Scott Pearson

ECE:5845 Modern Databases

Movie Recommendations Final Project

Part 1: Upload data into database

1. Primary Database Choice: Neo4j

For my movie recommendation system, I decided to use Neo4j as the primary database. This is because the data is inherently graph-shaped: users rate movies, movies share genres, and we eventually wish to compute similarities & recommendations based on connections between users and items. A property-graph model along with Neo4j's Cypher query language makes it easy to express and display these relationships as well as running graph algorithms, like similarity or community detection, over the rating network.

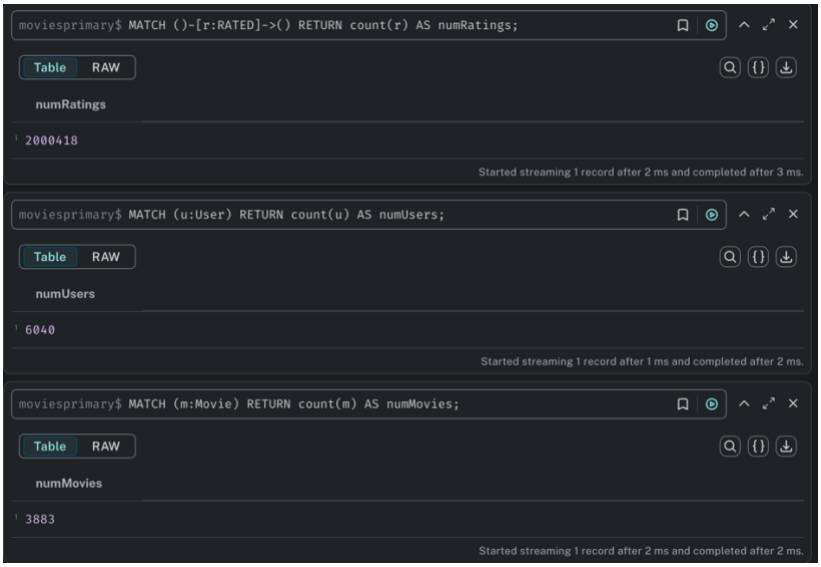
2. Importing tables (had to change the memory in my config file for ratings to be properly imported):

```
LOAD CSV WITH HEADERS FROM 'file:///movies.csv' AS row
WITH row,
     // Extract 4-digit year inside parentheses, e.g. "1995"
     apoc.text.regexGroups(row.title, '.*\\((\\d{4})\\)') AS yearMatch
MERGE (m:Movie {movieId: toInteger(row.movieId)})
SET m.title  = row.title,
    m.genres = CASE
                 WHEN row.genres IS NULL OR row.genres = '(no genres listed)'
                 THEN []
                 ELSE split(row.genres, '|')
               END,
    m.year = CASE
                WHEN yearMatch IS NULL OR size(yearMatch) = 0
                THEN NULL
                ELSE toInteger(yearMatch[0][1])
             END;

Created 3,883 nodes, set 15,532 properties, added 3,883 labels
Completed after 536 ms
```

- This was my import for the movies.csv
- I also used the APOC plugin as well

- This was me testing that the import worked properly

- This was my import for the ratings-1.csv (I had to change my config file to get everything imported fine)

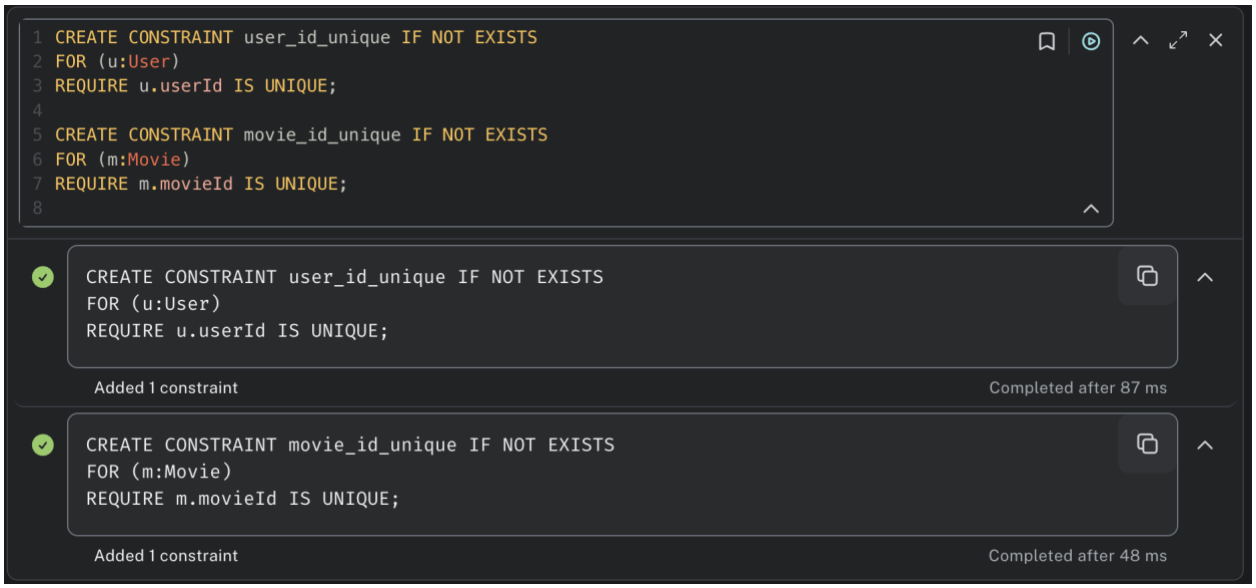


- This was me testing again to make sure that my import worked as intended

- I added in the movie and rating csv files into Neo4j's import directory. Then, I performed batched imports to avoid transaction memory limits.

3. Constraints for system:

I added the following constraints (before importing my data) to avoid duplicates and to speed up imports:

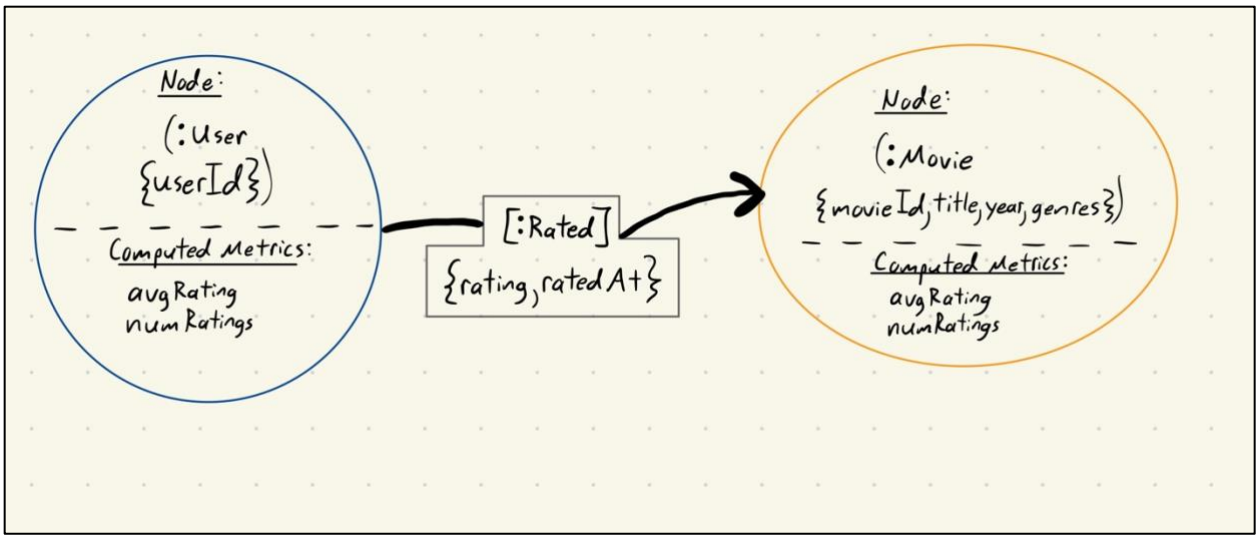


Part 2: Data design pattern and recommendation strategy

1.

- Data Model Design that supports movie recommendation queries:

To support the movie recommendation queries, I adopted a property-graph data model in Neo4j. In my model, users and movies are represented as nodes and ratings are represented as relationships. Each movie is modeled after a (:Movie {movieId, title, year, genres}) node. Each user is represented as a (:User {userId}) node. A user's interaction with a movie is stored through a [:RATED {rating, ratedAt}] relationship from a User node to a Movie node.



- Example figure of my Data Model Design

This structure can naturally fit collaborative filtering since it makes user-movie interactions explicit as edges. Similarity patterns begin to emerge as shared neighbors within the graph. To better support preference learning, I extended the nodes with aggregate metrics. For each user, I computed & stored avgRating and numRatings to capture the user's rating tendencies and activity level. For each movie, I computed avgRating and numRatings to represent overall quality and popularity. These metrics allow the recommendation system to leverage both global and personalized signals when ranking movies.

- Recommendation Strategy: User-Based Collaborative Filtering

My system implements a user-based collaborative filtering (CF) approach via Cipher queries that were executed within my database in Neo4j. The strategy follows the following 3 conceptual steps:

i. Identify movies the target user likes:

- A movie is considered "liked" if the user rated it 4 or higher.

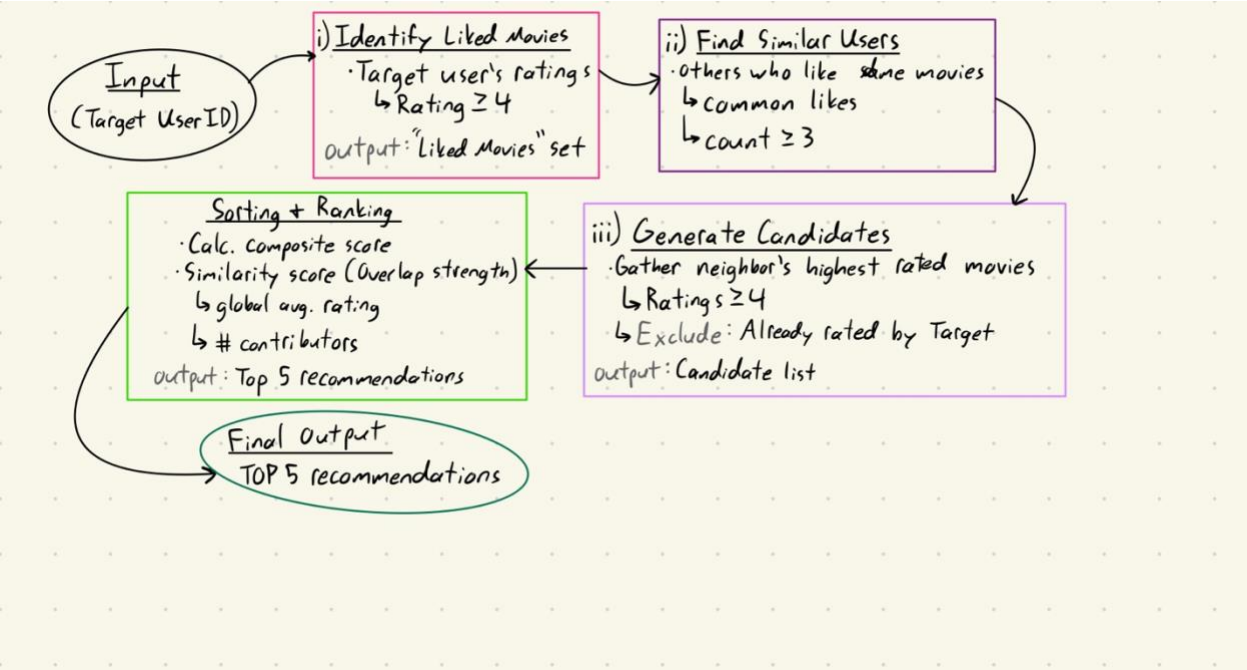
ii. Find similar users

- Users are considered similar if they also liked many of the same movies.
- Similarity is measured by the number of commonly liked movies (commonLikes).
- We apply a tunable similarity threshold (commonLikes ≥ 3) to filter meaningful neighbors.

iii. Generate recommendations

- From the similar users, we gather additional movies they rated highly (rating ≥ 4) that the target user has never rated.
- Each candidate movie receives a composite score based on:
    - similarityScore = sum of overlap strengths from all contributing similar users
    - globalAvgRating = the movie's overall average rating in the dataset
    - numContributors = number of similar users who support the recommendation

The top 5 movies ranked by these metrics are returned as the final recommendations.



- Example figure of my Recommendation Strategy as a flow chart

2. Structure data to incorporate similarity/preference metrics (avg rating + popularity):

To cover this section of the project, I decided to add per-user and per-movie statistics as properties:

a. User preference metrics:

```
// Compute and store each user's average rating and number of ratings
MATCH (u:User)-[r:RATED]->()
WITH u, avg(r.rating) AS avgRating, count(r) AS numRatings
SET u.avgRating  = avgRating,
    u.numRatings = numRatings;

```

Set 12,080 properties — Completed after 28,025 ms

b. Movie preference metrics:

```
// Compute and store each movie's average rating and number of ratings
MATCH (m:Movie)<-[r:RATED]-()
WITH m, avg(r.rating) AS avgRating, count(r) AS numRatings
SET m.avgRating  = avgRating,
    m.numRatings = numRatings;

```

Set 7,412 properties — Completed after 28,364 ms

- With these changes, my data now embeds preference metrics for the following:
  - Popularity: m.numRatings
  - Quality: m.avgRating
  - User bias: u.avgRating, u.numRatings
- These can be recomputed periodically as new ratings are added

3. Implement a recommendation approach using queries:

I decided to select the following strategy:

- Strategy: User-based collaborative filtering
  - A user "likes" a movie if rating ≥ 4
  - Similarity between two users = number of liked movies they share
  - Recommended movies: liked by similar users but unseen by the target user
  - Use m.avgRating and contributor counts as tiebreakers

4. Retrieve and provide the top 5 recommended movies for a user:

```cypher
// User-based CF: top 5 recommendations for a target user
WITH 1 AS targetUserId       // can change the targetUserID to anything (just is 1 for now)
MATCH (u1:User {userId: targetUserId})-[r1:RATED]->(m:Movie)
WHERE r1.rating >= 4   // movies the target user likes

// find "similar" users who also like those movies
MATCH (other:User)-[r2:RATED]->(m)
WHERE other <> u1 AND r2.rating >= 4

// similarity = number of commonly liked movies
WITH u1, other, count(DISTINCT m) AS commonLikes
WHERE commonLikes >= 3   // similarity threshold (tuneable)

// from similar users, pull other movies they like
MATCH (other)-[r3:RATED]->(rec:Movie)
WHERE r3.rating >= 4
  AND NOT (u1)-[:RATED]->(rec)   // ensure the target user has not rated rec

// aggregate a score per recommended movie
WITH rec,
     sum(commonLikes)        AS similarityScore,
     rec.avgRating           AS globalAvgRating,
     count(DISTINCT other)   AS numContributors

RETURN rec.title        AS recommendedTitle,
       similarityScore,
       globalAvgRating,
       numContributors
ORDER BY similarityScore DESC,
         globalAvgRating DESC,
         numContributors DESC
LIMIT 5;
```

- This was my query using my selected strategy (User-based collaborative filtering)

Table RAW

	recommendedTitle	similarityScore	globalAvgRating	numContributors
1	"Star Wars"	49372	4.2929765886287 8	2250
2	"American Beauty (1999)"	46014	4.3173862310384 99	2307
3	"Raiders of the Lost Ark (1981)"	45932	4.4777247414478 82	2018
4	"Silence of the Lambs, The (1991)"	44382	4.3518231186966 645	2029
5	"Shawshank Redemption, The (1994)"	43450	4.5545577009429 685	1930

Started streaming 5 records after 46 ms and completed after 2,489 ms.

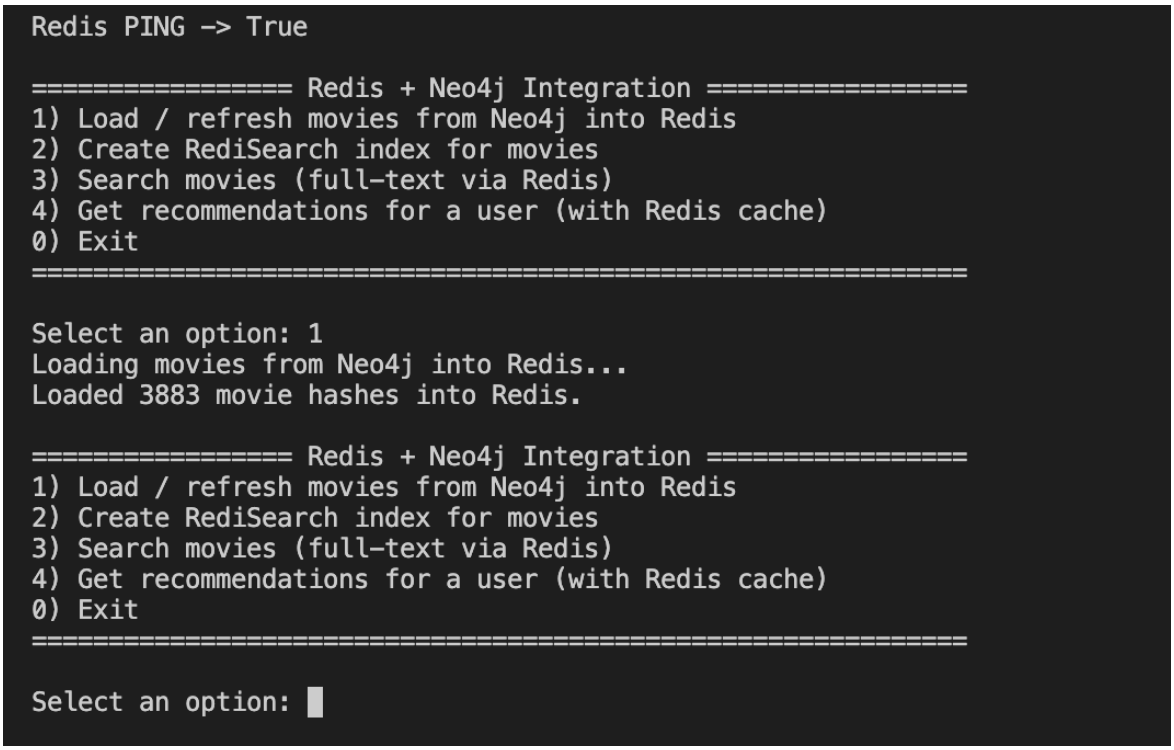
- These were the top 5 recommended movies for user 1

- As can be seen, this mimics my recommendation strategy using my data model design described in the first portion of this part of the project

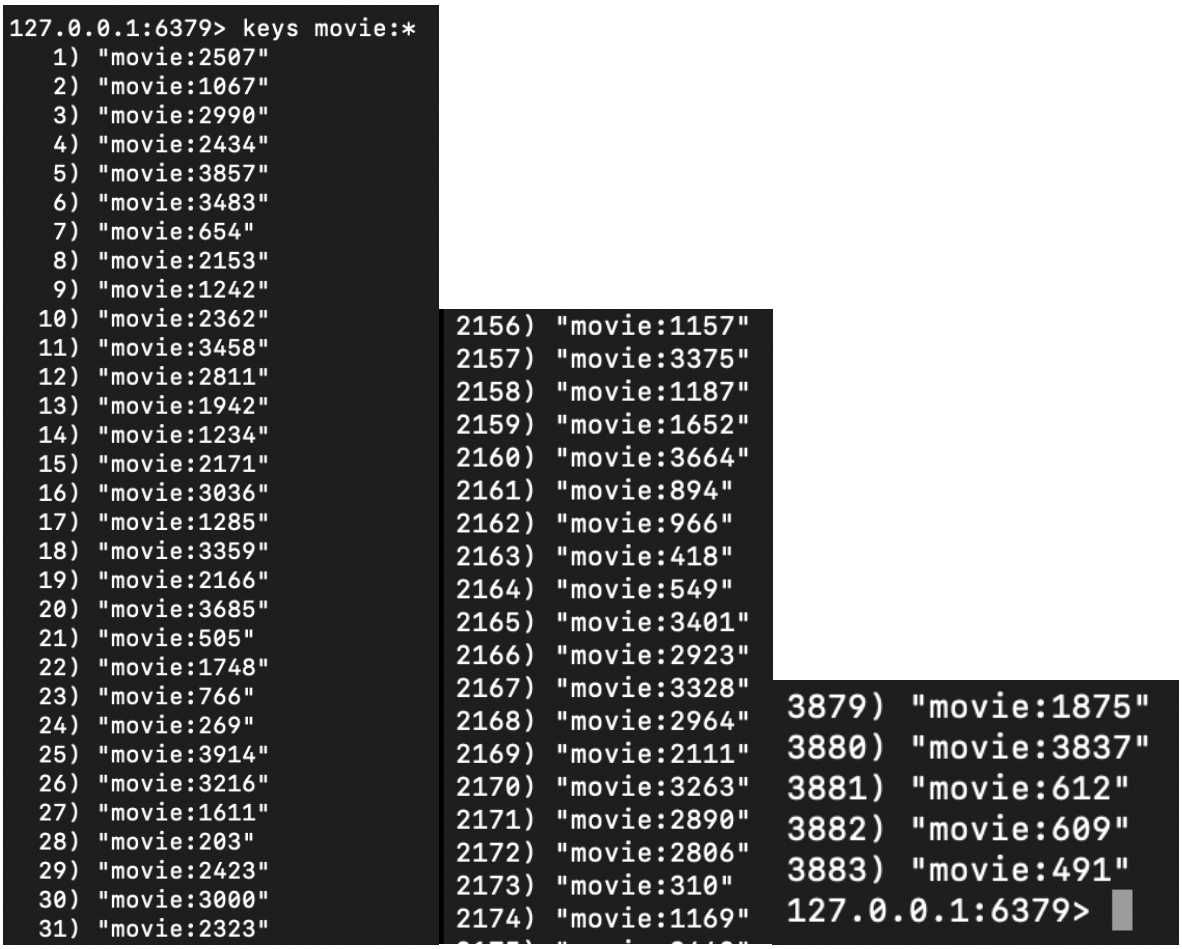
Part 3: Integration with Redis for caching and search.

- I attached my associated python file in for the submission for this section of the project.

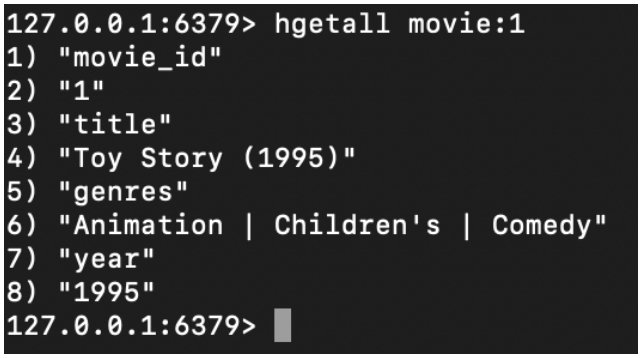
1.



- This screenshot shows that my script extracts the complete movie list from the primary database (Neo4j) and loads it into Redis.

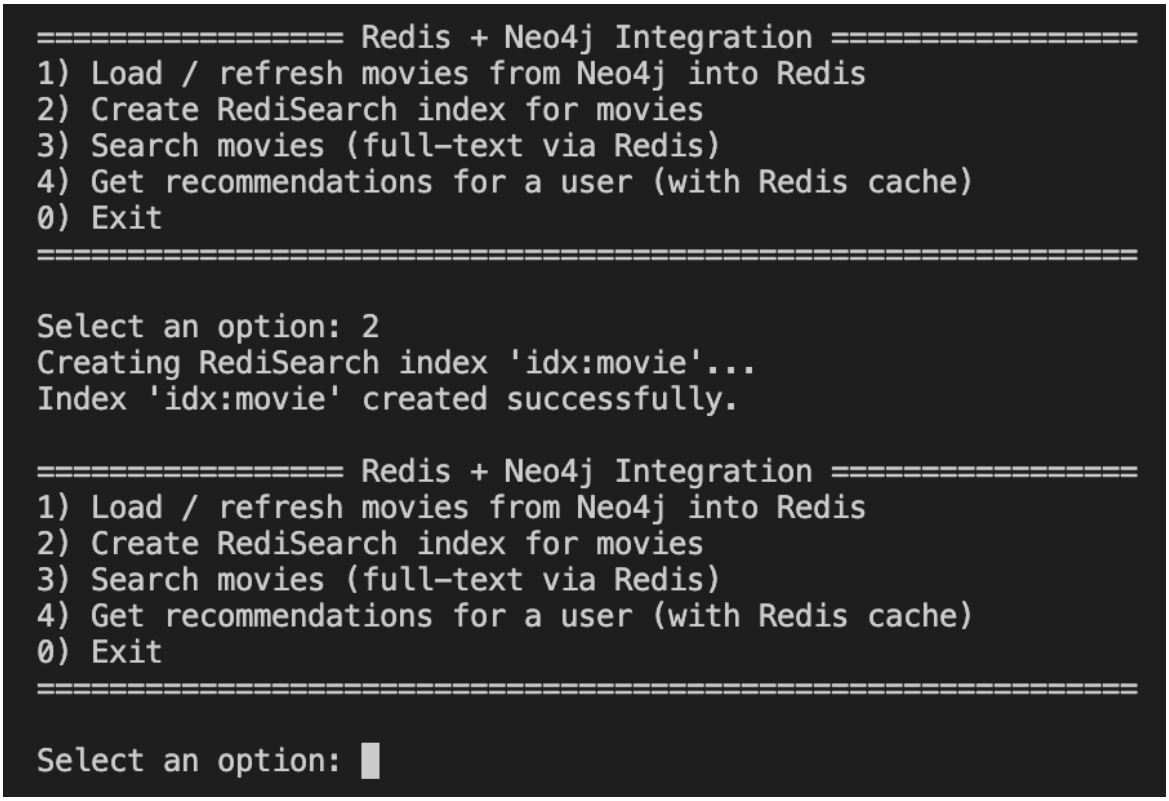


- These show that each movie is stored as a hash using the key format movie:<movieID>



- Example showing movieID 1 and its contents

2.

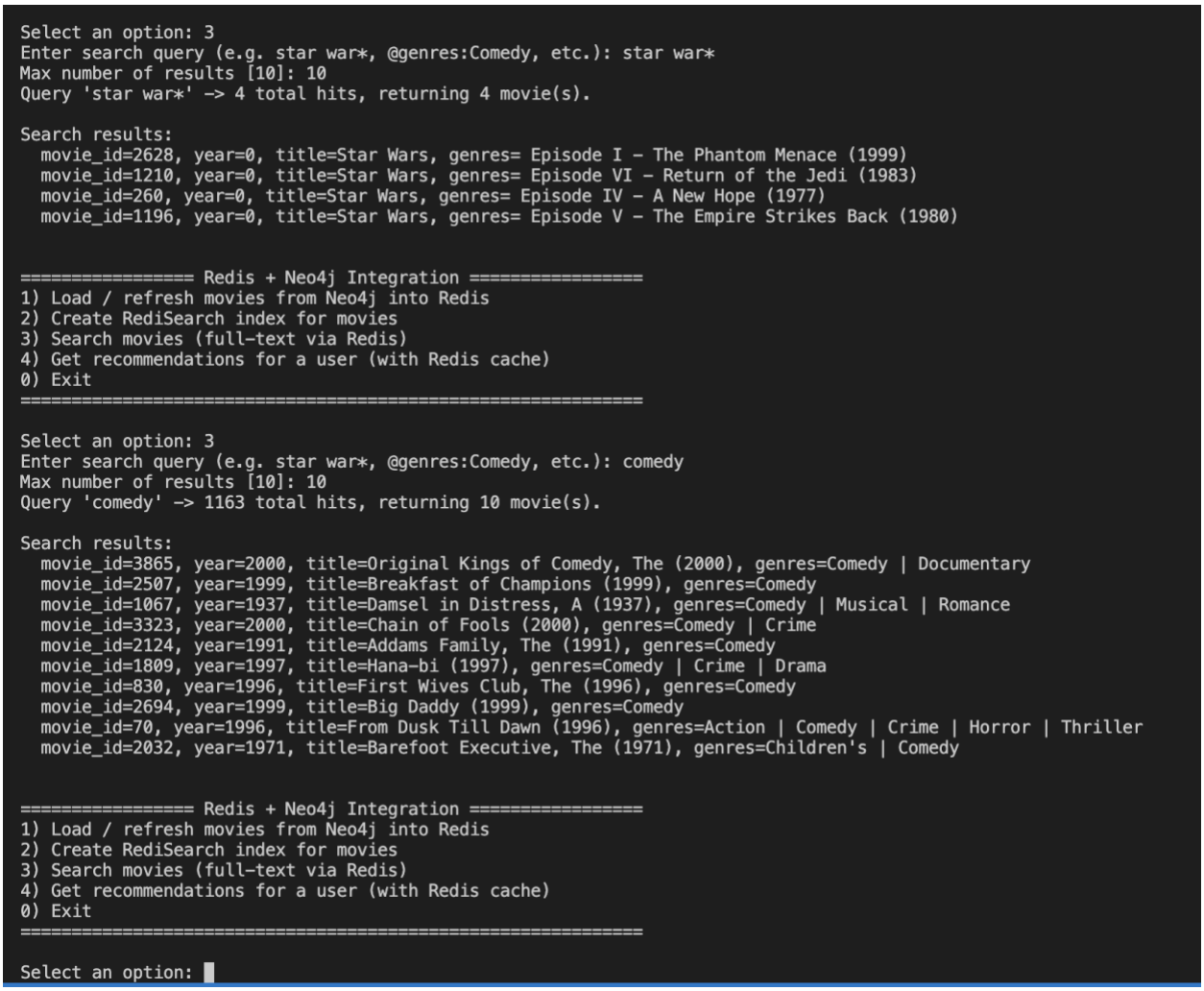


- This screenshot shows that that my script properly created the RediSearch index ‘id:movie’

```
127.0.0.1:6379> FT.INFO idx:movie
 1) index_name
 2) idx:movie
 3) index_options
 4) (empty array)
 5) index_definition
 6) 1) key_type
    2) HASH
    3) prefixes
    4) 1) movie:
    5) default_score
    6) "1"
 7) attributes
 8) 1) 1) identifier
       2) title
       3) attribute
       4) title
       5) type
       6) TEXT
       7) WEIGHT
       8) "5"
    2) 1) identifier
       2) genres
       3) attribute
       4) genres
       5) type
       6) TEXT
       7) WEIGHT
       8) "1"
    3) 1) identifier
       2) year
       3) attribute
       4) year
       5) type
       6) NUMERIC
       7) SORTABLE
       8) UNF
 9) num_docs
10) (integer) 3883
11) max_doc_id
12) (integer) 3883
13) num_terms
14) (integer) 6118
15) num_records
16) (integer) 28844
17) inverted_sz_mb
18) "0.7363471984863281"
19) vector_index_sz_mb
20) "0"
21) total_inverted_index_blocks
22) (integer) 6256
23) offset_vectors_sz_mb
24) "0.02398395538330078"
25) doc_table_size_mb
26) "0.3578014373779297"
27) sortable_values_size_mb
28) "0.08887481689453125"
29) key_table_size_mb
30) "0.11632347106933594"
31) tag_overhead_sz_mb
32) "0"
33) text_overhead_sz_mb
34) "0.2042102813720703"
35) total_index_memory_sz_mb
36) "1.564291000366211"
37) geoshapes_sz_mb
38) "0"
39) records_per_doc_avg
40) "7.428277015686035"
41) bytes_per_record_avg
42) "26.768686294555664"
43) offsets_per_term_avg
44) "0.8718971014022827"
45) offset_bits_per_record_avg
46) "8"
47) hash_indexing_failures
48) (integer) 0
49) total_indexing_time
50) "30.20599937438965"
51) indexing
52) (integer) 0
53) percent_indexed
54) "1"
55) number_of_uses
56) (integer) 1
57) cleaning
58) (integer) 0
59) gc_stats
60)  1) bytes_collected
     2) "0"
     3) total_ms_run
     4) "0"
     5) total_cycles
     6) "0"
     7) average_cycle_time_ms
     8) "nan"
     9) last_run_time_ms
    10) "0"
    11) gc_numeric_trees_missed
    12) "0"
    13) gc_blocks_denied
    14) "0"
61) cursor_stats
62) 1) global_idle
    2) (integer) 0
    3) global_total
    4) (integer) 0
    5) index_capacity
    6) (integer) 128
    7) index_total
    8) (integer) 0
63) dialect_stats
64) 1) dialect_1
    2) (integer) 0
    3) dialect_2
    4) (integer) 0
    5) dialect_3
    6) (integer) 0
    7) dialect_4
    8) (integer) 0
65) Index Errors
66) 1) indexing failures
    2) (integer) 0
    3) last indexing error
    4) N/A
    5) last indexing error key
    6) "N/A"
    7) background indexing status
    8) OK
67) field statistics
68) 1) 1) identifier
       2) title
       3) attribute
       4) title
       5) Index Errors
       6) 1) indexing failures
          2) (integer) 0
          3) last indexing error
          4) N/A
          5) last indexing error key
          6) "N/A"
    2) 1) identifier
       2) genres
       3) attribute
       4) genres
       5) Index Errors
       6) 1) indexing failures
          2) (integer) 0
          3) last indexing error
          4) N/A
          5) last indexing error key
          6) "N/A"
    3) 1) identifier
       2) year
       3) attribute
       4) year
       5) Index Errors
       6) 1) indexing failures
          2) (integer) 0
          3) last indexing error
          4) N/A
          5) last indexing error key
          6) "N/A"
127.0.0.1:6379>
```

- These screenshots show that Redis returned details of the index, including:
    - key type (HASH)
    - prefix (movie:)
    - indexed fields (title, genres, year)
    - number of documents indexed (3883)

This confirms that RediSearch is fully configured and operational for the movie dataset.



- This screenshot shows that my full-text search (option 3) via Redis functions as intended

```
================ Redis + Neo4j Integration ================
1) Load / refresh movies from Neo4j into Redis
2) Create RediSearch index for movies
3) Search movies (full-text via Redis)
4) Get recommendations for a user (with Redis cache)
0) Exit
============================================================

Select an option: 4
User ID: 5785
How many recommendations? [10]: 7
[CACHE MISS] recs:user:5785:k:7 — querying Neo4j...

Recommendations for user 5785:
  movie_id=2933, predicted_rating=5.00
  movie_id=2708, predicted_rating=5.00
  movie_id=2830, predicted_rating=5.00
  movie_id=1076, predicted_rating=5.00
  movie_id=666, predicted_rating=5.00
  movie_id=1925, predicted_rating=5.00
  movie_id=3038, predicted_rating=5.00


================ Redis + Neo4j Integration ================
1) Load / refresh movies from Neo4j into Redis
2) Create RediSearch index for movies
3) Search movies (full-text via Redis)
4) Get recommendations for a user (with Redis cache)
0) Exit
============================================================

Select an option: 4
User ID: 5785
How many recommendations? [10]: 7
[CACHE HIT] recs:user:5785:k:7

Recommendations for user 5785:
  movie_id=2933, predicted_rating=5.00
  movie_id=2708, predicted_rating=5.00
  movie_id=2830, predicted_rating=5.00
  movie_id=1076, predicted_rating=5.00
  movie_id=666, predicted_rating=5.00
  movie_id=1925, predicted_rating=5.00
  movie_id=3038, predicted_rating=5.00


================ Redis + Neo4j Integration ================
1) Load / refresh movies from Neo4j into Redis
2) Create RediSearch index for movies
3) Search movies (full-text via Redis)
4) Get recommendations for a user (with Redis cache)
0) Exit
============================================================

Select an option:
```

```
================ Redis + Neo4j Integration ================
1) Load / refresh movies from Neo4j into Redis
2) Create RediSearch index for movies
3) Search movies (full-text via Redis)
4) Get recommendations for a user (with Redis cache)
0) Exit
============================================================

Select an option: 4
User ID: 1200
How many recommendations? [10]: 8
[CACHE MISS] recs:user:1200:k:8 — querying Neo4j...

Recommendations for user 1200:
  movie_id=3855, predicted_rating=5.00
  movie_id=3746, predicted_rating=5.00
  movie_id=2175, predicted_rating=5.00
  movie_id=83, predicted_rating=5.00
  movie_id=3069, predicted_rating=5.00
  movie_id=1725, predicted_rating=5.00
  movie_id=1123, predicted_rating=5.00
  movie_id=572, predicted_rating=5.00


================ Redis + Neo4j Integration ================
1) Load / refresh movies from Neo4j into Redis
2) Create RediSearch index for movies
3) Search movies (full-text via Redis)
4) Get recommendations for a user (with Redis cache)
0) Exit
============================================================

Select an option: 4
User ID: 1201
How many recommendations? [10]: 8
[CACHE MISS] recs:user:1201:k:8 — querying Neo4j...

Recommendations for user 1201:
  movie_id=993, predicted_rating=5.00
  movie_id=1725, predicted_rating=5.00
  movie_id=824, predicted_rating=5.00
  movie_id=2962, predicted_rating=5.00
  movie_id=83, predicted_rating=5.00
  movie_id=1695, predicted_rating=5.00
  movie_id=363, predicted_rating=5.00
  movie_id=1860, predicted_rating=5.00


================ Redis + Neo4j Integration ================
1) Load / refresh movies from Neo4j into Redis
2) Create RediSearch index for movies
3) Search movies (full-text via Redis)
4) Get recommendations for a user (with Redis cache)
0) Exit
============================================================

Select an option:
```

- These screenshots show the recommendations for a user with the Redis cache. As you can see, when there is a cache hit (same user ID and recommendation #) the recommended movies will be the same. However, when there is a cache miss, the recommended movies will obviously be different

```
127.0.0.1:6379> KEYS recs:user:*:k:*
1) "recs:user:1000:k:3"
2) "recs:user:5785:k:7"
3) "recs:user:1001:k:4"
4) "recs:user:1301:k:2"
5) "recs:user:1300:k:2"
127.0.0.1:6379>
```

- Here is my Redis view showing all recommendation cache keys that are stored

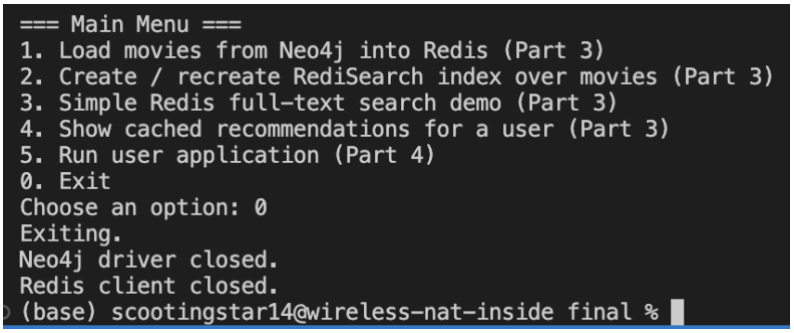
Part 4: User application requirements.

- All the operations for this section are persistent, and the application supports all the listed features described in the project specs.
- Also, all of part 3 still works as well. Here are some screenshots showing the updated menu as well as retrying the old menu options:

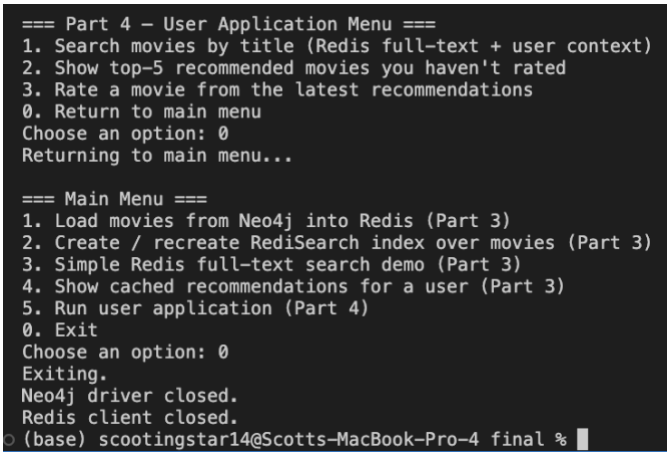
```
=== Main Menu ===
1. Load movies from Neo4j into Redis (Part 3)
2. Create / recreate RediSearch index over movies (Part 3)
3. Simple Redis full-text search demo (Part 3)
4. Show cached recommendations for a user (Part 3)
5. Run user application (Part 4)
0. Exit
Choose an option: 1
Loading movies from Neo4j into Redis...
Loaded 3883 movie hashes into Redis.
=== Main Menu ===
1. Load movies from Neo4j into Redis (Part 3)
2. Create / recreate RediSearch index over movies (Part 3)
3. Simple Redis full-text search demo (Part 3)
4. Show cached recommendations for a user (Part 3)
5. Run user application (Part 4)
0. Exit
Choose an option: 2
Existing index 'idx:movie' dropped.
RediSearch index 'idx:movie' created.

=== Main Menu ===
1. Load movies from Neo4j into Redis (Part 3)
2. Create / recreate RediSearch index over movies (Part 3)
3. Simple Redis full-text search demo (Part 3)
4. Show cached recommendations for a user (Part 3)
5. Run user application (Part 4)
0. Exit
Choose an option: 3
Enter search query (e.g. star war*, @genres:Comedy, etc.): star war*
Max number of results [10]: 9

Query 'star war*' → 4 total hits, returning 4 movie(s).

Search results:
    movie_id=2628, year=0, title=Star Wars, genres= Episode I - The Phantom Menace (1999)
    movie_id=1210, year=0, title=Star Wars, genres= Episode VI - Return of the Jedi (1983)
    movie_id=260, year=0, title=Star Wars, genres= Episode IV - A New Hope (1977)
    movie_id=1196, year=0, title=Star Wars, genres= Episode V - The Empire Strikes Back (1980)

=== Main Menu ===
1. Load movies from Neo4j into Redis (Part 3)
2. Create / recreate RediSearch index over movies (Part 3)
3. Simple Redis full-text search demo (Part 3)
4. Show cached recommendations for a user (Part 3)
5. Run user application (Part 4)
0. Exit
Choose an option: 3
Enter search query (e.g. star war*, @genres:Comedy, etc.): comedy
Max number of results [10]: 2

Query 'comedy' → 1163 total hits, returning 2 movie(s).

Search results:
    movie_id=2507, year=1999, title=Breakfast of Champions (1999), genres=Comedy
    movie_id=1067, year=1937, title=Damsel in Distress, A (1937), genres=Comedy | Musical | Romance
```

```
Choose an option: 4
Enter user ID for recommendations: 1000
How many recommendations? [10]: 3
[CACHE MISS] recs:user:1000:k:3 -> querying Neo4j...

Recommendations for user 1000:
    movie_id=993, predicted_rating=5.00
    movie_id=83, predicted_rating=5.00
    movie_id=1236, predicted_rating=5.00

=== Main Menu ===
1. Load movies from Neo4j into Redis (Part 3)
2. Create / recreate RediSearch index over movies (Part 3)
3. Simple Redis full-text search demo (Part 3)
4. Show cached recommendations for a user (Part 3)
5. Run user application (Part 4)
0. Exit
Choose an option: 4
Enter user ID for recommendations: 1000
How many recommendations? [10]: 3
[CACHE HIT] recs:user:1000:k:3

Recommendations for user 1000:
    movie_id=993, predicted_rating=5.00
    movie_id=83, predicted_rating=5.00
    movie_id=1236, predicted_rating=5.00

=== Main Menu ===
1. Load movies from Neo4j into Redis (Part 3)
2. Create / recreate RediSearch index over movies (Part 3)
3. Simple Redis full-text search demo (Part 3)
4. Show cached recommendations for a user (Part 3)
5. Run user application (Part 4)
0. Exit
Choose an option: 4
Enter user ID for recommendations: 1001
How many recommendations? [10]: 4
[CACHE MISS] recs:user:1001:k:4 -> querying Neo4j...

Recommendations for user 1001:
    movie_id=1925, predicted_rating=5.00
    movie_id=1076, predicted_rating=5.00
    movie_id=666, predicted_rating=5.00
    movie_id=3517, predicted_rating=5.00

=== Main Menu ===
1. Load movies from Neo4j into Redis (Part 3)
2. Create / recreate RediSearch index over movies (Part 3)
3. Simple Redis full-text search demo (Part 3)
4. Show cached recommendations for a user (Part 3)
5. Run user application (Part 4)
0. Exit
Choose an option:
```

- The first screenshot shows option 1 – 3 still function as intended
- The second shows option 4 still functions as intended



- As can be seen, selecting option 0 properly closes all database connections in a smooth manner before the application exits.



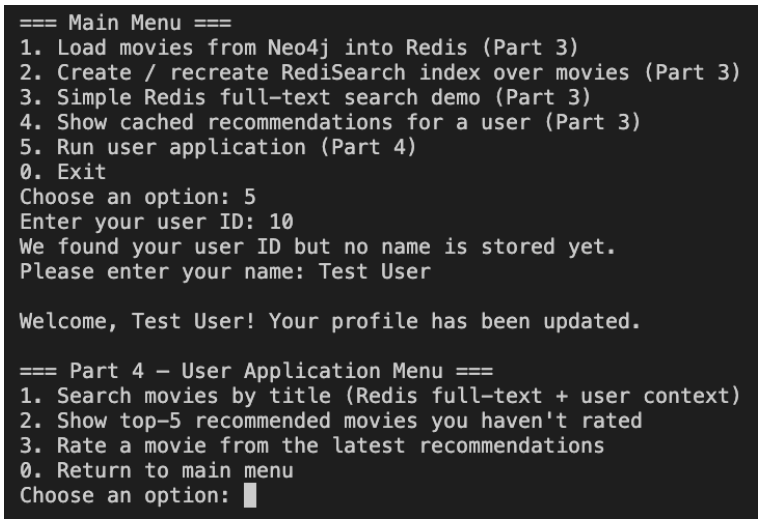
- When running option 5 (User Application requirements) and selecting option 0, users are sent back to the main menu. Again, selecting option 0 from the main menu cleanly shuts down database connections.

```
=== Main Menu ===
1. Load movies from Neo4j into Redis (Part 3)
2. Create / recreate RediSearch index over movies (Part 3)
3. Simple Redis full-text search demo (Part 3)
4. Show cached recommendations for a user (Part 3)
5. Run user application (Part 4)
0. Exit
Choose an option: 1
Loading movies from Neo4j into Redis...
Loaded 3883 movie hashes into Redis.
=== Main Menu ===
1. Load movies from Neo4j into Redis (Part 3)
2. Create / recreate RediSearch index over movies (Part 3)
3. Simple Redis full-text search demo (Part 3)
4. Show cached recommendations for a user (Part 3)
5. Run user application (Part 4)
0. Exit
Choose an option: 5
Enter your user ID: 14
We found your user ID but no name is stored yet.
Please enter your name: Scott

Welcome, Scott! Your profile has been updated.

=== Part 4 – User Application Menu ===
1. Search movies by title (Redis full-text + user context)
2. Show top-5 recommended movies you haven't rated
3. Rate a movie from the latest recommendations
0. Return to main menu
Choose an option:
```

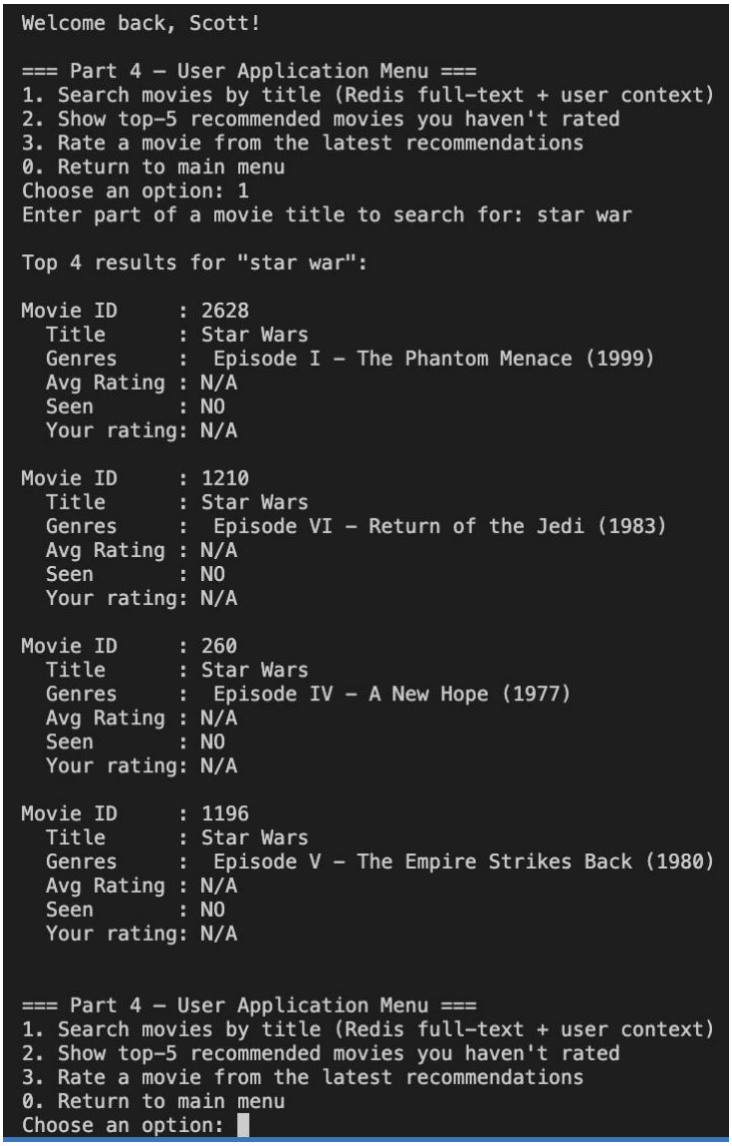
- Here is an example of running option 5. I select option 1 and set my name to update my profile.

```
=== Main Menu ===
1. Load movies from Neo4j into Redis (Part 3)
2. Create / recreate RediSearch index over movies (Part 3)
3. Simple Redis full-text search demo (Part 3)
4. Show cached recommendations for a user (Part 3)
5. Run user application (Part 4)
0. Exit
Choose an option: 5
Enter your user ID: 14

Welcome back, Scott!

=== Part 4 – User Application Menu ===
1. Search movies by title (Redis full-text + user context)
2. Show top-5 recommended movies you haven't rated
3. Rate a movie from the latest recommendations
0. Return to main menu
Choose an option:
```

- Here, I enter my ID (14) and log back into my profile.



- Here is another example of me running option 5 and getting a new user profile.

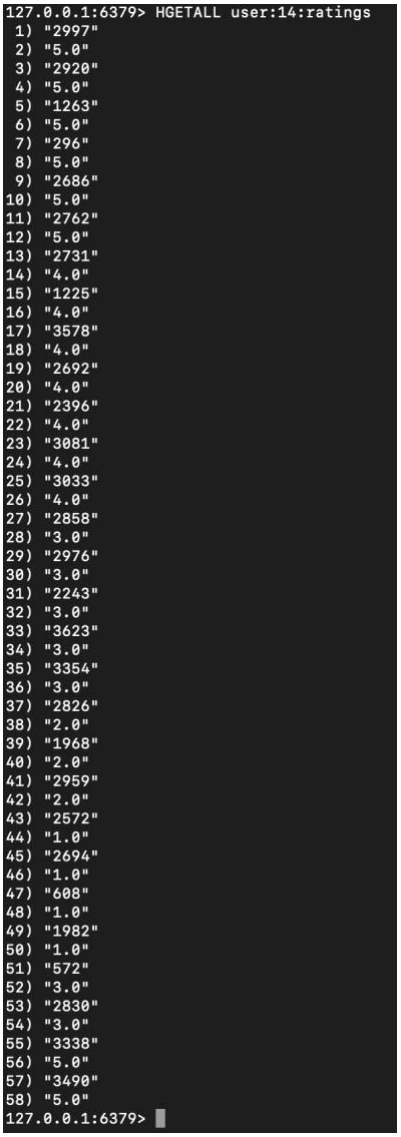


- In this portion, I am logged into my profile and select option 1 so I can search movies by title with Redis full text & user content. Here, I want to see movies with “star war” in the title since it is my favorite movie series. Here, I am shown the 4 results in the database along with their associated data.

- In this portion, I select option 2 so I can see my top 5 recommended movies that I have yet to rate. Now, I select option 3 so I can rate on of those 5 movies. I select to rate the movie Calbaret Balkan (Bure Baruta), movie 1 in the list, and I give it a rating of 3 out of 5 stars because I decided that I didn't like the movie even though it was one of the top 5 movies recommended to me. Now, the average rating for the movie is updated to 3.52 and is stored in my Redis server. After this, I decided to select option 2 again so I could see how my top 5 recommended movie list has changed. Because I rated the last movie

not well, my list has changed which makes sense intuitively. Then, I decided to use option 1 to search for the movie that I rated 3 out of 5 stars before to see its updated associated data. As can be seen, the data is different since the average rating, my rating, and the “seen” section has been updated with the corresponding data.

- Here, we can see an example of me checking out more movies that I have yet to see and are recommended for me. I rank one of the movies very highly. Then, I decided to use some random inputs for error checking to ensure my script is foolproof. Then, I check the movie that I rated to see that its data has been updated accordingly.



- Here is a quick screenshot showing that my Redis database is being updated with the average ratings throughout my operations.

- These screenshots of my file session running demonstrate that all user application requirements are met

Final Project Summary:

This project implemented a full movie-recommendation pipeline using Neo4j as the primary graph database and utilized Redis as a high-performance caching and search layer. Neo4j was selected over MongoDB and PostgreSQL based on the user-movie domain being naturally graph-structured, allowing user preferences, ratings, and similarity relationships to be expressed

intuitively as nodes and edges. The property-model graph, combined with Cypher, enabled efficient traversal-based queries needed for collaborative filtering.

The data schema consisted of User and Movie nodes connected through RATED relationships. Each was enriched with aggregate metrics such as rating and rating counts. Indexing strategies included Neo4j uniqueness constraints to prevent duplicates and speed up imports, Redis Hash keys for movie metadata (movie:<id>), a RediSearch full-text for movie title/genre queries, and structured cache keys (recs:user:<userId>:k:<k>) to support fast recommendation retrieval.

The system's design leveraged user-based collaborative filtering, where similarity was defined by the number of commonly "liked" movies (ratings >= 4). Recommendations were scored using overlap strength, contributor counts, and global movie averages. Redis caching was used to avoid recomputing recommendations for identical (user, k) pairs, significantly improving responsiveness and reducing repeated Neo4j queries. The combination of similarity-based scoring, aggregated movie metrics, and Redis TTL-based caching produced accurate and efficient recommendation output.

In terms of performance, Neo4j handled large rating imports efficiently after tuning memory parameters, and Redis delivered constant-time lookups for repeated recommendations and full-text search functionality. The user application demonstrated persistence, profile management, rating updates, dynamic recommendation changes, and accurate synchronization between Neo4j and Redis.

Lessons Learned:

Throughout this project, I gained a ton of experience designing multi-database systems and understanding how different technologies complement one another. I learned how graph models simplify recommendation workloads, how caching dramatically improves application speed/responsiveness, and how indexing strategies affect search performance. Integrating Neo4j and Redis also highlighted the importance of schema planning, transaction batching, memory tuning, and clear key-design patterns when building scalable data-driven applications.