These are pretty much the main steps to re-creating all of the queries since the bobross dataset.

-- creating, altering, and dropping tables

First, and this is considering that there is not an already created rivers and waterfalls table based from the bobross data, we drop the existing table which in most cases I already had one created that would apply to this criteria. We can’t create a brand new table from scratch if there is already a table with the same name, so the way I used one of the requirements which was to drop the table rivers\_and\_waterfalls that I had already created before so a new one can be created.

The query for it is:

DROP TABLE rivers\_and\_waterfalls; --dropping table so everything can be created again

Once we have dropped the table with the name, we can create a table. To create a table there’s also a query for CREATE IF NOT EXIST but for the purpose of this assignment I didn’t use it to excuse the drop of the table before this step.

This is the query I used:

CREATE TABLE rivers\_and\_waterfalls( Rivers VARCHAR(200), Waterfalls VARCHAR (200)); --creating table rivers\_and\_waterfalls

In this query I created a rivers\_and\_waterfalls table to give some sense to the 1s and 0s on the bobross dataset. It is a table with two colums: The first column accepts strings and is called Rivers. It keeps track of which episodes shown a river. The second column accepts strings as well and is called Waterfalls. It keeps track of which episodes shown a waterfall. Both of this have a max-character limit of 200 but since is VARCHAR it will not count as 200 unless there is actually 200 characters inside the string stored on the cell.

Lastly, I used alter to add a new column to the table. This column is called episodes, it takes in the EPISODE CODE from the bobross dataset of all those episodes that had either a river or a waterfall or both- for now is empty of course. It also has the acceptance of a string and also takes in up to 200 characters but doesn’t take the space of as many unless truly needed to.

ALTER TABLE rivers\_and\_waterfalls ADD COLUMN episodes VARCHAR(200); --altering table to add column episodes

Included this query just to populate the data, this is the insert query it inserts a full column of episodes names into rivers and waterfalls. Later on the episodes that had neither rivers nor waterfalls are cleared out/deleted:

INSERT INTO rivers\_and\_waterfalls(episodes) SELECT episode FROM bobross; --episodes from rivers\_and\_waterfalls and bobross.episode is the association between bobross table and rivers\_and\_waterfalls table.

-subqueries, -associations

As subquery examples I used the update queries. Each column of rivers or waterfalls will be updated with a one in the episodes that match in the bobross dataset to have a river in it, or respectively to have the river cell = 1. It has not one but two subqueries.

The first subquery is when is getting the title from the episode on the bobross dataset and comparing its episode code with the rivers\_and\_waterfalls episodes column and the second is inside the one that supports the logic of only getting the titles that have a river = 1.

The functionality of the second update is pretty much the same as this one but instead of rivers is waterfalls.

Both of these queries also show the example of an association between the episodes column in both tables. Which is why I counted these update queries as both subqueries and associations (unless I got wrong the meaning of associations tbh)

UPDATE rivers\_and\_waterfalls SET rivers =

(SELECT (

SELECT title FROM bobross WHERE bobross.episode = rivers\_and\_waterfalls.episodes)

)

FROM bobross WHERE bobross.episode = rivers\_and\_waterfalls.episodes AND bobross.river= 1; ---subqueries --bringing data to the new table. Titles of the episodes that have rivers

UPDATE rivers\_and\_waterfalls SET waterfalls = (SELECT (

SELECT title FROM bobross WHERE bobross.episode = rivers\_and\_waterfalls.episodes)

)FROM bobross WHERE bobross.episode = rivers\_and\_waterfalls.episodes AND bobross.waterfall= 1; --bringing data to the new table. Titles of the episodes that have waterfalls

Just to get the rid of nulls, just for the sake of it:

DELETE FROM rivers\_and\_waterfalls WHERE waterfalls IS NULL AND rivers IS NULL; --getting rid of episodes where there is no waterfall nor river

Making the left nulls look nicer and more readable.

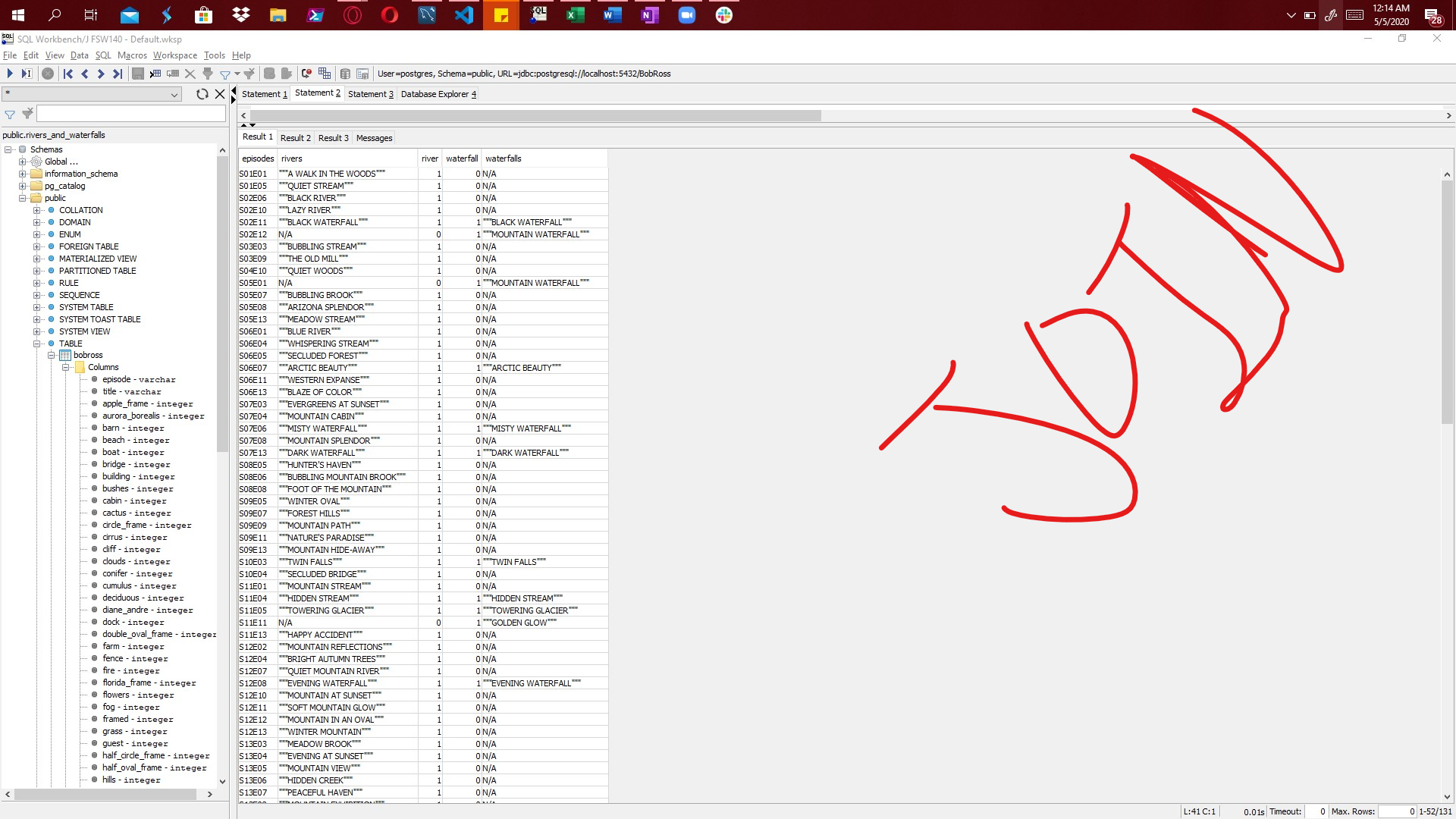
UPDATE rivers\_and\_waterfalls SET rivers = 'N/A' WHERE waterfalls IS NOT NULL AND rivers IS NULL; --labeling “N/A” where the is one of them but not the other. Here we label rivers.

UPDATE rivers\_and\_waterfalls SET waterfalls = 'N/A' WHERE rivers IS NOT NULL AND waterfalls IS NULL; --labeling “N/A” where the is one of them but not the other. Here we label waterfalls

 -joins and multiple table joins

Selecting both river and waterfall columns from bobross and rivers\_and\_waterfalls to compare them alongside the episodes column from I think rivers\_and\_waterfalls which its corresponding in the bobross dataset (I say I think because in bobross this column was either episode or episodes and on rivers\_and\_waterfalls it was the other, but since I haven’t used this data in a while I’m not sure which was which anymore). It uses a join to display the data from both tables together, hence why the screenshot says “JOIN”.

SELECT episodes, rivers, river, waterfall, waterfalls FROM rivers\_and\_waterfalls JOIN bobross ON rivers\_and\_waterfalls.episodes = bobross.episode --multiple table joins expected: columns episodes, waterfalls and rivers from rivers\_and\_waterfalls along side river and waterfall from bobross table



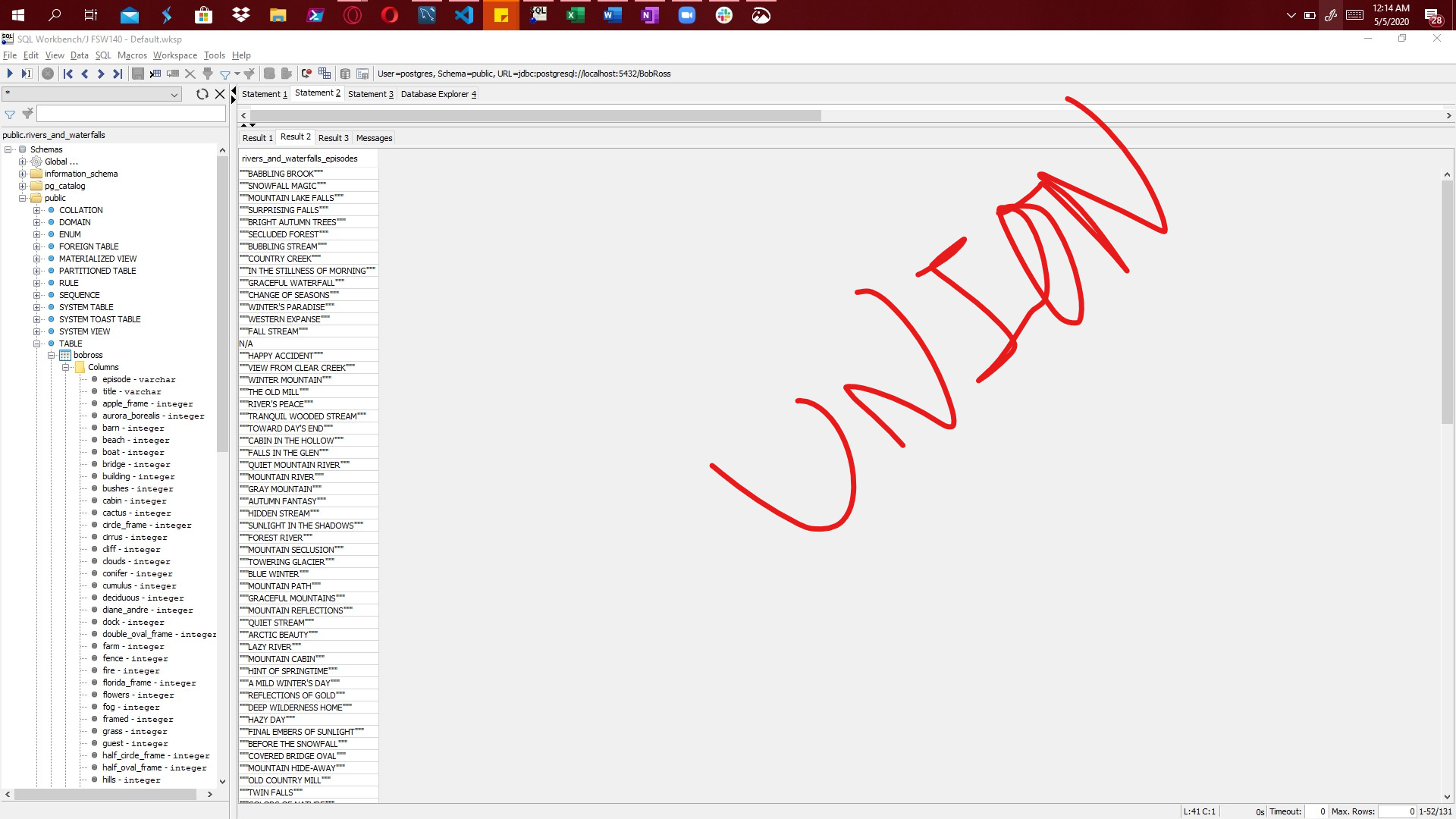
-set operations

It seemed set operations were queries like UNION and UNION ALL, I’m not sure if that is right but this is the UNION query. In this query we select the waterfalls column in the rivers\_and\_waterfalls table, name this column as rivers\_and\_waterfalls\_episodes and combine it with the rows from the rivers column from the same table, so we have a list of all the episodes that had either a river, a waterfall or both. Oh, the main difference from UNION and UNION ALL is that UNION does not over-repeat values that they might have in common whereas UNION ALL will repeat It as many times as it finds it.

SELECT waterfalls as rivers\_and\_waterfalls\_episodes FROM rivers\_and\_waterfalls

UNION

SELECT rivers FROM rivers\_and\_waterfalls; --set operations; ex: UNION expected: waterfall rows + rivers rows with no repetition of rows that are shared by both waterfalls and rivers.



-order of operation of queries

I thought the best way to portrait order of operations might be to include Group somehow, so this is the weird query I came up with:

First from rivers\_and\_waterfalls (because from happens first and joins are part of froms) we select \* everything, but then we filter it with WHERE to only show the rows where both rivers and waterfalls are not null. Then all the values are grouped per the rivers column and counted. Lastly only unique values are left after DISTINCT runs. This weird order of operations is quite funny.

SELECT DISTINCT COUNT(\*) FROM rivers\_and\_waterfalls WHERE rivers IS NOT NULL AND waterfalls IS NOT NULL GROUP BY rivers; --order of operations ; expected 1 and 5 – ALL titles that have waterfall and river are present only one per group, except for one that is 5, hence I added distinct to it and all that returns is the unique or distinct results which are 1 and 5.

