Data Storage and Retrieval

Fall 2018

Assignment 6

Diana Sujanto

October 7, 2018

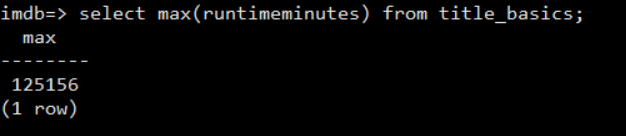
Show all your work!

For questions 1-4 use the Postgres imdb database.

1. Write queries for these problems, show your results, and explain your answer:
   * 1. Find the title of the longest-running movie. Is it a real movie? Find the longest real movie.

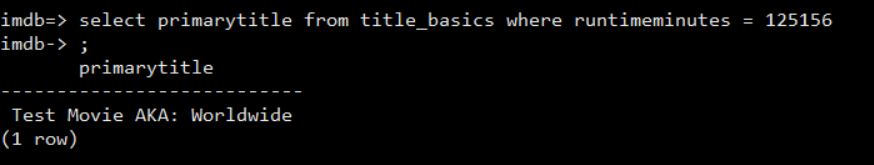
First, I found the longest running time, which is 125156 minutes.

SELECT MAX(runtimeminutes) FROM title\_basics;



However, as I ran a query to find the title with this running time, it was not a real movie. It was a ‘Test Movie AKA: Worldwide’.

SELECT primarytitle FROM title\_basics WHERE runtimeminutes = 125156;



Therefore, I redesigned a query as follows:

SELECT primarytitle, MAX(runtimeminutes) FROM title\_basics

WHERE primarytitle NOT LIKE ‘TEST%

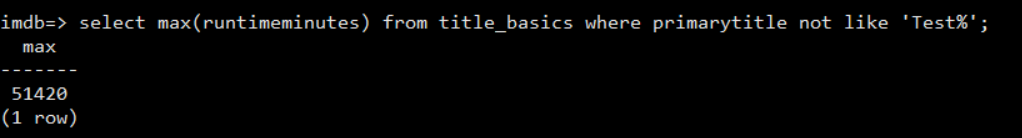
GROUP BY primarytitle;

However, the query took a long time to run. As you can see from the screenshot, I had to cancel the statement.

Then, I decided to separate the queries into two parts:

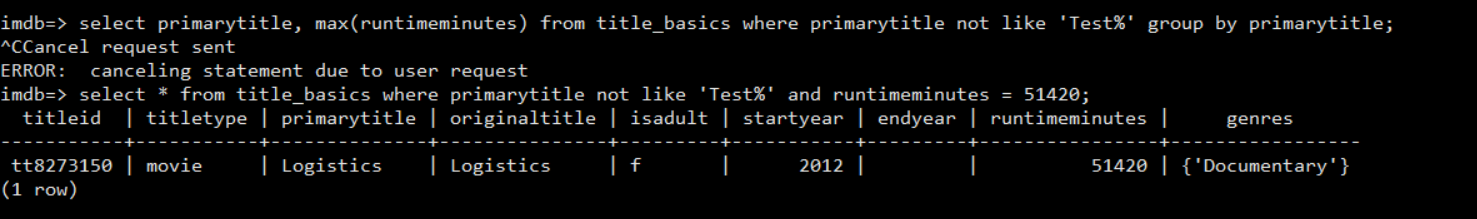
1. Find the maximum movie runtime where the movie is not a ‘Test’ movie.

SELECT MAX(runtimeminutes) FROM title\_basics WHERE primarytitle NOT LIKE 'Test%;



1. Find the title with the maximum runtime of 51420 minutes.

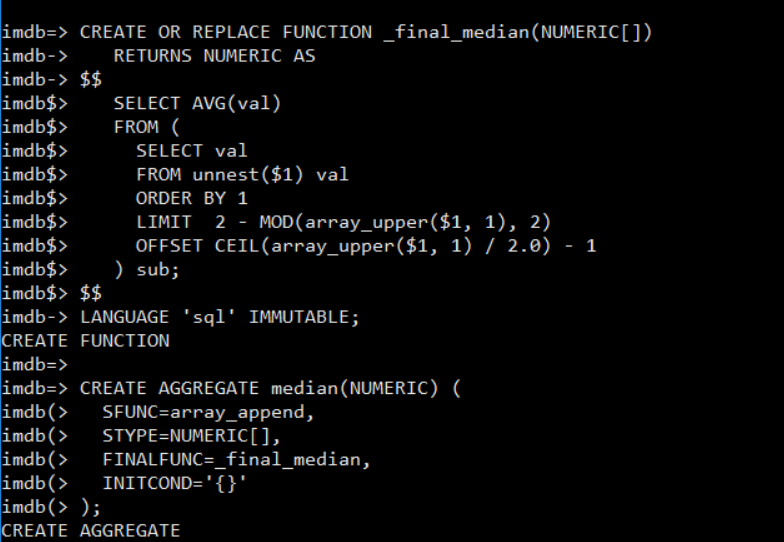
SELECT \* FROM title\_basics WHERE primarytitle NOT LIKE 'Test%' AND runtimeminutes = 51420;



The movie with the title “Logistics” has a runtime of 51420 minutes. According to Wikipedia, this is a 2012 [experimental](https://en.wikipedia.org/wiki/Experimental_film) art film conceived and created by Erika Magnusson and Daniel Andersson. At 51,420 minutes, (857 hours or 35 days and 17 hours), it is the [longest movie](https://en.wikipedia.org/wiki/List_of_longest_films) ever made.

* + 1. For each type of show, find the median running time in minutes. If you use some trial and error, explain your process and show your final query.

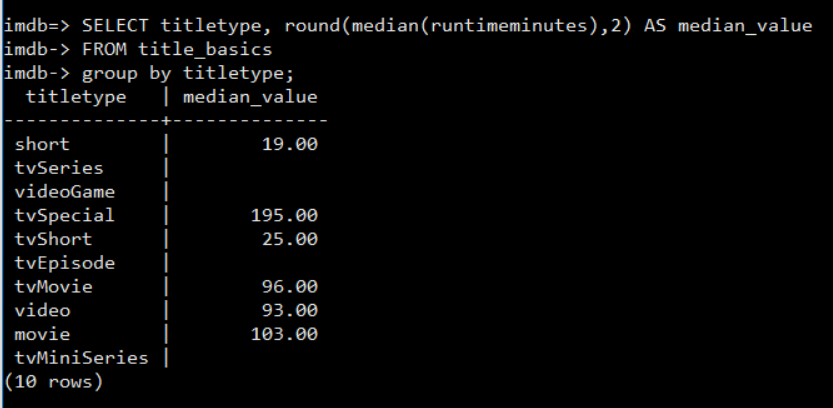
Postgres does not have a built-in median function. I found the aggregate function online and created it manually. Then, I rounded the median to 2 decimal points.



SELECT titletype, ROUND(median(runtimeminutes),2) AS median\_value

FROM title\_basics

GROUP BY titletype;

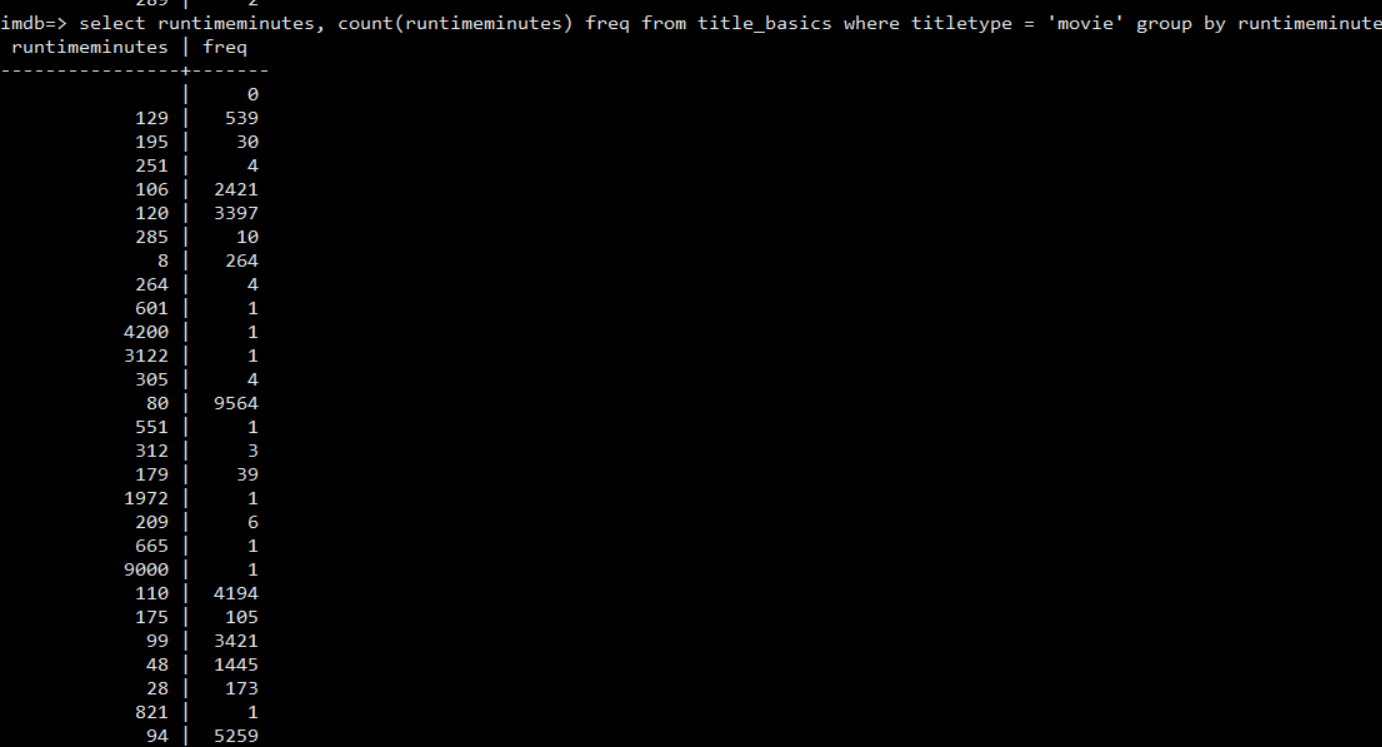


As you can see that some of the median values are NULL.

* + 1. For shows of type ‘movie’, find the *mode* of running time -- the running time in minutes for which there is a plurality of movies. Explain why the natural query you might try is probably wrong, by finding evidence of bad data. Then explain what you think the right answer is.

First, I counted the movie runtime group by the movie runtime to get a frequency table. Below is the example of the frequency table:

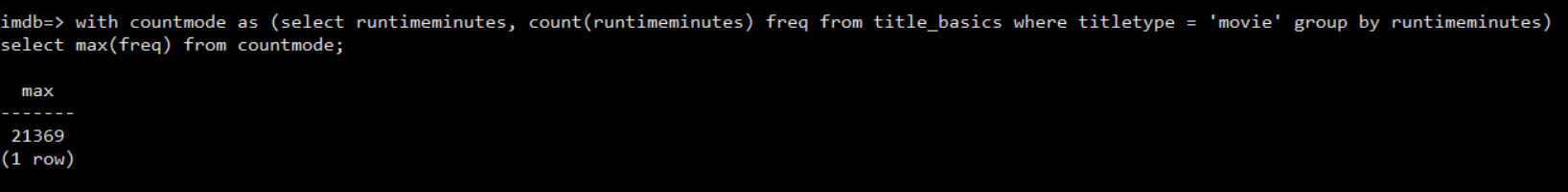
SELECT runtimeminutes, COUNT(runtimeminutes) freq FROM title\_basics WHERE titletype = 'movie' GROUP BY runtimeminutes;

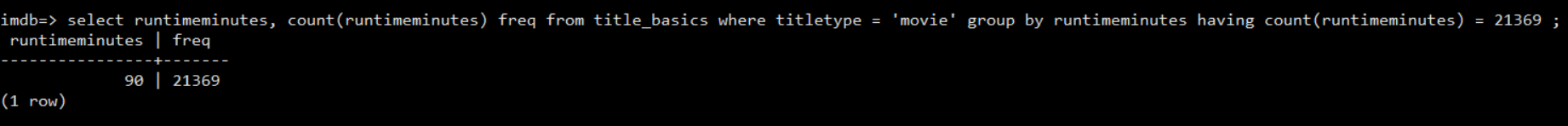


To get the mode of movie runtime, I used ‘WITH’ statement and find the maximum number of frequency(21369), which turned out to be the ’90 minutes’ movie runtime.

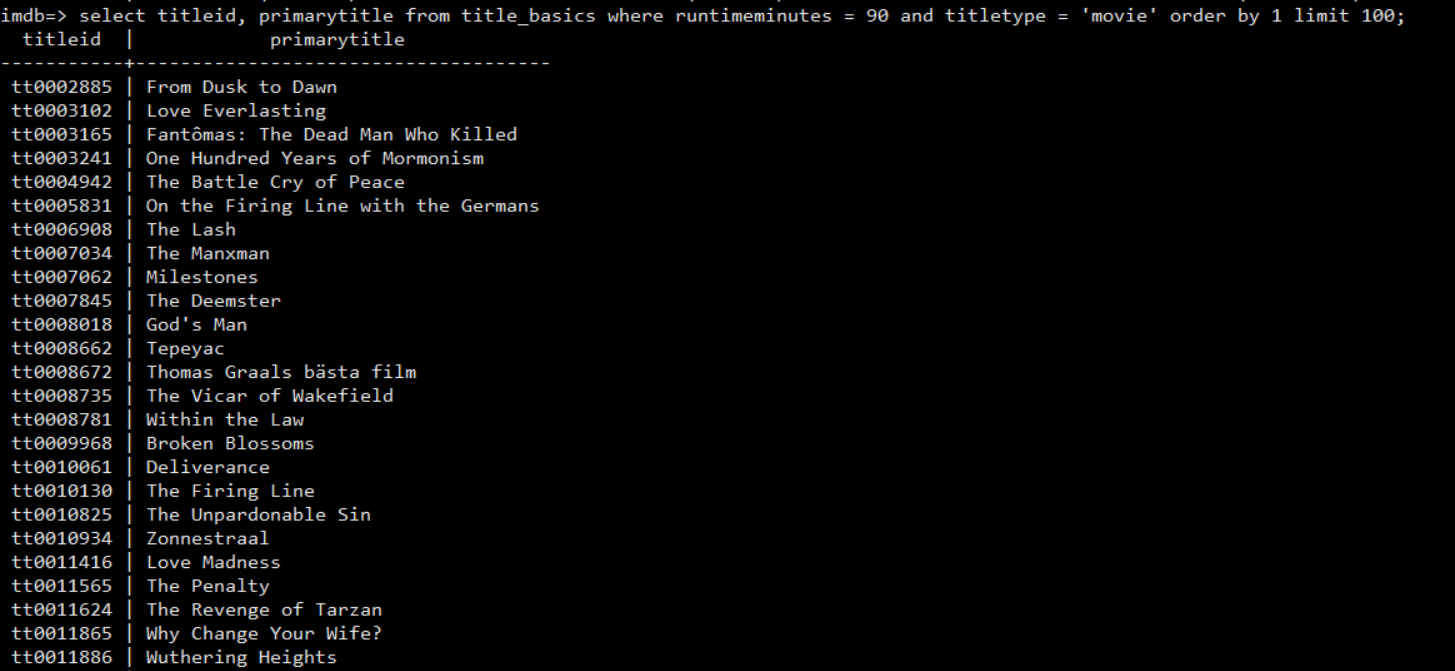
WITH countmode AS (SELECT runtimeminutes, COUNT(runtimeminutes) freq FROM title\_basics WHERE titletype = 'movie' GROUP BY runtimeminutes)

SELECT MAX(freq) FROM countmode;

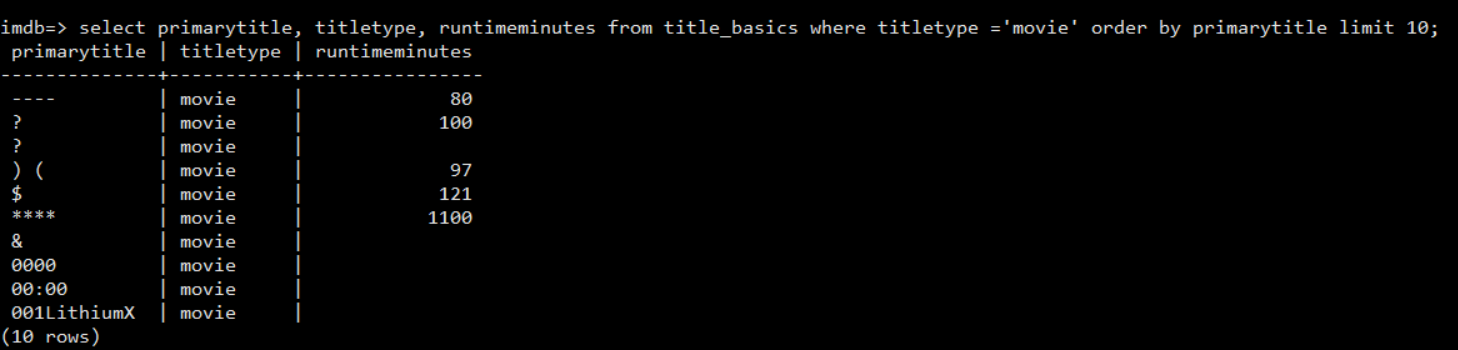




Below are some examples of the movies that ran for 90 minutes:



As I did more analysis, I found some bad movie titles with random characters having possibly valid movie runtime.



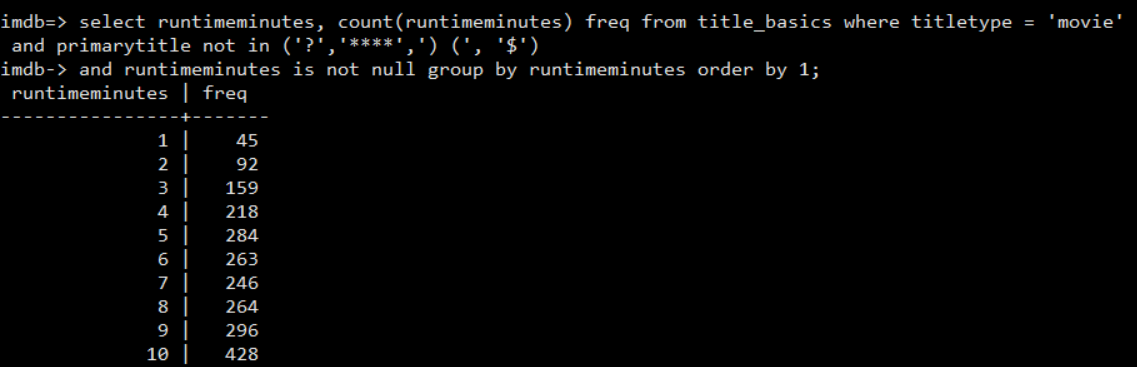
More possible bad data (movie title that starts with numbers or only contains some numbers, short characters)



Therefore, my original query to get the mode of movie runtime may not be correct 100%. Movies with invalid titles (bad data) definitely are included when counting the frequency of movie runtime. My solution would be to at least exclude the null values of movie runtime and movie titles with ‘regular expressions’, such as ‘?, \*\*\*\*, $, ) (, &’.

SELECT runtimeminutes, COUNT(runtimeminutes) freq FROM title\_basics WHERE titletype = 'movie' AND primarytitle NOT IN ('?','\*\*\*\*',') (', '$')

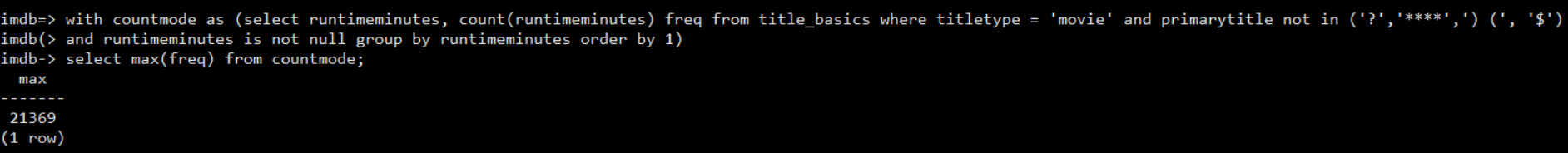
AND runtimeminutes IS NOT NULL GROUP BY runtimeminutes ORDER BY 1;



Then, I rerun the query to get the mode of the movie runtime

WITH countmode AS (SELECT runtimeminutes, COUNT(runtimeminutes) freq FROM title\_basics WHERE titletype = 'movie' AND primarytitle NOT IN ('?','\*\*\*\*',') (', '$') AND runtimeminutes IS NOT NULL GROUP BY runtimeminutes ORDER BY 1)

SELECT MAX(freq) FROM countmode;



It turns out the mode of the movie runtime is still 90 minutes. The number of movies that have 90 minutes movie runtime is 21369 (It does not change from the original query). Although I found some bad movie titles, it did not affect the mode of the movie runtime in this case.

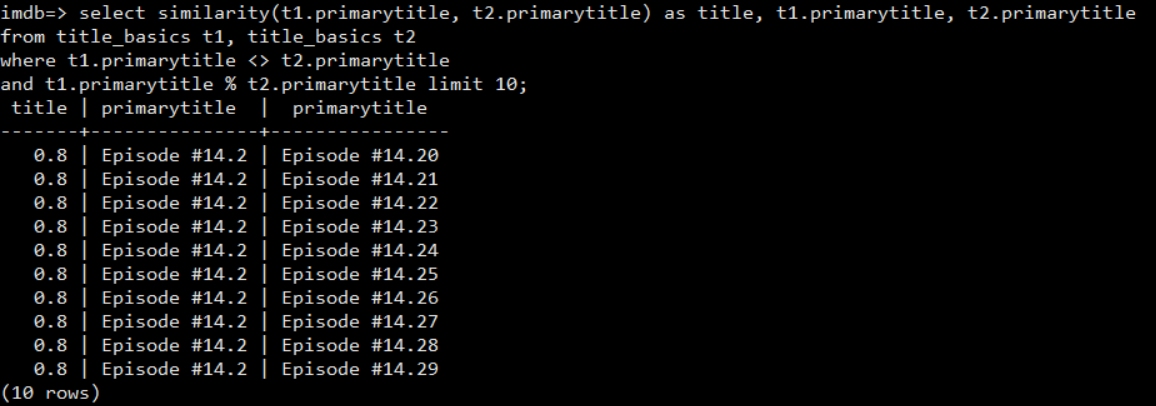
1. Come up with a way to measure the number of sequels of a movie. Show the top ten movies ranked by number of sequels.

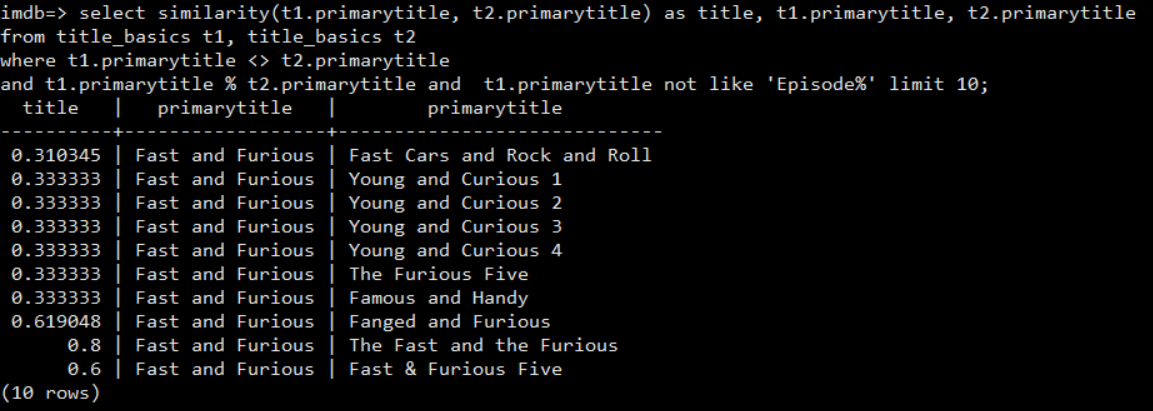
At first, I thought of using ‘similarity’ between movie titles to measure the number of sequels of a movie. It worked for some movies, like the example below, but ambiguous for the others.

SELECT similarity(t1.primarytitle, t2.primarytitle) AS title, t1.primarytitle, t2.primarytitle FROM title\_basics t1, title\_basics t2

WHERE t1.primarytitle <> t2.primarytitle

AND t1.primarytitle % t2.primarytitle limit 10;





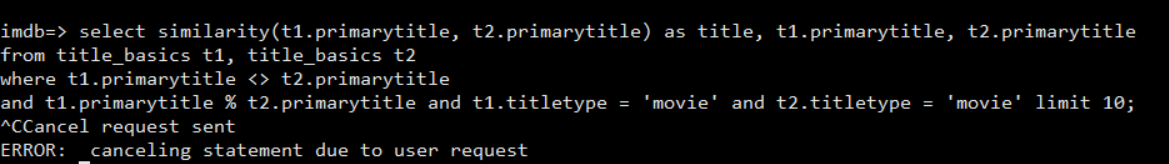
From the above example, you can see that there are possible movie sequels for Fast and Furious( Fast and Furious Five, The Fast and the Furious). The rest were not to be determined to be the sequels (such as Young and Curious, Famous and Handy, Fast Cars and Rock and Roll). So, I increased the threshold to 0.9 to see if we can get more accurate measurement for movie sequels. Also, I modified the query to limit to type = ‘movie’

SELECT SET\_LIMIT (0.9);

SELECT similarity(t1.primarytitle, t2.primarytitle) AS title, t1.primarytitle, t2.primarytitle FROM title\_basics t1, title\_basics t2

WHERE t1.primarytitle <> t2.primarytitle

AND t1.primarytitle % t2.primarytitle AND t1.titletype = 'movie' AND t2.titletype = 'movie' LIMIT 10;



The query did not come back after a long period of time. So, I reuse the original query to see if increasing threshold/limit helps produce better results. It turned out that the query lasted a long time and never came back with the results.

So, I reset the limit back to 0.8. With the logic of comparing between 2 movie title and assuming that the sequel has 3 characters more, I designed the query as the following; however, it ran for a very long time:

SELECT similarity(t1.primarytitle, t2.primarytitle) AS title, t1.primarytitle, t2.primarytitle

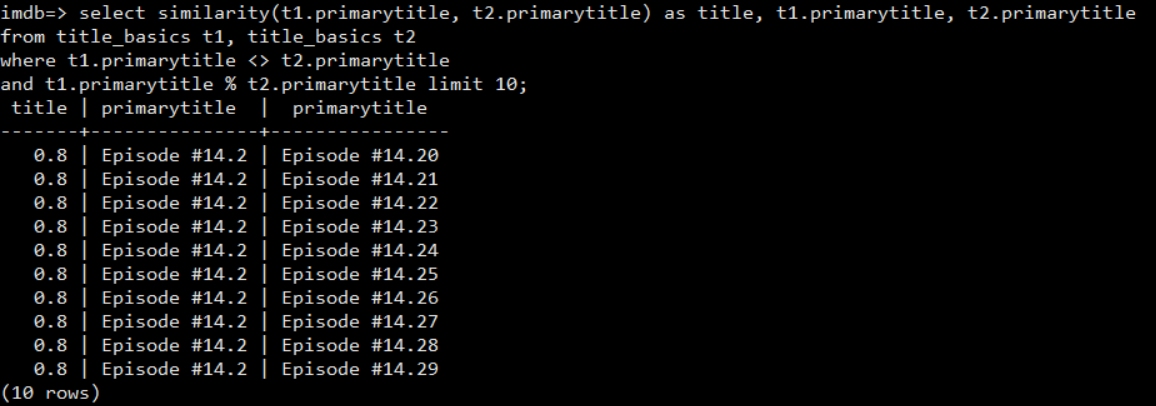
FROM title\_basics t1, title\_basics t2

WHERE t1.primarytitle <> t2.primarytitle

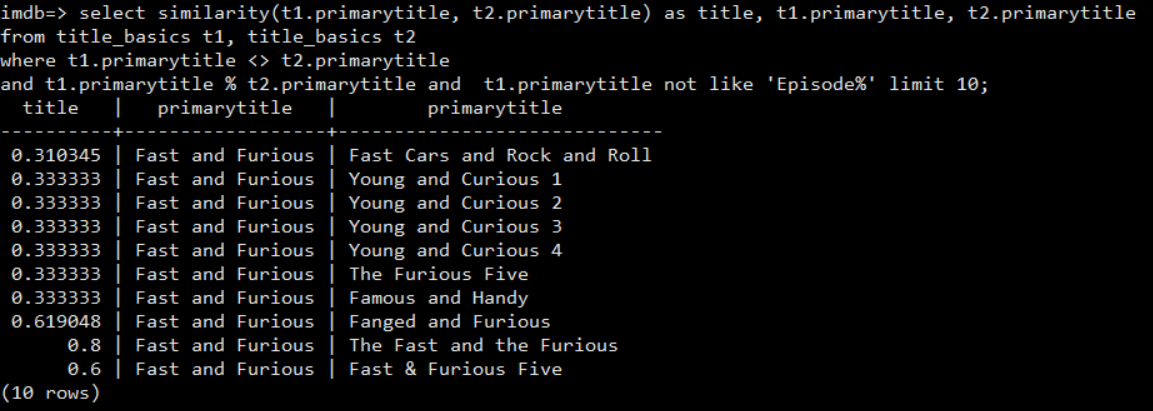
AND t1.primarytitle % t2.primarytitle

AND substring(t1.primarytitle, 1, length(t1.primarytitle)) LIKE substring(t2.primarytitle, 1, length(t2.primarytitle) - 3) AND length(t1.primarytitle) > 5 AND length(t2.primarytitle) > 5 AND t1.primarytitle not like 'Episode%' limit 10;

In the end, the best query I can get to measure sequels is:



Or if we want to see other movies than Episode, we can do:



1. Read the Postgres documentation on the page header and the pg\_stats and pg\_settings tables. Make the following queries.

Query A

select name, short\_desc, min\_val, max\_val from pg\_settings;

Query B

select attname, avg\_width from pg\_stats

where tablename = 'title\_basics';

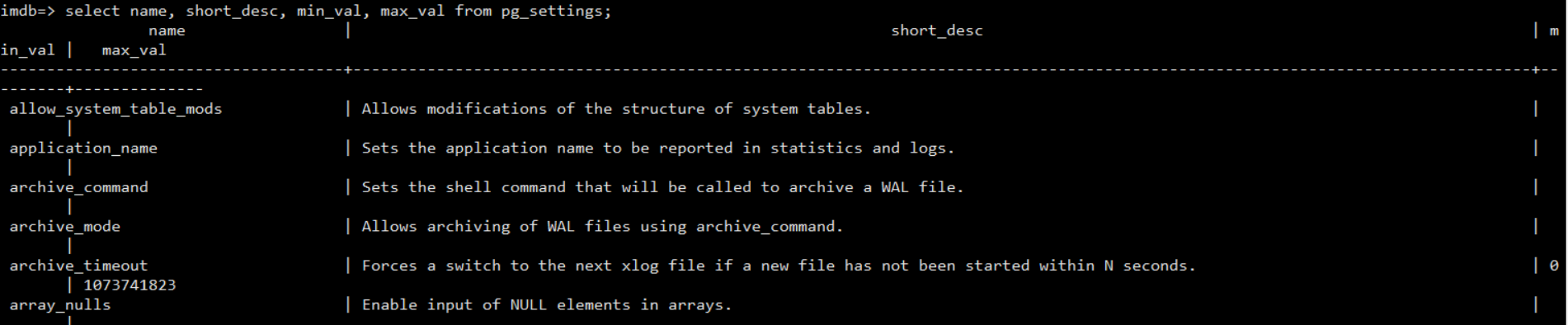
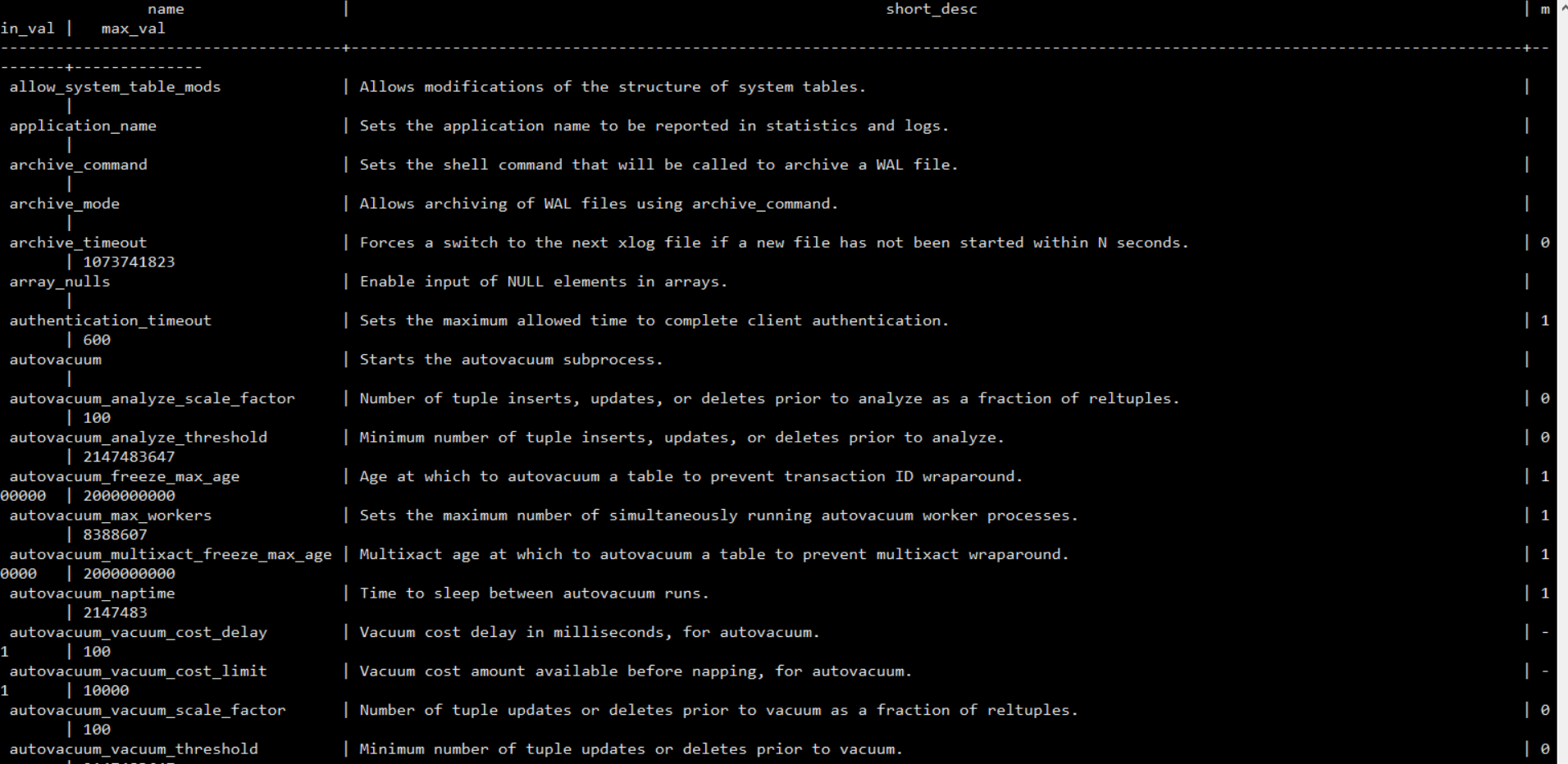
Query C

select primaryTitle, genres from title\_basics

where titletype = 'movie' and startyear = 1939;

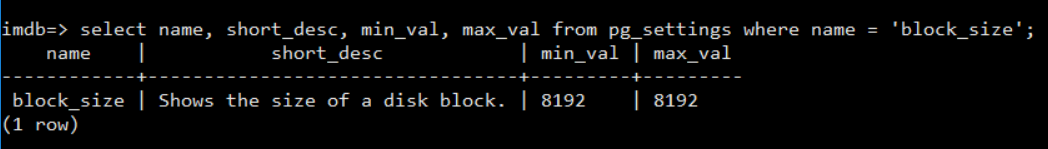
a.How big is one page of a file containing a Postgres table (Query A)?

At first, I started with the big query (Query A).

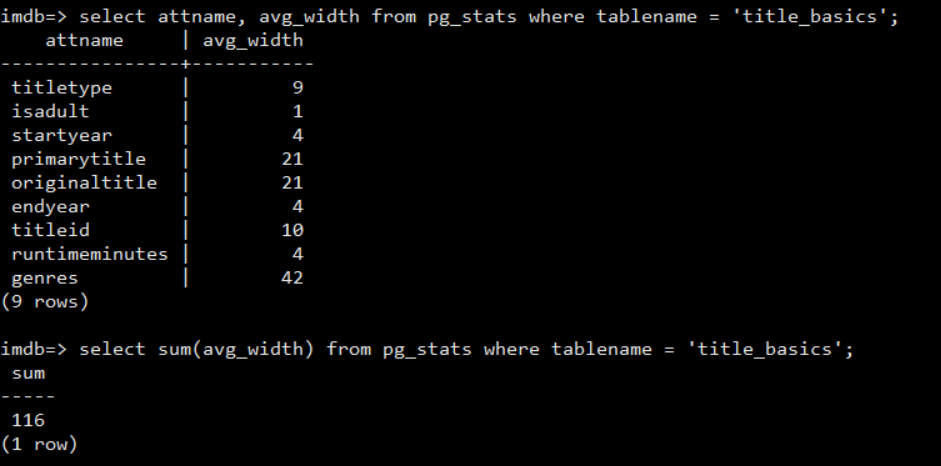
Since we are looking for an info on page size of a file, I added a ‘where’ clause to look for name = ‘block\_size’. One page of a file containing a Postgres table is 8192 bytes.

SELECT name, short\_desc, min\_val, max\_val FROM pg\_settings WHERE name = ‘block\_size’



1. What is the average size in bytes of one row of the title\_basics table (Query B)?

From Query B, I was able to get the average width for each column in title\_basics table. To get the average size in bytes of one row, I added the average width of all columns in title\_basics table, which results in 116 bytes.

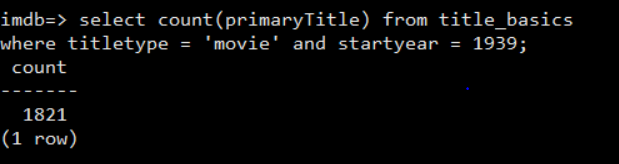


1. How many rows of title\_basics on average does one page contain,?

Since one page of a file is 8192 bytes for a table, average size in bytes of one row = 116, page header requires 20 bytes, and each row would have row offsets of 4 bytes, we can calculate the number of rows of title\_basics on average a page can contain:

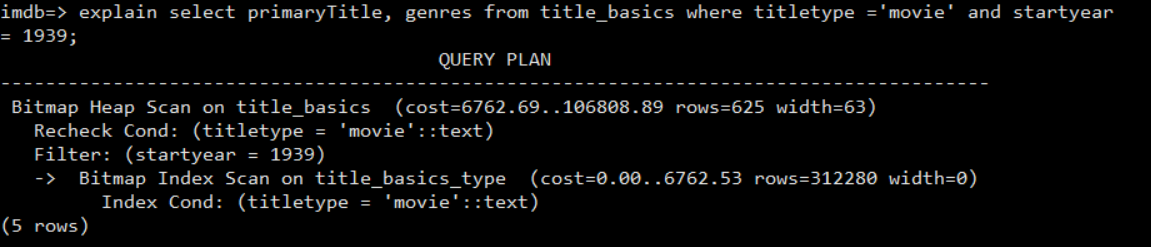
(8192 – 20)/ (116 + 4) = 68 rows/page

1. How many pages must Postgres retrieve from disk to respond to Query C? Compare your answer to the result of explain on Query C.



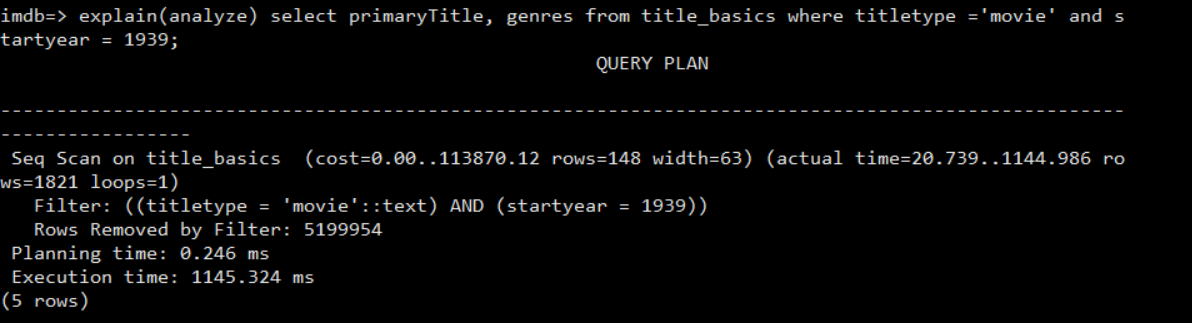
As I ran QUERY C, it returns with 1821 rows, and a page of a file for title\_basics table can contain 68 rows. Therefore, the number of pages that Postgres retrieve from the disk for Query C:

1821/68 = 26 pages

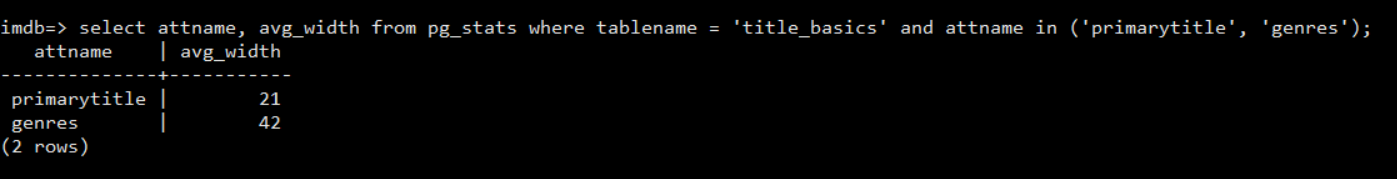


The **rows** value on the explain statement is not the same as the actual rows that are being retrieved from the QUERY C, because it is not the number of rows processed or scanned by the plan node, but rather the number emitted by the node (according to Postgres documentation). This is often less than the number scanned, as a result of filtering by any WHERE-clause conditions that are being applied at the node. In this example, when I did this query, it returned 1821 rows. This is not the number that was displayed in the ‘explain’.

With the ‘explain analyze’ we can get the actual rows.



e.Use more queries to pg\_stats to explain the rest of the numbers that appear in the result of explain on Query C.

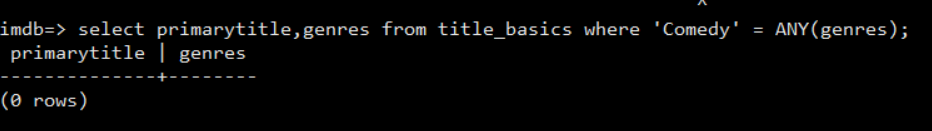


By using pg\_stats, we can see that since we only queried on primarytitle and genres, the average **width** = 63 bytes (21 + 42). The number matched with the result of explain on Query C.

1. How many movie comedies were made in 1939? (Read the Postgres documentation on the array data type.)

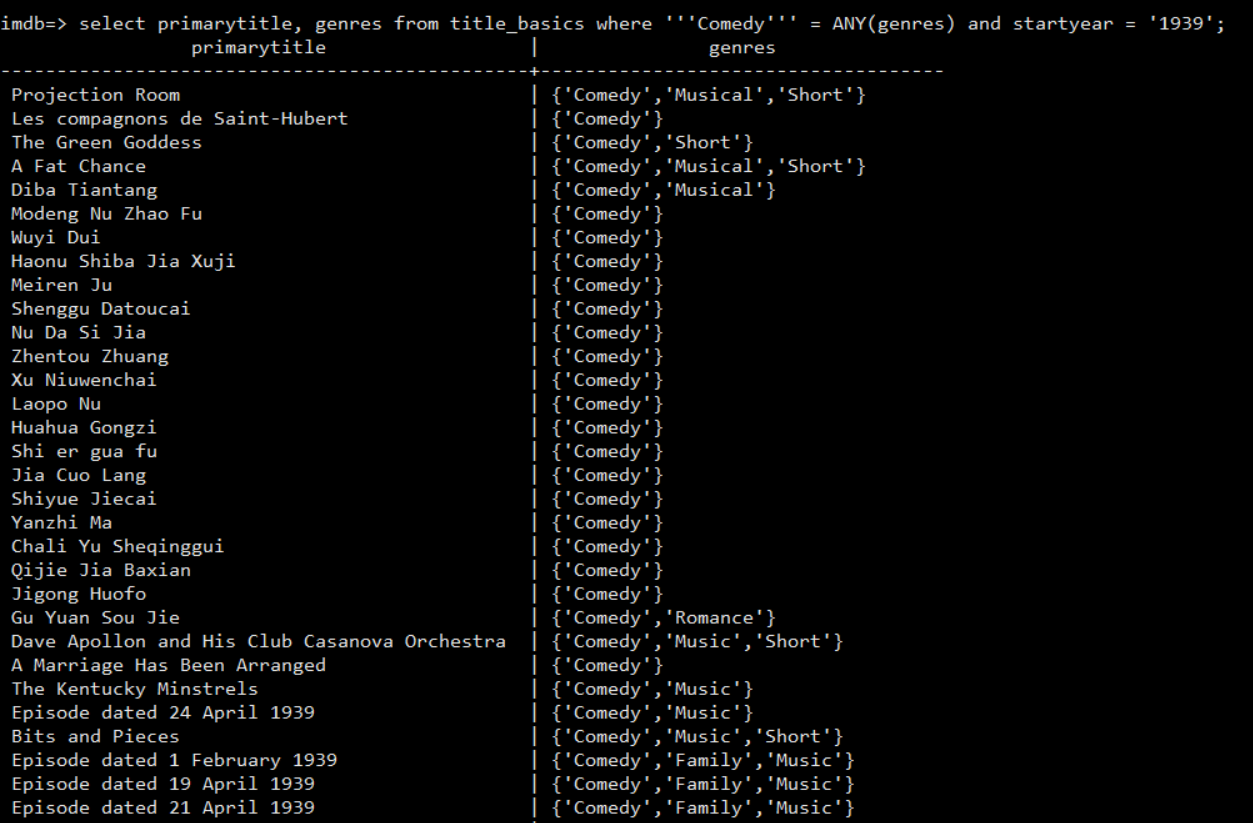
As I ran the query to find the movies with ‘Comedy’ genre that were made in 1939, it returns no rows. Since the genre column is an array datatype, I read the Postgres documentation on how to use it.

SELECT primarytitle, genres FROM title\_basics WHERE 'Comedy' = ANY(genres);



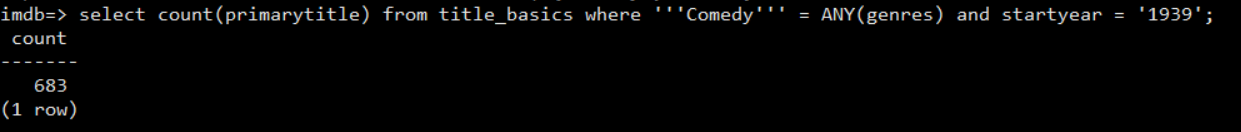
It turned out that there was ‘ ’ between the string that requires special treatment in order for the query to work.

SELECT primarytitle, genres FROM title\_basics WHERE ' ' 'Comedy' ' ' = ANY(genres) AND startyear = '1939';



Then I counted the number of movie comedies that were made in 1939:

SELECT COUNT(primarytitle) FROM title\_basics WHERE startyear = '1939';



There are 683 movie comedies that were made in 1939.

1. Notice the difference between title.basics.tsv and title.basics.braces.tsv in /usr/share/databases/IMDb.



The difference between title.basics.braces.tsv and title.basics.tsv:

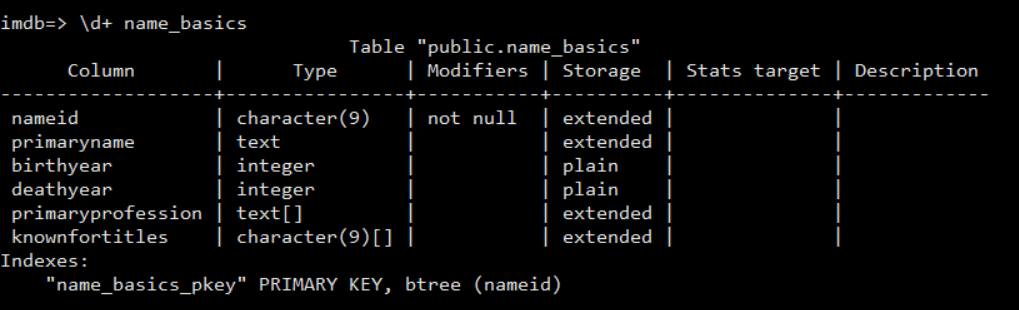
|  |  |
| --- | --- |
| Title.basics.braces.tsv | Title.basics.tsv |
|  |  |
| No header | header |
| Bigger file size:466548049 bytes | File size: 441101245 bytes |
|  |  |

a.Create a database imdb\_<your\_first\_name>

On compsci01, CREATEDB imdb\_diana



and create a name\_basics table like the one in imdb.



These are the steps to create name\_basics table similar to the one in imdb manually:

1. Create table name\_basics

CREATE TABLE name\_basics

(nameid character(9) primary key,

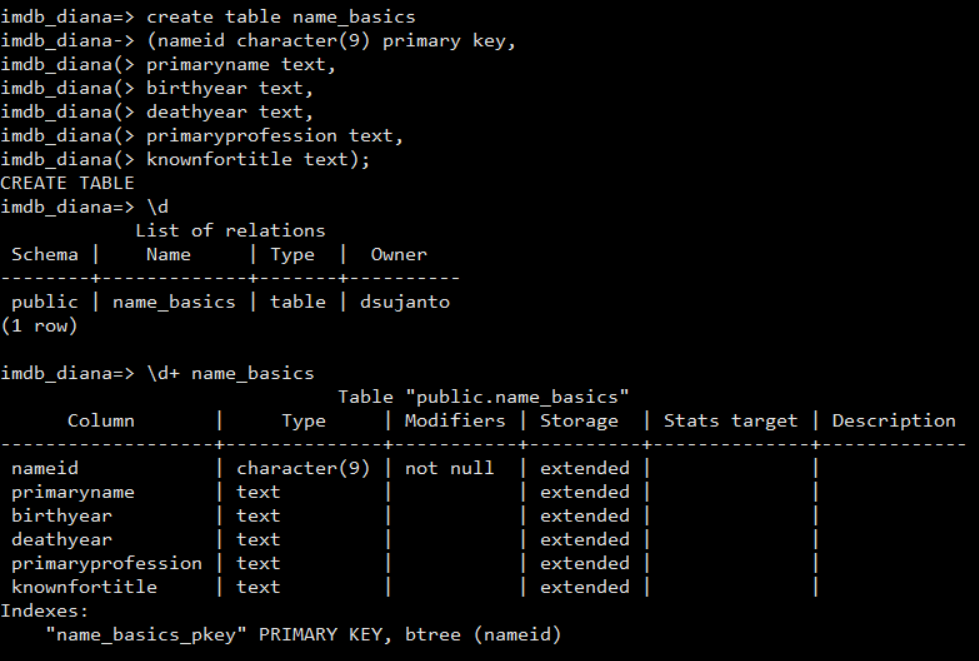
primaryname text,

birthyear text,

deathyear text,

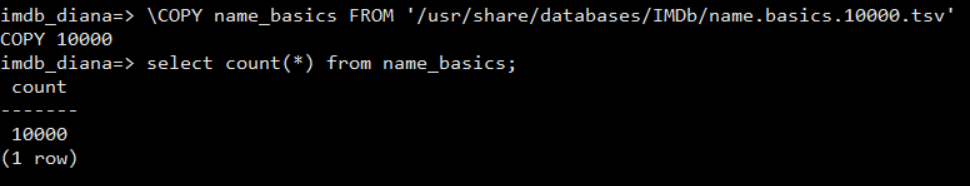
primaryprofession text,

knownfortitle text);



1. Copy name.basics.10000.tsv into name\_basics table.

\COPY name\_basics FROM '/usr/share/databases/IMDb/name.basics.10000.tsv'



1. Change the datatype on birthyear, deathyear, knownfortitle, and primaryprofession.

ALTER TABLE name\_basics

ALTER COLUMN primaryprofession TYPE text[]

USING array[primaryprofession]::text[];

ALTER TABLE name\_basics

ALTER COLUMN knownfortitle TYPE text[]

USING array[knownfortitle]::text[];

ALTER TABLE name\_basics

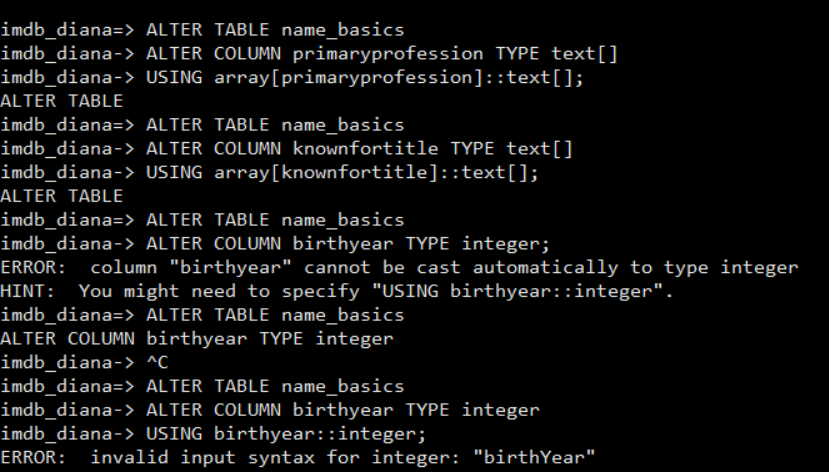
ALTER COLUMN birthyear TYPE integer

USING birthyear::integer;

ALTER TABLE name\_basics

ALTER COLUMN deathyear TYPE integer

USING deathyear::integer;



The reason that I did not put all the datatypes in place was because of the error I got when trying to copy the file with the way the table should be.

CREATE TABLE name\_basics

(nameid character(9) primary key,

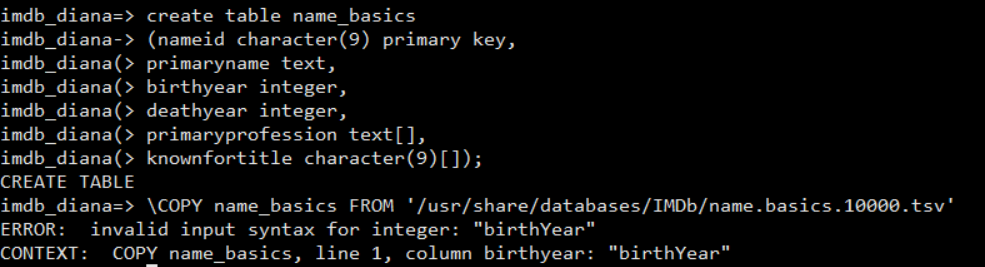
primaryname text,

birthyear integer,

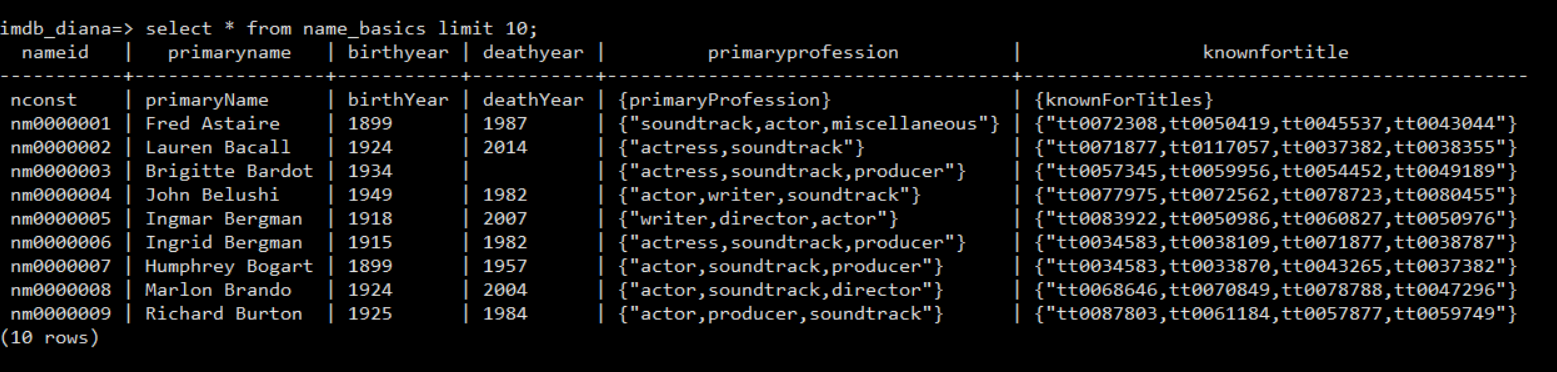
deathyear integer,

primaryprofession text[],

knownfortitle character(9)[]);

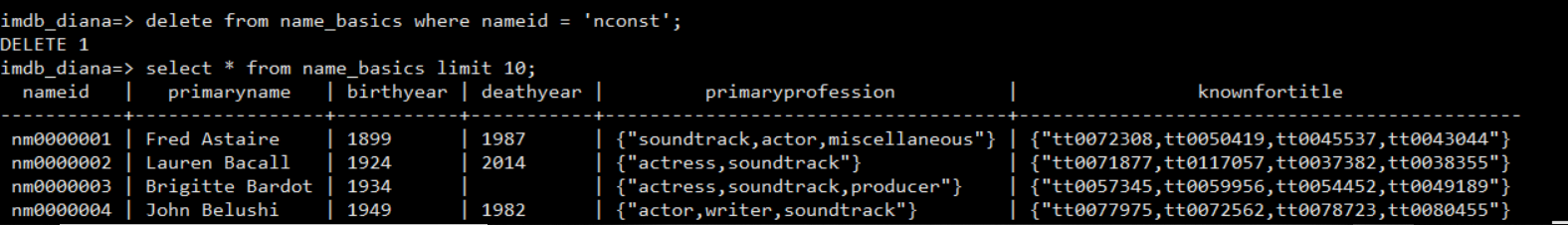


Even with the change in datatypes after loading the file, I still got an error. It turned out that the first row has header info on it. Birthyear and deathyear columns could not be integer datatypes because the headers were all character datatypes.

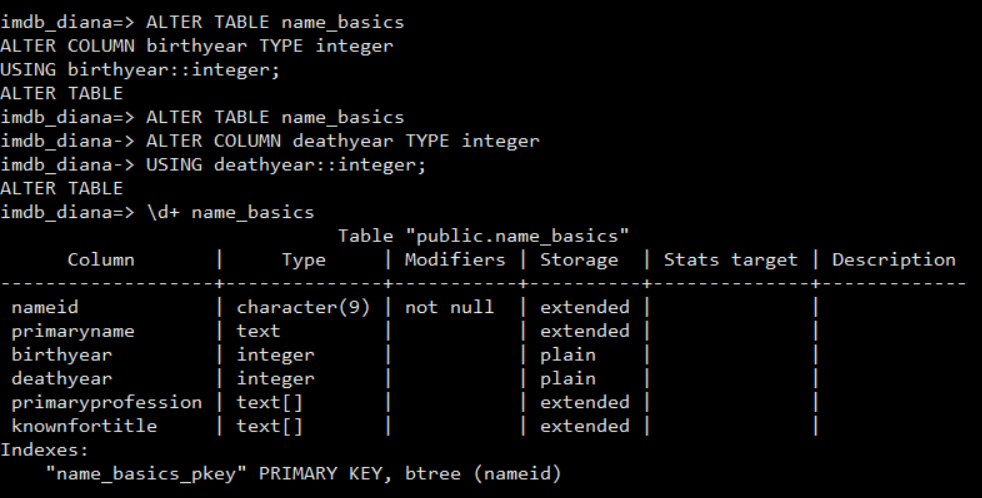


The solution is to delete the first row.

DELETE FROM name\_basics WHERE nameid = 'nconst';



Thus, I would not have any problem changing the datatypes on both columns, birthyear and deathyear. Lastly, I checked the table structure to make sure that the table is set properly. One difference between imdb.name\_basics and imdb\_diana.name\_basics is that the knownfortitle column cannot be character(9)[]. It is because it contains more than 9 characters. Therefore, I changed the datatype to text[] instead.



Write a python program to convert name.basics.10000.tsv into the form you need to \copy the file into the table.

This is the python program that I wrote and is located at

/usr/share/databases/dsujanto/hw6/tsvimport.py . This program will generate newfile.csv and substitute ‘\N’ on birthyear and deathYear into ‘’. Also, it will add {} for array columns.

**import csv**

**with open("/usr/share/databases/dsujanto/hw6/name.basics.10000.tsv",mode="r+",encoding="utf-8") as tsvfile:**

**reader = csv.reader(tsvfile, dialect='excel-tab')**

**next(reader, None)**

**with open ("/usr/share/databases/dsujanto/hw6/newfile.csv",mode="w",encoding="utf-8",newline='') as csvfile:**

**name\_writer = csv.writer(csvfile, delimiter=',',quotechar='"', quoting=csv.QUOTE\_MINIMAL,)**

**for row in reader:**

**if 'N' in row[2]:**

**row[2] = ''**

**if 'N' in row[3]:**

**row[3] = ''**

**vrow4 = '{'+row[4]+'}'**

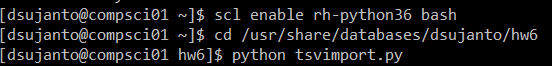
**vrow5 = '{'+row[5]+'}'**

**#print (all)**

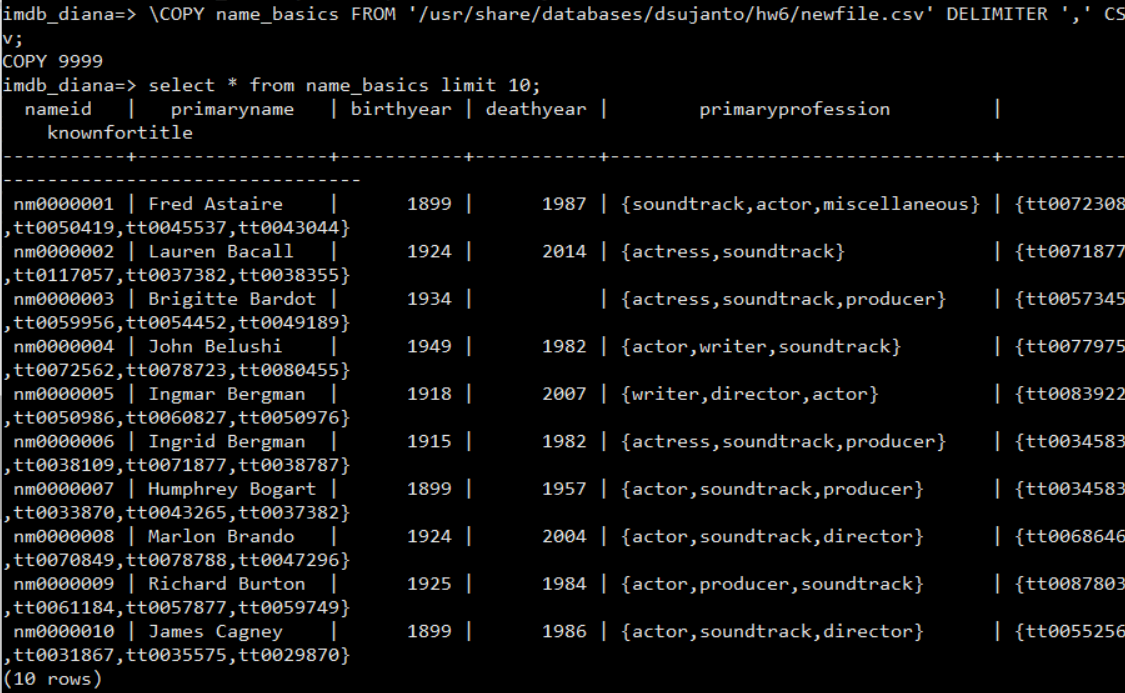
**name\_writer.writerow([row[0], row[1],row[2], row[3], vrow4, vrow5])**

**csvfile.close()**

**tsvfile.close()**



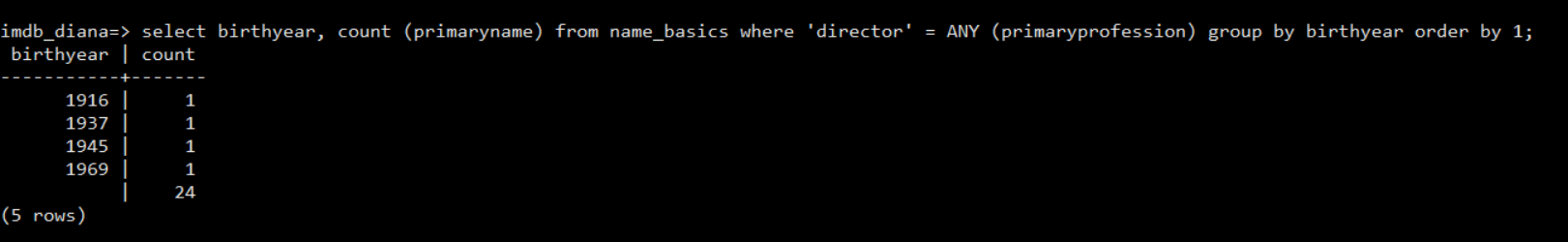
Then I copy the newfile.csv into the name\_basics table.



1. List the top five birth years of movie directors.

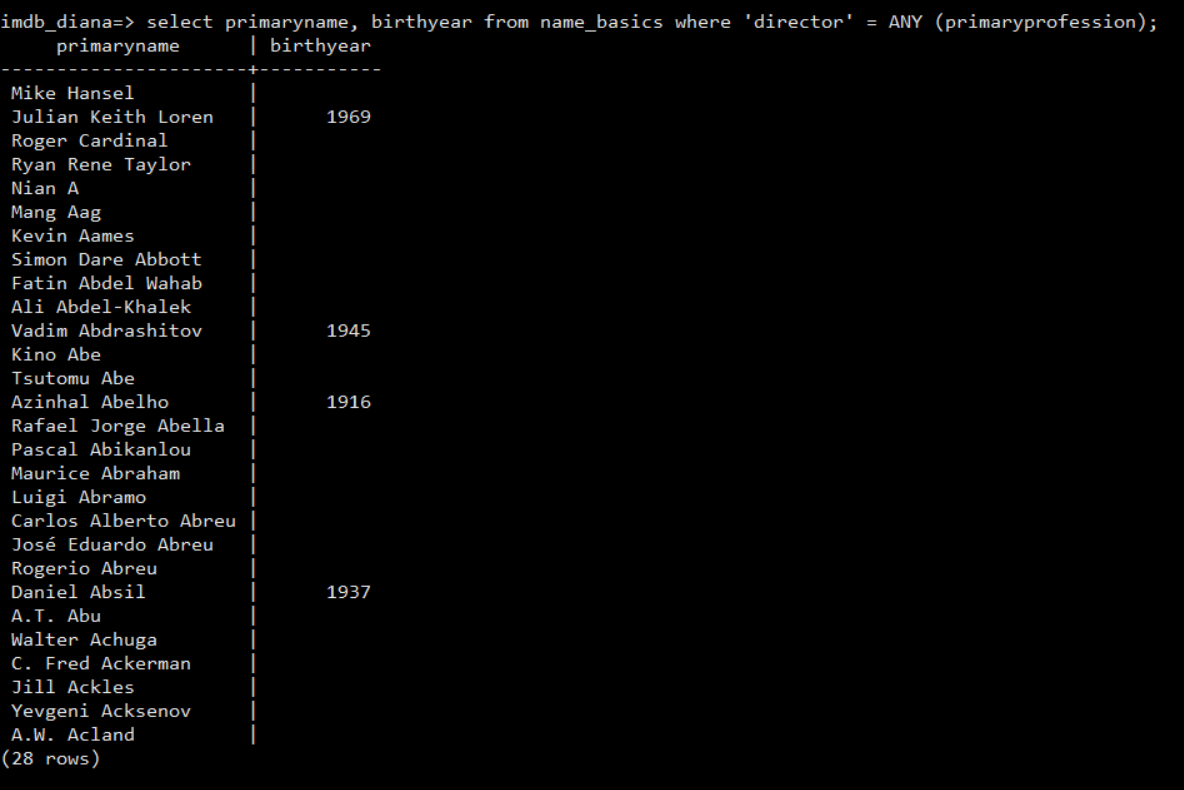
There were only four birthyear categories for movie directors: 1916, 1937, 1945, 1969 with only 1 record on each category. There were 24 records without birthyear information, which is quite interesting.

SELECT birthyear, COUNT(primaryname) FROM name\_basics WHERE 'director' = ANY (primaryprofession) GROUP BY birthyear ORDER BY 1;



To double check, I did the following query:

SELECT primaryname, birthyear FROM name\_basics WHERE ‘director’ = ANY (primaryprofession);



It is confirmed that there are only 4 out of 28 movie directors have birthyear information in name\_basics table.