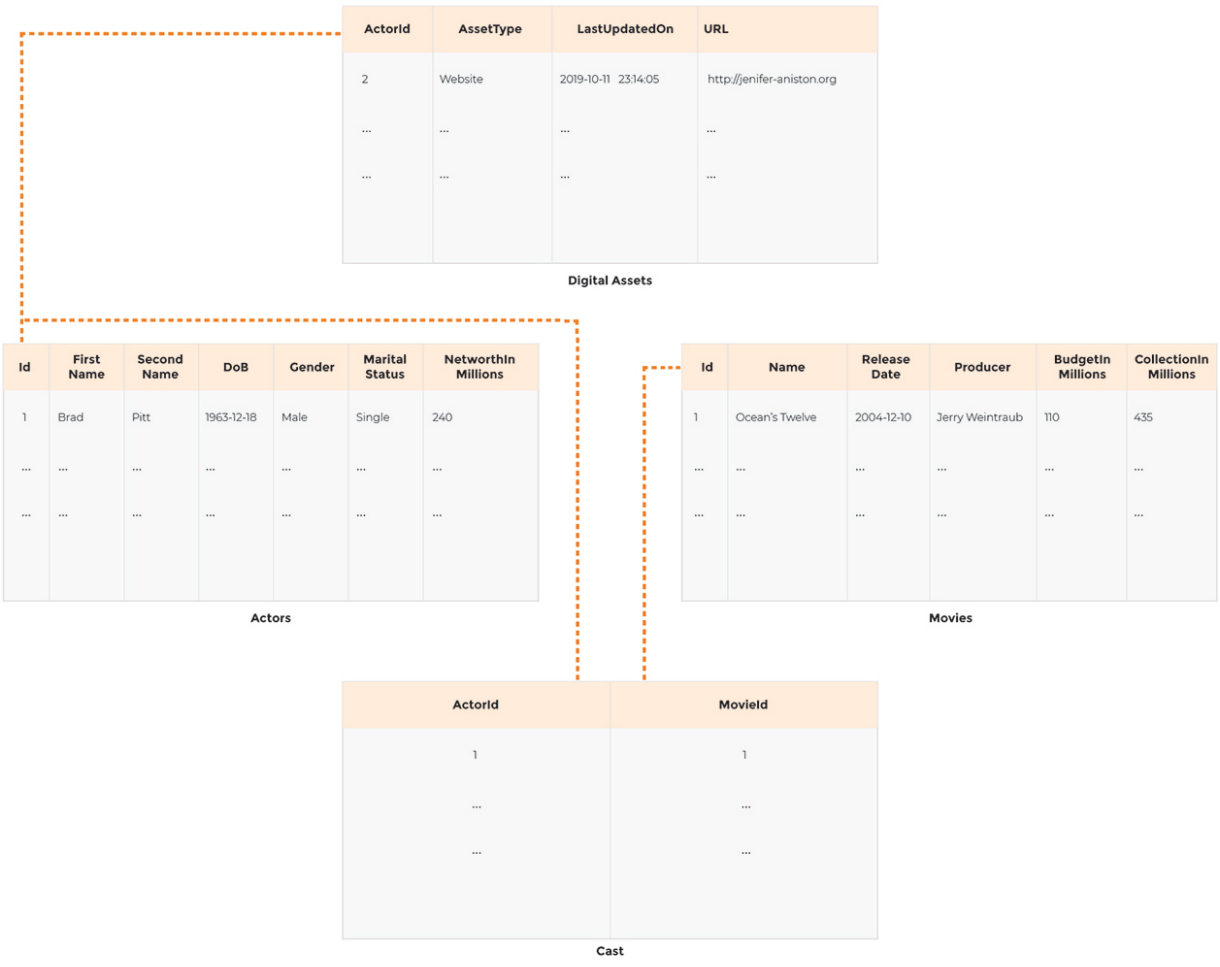
Solution Practice Set 4

Solution Practice Set 4

The database relationship model is reprinted below for reference.



Connect to the terminal below by clicking in the widget. Once connected, the command line prompt will show up. Enter or copy and paste the command **./DataJek/Lessons/quiz.sh** and wait for the MySQL prompt to

start-up.

```
-- The lesson queries are reproduced below for convenient copy/paste into the terminal.

-- Question # 1, Query 1
SELECT Producer
FROM Movies
GROUP BY Producer;

-- Question # 1, Query 2
SELECT Producer
FROM Movies
GROUP BY Producer
HAVING COUNT(Producer) > 1;

-- Question # 1, Query 3
SELECT Producer AS Producer_Name, AVG(CollectionInMillions) AS Average_Collection_In_Millions
FROM Movies
GROUP BY Producer
HAVING COUNT(Producer) > 1;

-- Question # 2, Query 1
SELECT CONCAT (FirstName, " ", SecondName) AS Actors, MovieId, Producer
FROM Actors JOIN Cast
ON Actors.Id = Cast.ActorId
JOIN Movies
ON Cast.MovieId = Movies.Id;

-- Question # 2, Query 2
SELECT CONCAT (FirstName, " ", SecondName) AS Actors, MovieId, Producer
FROM Actors JOIN Cast
ON Actors.Id = Cast.ActorId
JOIN Movies
ON Cast.MovieId = Movies.Id
AND Producer <> 'Ryan Seacrest';

-- Question # 2, Query 3
SELECT DISTINCT(CONCAT (FirstName, " ", SecondName)) AS Actors_Who_Have_Not_Worked_with_Ryan_
FROM Actors  JOIN Cast
ON Actors.Id = Cast.ActorId
JOIN Movies
ON Cast.MovieId = Movies.Id
AND Producer <> 'Ryan Seacrest';

-- Question # 2, Query 4
SELECT c.ActorID, c.MovieId, m.Producer
FROM Cast c, Movies m
WHERE c.MovieId = m.Id;

-- Question # 2, Query 5
SELECT c.ActorID, c.MovieId, m.Producer
FROM Cast c, Movies m
WHERE c.MovieId = m.Id
  AND m.Producer <> 'Ryan Seacrest';

-- Question # 2, Query 6
SELECT DISTINCT(CONCAT (a.FirstName, " ", a.SecondName)) AS Actors_Who_Have_Not_Worked_with_R
FROM Cast c, Movies m, Actors a
WHERE c.MovieId = m.Id
  AND m.Producer <> 'Ryan Seacrest'
```

```
  AND c.ActorId = a.Id;

-- Question # 3, Query 1

SELECT ActorId, AssetType, LastUpdatedOn
FROM DigitalAssets
ORDER BY ActorId ASC,
         LastUpdatedOn DESC;

-- Question # 3, Query 2
SELECT ActorId, MAX(LastUpdatedOn)
FROM DigitalAssets
GROUP BY ActorId;

-- Question # 3, Query 3
SELECT ActorId, AssetType, LastUpdatedOn
FROM DigitalAssets
WHERE (ActorId, LastUpdatedOn) IN
                               (SELECT ActorId, MAX(LastUpdatedOn)
                                FROM DigitalAssets
                                GROUP BY ActorID);

-- Question # 3, Query 4
CREATE TABLE DigitalActivityTrack (
Id INT AUTO_INCREMENT NOT NULL PRIMARY KEY,
Actor_Id INT NOT NULL,
Digital_Asset VARCHAR(20) NOT NULL,
Last_Updated_At DATETIME Not NULL DEFAULT NOW()
);

-- Question # 3, Query 5
INSERT INTO DigitalActivityTrack (Actor_Id, Digital_Asset, Last_Updated_At)
SELECT ActorId, AssetType, LastUpdatedOn FROM DigitalAssets
                WHERE (ActorId, LastUpdatedOn) In
                               (SELECT ActorId, MAX(LastUpdatedOn) FROM DigitalAssets
                                GROUP BY ActorID)
             ORDER BY LastUpdatedOn DESC;

-- Question # 3, Query 6
SELECT CONCAT(a.FirstName, " ", a.SecondName) AS Actor_Name, Digital_Asset, Last_Updated_At
FROM Actors a, DigitalActivityTrack
WHERE a.Id = Actor_Id;

-- Question # 4, Query 1
SELECT CONCAT (FirstName, " ", SecondName) AS Actor_Name, NetWorthInMillions AS 3rd_Lowest_Ne
From Actors a1
WHERE 2 = (SELECT COUNT(DISTINCT (NetWorthInMillions))
           FROM Actors a2
           WHERE a2. NetWorthInMillions < a1. NetWorthInMillions);

-- Question # 5, Query 1
SELECT ActorID, COUNT(ActorId)
FROM DigitalAssets
GROUP BY ActorId;

-- Question # 5, Query 2
SELECT ActorID, GROUP_CONCAT(AssetType)
FROM DigitalAssets
GROUP BY ActorId;

-- Question # 5, Query 3
SELECT CONCAT (FirstName, " ", SecondName) AS Actor_Name,
       GROUP_CONCAT(AssetType) AS Digital_Assets
```

```
FROM Actors INNER JOIN DigitalAssets
ON Actors.Id = DigitalAssets.ActorId
GROUP BY Id;

-- Question # 5, Query 4
SELECT CONCAT (FirstName, " ", SecondName) AS Actor_Name,
       GROUP_CONCAT(AssetType) AS Digital_Assets
FROM Actors LEFT JOIN DigitalAssets
ON Actors.Id = DigitalAssets.ActorId
GROUP BY Id;
```

Terminal

Question # 1

Write a query to display the average collection in millions of producers who have produced more than one movie.

To answer this question, we need to shortlist only those producers whose name appears more than once in the **Movies** table. This is hinting towards grouping the results by producers:

```
SELECT Producer
FROM Movies
GROUP BY Producer;
```

Since we are only interested in producers who have produced more than one movie, we will add a restriction on the number of times a producer's name appears in the **Movies** table. The **HAVING** clause used with the **GROUP BY** clause will give us the desired result.

```
SELECT Producer
FROM Movies
GROUP BY Producer
HAVING COUNT(Producer) > 1;
```

We are now left with two produces who have multiple movies. The last

step is to find the average collection in millions of the films by these two producers:

```
SELECT Producer AS Producer_Name, AVG(CollectionInMillions) AS Averag
e_Collection_In_Millions
FROM Movies
GROUP BY Producer
HAVING COUNT(Producer) > 1;
```

Question # 2

Find all those actors who have not worked with producer Ryan Seacrest.

Approach 1: Joining three tables in a single SQL query.

The information on actors, movies and producers is scattered in three tables; **Actors**, **Cast** and **Movies**. We need to join the tables together to find the answer. Joining three tables in a single SQL query can be a tricky concept. The first table is related to the second table and the second table is related to the third table. In our case the **Actors** table is related to the **Cast** table and the **Cast** table is related to the **Movies** table. We want the names of the actors from the **Actors** table and information on producers from the **Movies** table.

The **Cast** table joins the two tables **Actors** and **Movies** together and contains the primary key of both tables. The primary key (PK) of the **Actors** table is a foreign key (FK) in the **Cast** table. Similarly, the primary key of the **Movies** table is the foreign key in the **Cast** table. Understanding these table relationships is the key to joining multiple tables in a single MySQL query.

The basic syntax of joining three tables in MySQL is as follows:

```
SELECT table1.col, table2.col
```

SELECT **table1.col**, **table3.col**

FROM **table1** JOIN **table2**

ON **table1.PK** = **table2.FK**

JOIN **table3**

ON **table2.FK** = **table3.PK**

First we join table1 and table2 and then the combined data of the two tables is joined with table3. Here table2 is the table that connects table1 and table3. In our case this will translate to the following query:

```
SELECT CONCAT (FirstName, " ", SecondName) AS Actors, MovieId, Produc
er

FROM Actors JOIN Cast
ON Actors.Id = Cast.ActorId

JOIN Movies
ON Cast.MovieId = Movies.Id;
```

This query joins the **Actors** and **Movies** table using the connecting **Cast** table. We need only those actors who have not worked with producer Ryan Seacrest. Elinimation of the unwanted rows can be accomplished by adding a condition at the end of our multiple join query as:

```
SELECT CONCAT (FirstName, " ", SecondName) AS Actors, MovieId, Produc
er

FROM Actors JOIN Cast
ON Actors.Id = Cast.ActorId

JOIN Movies
ON Cast.MovieId = Movies.Id

AND Producer <> 'Ryan Seacrest';
```

The query now gives us 10 rows after removing three rows with Ryan Seacrest as producer. A glance at the results shows that some actors have worked in multiple movies and thus their name appears more than once. Since we are only interested in the names of those actors who have not worked with a particular producer, we need to apply **DISTINCT** clause. The final query looks like this:

```
SELECT DISTINCT(CONCAT (FirstName, " ", SecondName)) AS Actors_Who_Ha
ve_Not_Worked_with_Ryan_Seacrest

FROM Actors  JOIN Cast
ON Actors.Id = Cast.ActorId

JOIN Movies
ON Cast.MovieId = Movies.Id

AND Producer <> 'Ryan Seacrest';
```

Approach 2: Without using joins.

The information required to answer this question is scattered across three tables; **Actors**, **Cast** and **Movies**. First we will find the producers of the movies by combining data from **Cast** and **Movies** tables on the common column, movie ID.

```
SELECT c.ActorID, c.MovieId, m.Producer
FROM Cast c, Movies m
WHERE c.MovieId = m.Id;
```

The query adds the producers to each row of the **Cast** table. Next step is to find those actors who have not worked with producer Ryan Seacrest. This can be done by adding a condition Producer <> 'Ryan Seacrest'.

```
SELECT c.ActorID, c.MovieId, m.Producer
FROM Cast c, Movies m
WHERE c.MovieId = m.Id
```

```
  AND m.Producer <> 'Ryan Seacrest';
```

We are left with 10 rows now and it can be seen that some actors are appearing multiple times having acted in more than one movie. The last step is to display the names of the actors after removing duplicate entries. The **DISTINCT** clause will remove duplicates. For the actor names, we will add columns from **Actors** table to the query created so far:

```
SELECT DISTINCT(CONCAT (a.FirstName, " ", a.SecondName)) AS Actors_Wh
o_Have_Not_Worked_with_Ryan_Seacrest

FROM Cast c, Movies m, Actors a

WHERE c.MovieId = m.Id
  AND m.Producer <> 'Ryan Seacrest'
  AND c.ActorId = a.Id;
```

Question # 3

Populate a table DigitalActivityTrack with the last digital activity of each actor along with the asset type on which the activity occurred.

Every actor has multiple digital assets which they update at different times. We need to find the digital asset which was last updated by the actor.

We need to track the update time of the digital assets. This information is contained in the **DigitalAssets** table. Since an actor has multiple digital assets, we need to group the rows of the **DigitalAssets** table according to **ActorId**.

```
SELECT ActorId, AssetType, LastUpdatedOn
FROM DigitalAssets
```

```
ORDER BY ActorId ASC,
         LastUpdatedOn DESC;
```

This query orders the table according to actors. We have sorted the **LastUpdatedOn** column in descending order because we want to track the last digital activity. Since we need the time when an actor last updated a digital asset, we are interested in the first row for each actor from the output shown above. We will use the **MAX** function to limit the output so that only the first row of each group is displayed.

```
SELECT ActorId, MAX(LastUpdatedOn)
FROM DigitalAssets
GROUP BY ActorId;
```

Here we find the latest digital activity of the actors but we cannot display the asset type when using the aggregate function in the **SELECT** clause. For this purpose we need to use the **IN** operator to move the aggregate function to the **WHERE** clause as follows:

```
SELECT ActorId, AssetType, LastUpdatedOn
FROM DigitalAssets
WHERE (ActorId, LastUpdatedOn) IN
                                (SELECT ActorId, MAX(LastUpdatedOn)
                                 FROM DigitalAssets
                                 GROUP BY ActorID);
```

The output shows that we need an **ORDER BY** clause to sort the output based on **LastUpdatedOn**. Now that we have the required columns, we need to create table **DigitalActivityTrack** that contains the actor name, the name of the digital asset and the time it was last updated.

```
CREATE TABLE DigitalActivityTrack (
Id INT AUTO_INCREMENT NOT NULL PRIMARY KEY,
Actor_Id INT NOT NULL,
Digital_Asset VARCHAR(20) NOT NULL,
Last_Updated_At DATETIME Not NULL DEFAULT NOW()
);
```

To populate the **DigitalAcitvityTrack** table with the query created above, we will use the **INSERT INTO SELECT** statement as follows:

```
INSERT INTO DigitalActivityTrack (Actor_Id, Digital_Asset, Last_Updat
ed_At)
SELECT ActorId, AssetType, LastUpdatedOn FROM DigitalAssets
                WHERE (ActorId, LastUpdatedOn) In
                                    (SELECT ActorId, MAX(LastUpdatedOn) F
ROM DigitalAssets
                                     GROUP BY ActorID)
             ORDER BY LastUpdatedOn DESC;
```

As a final step, we will display the results with actor names:

```
SELECT CONCAT(a.FirstName, " ", a.SecondName) AS Actor_Name, Digital_
Asset, Last_Updated_At
FROM Actors a, DigitalActivityTrack
WHERE a.Id = Actor_Id;
```

Question # 4

Find the actor with the third lowest Net Worth in Millions without using the LIMIT clause.

This is practice for a widely asked interview question where the candidate is required to find the nth highest value of a column where n could be any number greater than 1. So for example, you could be asked to find the 3rd highest salary of an employee. We have twisted this question to find the 3rd lowest net worth. The simplest way to solve this problem is by using the **IN** clause but that solution does not scale up well as the value of n increases. A more efficient way to solve this question by using a correlated sub-query.

A **correlated** sub-query is much like a recursive function where the inner query is executed for every row of the outer query. This is because the data returned by the inner query is compared to the row from the outer query at each step. This makes correlated queries slow and in general they are avoided unless there is no better way to solve the problem.

Mapping this concept of correlated queries to our problem, we need to compare the **NetWorthInMillions** returned by the outer query with other values of net worth to find out exactly how many values are lower than this value.

To generalize this concept, we can use the following syntax to find the nth lowest value:

> SELECT **col1**, **col2**
>
> From **table_name t1**
>
> WHERE **N** – 1 =
>
> (SELECT COUNT (DISTINCT (**col2**))
>
> FROM **table_name t2**
>
> WHERE **t2.col2** < **t1.col2**)

If we are finding the 2nd lowest value, then N will be 2. If we are asked to find the 4th lowest value then N will be 4.The first row of the **Actors** table has **NetWorthInMillions** equal to 240. This value is plugged in the correlated sub-query and the sub-query is then evaluated to find out how many values are lower than 240. The sub-query becomes:

```
SELECT COUNT(DISTINCT (NetWorthInMillions))
FROM Actors t2
WHERE t2.NetWorthInMillions < 240
```

The **DISTINCT** keyword will remove any duplicate net worth values. If **COUNT** returns 2, that means there are two values lower than 240 which makes it the third lowest value. If the answer is not 2, then we move on the second row of the **Actors** table and repeat the same process till we

find a value for **NetWorthInMillions** which has exactly two values lower than it. This explains why we have used **N-1** in the **WHERE** clause.

Based on the above discussion, to find the actor with the third lowest net worth we will use the following query:

```
SELECT CONCAT (FirstName, " ", SecondName) AS Actor_Name, NetWorthInM
illions AS 3rd_Lowest_Net_Worth_In_Millions
From Actors a1
WHERE 2 = (SELECT COUNT(DISTINCT (NetWorthInMillions))
           FROM Actors a2
           WHERE a2. NetWorthInMillions < a1. NetWorthInMillions);
```

If we examine the **Actors** table we can see that there are two actors with the third lowest NetWorth namely Kourtney Kardashian and Abhishek Bachan. Let's execute our query and see if it gives the same result:

This is a generic solution but it is slow because the inner query is processed for each row of the outer query.

Question # 5

Write a query to display actors along with a comma separated list of their digital assets.

Actors have multiple digital assets and this information is contained in the **DigitalAssets** table. We need to find duplicate rows based on the the **ActorId** column. In the **DigitalAssets** table, the column **ActorId** has duplicate values but when **ActorId** is coupled with **AssetType**, the value is unique. In MySQL the **GROUP BY** clause is used to identify duplicates.

```
SELECT ActorID, COUNT(ActorId)
FROM DigitalAssets
GROUP BY ActorId;
```

This query has found duplicate entries in the **DigitalAssets** table based on **ActorId**. Instead of a count of digital assets that the actor has, we are interested in a comma separated list. The **GROUP_CONCAT()** function comes to our aid as it is built specifically for the purpose of concatenating the results into a comma separated list.

```
SELECT ActorID, GROUP_CONCAT(AssetType)
FROM DigitalAssets
GROUP BY ActorId;
```

The last step is to join this table with **Actors** table to get the names of the actors.

```
SELECT CONCAT (FirstName, " ", SecondName) AS Actor_Name,

       GROUP_CONCAT(AssetType) AS Digital_Assets
FROM Actors
           INNER JOIN
     DigitalAssets
ON Actors.Id = DigitalAssets.ActorId
GROUP BY Id;
```

The result set is missing actors without any digital assets. So we will change the join type to LEFT JOIN.

```
SELECT CONCAT (FirstName, " ", SecondName) AS Actor_Name,
       GROUP_CONCAT(AssetType) AS Digital_Assets
FROM Actors
           LEFT JOIN
     DigitalAssets
ON Actors.Id = DigitalAssets.ActorId
GROUP BY Id;
```