ASSIGNMENT 4-DOCUMENT DATABASES

Q1.Loading the data

Creating Movies.json file:

```
COPY (SELECT json_strip_nulls(row_to_json(r1))
FROM(SELECT id as _id, type, title, originaltitle, startYear, endYear, runtime,
avgrating, numvotes, genres, actors, directors, producers, writers FROM Title

LEFT JOIN
(SELECT title as tid, array_agg(ge.genre) as genres
FROM title_genre JOIN Genre ge ON title_genre.genre = ge.id
GROUP BY tid ORDER BY tid) as ge ON Title.id = ge.tid

LEFT JOIN
(SELECT title as tid, array_agg(director) as directors
 FROM title_director GROUP BY tid) as dir ON Title.id = dir.tid

LEFT JOIN
(SELECT title as tid, array_agg(producer) as producers
 FROM title_producer GROUP BY tid) as pro ON Title.id = pro.tid

LEFT JOIN
(SELECT title as tid, array_agg(writer) as writers
FROM title_writer GROUP BY tid) as wri ON Title.id = wri.tid

LEFT JOIN
(SELECT titleid, json_agg(actors) as actors FROM
(SELECT tt.titleid, json_build_object('actor', tt.actorid, 'roles', tt.roles) as actors FROM
(SELECT act.title as titleid, act.actor as actorid, array_agg(character.character) as roles
 FROM character JOIN actor_title_character act ON character.id = act.character
 GROUP BY (act.title, act.actor) ORDER BY (act.title, act.actor)) as tt) AS aa
 GROUP BY titleid) as ti ON ti.titleid = Title.id) r1) to
'C:/Users/Revaa/Desktop/jsonfiles/Movies.json' with (FORMAT TEXT, HEADER false);
```

Creating Members.json file:

```
COPY (SELECT json_strip_nulls(row_to_json(r2))FROM(
SELECT id as _id, name, birthyear, deathyear FROM Member)r2)
TO 'C:/Users/Revaa/Desktop/jsonfiles/Members.json' with (FORMAT TEXT, HEADER false);
```

Loading the json files into mongodb:

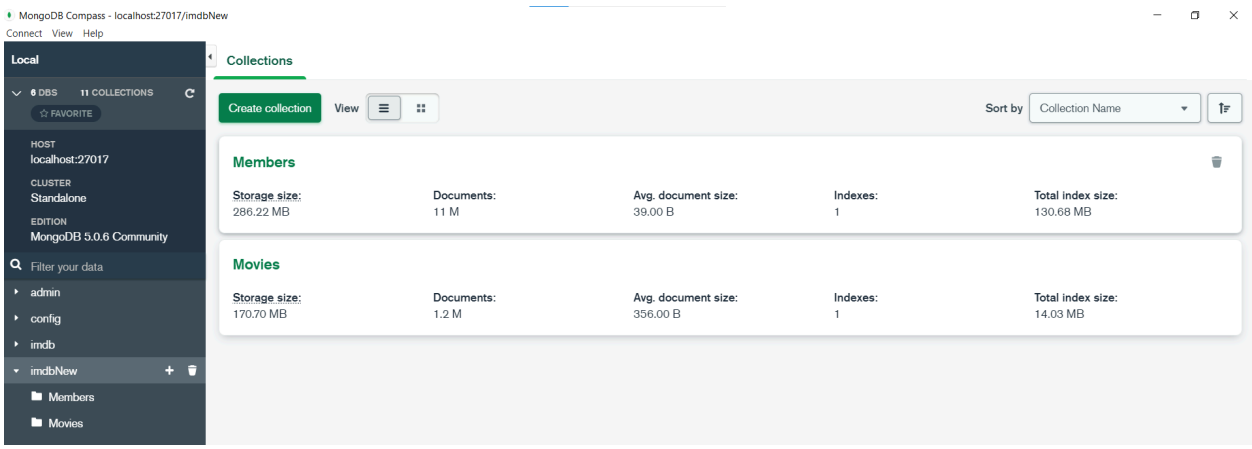
Movies.json:

C:\Users\Revaa>mongoimport- -db imdbNew --collection Movies --file C:\Users\Revaa\Desktop\jsonfiles\Movies.json

Members.json:

C:\Users\Revaa>mongoimport- -db imdbNew --collection Members --file C:\Users\Revaa\Desktop\jsonfiles\Members.json

Collections in imdbNew database:

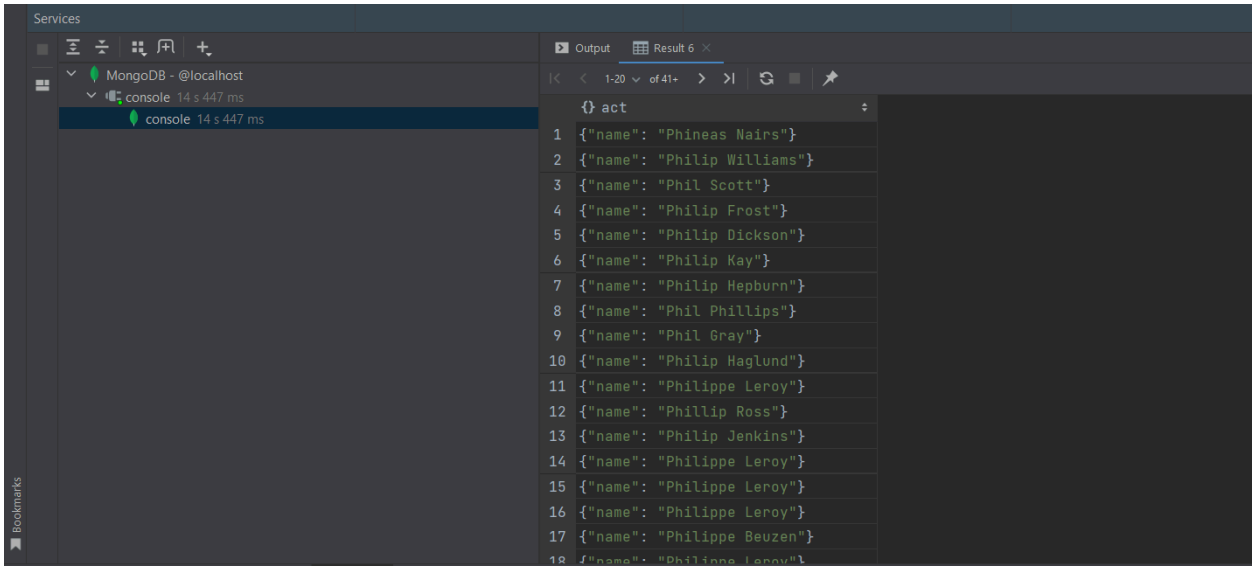


Q2.QUERIES:

2.1. Alive actors whose name starts with "Phi" and did not participate in any movie in 2014.

```
db.Movies.aggregate([{
 $match:{"startyear":{$ne:2014}}},
 {$lookup:
 {from: "Members",
 localField:"actors.actor",
 foreignField:"_id",
 as:"act"}},
 {$unwind: "$act"},
 {$match:{"act.name":/^Phi/,"act.deathyear":null}},
 {$project:{"_id":0,"act.name":1}}]);
```

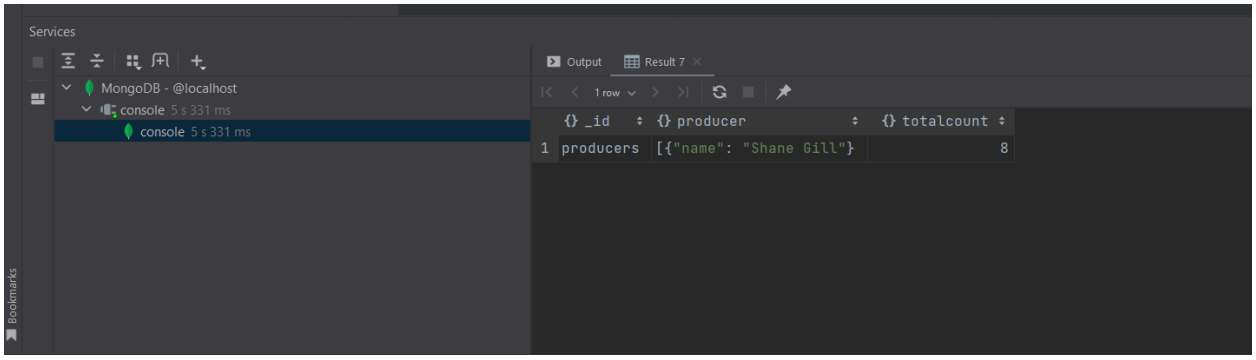
Total time: 14s 447 ms



2.2. Producers who have produced more than 50 talk shows in 2017 and whose name contain “Gill”.

```
db.Movies.aggregate([{
 $match:{"startyear":{$eq:2017},"genres":{$eq:"Talk-Show"}}},
 {$lookup:
 {from: "Members",
 localField:"producers",
 foreignField:"_id",
 as:"act"}},
 {$unwind: "$act"},
 {$match:{"act.name":{"$regex":/Gill/}}},
 {$group: { _id: "producers", totalcount: {$count:{}},
   producer:{$push:{name:"$act.name"}}}}]);
```

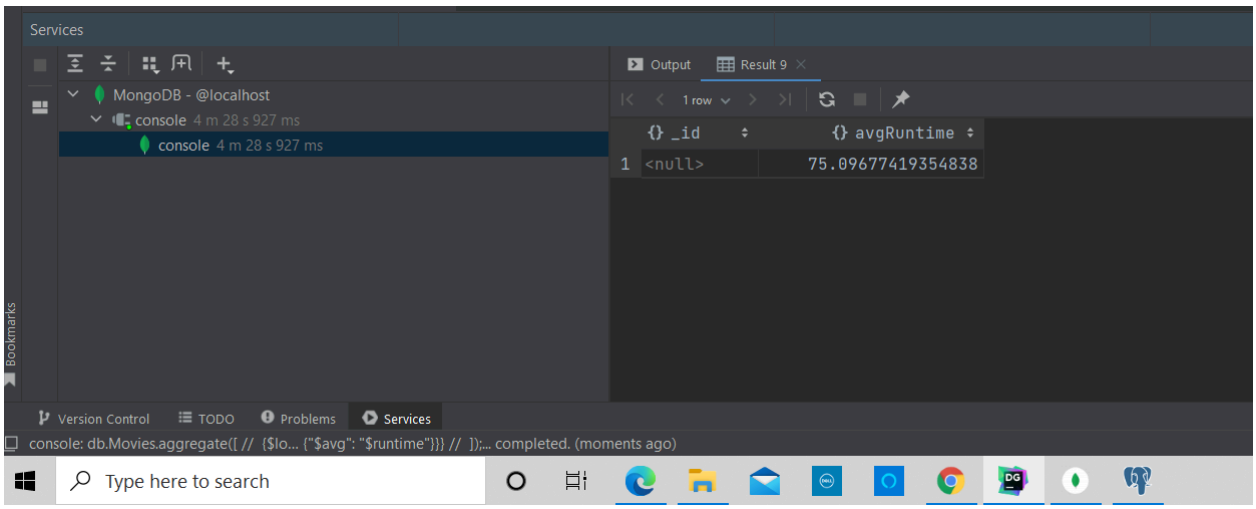
Total time: 5s 331 ms



2.3. Average runtime for movies that were written by members whose names contain "Bhardwaj" and are still alive.

```
db.Movies.aggregate([
 {$lookup:
 {from: "Members",
 localField:"writers",
 foreignField:"_id",
 as:"act"}},
 {$unwind: "$act"},
 {$match:{"act.name":/Bhardwaj/, "act.deathyear": null}},
 {$group:{"_id": null, "avgRuntime": {"$avg": "$runtime"}}}]);
```

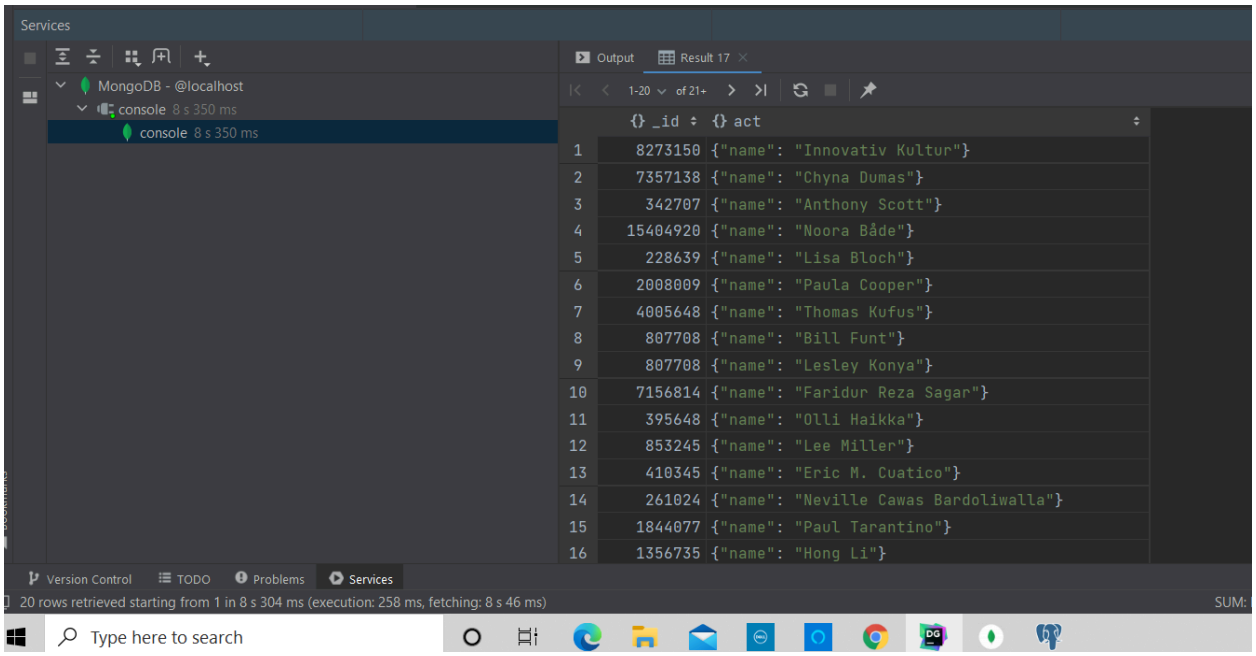
Total time:4m 28s 927ms



2.4. Alive producers with the greatest number of long-run movies produced (runtime greater than 120 minutes).

```
db.Movies.aggregate([{
 $match:{"runtime":{$gt:120}}},
 {$lookup:
 {from: "Members",
 localField:"producers",
 foreignField:"_id",
 as:"act"}},
 {$unwind: "$act"},
 {$match:{"act.deathyear":null}},
 {$sort:{"runtime":-1}},
 {$project:{"_id":1,"act.name":1}}]);
```

Total time: 8s 350ms



2.5. Sci-Fi movies directed by James Cameron and acted in by Sigourney Weaver.

```
db.Movies.aggregate([
  {$match:{"genres": {"$eq": "Sci-Fi"}}},
  {$lookup:
     {from: "Members",
     localField:"directors",
     foreignField:"_id",
     as:"result"}
     },
    {$match:{"result.name": {"$eq":"James Cameron"}}},
     {$lookup:
     {from: "Members",
     localField:"actors.actor",
     foreignField:"_id",
     as:"result1"}
     },
     {$match:{"result1.name": {"$eq":"Sigourney Weaver"}}},
     {$project:{_id:1, "title":1}}]);
```

Total time: 7s 350ms

Q3.

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

```
db.Movies.aggregate([{
 $match:{"startyear":{$ne:2014}}},
```

```
 {$lookup:
 {from: "Members",
 localField:"actors.actor",
 foreignField:"_id",
 as:"act"}},
 {$unwind: "$act"},
 {$match:{"act.name":/^Phi/,"act.deathyear":null}},
 {$project:{"_id":0,"act.name":1}}]).explain();
```

After using.explain() we get the different stages of the query in the form of a tree. There is a queryPlanner which contains namespace,indexFilterSet,queryHash,planCacheKey. The queryPlanner contains information about the query plan.queryPlanner.**namespace** contains the name of the name of the database and the collection that is being used. Here, the database is imdbNew and the collection is Movies**.IndexFilterSet** contains boolean values. It shows True if any index filter is applied, else False.The **queryHash** of queryPlanner contains the hash of the query shape and it is represented by a hexadecimal string. Slow queries can be identified with the help of queryHash.**planCacheKey** is the hash of the key for the plan cache which is related to the query.
explain.queryPlanner.**winningPlan** is a document that contains the plan selected by the query optimizer.There are no rejected plans in our query. **executionStats** gives us the information about the execution of the winning plan.executionSuccess tells us if the query was executed properly or not.It contains boolean values.**executionTimeMillis** gives us the total time in milliseconds. Here the time taken was: **166321**.The total number of documents examined is given by **totalDocsExamined**. Here the **totalDocsExamined were: 1188090**.The number of documents that match the query which was executed is given by explain.**nReturned**. execution**TimeMillisEstimate** gives the estimated time in milliseconds for the query and for this query it was found to be **80ms**.the total number of work units performed by the query execution stage is given by **works**.For this query **works:1188092.** The number of work cycles that did not advance an intermediate result is given by the **needTime** which is equal to **46711**. **needYield** contains the number of times the storage layer requested that the query stage suspend processing and yield its locks which was 0 for this query.**restoredState** gives the number of times the query stage restored a saved execution state.Here the restoredState was equal to 1247.**isEOF** specifies the end for the execution stage.It contains boolean values. **serverInfo** contains information about the port,host,version etc.

3.2. Producers who have produced more than 50 talk shows in 2017 and whose name contain “Gill”.

```
db.Movies.aggregate([{
 $match:{"startyear":{$eq:2017},"genres":{$eq:"Talk-Show"}}},
 {$lookup:
 {from: "Members",
 localField:"producers",
 foreignField:"_id",
 as:"act"}},
 {$unwind: "$act"},
 {$match:{"act.name":{"$regex":/Gill/}}},
```

```
{$group: { _id: "producers", totalcount: {$count:{}},
  producer:{$push:{name:"$act.name"}}}}].explain();
```

By using.explain() we get the different stages of the query in the form of a tree. There is a queryPlanner which contains namespace,indexFilterSet,queryHash,planCacheKey. The queryPlanner contains information about the query plan.queryPlanner.**namespace** contains the name of the name of the database and the collection that is being used. Here, the database is imdbNew and the collection is Movies**.IndexFilterSet** contains boolean values. It shows True if any index filter is applied, else False.The **queryHash** of queryPlanner contains the hash of the query shape and it is represented by a hexadecimal string. Slow queries can be identified with the help of queryHash.**planCacheKey** is the hash of the key for the plan cache which is related to the query.
explain.queryPlanner.**winningPlan** is a document that contains the plan selected by the query optimizer.There are no rejected plans in our query. **executionStats** gives us the information about the execution of the winning plan.executionSuccess tells us if the query was executed properly or not.It contains boolean values.**executionTimeMillis** gives us the total time in milliseconds. Here the time taken was: **2926**.The total number of documents examined is given by **totalDocsExamined**. Here the **totalDocsExamined were: 1188090**.The number of documents that match the query which was executed is given by explain.**nReturned**. execution**TimeMillisEstimate** gives the estimated time in milliseconds for the query and for this query it was found to be **373ms**.the total number of work units performed by the query execution stage is given by **works**.For this query **works:1188092.** The number of work cycles that did not advance an intermediate result is given by the **needTime** which is equal to **1185215**. **needYield** contains the number of times the storage layer requested that the query stage suspend processing and yield its locks which was **0** for this query.**restoreState** gives the number of times the query stage restored a saved execution state.Here the restoredState was equal to **1189**.**isEOF** specifies the end for the execution stage.It contains boolean values.For the query isEOF:1. **serverInfo** contains information about the port,host,version etc.

3.3. Average runtime for movies that were written by members whose names contain "Bhardwaj" and are still alive.

```
db.Movies.aggregate([
 {$lookup:
 {from: "Members",
 localField:"writers",
 foreignField:"_id",
 as:"act"}},
 {$unwind: "$act"},
 {$match:{"act.name":/Bhardwaj/, "act.deathyear": null}},
 {$group:{"_id": null, "avgRuntime": {"$avg": "$runtime"}}}]).explain();
```

By using.explain() we get the different stages of the query in the form of a tree. There is a queryPlanner which contains namespace,indexFilterSet,queryHash,planCacheKey. The queryPlanner contains information about the query plan.queryPlanner.**namespace** contains the name of the name of the database and the collection that is being used. Here, the database is imdbNew and the collection is Movies**.IndexFilterSet** contains boolean values. It shows True if

any index filter is applied, else False.The **queryHash** of queryPlanner contains the hash of the query shape and it is represented by a hexadecimal string. Slow queries can be identified with the help of queryHash.**planCacheKey** is the hash of the key for the plan cache which is related to the query.
explain.queryPlanner.**winningPlan** is a document that contains the plan selected by the query optimizer.There are no rejected plans in our query. **executionStats** gives us the information about the execution of the winning plan.executionSuccess tells us if the query was executed properly or not.It contains boolean values.**executionTimeMillis** gives us the total time in milliseconds. Here the time taken was: **199391**.The total number of documents examined is given by **totalDocsExamined**. Here the **totalDocsExamined were: 1188090**.The number of documents that match the query which was executed is given by explain.**nReturned**. execution**TimeMillisEstimate** gives the estimated time in milliseconds for the query and for this query it was found to be **178ms**.the total number of work units performed by the query execution stage is given by **works**.For this query **works:1188092.** The number of work cycles that did not advance an intermediate result is given by the **needTime** which is equal to **1185215**. **needYield** contains the number of times the storage layer requested that the query stage suspend processing and yield its locks which was **0** for this query.**restoreState** gives the number of times the query stage restored a saved execution state.Here the restoredState was equal to **1240**.**isEOF** specifies the end for the execution stage.The isEOF was 1.It contains boolean values. **serverInfo** contains information about the port,host,version etc.

3.4. Alive producers with the greatest number of long-run movies produced (runtime greater than 120 minutes).

```
db.Movies.aggregate([{
 $match:{"runtime":{$gt:120}}},
 {$lookup:
 {from: "Members",
 localField:"producers",
 foreignField:"_id",
 as:"act"}},
 {$unwind: "$act"},
 {$match:{"act.deathyear":null}},
 {$sort:{"runtime":-1}},
 {$project:{"_id":1,"act.name":1}}]).explain();
```

Using.explain() we get the different stages of the query in the form of a tree. There is a queryPlanner which contains namespace,indexFilterSet,queryHash,planCacheKey. The queryPlanner contains information about the query plan.queryPlanner.**namespace** contains the name of the name of the database and the collection that is being used. Here, the database is imdbNew and the collection is Movies**.IndexFilterSet** contains boolean values. It shows True if any index filter is applied, else False.The **queryHash** of queryPlanner contains the hash of the query shape and it is represented by a hexadecimal string. Slow queries can be identified with the help of queryHash.**planCacheKey** is the hash of the key for the plan cache which is related to the query.

explain.queryPlanner.**winningPlan** is a document that contains the plan selected by the query optimizer.There are no rejected plans in our query. **executionStats** gives us the information about the execution of the winning plan.executionSuccess tells us if the query was executed properly or not.It contains boolean values.**executionTimeMillis** gives us the total time in milliseconds. Here the time taken was: **4954**.The total number of documents examined is given by **totalDocsExamined**. Here the **totalDocsExamined were: 1188090**.The number of documents that match the query which was executed is given by explain.**nReturned**. execution**TimeMillisEstimate** gives the estimated time in milliseconds for the query and for this query it was found to be **46ms**.the total number of work units performed by the query execution stage is given by **works**.For this query **works:1188092.** The number of work cycles that did not advance an intermediate result is given by the **needTime** which is equal to **1151893**. **needYield** contains the number of times the storage layer requested that the query stage suspend processing and yield its locks which was **0** for this query.**restoreState** gives the number of times the query stage restored a saved execution state.Here the restoredState was equal to **1190**.**isEOF** specifies the end for the execution stage.The isEOF was 1.It contains boolean values. **serverInfo** contains information about the port,host,version,gitversion and the serverParameters contains the information about the usage of bytes and buffer size.

3.5. Sci-Fi movies directed by James Cameron and acted in by Sigourney Weaver.

```
db.Movies.aggregate([
  {$match:{"genres": {"$eq": "Sci-Fi"}}},
  {$lookup:
     {from: "Members",
     localField:"directors",
     foreignField:"_id",
     as:"act"}
     },
    {$match:{"act.name": {"$eq":"James Cameron"}}},
     {$lookup:
     {from: "Members",
     localField:"actors.actor",
     foreignField:"_id",
     as:"act1"}
     },
     {$match:{"act1.name": {"$eq":"Sigourney Weaver"}}},
     {$project:{_id:1, "title":1}}]).explain();
```

Using.explain() we get the different stages of the query in the form of a tree. There is a queryPlanner which contains namespace,indexFilterSet,queryHash,planCacheKey. The queryPlanner contains information about the query plan.queryPlanner.**namespace** contains the name of the name of the database and the collection that is being used. Here, the database is imdbNew and the collection is Movies**.IndexFilterSet** contains boolean values. It shows True if any index filter is applied, else False.The **queryHash** of queryPlanner contains the hash of the query shape and it is represented by a hexadecimal string. Slow queries can be identified with the help of queryHash.**planCacheKey** is the hash of the key for the plan cache which is related to the query.

explain.queryPlanner.**winningPlan** is a document that contains the plan selected by the query optimizer.There are no rejected plans in our query. **executionStats** gives us the information about the execution of the winning plan.executionSuccess tells us if the query was executed properly or not.It contains boolean values.**executionTimeMillis** gives us the total time in milliseconds. Here the time taken was: **2847**.The total number of documents examined is given by **totalDocsExamined**. Here the **totalDocsExamined were: 1188090**.The number of documents that match the query which was executed is given by explain.**nReturned**. execution**TimeMillisEstimate** gives the estimated time in milliseconds for the query and for this query it was found to be **49ms**.the total number of work units performed by the query execution stage is given by **works**.For this query **works:1188092.** The number of work cycles that did not advance an intermediate result is given by the **needTime** which is equal to **1157865**. **needYield** contains the number of times the storage layer requested that the query stage suspend processing and yield its locks which was **0** for this query.**restoreState** gives the number of times the query stage restored a saved execution state.Here the restoredState was equal to **1190**.**isEOF** specifies the end for the execution stage.The isEOF was 1.It contains boolean values. **serverInfo** contains information about the port,host,version,gitversion and the serverParameters contains the information about the usage of bytes and buffer size.

Q4.Creating Indexes

Indexing helps in improving 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 the queries are first executed without indexes and the time is noted. Then we create indexes for the tables and then run the same queries. We can see that after executing the queries with indexes the time to execute them has reduced.Before indexes the queries took more time to execute.

```
db.Movies.createIndex(
 { runtime: 1}
 )

db.Movies.createIndex(
  { startyear: -1}
)

db.Movies.createIndex(
{genre: -1}
)

db.Members.createIndex(
 { name: 1}
)

db.Members.createIndex(
 { name: 1, deathyear: -1}
)
```

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

```
db.Movies.aggregate([{
 $match:{"startyear":{$ne:2014}}},
 {$lookup:
 {from: "Members",
 localField:"actors.actor",
 foreignField:"_id",
 as:"act"}},
 {$unwind: "$act"},
 {$match:{"act.name":/^Phi/,"act.deathyear":null}},
 {$project:{"_id":0,"act.name":1}}]);
```

Previous Total time: 14s 447 ms
Total time after index: 5s 761 ms

4.2. Producers who have produced more than 50 talk shows in 2017 and whose name contain “Gill”.

```
db.Movies.aggregate([{
 $match:{"startyear":{$eq:2017},"genres":{$eq:"Talk-Show"}}},
 {$lookup:
 {from: "Members",
 localField:"producers",
 foreignField:"_id",
 as:"act"}},
 {$unwind: "$act"},
 {$match:{"act.name":{"$regex":/Gill/}}},
 {$group: { _id: "producers", totalcount: {$count:{}},
   producer:{$push:{name:"$act.name"}}}}]);
```

Previous Total time: 5s 331 ms
Total time after index: 751 ms

4.3. Average runtime for movies that were written by members whose names contain “Bhardwaj” and are still alive.

```
db.Movies.aggregate([
 {$lookup:
 {from: "Members",
 localField:"writers",
 foreignField:"_id",
 as:"act"}},
 {$unwind: "$act"},
 {$match:{"act.name":/Bhardwaj/, "act.deathyear": null}},
 {$group:{"_id": 0, "avgRuntime": {"$avg": "$runtime"}}}]);
```

Previous Total time:4m 28s 927ms
Total time after index:2m 38s 354 ms

4.4. Alive producers with the greatest number of long-run movies produced (runtime greater than 120 minutes).

```
db.Movies.aggregate([{
 $match:{"runtime":{$gt:120}}},
 {$lookup:
 {from: "Members",
 localField:"producers",
 foreignField:"_id",
 as:"act"}},
 {$unwind: "$act"},
 {$match:{"act.deathyear":null}},
 {$sort:{"runtime":-1}},
 {$project:{"_id":1,"act.name":1}}]);
```

Previous Total time: 8s 350ms
Total time after index: 4s 200 ms

4.5. Sci-Fi movies directed by James Cameron and acted in by Sigourney Weaver.

```
db.Movies.aggregate([
  {$match:{"genres": {"$eq": "Sci-Fi"}}},
  {$lookup:
     {from: "Members",
     localField:"directors",
     foreignField:"_id",
     as:"act"}
     },
    {$match:{"act.name": {"$eq":"James Cameron"}}},
     {$lookup:
     {from: "Members",
     localField:"actors.actor",
     foreignField:"_id",
     as:"act1"}
     },
     {$match:{"act1.name": {"$eq":"Sigourney Weaver"}}},
     {$project:{_id:1, "title":1}}]);
```

Total time: 7s 350ms