ASSIGNMENT 2- SQL

Q1. Loading data into the tables

1. Table1: Title
   Creating the title table

```
CREATE TABLE Title( id INTEGER PRIMARY KEY,
                    type VARCHAR(15),
                    title TEXT,
                    originalTitle TEXT,
                    startYear INTEGER,
                    endYear INTEGER,
                    runtime INTEGER,
                    avgRating FLOAT,
                    numVotes INTEGER);
```

   Inserting the values into Title table

```
INSERT INTO Title
SELECT ti.titleId, ti.titleType, ti.primaryTitle,
ti.originalTitle, ti.startYear, ti.endYear, ti.runtime,
tr.averageRating, tr.numVotes
FROM titleInfo as ti
JOIN ratings AS tr
ON ti.titleId = tr.titleId
WHERE ti.isAdult = 'false';
```

2. Table2: Genre

   Creating the a temporary genre table TempGenre

```
CREATE TABLE TempGenre(id serial primary key,
                       genre varchar(255));
```

Inserting the values into the TempGenre Table

```
INSERT INTO TempGenre(genre)
SELECT ti.genres
FROM Titleinfo as ti;
```

Converting the data in genre column of the temporary table TempGenre to array using the string_to_array command

```
ALTER TABLE TempGenre
ALTER genre type varchar[] using string_to_array(genre, ',');
```

Creating the Genre Table

```
CREATE TABLE Genre(
id serial primary key,
genre varchar(300));
```

Inserting the values from the temporary table TempGenre into the Genre table

```
INSERT INTO Genre(genre)
SELECT DISTINCT unnest(genre) FROM TempGenre;
```

Converting the data in the genre column of the titleInfo table to array type.

```
ALTER TABLE TitleInfo
ALTER genres type varchar[] using string_to_array(genres,',');
```

3. **Table3: Title_Genre Table**

Creating a temporary title_genre table

```
CREATE TABLE Temp_Title_Genre(titleId integer,
                              genres varchar(300));
```

Inserting the values into the temporary table

```
INSERT INTO Temp_Title_Genre
SELECT titleId, unnest(genres)
FROM TitleInfo;
```

Creating the Title_Genre Table

```
CREATE TABLE Title_Genre(genre integer,
                         title integer,
                         primary key(genre, title));
```

Inserting the values into the Title_Genre table

```
INSERT INTO Title_Genre
SELECT DISTINCT g.id, tg.titleId
FROM Temp_Title_Genre as tg
JOIN Genre as g on g.genre = tg.genres;
```

Setting the foreign key

```
ALTER TABLE Title_Genre ADD CONSTRAINT fk_titlegenre_titleId FOREIGN
KEY(genre) REFERENCES Genre(id);

DELETE FROM Title_Genre WHERE NOT exists ( SELECT NULL FROM Title WHERE
Title_Genre.title = Title.Id);

ALTER TABLE Title_Genre ADD CONSTRAINT fk_titlegenret_titleId FOREIGN
KEY(title) REFERENCES Title(Id);
```

4. **Table 4: Member Table:**

```
CREATE TABLE Member(id INTEGER PRIMARY KEY,
                    name VARCHAR(200) NOT NULL,
                    birthYear VARCHAR(5),
                    deathYear VARCHAR(5));
```

Inserting the values into the Member Table from the nameinfo table

```
INSERT INTO Member
SELECT n.nameId, n.primaryName, n.birthYear, n.deathYear
FROM NameInfo AS n;
```

5. **Table 5: Titile_Actor table:**

Creating the Title_Actor table

```
CREATE TABLE Title_Actor(actor integer,
                         title integer,
                         primary key(actor,title));
```

Inserting the values into the Title_Actor table from the principals table where the category is actor

```
INSERT INTO Title_Actor
SELECT distinct p.nameId, p.titleId
FROM Principals as p
WHERE p.category = 'actor';
```

Setting the foreign key

```
ALTER TABLE Title_Actor ADD CONSTRAINT fk_titleactor_titleId FOREIGN KEY
(actor) REFERENCES Member(id);

DELETE FROM Title_Actor WHERE NOT exists ( SELECT NULL FROM Title WHERE
Title_Actor.title = Title.Id);

ALTER TABLE Title_Actor ADD CONSTRAINT fk_titleactort_titleId FOREIGN KEY (title)
REFERENCES Title(Id);
```

6. **Table 6: Title_Writer:**

Creating the Title_Writer table

```
CREATE TABLE Title_Writer(writer integer,
                          title integer,
                          primary key(writer, title));
```

Inserting into Title_Writer Table

```
INSERT INTO Title_Writer
SELECT p.nameId, p.titleId
FROM Principals as p
WHERE p.category = 'writer';
```

Setting the foreign key

```
ALTER TABLE Title_Writer ADD CONSTRAINT fk_titlewriter_titleId FOREIGN KEY
(writer) REFERENCES Member(id);

DELETE FROM Title_Writer WHERE NOT exists ( SELECT NULL FROM Title WHERE
Title_Writer.title = Title.Id );

ALTER TABLE Title_Writer ADD CONSTRAINT fk_titlewritert_titleId FOREIGN KEY
(title) REFERENCES Title(Id);
```

7. **Table 7: Title_Director**
   Creating the Title_Director table

```
CREATE TABLE Title_Director(director integer,
                            title integer,
                            primary key(director, title));
```

Inserting the values into the Title_Director Table from the principals table where the category is director.

```
INSERT INTO Title_Director
SELECT p.nameId, p.titleId
FROM Principals as p
WHERE p.category = 'director';
```

Setting the foreign key

```
ALTER TABLE Title_Director ADD CONSTRAINT fk_titledirector_titleId FOREIGN KEY
(director) REFERENCES Member(id);

DELETE FROM Title_Director WHERE NOT exists ( SELECT NULL FROM Title
WHERE Title_Director.title = Title.Id );

ALTER TABLE Title_Director ADD CONSTRAINT fk_titledirectort_titleId FOREIGN KEY
(title) REFERENCES Title(Id);
```

8. **Table 8: Title_Producer:**
   Creating the Title_Producer table

```
CREATE TABLE Title_Producer(producer integer,
                            title integer,
                            primary key(producer, title));
```

Inserting the values into the Title_Producer Table from the principles table where category is producer

```
INSERT INTO Title_Producer
SELECT p.nameId, p.titleId
FROM Principals as p
WHERE p.category = 'producer';
```

Setting the foreign key

```
ALTER TABLE Title_Producer ADD CONSTRAINT fk_titleproducer_titleId FOREIGN
KEY (producer) REFERENCES Member(id);


DELETE FROM Title_Producer WHERE NOT exists ( SELECT NULL FROM Title
WHERE Title_Producer.title = Title.Id );

ALTER TABLE Title_Producer ADD CONSTRAINT fk_titleproducert_titleId FOREIGN
KEY (title) REFERENCES Title(Id);
```

9. **Table 9:Character Table:**

Creating a temporary table TempCharacter

```
CREATE TABLE TempCharacter(cId serial primary key,
                                character text);
```

Inserting the values into the temporary TempCharacter table from the principals table

```
INSERT INTO TempCharacter(character)
SELECT p.charactersname
FROM Principals as p;
```

Updating the values of the character column of the temporary table using the replace command. Replace is used to replace the brackets and quotes.

```
UPDATE TempCharacter SET character = REPLACE(character, '[', '');

UPDATE TempCharacter SET character = REPLACE(character, ']', '');

UPDATE TempCharacter SET character = REPLACE(character, '"', '');
```

Converting the values of character column of the temporary table to array using string_to_array

```
ALTER TABLE TempCharacter
ALTER character type text[] USING string_to_array(character, ',');
```

Creating the character table

```
CREATE TABLE Character(Id serial primary key,
                       Character text);
```

Inserting the values into the character table from the temporary TempCharacter table

```
INSERT INTO Character(character)
SELECT DISTINCT unnest(character) FROM TempCharacter;
```

10. **Table 10: Actor_Title_Character**
    Creating two temporary tables Temp_Actor_Title_Character and Actor_Title_Character_NewTemp.First, updating the values of the Temp_Actor_Title_Character table and then copying its data into the second temporary table. Then copying the data from the second temporary table to the Actor_Title_Character table.

    Creating the temporary table Temp_Actor_Title_Character

```
Create Table Temp_Actor_Title_Character(actor integer, title integer,
character text);
```

    Inserting values into Temp_Actor_Title_Character from the principals table

```
INSERT INTO Temp_Actor_Title_Character
SELECT p.nameId,p.titleId,p.characters
FROM principals AS p;
```

    Updating the values of the character column of the temporary table using the replace command. Replace is used to replace the brackets and quotes.

```
UPDATE TempActor_Title_Character SET character = REPLACE(character, '[', '');

UPDATE TempActor_Title_Character SET character = REPLACE(character, ']', '');

UPDATE TempActor_Title_Character SET character = REPLACE(character, '"', '');
```

    Converting the values of character column of the temporary table to array using string_to_array

```
ALTER TABLE TempActor_Title_Character
ALTER character TYPE varchar[] USING string_to_array(character,',');
```

    Creating the second temporary table Actor_Title_CharacterNewTemp table

```
CREATE TABLE Actor_Title_CharacterNewTemp(actor integer,title integer, Character
varchar);
```

Inserting the values into the second temporary table from the first temporary table created.

```
INSERT INTO Actor_Title_CharacterNewTemp
SELECT actor,title,unnest(character) FROM TempActor_Title_Character;
```

Creating the table Actor_Title_Character

```
CREATE TABLE Actor_Title_Character(actor integer,
                                   title integer,
                                   character integer,
                                   primary key (actor, title, character));
```

Inserting the values into Actor_Title_Character from the second temporary table Actor_Title_CharacterNewTemp

```
INSERT INTO Actor_Title_Character
SELECT DISTINCT at.actor,at.Id,c.cId
FROM Actor_Title_CharacterNewTemp AS at
JOIN Character AS c
ON c.character= at.character;
```

Setting the foreign key

```
ALTER TABLE Actor_Title_Character ADD CONSTRAINT fk_actor_title_character_titleId
FOREIGN KEY (character) REFERENCES Character(cId);

DELETE FROM Actor_Title_Character WHERE NOT exists ( SELECT NULL FROM
Title_Actor WHERE Actor_Title_Character.actor = Title_Actor.actor and
Actor_Title_Character.title = Title_Actor.title );

ALTER TABLE Actor_Title_Character ADD CONSTRAINT
fk_actor_title_charactert_titleId FOREIGN KEY (actor, title) REFERENCES
Title_Actor(actor, title);
```

Q2.

- **2.1 Number of invalid Title_Actor relationships with respect to characters.**

```
SELECT count(*)
```

```
FROM Title_Actor as ta
LEFT JOIN Actor_Title_Character as ac ON ac.title = ta.title
WHERE ac.character is null;
```

Number of rows: 139239
Time : 2 secs 12 msec

- **2.2 Alive actors whose name starts with “Phi” and did not participate in any movie in 2014.**

```
SELECT name
FROM Title as t
join Title_Actor as at on t.Id = at.title
join Member as m on m.id = at.actor
where name like 'Phi%' and deathyear is null and startyear <> 2014;
```

Number of rows: 8425
Time : 2 secs

- **2.3 Producers who have produced the most talk shows in 2017 and whose name contains “Gill”.**

```
SELECT name, count(Id)
FROM Member AS m
JOIN Title_Producer AS p on p.producer = m.id
JOIN Title as t on t.Id = p.title
JOIN Title_Genre as g on g.title = t.Id
JOIN Genre as g on g.id = g.genre
WHERE m.deathYear is null and t.startYear = 2017 and g.genre = 'Talk-Show' and
m.name like '%Gill%'
GROUP BY name
ORDER BY count(t.Id) DESC;
```

Number of rows: 8
Time : 568 msec

- **2.4 Alive producers ordered by the greatest number of long-run titles produced (runtime greater than 120 minutes)**

```
SELECT name, runtime
FROM Title AS t
JOIN Title_Producer AS p ON p.title=t.Id
JOIN Member AS m ON m.id = p.producer
WHERE m.deathYear is null and t.runtime > 120
ORDER BY runtime desc;
```

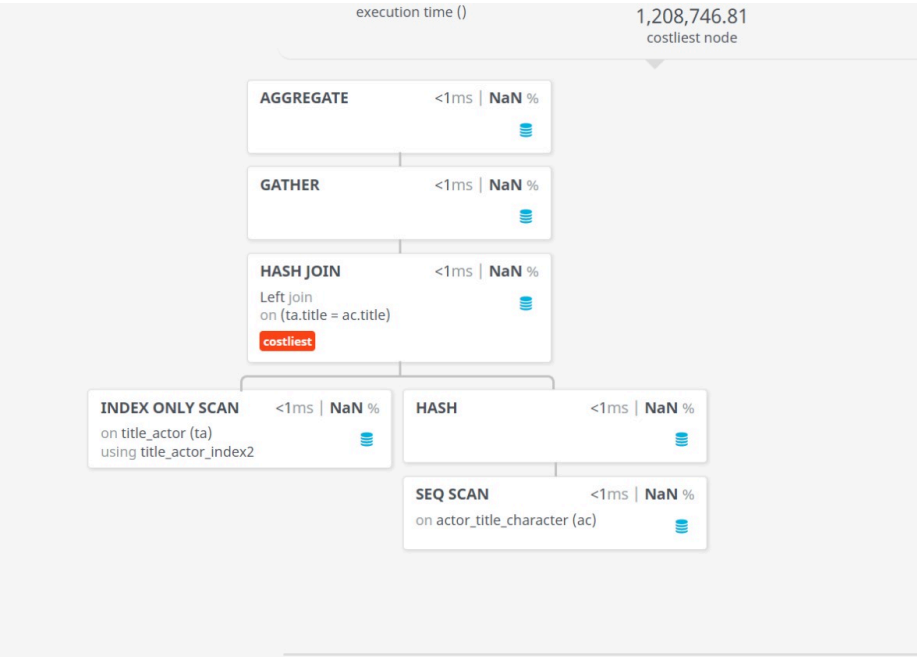
Number of rows: 23441
Time : 1 sec

- **2.5 Alive actors who have portrayed Jesus Christ (simply look for a character with this specific name)**

  ```
  SELECT name
  FROM Member as m
  JOIN Actor_Title_Character as ac on ac.actor = m.id
  JOIN Character as c on c.cid = ac.character
  WHERE c.character = 'Jesus Christ' and deathYear is null;
  ```
  Number of rows: 87
  Time : 654 msec

Q3.

3.1



Query:

```
EXPLAIN (format JSON) SELECT count(*)
FROM Title_Actor as ta
LEFT JOIN Actor_Title_Character as ac ON ac.title = ta.title
WHERE ac.character is null;
```

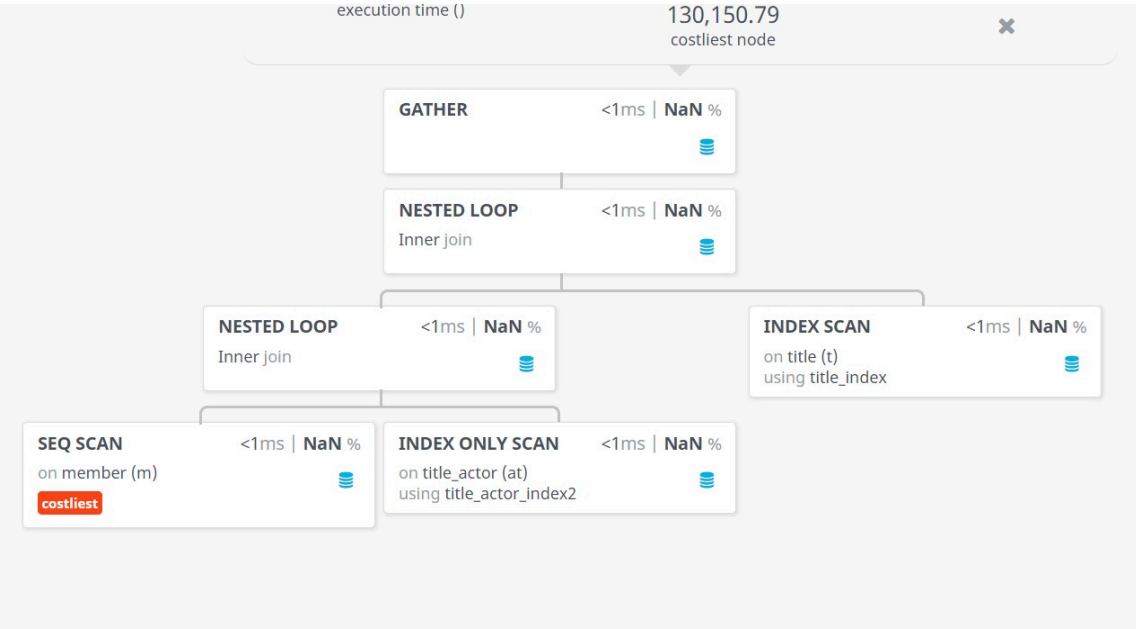
The parsing in the tree is from bottom to top, the first sequential scan takes place on the table actor_title_character. When sequential scan takes place the whole actor_title_character table is scanned.

Then an Index only scan takes place on the title_actor table. The Index scan goes to a particular index of the table. This scan gives more efficient results when only a single row is required.

A left join on title_actor and actor_title_character table is performed by comparing the id's of both the tables. The hash table that is generated is used by the left join. The gather node collects all the information given by the where condition in our query. The node at the top is the aggregate node and it is responsible for giving the results of the query.

According to the query, the condition where ac.character is null is handled by the gather node as it collects all the information based on the condition clause. Then a left join on both the tables takes place and the count(*) from the Title_Actor table is to be displayed which is the aggregate function.

3.2



Query:

```
EXPLAIN (format JSON) SELECT name
FROM Title as t
join Title_Actor as at on t.Id = at.title
join Member as m on m.id = at.actor
where name like 'Phi%' and deathyear is null and startyear <> 2014;
```

Movement in the tree is from bottom to top. In this query a sequential scan is performed on the member table. Sequential scan scans the complete table. An inner join is performed on the tables title_actor and member on the condition where the name starts with ‘Phi’ and startyear is not 2014.The tables are joined on m.id(member id)=at.actor(Title_actor actor). Again an inner join is performed on the tables but on the condition where the id of Title_actor table is equal to the id of the Title table.

The query performs two joins, first it joins the member table and the title_actor table and then it joins the Title table along with the Title_Actor table. Only the names which start with Phi and the startyear not equal to 2014 and the deathyear is null are obtained as per the condition given and then displayed accordingly.

3.3

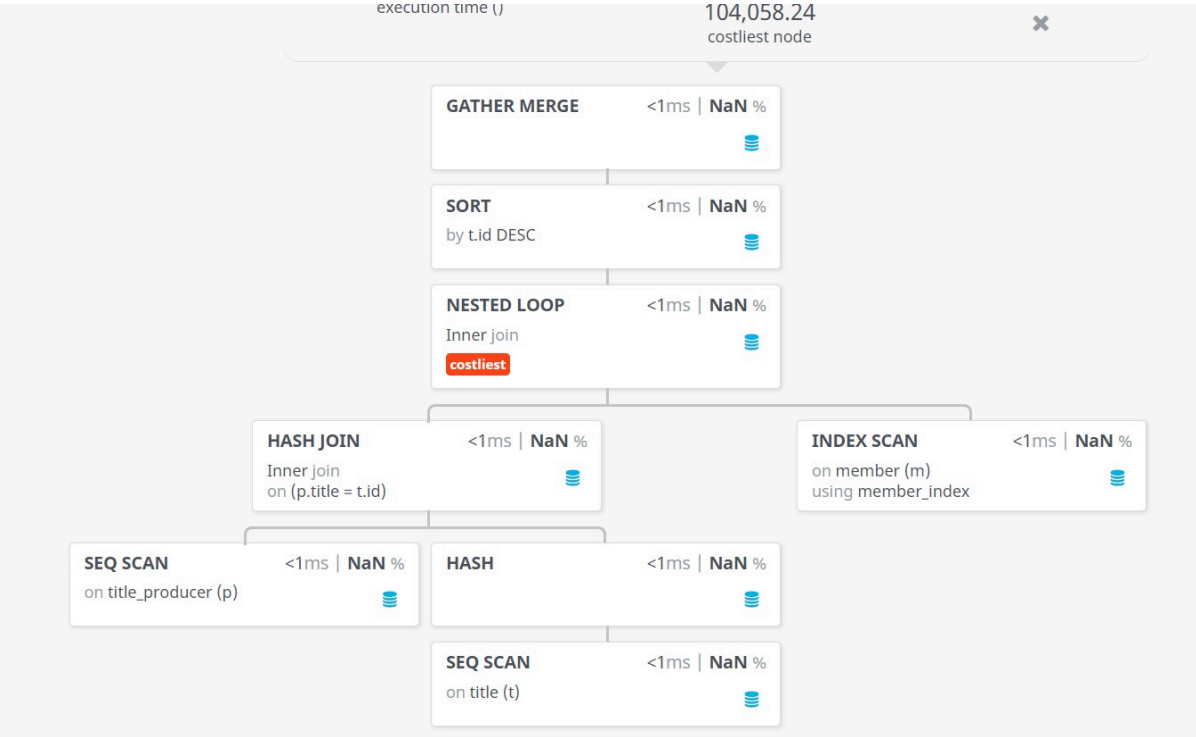
Query:

```
EXPLAIN (format JSON) SELECT name, count(Id)
FROM Member AS m
JOIN Title_Producer AS p on p.producer = m.id
JOIN Title as t on t.Id = p.title
JOIN Title_Genre as tg on tg.title = t.Id
JOIN Genre as g on g.id = tg.genre
WHERE m.deathYear is null and t.startYear = 2017 and g.genre = 'Talk-Show' and
m.name like '%Gill%'
GROUP BY name
ORDER BY count(t.Id) DESC;
```

Considering the parsing of tree from bottom to top. A sequential scan on the genre table is performed and index only scan is performed on the title_genre table. An inner join is performed on the genre table and title_genre table.These tables are joined on the condition where the id of title_genre table is equal to the id of the genre table. A sequential scan takes place on the tile_producer table. Again a join is performed on the title_producer table and the member table.

When we join the title table and the title_producer table we get the title for the producers who produced Talk-Show. When title_producer is joined with the member table we get producer id and name containing Gill. We want the talk shows ordered in descending order. Hash table is generated which is used whenever a hash join is performed. The aggregate function is count().

3.4

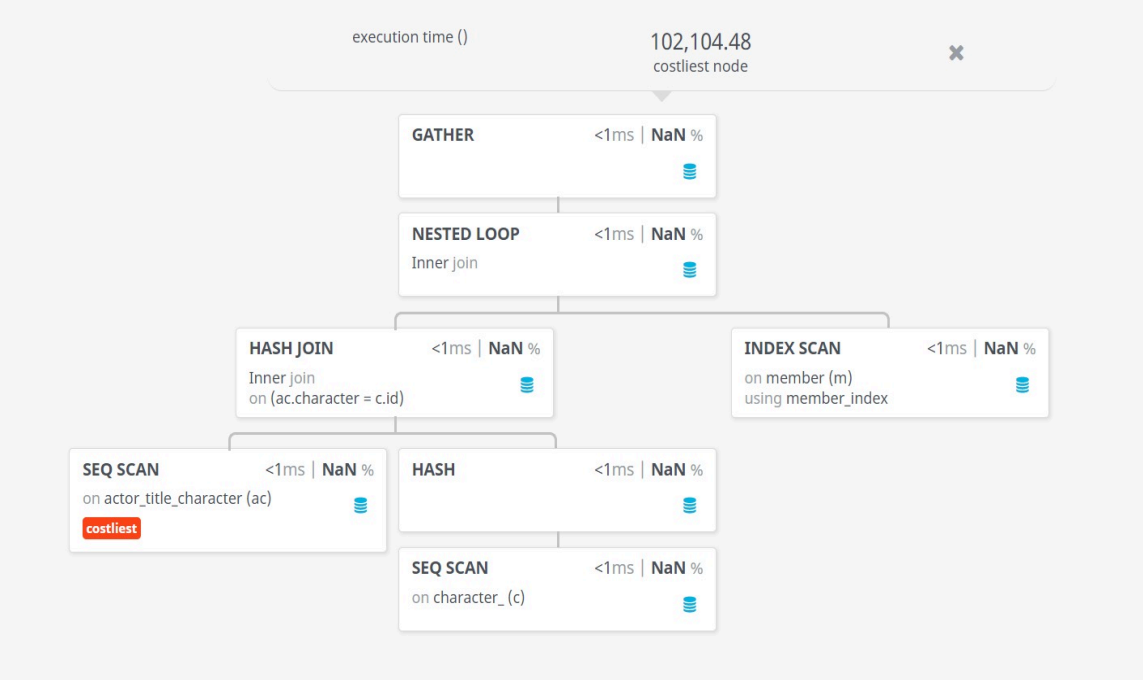


Query:

```
EXPLAIN (format JSON) SELECT name, runtime
FROM Title AS t
JOIN Title_Producer AS p ON p.title=t.Id
JOIN Member AS m ON m.id = p.producer
WHERE m.deathYear is null and t.runtime > 120
ORDER BY runtime desc;
```

Sequential scan is performed on the title table. From the title table the values having runtime greater than 120 are obtained. A join is performed on it along with Title_Producer to get the titles. Again a join is performed where the member table is joined with the Title_Producer table to get the id for producer and runtime.
It also checks if the deathyear is null so that we get the producers that are alive. Whenever a join is performed a hash table is generated. The hash join uses a hash table.The sort node at the top does the order by runtime and sorts the obtained values in descending order. The gather merge at the top does the job of merging the data that is gathered by the select condition and then displaying the final result of the query.

3.5



Query:

```
EXPLAIN (format JSON) SELECT name
FROM Member as m
JOIN Actor_Title_Character as ac on ac.actor = m.id
JOIN Character as c on c.cid = ac.character
WHERE c.character = 'Jesus Christ' and deathYear is null;
```

A sequential scan is performed on the character table. Another sequential scan is performed on the actor_title_character table. Sequential scan does the job of scanning the whole table.From the character table the characters having the values ‘Jesus Christ’ are scanned when sequel scan is performed and also the id related to them are obtained.Then a join is performed on this along with Actor_Title_Character on the basis of their id’s. A hash table is obtained whenever a hash join is performed. Again a join is performed where the Actor_Title_Character table is joined with the Member table on the basis of their id’s and a condition where the deathyear is null is applied so that all the actors that are alive are obtained.

Q 4

4.1 $\Pi_{count(*)}$
$(\sigma_{ac.character = NULL}$
$(\rho_{ta}\ title\_actor \bowtie_{ac.title = ta.title}$
$\rho_{ac}\ actor\_title\_character))$

4.2 Π_{name}
$(\sigma_{name\ Like\ "Phi\%"\ AND\ deathYear = NULL\ AND\ startYear <> 2014}$
$(\rho_{t}\ title \bowtie_{t.titleid = at.title}$
$\rho_{at}\ title\_actor \bowtie_{m.id = at.actor}$
$\rho_{m}\ member))$

4.3 $\Pi_{count(titleid), name}$
$\sigma_{m.deathyear = NULL\ AND\ t.startYear = 2017\ AND\ g.genre = "TalkShow"\ AND\ m.name\ LIKE\ "\%Gill\%"}$
$(\rho_{m}\ member \bowtie_{p.producer = m.id}$
$\rho_{p}\ title\_producer \bowtie_{t.titleid = p.title}$
$\rho_{t}\ title \bowtie_{g.title = t.titleid}$
$\rho_{g}\ title\_genre \bowtie_{g.id = g.genre}$
$\rho_{g}\ genre)$

4.4 $\Pi_{name, runtime}$
$\sigma_{m.deathyear = NULL\ AND\ t.runtime > 120}$
$(\rho_{t}\ title \bowtie_{p.title = t.id}$
$\rho_{p}\ title\_producer \bowtie_{m.id = p.producer}$
$\rho_{m}\ member)$

4.5 Π_{name}
$\sigma_{c.character = "Jesus\ Christ"\ AND\ deathyear = NULL}$
$(\rho_{m}\ member \bowtie_{ac.actor = m.id}$
$\rho_{ac}\ actor\_title\_character \bowtie_{c.cid = ac.character}$
$\rho_{c}\ character)$

Q 5

Indexing is used to improve the performance of the database. With the help of indexing we can access the specific rows or find the result of the queries more efficiently. For this question we will first run the queries without indexes and note the time. Then we will create indexes for the tables and then run the same queries. We can see in the result that the time after using indexes is much lesser than the time taken before using the indexes.

Creating Indexes:

- **Title table index**

  ```
  CREATE INDEX Title_Index
  ON Title (Id);
  ```

- **Member table index**

  ```
  CREATE INDEX Member_Index
  on Member (id);

  CREATE INDEX  Member_Indx
  on Member (name);

  CREATE INDEX  Member_Ind
  on Member (deathYear);
  ```

- **Title_Actor table index**

  ```
  CREATE INDEX Title_Actor_Indx
  ON Title_Actor (title);
  ```

- **Title_Genre table index**

  ```
  CREATE INDEX Title_Genre_Indx
  on Title_Genre(title);
  ```

- **Actor_Title_Character table index**

  ```
  CREATE INDEX Actor_Title_Character_Index
  ON Actor_Title_Character (title);

  CREATE INDEX Actor_Title_Character_Ind
  ON Actor_Title_Character (actor, character);
  ```

- **Title_Producer table index**

  CREATE INDEX Producer_Index
  on Title_Producer(producer);

  CREATE INDEX Producer_Indx
  on Title_Producer(title);

1. **Number of invalid Title_Actor relationships with respect to characters.**

   SELECT count(*)
   FROM Title_Actor as ta
   LEFT JOIN Actor_Title_Character as ac ON ac.title = ta.title
   WHERE ac.character is null;

   The time taken without indexes is 2 sec whereas with the help of indexing the time was reduced to 864 msec.

2. **Alive actors whose name starts with “Phi” and did not participate in any movie in 2014.**

   SELECT name
   FROM Title as t
   join Title_Actor as at on t.Id = at.title
   join Member as m on m.id = at.actor
   where name like 'Phi%' and deathyear is null and startyear <> 2014;

   The time for the query to run without indexes was 2 sec 212 msec and after using indexes the time was reduced to 821 msec.

3. **Producers who have produced the most talk shows in 2017 and whose name contains “Gill”**

   SELECT name, count(Id)
   FROM Member AS m
   JOIN Title_Producer AS p on p.producer = m.id
   JOIN Title as t on t.Id = p.title
   JOIN Title_Genre as tg on tg.title = t.Id
   JOIN Genre as g on g.id = tg.genre
   WHERE m.deathYear is null and t.startYear = 2017 and g.genre = 'Talk-Show' and m.name like '%Gill%'

```
GROUP BY name
ORDER BY count(t.Id) DESC;
```

The total time for this query without indexes was 863 msec whereas the time for the query after using the indexes was found to be 630 msec

4. **Alive producers ordered by the greatest number of long-run titles produced**

```
SELECT name, runtime
FROM Title AS t
JOIN Title_Producer AS p ON p.title=t.Id
JOIN Member AS m ON m.id = p.producer
WHERE m.deathYear is null and t.runtime > 120
ORDER BY runtime desc;
```

The time for this query without indexes was 819 msec. After using indexes the reduced time was 519 msec

5. **Alive actors who have portrayed Jesus Christ**

```
SELECT name
FROM Member as m
JOIN Actor_Title_Character as ac on ac.actor = m.id
JOIN Character as c on c.cid = ac.character
WHERE c.character = 'Jesus Christ' and deathYear is null;
```

When this query was executed without indexes the time obtained for its execution was 1 sec 25 msec. After executing the query with the help on index tha runtime of the query was reduced and it was more efficient. The time with the help of indexes was 553 msec.