

## Section One – Aggregating Data

### Step 1 – Creating Table Structure and Data

#### Creating tables



CS669/postgres@PostgreSQL 13 ▾

Query Editor Query History

```
1 CREATE TABLE Genre (  
2 genre_id DECIMAL(12) NOT NULL PRIMARY KEY,  
3 genre_name VARCHAR(64) NOT NULL  
4 );  
5  
6 CREATE TABLE Creator (  
7 creator_id DECIMAL(12) NOT NULL PRIMARY KEY,  
8 first_name VARCHAR(64) NOT NULL,  
9 last_name VARCHAR(64) NOT NULL  
10 );  
11  
12 CREATE TABLE Movie_series (  
13 movie_series_id DECIMAL(12) NOT NULL PRIMARY KEY,  
14 genre_id DECIMAL(12) NOT NULL,  
15 creator_id DECIMAL(12) NOT NULL,  
16 series_name VARCHAR(255) NOT NULL,  
17 suggested_price DECIMAL(8,2) NULL,  
18 FOREIGN KEY (genre_id) REFERENCES Genre(genre_id),  
19 FOREIGN KEY (creator_id) REFERENCES Creator(creator_id)  
20 );  
21  
22 CREATE TABLE Movie (  
23 movie_id DECIMAL(12) NOT NULL PRIMARY KEY,  
24 movie_series_id DECIMAL(12) NOT NULL,  
25 movie_name VARCHAR(64) NOT NULL,  
26 length_in_minutes DECIMAL(4),  
27 FOREIGN KEY (movie_series_id) REFERENCES Movie_series(movie_series_id)  
28 );  
29
```

Data Output Explain Messages Notifications

CREATE TABLE

Query returned successfully in 151 msec.

### Insert values

```
30  --Genre
31  INSERT INTO Genre(genre_id, genre_name)
32  VALUES(1, 'Fantasy');
33  INSERT INTO Genre(genre_id, genre_name)
34  VALUES(2, 'Family Film');
35  INSERT INTO Genre(genre_id, genre_name)
36  VALUES(3, 'Romance');
37  INSERT INTO Genre(genre_id, genre_name)
38  VALUES(4, 'Comedy');
```

---

Data Output	Explain	<u>Messages</u>	Notifications
-------------	---------	-----------------	---------------

---

INSERT 0 1

Query returned successfully in 151 msec.

```
40  --Creator
41  INSERT INTO Creator(creator_id, first_name, last_name)
42  VALUES(01, 'George', 'Lucas');
43  INSERT INTO Creator(creator_id, first_name, last_name)
44  VALUES(02, 'John', 'Lasseter');
45  INSERT INTO Creator(creator_id, first_name, last_name)
46  VALUES(03, 'John', 'Tolkien');
47  INSERT INTO Creator(creator_id, first_name, last_name)
48  VALUES(04, 'Jane', 'Doe');
```

---

Data Output	Explain	<u>Messages</u>	Notifications
-------------	---------	-----------------	---------------

---

INSERT 0 1

Query returned successfully in 106 msec.

```

50 --Movie_series
51 INSERT INTO Movie_series(movie_series_id, genre_id, creator_id, series_name, suggested_price)
52 VALUES(101, 1, 01, 'Star Wars', 129.99);
53 INSERT INTO Movie_series(movie_series_id, genre_id, creator_id, series_name, suggested_price)
54 VALUES(102, 2, 02, 'Toy Story', 22.13);
55 INSERT INTO Movie_series(movie_series_id, genre_id, creator_id, series_name, suggested_price)
56 VALUES(103, 1, 03, 'Lord of the Rings', null);
57 INSERT INTO Movie_series(movie_series_id, genre_id, creator_id, series_name, suggested_price)
58 VALUES(104, 3, 04, 'Love Story', 99.99);
59 INSERT INTO Movie_series(movie_series_id, genre_id, creator_id, series_name, suggested_price)
60 VALUES(105, 4, 04, 'Happy Moment', 88.88);
61

```

Data Output Explain Messages Notifications

INSERT 0 1

Query returned successfully in 129 msec.

```

62 --Movie
63 INSERT INTO Movie(movie_id, movie_series_id, movie_name, length_in_minutes)
64 VALUES(1001, 101, 'Episode I: The Phantom Menace', 136);
65 INSERT INTO Movie(movie_id, movie_series_id, movie_name, length_in_minutes)
66 VALUES(1002, 101, 'Episode II: Attack of the Clones', 142);
67 INSERT INTO Movie(movie_id, movie_series_id, movie_name, length_in_minutes)
68 VALUES(1003, 101, 'Episode III: Revenge of the Sith', 140);
69 INSERT INTO Movie(movie_id, movie_series_id, movie_name, length_in_minutes)
70 VALUES(1004, 101, 'Episode IV: A New Hope', 121);
71 INSERT INTO Movie(movie_id, movie_series_id, movie_name, length_in_minutes)
72 VALUES(1005, 102, 'Toy Story', 121);
73 INSERT INTO Movie(movie_id, movie_series_id, movie_name, length_in_minutes)
74 VALUES(1006, 102, 'Toy Story 2', 135);
75 INSERT INTO Movie(movie_id, movie_series_id, movie_name, length_in_minutes)
76 VALUES(1007, 102, 'Toy Story 3', 148);
77 INSERT INTO Movie(movie_id, movie_series_id, movie_name, length_in_minutes)
78 VALUES(1008, 103, 'The Lord of the Rings: The Fellowship of the Ring', 228);
79 INSERT INTO Movie(movie_id, movie_series_id, movie_name, length_in_minutes)
80 VALUES(1009, 103, 'The Lord of the Rings: The Two Towers', 235);
81 INSERT INTO Movie(movie_id, movie_series_id, movie_name, length_in_minutes)
82 VALUES(1010, 103, 'The Lord of the Rings: The Return of the King', 200);
83 INSERT INTO Movie(movie_id, movie_series_id, movie_name, length_in_minutes)
84 VALUES(1011, 104, 'Loving', 99);
85 INSERT INTO Movie(movie_id, movie_series_id, movie_name, length_in_minutes)
86 VALUES(1012, 105, 'Laughing', 88);
--

```

Data Output Explain Messages Notifications



INSERT 0 1

Query returned successfully in 91 msec.

## Step 2 – Counting Matches

– A video reseller needs to know how many movies are available that are at least two hours and fifteen minutes long. Write a single query to fulfill this request.




```
88  SELECT COUNT(movie_id) AS long_movies
89  FROM Movie
90  WHERE length_in_minutes >= 135;
```

Data Output	Explain	Messages	Notifications
 <b>long_movies</b> bigint 			
1	8		

## Step 3 – Determining Highest and Lowest

– The same video reseller needs to know the price of the most expensive and least expensive series. Write a single query that fulfills this request.

```
92  SELECT TO_CHAR(MIN(suggested_price), '$999.99') AS least_expensive,
93  TO_CHAR(MAX(suggested_price), '$999.99') AS most_expensive
94  FROM Movie_series;
```

Data Output	Explain	Messages	Notifications
 <b>least_expensive</b> text 	<b>most_expensive</b> text 		
1 \$ 22.13	\$ 129.99		

– Explain how and why the SQL processor treated the suggested price for the Lord of the Rings series differently than the other suggested price values.

Note that the suggested\_price for the Lord of the Rings series is null (has no value); therefore, they are not taken into account, because the SQL query MIN and MAX function doesn't consider the entries that are null values, the suggested price for the Lord of the Rings series will be excluded from the considerations when using MIN and MAX functions in the SQL processor. Other series all have a value for suggested\_price, so MIN (smallest value) and MAX (biggest value) will be from these series but not the Lord of the Rings series due to its null values.

#### Step 4 – Grouping Aggregate Results

– A film production company is considering purchasing the rights to extend a series, and needs to know the name of each movie series, along with the number of movies in each series. Write a single query to fulfill this request.

```
1 SELECT series_name, COUNT(DISTINCT movie_id) AS number_of_movies
2 FROM Movie_series
3 JOIN Movie ON Movie.movie_series_id = Movie_series.movie_series_id
4 GROUP BY series_name;
```

Data Output Explain Messages Notifications

	series_name character varying (255)	number_of_movies bigint
1	Happy Moment	1
2	Lord of the Rings	3
3	Love Story	1
4	Star Wars	4
5	Toy Story	3

#### for #9 and #10 data visualization with 2 measures

```
1 SELECT series_name, TO_CHAR(MIN(suggested_price), '$999.99') AS least_expensive, COUNT(DISTINCT movie_id) AS number_of_movies
2 FROM Movie_series
3 JOIN Movie ON Movie.movie_series_id = Movie_series.movie_series_id
4 GROUP BY series_name;
```

Data Output Explain Messages Notifications

	series_name character varying (255)	least_expensive text	number_of_movies bigint
1	Happy Moment	\$ 88.88	1
2	Lord of the Rings	[null]	3
3	Love Story	\$ 99.99	1
4	Star Wars	\$ 129.99	4
5	Toy Story	\$ 22.13	3

### Step 5 – Limiting Results by Aggregation

– The same film production company wants to search for genres that have at least 6 associated movies. Write a single query to fulfill this request, making sure to list only genres that have at least 6 movies, along with the number of movies for the genre.

```
1 SELECT genre_name, COUNT (movie_id) AS number_of_movies
2 FROM Movie_series
3 JOIN Genre ON Genre.genre_id = Movie_series.genre_id
4 JOIN Movie ON Movie.movie_series_id = Movie_series.movie_series_id
5 GROUP BY genre_name
6 HAVING COUNT(movie_id) >= 6;
```

Data Output Explain Messages Notifications

	genre_name character varying (64)	number_of_movies bigint	
1	Fantasy	7	

### Step 6 – Adding Up Values

– Boston University wants to offer its students a movie-binge weekend by playing every movie in a series. To make sure the series is as bingeable as possible, BU wants to be sure the series will run for at least 9 hours. Write a single query that gives this information, with useful columns.

```
8 SELECT series_name, SUM(length_in_minutes) AS how_long
9 FROM Movie_series
10 JOIN Movie ON Movie.movie_series_id = Movie_series.movie_series_id
11 GROUP BY series_name
12 HAVING SUM(length_in_minutes) > 540;
```

Data Output Explain Messages Notifications

	series_name character varying (255)	how_long numeric	
1	Lord of the Rings	663	

## Step 7 – Integrating Aggregation with Other Constructs

– A research institution requests the names of all movie series’ creators, as well as the number of “Fantasy” movies they have created (even if they created none). The institution wants the list to be ordered from most to least; the creator who created the most fantasy films will be at the top of the list, and the one with the least will be at the bottom. Write a single query that gives this information, with useful columns.

```
15 SELECT first_name || ' ' || last_name AS creator_full_name, COUNT(Genre.genre_id) AS number_of_fantasy
16 FROM Creator
17 LEFT JOIN Movie_series ON Movie_series.creator_id = Creator.creator_id
18 LEFT JOIN Movie ON Movie.movie_series_id = Movie_series.movie_series_id
19 LEFT JOIN Genre ON Genre.genre_id = Movie_series.genre_id
20 AND Genre.genre_name = 'Fantasy'
21 GROUP BY first_name, last_name
22 ORDER BY COUNT(Genre.genre_id) DESC;
```

Data Output Explain Messages Notifications

	creator_full_name text	number_of_fantasy bigint	
1	George Lucas	4	
2	John Tolkien	3	
3	John Lasseter	0	
4	Jane Doe	0	

## Section Two – Normalization and Data Visualization

### Step 8 – Creating Normalized Table Structure

- a. Identify all functional dependencies in the set of fields listed above in the spreadsheet. These can be listed in the form of: column1,column2,... → column3, column4...Make sure to explain your reasoning for the functional dependency choices.

case\_number → case\_description

- Once you know the case number, you can know the description of this case.

case\_number → plaintiff\_first\_name, plaintiff\_last\_name

- Once you know the case number, you can know who is the plaintiff in this case.

case\_number → defendant\_first\_name, defendant\_last\_name

- Once you know the case number, you can know who is the defendant in this case.

case\_number → attorney1\_first\_name, attorney1\_last\_name, (attorney2\_first\_name, attorney2\_last\_name, attorney3\_first\_name, attorney3\_last\_name)

- Once you know the case number, you can know who is/are the attorney or attorneys in this case because each case has one plaintiff and one defendant, but each plaintiff and defendant may retain multiple attorneys for each court appearance.

case\_number → appearance\_date

- Once you know the case number, you can find the court appearance dates for this case. Each case can have one or more appearance\_date, but since there can only be one court appearance per day for the case, each appearance\_date stored in the specific case only has one record.

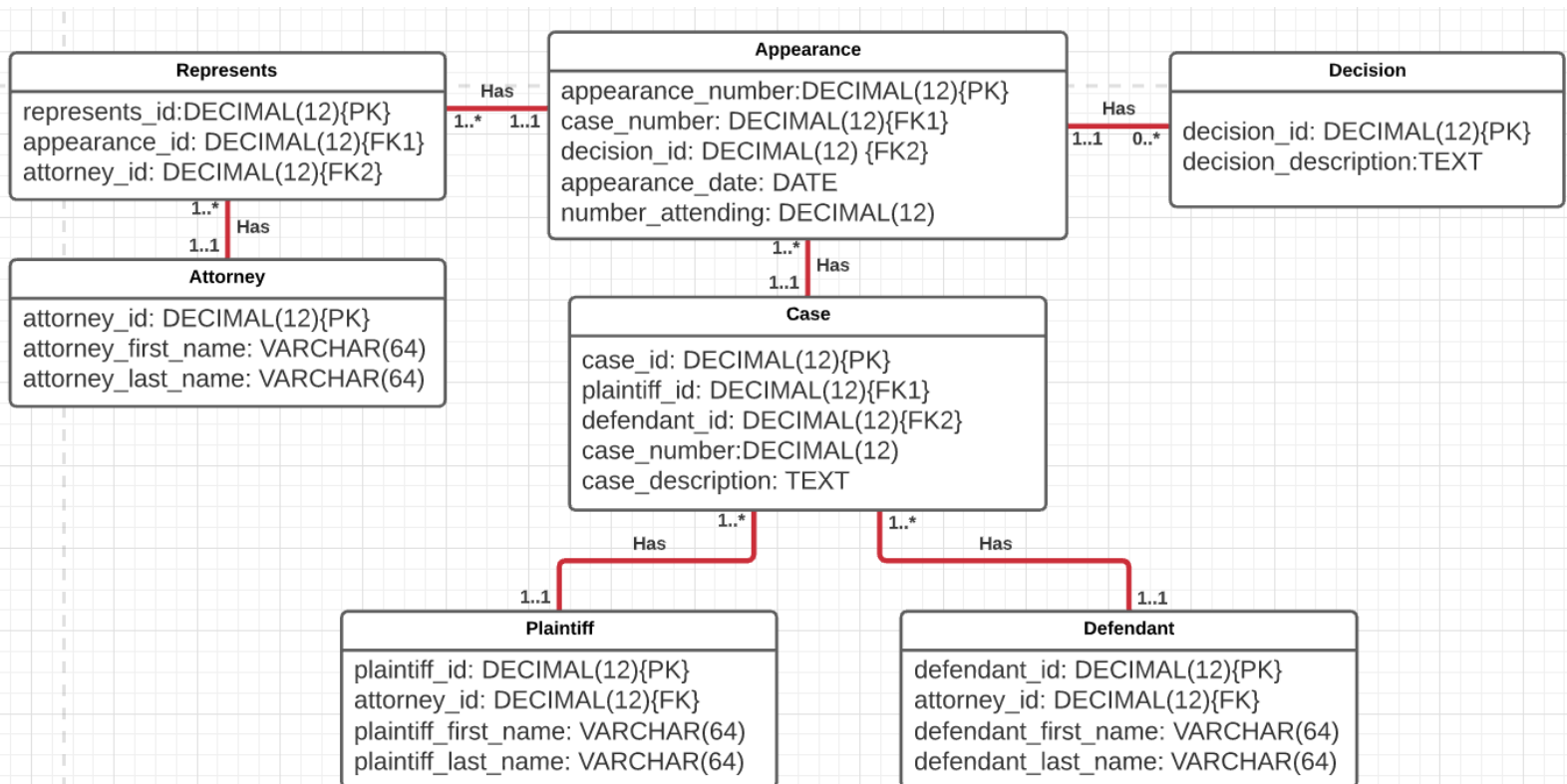
case\_number, appearance\_date → number\_attending, decision1\_description, (decision2\_description)

- Once you know the case number and its corresponding appearance\_date, you can find out the number of people attending for this court appearance, its decision or decisions (multiple decisions about the case may be made during each court appearance)

extra\_appearance\_notes → attorney1\_last\_name, attorney2\_first\_name, attorney2\_last\_name, attorney3\_first\_name, attorney3\_last\_name, decision1\_description, decision2\_description

- Once you have the extra\_appearance\_note, you can see at least three attorneys and at least two decisions are stored in this note.

b. Suggest a set of normalized relational tables derived from how the court operates and the fields they store. Create a DBMS physical ERD representing this set of tables, which contains the entities, primary and foreign keys, attributes, relationships, and relationship constraints. You may add synthetic primary keys where needed. Make sure that the tables are normalized to BCNF, and to explain your choices.



The court has many cases but each case has one plaintiff and one defendant. Using this information, I use 1..1 to indicate that each case has a single plaintiff and a single defendant. I am storing both plaintiff id and the defendant id as the foreign keys, so when checking the case number, it is easy to refer to who the plaintiff and the defendant to the case. As the people are labeled as the plaintiff and defendant, they have mandatory at least one case in the court, but



maybe with many open dockets, so I am using 1..\*. The supertype Case can not have it by itself but the case is made up with plaintiff and defendant; therefore it is totally complete, so I am using UML symbol as {mandatory} to it. The subtype plaintiff and defendant are exclusive to the role to itself, so none of them are overlapping, they are disjointed, so I am using UML symbol as {or} to it.

Because each plaintiff and defendant may retain multiple attorneys for each court appearance, and there is no specific rule stating that the plaintiff or the defendant can waive the right to have an attorney, so I am assuming under the civil right to counsel that it is mandatory to have an attorney, and maybe many attorneys, so I am using 1..\* to present it. The attorneys that are in the court system should already have a case, maybe even more than 1 case, so I am using 1..\* to show the relationship. I am storing the attorney id as the foreign key to both the plaintiff and the defendant table so it can refer to who the attorney is.

While each case may have one or more court appearances, so I am using 1..\* to represent that it is a mandatory plural relationship. While each appearance is associated with one court case, so I am using 1..1 to show that it is mandatory and singular. While there are multiple appearances on the same day but for different cases, I am using appearance id as the primary key and the case number as the foreign key to link the relationship, and each appearance id is associated with a date

There are multiple decisions about the case that may be made at each court appearance, but I am also assuming that there are no decisions made for each appearance, so it is optional and plural, I am using 0..\*. Each decision made by the court would be associated with an appearance, so I am using 1..1 for this relationship.

The extra appearance note is set to when there are more than three attorneys or more than two decisions at a court appearance, so I am using 0..1 to show that each appearance is optional to have the note if the condition is not met, or if the conditional is met, it is only one note associated to the appearance. If a note is written for a specific court appearance, it must have a court appearance and I link the appearance number as the foreign key. I am using the 1..1 to present this relationship.

My choices are based on the BCNF (Boyce-Codd Normal Form) that meet 1NF, 2NF and 3NF to meet data normalization, which is a database design technique that aims to reduce redundant (repetitive, useless) data and eliminate anomalies like insertion, update and deletion; and achieve that all stored data logically makes sense.

For 1NF, I made sure that all rows are unique, each cell contains a single value and each value can not be split down further. For 2NF, I use foreign keys to eliminate partial dependencies. For 3NF, I again divide tables and create new tables to move the transitive dependencies to a different separate table to eliminate transitive dependencies. Finally, the BCNF stage makes sure that superkeys contain candidate keys but not vice versa.

## Step 9 – Visualizing Data with One or Two Measures

```

1 SELECT series_name, TO_CHAR(MIN(suggested_price), '$999.99') AS least_expensive, COUNT(DISTINCT movie_id) AS number_of_movies
2 FROM   Movie_series
3 JOIN   Movie ON Movie.movie_series_id = Movie_series.movie_series_id
4 GROUP BY series_name;

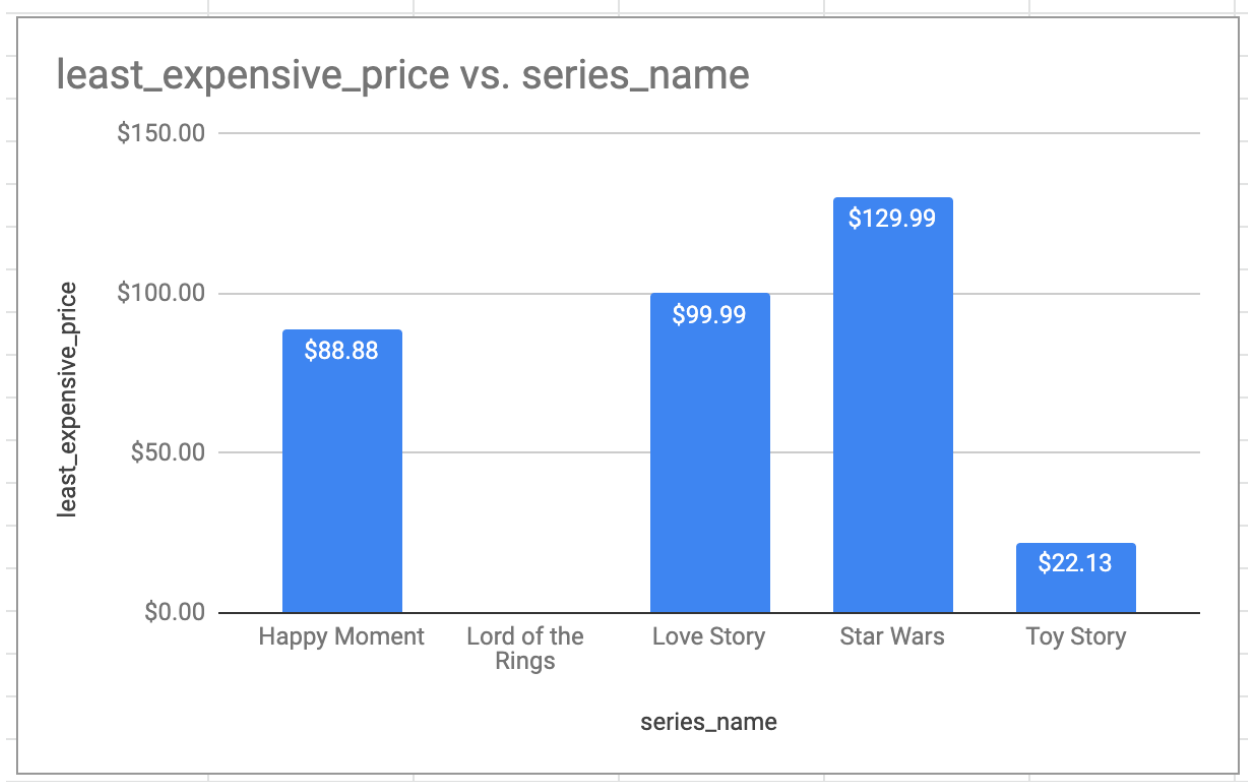
```

Data Output Explain Messages Notifications

	series_name character varying (255)	least_expensive text	number_of_movies bigint	
1	Happy Moment	\$ 88.88	1	
2	Lord of the Rings	[null]	3	
3	Love Story	\$ 99.99	1	
4	Star Wars	\$ 129.99	4	
5	Toy Story	\$ 22.13	3	

series_name	least_expensive	number_of_movies
Happy Moment	\$88.88	1
Lord of the Rings	NULL	3
Love Story	\$99.99	1
Star Wars	\$129.99	4
Toy Story	\$22.13	3

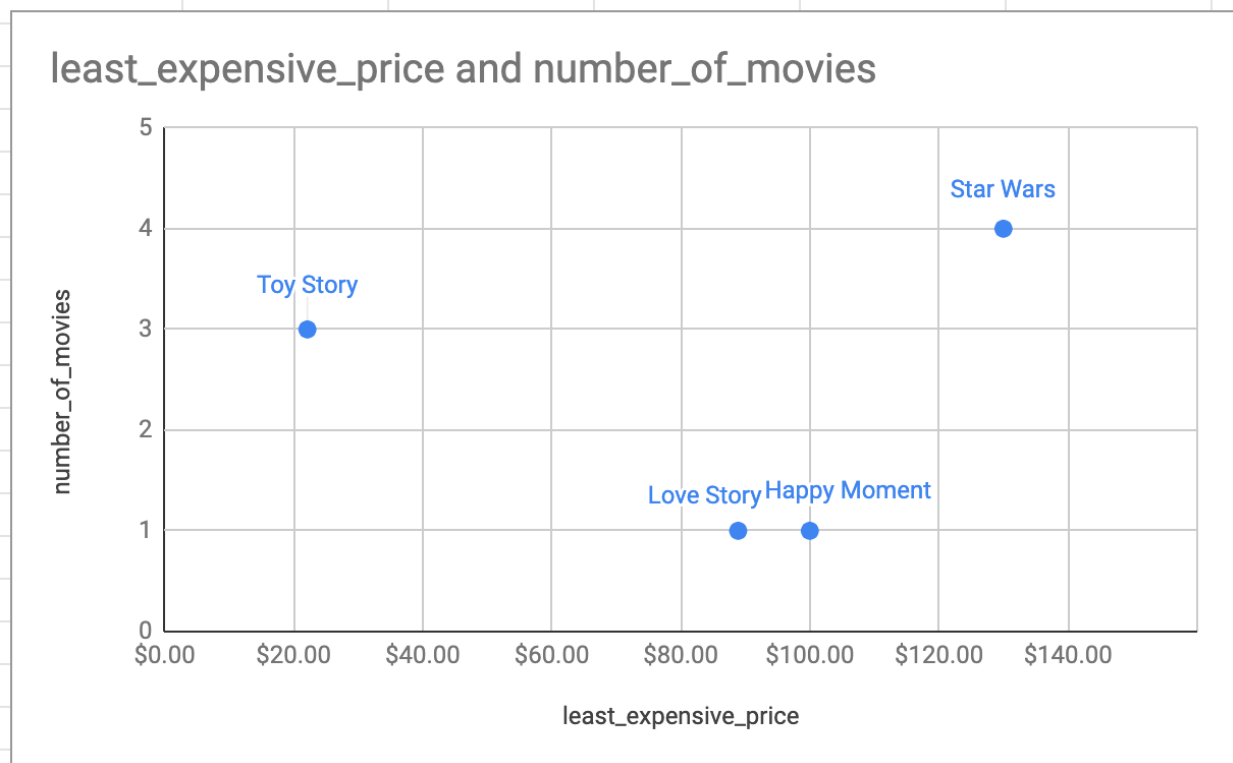
Create a bar chart with the movie name as one axis, and the series' price as another axis. Explain the story this visualization describes.



We use a bar chart because it is useful for comparing the same measure across different items of the same type. In this case, our result set only has one type of item as the series, and one measure as the least expensive price. So, the bar chart perfectly allows us to compare the least expensive price across different movie series.

I would summarize the story as follows: Although the Lord of Ring does not have any data for its price, we can still see that different series have a different least expensive price, and the price varies from \$22.13 to \$129.99. The Star Wars series has the most expensive price for its movies, and the Toy Story series has the least expensive price for its movies.

Create a scatterplot with the series' price as one axis, and the number of movies in the series as another axis. Ensure that each series is labeled with its name, either directly or with a legend. Explain the story this visualization describes.



The least\_expensive\_price is displayed in the X axis, the number\_of\_movies in the Y axis, and the series names are shown as labels in the scatterplot. The scatterplot allows us to visually compare two measures quickly, so can be a valuable tool when two measures are involved.

From a visual glance, we can see that Star Wars has the most movies - 4 movies and the most expensive least\_expensive\_price, and the Toy Story has the least least\_expensive\_price and slightly less number of movies in its series - 3 movies. Both Love Story and Happy Moment have only one movie in its series while the least expensive prices fall in between Star Wars and Toy Story. Comparing Love Story and Happy Moment, the least\_expensive\_price for Happy Moment costs more than the Love Story.

## Step 10 – Another Data Visualization

```

1 SELECT movie_name, length_in_minutes, Movie_series.suggested_price
2 FROM Movie
3 JOIN Movie_series ON Movie_series.movie_series_id = Movie.movie_series_id

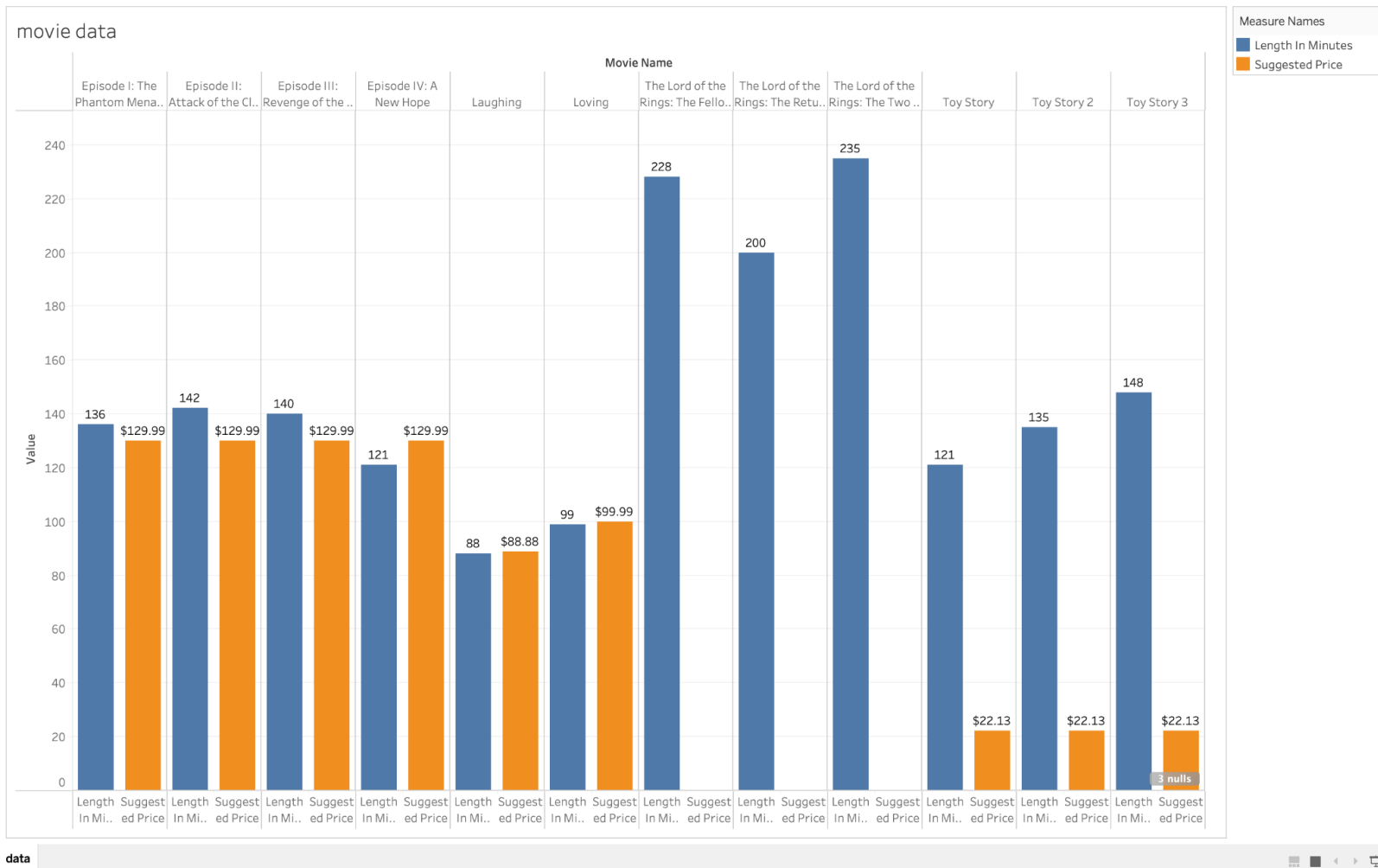
```

[Data Output](#) [Explain](#) [Messages](#) [Notifications](#)

	movie_name character varying (64)	length_in_minutes numeric (4)	suggested_price numeric (8,2)
1	Episode I: The Phantom Men...	136	129.99
2	Episode II: Attack of the Clo...	142	129.99
3	Episode III: Revenge of the S...	140	129.99
4	Episode IV: A New Hope	121	129.99
5	Toy Story	121	22.13
6	Toy Story 2	135	22.13
7	Toy Story 3	148	22.13
8	The Lord of the Rings: The F...	228	[null]
9	The Lord of the Rings: The T...	235	[null]
10	The Lord of the Rings: The R...	200	[null]
11	Loving	99	99.99
12	Laughing	88	88.88

movie_name	length_in_minut	suggested_price
Episode I: The P	136	129.99
Episode II: Attac	142	129.99
Episode III: Reve	140	129.99
Episode IV: A Ne	121	129.99
Toy Story	121	22.13
Toy Story 2	135	22.13
Toy Story 3	148	22.13
The Lord of the f	228	NULL
The Lord of the f	235	NULL
The Lord of the f	200	NULL
Loving	99	99.99
Laughing	88	88.88

## Using Tableau Public as the data visualization software tool



### Explain the data story, and to explain why you chose that particular chart or visualization.

I am using the side-by-side bar to compare the two measure values, the length\_in\_minutes (in blue color) with its label in minutes, and the suggested\_price (in orange color) with its label in dollars along with all the movies in the database. The movie name and the measure names are displayed in the X axis, while the numeral measure values displayed in the Y axis.

I would summarize the story as follows:

The Lord of the Rings: The Two Towers has the longest length of play time as of 235 minutes, the Lord of the Rings: The Fellowship of the Ring has the second longest length as of 228 minutes, and The Lord of the Rings: The Return of the King has the third longest length as of 200 minutes among all the movies in the list, but they all have no information for their prices.

All Toy Story, Toy Story 2 and Toy Story 3 have the same price of \$22.13, and they are the cheapest movies in this movie list.

All Episode I: The Phantom Menace, Episode II: Attack of the Clones, Episode III: Revenge of the Sith, and Episode IV: A New Hope have the same price of \$129.99, and they are the most expensive movies in this movie list.

The movie Laughing is the shortest movie in the list with 88 mins, followed by Loving, 99 mins.

Side-by-side bar is a great tool for comparing things between different groups, and it allows us to visually compare two measures quickly. I chose this particular visualization because it is very clear to see the price and the length of the movies in two colored side bars next to each other with labels on top and the whole name of the movies as the column. If I want to choose a movie to watch based on this chart, it helps me easily to see the length of the movie with its corresponding price as well as the name of the movie.

### Present data information in Python

```
data.py > ...
1  import csv
2
3  with open('data-2.csv', 'r') as file:
4      reader = csv.reader(file, delimiter=",")
5      header = next(reader)
6      for row in reader:
7          name = row[0]
8          minutes = row[1]
9          price = row[2]
10         line = 'The movie name is \'{}\'' '\n\t--> this movie has the length of {} in
11             minutes, and the price is ${}.'.format(name, minutes, price)
12         print(line)
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL 1: terminal-tools

```
Hanwens-Air-2:lab3 hanwenzhang$ python data.py
The movie name is 'Episode I: The Phantom Menace'
--> this movie has the length of 136 in minutes, and the price is $129.99.
The movie name is 'Episode II: Attack of the Clones'
--> this movie has the length of 142 in minutes, and the price is $129.99.
The movie name is 'Episode III: Revenge of the Sith'
--> this movie has the length of 140 in minutes, and the price is $129.99.
The movie name is 'Episode IV: A New Hope'
--> this movie has the length of 121 in minutes, and the price is $129.99.
The movie name is 'Toy Story'
--> this movie has the length of 121 in minutes, and the price is $22.13.
The movie name is 'Toy Story 2'
--> this movie has the length of 135 in minutes, and the price is $22.13.
The movie name is 'Toy Story 3'
--> this movie has the length of 148 in minutes, and the price is $22.13.
The movie name is 'The Lord of the Rings: The Fellowship of the Ring'
--> this movie has the length of 228 in minutes, and the price is $NULL.
The movie name is 'The Lord of the Rings: The Two Towers'
--> this movie has the length of 235 in minutes, and the price is $NULL.
The movie name is 'The Lord of the Rings: The Return of the King'
--> this movie has the length of 200 in minutes, and the price is $NULL.
The movie name is 'Loving'
--> this movie has the length of 99 in minutes, and the price is $99.99.
The movie name is 'Laughing'
--> this movie has the length of 88 in minutes, and the price is $88.88.
Hanwens-Air-2:lab3 hanwenzhang$
```