Chapter 1

Graph database

The DBSM chosen for the graph database is Neo4j, it was use to manege the social part of the site and also to keep track of the reviews made for the feed and suggestion functionality.

The entities used in the database are:

- User
- TopCritic
- Movie

The nodes themselves do not store a lot of data, we decided to keep the bare minimum by having the following attributes:

- for the User and TopCritic:
  - id
  - name
- for Movie:
  - id
  - title

The relationship present are:

- User -[:FOLLOWS]-> TopCritic
- User -[:REVIEWED]-> Movie
- TopCritic -[:REVIEWED]-> Movie

In particular the *REVIEWED* relationship contains the following information:

- content
- date
- freshness

Below is present a snapshot of the graph database taken from Neo4j.

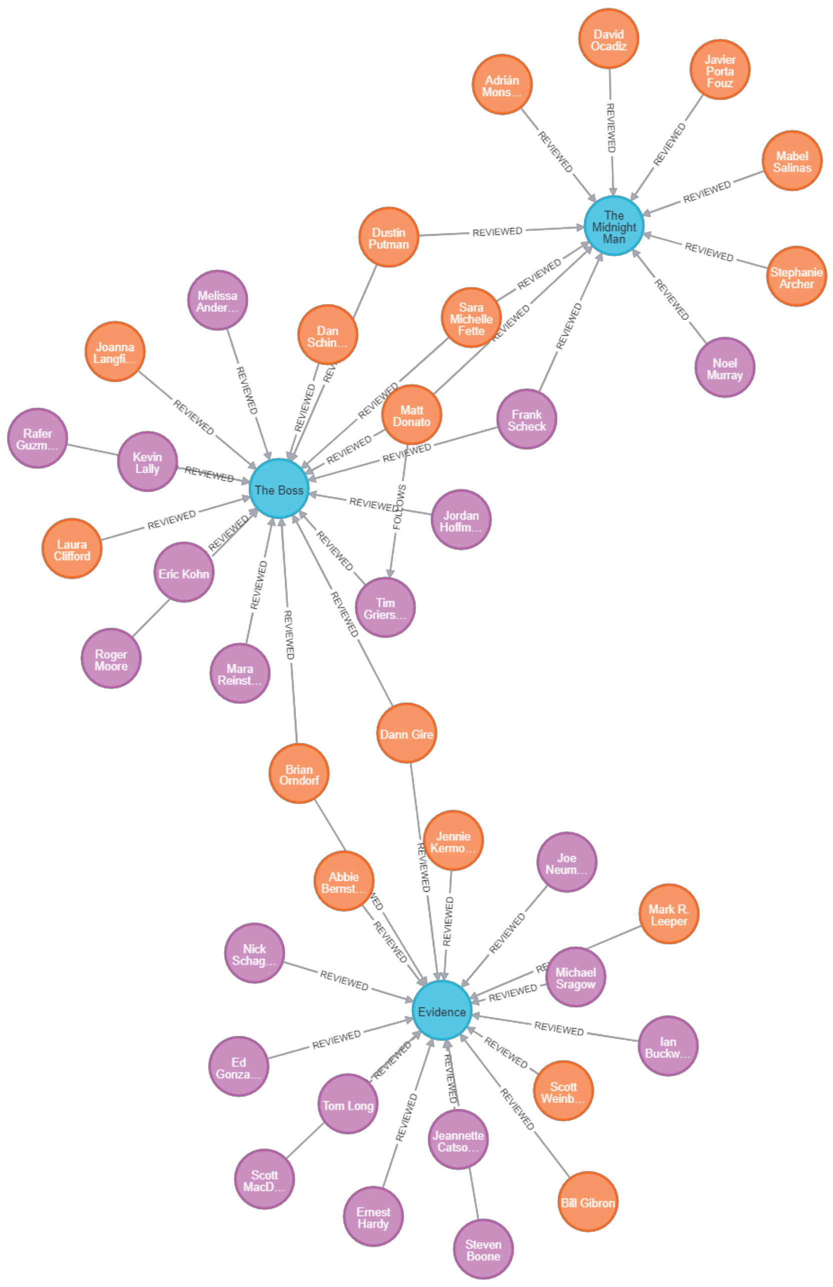


Figure 1.1: graph DB

As previously stated in the Feasibility study section 2 , the graph database design was heavily influenced by the document one, it was in fact generated by python script (Appendix A .6) that starts from the movie collection and, after generating the nodes for the *Movie* entity, does the same for the user by dividing them between normal user and top critic. It then generates the *REVIEWED* relationship based upon the data stored in the document database and finally it generates the *FOLLOWS* relationship randomly

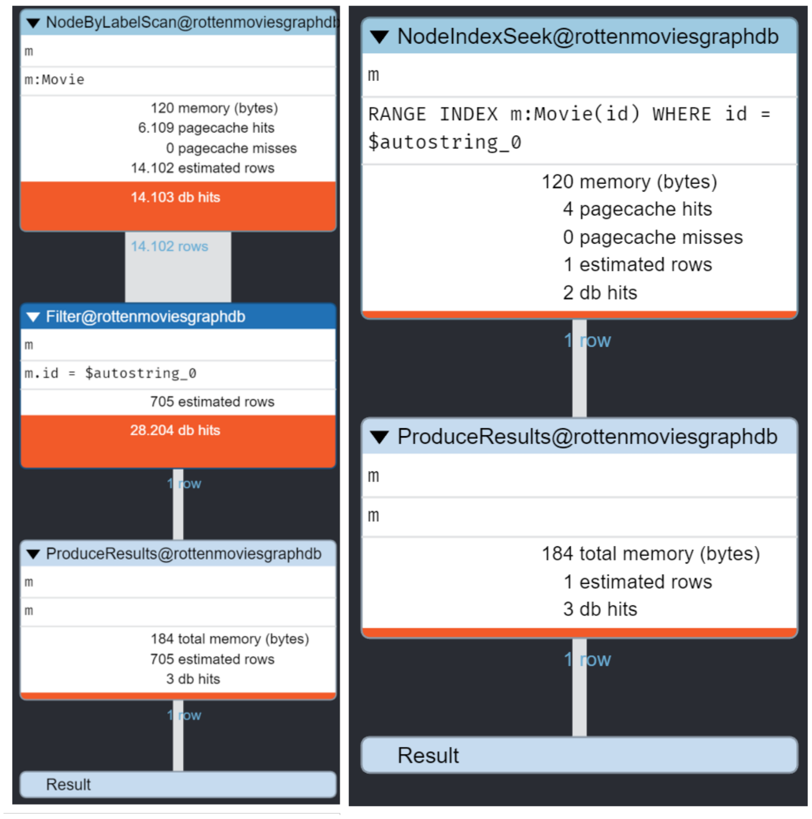
1 Index

The *id* attributes is common to all nodes and it stores a string which is also used as id by the document database in the _*id* field, this choice was made so that we could have a way to identify the same object across the different databases with only one string from the application. Due to the OOP language used to build the latter the data was manipulated with classes containing the id of the object so, to improved the performance of the graph database we decided to use the *id* attribute as a index.

Below the information in a json format from the Neo4j DBSM about the new index (some attributes were omitted for space)

```
[
  {
    "id": 3,
    "name": "movie_id_index",
    "type": "RANGE",
    "entityType": "NODE",
    "labelsOrTypes": [
      "Movie"
    ],
    "properties": [
      "id"
    ]
  },
  {
    "id": 5,
    "name": "topcritic_id_index",
    "type": "RANGE",
    "entityType": "NODE",
    "labelsOrTypes": [
      "TopCritic"
    ],
    "properties": [
      "id"
    ]
  },
  {
    "id": 4,
    "name": "user_id_index",
    "type": "RANGE",
    "entityType": "NODE",
    "labelsOrTypes": [
      "User"
    ],
    "properties": [
      "id"
    ]
  }
]
```

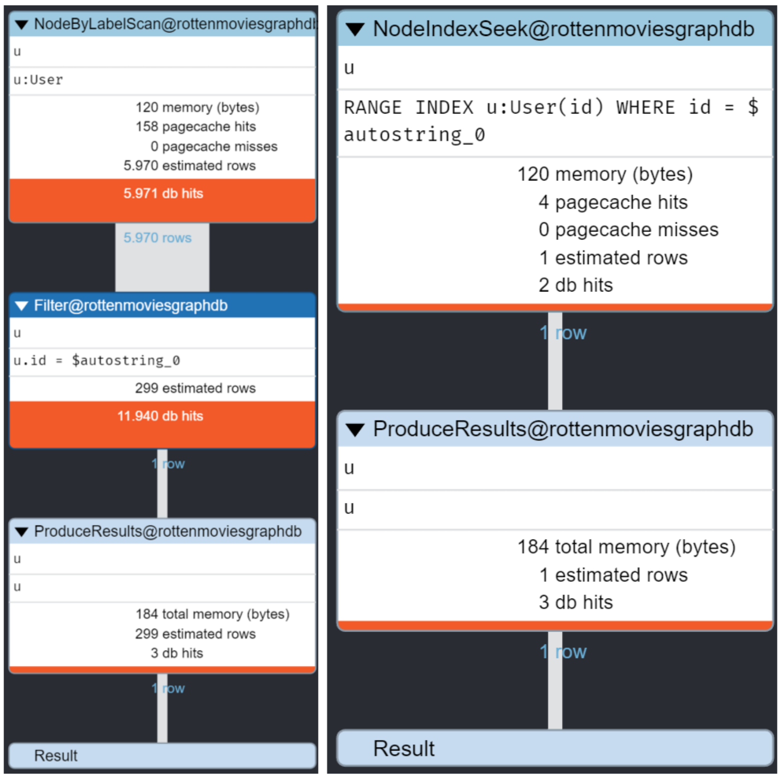
We then proceed to confront the profile of the queries with and without the above indexes, here we can see the result of this analysis:



```
▼ NodeByLabelScan@rottenmoviesgraphdb
m
m:Movie
120 memory (bytes)
6.109 pagecache hits
0 pagecache misses
14.102 estimated rows
14.103 db hits
14.102 rows
▼ Filter@rottenmoviesgraphdb
m
m.id = $autostring_0
705 estimated rows
28.204 db hits
1 row
▼ ProduceResults@rottenmoviesgraphdb
m
m
184 total memory (bytes)
705 estimated rows
3 db hits
1 row
Result
```

```
▼ NodeIndexSeek@rottenmoviesgraphdb
m
RANGE INDEX m:Movie(id) WHERE id =
$autostring_0
120 memory (bytes)
4 pagecache hits
0 pagecache misses
1 estimated rows
2 db hits
1 row
▼ ProduceResults@rottenmoviesgraphdb
m
m
184 total memory (bytes)
1 estimated rows
3 db hits
1 row
Result
```

Figure 1.2: search by id without and with movie_id _index



```
▼ NodeByLabelScan@rottenmoviesgraphdb
u
u:User
120 memory (bytes)
158 pagecache hits
0 pagecache misses
5.970 estimated rows
5.971 db hits
5.970 rows
▼ Filter@rottenmoviesgraphdb
u
u.id = $autostring_0
299 estimated rows
11.940 db hits
1 row
▼ ProduceResults@rottenmoviesgraphdb
u
u
184 total memory (bytes)
299 estimated rows
3 db hits
1 row
Result
```

```
▼ NodeIndexSeek@rottenmoviesgraphdb
u
RANGE INDEX u:User(id) WHERE id = $
autostring_0
120 memory (bytes)
4 pagecache hits
0 pagecache misses
1 estimated rows
2 db hits
1 row
▼ ProduceResults@rottenmoviesgraphdb
u
u
184 total memory (bytes)
1 estimated rows
3 db hits
1 row
Result
```

Figure 1.3: search by id without and with user_id _index

2 Queries

In the following section we will show the queries that drove the design of the graph database.

Graph-Centric Query	Domain-Specific Query
Which are the User nodes with the most outgoing edges of type REVIEWED	Which are the non top critic user with most reviews made
Which are the TopCritic nodes with the most incoming edges of type FOLLOWS	Which are the most followed top critics
Which are the Movies nodes that have been most recently connected with TopCritic nodes by a REVIEWED relationship meanwhile the latter are connected to a User node with a FOLLOWS relationship	Which are the latest reviewed movies by a followed top critics given a starting user
Which are the User and TopCritic nodes that have outgoing REVIEWED edges to the same Movie node without having a FOLLOWS edge between them, which of these TopCritic nodes have more similar attribute on the REVIEWED relationship with the one on the same type going towards the same Movie node given a User node	Which are the non-followed top critics with the most affinity towards a given user
Which Movie nodes have different ratio of freshness in the incoming REVIEWED edges spitted in base of their date attribute	Which movies have been targeted by review bombing in the last x month