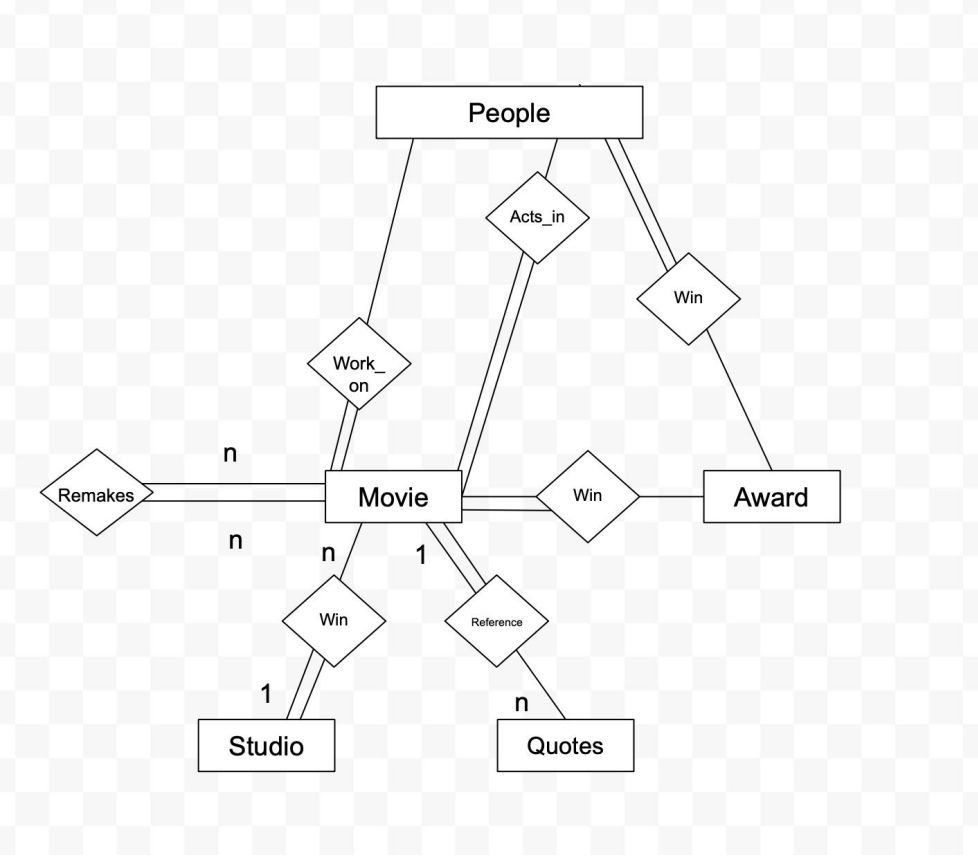
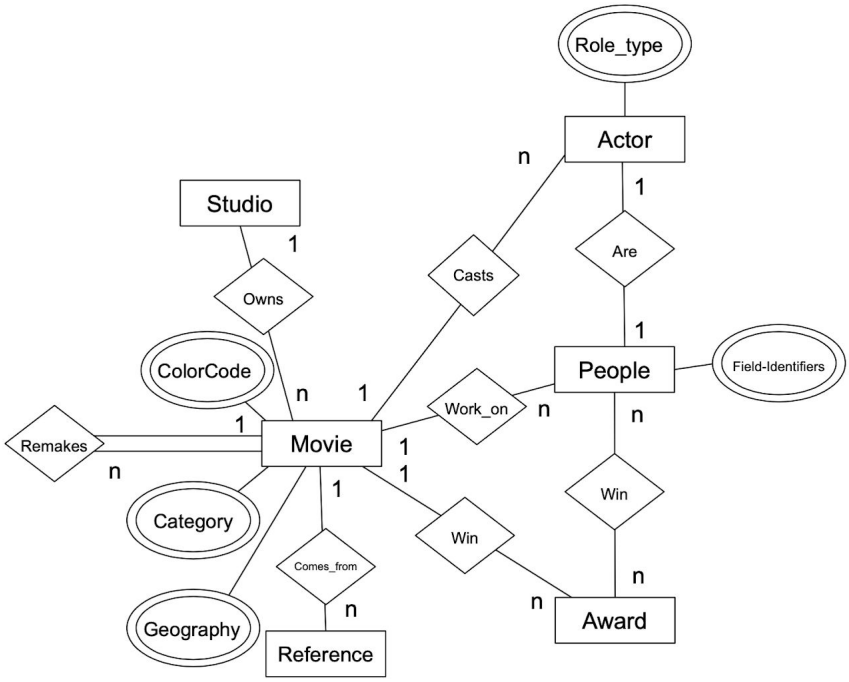
Final Report

ER Diagram Versions:

Original:




Role_type
Actor
n
1
Studio
1
Are
Casts
Owns
1
ColorCode
n
1
People
Field-Identifiers
Work_on
n
n
Remakes
1
Movie
1
1
n
Win
1
Category
Win
Comes_from
n
n
Geography
n
Award
Reference

Revised:

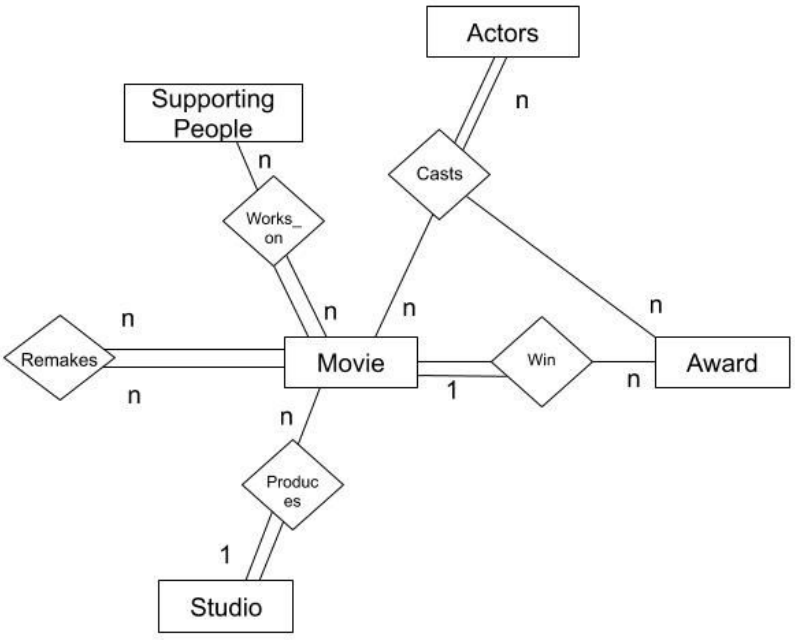

Actors
n
Supporting People
n
Casts
Works_ on
n
n
n
n
Remakes
n
Movie
Win
Award
n
1
n
n
Produc es
1
Studio

Table Definitions:

Revised:

MOVIE:

film_id	Title	Year	Director	Producer	Studio	Prc (Process)	Category	Award	Location	Notes

ACTOR:

Actor_id	Stage_name	Date_start	Date_end	DOB	Birth_name	First_name	Gender	DOB	DOD	Type	Origin

PEOPLE

Person_id	Ref_name	Codes	DID	Year_start	Year_end	Last_n	First_n	DOB	DOD	BKGD	Notes

MOVIE_STUDIO

Name	Company	City	Country	Fdate	End_date	Founder	Successor	Notes

REMAKE

Film_id	title	year	fraction	priorFilm	priorTitile	Prior Year

AWARD

Award	Organization	Country	Colloquial	Year	Notes

CAST

Film_id	title	actor	role_type	role	award

REFERENCES, GEOGRAPHY, CATEGORIES, COLOR_CODES, ROLE_TYPES, FIELD_ID, AWARD_TYPES, IMAGES and ICONS are all secondary indexes
NOTE:

- **Color Codes are just an attribute under MOVIE**
- **So is Category**
- **So is Geography**

- **Field-identifiers are attributes under people**

-Role Types under actor

- **Attributes of remakes are [The rest can be null]**
  - **Prior year**
  - **Prior title**
  - **Prior movie**

- References quote Movie

- Removing Casts

- Moving actors into a relation, with people

Table Considerations:

- At the beginning of the project, our table definitions started off looking similar to how they do now. However, a few necessary changes were made in order to correct some previous flaws. One of these main concerns in our original tables was the fact that we made the actor name and person name the primary key of the respective tables. This proved to be very problematic because there is no guarantee that the names would be unique. To solve this problem, we decided to implement a synthetic key of an assigned ID to both actors and people to guarantee their uniqueness. One of the other design implementations that we considered was combing the actor and people tables into one entity. Ultimately however, we decided against joining these two tables because the people had completely different supporting roles than actors do in the forms of directors, producers, writers, etc. Separating these two entities allowed for a clear distinction between actors, who would be the most well known people from movies and henceforth the most likely to be queried in detail, and the rest of the roles played within a movies production. Furthermore, our final determination to separate the two was determined by the fact that actors have more specific details that separate them apart from others in the people table such as stage names and photos.

Data Loading Process:

Original Approach:

- In a previous milestone, when describing how we intended to load the database, we said: “The loading language that will be used is python, using a psycopg database adapter (1) at access the Postgresql server. To handle the HTML files, the codecs library (2) will be used. These two things combined, will allow for iterative reading of the HTML files, iterative processing and error correction, and eventual loading into the database.”

Revised Approach:

- **Parsing:** We decided to use sax from Python to parse the XML file and store the correctly formatted data as text in a .txt file. An example line from the actor file looks like:

```
0>'Willie Aames'
1>''
2>''
3>'Aames'
4>'William'
5>'M'
6>'1960'
7>'1984+'
8>'unknown'
9>'\Am'
```

```
0>'Bud Abbott'
1>'1939'
2>'1956'
3>' Abbott'
4>'William'
5>'M'
6>'1895'
7>'1974'
8>'straight comedian'
9>'\Am'
```

  Since we didn’t parse the file directly from the XML to the database, we would leave any inconsistencies in the text file. We wrote a separate Python script that would remove duplicates, empty entries, and some of the other problematic data.
  To parse the HTML we used jsoup in java which would read the pods of the HTML tables and store the data in txt files.

- **Loading:** When it came to loading the database with the parsed text files, we decided to use a Java approach with JDBC similar to what we used with the CompanyQuery assignment. This approach proved to be more difficult than initially expected due to inconsistencies within the text that did not align with the corresponding data types (i.e. Date = ‘19xx’ where our data type was an integer for sorting purposes), duplicate keys within the movie files, redundant data, missing keys (i.e. actor names), and references within the casts file to movies that did not exist in the movie

table. In order to load the database with some of the nonsensical data that did not match our data types and schema within our parsed data, we were able to use a try-catch block in order to insert the valid data and catch our exceptions that were invalid queries. After the exceptions were caught, we printed them out in order to dissect the errors at the end and then continued on with inserting our data. Once the database was loaded and the remaining errors were printed out, we sorted through what went wrong and corrected all of the errors in our data parsing.

Summarizing our Experience:

Our experience overall was excruciating and sleep depriving. I learned that data from the real world is messy and a pain to sort through. The different parsers from different languages were interesting. I originally thought we had to load references, and so parsed all 165 of the references by hand. After that I came to appreciate parses much more.

POSTGRES FINAL QUERIES:

---

(1)

```
SELECT Title, Year
FROM Movie
WHERE LOWER(Title) like '%george'
ORDER BY Title;
```

Rows: 4

“Here it’s a simple comparison of all the names of the movies table. To account for casing, we made all the titles lowercase. To account for it being on the end, we used the like to have a similarity comparison”

(2)

```
SELECT c1.Actor, m1.title, r2.title, m1.year, r2.year, c1.roletype

FROM Movie as m1 JOIN Remakes as r1 on (m1.film_id = r1.wasfilm) Join Remakes
as r2 on (r1.film_id = r1.wasfilm) Join Casts as c1 on (m1.film_id =
c1.film_id) Join Casts as c2 on (r2.film_id = c2.film_id)

WHERE c1.Actor = c2.Actor
ORDER BY c1.Actor, m1.year, m1.title
```

Rows: 81

"Here joined the Movie, Remakes, and Casts tables, with the Movie table being the base movie, and the second remake table being the remake of a remake. From there, it was just a comparison to see if an actor worked both on the Movie, and the second layer remake"

(3)

```
SELECT m1.title, m1.year, m1.director, m2.title, m2.year, m2.director, r2.part
FROM ((SELECT MAX(r1.part) as max1
      FROM Remakes AS r1 JOIN Movie AS m2 ON (r1.Film_id = m2.film_id)) as x1
Join Remakes as r2 on (r2.part = x1.max1)) join Movie as m1 on (m1.film_id =
r2.film_id) join Movie as m2 on (r2.wasfilm = m2.film_id)
```

Rows: 1070

"Here we get the max value of the Part from remakes, and compare that to all the other remakes to get all the films with the max Part value. Form there, we joined Movies onto the remaining remakes to get the attributes required."

(4)

```
Cannot do it, recursive call.
```

(5)

```
SELECT DISTINCT Movie.title

FROM Movie Left Join Remakes as r1 ON (Movie.film_id = r1.film_id) Left Join
Remakes as r2 on (Movie.film_id = r2.wasfilm)

Where r1.film_id IS NOT NULL AND r2.film_id IS NOT NULL

ORDER BY Movie.title
```

Rows: 9872

“Here we took the movie table, and attached the movie film_id to the remake wasFilm id, to determine if the film was ever remade. We also attacked the remake_id to the movie_id, in order to determine is the movie is a remake itself”

(6)

```
SELECT *

FROM(SELECT DISTINCT ON (Actor.Stagenm) Actor.Stagenm, Casts.role, Movie.title,
Movie.year as m1

FROM Actor Join Casts on (Casts.Actor = Actor.Stagenm) Join Movie on
(Movie.film_id = Casts.film_id)

WHERE Movie.awards IS NOT NULL) as z1

ORDER BY z1.m1 DESC
```

ROWS: 735

“Here we joined the actors table with the cast tabe with the movie table, in order to link the movies table with the actors table, in determine the awards status”

(7)

```
SELECT Movie.title, Movie.awards, Movie.year

FROM Movie

WHERE Movie.year = 1970 AND Movie.awards IS NOT NULL
```

ROWS: 46

“For this query, we had to assume that the movie year and the award year are synonymous, because of the awards category does not have a year. Under this assumption, the Movie table has all the attributes needed for the query”

(8)

```
SELECT DISTINCT m1.title, r1.title, m1.year, r1.year

FROM movie as m1 JOIN remakes as r1 on (m1.film_id = r1.wasfilm) JOIN Movie as
m2 on (r1.film_id = m2.film_id)

WHERE m1.awards IS NOT NULL AND m2.awards IS NOT NULL

ORDER BY m1.title, r1.title
```

ROWS: 405

“Here we joined movie, with remakes on the wasFilm ID to get the immediate remakes of the movies. From there we joined movies on the remake ID. That gave us the the movies that has remakes, and if both had awards”

(9)

```
SELECT x2.Y1, x2.T1
FROM ( ( SELECT MIN(x1.L1) M1
FROM
( SELECT Length(Movie.title) as L1
FROM Movie ) as x1 ) as x3

JOIN

( SELECT Length(Movie.title) as L1, Movie.year as Y1, Movie.title T1
FROM Movie
GROUP BY Movie.year, Movie.title) as x2 on (x2.L1 = x3.M1))
```

ROWS: 6

“Here we found the MIN length of the movie titles, and Joined it to a table that had the lengths of the movie titles, and got the associated attributes from that.”
(10)

```
SELECT *
FROM person as p1
WHERE LOWER(firstnm) like 'george' AND LOWER(lastnm) like 'fox' AND pcode =
'Writer'
```

ROWS: 0
“Here we looked at the writers in the People section, and compared that to the name ‘George Fox’, making all the letters lowercase”

(11)

```
SELECT c1.Title, c1.role, c2.role, c1.Actor

FROM Casts as c1 JOIN Casts as c2 ON (c1.film_id = c2.film_id)

WHERE c1.Actor = c2.Actor AND c1.role <> c2.role
```

ROWS: 0

“Here we joined two copies of the Casts table on the film ID. This will get every combination of actor who has worked on the same movie. Them Comparing the Actors and Role Types, there are not actors under this category”

(12)

```
SELECT DISTINCT ON (z1.a1, z1.a2) z1.a1, z1.a2, z1.f1, z1.y1
FROM
(SELECT DISTINCT c1.Actor as a1, c2.Actor as a2, c1.film_id as f1, c2.film_id
as f2, m2.year as y1
       FROM Casts as c1 join Casts as c2 on (c1.film_id = c2.film_id) join
Movie as m2 on (m2.film_id = c1.film_id) join Movie as m3 on (m3.film_id =
c2.film_id)
       WHERE c1.film_id = c2.film_id AND c1.Actor <> c2.Actor AND m2.year =
m3.year) as z1

    Join

      ( SELECT DISTINCT c1.Actor as a1, c2.Actor as a2, c1.film_id as f1,
c2.film_id as f2, m2.year as y1
       FROM Casts as c1 join Casts as c2 on (c1.film_id = c2.film_id)join Movie
as m2 on (m2.film_id = c1.film_id) join Movie as m3 on (m3.film_id =
c2.film_id)
       WHERE c1.film_id = c2.film_id AND c1.Actor <> c2.Actor AND m2.year =
m3.year) as z2

    on (z1.a1 = z2.a1 AND z1.a2 = z2.a2)

 WHERE z1.a1 = z2.a1 AND z1.a2 = z2.a2 AND z1.f1 <> z2.f1 AND z1.y1 = z2.y1
```

ROWS: 1516
“Here we set two sets of tables: Actors that have worked together on the same films. Then comparing these two tables to each other, to see whether the pair match a row in the other table, where the films are not the same. We eliminate the possibility of a three way repeat, because of the outward distinct.”

(13)

```
SELECT Movie.title, Movie.year, role
FROM Casts Join Movie on (Casts.film_id = Movie.film_id)
WHERE LOWER(casts.Actor) LIKE 'tom cruise'
ORDER BY Movie.Year
```

ROWS: 22
“Here we took the name of the Actor in Casts, and compared it to ‘Tom cruise’, then joined the movies off of that”

(14)

```
SELECT DISTINCT ON (z1.a2) z1.a2
FROM
  (SELECT c1.Actor as a1, c2.Actor as a2, c1.film_id as f1
  FROM Casts as c1 join Casts as c2 on (c1.film_id = c2.film_id)
  WHERE c1.film_id = c2.film_id AND LOWER(c1.Actor) = 'val kilmer' AND c1.Actor
<> c2.Actor) as z1 join

  (SELECT c1.Actor as a1, c2.Actor as a2, c1.film_id as f1
  FROM Casts as c1 join Casts as c2 on (c1.film_id = c2.film_id)
  WHERE c1.film_id = c2.film_id AND LOWER(c1.Actor) = 'clint eastwood' AND
c1.Actor <> c2.Actor) as z2 on (z1.a2 = z2.a2)
```

ROWS: 1

“Here we created two relations: One with all the actors that have worked with Val Kilmer, and those that have worked with Clint Eastwood. From there we determined if any actors were in the intersection”

(15)

```
SELECT DISTINCT c42.stagenm
FROM
  (SELECT DISTINCT c40.actor as c41
  FROM (
    SELECT DISTINCT c37.film_id as c38
    FROM (
      SELECT DISTINCT c34.actor as c35
      FROM (
        SELECT DISTINCT c31.film_id as c32
        FROM (
          SELECT DISTINCT c28.actor as c29
          FROM (
            SELECT DISTINCT c24.film_id as c25
            FROM (
              SELECT DISTINCT c20.actor as c21
              FROM (
                SELECT DISTINCT c16.film_id as c17
                FROM (
                  SELECT DISTINCT c12.actor as c13
                  FROM (
                    SELECT DISTINCT c8.film_id as c9
                    FROM (
                      SELECT DISTINCT c2.actor as c5
                      FROM (
                        SELECT c1.film_id as c3
                        FROM Casts as c1
                        WHERE c1.Actor = 'Kevin Bacon') AS c4 Left Join Casts
as c2 ON (c2.film_id = c4.c3)
```

```
                    WHERE c4.c3 IS NOT NULL) as c7 Left Join casts as c8 on
(c7.c5 = c8.Actor)
                  WHERE c7 IS NOT NULL ) AS c11 Left Join Casts as c12 ON
(c12.film_id = c11.c9)
                 WHERE c11.c9 IS NOT NULL) As c15 Left Join casts as c16 ON
(c15.c13 = c16.Actor)
               WHERE c15 IS NOT NULL) AS c19 Left Join Casts as c20 ON
(c20.film_id = c19.c17)
             WHERE c19.c17 IS NOT NULL ) as c23 Left Join casts as c24 on
(c23.c21 = c24.Actor)
           WHERE c23 IS NOT NULL) AS c27 Left Join Casts as c28 ON
(c28.film_id = c27.c25)
         WHERE c27.c25 IS NOT NULL)as c30 Left Join casts as c31 on (c30.c29 =
c31.Actor)
       WHERE c30 IS NOT NULL) AS c33 Left Join Casts as c34 ON (c34.film_id =
c33.c32)
     WHERE c33.c32 IS NOT NULL)as c36 Left Join casts as c37 on (c36.c35 =
c37.Actor)
   WHERE c36 IS NOT NULL) AS c39 Left Join Casts as c40 ON (c40.film_id =
c39.c38)
  WHERE c39.c38 IS NOT NULL) AS c43 INNER JOIN actor as c42 ON (c43.c41 =
c42.stagenm)
WHERE c42 IS NOT NULL
```

ROWS: 4932

“Here we got the actors that have worked with Kevin Bacon, the movies those actors have been in, and their corresponding unique actors. This repeated for eight layers out”

(16)

```
SELECT DISTINCT c7.actor, c3.Director, COUNT(c7.actor)
FROM Movie as c3 JOIN casts as c7 on (c3.film_id = c7.film_id)
WHERE LOWER(c3.Director) LIKE 'clint eastwood'
GROUP BY c7.actor, c3.Director
ORDER BY COUNT(c7.actor) DESC;
```

ROWS: 97

“This gets all the actors with the corresponding movies, and determines which have been directed by Clint Eastwood. The results are formated, with the COUNT being a category and ORDER”

(17)

```
SELECT m1.Cat
FROM Movie as m1 Join Casts as c1 on (c1.film_id = m1.film_id)
WHERE LOWER(c1.Actor) LIKE 'ronald regan'
```

ROWS: 4
“Here we queried the Movies table joined the Casts table, to determine if the actor is Ronald Regan ”

(18)

```
SELECT z3.m1, z2.s1
FROM

( SELECT Max(z1.s2) as m1
FROM
      ( SELECT Count(Studio) as s2, Studio as s1
      FROM Movie
      WHERE Studio IS NOT NULL
      GROUP BY Studio ) as z1 ) as z3

JOIN

     (SELECT Count(Studio) as s2, Studio as s1
      FROM Movie
      WHERE Studio IS NOT NULL
      GROUP BY Studio ) as z2 on (z3.m1 = z2.s2)

WHERE z3.m1 = z2.s2
```

ROWS: 1
“Here we found the max count of the studios, and compared it to another table that has the count of the studio. When joined, the result is the studio name along with the max count of movies produced”
(19)

```
SELECT s5.s3, s5.s4
FROM

(SELECT MAX(s2.s3) as x1
FROM
(SELECT COUNT(s1.Cat) as s3, s1.cat as s4
FROM Movie as s1
WHERE s1.Studio = 'Paramount'
GROUP BY s1.cat ) as s2) as x2

JOIN

( SELECT COUNT(s1.Cat) as s3, s1.cat as s4
FROM Movie as s1
WHERE s1.Studio = 'Paramount'
```

```
GROUP BY s1.cat ) as s5 on (x2.x1 = s5.s3)

WHERE x2.x1 = s5.s3
```

ROWS: 1
“Here we get the max count of the Paramount category, where we compare it to another table with the counts. This gets the max count, along with the name of the category”

(20)

```
SELECT c1.role, c1.Actor, m1.title
FROM Movie as m1 Join Casts as c1 on (m1.film_id = c1.film_id) join Actor as a1
on (c1.Actor = a1.Stagenm)
WHERE m1.Director = m1.Producers AND m1.Producers = c1.Actor
```

ROWS: 0
“Here we join the Movie, Casts and Actors table. From there we determine if the person directed, produce and acted in the movie. ”
(21)

```
SELECT COUNT(c1.roletype), c1.roletype
FROM Casts as c1
GROUP BY c1.roletype
ORDER BY COUNT(c1.roletype) DESC;
```

ROWS: 182
“This is simply determining the role types from Casts, and grouping them by type, ordering by count”
(22)

```
SELECT COUNT(m1.cat), m1.cat
FROM Movie as m1
GROUP BY m1.cat
ORDER BY COUNT(m1.cat);
```

ROWS: 99
“This simply provides a list of the categories from the movies table”
(23)

```
SELECT z4.r2, z4.r3
FROM
( SELECT MAX(r1.r2) as m1
FROM
  (SELECT (m1.year - a1.dob) as r2, a1.stagenm as r3
```

```
   FROM Actor as a1 Join Casts as c1 on (a1.stagenm = c1.actor) Join Movie as
m1 on (m1.film_id = c1.film_id)
   WHERE m1.year IS NOT NULL AND a1.dob IS NOT NULL) as r1 ) as z3

     JOIN

  (SELECT (m1.year - a1.dob) as r2, a1.stagenm as r3
   FROM Actor as a1 Join Casts as c1 on (a1.stagenm = c1.actor) Join Movie as
m1 on (m1.film_id = c1.film_id)
   WHERE m1.year IS NOT NULL AND a1.dob IS NOT NULL) as z4 on (z3.m1 = z4.r2)

WHERE z3.m1 = z4.r2
```

ROWS: 1

“Here we gathered two two relations: The max age of the actor from a movie, and a relation with the count and attributes. From there we joined the two tables, and found the max count with the attibutes”