handle equality constraints against a single column. An example of this syntax can be shown below where we use it to create a new columns which adjusts the annual fee paid by inflation.

```
select
    case year
        WHEN 2005 THEN annualfee * 1.053
        WHEN 2006 THEN annualfee * 1.051
        WHEN 2007 THEN annualfee * 1.05
        WHEN 2008 THEN annualfee * 1.04
        WHEN 2009 THEN annualfee * 1.038
        WHEN 2010 THEN annualfee * 1.035
        WHEN 2011 THEN annualfee * 1.03
        WHEN 2012 THEN annualfee * 1.01
        WHEN 2013 THEN annualfee
    end as annualfeeInflation
from cls.cars;


  annualfeeinflation
--------------------
               707.2
              1427.58
            312951

[...]
```

When using this syntax we first specify which column we are going to compare on (in this case that column is year). For each row the year column is compared against the value after `WHEN` and, if that conditional is true, the `THEN` clause is evaluated.

- A useful application of the `CASE` statement is dealing with divide by zero. Previously we had dealt with division by zero problems by removing those rows using a `WHERE` clause. If, instead of removing that row, we wish to keep it but return a different value we can use `CASE`:

```
select
    case
        when annualfee > 0 then registrations / annualfee
        else null
    end as regPerDollar
from
    cls.cars;


  regperdollar
--------------
    0.00735294
    0.142857
    0.0162013
    0.0198773
    0.0187523
[...]
```

The `annualfee` values which are either Null or equal to zero will be caught by the case statement

and, rather than returning an error, the database will return a Null.

- We can use the `CASE` statement to implement the `LEAST` and `GREATEST` operator on two columns, but will need to be careful about nulls. Consider the following example:

```
select
    case when X >= Y then X else Y end as larg
from
    tablename;
```

In this case, if X is Null, then Y is returned. However, if Y is Null *the Null is returned*, which is NOT what we want. In to implement `GREATEST` (or `LEAST`) via a `CASE` statement we have to verify that the variable is not Null, as the query below demonstrates:

```
select
    case when Y is null or X >= Y then X else Y end as lrg
from
    tablename;
```

In this case if Y is Null then X is returned, no matter the value in X while if X is Null then Y is returned, no matter Y's value.

4 The DISTINCT Operator

- The `DISTINCT` operator can be used in a number of ways in SQL. The first way that we will describe is how it can be used is to remove duplicates from the data that is being returned.
- If we want to know what years are in the Iowa cars table we can run the following command:

```
SELECT DISTINCT year from cls.cars;


  year
------
  2013
  2021
  2015
  2008
  2010
[...]
```

which is a list of every distinct year in the table. We can combine this with the order by command to see an ordered list of the years in the database:

```
SELECT DISTINCT
     year
FROM
    cls.cars
ORDER BY year;


  year
------
  2005
  2006
  2007
  2008
  2009
[...]
```

- When learning SQL, it helps to think of `SELECT` and `SELECT DISTINCT` as two different functions. `DISTINCT` is not modifying a column, it is more fundamentally changing what is returned.
- `DISTINCT` is computationally expensive. Novice query writers often make the mistake of putting it in queries when it is not required and causing the queries to be slower than necessary.
- Let us use the following dataset to understand how Nulls and multiple columns are handled. The table "BillPaid" contains information from a credit card company. In particular, it contains information about if a person paid their bill at the end of each month. The column `paytype` represents how the Person paid their bill and is Null if a person did not pay. If a person didn't pay, the amount is zero to zero.

PersonID	Month	Paid	paytype	Amt
1	1	1	Visa	15
1	2	1	Visa	100
1	3	1	Visa	15
2	1	1	Visa	25
2	2	0	NULL	0
2	3	1	Visa	25
3	1	1	Check	10
3	2	0	NULL	0
3	3	0	NULL	0

Figure 3.1: "BillPaid" Table

- As before, we can use `DISTINCT` on a single column:

```
select distinct PersonID from cls.BillPaid;


  personid
----------
         3
         2
         1
```

as well as on multiple columns:

```
select distinct PersonID, PayType from cls.BillPaid;


  personid  paytype
----------  ---------
         3
         3  Check
         2
         2  Visa
         1  Visa
```

Note that this command does not create any data – only takes the unique entries by row. Also demonstrated is that Null is handled as if it was its own, unique, value.

- A common error with `DISTINCT` is trying to sort on a column which is not in the `SELECT`. Consider the following query:

```
select distinct PersonID from cls.BillPaid order by amt desc;
```

Looking at the table, we can see that PersonID #1 has a value equal to 100, which is larger than any other value – so should it go first? At the same time, PersonID #2 has a value of 25, which is larger than PersonID #1 in months 1 and 3, so should it be first? Since the database is not sure which to do, it does something different: it responds with an error.

```
ERROR:  for SELECT DISTINCT, ORDER BY expressions must appear in select list
```

- Importantly, `DISTINCT` and `ORDER BY` *can* be used at the same time, but only if the column being sorted is the same one as the column being made distinct, as can be seen in the query below.

```
SELECT distinct amt from cls.BillPaid order by amt desc;


  amt
-----
  100
   25
   15
   10
    0
```

5 Subqueries (IN, ANY, ALL)

- Up to this point, we have used `SELECT` and simple `WHERE` clauses to choose which rows and columns to return in a query. Simple `WHERE` clauses allow us to choose rows based on other data within that row, but not on information outside that row. In this section we will write subqueries to filter rows based on data not present in that row. We will continue to use Table 3.1, the "Bill Paid" table.

  Looking over this table, you can see that there are three people who had bills. To write a query which identifies missing payments we could write the following query:

```
select
    *
from
    cls.BillPaid
where
    Paid = 0;


  personid    month    paid  paytype      amt
----------  -------  ------  ---------  -----
         2        2       0                 0
         3        2       0                 0
         3        3       0                 0
```

  Which will return three rows, two from person #3 and one from #2.

- Assume we want to analyze *all the rows* from people who have ever missed a payment. The `WHERE` clause above will not work in this scenario since we need to know information about rows outside the one being evaluated. In this case we use the `IN` clause and a subquery:

```
select
    *
from
    cls.BillPaid
where
    personid IN (select personid from cls.BillPaid where paid = 0);


  personid    month    paid  paytype      amt
----------  -------  ------  ---------  -----
         2        1       1  Visa          25
         2        2       0                 0
         2        3       1  Visa          25
         3        1       1  Check         10
         3        2       0                 0
[...]
```

  The `IN` clause used with the `WHERE` is evaluated exactly as you would expect: for each row in the table, the query determines if that `countyname` is in the list generated by the subquery. These types of subqueries are called *uncorrelated* because nothing in the subquery references anything outside that subquery.

- When using this syntax, the subquery needs to return a single column of data. Looking at the above we can see that the subquery above satisfies this constraint.
- The opposite of `IN` is `NOT IN`, which only accepts rows do not match the contents of the subquery. For example, the following would return only the rows associated with people who have never missed a payment:

```
select
    *
from
    cls.BillPaid
where
    personid NOT IN (select personid from cls.BillPaid where paid = 0);


  personid    month    paid  paytype      amt
----------  -------  ------  ---------  -----
         1        1       1  Visa          15
         1        2       1  Visa         100
         1        3       1  Visa          15
```

- Note that the subquery syntax does not look at the name of the column within the subquery. For example, the following query will work as well:

```
select
    *
from
    cls.BillPaid
where
    personid IN (select personid as sillyColumnName
            from cls.BillPaid where paid = 0);


  personid    month    paid  paytype      amt
----------  -------  ------  ---------  -----
         2        1       1  Visa          25
         2        2       0                 0
         2        3       1  Visa          25
         3        1       1  Check         10
         3        2       0                 0
[...]
```

- Keep in mind that the reason we need to use this syntax is because we need information that is outside of the current row to evaluate the current row. A simple `WHERE` clause can only access the information in the current row.
- The `IN` clause can be used without a `SELECT` as a subquery:

```
select
    *
from
    cls.billpaid
where
    personid in (1,2);


  personid    month    paid  paytype      amt
----------  -------  ------  ---------  -----
         1        1       1  Visa          15
         1        2       1  Visa         100
         1        3       1  Visa          15
         2        1       1  Visa          25
         2        2       0                 0
[...]
```

In this case there is no official subquery – the query itself contains the data to be filtered on.

- An important consideration when writing subqueries is the use of `DISTINCT` in the subquery itself. The `IN` operator verifies if a particular value is within a list. If the list has duplicates then the verification process will take longer. In the above example, the subquery returns 3 values (2,3,3), two of which are duplicates. When making the comparison, having duplicates in the subquery list will result in an inefficient comparison. To avoid this, we generally add a `DISTINCT` operator to the subquery:

```
SELECT
    *
FROM
    cls.BillPaid
WHERE
    personid IN (select distinct personid from cls.BillPaid where paid = 0);


  personid    month    paid  paytype      amt
----------  -------  ------  ---------  -----
         2        1       1  Visa          25
         2        2       0                 0
         2        3       1  Visa          25
         3        1       1  Check         10
         3        2       0                 0
[...]
```

This will yield a more efficient query. Because the dataset is small, the difference in this query will be negligible, for larger datasets this change may be necessary for the query to run in a manageable amount of time.

- There are two other operators that are used in a similar fashion, though I do not find myself using them frequently, `ANY` and `ALL`, which are used in the following manner:

```
WHERE column [OPERATOR] ANY/ALL (SUBQUERY)
```

- For example, consider the following two examples:

```
SELECT *
FROM cls.BillPaid
where amt
    <= ALL (select amt from cls.billpaid where personID = 1);


  personid    month    paid  paytype      amt
----------  -------  ------  ---------  -----
         1        1       1  Visa          15
         1        3       1  Visa          15
         2        2       0                 0
         3        1       1  Check         10
         3        2       0                 0
[...]
```

```
SELECT *
FROM cls.BillPaid
where amt
    <= ANY (select amt from cls.billpaid where personID = 1);


  personid    month    paid  paytype      amt
----------  -------  ------  ---------  -----
         1        1       1  Visa          15
         1        2       1  Visa         100
         1        3       1  Visa          15
         2        1       1  Visa          25
         2        2       0                 0
[...]
```

In the first example, only those rows where `amt` is less than all values from PersonID #1 (15,100,15), are returned. This would return the 4 rows with amt = 0 and amt =10, this is equivalent to ≤ 15. The second query, on the other hand, only checks to see if it less than a single value within that list, so this is equivalent to ≤ 100, which returns all rows in the table.

6 Correlated Subqueries

- A correlated subquery references the outer query within the subquery. For example, consider the following query:

```
select
    *
from
    cls.cars as A
where
    vehicletype = 'Motorcycle'
    and year <> 2010
    and countyname in
    (select
        countyname
    from
        cls.cars as B
    where
        A.countyname = B.countyname
        and B.year = 2010
        and B.vehicletype = 'Motorcycle'
        and A.registrations > B.registrations);


  year  countyname    motorvehicle    vehiclecat    vehicletype    tonnage      r [...]
------  ------------  --------------  ------------  -------------  ---------  --- [...]
  2016  Jasper        Yes             Motorcycle    Motorcycle                    [...]
  2011  Ringgold      Yes             Motorcycle    Motorcycle                    [...]
  2013  Clayton       Yes             Motorcycle    Motorcycle                    [...]
  2013  Grundy        Yes             Motorcycle    Motorcycle                    [...]
  2016  Davis         Yes             Motorcycle    Motorcycle                    [...]
[...]
```

This will return all motorcycle rows, for each county that have more registrations than that same county's registrations for 2010. For example, Lucas county has the following number of registrations for each year, for motorcycles:

```
select
    year
    , registrations
from
    cls.cars
where
    countyname ='Lucas'
    and vehicletype = 'Motorcycle'
order by 1;


  year    registrations
------  ---------------
  2005              530
  2006              586
  2007              606
  2008              592
  2009              587
[...]
```

This statement will only evaluate positive in 2006 and 2007, the rows that have more registrations than 2010. To furuther understand this query, think through each row as an item within a loop, with the subquery being evaluated each time.

In Lucas, year 2005, for example, the subquery will look like :

```
(select
     countyname
from
    cls.cars as B
where
    'Lucas' = B.countyname
    and B.year = 2010
    and B.vehicletype = 'Motorcycle'
    and 530 > B.registrations);


countyname
------------
```

This subquery will return Null since no `countyname` will match the constraints in the where clause. Since it returns Null, the outer where clause evaluates False and 2005 is not returned.

- If we wanted to find all counties which increased the number of motorcycle registrations from 2005 to 2006 we could write the following query:

```
select
    countyname
from
    cls.cars as A
where
    A.year = 2006
    and A.vehicletype = 'Motorcycle'
    and countyname in
        (select countyname
            from
                cls.cars as B
            where
                year = 2005
                and A.countyname = B.countyname
                and B.vehicletype = 'Motorcycle'
                and A.registrations > B.registrations);


countyname
------------
Adair
Osceola
Madison
Worth
Hancock
[...]
```

- Correlated subqueries are costly computationally since the subquery is reevaluated for row, you can think of them as FOR LOOPS in SQL. They are also incredibly difficult to read. **Generally**

speaking, they should be avoided. We will learn techniques for avoiding them later.

- There is one interesting case for correlated subqueries, which is identifying the "first row" of a particular group. Consider the following query:

```
select
    a.countyname, a.registrations
from
    cls.cars as a
where
    a.registrations =
        (select
            registrations
        from
            cls.cars as b
        where
            a.countyname = b.countyname
        order by
            b.registrations desc
            limit 1)
```

Note that this query will take an incredibly long time to evaluate.[1] It will return, for each county, the *largest* number of registrations for a row. In other words, correlated subqueries can be used to determine the first value for a particular row. This same technique can be used to determine the maximum or minimum value of a particular column within subgroups. Later on we will learn much smarter techniques for doing this.

[1] I stopped it after one minute so I'm not sure how long it takes in total.