COMP5338 - Advanced Data Models

Assignment: Polyglot Persistence using NoSQL Systems

Xuhong Guo (450489321),
Zhongnan Li (450585656),
Weijie Liang (450598885)

June 30, 2017

Contents

1 Introduction

In recent years, with the demand for storing and analysing big data, a number of systems have emerged to provide good scalability for data storage and simple read/write database operations distributed over several servers, and many of these new systems are referred to as "NoSQL (Not Only SQL)" data stores [1]. These systems differ from one another and have their corresponding advantages and characteristics. In this assignment, three NoSQL systems, MongoDB, Neo4j and HBase are used respectively to store the same data set and run a list of target queries. The data set is a music data set (Last.fm (http://www.last.fm)) released in HetRec 2011 (http://ir.ii.uam.es/hetrec2011/). In this data set, information about "social networking, tagging, and music artist listening information from a set of 2k users from Last.fm online music system" is contained.
For each of the three NoSQL systems, the schema designed for storing this data set and running target queries is described, the query sentences and query results are illustrated, and the query performance and statistics are also explained. At last, MongoDB, Neo4j and HBase are compared in terms of ease of use, query performance and schema difference to further distinguish the characteristics of the three systems.

2 MongoDB

As a representative of NoSQL database management systems and released in 2009, MongoDB stores JSON-like documents which have dynamic schemas (BSON). MongoDB has advantages in terms of speed, power, flexibility and ease of use, and it also supports indexing to improve query performance and provides drives for various programming languages such as Java, Python, etc. [2].

2.1 Data Set

Access 2016 is used to merge tables. In order to implement all the queries, all the data is reconstructed and three tables are obtained after merging.

1. The first table stores tag-related information, and it contains the fields of id, name (artist name), url, pictureURL, userID, tagID, tagValue and timestamp. Table 1 shows an example entry of the first table and the corresponding SQL sentence in Access 2016 is:

```
SELECT artists.id, artists.name, artists.url, artists.pictureURL,
 ↪ user_taggedartists-timestamps.userID, user_taggedartists-timestamps.tagID,
 ↪ tags.tagValue, user_taggedartists-timestamps.timestamp
FROM (user_taggedartists-timestamps LEFT JOIN tags ON
 ↪ user_taggedartists-timestamps.tagID = tags.tagID) RIGHT JOIN artists ON
 ↪ user_taggedartists-timestamps.artistID = artists.id;
```

id	name	url	pictureURL	userID	tagID	tagValue	timestamp
1	MALICE MIZER	http://www.la	http://user	1984	139	j-rock	1.201820E+12

Table 1: Tag

2. In the second table, data related to weight is stored, and the columns are respectively id, name (artist name), url, pictureURL, userID and weight. Table 2 shows an example entry (duplicated data of the full data set is highlighted) and the corresponding SQL sentence in Access 2016 is:

```
SELECT artists.id, artists.name, artists.url, artists.pictureURL,
 ↪  user_artists.userID, user_artists.weight
FROM artists LEFT JOIN user_artists ON artists.id = user_artists.artistID;
```

id	name	url	pictureURL	userID	weight
1	MALICE MIZER	http://www.la	http://user	212	3452

Table 2: Weight

3. The third table stores friend-related information. An example entry is as shown in Table 3.

userID	friendID
2	275

Table 3: Friend

The first two tables Tag and Weight are converted into JSON files using an online converter.

2.2 Schema

In MongoDB, the three tables in Section 2.1 are stored as collections Tag, Weight and Friend. The schema of the three collections will be shown as follows:

1. One example document in collection Tag is:

```
{
  "id": 1,
  "name": "MALICE MIZER",
  "url": "http://www.last.fm/music/MALICE+MIZER",
  "pictureURL": "http://userserve-ak.last.fm/serve/252/10808.jpg
    ",
  "userID": 785,
  "tagID": 2850,
  "tagValue": "better than lady gaga",
  "timestamp": 1280610000000
}
```

   This document contains eight fields: “id” (artist ID) which is the primary key, “name” which indicates the artist name, “url” and “pictureURL” which relate to information corresponding to the artist, “userID” which indicates the user who listens to this artist, “tagID” which indicates the tag given by the user, “tagValue” which shows the value corresponding to the tagID, and “timestamp” which indicates the time when the user has assigned the tag corresponding to the tagID to the artist, while the unix timestamp is used in this document.

2. One example document in collection Weight is:

```
{
    "id": 1,
    "name": "MALICE MIZER",
    "url": "http://www.last.fm/music/MALICE+MIZER",
    "pictureURL": "http://userserve-ak.last.fm/serve/252/10808.jpg
       ",
    "userID": 785,
    "weight": 76
 }
```

This document contains six fields: "id" (artist ID) which is the primary key, "name" which indicates the artist name, "url" and "pictureURL" which relate to information corresponding to the artist, "userID" which indicates the user who listens to this artist, and "weight" which indicates the listening count of the user listening to this artist. The fields "name", "url" and "pictureURL" in collection Tag are duplicated and also appear in this collection for ease of query.

3. One example document in collection Friend is:

```
{
    "id": 1,
    "friend": 275
 }
```

This document only contains two fields: "id" (userID) which is the primary key, and "friend" which indicates the friend ID of the user with the corresponding "id".

2.3 Solution

All the MongoDB query sentences are embedded in Python.

2.3.1 Simple Query

1. Given a user id, find all artists the user's friends listen.

```
friend_artist = []
userID = int(input('   Input the userID: '))
# find user's friend
for doc in friend.find({'userID':userID}):
    # find artists listened by user's friends
    for artist in weight.find({'userID':
     ↪  doc['friendID']},{'id':1,'name':1,'url':1,'pictureURL':1}):
        print(artist)

Input the userID: 2
{'name': 'And One', '_id': ObjectId('57e0a407bbaa62790cfc9fc1'), 'id': 30,
 ↪  'pictureURL': 'http://userserve-ak.last.fm/serve/252/50818861.jpg', 'url':
 ↪  'http://www.last.fm/music/And+One'}
```

```
{'name': 'Duran Duran', '_id': ObjectId('57e0a408bbaa62790cfca0b1'), 'id': 51,
 ↪  'pictureURL': 'http://userserve-ak.last.fm/serve/252/155668.jpg', 'url':
 ↪  'http://www.last.fm/music/Duran+Duran'}
{'name': 'New Order', '_id': ObjectId('57e0a40ebbaa62790cfca3c5'), 'id': 59,
 ↪  'pictureURL': 'http://userserve-ak.last.fm/serve/252/6650979.jpg', 'url':
 ↪  'http://www.last.fm/music/New+Order'}
...
```

This query includes two steps: step 1, querying collection Friend to find friends of a given user; and step 2, querying collection Weight to find artists to whom the user's friends listen.

- Execution Statistics with Default Index
  From Figure 1, it can be seen that in order to find the user's friends, 25,434 documents

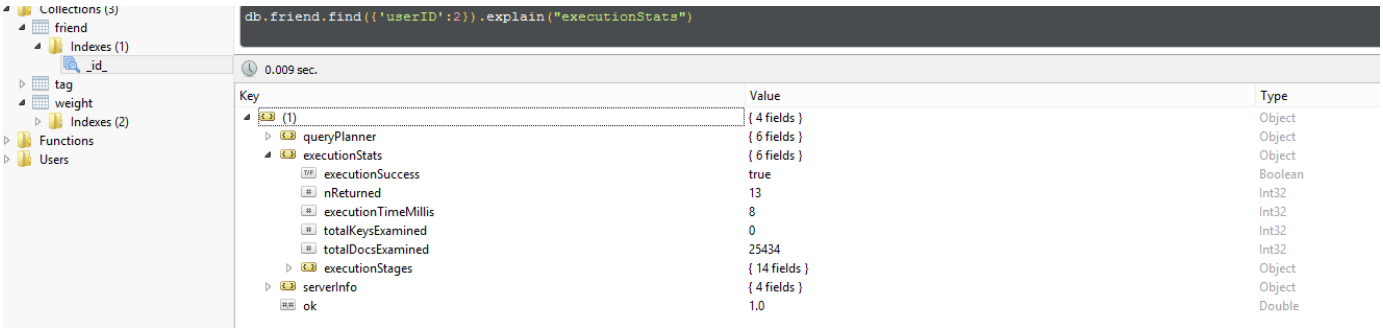

Figure 1: Simple Query 1 with Default Index in Collection Friend

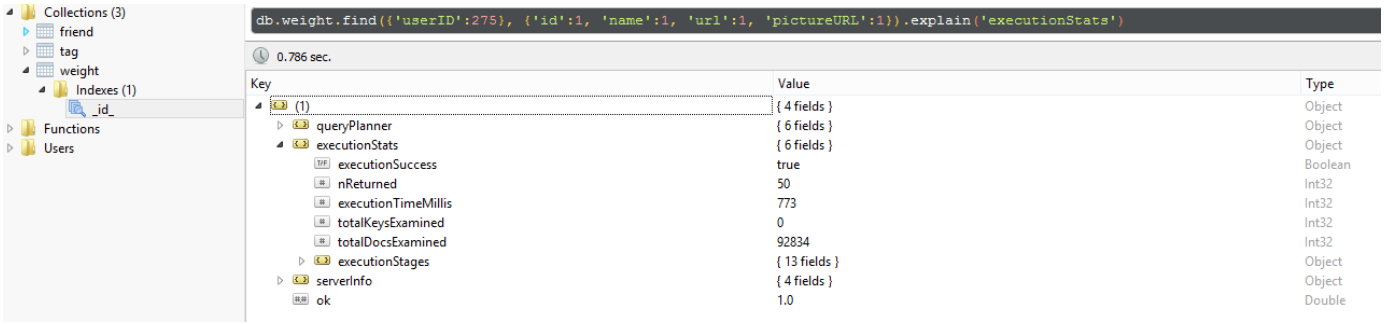

Figure 2: Simple Query 1 with Default Index in Collection Weight

need to be examined to return 13 results; and from Figure 2, taking friend 275 of user2 as an example to illustrate the query performance, 92,834 documents are examined to return 50 results.

- Execution Statistics with Created Index
  Since the query specifies "given a userID", "userID" is chosen to create an index for both collections Friend and Weight. From Figure 3, only 13 documents are examined to return 13 results, and Figure 4 shows that only 50 documents are examined to return 50 results. However, in terms of query time, when querying collection Friend, the query time (8) with the default index is shorter than that (18) with the created index "userID", while in the query of collection Weight, the query time (773) with the default index is much longer than that (48) with the created index "userID".

Therefore, it is obvious that the number of documents examined is less than that without creating a new index, whereas the query time does not always show corresponding performance improvement.

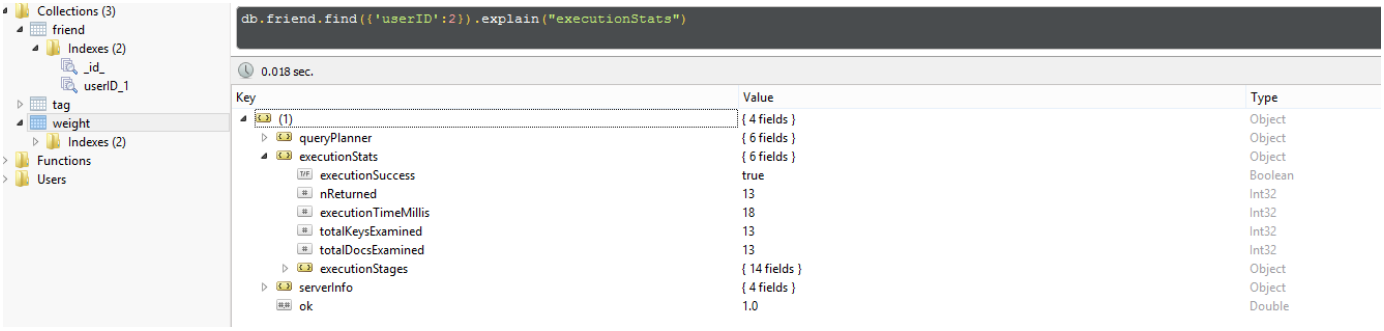

Figure 3: Simple Query 1 with Index "userID" in Collection Friend

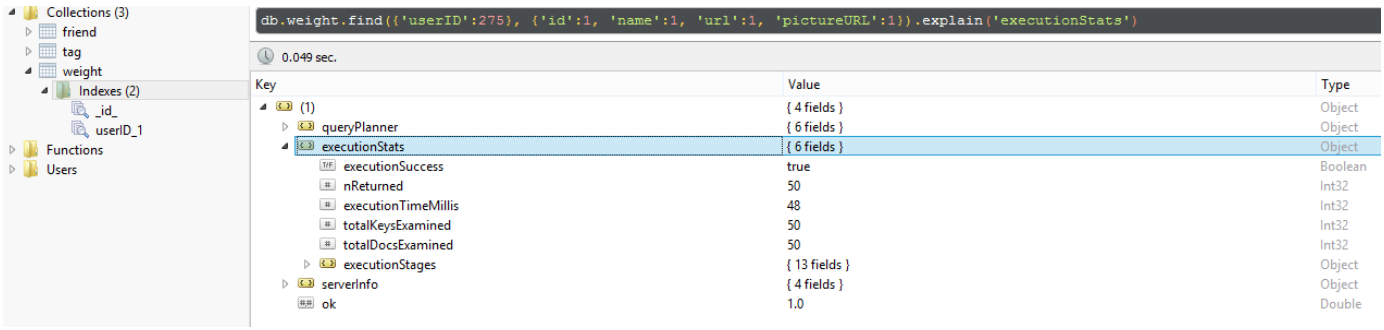

Figure 4: Simple Query 1 with Index "userID" in Collection Weight

2. Given an artist name, find the most recent 10 tags that have been assigned to it.

```
tagValue_matrix = []
artist_name = input('   Input the artist name: ')
for tagValue in tag.find({'name': artist_name}, {'tagValue':
 ↪  1}).sort("timestamp", -1).limit(10):
    tagValue_matrix.append(tagValue['tagValue'])
print('   The most recent 10 recent 10 tags that have been assigned to it:')
print('  ',tagValue_matrix)
```

```
Input the artist name: Diary of Dreams
['german', 'darkwave', 'gothic', 'german', 'darkwave', 'gothic', 'gothic rock',
 ↪  'industrial', 'german', 'darkwave']
```

In this query, given the artist name "Diary of Dreams", collection Tag is queried to find all the tags assigned to this artist, then the tags are sorted according to timestamp in a descending order, and finally the latest 10 tags are displayed.

- Execution Statistics with Default Index
  Figure 5 shows that 190,440 documents are examined to return 27 results (27 tags), and the query time is relatively long (1,798).

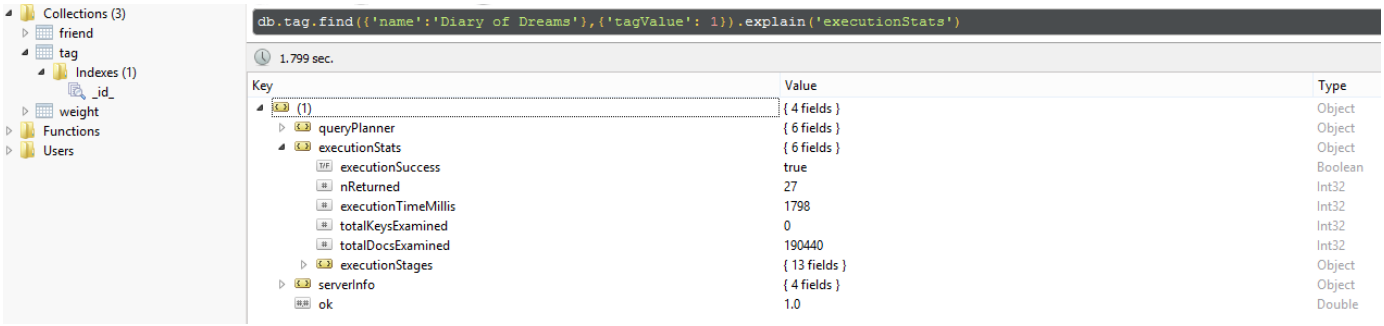

Figure 5: Simple Query 2 with Default Index in Collection Tag

- Execution Statistics with Created Index
  Since the query specifies "given an artist name", "name" is chosen to create an index for

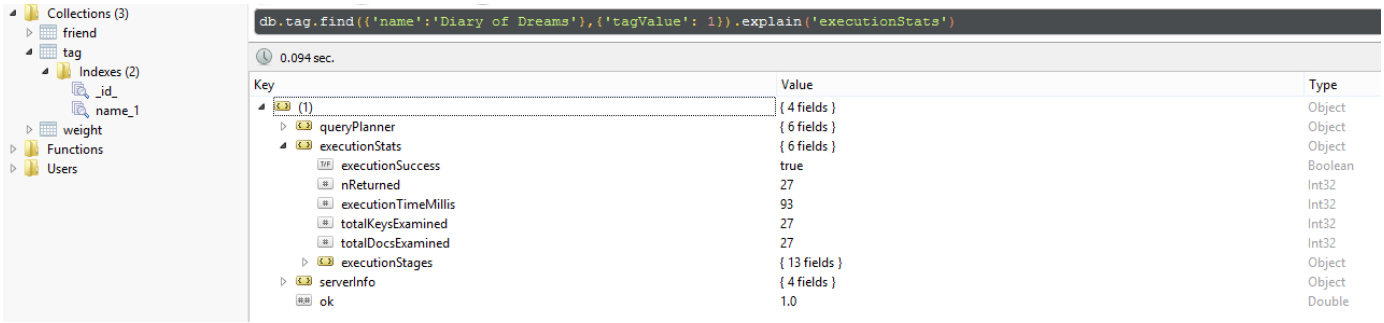

Figure 6: Simple Query 2 with Index "name" in Collection Tag

  collection Tag.
  Figure 6 shows that both the documents examined (27) and the query time (93) are much less than those with the default index, which indicates that the query performance is improved significantly.

3. Given an artist name, find the top 10 users based on their respective listening counts of this artist. Display both the user id and the listening count.

```
artist_name = input('   Input the artist name: ')
for user_ID in weight.find({'name':
↪   artist_name},{'userID':1,'weight':1}).sort("weight", -1).limit(10):
    print(user_ID)
```

```
Input the artist name: Diary of Dreams
{'userID': 325, '_id': ObjectId('57e0a404bbaa62790cfc9e47'), 'weight': 3466}
{'userID': 135, '_id': ObjectId('57e0a403bbaa62790cfc9e44'), 'weight': 1021}
{'userID': 1551, '_id': ObjectId('57e0a404bbaa62790cfc9e4b'), 'weight': 868}
{'userID': 580, '_id': ObjectId('57e0a403bbaa62790cfc9e42'), 'weight': 803}
{'userID': 1604, '_id': ObjectId('57e0a404bbaa62790cfc9e49'), 'weight': 455}
{'userID': 935, '_id': ObjectId('57e0a403bbaa62790cfc9e41'), 'weight': 428}
{'userID': 1601, '_id': ObjectId('57e0a404bbaa62790cfc9e4a'), 'weight': 385}
{'userID': 1679, '_id': ObjectId('57e0a404bbaa62790cfc9e48'), 'weight': 161}
{'userID': 257, '_id': ObjectId('57e0a403bbaa62790cfc9e45'), 'weight': 152}
{'userID': 560, '_id': ObjectId('57e0a403bbaa62790cfc9e43'), 'weight': 134}
```

In this query, given the artist name "Diary of Dreams", collection Weight is queried to find all users who listen to this artist, then the users are sorted according to weight (listening count) in a descending order, and finally 10 users who listen to this artist most are displayed together with their respective listening counts.

- Execution Statistics with Default Index
  Figure 7 shows that 92,834 documents are examined to return 12 results (12 users), and

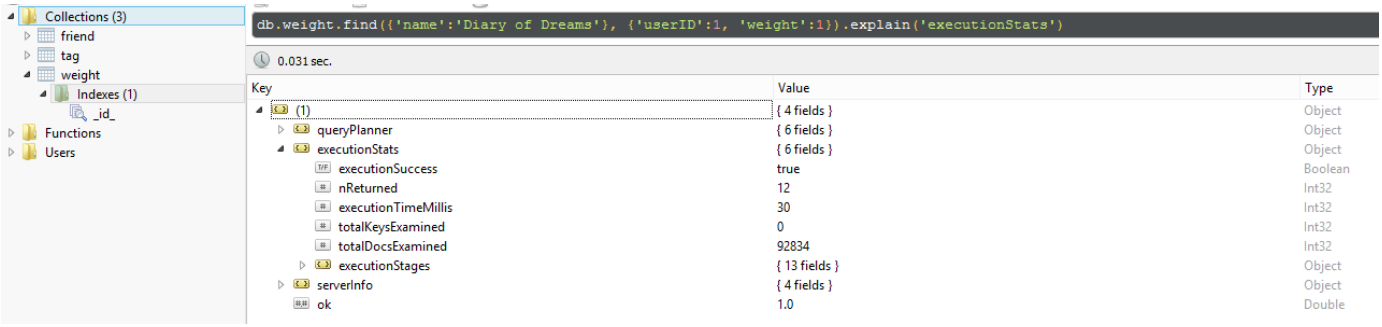

Figure 7: Simple Query 3 with Default Index in Collection Weight

the query time is 30.

- Execution Statistics with Created Index
  Since the query specifies "given an artist name", "name" is chosen to create an index for

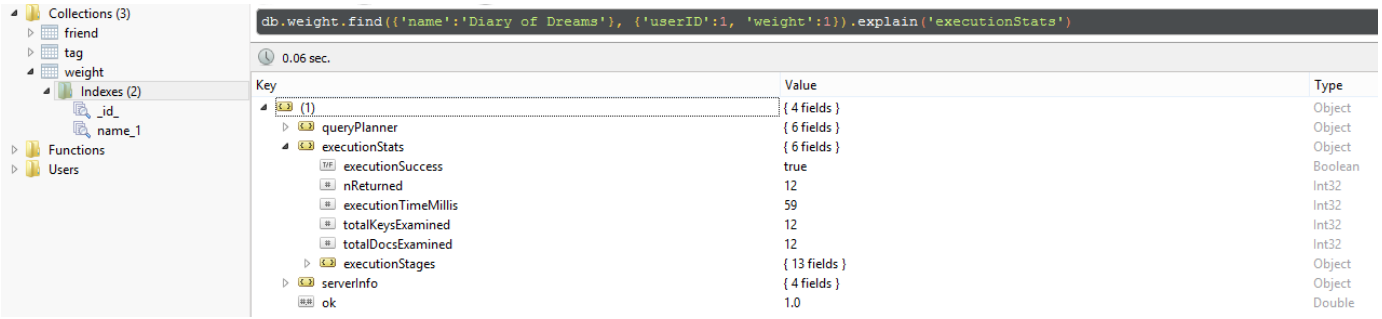

Figure 8: Simple Query 3 with Index "name" in Collection Weight

collection Weight.
Figure 8 shows that although the number of documents examined (12) is less than that with the default index, the query time (59) is longer.

4. Given a user id, find the most recent 10 artists the user has assigned tag to.

```
n = 0
k=''
userID = int(input('   Input the userID: '))
for artist_name in tag.find({'userID':userID},{'name':1,'userID':1,
'tagValue':1,'timestamp':1}).sort('timestamp',-1):
    if n<= 10 and k != artist_name['name']:
        print('    name: ', artist_name['name'], ' time: ',
         ↪  artist_name['timestamp'])
        n += 1
```

```
        k = artist_name['name']

Input the userID: 2
name:  China Crisis  time:  1241130000000
name:  Colin Newman  time:  1241130000000
name:  Loscil  time:  1241130000000
name:  Chicane  time:  1241130000000
name:  Sigue Sigue Sputnik  time:  1241130000000
name:  Morcheeba  time:  1238540000000
name:  Enigma  time:  1238540000000
name:  Caf Del Mar  time:  1238540000000
name:  Ministry of Sound  time:  1238540000000
name:  Fleetwood Mac  time:  1238540000000
```

In this query, collection Tag is queried to find the artists to whom a given user has assigned tags, and by sorting according to timestamp in a descending order, 10 artists to whom the user has assigned tags most recently are returned.

- Execution Statistics with Default Index
  Figure 9 shows that 190,440 documents are examined to return 45 results (45 artists), and

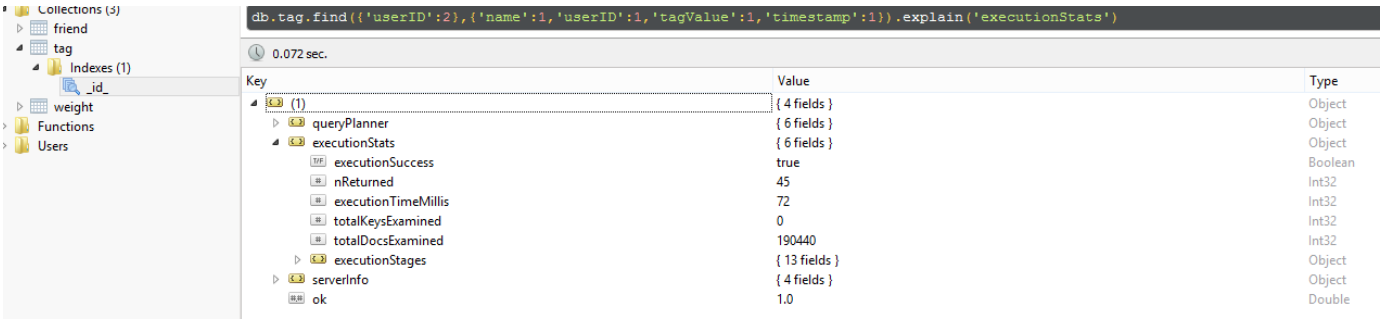

Figure 9: Simple Query 4 with Default Index in Collection Tag

the query time is 72.

- Execution Statistics with Created Index
  Since the query specifies "given a user id", "userID" is chosen to create an index for

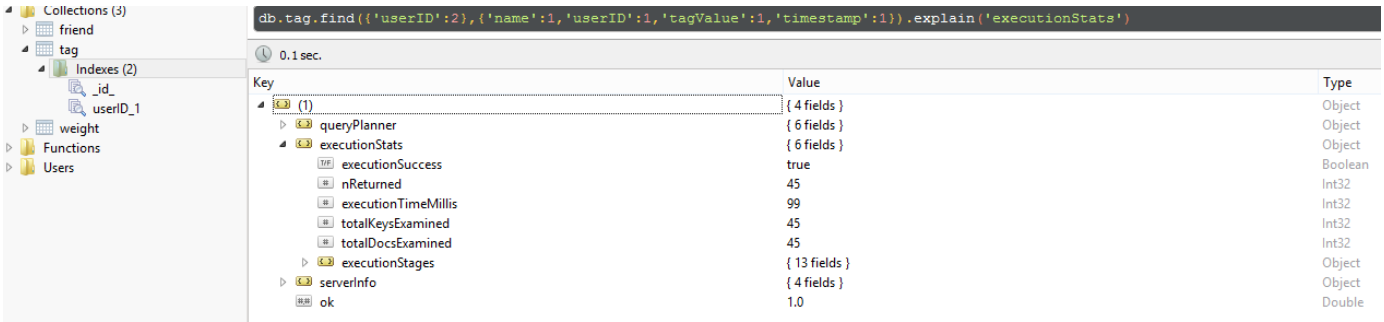

Figure 10: Simple Query 4 with Index "userID" in Collection Tag

collection Tag.
Figure 10 shows that although the number of documents examined (45) is much less than that with the default index, the query time (99) is slightly longer.

2.3.2 Complex Query

1. Find the top 5 artists ranked by the number of users listening to it.

```
for artist_name in
 ↪  weight.aggregate([{"$group":{'_id':'$name',"count":{'$sum':1}}},
{'$sort':{'count':-1}},{'$limit':5}]):
    print(artist_name)
```

```
{'count': 611, '_id': 'Lady Gaga'}
{'count': 522, '_id': 'Britney Spears'}
{'count': 484, '_id': 'Rihanna'}
{'count': 480, '_id': 'The Beatles'}
{'count': 473, '_id': 'Katy Perry'}
```

In this query, collection Weight is grouped by artist name and the artist's total listening count, then sorted by listening count in a descending order, and finally top 5 artists to whom users listen most are returned.

- Execution Statistics
  This query does not involve querying by a given field; therefore, no index is created. Figure 11 shows that this complex query requires a full collection scan.

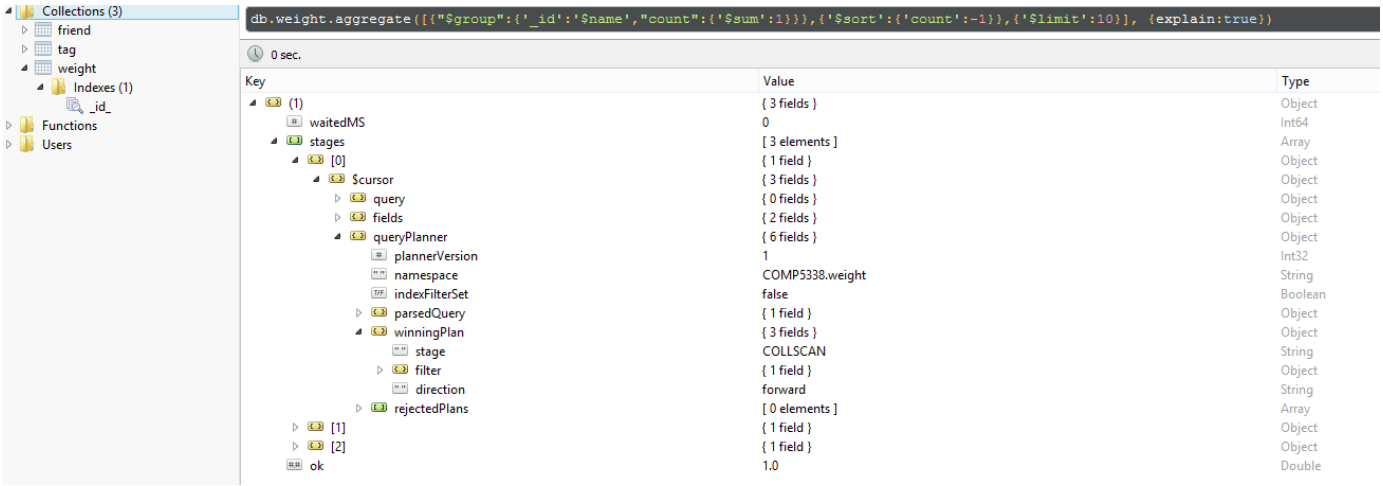

Figure 11: Complex Query 1 with Default Index in Collection Weight

2. Given an artist name, find the top 20 tags assigned to it. The tags are ranked by the number of times it has been assigned to this artist.

```
artist_name = input('   Input the artist name: ')
for tag_Value in
 ↪  tag.aggregate([{'$match':{'name':artist_name}},{'$group':{'_id':'$tagValue',
'count':{'$sum':1}}},{'$sort':{'count':-1}},{'$limit':20}]):
    print('  ',tag_Value)
```

```
Input the artist name: Diary of Dreams
{'count': 8, '_id': 'darkwave'}
```

```
{'count': 5, '_id': 'gothic'}
{'count': 5, '_id': 'german'}
{'count': 2, '_id': 'seen live'}
{'count': 1, '_id': 'gothic rock'}
{'count': 1, '_id': 'vocal'}
{'count': 1, '_id': 'electronic'}
{'count': 1, '_id': 'true goth emo'}
{'count': 1, '_id': 'industrial'}
{'count': 1, '_id': 'dark'}
{'count': 1, '_id': 'ambient'}
```

This query involves an aggregation pipeline in collection Tag. In the query pipeline, it starts with matching a given artist name ("Diary of Dreams" in this case), then group by tagValue and total tag count corresponding to each tag, next the returned tags are sorted according to count in a descending order, and finally top 20 tags assigned to this artist are returned.

- Execution Statistics with Default Index
  Figure 12 shows that this complex query involves a full collection scan with no new index

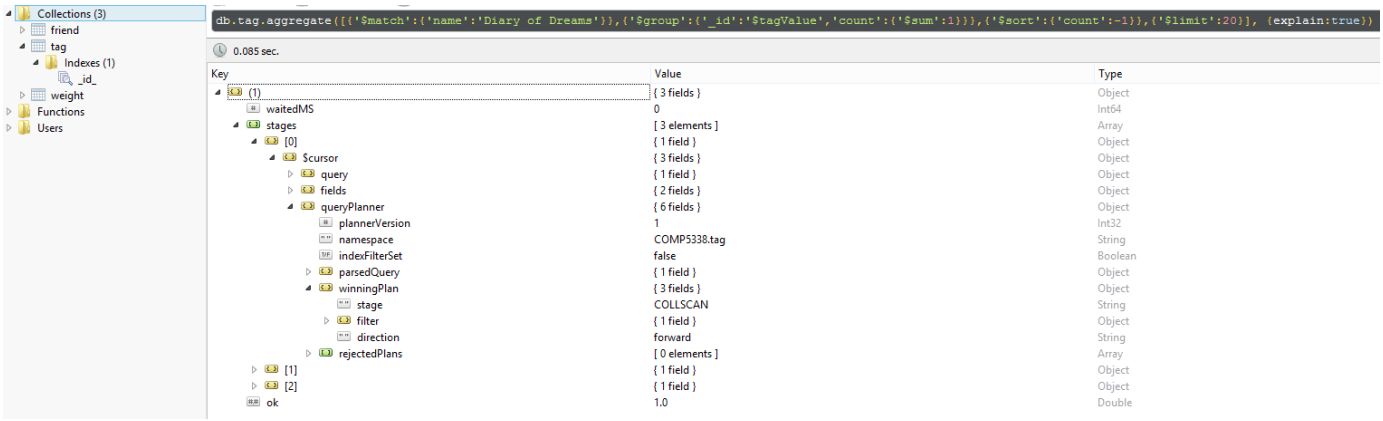

Figure 12: Complex Query 2 with Default Index in Collection Tag

created.

- Execution Statistics with Created Index
  Since the artist name needs to be matched in this query, "name" is created as an index in collection Tag. Figure 13 indicates that the query does not involve a full collection scan, but fetches data instead.

3. Given a user id, find the top 5 artists listened by his friends but not him. We rank artists by the sum of friends' listening counts of the artist.

```
friend_ID_matrix = []
self_artist_matrix = []
name_matrix = {}
userID = int(input('   Input the userID: '))
# find user's friends
for friend_ID in friend.find({'userID': userID}):
    friend_ID_matrix.append(friend_ID['friendID'])
# find artists listened by user
```

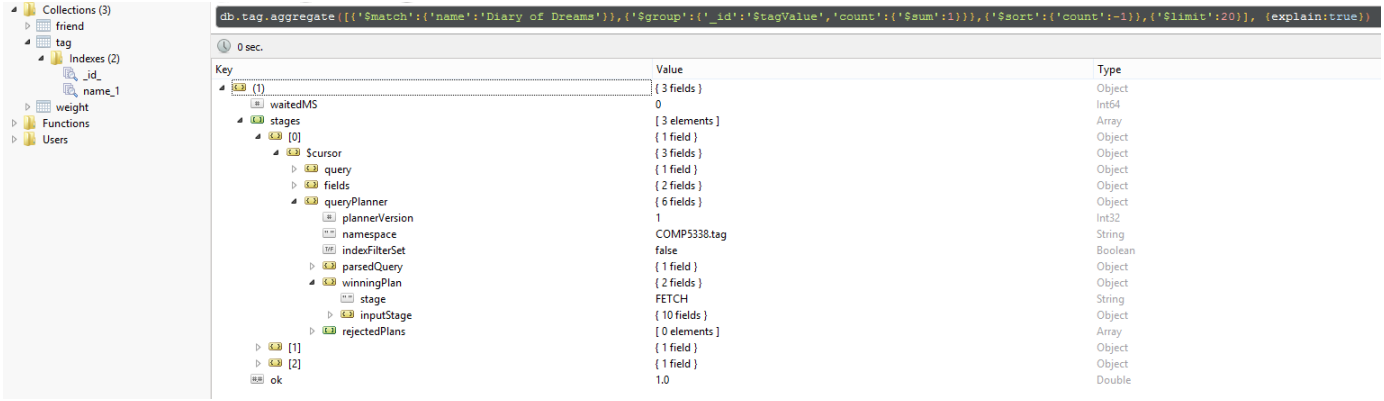

Figure 13: Complex Query 2 with Index "name" in Collection Tag

```
for self_artist in weight.find({'userID':userID}):
    self_artist_matrix.append(self_artist['name'])
count = 0
# find artists listened by user's friends
for user_weight in
 ↪  weight.aggregate([{'$match':{'userID':{'$in':friend_ID_matrix}}},{'$group':
{'_id':'$name','weight':{'$sum':'$weight'}}},{'$sort':{'weight':-1}},{'$limit':10}]):
    # exclude artists listened by user from artists listened by user's
     ↪  friends
    if user_weight['_id'] not in self_artist_matrix and count < 5:
       print(user_weight)
       count += 1
```

```
   Input the userID: 2
{'weight': 39369, '_id': 'Panic! At the Disco'}
{'weight': 36956, '_id': 'Rammstein'}
{'weight': 26447, '_id': 'U2'}
{'weight': 20776, '_id': 'The Cure'}
{'weight': 20596, '_id': 'Pet Shop Boys'}
```

This query involves two collections, Friend and Weight. It starts by finding friends of a given user in collection Friend and finding artists listened by the user in collection Weight, and then an aggregation pipeline is executed in collection Weight to find the artists listened by the user's friends and group the artists by total listening count (summing up weight), sort the artists according to weight in a descending order and limit the number of artists returned. Finally, the artists listened by the user are excluded from the list of artists listened by his friends and top 5 artists with the greatest listening count are returned.

- Execution Statistics with Default Index
  Figures 14 and 15 show the execution statistics of the first two steps with only the default index. When querying the user's friends, 25,434 documents are examined and 13 results are returned, and the query time is 8; and when querying the artists listened by the user, 92,834 documents are examined and 50 results are returned, and the query time is 33. Figure 16 shows that with only the default index and no new index created, the aggregation involves a full collection scan.

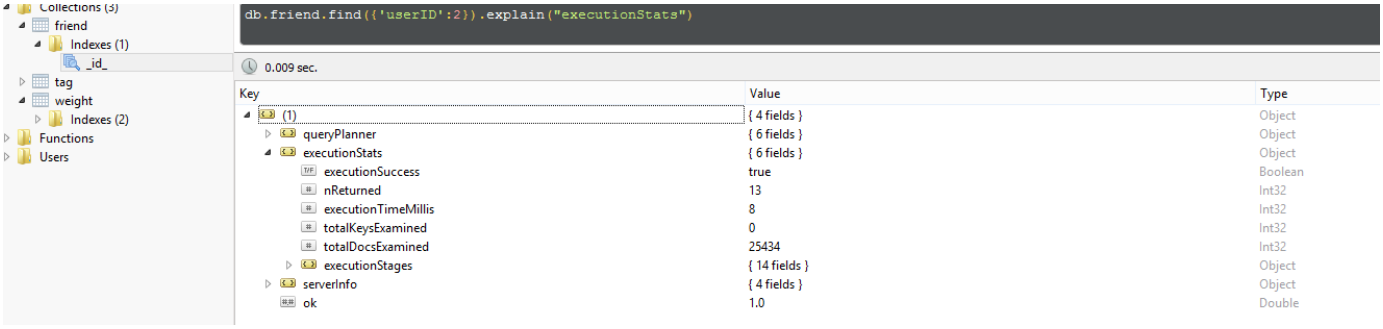

Figure 14: Complex Query 3 with Default Index in Collection Friend

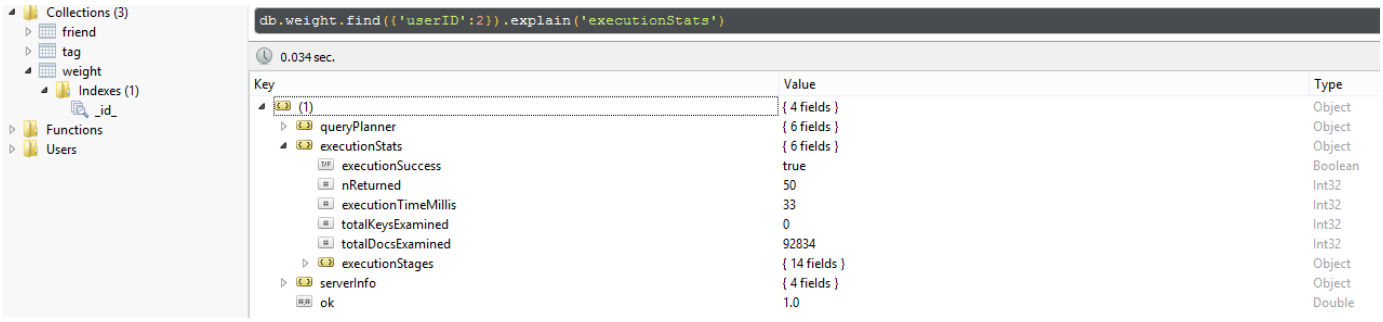

Figure 15: Complex Query 3 with Default Index in Collection Weight

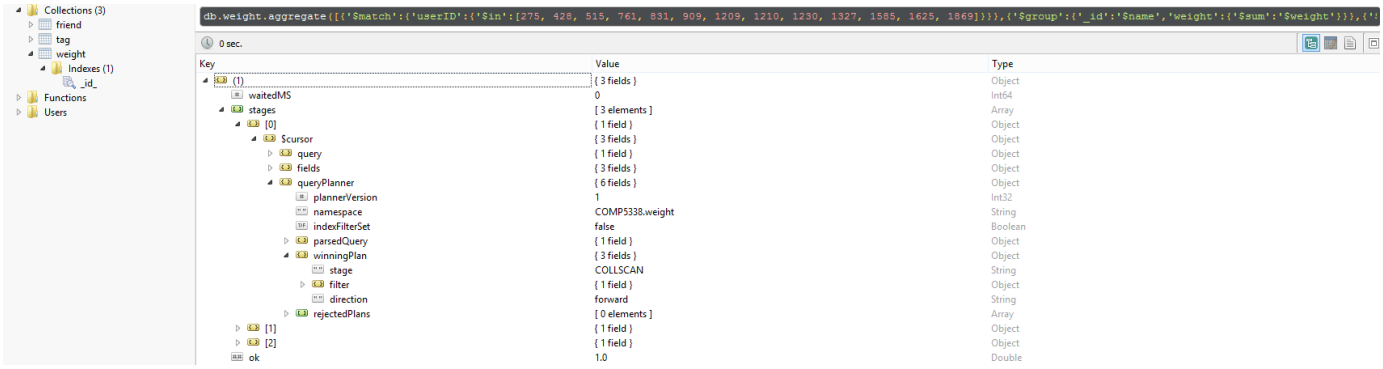

Figure 16: Complex Query 3 with Default Index in Aggregation

- Execution Statistics with Created Index
  Figures 17 and 18 show the execution statistics of the first two steps with the index

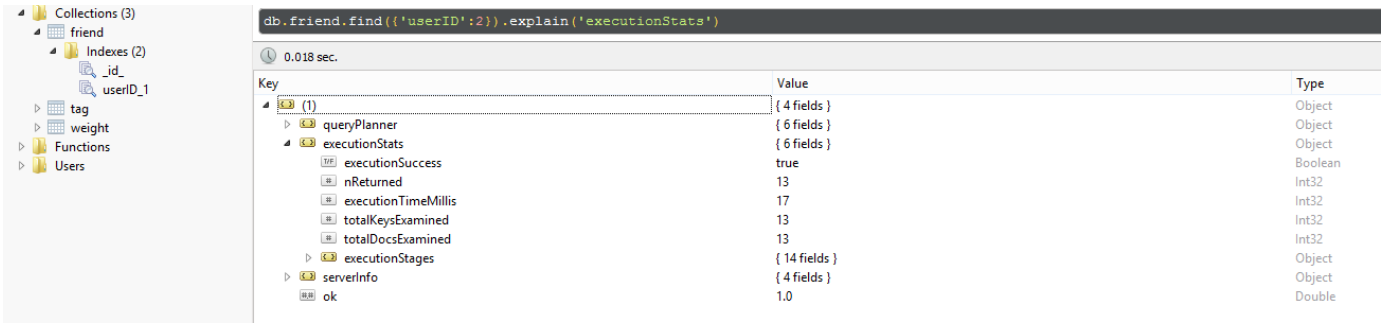

Figure 17: Complex Query 3 with Index “userID” in Collection Friend

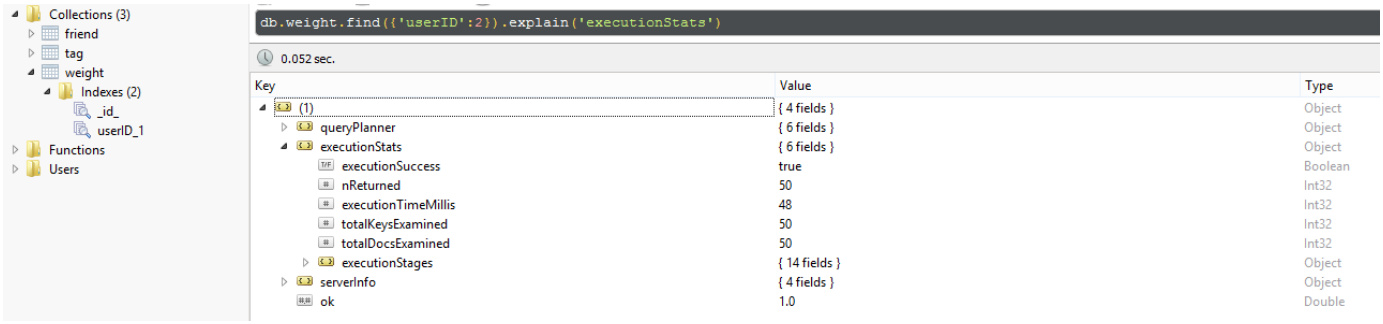

Figure 18: Complex Query 3 with Index "userID" in Collection Weight

"userID" created in both collections Friend and Weight. When querying the user's friends, 13 documents are examined and 13 results are returned, and the query time is 17; and when querying the artists listened by the user, 50 documents are examined and 50 results are returned, and the query time is 48. In both steps, while the number of documents examined is less, the query time increases.

Figure 19 shows that with the index "userID" created, the aggregation fetches data instead

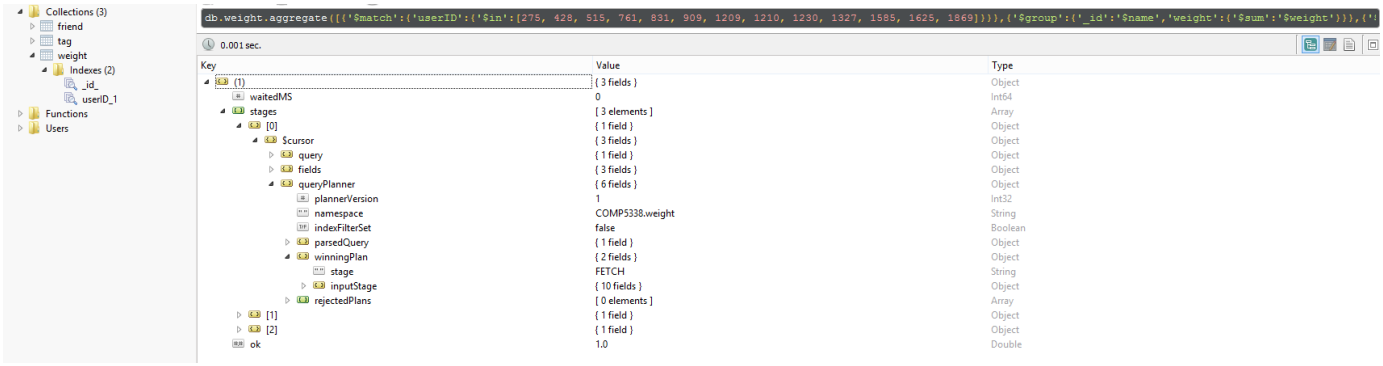

Figure 19: Complex Query 3 with Index "userID" in Aggregation

of a full collection scan.

4. Given an artist name, find the top 5 similar artists. Here similarity between a pair of artists is defined by the number of unique users that have listened both. The higher the number, the more similar the two artists are.

```
user_ID_matrix = []
artist_rank = {}
artist_name = input('   Input the artist name: ')
# find users who listen to this artist
for artistsname in weight.aggregate([{'$match':{'name':artist_name}},{'$group':
{'_id':'$name','user':{'$push':'$userID'}}}]):
    user_ID_matrix = artistsname['user']
k = 0
# find top 6 similar artists
for artistsname_all in
 ↪  weight.aggregate([{'$match':{'userID':{'$in':user_ID_matrix}}},{'$group':
{'_id':'$name','count':{'$sum':1}}},{'$sort':{'count':-1}},{'$limit':6}]):
```

```
    # exclude this artist
    if k != 0:
        print(artistsname_all)
    k += 1
```

```
Input the artist name: Diary of Dreams
{'_id': 'Combichrist', 'count': 6}
{'_id': 'Blutengel', 'count': 6}
{'_id': 'Depeche Mode', 'count': 6}
{'_id': 'The Birthday Massacre', 'count': 6}
{'_id': 'Assemblage 23', 'count': 5}
```

This complex query involves two aggregation pipelines in collection Weight. In the first aggregation, the artist name is matched and then the returned results are grouped by name together with the corresponding users who listen to this artist. In the second aggregation, all users returned in the last step are matched respectively and grouped by artist name and the corresponding total listening count. Then the artists with the same listening users as the given artist are sorted according to the number of users in a descending order, and 6 artists are returned including this given artist. Finally, top 5 artists are output excluding the given artist as similar artists.

- Execution Statistics with Default Index
  Figures 20 and 21 show the execution statistics of the two aggregations with no new index

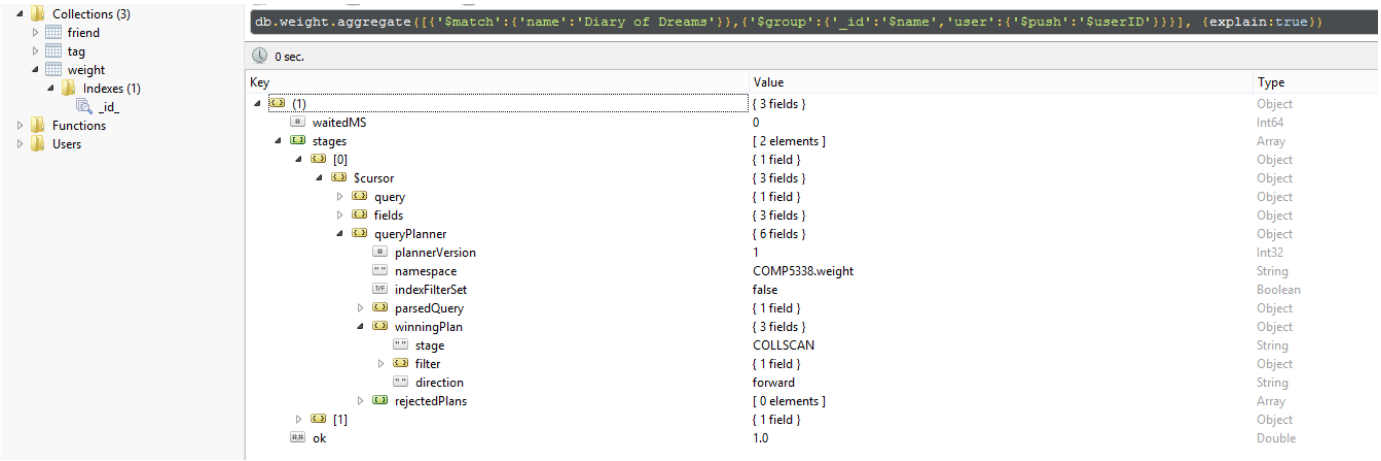

Figure 20: Complex Query 4 with Default Index in First Aggregation

created. These two aggregations both involve a full collection scan.

- Execution Statistics with Created Index

  Figure 22 shows the execution statistics of the first aggregation when "name" is created as an index in collection Weight. After this index is created, it fetches data rather than a full collection scan.
  Figure 23 shows the execution statistics of the second aggregation when "name" is created as an index in collection Weight. Since the second aggregation matches the field "userID", there is no effect on the query performance when the index "name" is created. This query still involves a full collection scan.
  Figure 24 shows the execution statistics of the second aggregation when "userID" is also

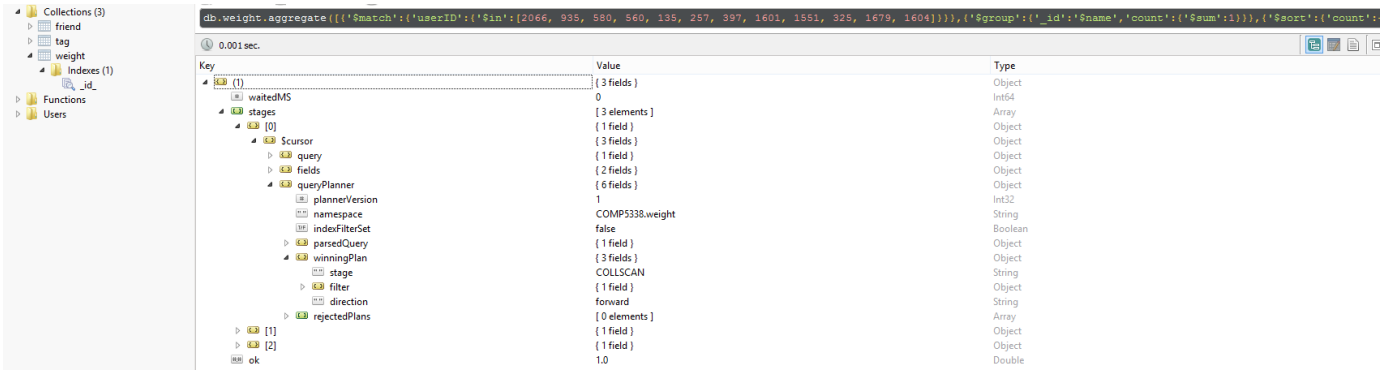

Figure 21: Complex Query 4 with Default Index in Second Aggregation

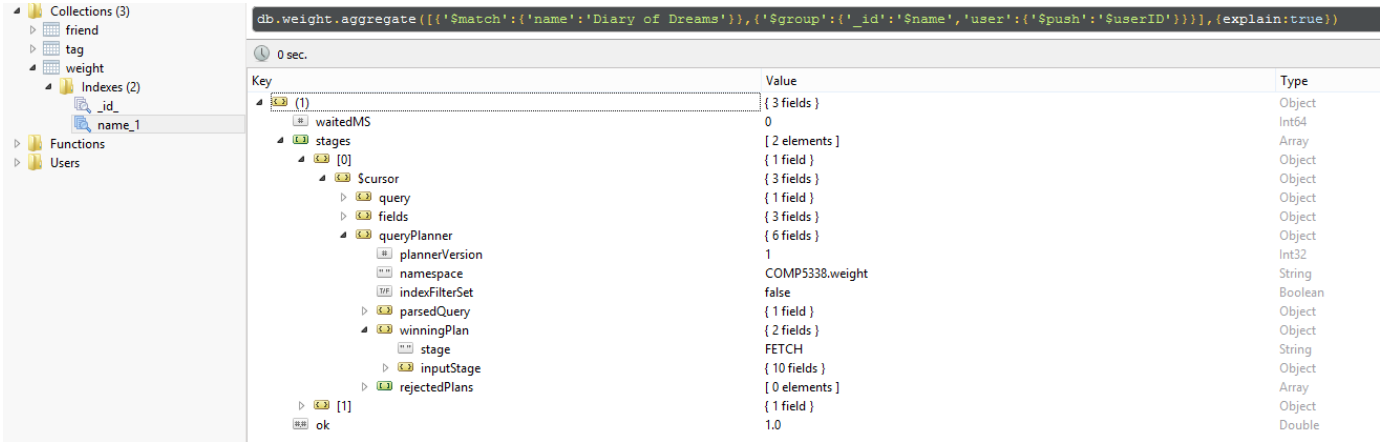

Figure 22: Complex Query 4 with Index “name” in First Aggregation

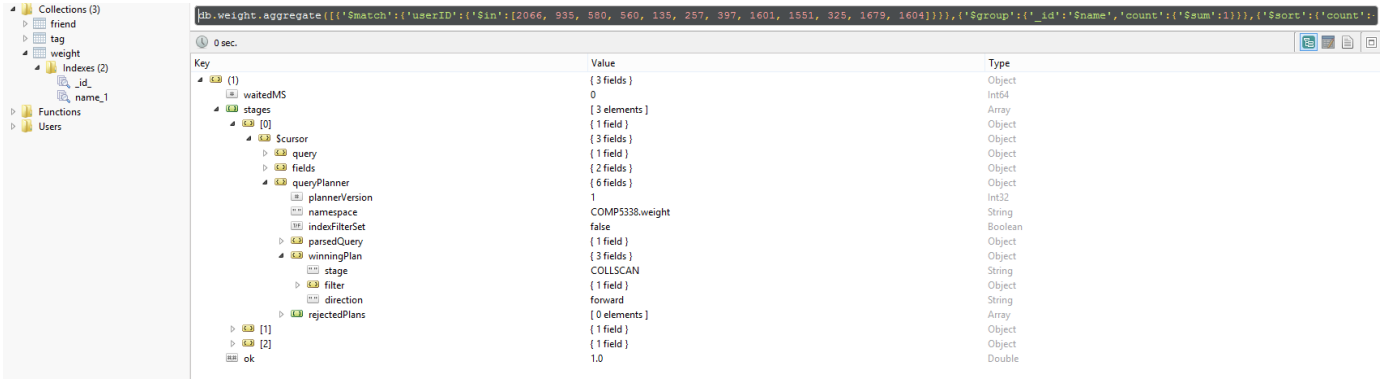

Figure 23: Complex Query 4 with Index “name” in Second Aggregation

created as an index in collection Weight, while the previously created index “name” is not removed. After the index “userID” is created, this aggregation does not involve a full collection scan but fetches data from the collection.

2.4 Schema Justification

Table 4 illustrates in which collections the queries are executed (SQ refers to simple query and CQ refers to complex query).

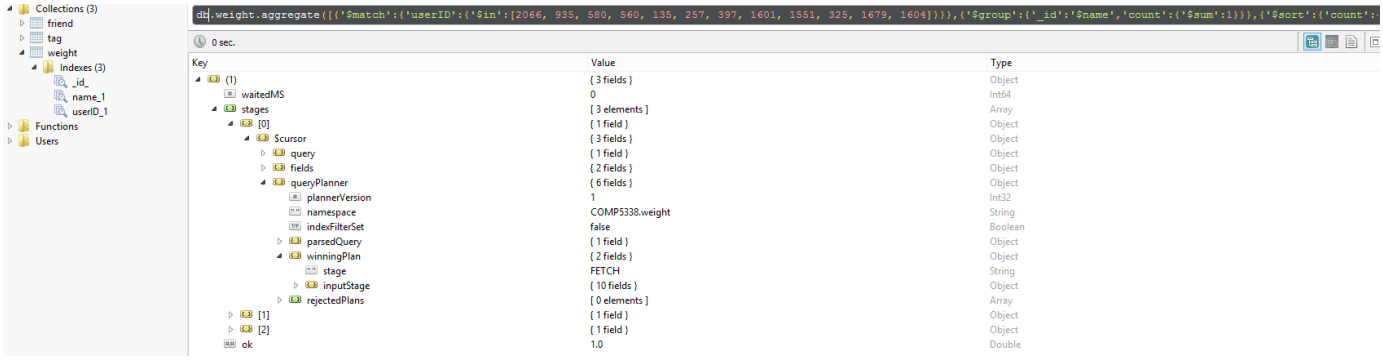

Figure 24: Complex Query 4 with Indexes "name" and "userID" in Second Aggregation

Collection	**Query**
Tag	SQ2, SQ4, CQ2
Weight	SQ1, SQ3, CQ1, CQ3, CQ4
Friend	SQ1, CQ3

Table 4: Query in Collection

Collection Friend is queried only when the query requires information about a user's friends; therefore, when a query does not involve information about friends, this collection does not need to be queried.

If a query relates to information about tagging or tags, collection Tag needs to be queried since it stores all information regarding tagging or tags, and if a query relates to information about artists listened by a user or listening count, collection Weight will be queried since all the information related to the listening behaviours of users is stored in this collection. Information about artists exists in both collections in order to return required results.

With such schema design, all the queries are satisfied by querying either a single collection or a combination of two collections.

3 Neo4j

Neo4j is a graph database which stores data as several nodes, and it also shows the relationship between the nodes. Neo4j has a number of advantages, one of which is that it uses a graph query language called "Cypher" specifically in order to make the queries become logical as well as easy to read and understand [3].

3.1 Schema

In Neo4j, the file "tag" and the file "user_taggedartists" are merged together into a file called "neo4j_tag.csv". Additionally, two nodes (User and Artist) and three relationships (Friend, Tag and Weight) are created in order to run the queries in high efficiency. The examples of each node and relationship are shown respectively as below:

- One example of node User is:

id
1

Table 5: User

- For node Artist, one example is shown as follows:

id	**name**	**url**	**pictureURL**
1	MALICE MIZER	http://www.last.fm	http://userserve-ak.fm

Table 6: Artist

- In order to connect users with their friends, relationship Friend is created and an example of which is shown below:

userID	**friendID**
1	275

Table 7: Friend

- tagID, combined with tagValue, is created in relationship Tag, with userID, artistID and timestamp included. An example of this relationship is:

userID	**artistID**	**tagID**	**timestamp**	**tagValue**
1679	9242	1	1.2674E+12	metal

Table 8: Tag

- The number of listening times of an artist by a user is shown in relationship Weight, an example of which is:

userID	artistID	weight
2	51	13883

Table 9: Weight

In addition, the .dat files are stored as .csv files in Neo4j.
An example of the schema which includes all nodes and relationships is shown as Figure 25:
In this sample, the initial node display is set to be 30, the max neighbours and max rows are set to be 10, and the max raw size is set to be 50. In this case, the implementation time of the schema can be reduced and the whole schema can be simplified.
The tags that assigned to the artist called "Diary of Dreams" are displayed in this schema, together with their belonging users. In addition, the relationships between friends and users, and the times that a user listens to an artist are both shown in this sample.

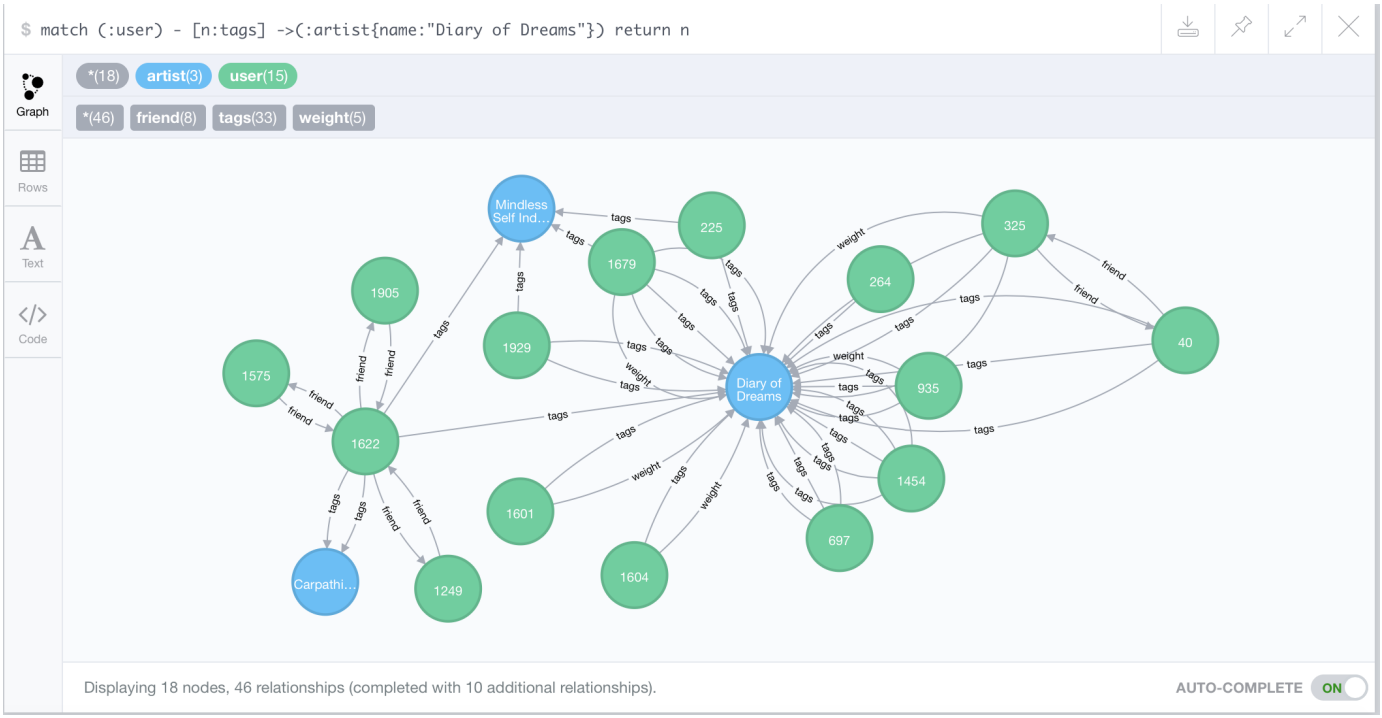

Figure 25: Sample Schema in Neo4j

3.2 Index

The following two indexes are created in Neo4j:

- artist
- user

The advantages of adding these indexes are stood out at the very beginning of the whole implementation. During the process of importing file Tag, when the indexes were not added, after more than 30 minutes were spent on importing data, the running was still not completed. However, after the indexes were added, the speed of loading data became obviously faster – it took only three minutes to import the whole file successfully.
As a result, all the solutions below are implemented based on these two indexes.

3.3 Solution

All the Neo4j query sentences are embedded in Python.
In addition, because the indexes "artist" and "user" are added initially, full records scan is not involved in any query.

3.3.1 Simple Query

1. Given a user id, find all artists the user's friends listen.

```
friend_artist_matrix = []
user_ID = str(input('Input your user ID: '))
# find user's friends
friend_raw = graph.data('match (:user{userID:{id}}) - [:friend]->(n) return
 ↪  (n.userID)',id=user_ID)
for user_ID_length in range(len(friend_raw)):
    # find artists listened by user's friends
    friend_artist = graph.data('match (:user{userID:{f_id}}) - [:weight] ->(n)
     ↪  return n.name, n.url, n.pictureURL',f_id =
     ↪  friend_raw[user_ID_length]['(n.userID)'] )
    print(friend_artist)
```

```
Input the userID: 2
{'pictureURL': 'http://userserve-ak.last.fm/serve/252/47390093.png', 'id': 89,
 ↪  '_id': ObjectId('57e0a41dbbaa62790cfcab86'), 'name': 'Lady Gaga', 'url':
 ↪  'http://www.last.fm/music/Lady+Gaga'}
{'pictureURL': 'http://userserve-ak.last.fm/serve/252/300983.jpg', 'id': 173,
 ↪  '_id': ObjectId('57e0a431bbaa62790cfcb63a'), 'name': 'Placebo', 'url':
 ↪  'http://www.last.fm/music/Placebo'}
{'pictureURL': 'http://userserve-ak.last.fm/serve/252/416514.jpg', 'id': 190,
 ↪  '_id': ObjectId('57e0a438bbaa62790cfcb9bb'), 'name': 'Muse', 'url':
 ↪  'http://www.last.fm/music/Muse'}
{'pictureURL': 'http://userserve-ak.last.fm/serve/252/363017.jpg', 'id': 198,
 ↪  '_id': ObjectId('57e0a43dbbaa62790cfcbc7e'), 'name': 'System of a Down',
 ↪  'url': 'http://www.last.fm/music/System+of+a+Down'}
```

   - Execution Statistics with Created Index
     Figure 26 gives information about how Neo4j estimates to find user2's friends. As the userID is given, the system only needs to estimate one row to find out the location of user2, then estimate 13 rows to figure out the friends of user2.

2. Given an artist name, find the most recent 10 tags that have been assigned to it.

```
artist_name = input('Input artist name: ')
print(graph.data('match (:user) - [n:tags] ->(:artist{name:{name}}) return
 ↪  n.tagValue,n.time order by n.time desc limit 10',name = artist_name ))
```

```
Input the artist name: Diary of Dreams
The most recent 10 recent 10 tags that have been assigned to it:
['german', 'darkwave', 'gothic', 'german', 'darkwave', 'gothic', 'gothic rock',
 ↪  'industrial', 'german', 'darkwave']
```

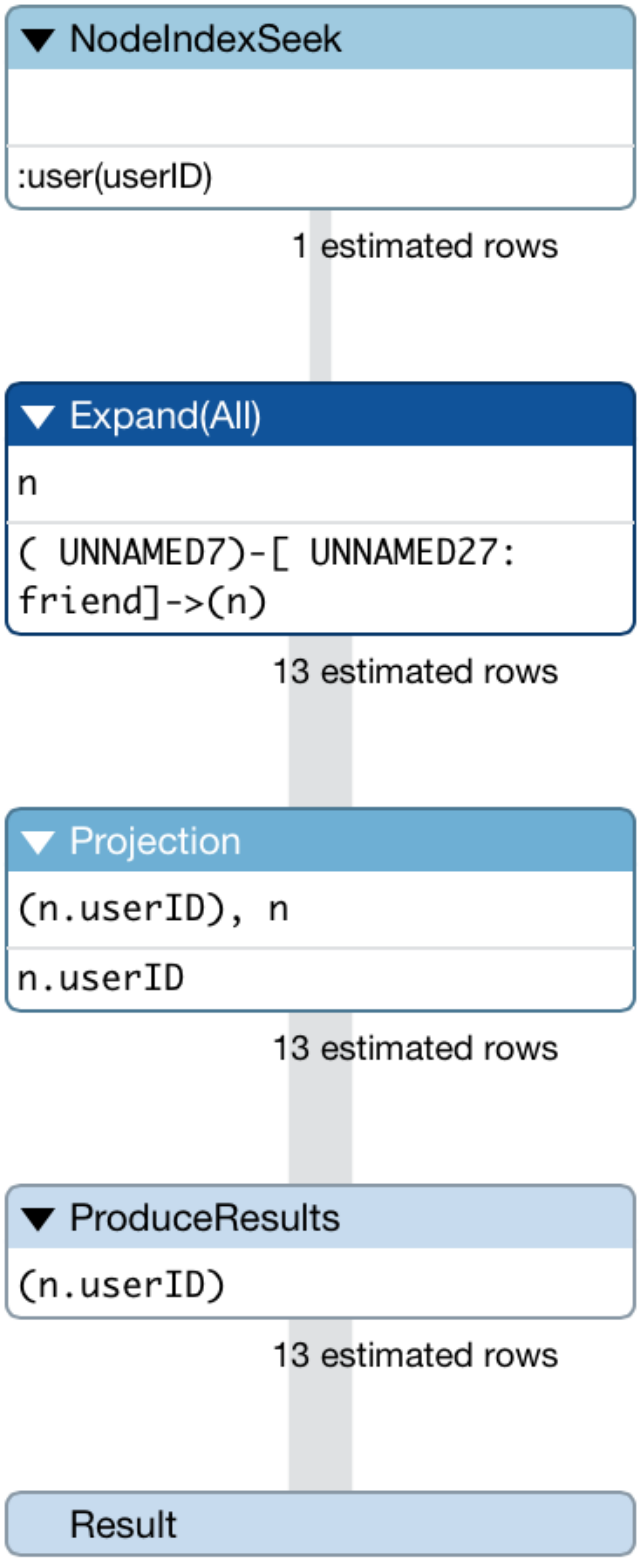

Figure 26: Execution Statistics of Simple Query 1

- Execution Statistics with Created Index
  Figure 27 shows the process of how Neo4j estimates to solve simple query 2.
  The system scans 17,632 rows after the artist name is given, followed by 1,763 rows esti-

mated after the step of filter. When the relationship Tag is used, 18,494 rows are estimated to find out which tags are assigned to the artist. The results of tags should be sorted in a descending order in terms of time so as to figure out the most recent 10 tags assigned to the artist.

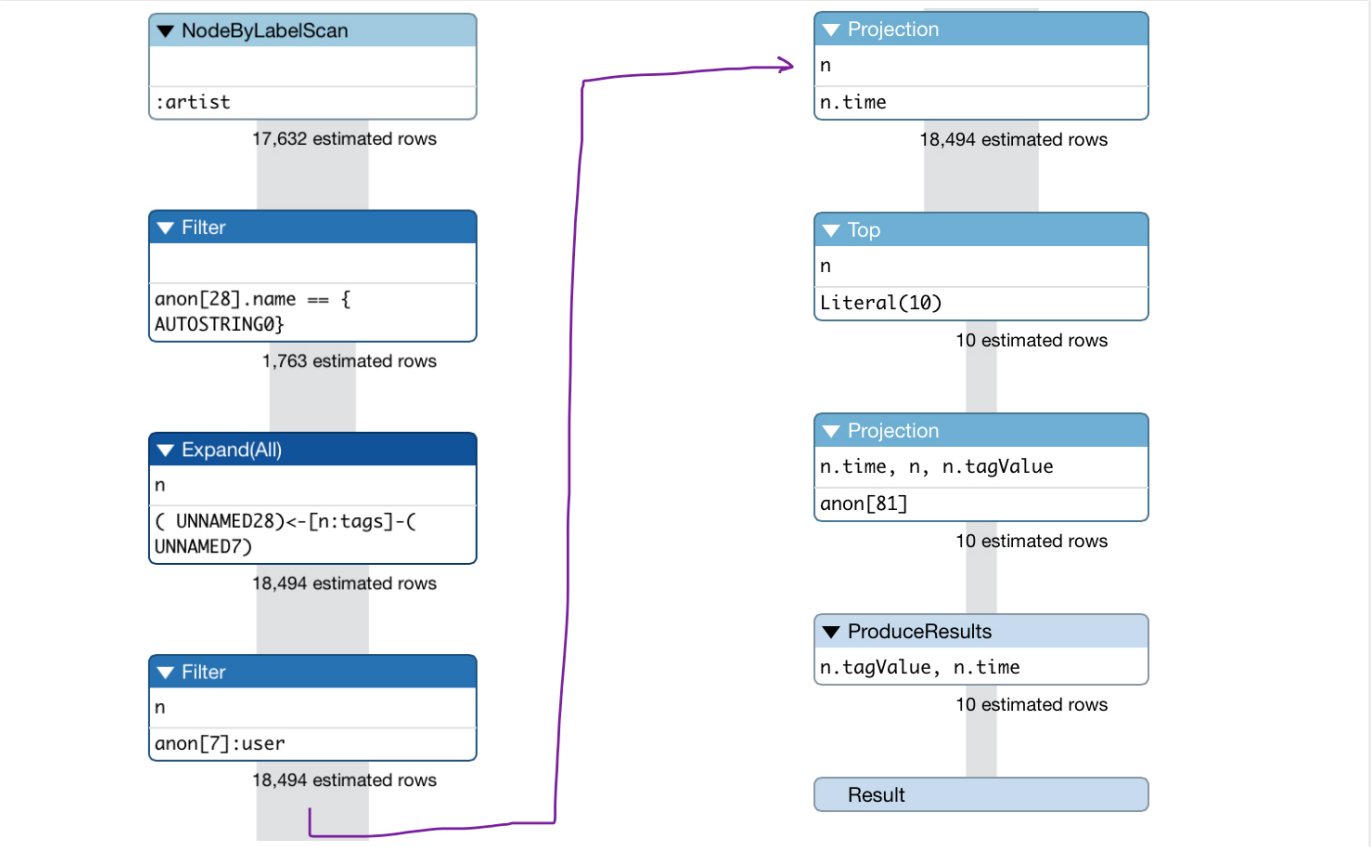

Figure 27: Execution Statistics of Simple Query 2

3. Given an artist name, find the top 10 users based on their respective listening counts of this artist. Display both the user id and the listening count.

```
weightnumber_dic = {}
artist_name = input('Input artist name: ')
# find listening counts of users
weightnumber = graph.data('match (u:user) - [n:weight] ->(:artist{name:{name}})
 ↪  return u.userID,n.weightnumber order by n.weightnumber desc
 ↪  ',name=artist_name)
for i in range(len(weightnumber)):
    weightnumber_dic['useID:%s'%weightnumber[i]['u.userID']]=
    int(weightnumber[i]['n.weightnumber'])
# print top 10 users
for i in range(10):
    print(sorted(weightnumber_dic, key=weightnumber_dic.__getitem__ ,reverse =
     ↪  True)[i],weightnumber_dic[sorted(weightnumber_dic,
     ↪  key=weightnumber_dic.__getitem__ ,reverse = True)[i]])
```

```
Input the artist name: Diary of Dreams
{'_id': ObjectId('57e0a404bbaa62790cfc9e47'), 'weight': 3466, 'userID': 325}
{'_id': ObjectId('57e0a403bbaa62790cfc9e44'), 'weight': 1021, 'userID': 135}
{'_id': ObjectId('57e0a404bbaa62790cfc9e4b'), 'weight': 868, 'userID': 1551}
{'_id': ObjectId('57e0a403bbaa62790cfc9e42'), 'weight': 803, 'userID': 580}
{'_id': ObjectId('57e0a404bbaa62790cfc9e49'), 'weight': 455, 'userID': 1604}
{'_id': ObjectId('57e0a403bbaa62790cfc9e41'), 'weight': 428, 'userID': 935}
{'_id': ObjectId('57e0a404bbaa62790cfc9e4a'), 'weight': 385, 'userID': 1601}
{'_id': ObjectId('57e0a404bbaa62790cfc9e48'), 'weight': 161, 'userID': 1679}
{'_id': ObjectId('57e0a403bbaa62790cfc9e45'), 'weight': 152, 'userID': 257}
{'_id': ObjectId('57e0a403bbaa62790cfc9e43'), 'weight': 134, 'userID': 560}
```

- Execution Statistics with Created Index
  As shown in Figure 28, the system estimates 17,632 rows to find out the given artist name. After given relationship Weight, 9,283 estimated rows are involved aiming at finding the users and the corresponding weights with regard to the artist.

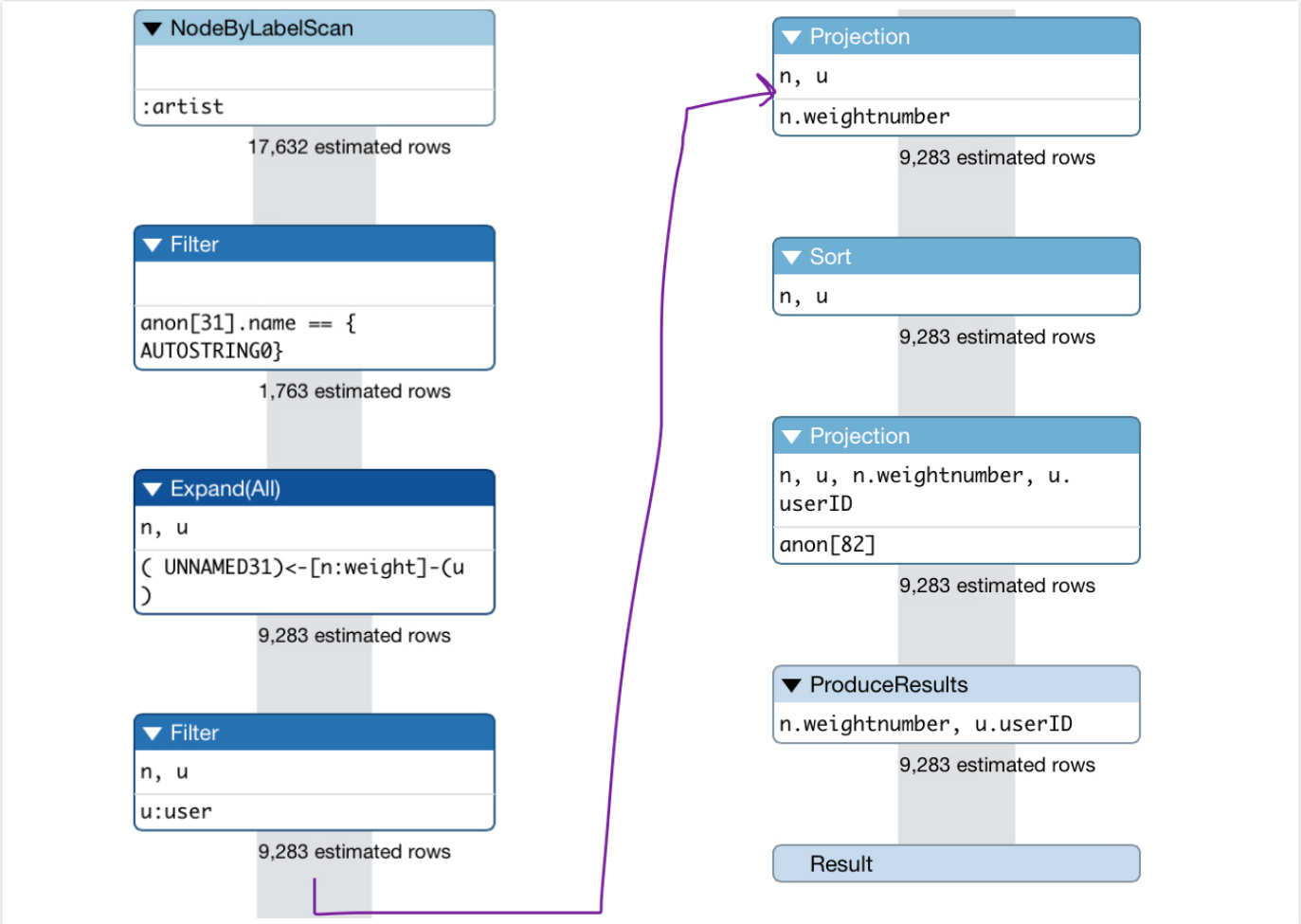

Figure 28: Execution Statistics of Simple Query 3

4. Given a user id, find the most recent 10 artists the user has assigned tag to.

```
user_ID = str(input('Input your user ID: '))
print(graph.data('match (u:user{userID:{id}}) - [n:tags] ->(a:artist) with n,a
 ↪  order by n.time desc return distinct a.name limit 10',id=user_ID))

Input the userID: 2
name:  China Crisis  time:  1241130000000
name:  Colin Newman  time:  1241130000000
name:  Loscil  time:  1241130000000
name:  Chicane  time:  1241130000000
name:  Sigue Sigue Sputnik  time:  1241130000000
name:  Morcheeba  time:  1238540000000
name:  Enigma  time:  1238540000000
name:  Caf Del Mar  time:  1238540000000
name:  Ministry of Sound  time:  1238540000000
name:  Fleetwood Mac  time:  1238540000000
```

- Execution Statistics with Created Index
  In Figure 29, the system estimates only one row as the user id and user index are given. Then, 98 rows are scanned to figure out the tags assigned by the user as well as their belonging artists. The sorting of the tags is based on 93 estimated rows so as to find out the most recent 10 artists the user has assigned tags to.

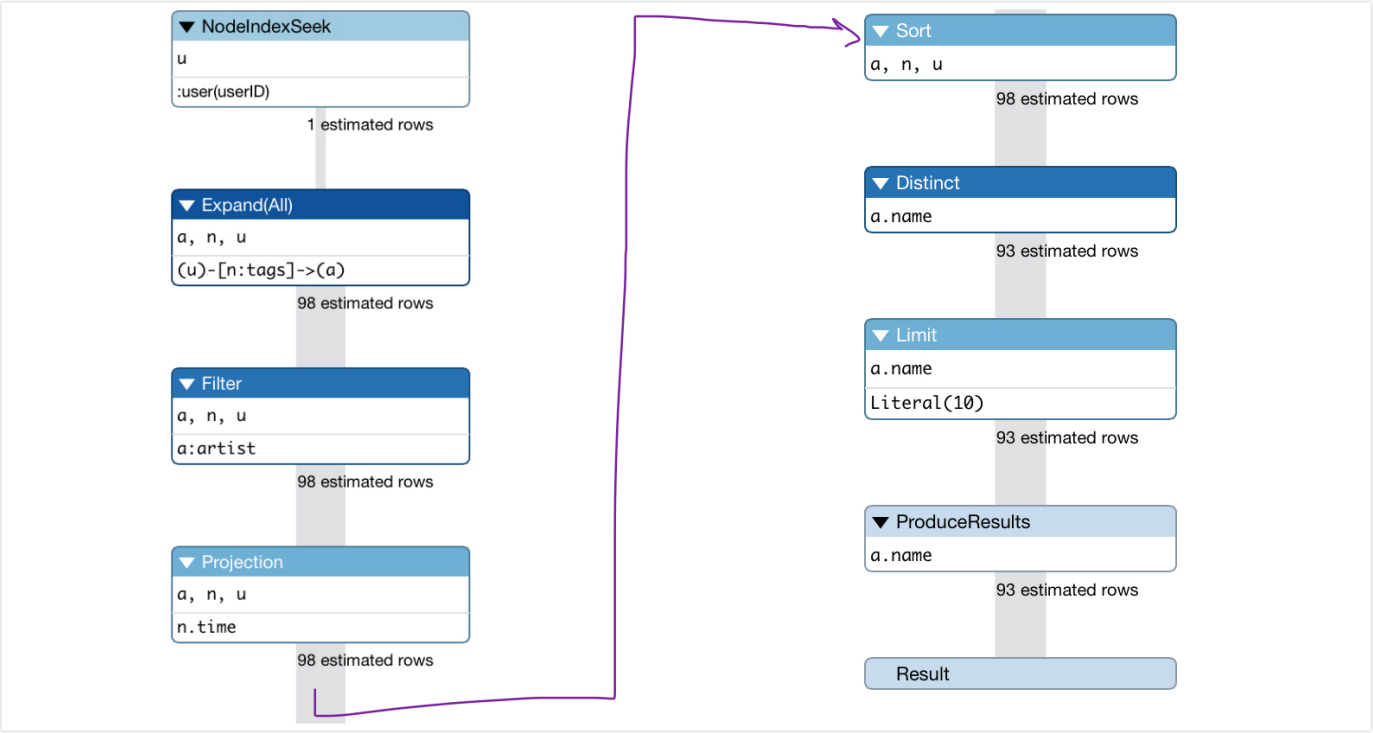

Figure 29: Execution Statistics of Simple Query 4

3.3.2 Complex Query

1. Find the top 5 artists ranked by the number of users listening to it.

```
print(graph.data('match (u:user) - [n:weight] ->(a:artist) return
↪  a.name,count(n) order by count(n) desc limit 5'))
```

```
{'count': 611, '_id': 'Lady Gaga'}
{'count': 522, '_id': 'Britney Spears'}
{'count': 484, '_id': 'Rihanna'}
{'count': 480, '_id': 'The Beatles'}
{'count': 473, '_id': 'Katy Perry'}
```

- Execution Statistics with Created Index
  Figure 30 shows that the system scans 1,892 rows in User before it scans 92,834 rows in Weight. After finding out the artists that each user listens, the system scans 305 rows to figure out the top 5 artists listened by users, then sorts the artists in terms of weight and returns 5 rows.

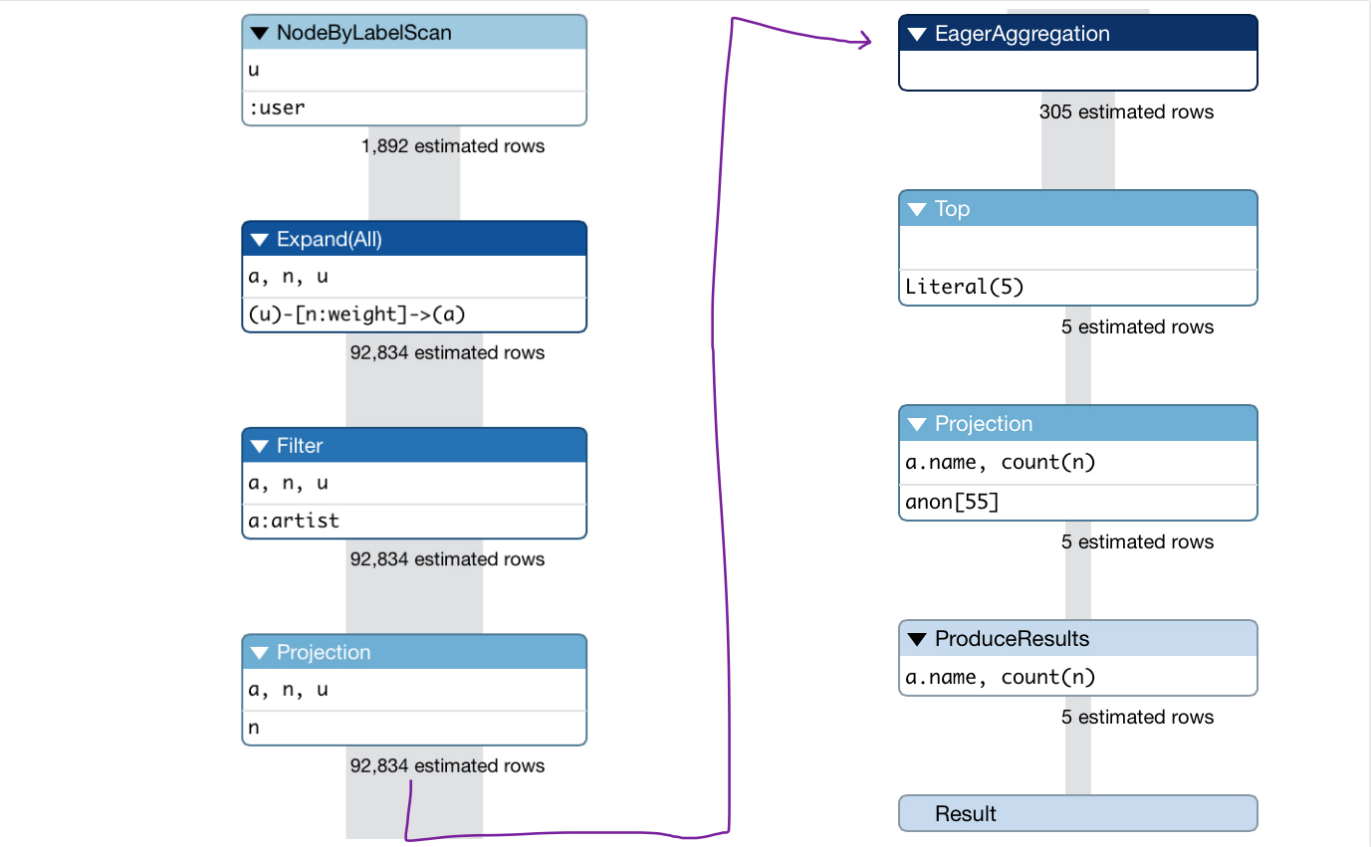

Figure 30: Execution Statistics of Complex Query 1

2. Given an artist name, find the top 20 tags assigned to it. The tags are ranked by the number of times it has been assigned to this artist.

```
artist_name = input('Input artist name: ')
print(graph.data('match (p:user) - [n:tags] ->(:artist{name:{name}}) return
↪  n.tagValue, count(n) order by count(n) desc  limit 20',name=artist_name))
```

```
Input the artist name: Diary of Dreams
```

```
{'_id': 'darkwave', 'count': 8}
{'_id': 'gothic', 'count': 5}
{'_id': 'german', 'count': 5}
{'_id': 'seen live', 'count': 2}
{'_id': 'gothic rock', 'count': 1}
{'_id': 'vocal', 'count': 1}
{'_id': 'electronic', 'count': 1}
{'_id': 'true goth emo', 'count': 1}
{'_id': 'industrial', 'count': 1}
{'_id': 'dark', 'count': 1}
{'_id': 'ambient', 'count': 1}
```

- Execution Statistics with Created Index
  In Figure 31, 17,632 rows are estimated to find out the given artist. Then, 18,494 rows are estimated to find out the tags that are assigned to the artist, based on which 18,494 estimated rows are involved to find out users who assign these tags. Due to the need to figure out the top 20 tags assigned to the artist, 136 rows are estimated. Finally, the tags are sorted to return 20 rows.

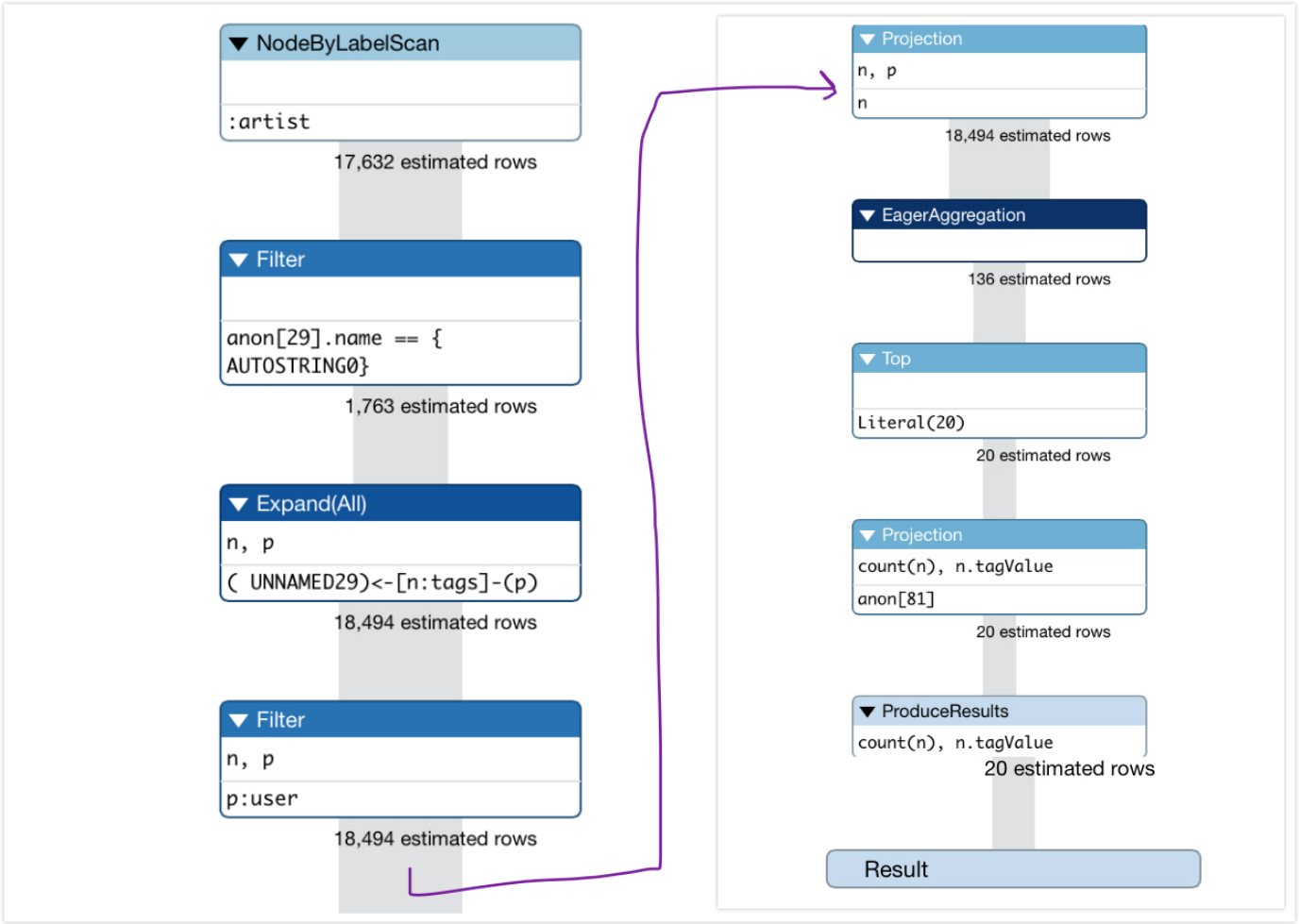

Figure 31: Execution Statistics of Complex Query 2

3. Given a user id, find the top 5 artists listened by his friends but not him. We rank artists by

the sum of friends' listening counts of the artist.

```
friend_matrix = []
self_artist_matrix =[]
user_ID = str(input('Input your user ID: '))
# find user's friends
friend_raw = graph.data('match (:user{userID:{id}}) - [:friend]->(n) return
↪  (n.userID)', id=user_ID)
for i in range(len(friend_raw)):
    friend_matrix.append(friend_raw[i]['(n.userID)'])
# find artists listened by user
self_artist = graph.data('match (:user{userID:{id}}) - [:weight]->(n) return
↪  (n.name)', id=user_ID)
for j in range(len(self_artist)):
    self_artist_matrix.append(self_artist[j]['(n.name)'])
# find artists listened by user's friends and listening counts
artist = graph.data('match (u:user) - [n:weight] ->(a:artist) where u.userID IN
↪  {matrix} return a.name,sum(toInt(n.weightnumber)) order by
↪  sum(toInt(n.weightnumber)) desc limit 10',matrix = friend_matrix)
count = 0
# exclude artists listened by user
for k in range(len(artist)):
    if artist[k]['a.name'] not in self_artist_matrix and count < 5:
        print(artist[k])
        count +=1
```

```
Input the userID: 2
{'weight': 39369, '_id': 'Panic! At the Disco'}
{'weight': 36956, '_id': 'Rammstein'}
{'weight': 26447, '_id': 'U2'}
{'weight': 20776, '_id': 'The Cure'}
{'weight': 20596, '_id': 'Pet Shop Boys'}
```

- Execution Statistics with Created Index
  Figure 32 shows how the system finds out friends of a given user. As the userID is provided, only one row is estimated. When it comes to relationship Friend, 13 rows are estimated.

4. Given an artist name, find the top 5 similar artists. Here similarity between a pair of artists is defined by the number of unique users that have listened both. The higher the number, the more similar the two artists are.

```
user_ID_matrix =[]
artist_name = input('Input artist name: ')
# find users who listen to this artist
user_ID = graph.data('match (p:user) - [n:weight] ->(:artist{name:{name}})
↪  return p.userID',name=artist_name)
for i in range(len(user_ID)):
    user_ID_matrix.append(user_ID[i]['p.userID'])
# find top 6 similar artists
```

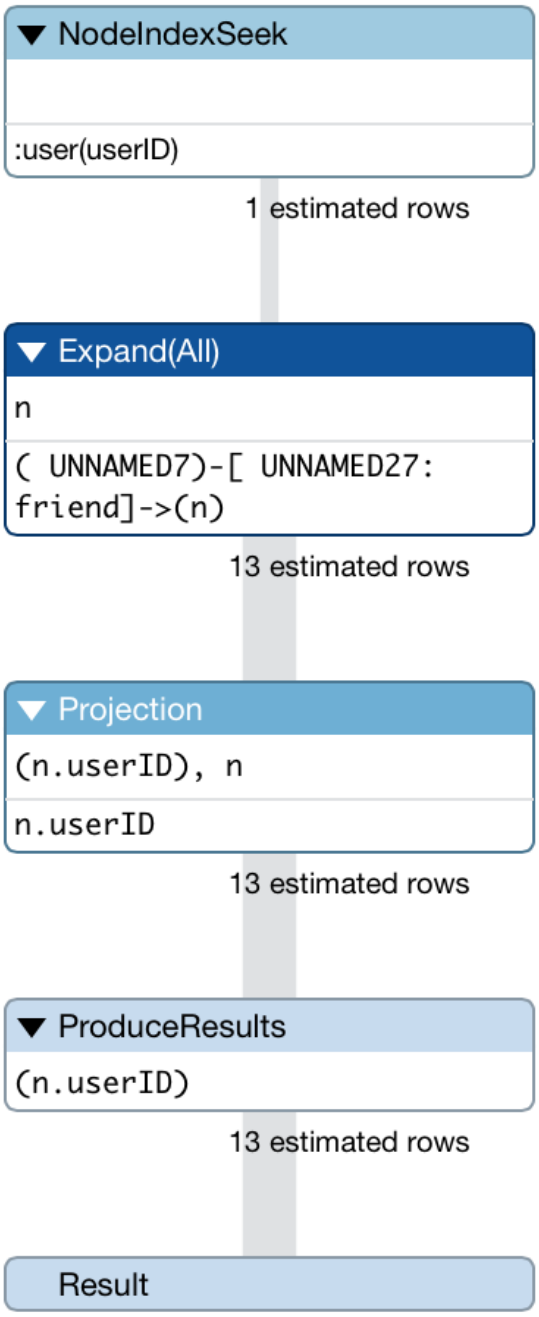

Figure 32: Execution Statistics of Step 1 of Complex Query 3

```
artist = graph.data('match (p:user) - [n:weight] ->(a:artist) where p.userID IN
 ↪  {matrix} return a.name, count(n) order by count(n) desc limit 6',matrix =
 ↪  user_ID_matrix)
# exclude this artist
for i in artist:
    if i['a.name'] != artist_name:
        print(i)

Input the artist name: Diary of Dreams
```

```
[{'count(n)': 6, 'a.name': 'Depeche Mode'}, {'count(n)': 6, 'a.name':
↪  'Combichrist'}, {'count(n)': 6, 'a.name': 'Blutengel'}, {'count(n)': 6,
↪  'a.name': 'The Birthday Massacre'}, {'count(n)': 5, 'a.name': 'Assemblage
↪  23'}]
```

- Execution Statistics with Created Index
  As it is shown in Figure 33, the location of a given artist is found after scanning 17,632 rows. Then, 9,283 rows are estimated in Weight and User in order to find out the artists who are similar to the given artist.

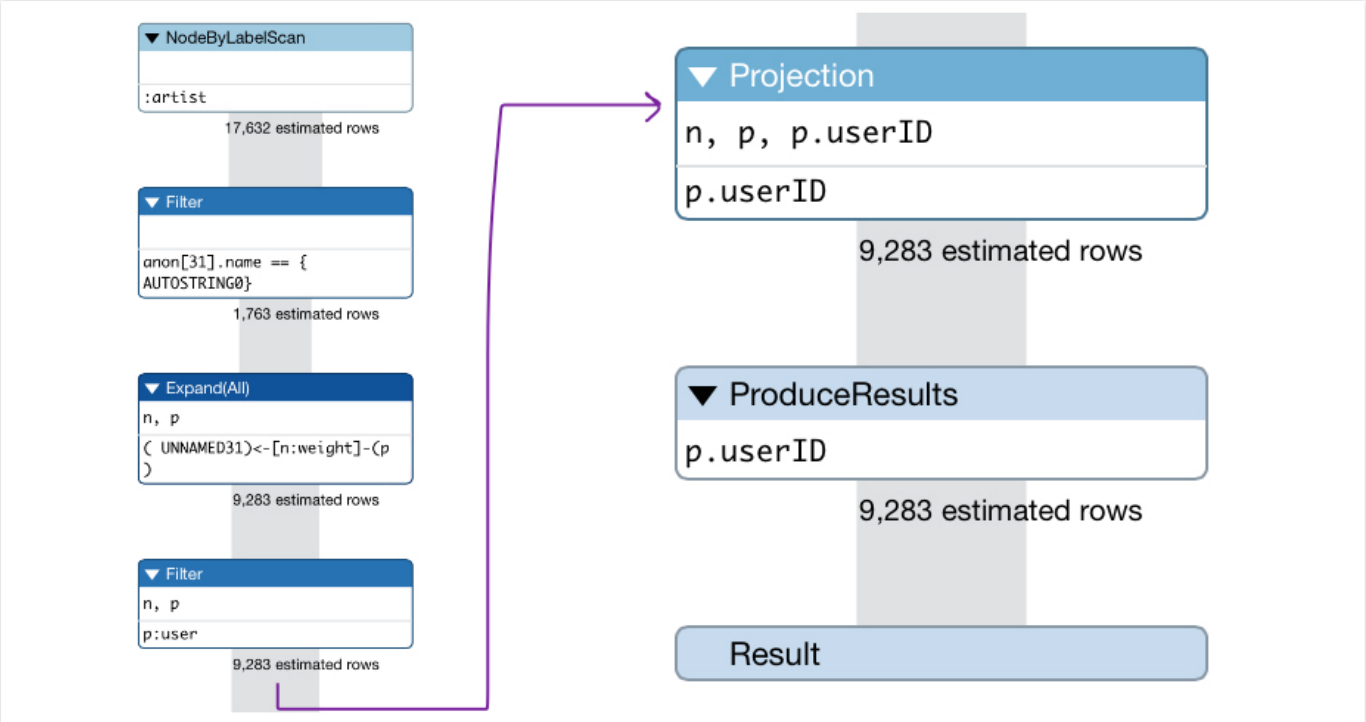

Figure 33: Execution Statistics of Complex Query 4

3.4 Schema Justification

From the queries above, it is obvious that the files "users" and "artists" are used in every query. As a result, it is reasonable that users and artists should be separated as two files. In addition, users and artists can be considered as two entities so that they are created as two nodes.
Meanwhile, the relationship between two users is Friend and the user can be related to an artist through the assignment of tags. Therefore, the relationship between users and artists is created.
In addition to Tag, it is necessary to create a relationship called "Weight" to show the listening count per user per artist in order to implement some target queries.

4 HBase

HBase is considered as an imitation of BigTable which is a system that can manage structured data and scale the data even it is in large size. HBase can organize data logically by forming it into tables, rows and columns [4].

4.1 Schema

There are four tables in HBase: Friend, Tag, Weight and User, and each table is designed to implement one query. Example rows of the four tables are shown as follows:

- The row key in table Friend is id and the column families are friend and artist. One row in Friend table below indicates the friends that user3 has and the artist that user3 listens. In this table, 6 rows of data (partial data of three users) are imported.

id	friend	artist
3	[78, 255]	[Pleq, Segue, Max Richter]

Table 10: Friend

- With regard to table Tag, the row key is artistname:timestamp and the column family is tag:userID - tagValue. One of the rows in Tag table shown below provides information about the time when artist "Justin Bieber" is tagged "canadian" by user403. In this table, 28 rows of data (partial data of two artists) are imported.

artistname:timestamp	tag:userID - tagValue
Justin Bieber:1291160000000	tag:403 - canadian

Table 11: Tag

- In table Weight, the row key is name:weight_userID and the column family is user:userID - weight. The row below is given to show the number of times that user330 listens to artist "Justin Bieber". It should be noted that since the row key values in HBase are sorted according to the first digit then second, third... digit, "0" needs to be complemented to the first digit(s) of the value of weight in order to make all the weights have the same number of digits so as to sort the weight values according to magnitude. This design will be further discussed in Section 5.3. In this table, 22 rows of data (partial data of two artists) are imported.

name:weight_userID	user:userID - weight
Justin Bieber:0665_330	**user:330 - 665**

Table 12: Weight

- In table User, the row key is id:timestamp and the column family is artist:name - tagValue. One row in User table is displayed below, which shows information about the time when user3 assigns a tag "glitch" to artist "Pleq". In this table, 21 rows of data (partial data of two users) are imported.

id:timestamp	artist:name - tagValue
3:1246400000000	artist:Pleq - glitch

Table 13: User

4.2 Solution

4.2.1 Simple Query

1. Given a user id, find all artists the user's friends listen.

```
hbase(main)> get 'xguo8788:friend','3','friend'
hbase(main)> get 'xguo8788:friend','78','artist'
hbase(main)> get 'xguo8788:friend','255','artist'
```

```
hbase(main):043:0> get 'xguo8788:friend','3','friend'
COLUMN                CELL
 friend:              timestamp=1475232040085, value=[78,255]
1 row(s) in 0.0040 seconds

hbase(main):044:0> get 'xguo8788:friend','78','artist'
COLUMN                CELL
 artist:              timestamp=1475232264151, value=[Pleq,Radiohead,Infected Mus
                      hroom]
1 row(s) in 0.0030 seconds

hbase(main):045:0> get 'xguo8788:friend','255','artist'
COLUMN                CELL
 artist:              timestamp=1475232315162, value=[Portishead,Pleq,Deru]
1 row(s) in 0.0040 seconds
```

Figure 34: Simple Query 1

In Figure 34, table Friend is queried. The first query returns the friends of a given user 3; then the artists listened by the user's friends are queried respectively as written by the second and third query, and the artists listened by the user's friends are returned.

- Query Performance
  The first query only scans 2 rows with the row key "3" and the column family "friend"; the second query only scans 2 rows with the row key "78" and the column family "artist"; and the third query only scans 2 rows with the row key "255" and the column family "artist". None of these three scans involves a full scan. The query time is also displayed in Figure 34.

2. Given an artist name, find the most recent 10 tags that have been assigned to it.

```
hbase(main)> scan 'xguo8788:tag',{REVERSED=>true, STARTROW=>'Justin
↪  C',ENDROW=>'Justin A',LIMIT=>10}
```

In Figure 35, table Tag is queried. The most recent 10 tags assigned to a given artist "Justin Bieber" need to be returned. This query involves a reverse scan to find the latest tags assigned.

Figure 35: Simple Query 2

The startrow and endrow are also specified in order to include the artist name within the query range. By limiting the returned results as 10, 10 queried results are returned.

- Query Performance
  This query only scans 13 rows with the row key values between "Justin C" and "Justin A" and does not need to scan other rows, and therefore no full scan is required. The query time is also displayed in Figure 35.

3. Given an artist name, find the top 10 users based on their respective listening counts of this artist. Display both the user id and the listening count.

```
hbase(main)> scan 'xguo8788:weight',{REVERSED=>true, STARTROW=>'Justin
↪  C',ENDROW=>'Justin A',LIMIT=>10}
```

In Figure 36, table Weight is queried. The top 10 users who listen to the given artist "Justin Bieber" need to be returned. This query involves a reverse scan to sort the listening counts of users in a descending order. The startrow and endrow are also specified in order to include the artist name within the query range. By limiting the returned results as 10, 10 queried results are returned. This query will be further discussed in Section 5.3.

- Query Performance
  This query only scans 11 rows with the row key values between "Justin C" and "Justin A"

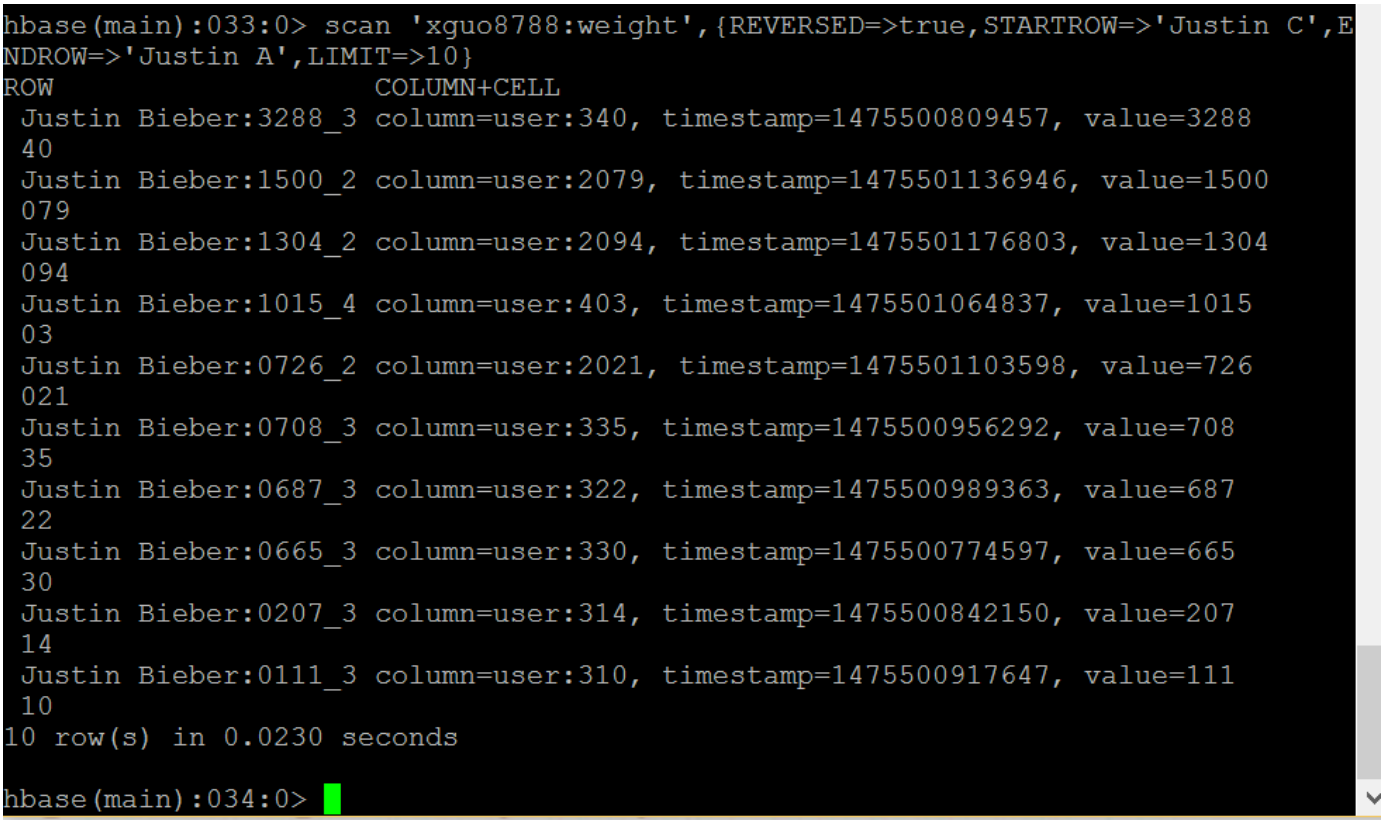

Figure 36: Simple Query 3

and does not need to scan other rows, and therefore no full scan is required. The query time is also displayed in Figure 36.

4. Given a user id, find the most recent 10 artists the user has assigned tag to.

```
hbase(main)> scan
↪ 'xguo8788:user',{REVERSED=>true,STARTROW=>'4',ENDROW=>'3',LIMIT=>10}
```

In Figure 37, table User is queried. The most recent 10 artists a given user 3 has assigned tag to need to be returned. This query involves a reverse scan to find the latest results. The startrow and endrow are also specified in order to include the user within the query range. By limiting the returned results as 10, 10 queried results are returned.

- Query Performance
  This query only scans 12 rows with the row key values between "4" and "3" and does not need to scan other rows, and therefore no full scan is required. The query time is also displayed in Figure 37.

4.3 Schema Justification

Four tables, Friend, Tag, Weight and User are designed to execute required queries respectively. Since HBase query is implemented based on row keys, the row key of each table is specifically designed in order to fulfill the queries.
For the first query which queries table Friend, the row key is designed as the userID and two column families are designed to extract data stored in the corresponding column family with regard to each

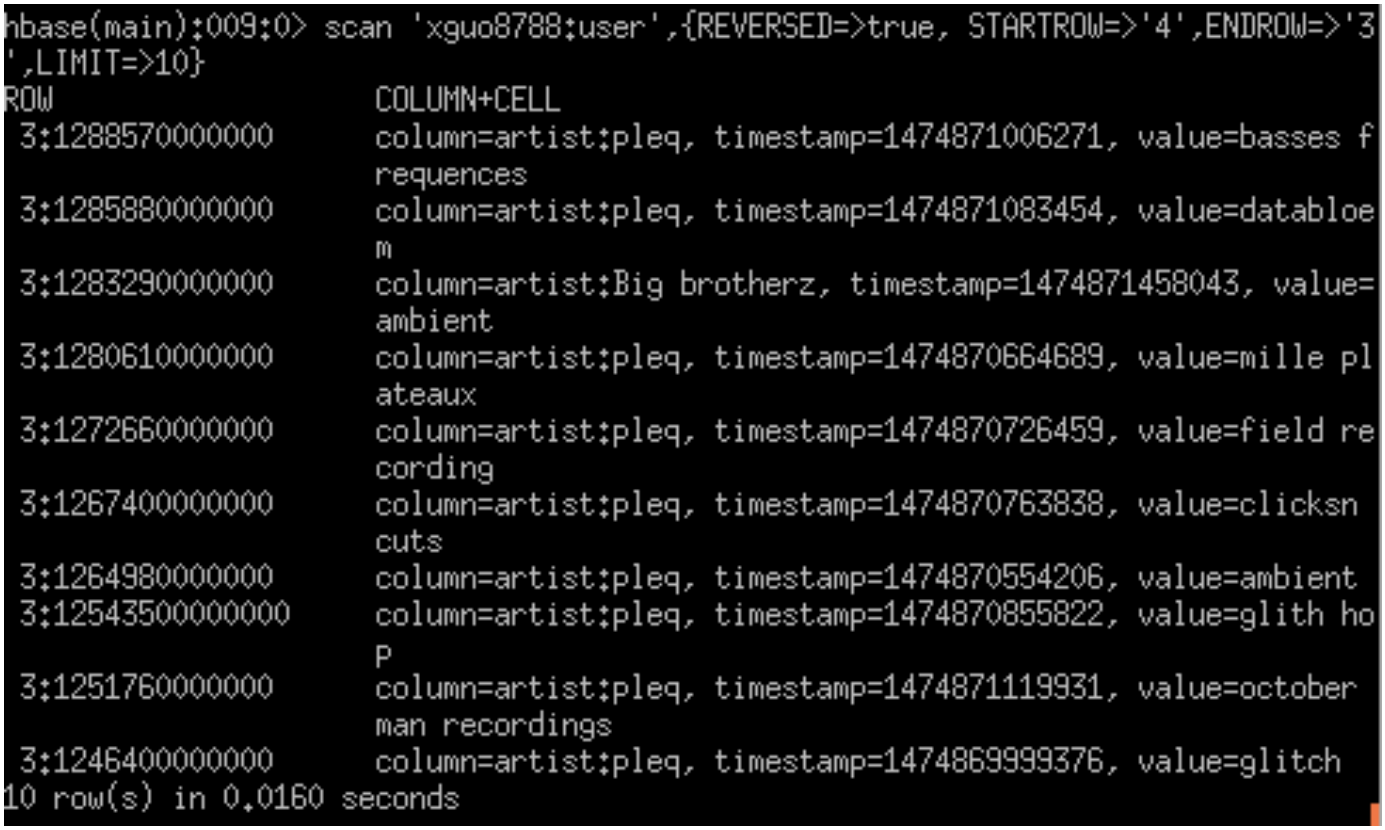

Figure 37: Simple Query 4

query sentence.

For the second query which queries table Tag, since this query involves both the scanning of artist name and the sorting according to time, and row key values are stored after sorting by HBase, the row key is designed to include both the artist name and timestamp in order to query corresponding rows without the need of a full table scan and then return required results of tags.

For the third query which queries table Weight, both the query for rows of a given artist and sorting results are involved; therefore, the row key is designed to include the artist name and a user's listening count for query and sorting.

For the fourth query which queries table User, it still involves scanning and sorting. Hence, the schema design takes the advantage of sorted row key values stored in HBase to encompass both the userID and timestamp in the row key for scanning corresponding rows in this table to return query results.

5 System Comparison

5.1 Ease of Use

1. Complexity for Booting System
   The first step before executing queries is to boot the NoSQL system on the computer.

   - MongoDB
     The database engine is started by entering several commands in a command window or powershell window first. Then, Robomongo is started as client shell GUI. Finally, another command window or powershell needs to be opened to run commands in order to import files. These steps are quite tricky for people who are not familiar with MongoDB as they need to enter a number of commands to start up the MongoDB server as well as set files to the server before they begin to write queries. However, MongoDB supports not only Windows but also other operating systems like OSX, Linux and Solaris, which enables flexibility and portability for users during MongoDB implementation.
   - Neo4j
     The booting steps for Neo4j are the easiest among the three systems. Users only need to download Neo4j from the website and open it by clicking the software directly instead of opening another window and running several commands. In addition, it is convenient to run Neo4j as it is suitable for operating systems like Windows, OSX and Linux although it still cannot support Solaris like MongoDB nor Unix like HBase.
   - HBase
     A browser window should be opened and pointed to the HBase WebUI. HBase can only support Windows, Linux and Unix operating systems, which means that it might be less portable and flexible than MongoDB.

2. Time for Importing Files

   - MongoDB
     Although a few commands in another command window or powershell window are needed when importing files to MongoDB, the time needed to import files mainly depends on the time that users spend on coding commands. In other words, users can start querying the database as soon as they input commands in the command window or powershell window.
   - Neo4j
     Compared to MongoDB, the time for importing files in Neo4j is slightly longer, especially regarding files in large size. As a result, indexes are always created at the beginning of importing large-size files.
   - HBase
     HBase might be the one that costs the most time among these three NoSQL systems as users need to input all data in files manually, which may take a plenty of time when encountered files in large size.

3. Implementation Interface

   - MongoDB
     MongoDB provides users with a specific query user interface, which is concise and user-friendly. In this interface, a few collections are displayed on the left. Users can input query

sentences and the query results are displayed below the queries. Users can also view all the data in one collection after clicking on the corresponding collection item on the left.

- Neo4j
  As a representative of graph databases, Neo4j turns data into nodes and relationships, and then displays them in graphs, which makes the implementation interface become visualized. Taking Figure 25 as an example, when a query to find users who have assigned tags to the artist "Diary of Dreams" is input, this artist will be displayed as a node together with nodes of users who have tagged this artist. In addition, the relationships among nodes are also shown so that users can gain a better understanding of the query result within a glance. Moreover, when clicking on one of the nodes, for example, user1622, the other nodes related to user1622 can also be displayed on the same interface. In this case, users can obtain a further understanding of other nodes without the need of entering another query. As a result, Neo4j is considered as the most user-friendly system among these three NoSQL systems.
- HBase
  HBase does not provide a certain interface for users as all queries in HBase are implemented through several commands.

5.2 Query Performance

1. Running Time
   The running time of these three NoSQL systems is compared with each other through their performance on simple queries. The least running time for each simple query in each NoSQL system is shown in Table 14. However, it should be noted that all data is imported in MongoDB and Neo4j, while only a small volume of data is imported in HBase.

Query	MongoDB	Neo4j	HBase
Simple Query 1	56ms	40ms	11ms
Simple Query 2	93ms	66ms	21ms
Simple Query 3	30ms	78ms	23ms
Simple Query 4	72ms	74ms	16ms

Table 14: Running Time

2. Query Mechanism

   - MongoDB
     MongoDB scans all the documents according to id in the case of no new index being created. If a new index is created, it will scan documents according to this index as well. In MongoDB, the number of documents examined with an appropriate index created is less than that with only the default index, while the query time is not always shorter than that without an appropriately created index.
   - Neo4j
     Since indexes are created at the beginning, Neo4j starts with one or a set of nodes by using these indexes. After matching a given path, a series of transactions like row filtering, skipping, sorting, projection, etc. are executed in order to return the query result.

- HBase
  HBase can only scan a table according to the row key and then extract data based on column family keys and column keys, and both the row key values and column key values are stored after being sorted. Therefore, it is important to design an appropriate row key in order to execute target queries.

5.3 Problem Discussion

- HBase
  During the implementation of Simple Query 3 in HBase, the results initially returned were not correct, as shown in Figure 38.
  Figure 38 illustrates that the values of weight returned are not sorted according to magnitude, which means that some values which should not be ranked as top 10 were also returned. After analysing the reasons resulting in such problem, it is found that the way HBase sorts row key values is based on the magnitude of each digit, that is, the first digits are sorted according to the magnitude, then second, then third..., regardless of the total digits of a number.
  In order to solve this problem, the number of digits of each value should be unified, for example, all the values have 4 digits respectively. For this purpose, complementing digits of "0" were added to one or more digits before the actual value for unifying the number of digits of each value without changing its magnitude. Thus, HBase can return the correct results by sorting all the values from the highest digit to the lowest digit.

```
hbase(main):018:0> scan 'xguo8788:weight',{REVERSED=>true,ENDROW=>'Christina Af'
,STARTROW=>'Christina Ah',LIMIT=>10}
ROW                    COLUMN+CELL
 Christina Aguilera:7 column=user:302, timestamp=1474875637412, value=7945
 945_302
 Christina Aguilera:7 column=user:310, timestamp=1474875842179, value=7099
 099_310
 Christina Aguilera:7 column=user:335, timestamp=1474876061098, value=708
 08_335
 Christina Aguilera:6 column=user:322, timestamp=1474876080047, value=687
 87_322
 Christina Aguilera:6 column=user:330, timestamp=1474875983636, value=665
 65_330
 Christina Aguilera:5 column=user:2031, timestamp=1474875892968, value=5779
 779_2031
 Christina Aguilera:5 column=user:327, timestamp=1474875779187, value=569
 69_327
 Christina Aguilera:4 column=user:2059, timestamp=1474875815566, value=47630
 7630_2059
 Christina Aguilera:4 column=user:348, timestamp=1474876096871, value=42
 2_348
 Christina Aguilera:4 column=user:2037, timestamp=1474875755559, value=411
 11_2037
10 row(s) in 0.0330 seconds
```

Figure 38: Initial Results of Simple Query 3 in HBase

5.4 Schema Difference

- MongoDB
  MongoDB is a row-based data store and stores data in JSON-like documents. The schema in MongoDB is free – it does not require the keys of a document to be fixed or defined in advance and data can be stored as an array or an object array, which means that one key can correspond to a number of values. In the schema, MongoDB can support various databases and each database can have many collections. There are a variety of documents under one collection, each of which can have different keys [5].
  In this assignment, the schema is made up by three collections: Tag, Weight and Friend, and documents in the collections are stored in the format of BSON.

- Neo4j
  Basically, the schema of Neo4j consists of different kinds of nodes and relationships between nodes. Users can attach one label and various properties to a node. There are several types of relationships and each relationship can have a set of properties [6].
  There are only two nodes in Neo4j in this assignment: User and Artist, while other files are considered as relationships between user and artist. In this case, all items in files can be shown in one graph, which provides users with a better understanding of the schema within one glance in an intuitive way.

- HBase
  HBase is a column-based distributed data store and data in HBase is classified into one or more tables so the schema of HBase is mainly made up by a number of tables which contain rows specified by the row key and column families. The orientation of the storage format in HBase is column, which allows users to store numerous details in the same table and this table can further be split into several tables as the volume of data increases [7]. Since HBase scans a table by rows, it is of great importance to design an appropriate row key to facilitate subsequent queries. Four tables (Friend, Tag, Weight and User) are created in HBase in this assignment and the row keys are specifically designed in order to execute target queries.

6 Reference

[1] Cattell, R. (2011). Scalable SQL and NoSQL data stores. *Acm Sigmod Record*, *39*(4), 12-27.
[2] Boicea, A., Radulescu, F., & Agapin, L. I. (2012, September). MongoDB vs Oracle-Database Comparison. In *EIDWT* (pp. 330-335).
[3] Jaiswal, G., & Agrawal, A. P. (2013). Comparative analysis of Relational and Graph databases. *IOSR Journal of Engineering (IOSRJEN).*
[4] Vora, M. N. (2011, December). Hadoop-HBase for large-scale data. In *Computer science and network technology (ICCSNT), 2011 international conference on* (Vol. 1, pp. 601-605). IEEE.
[5] Khan, S., & Mane, V. (2013). SQL support over MongoDB using metadata. *International Journal of Scientific and Research Publications*, *3*(10), 1-5.
[6] Robinson, I., Webber, J., & Eifrem, E. (2015). *Graph Databases: New Opportunities for Connected Data.* "O'Reilly Media, Inc.".
[7] George, L. (2011). Client API: Administrative Features. *HBase: the definitive guide* (pp. 207-233). "O'Reilly Media, Inc.".