Kelly Compton

Complete Assignment: Bob Ross

CREATE TABLE & ASSOICATIONS

Below I have provided screenshots of creating a new table and creating associations. As shown below I am using the bobross schema then creating a new table called ADDTL\_EPOSIDE\_INFO. I am then creating the columns EPISODE, EPISODE\_RERUN, RATINGS, EPISODE\_LENGTH, NUMBER\_RERUNS and the type of data they can hold, int and varchar. I then create the association by adding an EPISODE column like the original elements-by-episode table. I first made the elements-by-episode table’s column EPISODE into a primary key. Then the EPISODE column I created in the new table I made the foreign key referencing the EPISODE column on the original table to create the association. I then went on to insert values into the columns on my new table. Once run, it created 5 columns and 12 rows.

**QUERY:**

USE bobross;

CREATE TABLE ADDTL\_EPISODE\_INFO (

EPISODE varchar(6) NOT NULL,

EPISODE\_RERUN varchar(3) NOT NULL,

RATINGS int,

EPISODE\_LENGTH int,

NUMBER\_RERUNS int,

FOREIGN KEY (EPISODE)

REFERENCES `elements-by-episode` (EPISODE));

INSERT INTO ADDTL\_EPISODE\_INFO (EPISODE, EPISODE\_RERUN, RATINGS, EPISODE\_LENGTH, NUMBER\_RERUNS)

VALUES

('S01E01', 'Yes', '3', '49', '5'),

('S01E02', 'Yes', '5', '60', '3'),

('S01E03', 'Yes', '2', '55', '4'),

('S01E04', 'Yes', '3', '65', '2'),

('S01E05', 'Yes', '1', '45', '5'),

('S01E06', 'No', '5', '40', '0'),

('S01E07', 'No', '5', '50', '0'),

('S01E08', 'No', '4', '60', '0'),

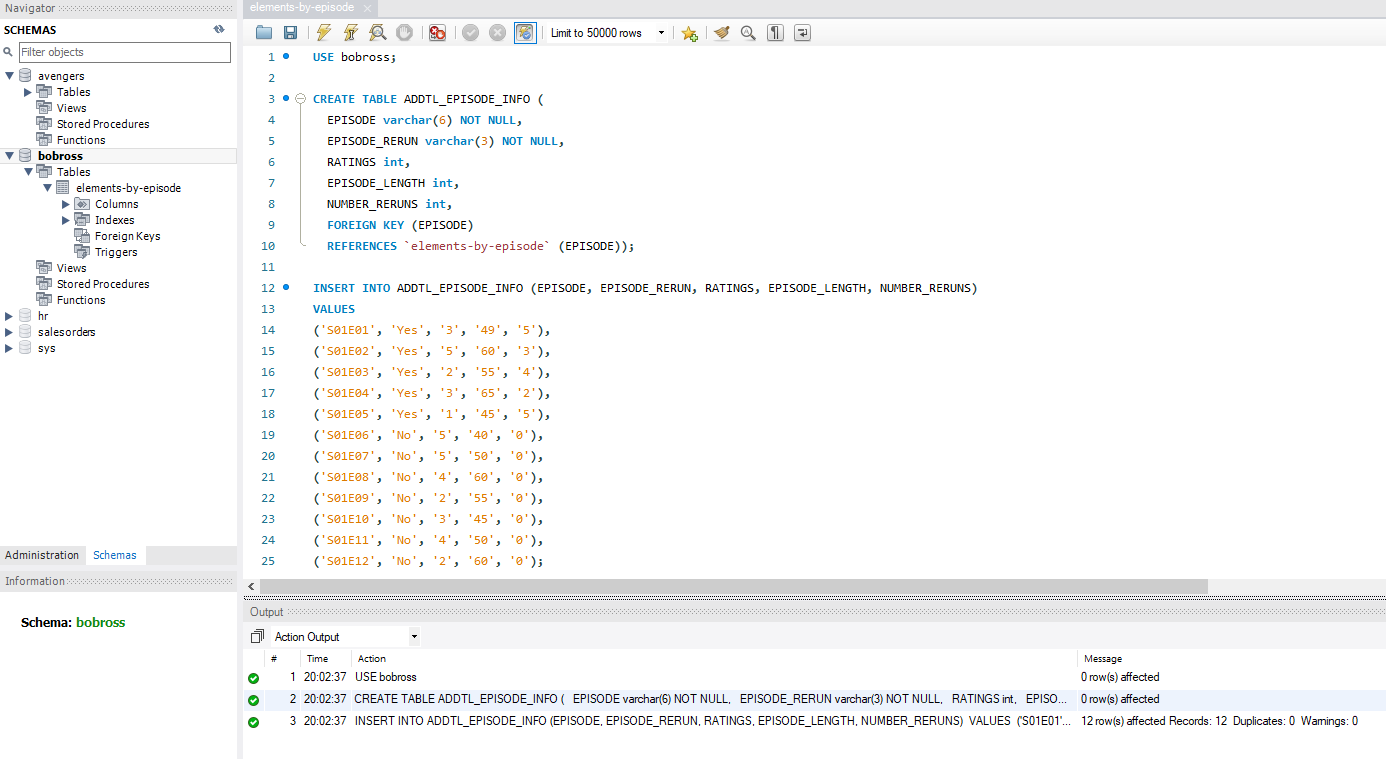
('S01E09', 'No', '2', '55', '0'),

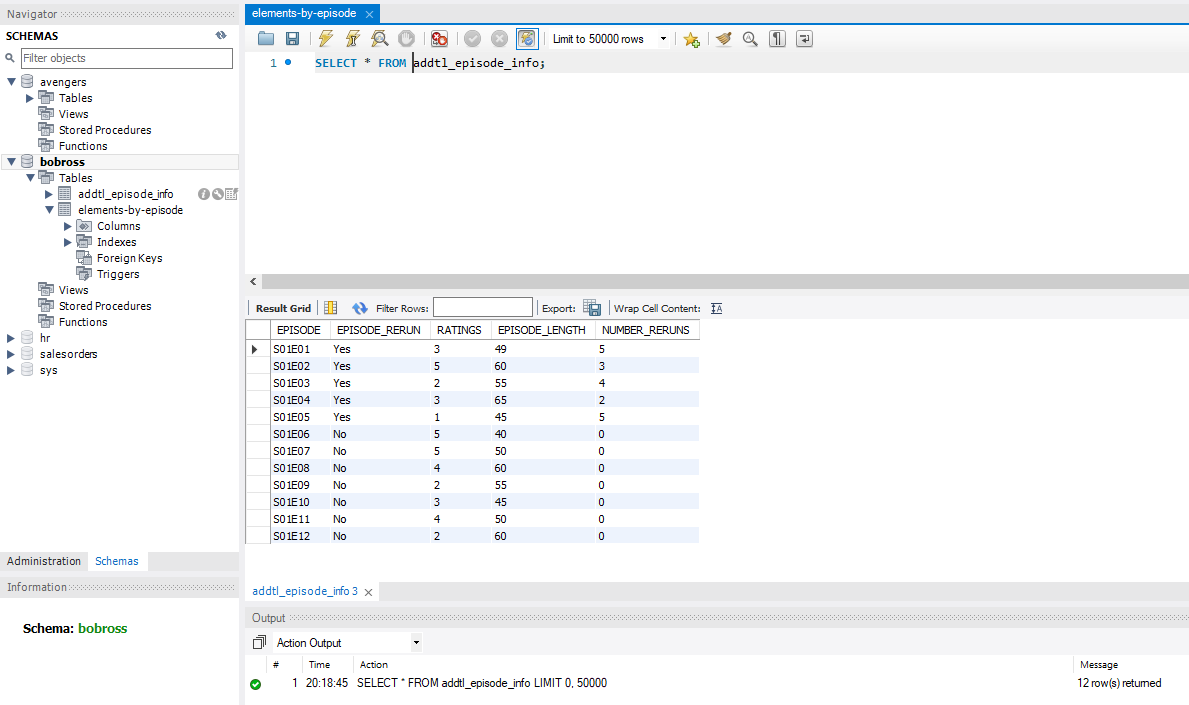
('S01E10', 'No', '3', '45', '0'),

('S01E11', 'No', '4', '50', '0'),

('S01E12', 'No', '2', '60', '0');

**SCREENSHOTS:**





ALTER & DROPPING TABLES

In this example I am altering and dropping the new table I created. However, I will only show the query to drop the table successfully but will not actually drop the table since I must use it in other queries. In the ALTER query I am first selecting the table I would like to alter the addtl\_episode\_info, then I am adding a new column called LENGTH\_OF\_COMMERCIALS and giving it a datatype of integer. Screenshot has been provided showing the new column that was added. Then below in the query section I have added the DROP table query, which is honestly very simple. All that is needed is DROP TABLE then insert the table you wish to drop after, which I chose addtl\_eposide\_info, again only for reference.

**QUERY:**

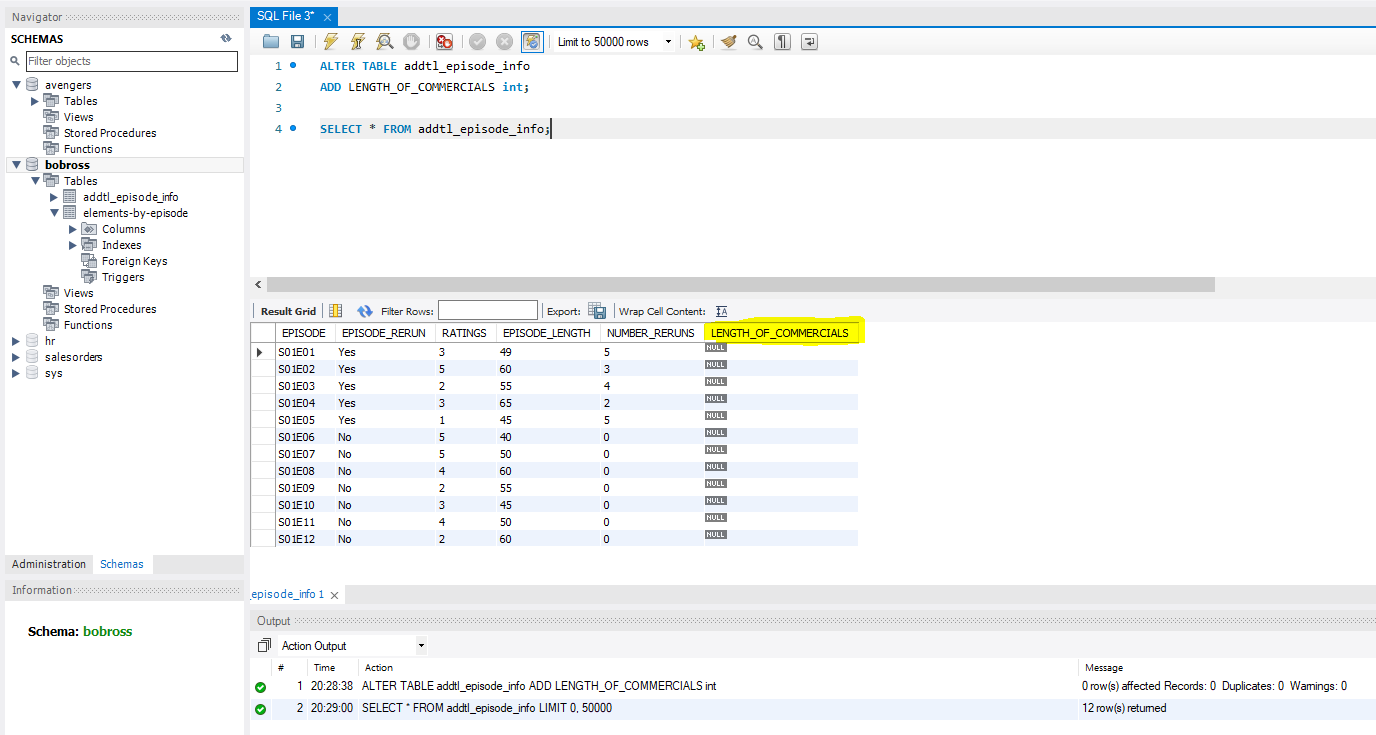
ALTER TABLE addtl\_episode\_info

ADD LENGTH\_OF\_COMMERCIALS int;

&

DROP TABLE addtl\_episode\_info;

**SCREENSHOT:**



SET OPERATIONS

The SET operation I will be preforming today is UNION. In the query below I will be using UNION and UNION ALL to combine the title from elements-by-episode table and eposide\_rerun from the addtl\_episode\_info table into one single column. In the first query you will see that UNION will return all the titles since they are unique values, but it only lists “Yes” and “No” once. This is because UNION alone only will grab unique values and not duplicate values. This leaves us with 403 rows that were returned. When I use UNION ALL as shown on the next query and screenshot you will see that it brings back all titles and well as ALL the “Yes” and “No” values from the addtl\_episode\_info table. This increases the total rows returned to 415, since duplicate values are now included.

**QUERY:**

SELECT TITLE FROM `elements-by-episode`

UNION

SELECT EPISODE\_RERUN FROM addtl\_episode\_info;

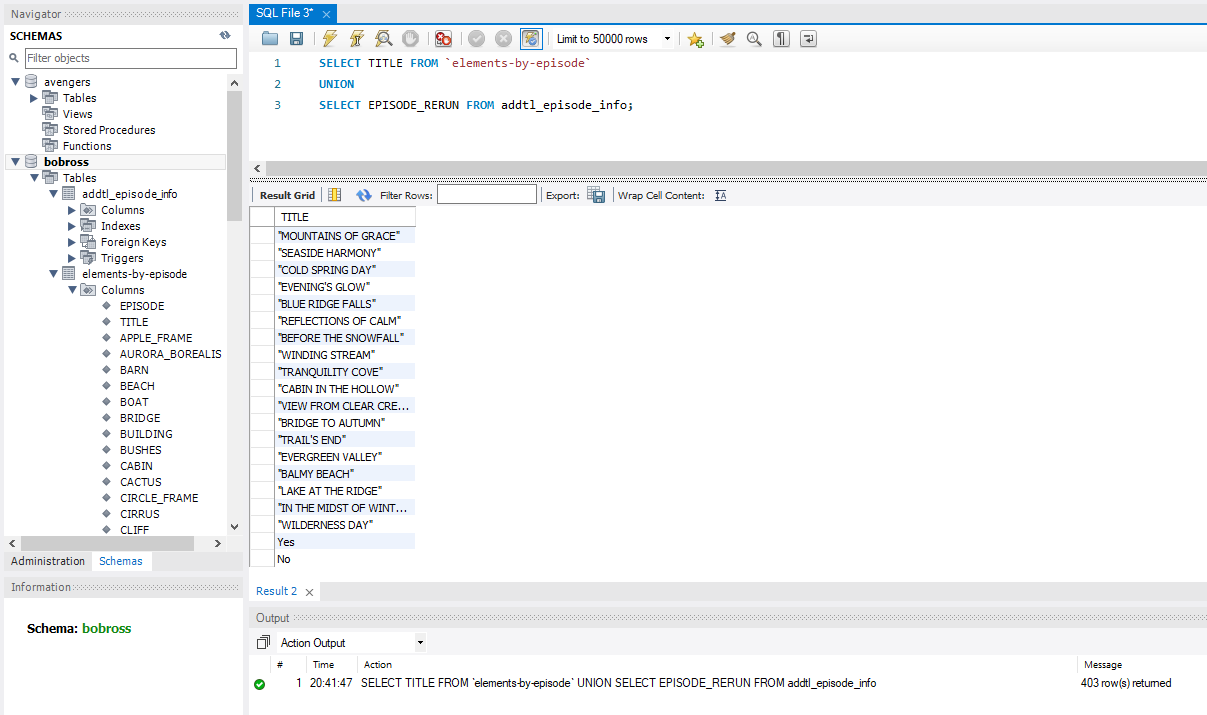
&

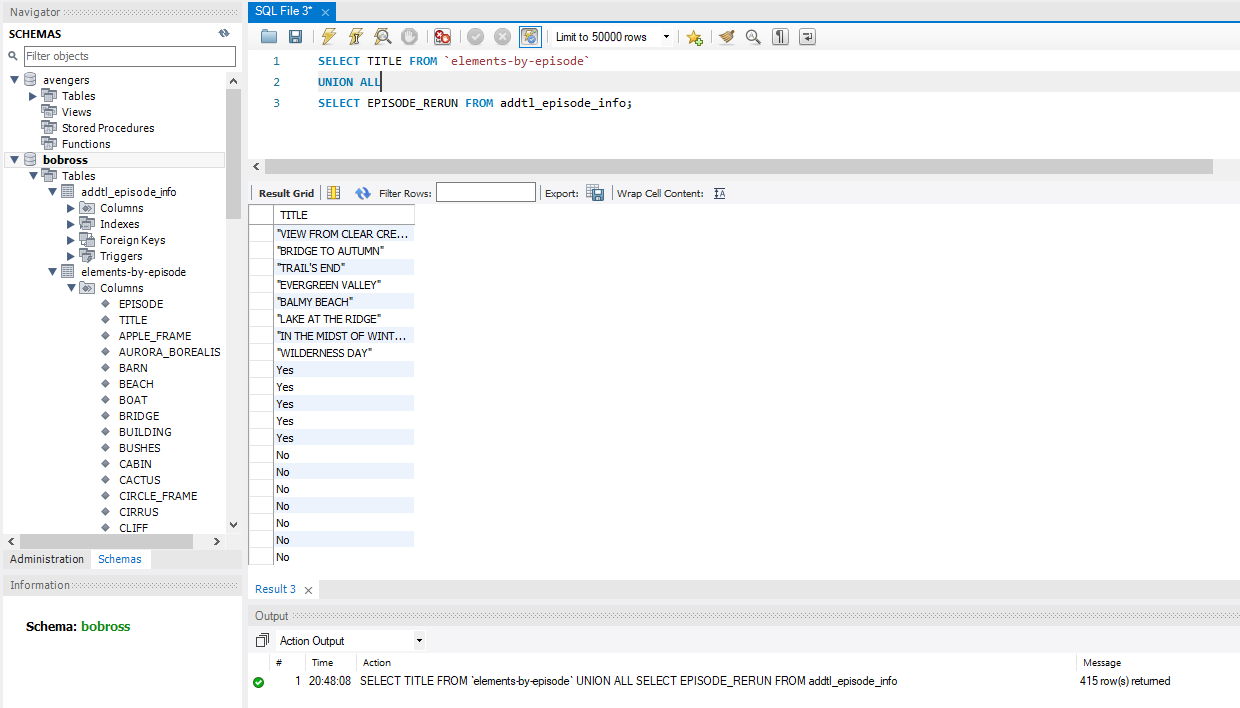
SELECT TITLE FROM `elements-by-episode`

UNION ALL

SELECT EPISODE\_RERUN FROM addtl\_episode\_info;

**SCREENSHOT:**





ORDER OF OPERATIONS OF QUERIES

The order of operations of queries is the order in which each part of the query is executed. The example I am providing you today I used, SELECT, FROM, WHERE, GROUP BY, HAVING, ORDER BY and LIMIT. The first operation to be executed is FROM, it will go to the table listed in the FROM command. Next, it will look for the WHERE clause so it can start to filter the data as requested. Then it will move on to the GROUP BY clause aggregating the data according to the column TITLE. After we grouped the data the HAVING clause will be next and this is like the WHERE clause, in the sense that it will filter the data even more based on the query written. Next, is the SELECT statement which will return the final filtered data from the columns requested. Once that is executed it will move on the ORDER BY clause ordering the filtered data in ascending or descending order. Lastly, is the LIMIT, the LIMIT will take the number listed and discard all rows remaining after the number presented This will leave you with only the first number of rows up to the number listed. As you can see in the screenshot 22 rows were returned based on all the conditions listed.

**QUERY:**

SELECT TITLE, CLOUDS, TREE

FROM `elements-by-episode`

WHERE CLOUDS = 1

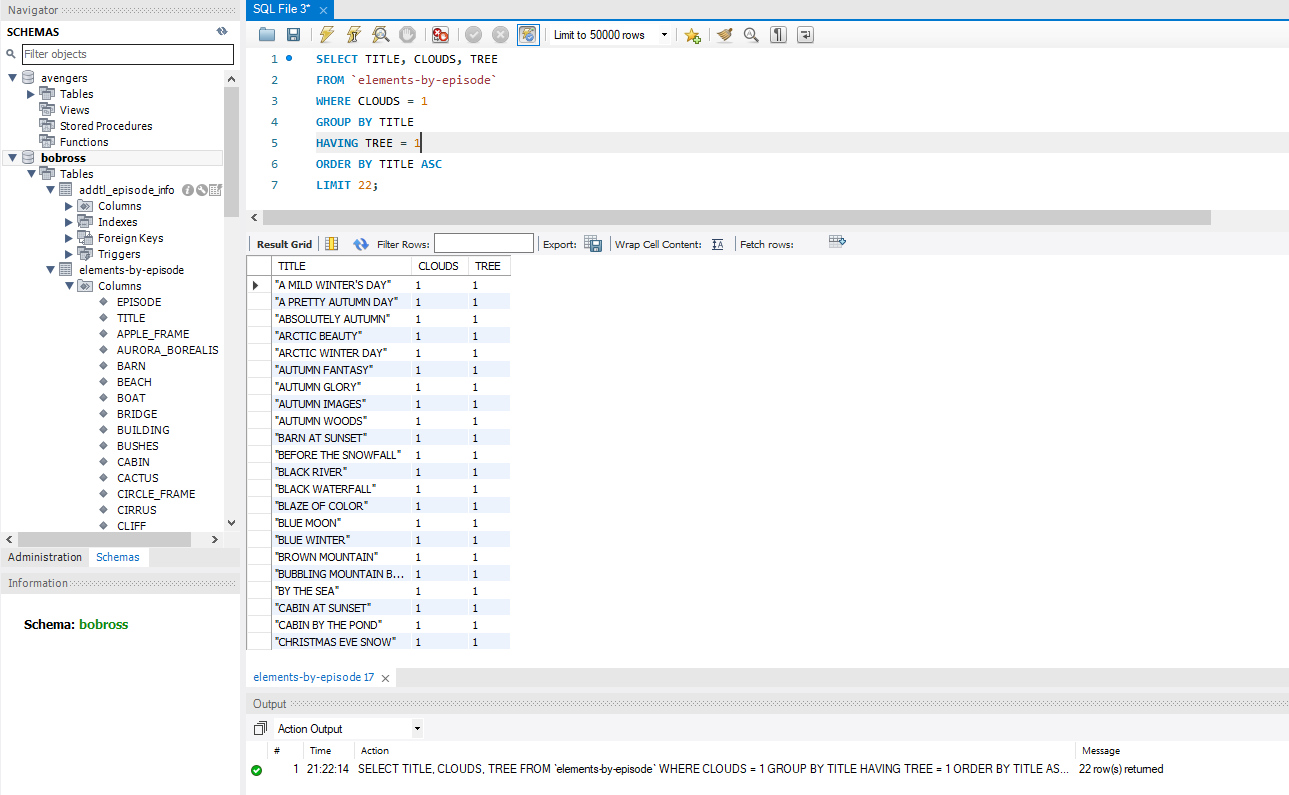
GROUP BY TITLE

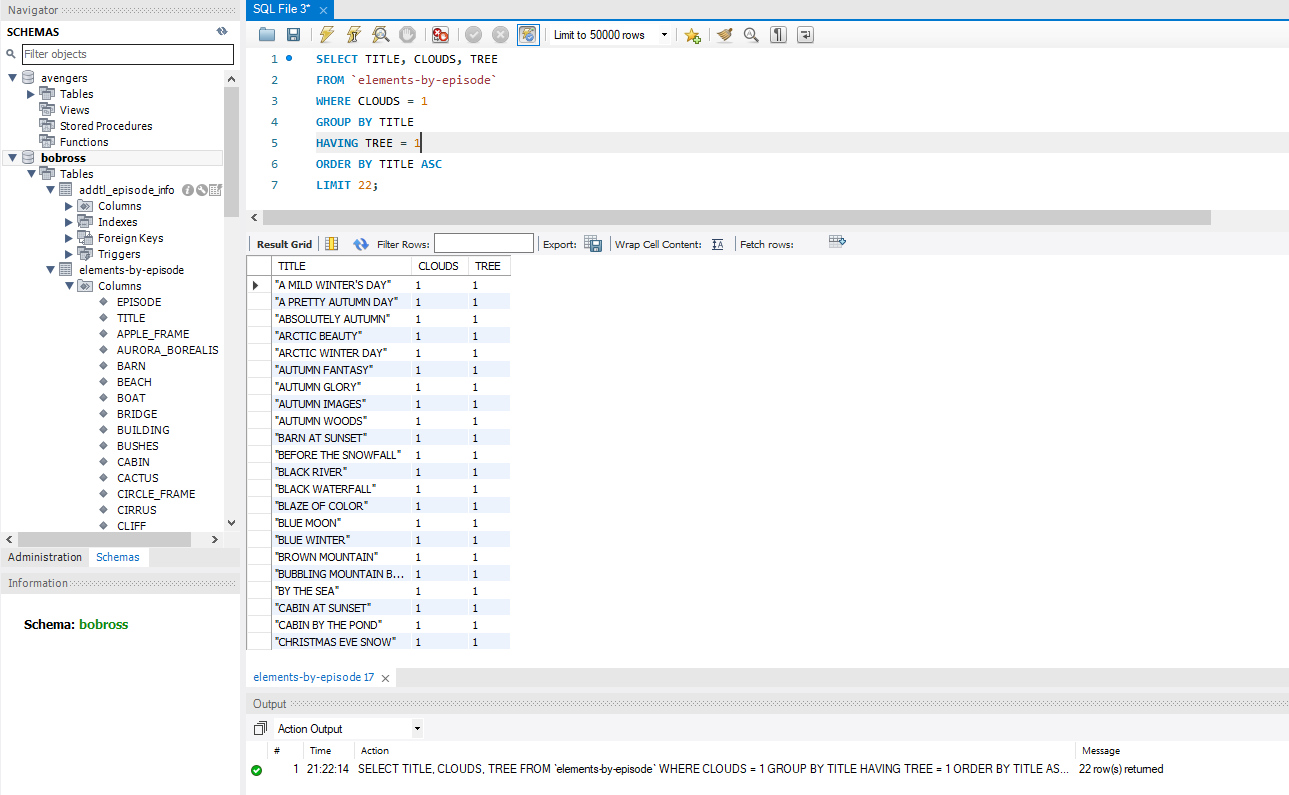
HAVING TREE = 1

ORDER BY TITLE ASC

LIMIT 22;

**SCREENSHOT:**





SUBQUERIES

A subquery simply put is a query within another query. The subquery is a query that is embedded in the WHERE clause of another SQL query. In my example below first, we are going to the elements-by-episode table to pull from. Next, the subquery that is wrapped in parenthesis is executed returning the episode WHERE the cabin column is equal to 1 and the cirrus column is equal to 1. Then the outer-query executes it and returns all columns from the elements-by-episode table whose EPISODE is returned from the inner subquery. This will leave us with a row count of 4 and only the episodes that have both a CABIN and CIRRUS drawn in them.

**QUERY:**

SELECT \*

FROM `elements-by-episode`

WHERE EPISODE IN (SELECT EPISODE

FROM `elements-by-episode`

WHERE CABIN = 1 AND CIRRUS = 1);

**SCREENSHOT:**

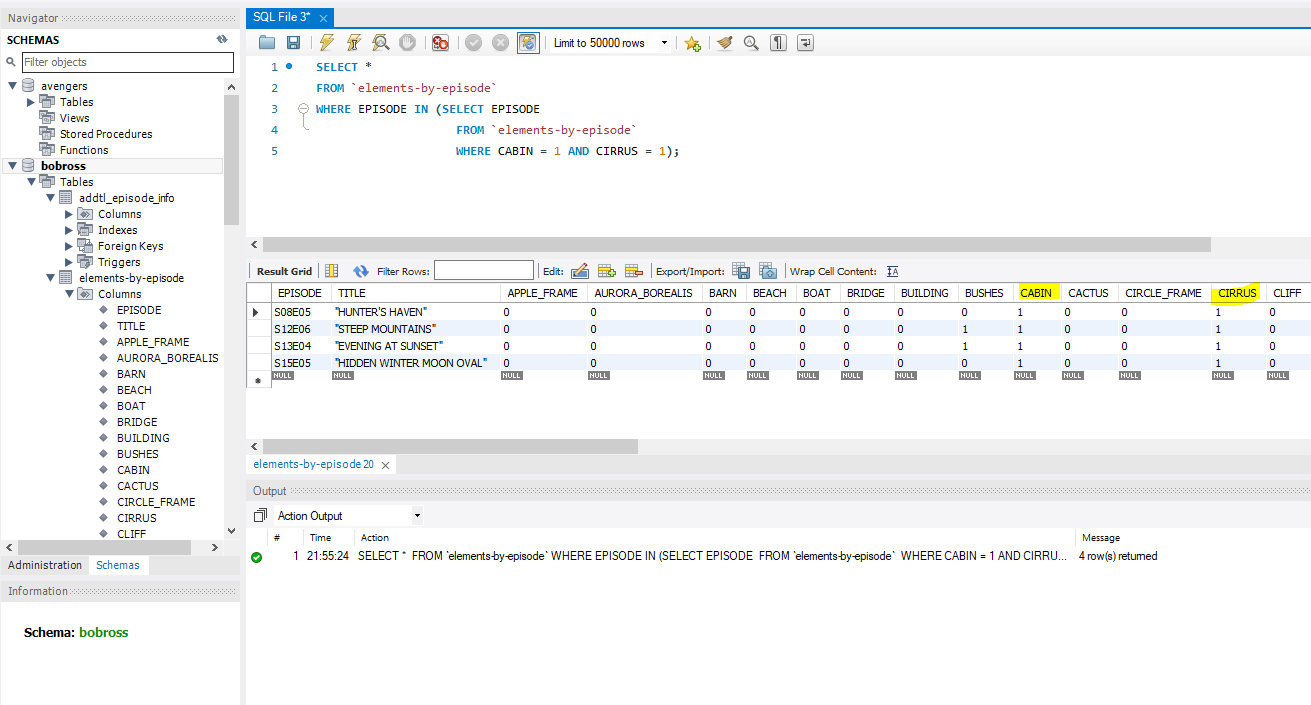


TABLE JOINS

A JOIN clause is used to combine rows from two or more tables based on a related column. The JOIN that I will be demonstrating today is an INNER JOIN. The INNER JOIN selects records that have matching values in both tables. In the example that I provided below; I am joining both tables. First, I am selecting the elements-by-episode table, then I am running an INNER JOIN on the addtl\_episode\_info table. This INNER JOIN will select the matching values of the EPISODE column on each table. Then I am selecting the TITLE column from elements-by-episode table and EPISODE\_RERUN, RATINGS and EPISODE\_LENGTH from addtl\_episode\_info table. The results are shown on the screenshot below returning exactly 12 rows, since there are only 12 matching values between both EPISODE columns.

**QUERY:**

SELECT `elements-by-episode`.TITLE, addtl\_episode\_info.EPISODE\_RERUN, addtl\_episode\_info.RATINGS, addtl\_episode\_info.EPISODE\_LENGTH

FROM `elements-by-episode`

INNER JOIN addtl\_episode\_info

ON `elements-by-episode`.EPISODE = addtl\_episode\_info.EPISODE;

**SCREENSHOT:**

