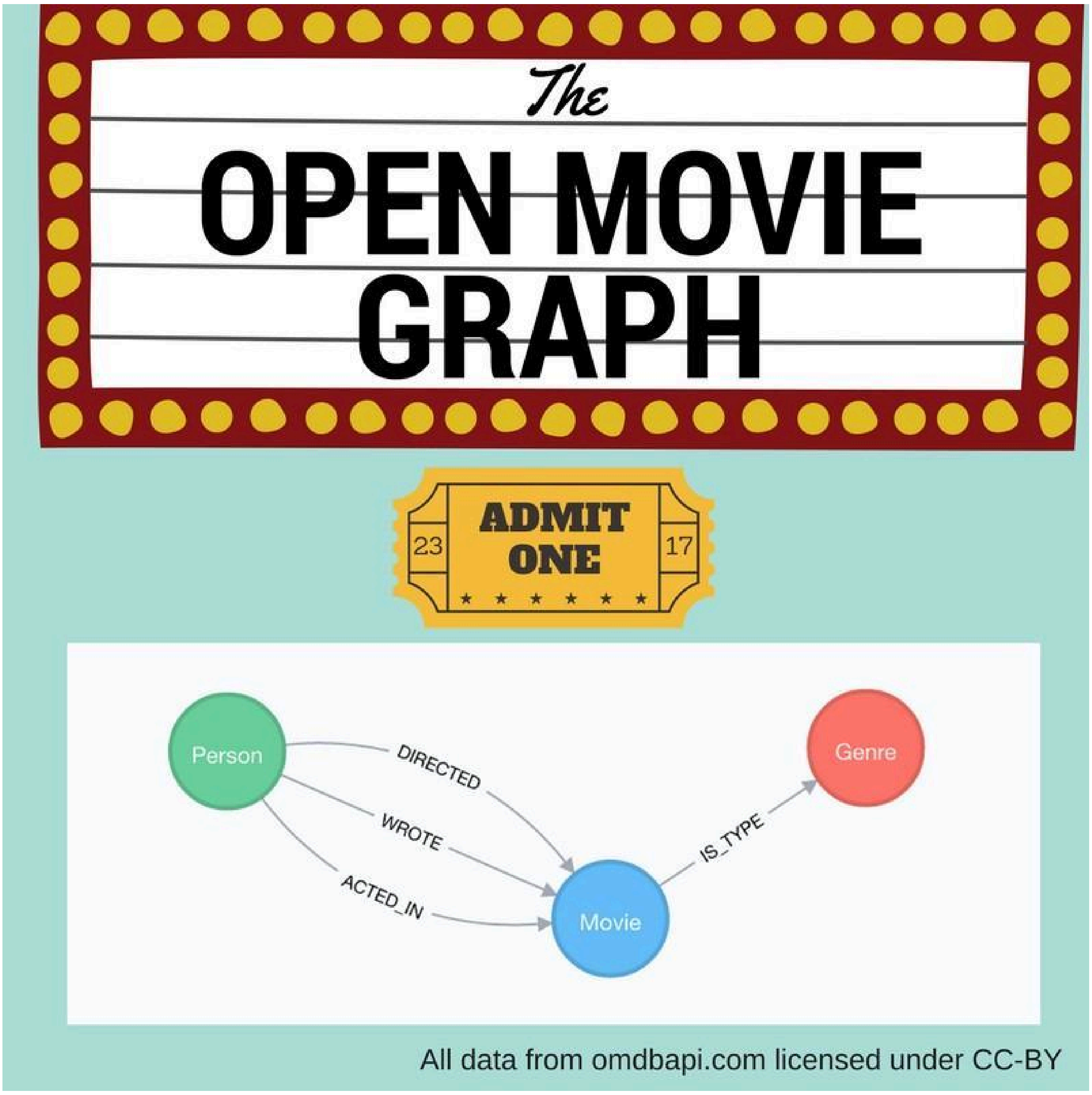
Final Project : Build Recommendations system using movies data

Personalized Product Recommendations with Neo4j



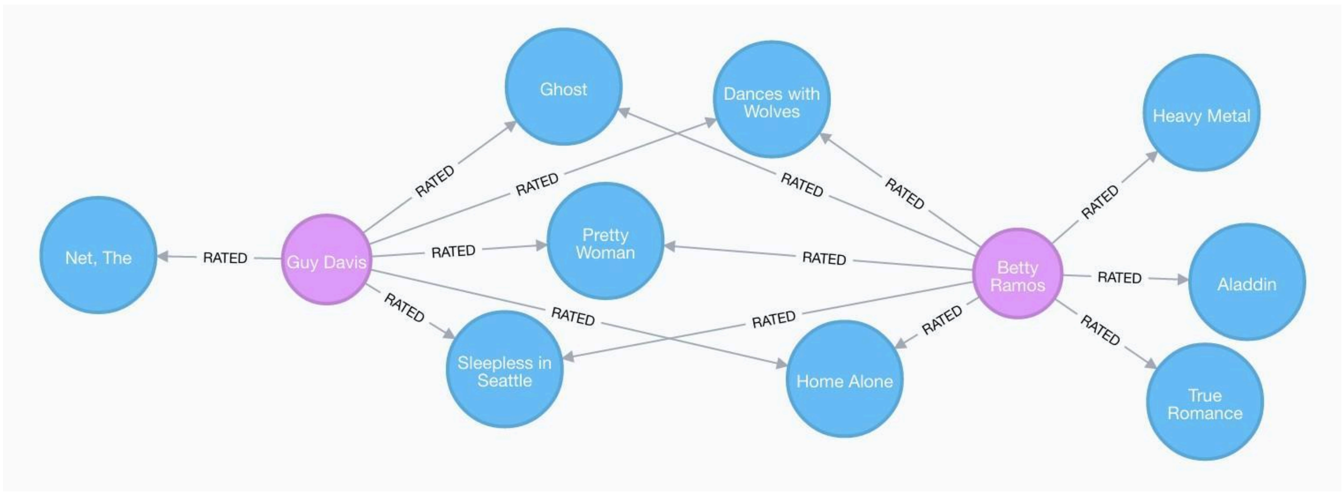
Recommendations

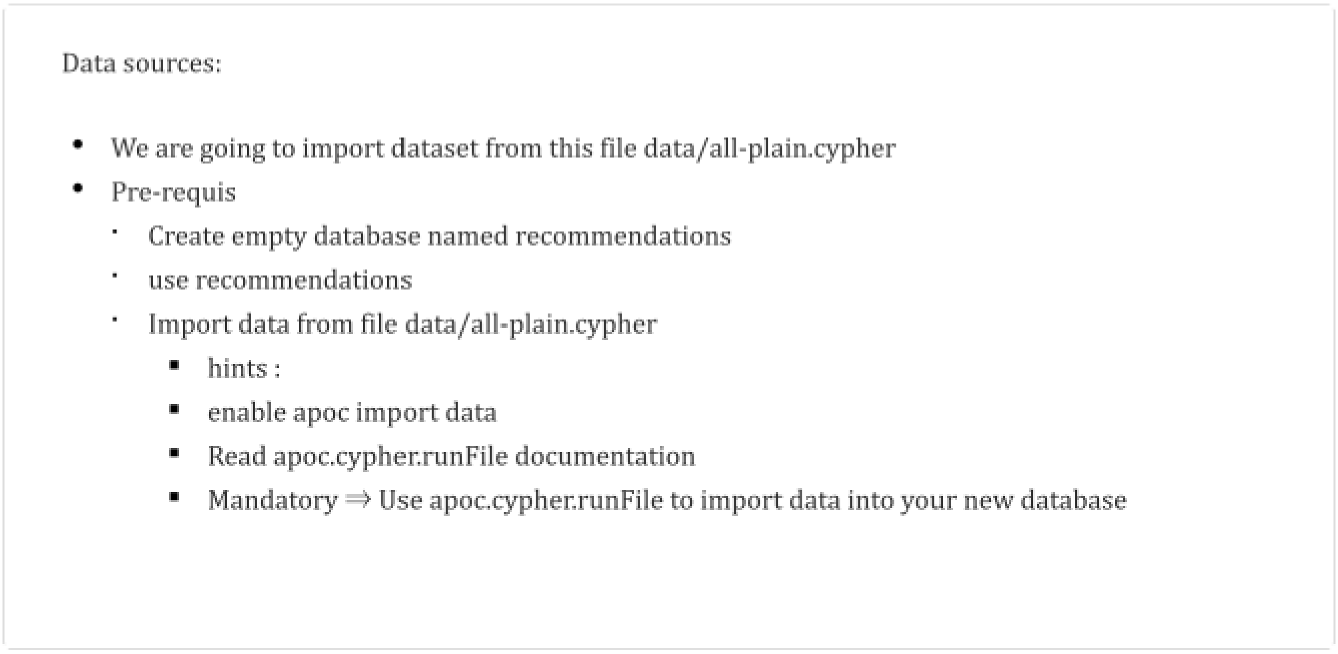
Personalized product recommendations can increase conversions, improve sales rates and provide a better experice for users. In this Neo4j Browser guide, we'll take a look at how you can generate graph-based real-time personalized product recommendations using a dataset of movies and movie ratings, but these techniques can be applied to many different types of products or content.

Graph-Based Recommendations

Generating **personalized recommendations** is one of the most common use cases for a graph database. Some of the main benefits of using graphs to generate recommendations include:

1. **Performance**. Index-free adjacency allows for **calculating recommendations in real time**, ensuring the recommendation is always relevant and reflecting up-to-date information.
2. **Data model**. The labeled property graph model allows for easily combining datasets from multiple sources, allowing enterprises to **unlock value from previously separated data silos.**




Data sources:

- We are going to import dataset from this file data/all-plain.cypher
- Pre-requis
  - Create empty database named recommendations
  - use recommendations
  - Import data from file data/all-plain.cypher
    - hints :
    - enable apoc import data
    - Read apoc.cypher.runFile documentation
    - Mandatory ⇒ Use apoc.cypher.runFile to import data into your new database

The Open Movie Graph Data Model

The Property Graph Model

The data model of graph databases is called the labeled property graph model.

Nodes: The entities in the data.

Labels: Each node can have one or more **label** that specifies the type of the node.

Relationships: Connect two nodes. They have a single direction and type.

Properties: Key-value pair properties can be stored on both nodes and relationships.

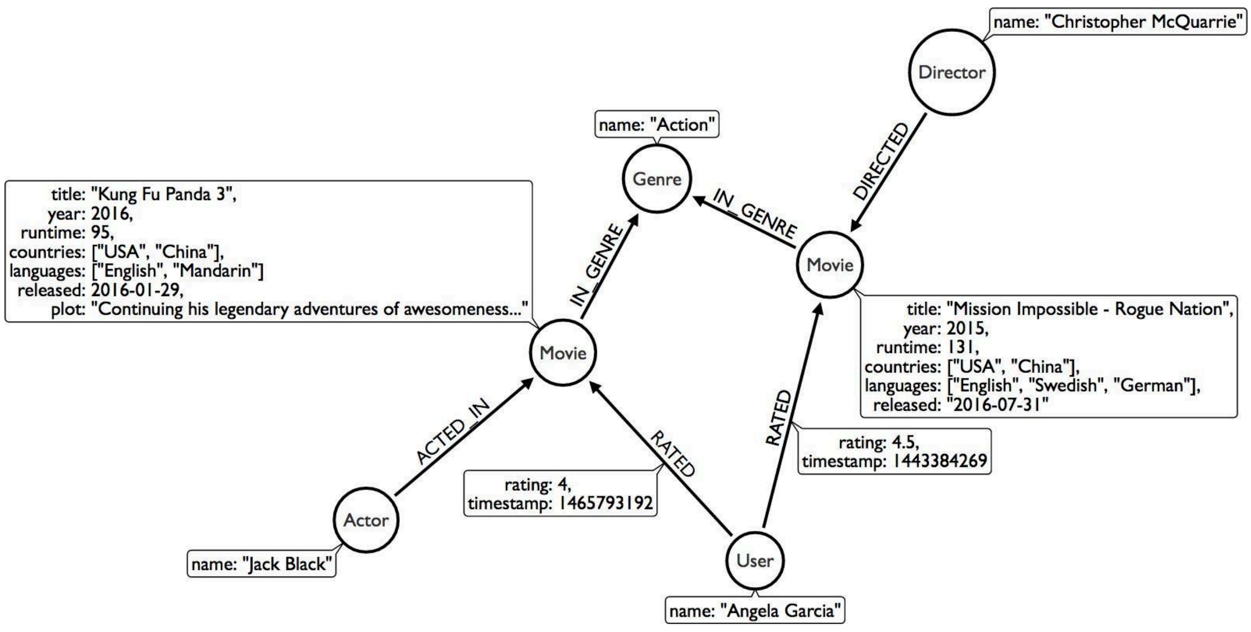
Eliminate Data Silos

In this use case, we are using graphs to combine data from multiple sources.

Product Catalog: Data describing movies comes from the product catalog silo.

User Purchases / Reviews: Data on user purchases and reviews comes from the user or transaction source.

By combining these two in the graph, we are able to query across datasets to generate personalized product recommendations.


name: "Christopher McQuarrie"
Director
DIRECTED
name: "Action"
Genre
IN_GENRE
IN_GENRE
Movie
title: "Kung Fu Panda 3", year: 2016, runtime: 95, countries: ["USA", "China"], languages: ["English", "Mandarin"], released: 2016-01-29, plot: "Continuing his legendary adventures of awesomeness..."
Movie
title: "Mission Impossible - Rogue Nation", year: 2015, runtime: 131, countries: ["USA", "China"], languages: ["English", "Swedish", "German"], released: "2016-07-31"
ACTED_IN
RATED
rating: 4, timestamp: 1465793192
RATED
rating: 4.5, timestamp: 1443384269
Actor
name: "Jack Black"
User
name: "Angela Garcia"

Nodes

`Movie`, `Actor`, `Director`, `User`, `Genre` are the labels used in this example.

Relationships

`ACTED_IN`, `IN_GENRE`, `DIRECTED`, `RATED` are the relationships used in this example.

Properties

`title`, `name`, `year`, `rating` are some of the properties used in this example.

Memo on Cypher

In order to work with our labeled property graph, we need a query language for graphs.

Graph Patterns

Cypher is the query language for graphs and is centered around **graph patterns**. Graph patterns are expressed in Cypher using ASCII-art like syntax.

Nodes

Nodes are defined within parentheses `()`. Optionally, we can specify node label(s): `(:Movie)`

Relationships

Relationships are defined within square brackets `[]`. Optionally we can specify type and direction:

<code>(:Movie)<strong><-[:RATED]-</strong>(:User)</code>

Variables

Graph elements can be bound to variables that can be referred to later in the query:

<code>(<strong>m</strong>:Movie)<-[<strong>r</strong>:RATED]-(<strong>u</strong>:User)</code>

Predicates

Filters can be applied to these graph patterns to limit the matching paths. Boolean logic operators, regular expressions and string comparison operators can be used here within the `WHERE` clause, e.g. `WHERE m.title CONTAINS 'Matrix'`

Aggregations

There is an implicit group of all non-aggregated fields when using aggregation functions such as `count`.

Use the Cypher Refcard as a syntax reference.

WORK TO DO

Dissecting a Cypher Statement

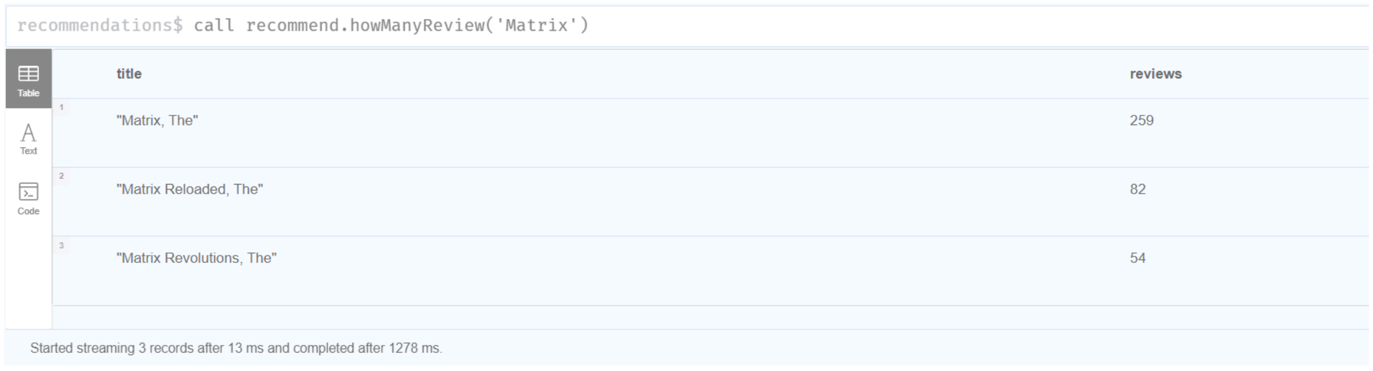
Let’s implemente a Cypher query that answers the question "How many reviews does each Matrix movie have?". Don’t worry if this seems complex, we’ll build up our understanding of Cypher as we move along.

```
MATCH (m:Movie)<-[:RATED]-(u:User)

WHERE m.title CONTAINS 'Matrix'

WITH m, count(*) AS reviews

RETURN m.title AS movie, reviews

ORDER BY reviews;
```

Int: Replace ??? by the corrects values

2/ After you completed previous request and tested it, create your own User defined procedure

to do the same work.



```
recommendations$ call recommend.howManyReview('Matrix')
```

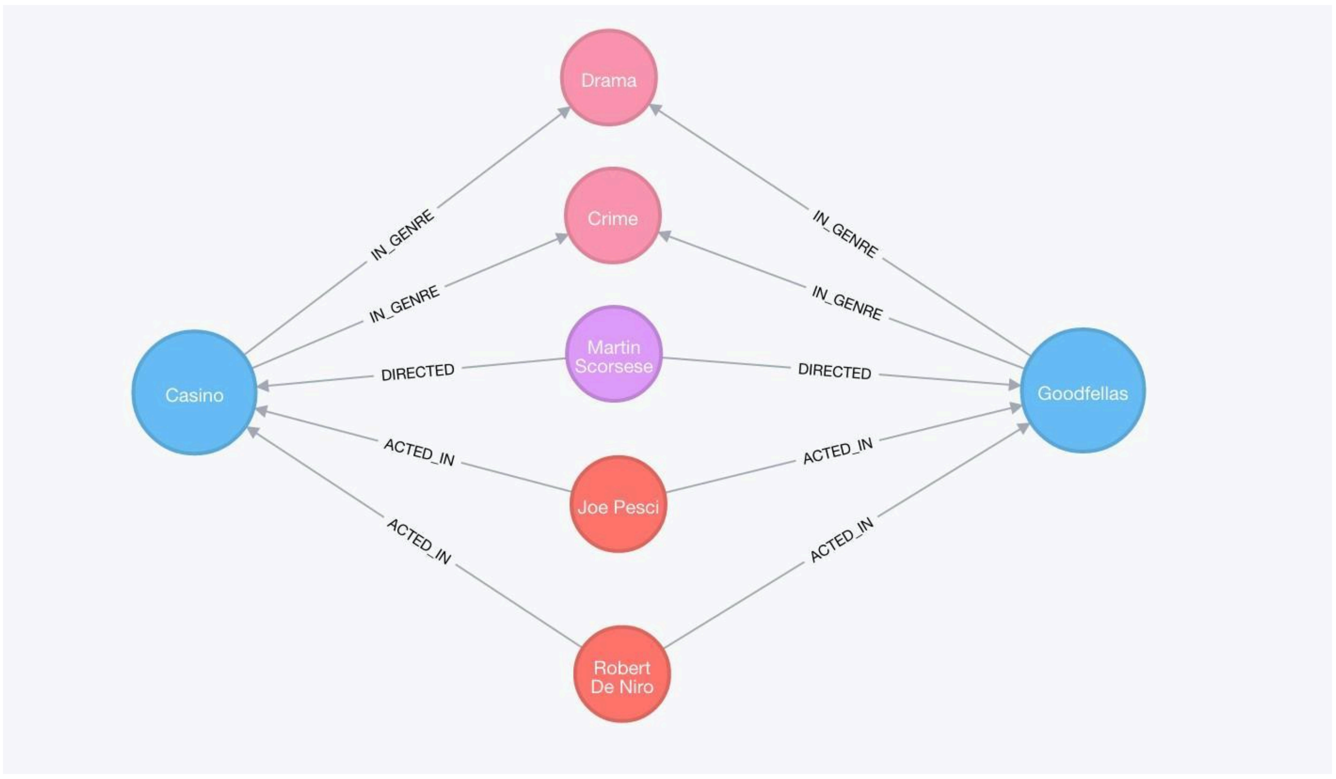
	title	reviews
1	"Matrix, The"	259
2	"Matrix Reloaded, The"	82
3	"Matrix Revolutions, The"	54

Started streaming 3 records after 13 ms and completed after 1278 ms.

Personalized Recommendations

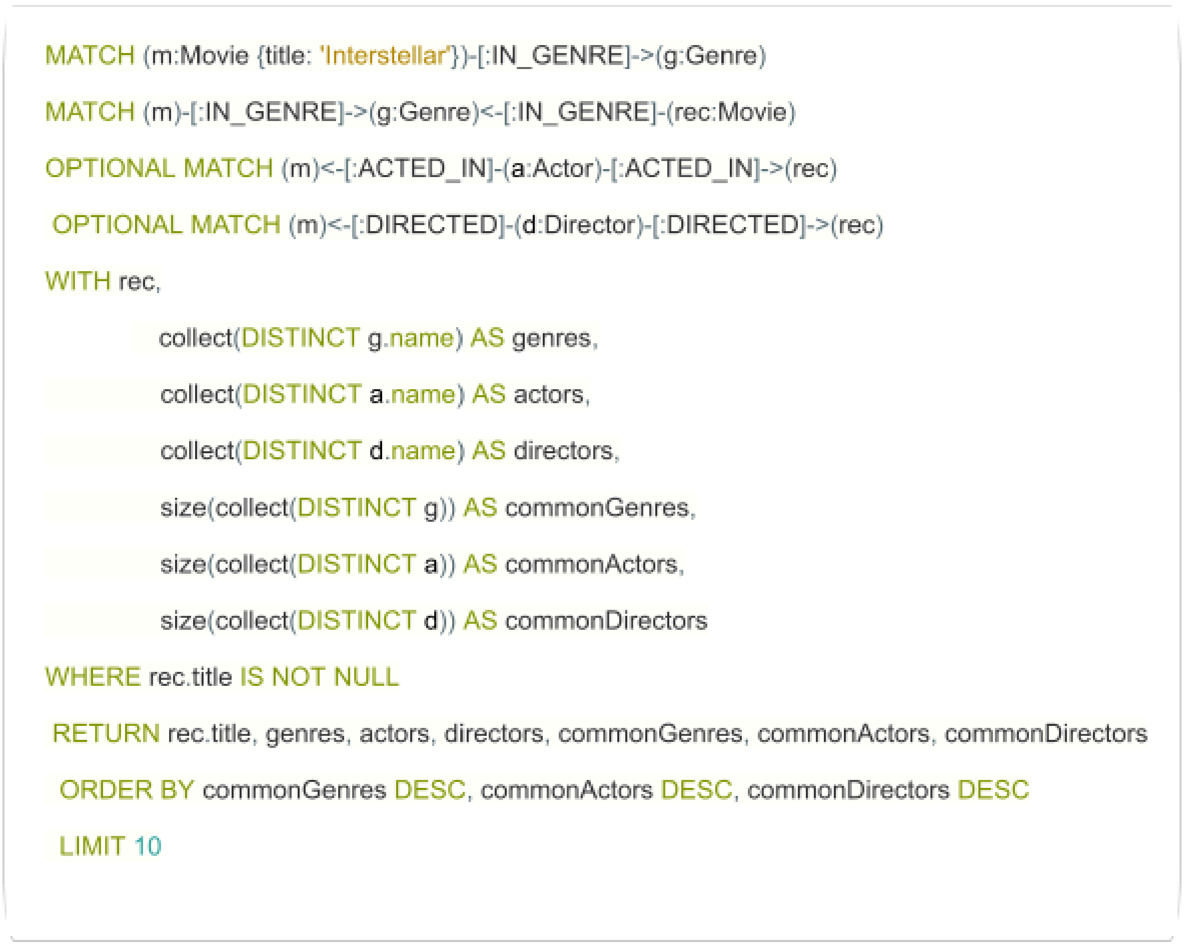
Now let's start generating some recommendations. There are two basic approaches to recommendation algorithms.

Content-Based Filtering



Recommend items that are similar to those that a user is viewing, rated highly or purchased previously.

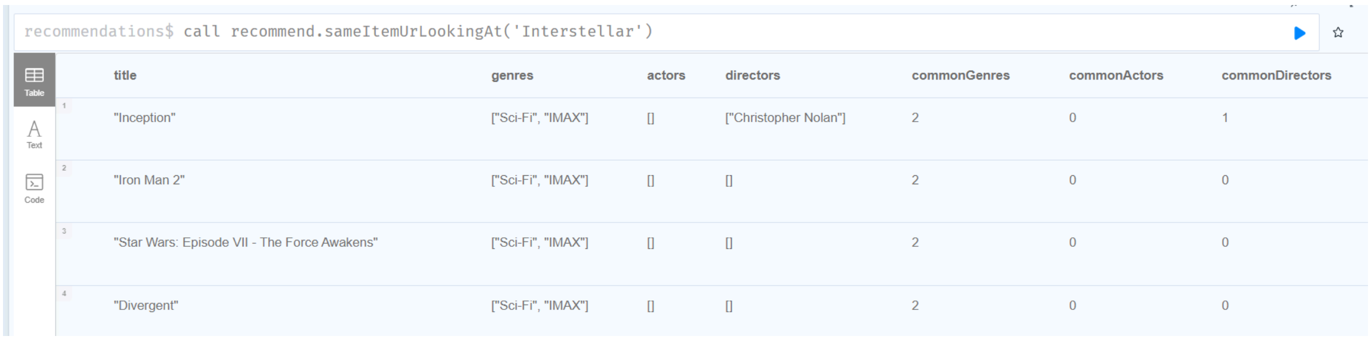
1/ "Find Items similar to the item you're looking at now"



```
MATCH (m:Movie {title: 'Interstellar'})-[:IN_GENRE]->(g:Genre)
MATCH (m)-[:IN_GENRE]->(g:Genre)<-[:IN_GENRE]-(rec:Movie)
OPTIONAL MATCH (m)<-[:ACTED_IN]-(a:Actor)-[:ACTED_IN]->(rec)
 OPTIONAL MATCH (m)<-[:DIRECTED]-(d:Director)-[:DIRECTED]->(rec)
WITH rec,
          collect(DISTINCT g.name) AS genres,
          collect(DISTINCT a.name) AS actors,
          collect(DISTINCT d.name) AS directors,
          size(collect(DISTINCT g)) AS commonGenres,
          size(collect(DISTINCT a)) AS commonActors,
          size(collect(DISTINCT d)) AS commonDirectors
WHERE rec.title IS NOT NULL
 RETURN rec.title, genres, actors, directors, commonGenres, commonActors, commonDirectors
  ORDER BY commonGenres DESC, commonActors DESC, commonDirectors DESC
  LIMIT 10
```

2/ After you completed previous request and tested it, create your own User defined procedure

to do the same work.

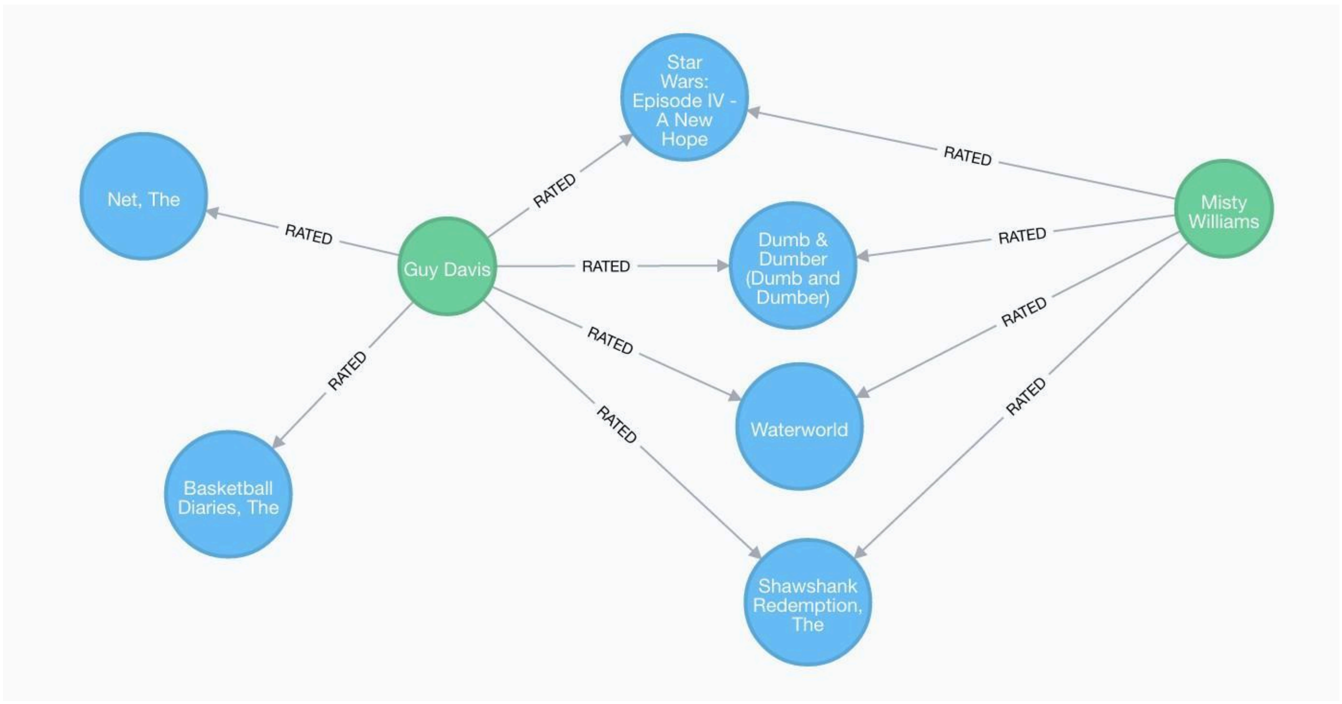


```
recommendations$ call recommend.sameItemUrLookingAt('Interstellar')
```

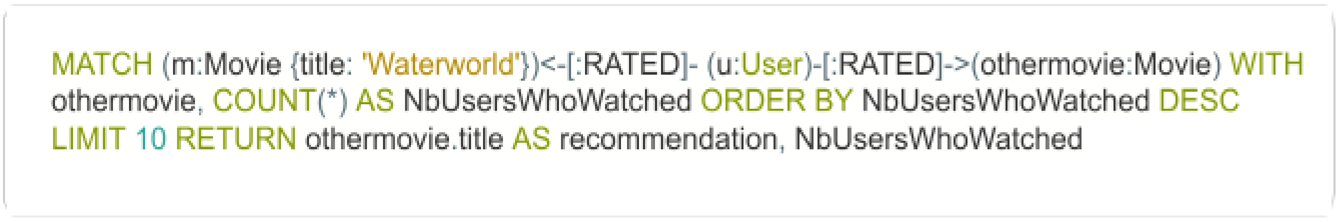
	title	genres	actors	directors	commonGenres	commonActors	commonDirectors
1	"Inception"	["Sci-Fi", "IMAX"]	[]	["Christopher Nolan"]	2	0	1
2	"Iron Man 2"	["Sci-Fi", "IMAX"]	[]	[]	2	0	0
3	"Star Wars: Episode VII - The Force Awakens"	["Sci-Fi", "IMAX"]	[]	[]	2	0	0
4	"Divergent"	["Sci-Fi", "IMAX"]	[]	[]	2	0	0

Collaborative Filtering

Use the preferences, ratings and actions of other users in the network to find items to recommend.


Star Wars: Episode IV - A New Hope
Net, The
Guy Davis
Dumb & Dumber (Dumb and Dumber)
Misty Williams
Waterworld
Basketball Diaries, The
Shawshank Redemption, The
RATED

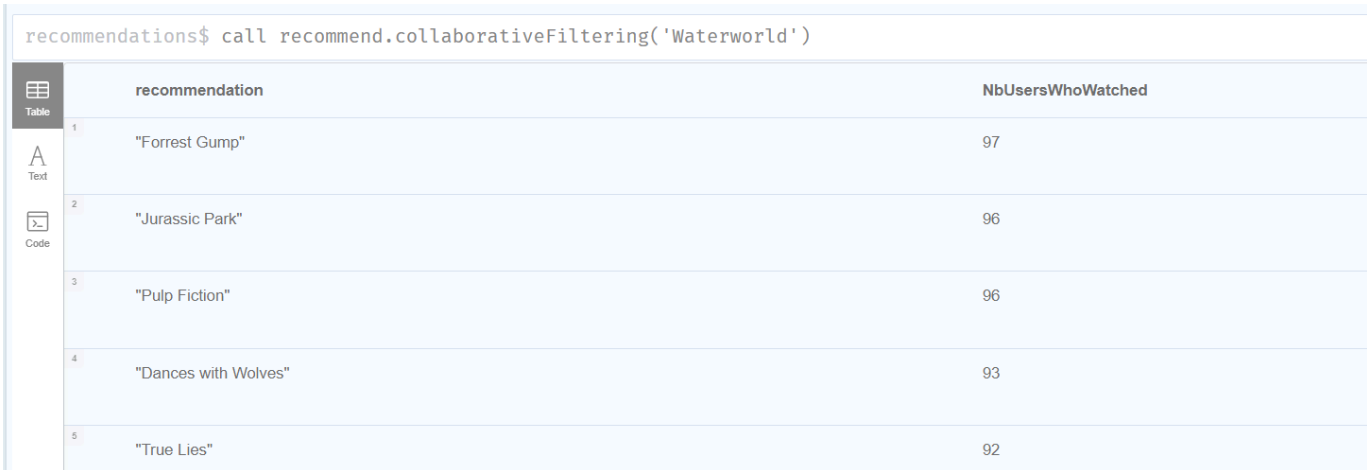
1/ " Get Users who got this item, also got that other item."



```
MATCH (m:Movie {title: 'Waterworld'})<-[:RATED]- (u:User)-[:RATED]->(othermovie:Movie) WITH
othermovie, COUNT(*) AS NbUsersWhoWatched ORDER BY NbUsersWhoWatched DESC
LIMIT 10 RETURN othermovie.title AS recommendation, NbUsersWhoWatched
```

2/ After you completed previous request and tested it, create your own User defined procedure

to do the same work.

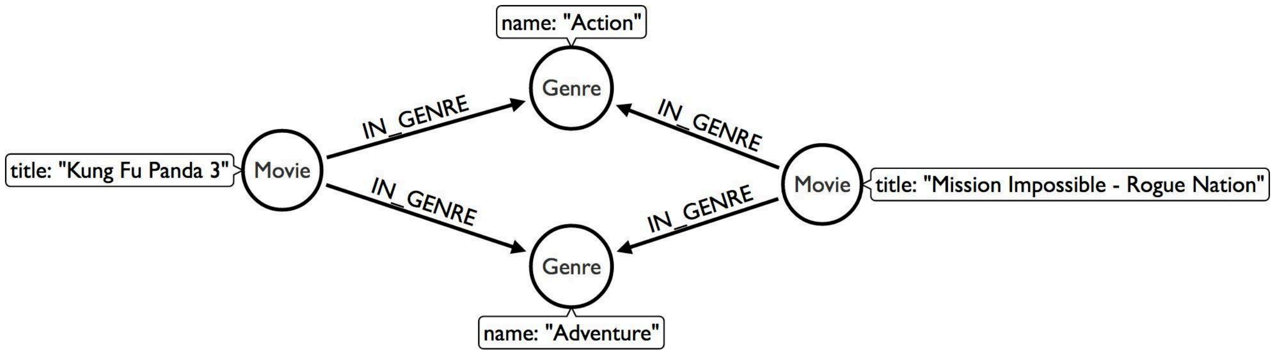


```
recommendations$ call recommend.collaborativeFiltering('Waterworld')
```

	recommendation	NbUsersWhoWatched
1	"Forrest Gump"	97
2	"Jurassic Park"	96
3	"Pulp Fiction"	96
4	"Dances with Wolves"	93
5	"True Lies"	92

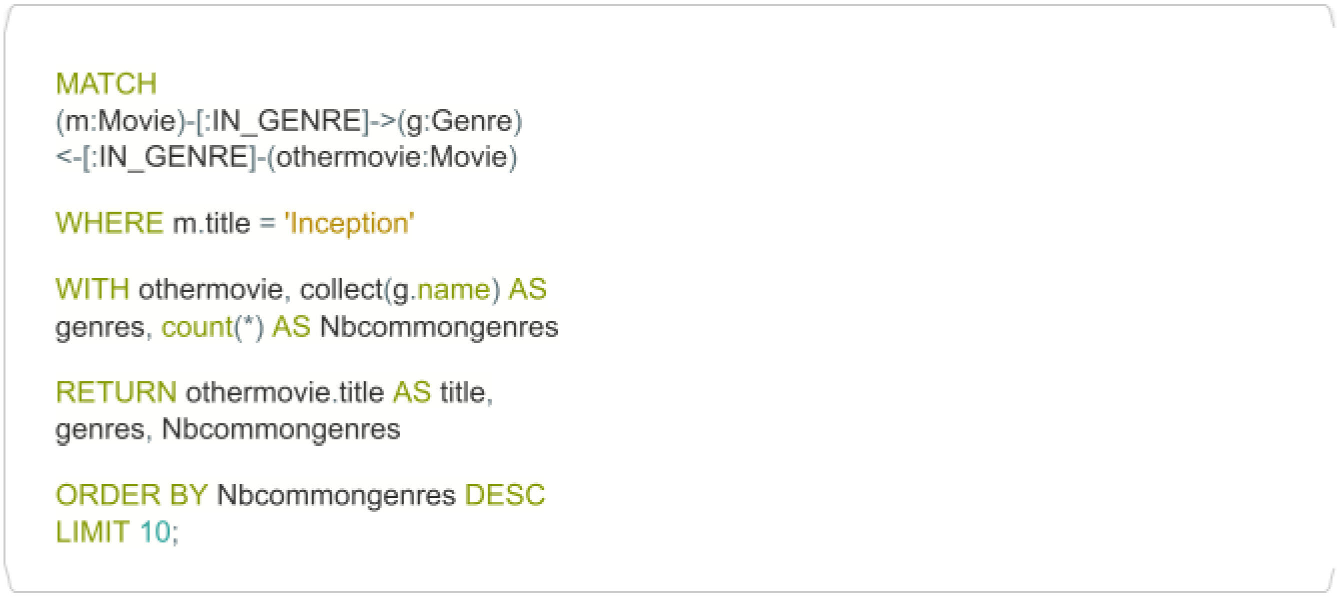
Content-Based Filtering

The goal of content-based filtering is to find similar items, using attributes (or traits) of the item. Using our movie data, one way we could define similarlity is movies that have common genres.



Similarity Based on Common Genres

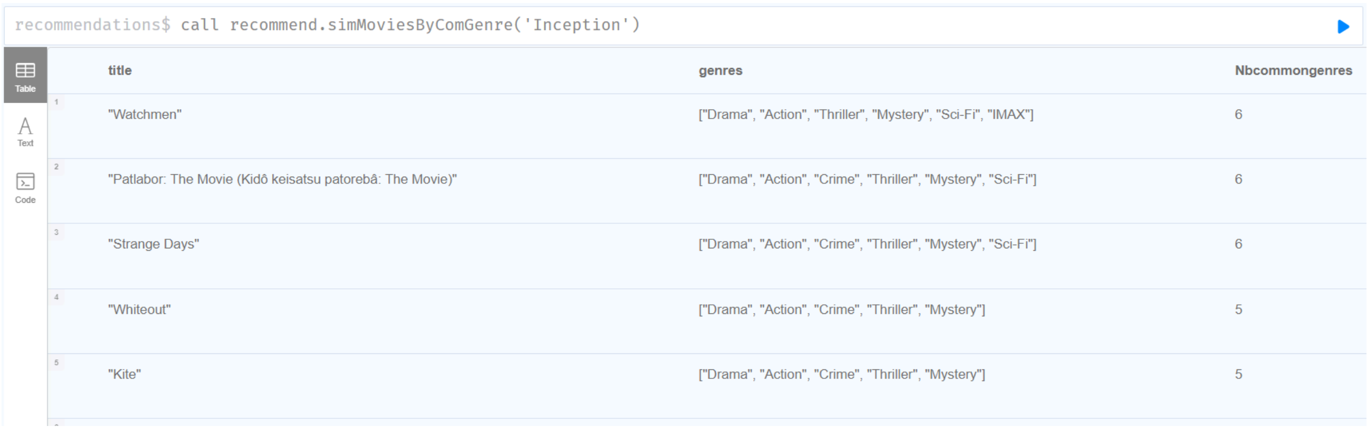
1/ Find movies most similar to Inception based on shared genres



```
MATCH
(m:Movie)-[:IN_GENRE]->(g:Genre)
<-[:IN_GENRE]-(othermovie:Movie)

WHERE m.title = 'Inception'

WITH othermovie, collect(g.name) AS
genres, count(*) AS Nbcommongenres

RETURN othermovie.title AS title,
genres, Nbcommongenres

ORDER BY Nbcommongenres DESC
LIMIT 10;
```

2/ After you completed previous request and tested it, create your own User defined procedure

to do the same work.



```
recommendations$ call recommend.simMoviesByComGenre('Inception')
```

	title	genres	Nbcommongenres
1	"Watchmen"	["Drama", "Action", "Thriller", "Mystery", "Sci-Fi", "IMAX"]	6
2	"Patlabor: The Movie (Kidô keisatsu patorebâ: The Movie)"	["Drama", "Action", "Crime", "Thriller", "Mystery", "Sci-Fi"]	6
3	"Strange Days"	["Drama", "Action", "Crime", "Thriller", "Mystery", "Sci-Fi"]	6
4	"Whiteout"	["Drama", "Action", "Crime", "Thriller", "Mystery"]	5
5	"Kite"	["Drama", "Action", "Crime", "Thriller", "Mystery"]	5

Personalized Recommendations Based on Genres

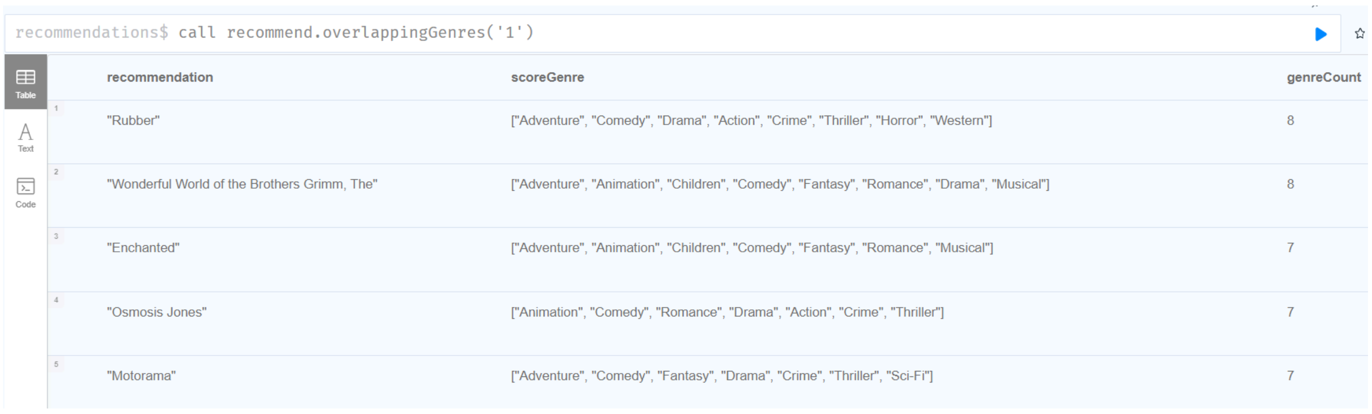
If we know what movies a user has watched, we can use this information to recommend similar movies:

1/ Recommend movies similar to those the user has already watched

```
MATCH (u:User {userId:
'1'})-[:RATED]->(m:Movie)-[:IN_GENRE]->(g:Genre)<-[:IN_GENRE]-(similar:Movie)
WHERE NOT (u)-[:RATED]->(similar)
WITH similar, COLLECT(DISTINCT g.name) AS genres
RETURN similar.title AS recommendedMovie, genres, SIZE(genres) AS genreCount
ORDER BY genreCount DESC
LIMIT 10;
```

2/ After you completed previous request and tested it, create your own User defined procedure

to do the same work.



```
recommendations$ call recommend.overlappingGenres('1')
```

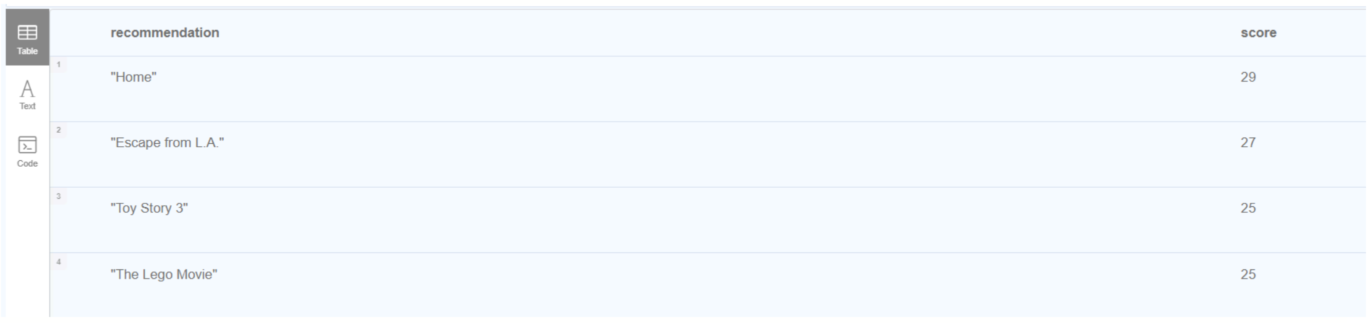
	recommendation	scoreGenre	genreCount
1	"Rubber"	["Adventure", "Comedy", "Drama", "Action", "Crime", "Thriller", "Horror", "Western"]	8
2	"Wonderful World of the Brothers Grimm, The"	["Adventure", "Animation", "Children", "Comedy", "Fantasy", "Romance", "Drama", "Musical"]	8
3	"Enchanted"	["Adventure", "Animation", "Children", "Comedy", "Fantasy", "Romance", "Musical"]	7
4	"Osmosis Jones"	["Animation", "Comedy", "Romance", "Drama", "Action", "Crime", "Thriller"]	7
5	"Motorama"	["Adventure", "Comedy", "Fantasy", "Drama", "Crime", "Thriller", "Sci-Fi"]	7

Weighted Content Algorithm

Of course there are many more traits in addition to just genre that we can consider to compute similarity, such as actors and directors. Let’s use a weighted sum to score the recommendations based on the number of actors (3x), genres (5x) and directors (4x) they have in common to boost the score:

Compute a weighted sum based on the number and types of overlapping traits

```
MATCH (u:User {userId:
'1'})-[:RATED]->(m:Movie)-[:IN_GENRE]->(g:Genre)<-[:IN_GENRE]-(othermovie:Movie)
WITH m, othermovie, count(*) AS genres
OPTIONAL MATCH (m)<-[:ACTED_IN]-(a)-[:ACTED_IN]->(othermovie)
WITH m, othermovie, genres, count(a) AS actors
OPTIONAL MATCH (m)<-[:DIRECTED]-(d)-[:DIRECTED]->(othermovie)
WITH m, othermovie, genres, actors, count(d) AS directors
RETURN othermovie.title AS recommendation, (5*genres)+(3*actors)+(4*directors) AS score
ORDER BY score DESC
LIMIT 10
```



recommendation	score
"Home"	29
"Escape from L.A."	27
"Toy Story 3"	25
"The Lego Movie"	25

Content-Based Similarity Metrics

So far we've used the number of common traits as a way to score the relevance of our recommendations. Let's now consider a more robust way to quantify similarity, using a similarity metric. Similarity metrics are an important component used in generating personalized recommendations that allow us to quantify how similar two items (or as we'll see later, how similar two users preferences) are.

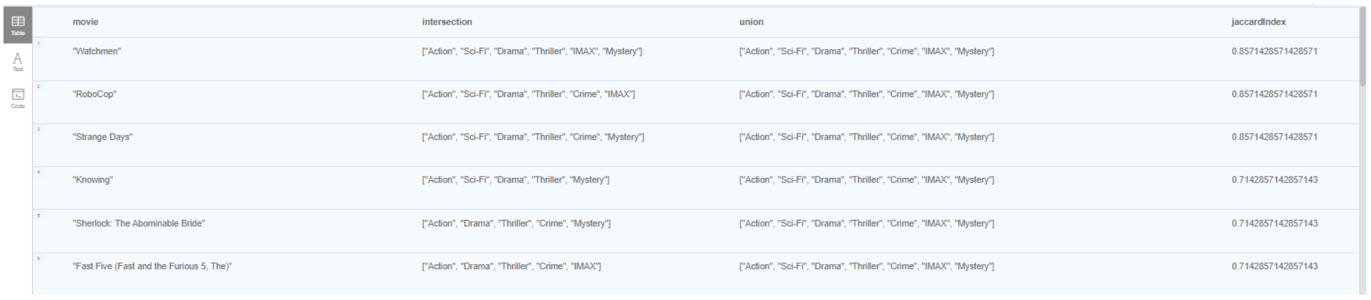
Jaccard Index

The Jaccard index is a number between 0 and 1 that indicates how similar two sets are. The Jaccard index of two identical sets is 1. If two sets do not have a common element, then the Jaccard index is
0. The Jaccard is calculated by dividing the size of the intersection of two sets by the union of the two sets.

We can calculate the Jaccard index for sets of movie genres to determine how similar two movies are.

What movies are most similar to Inception based on Jaccard similarity of genres?

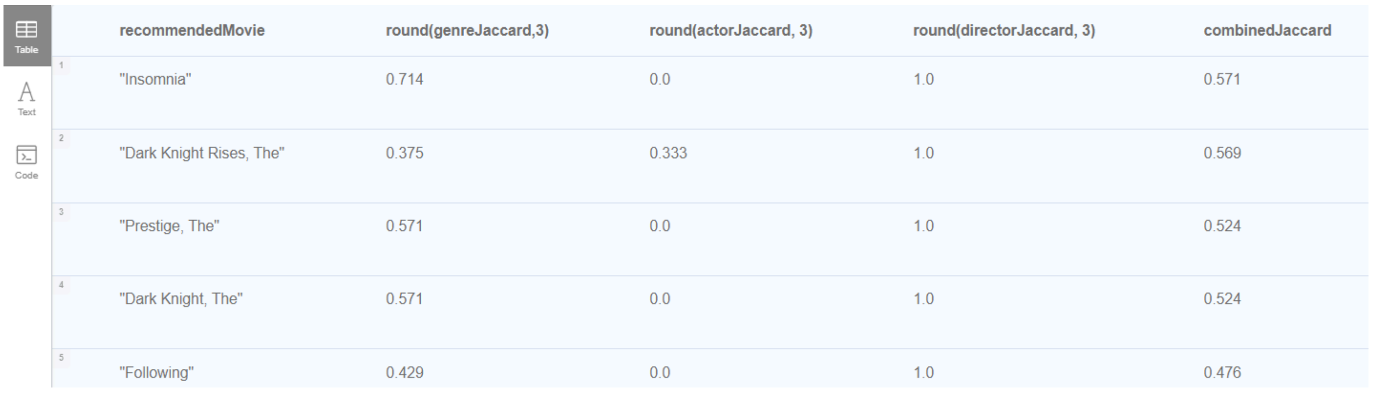
```
MATCH (inception:Movie {title: 'Inception'})-[:IN_GENRE]->(g:Genre)

WITH collect(g.name) AS inceptionGenres

MATCH (m:Movie)-[:IN_GENRE]->(g:Genre)

WHERE m.title <> 'Inception'

WITH m.title AS movie, collect(g.name) AS movieGenres, inceptionGenres

RETURN movie,

    apoc.coll.intersection(inceptionGenres, movieGenres) AS intersection,

    apoc.coll.union(inceptionGenres, movieGenres) AS union,

    toFloat(size(apoc.coll.intersection(inceptionGenres, movieGenres))) /
size(apoc.coll.union(inceptionGenres, movieGenres)) AS jaccardIndex

ORDER BY jaccardIndex DESC

LIMIT 25;
```



	movie	intersection	union	jaccardIndex
1	"Watchmen"	["Action", "Sci-Fi", "Drama", "Thriller", "IMAX", "Mystery"]	["Action", "Sci-Fi", "Drama", "Thriller", "Crime", "IMAX", "Mystery"]	0.8571428571428571
2	"RoboCop"	["Action", "Sci-Fi", "Drama", "Thriller", "Crime", "IMAX"]	["Action", "Sci-Fi", "Drama", "Thriller", "Crime", "IMAX", "Mystery"]	0.8571428571428571
3	"Strange Days"	["Action", "Sci-Fi", "Drama", "Thriller", "Crime", "Mystery"]	["Action", "Sci-Fi", "Drama", "Thriller", "Crime", "IMAX", "Mystery"]	0.8571428571428571
4	"Knowing"	["Action", "Sci-Fi", "Drama", "Thriller", "Mystery"]	["Action", "Sci-Fi", "Drama", "Thriller", "Crime", "IMAX", "Mystery"]	0.7142857142857143
5	"Sherlock: The Abominable Bride"	["Action", "Drama", "Thriller", "Crime", "Mystery"]	["Action", "Sci-Fi", "Drama", "Thriller", "Crime", "IMAX", "Mystery"]	0.7142857142857143
6	"Fast Five (Fast and the Furious 5, The)"	["Action", "Drama", "Thriller", "Crime", "IMAX"]	["Action", "Sci-Fi", "Drama", "Thriller", "Crime", "IMAX", "Mystery"]	0.7142857142857143

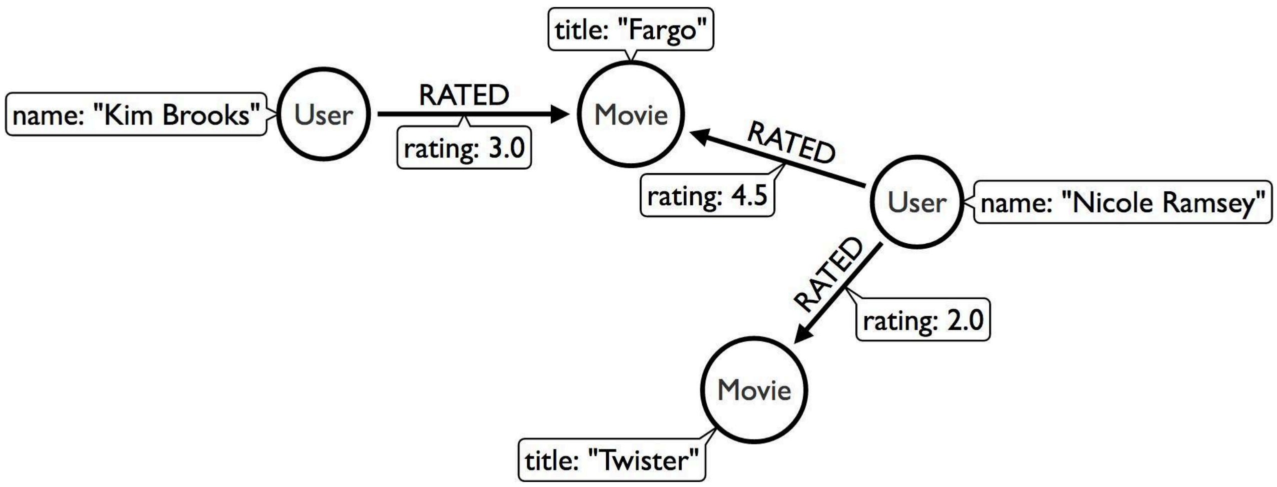
Apply this same approach to all "traits" of the movie (genre, actors, directors, etc.):

```
MATCH (m:Movie {title: 'Inception'})
OPTIONAL MATCH (m)-[:IN_GENRE]->(g:Genre)
WITH m, collect(g.name) AS inceptionGenres
OPTIONAL MATCH (m)<-[:ACTED_IN]-(a:Actor)
WITH m, inceptionGenres, collect(a.name) AS inceptionActors
OPTIONAL MATCH (m)<-[:DIRECTED]-(d:Director)
WITH inceptionGenres, inceptionActors, collect(d.name) AS inceptionDirectors

MATCH (other:Movie)
WHERE other.title <> 'Inception'
OPTIONAL MATCH (other)-[:IN_GENRE]->(og:Genre)
WITH other, inceptionGenres, inceptionActors, inceptionDirectors, collect(og.name) AS
movieGenres
OPTIONAL MATCH (other)<-[:ACTED_IN]-(oa:Actor)
WITH other, inceptionGenres, inceptionActors, inceptionDirectors, movieGenres, collect(oa.name)
AS movieActors
OPTIONAL MATCH (other)<-[:DIRECTED]-(od:Director)
WITH other.title AS recommendedMovie, inceptionGenres, inceptionActors, inceptionDirectors,
movieGenres, movieActors, collect(od.name) AS movieDirectors

// Calculate Jaccard similarities
WITH recommendedMovie,
   toFloat(size(apoc.coll.intersection(inceptionGenres, movieGenres))) /
size(apoc.coll.union(inceptionGenres, movieGenres)) AS genreJaccard,
   toFloat(size(apoc.coll.intersection(inceptionActors, movieActors))) /
size(apoc.coll.union(inceptionActors, movieActors)) AS actorJaccard,
   toFloat(size(apoc.coll.intersection(inceptionDirectors, movieDirectors))) /
size(apoc.coll.union(inceptionDirectors, movieDirectors)) AS directorJaccard

// Combine Jaccard similarities and return results
RETURN recommendedMovie,
      round(genreJaccard,3),
      round(actorJaccard, 3),
      round(directorJaccard, 3),
      round((genreJaccard + actorJaccard + directorJaccard) / 3, 3) AS combinedJaccard
ORDER BY combinedJaccard DESC
LIMIT 5;
```



	recommendedMovie	round(genreJaccard,3)	round(actorJaccard, 3)	round(directorJaccard, 3)	combinedJaccard
1	"Insomnia"	0.714	0.0	1.0	0.571
2	"Dark Knight Rises, The"	0.375	0.333	1.0	0.569
3	"Prestige, The"	0.571	0.0	1.0	0.524
4	"Dark Knight, The"	0.571	0.0	1.0	0.524
5	"Following"	0.429	0.0	1.0	0.476

Collaborative Filtering – Leveraging Movie Ratings



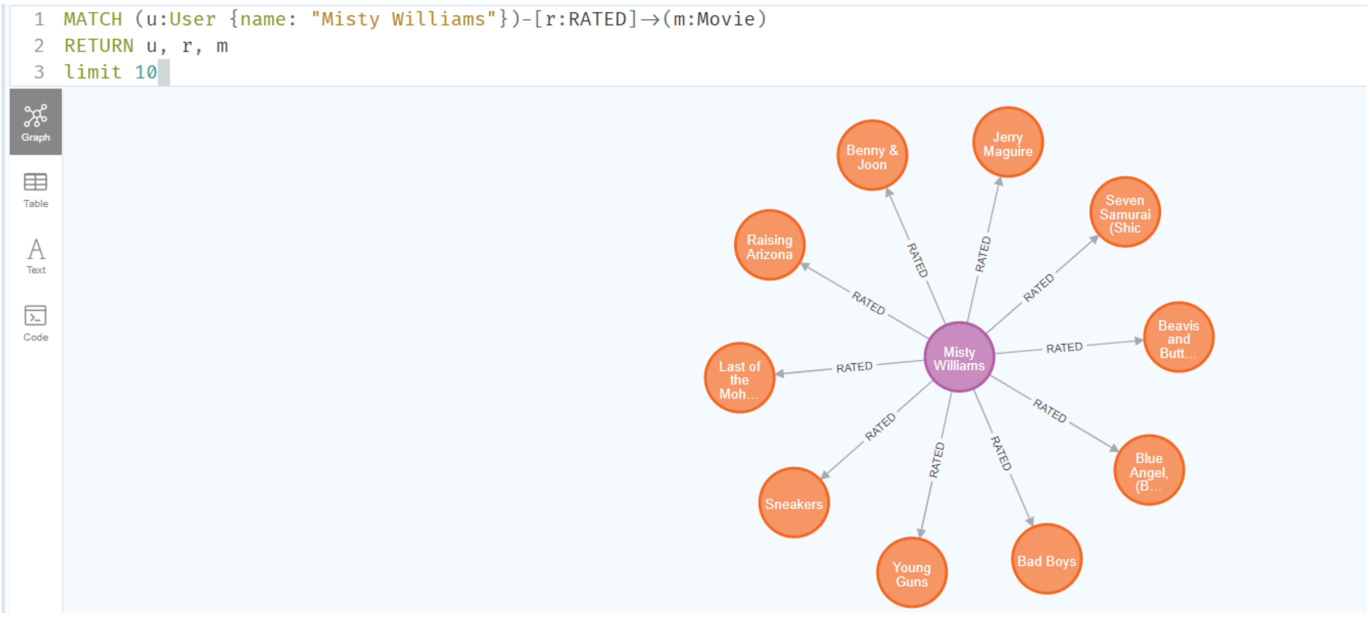
Notice that we have user-movie ratings in our graph. The collaborative filtering approach is going to make use of this information to find relevant recommendations.

Steps:

1. Find similar users in the network (our peer group).
2. Assuming that similar users have similar preferences, what are the movies those similar users like?

Show all ratings by Misty Williams

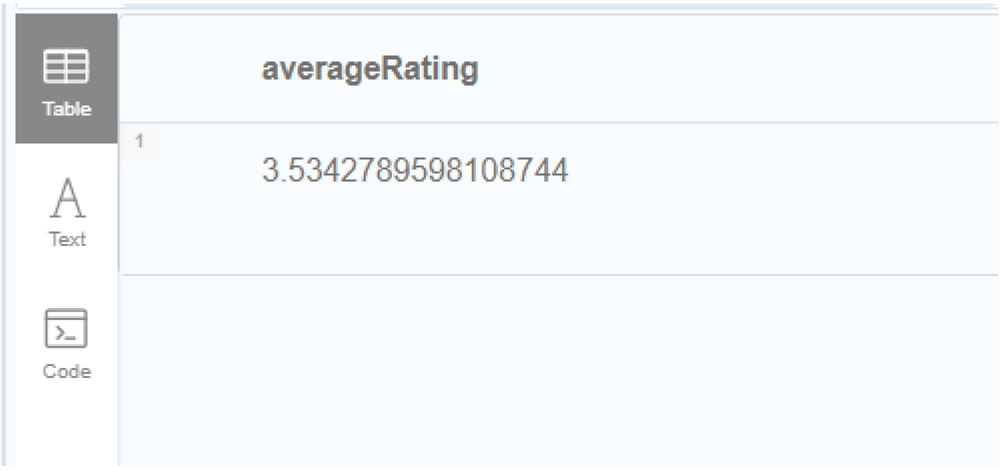
```
MATCH (u:User {name: "Misty Williams"})-[r:RATED]->(m:Movie)

RETURN u, r, m
```



```
MATCH (u:User {name: "Misty Williams"})-[r:RATED]→(m:Movie)
RETURN u, r, m
limit 10
```

Find Misty's average rating

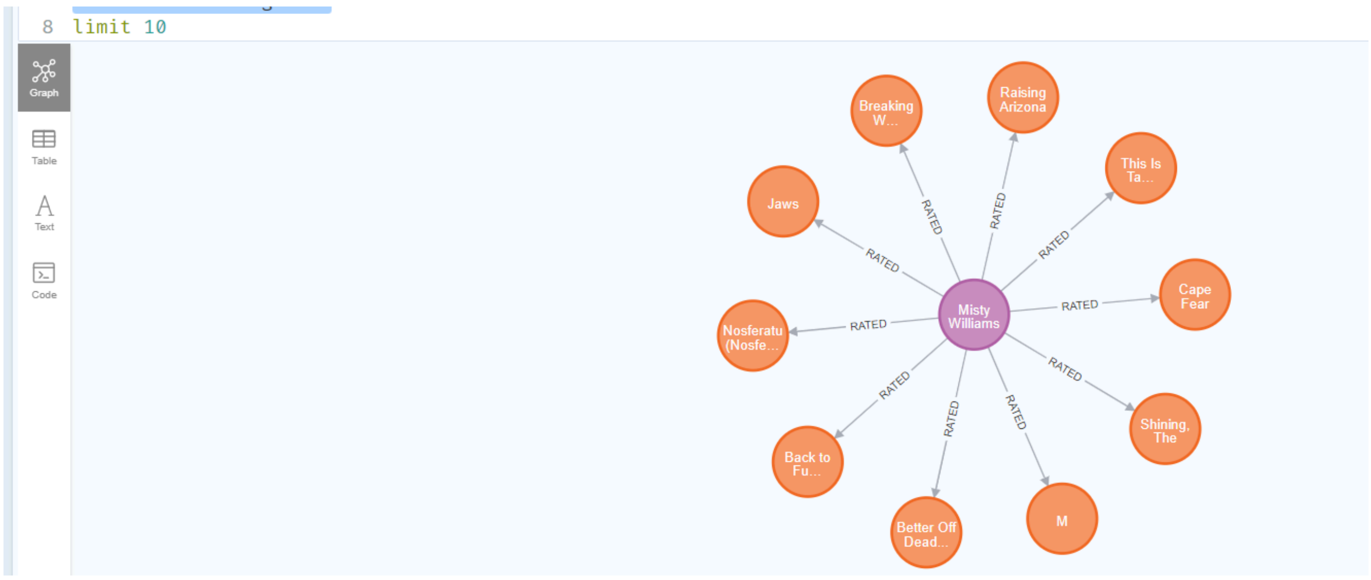
```
MATCH (u:User {name: 'Misty
Williams'})-[r:RATED]->(m:Movie)

RETURN AVG(r.rating) AS
averageRating;
```



```
averageRating
1  3.5342789598108744
```

What are the movies that Misty liked more than average?

```
MATCH (u:User {name: "Misty
Williams"})-[r:RATED]->(m:Movie)

WITH u, AVG(r.rating) AS averageRating


MATCH (u)-[r:RATED]->(m:Movie)

WHERE r.rating > averageRating

RETURN u, r, m

ORDER BY r.rating DESC
```


```
limit 10
```

Collaborative Filtering – The Wisdom of Crowds

Simple Collaborative Filtering

Here we just use the fact that someone has rated a movie, not their actual rating to demonstrate the structure of finding the peers. Then we look at what else the peers rated, that the user has not rated themselves yet.

```
MATCH (u:User {name: 'Cynthia
      Freeman'})-[:RATED]->
      (:Movie)<-[:RATED]-(peer:User)
MATCH (peer)-[:RATED]->(rec:Movie)
WHERE NOT EXISTS { (u)-[:RATED]->(rec) }
RETURN rec.title, rec.year,
rec.plot LIMIT 25
```

Of course this is just a simple appraoch, there are many problems with this query, such as not normalizing based on popularity or not taking ratings into consideration. We’ll do that next, looking at movies being rated similarly, and then picking highly rated movies and using their rating and frequency to sort the results.

```
 MATCH (u:User {name: 'Cynthia
       Freeman'})-[r1:RATED]->
       (:Movie)<-[r2:RATED]-(peer:User)
```

```
WHERE abs(r1.rating-r2.rating) < 2 // similarly
rated WITH distinct u, peer
MATCH
(peer)-[r3:RATED]->(rec:Movie)
WHERE r3.rating > 3
  AND NOT EXISTS { (u)-[:RATED]->(rec) }
WITH rec, count(*) as freq, avg(r3.rating) as
rating RETURN rec.title, rec.year, rating, freq,
rec.plot ORDER BY rating DESC, freq DESC
LIMIT 25
```

In the next section, we will see how we can improve this approach using the **kNN method**.

Only Consider Genres Liked by the User

Many recommender systems are a blend of collaborative filtering and content-based approaches:

For a particular user, what genres have a higher-than-average rating? Use this to score similar movies

```
  // 1 compute mean rating

MATCH (u:User {name: 'Misty Williams'})-[r:RATED]->(m:Movie)

WITH u, AVG(r.rating) AS averageRating



// 2 find genres with higher than average rating and their number of rated movies

MATCH (u)-[r:RATED]->(m:Movie)-[:IN_GENRE]->(g:Genre)

WHERE r.rating > averageRating

WITH u, g, AVG(r.rating) AS avgGenreRating, COUNT(m) AS ratedMovies



// 3 find movies in those genres, that have not been watched yet

MATCH (g)<-[:IN_GENRE]-(recommended:Movie)

WHERE NOT EXISTS((u)-[:RATED]->(recommended))

RETURN recommended.title AS recommendedMovie, g.name AS genre, avgGenreRating,
ratedMovies

ORDER BY avgGenreRating DESC, ratedMovies DESC

LIMIT 10
```

Collaborative Filtering – Similarity Metrics

We use similarity metrics to quantify how similar two users or two items are. We've already seen Jaccard similarity used in the context of content-based filtering. Now, we'll see how similarity metrics are used with collaborative filtering.

Cosine Distance

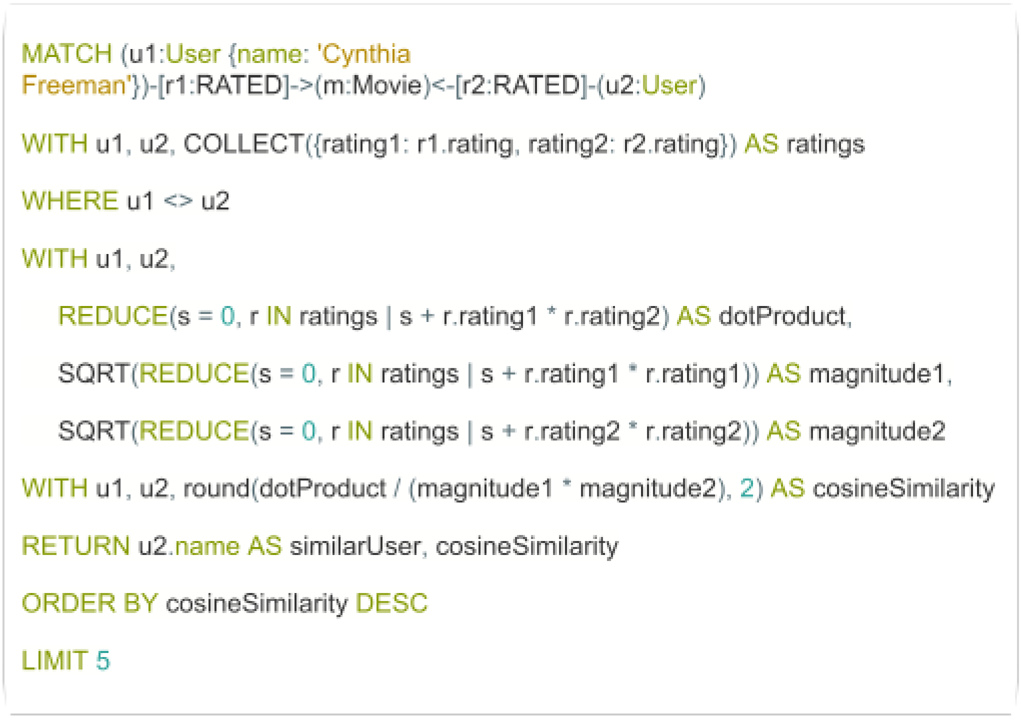
Jaccard similarity was useful for comparing movies and is essentially comparing two sets (groups of genres, actors, directors, etc.). However, with movie ratings each relationship has a **weight** that we can consider as well.

Cosine-Similarity

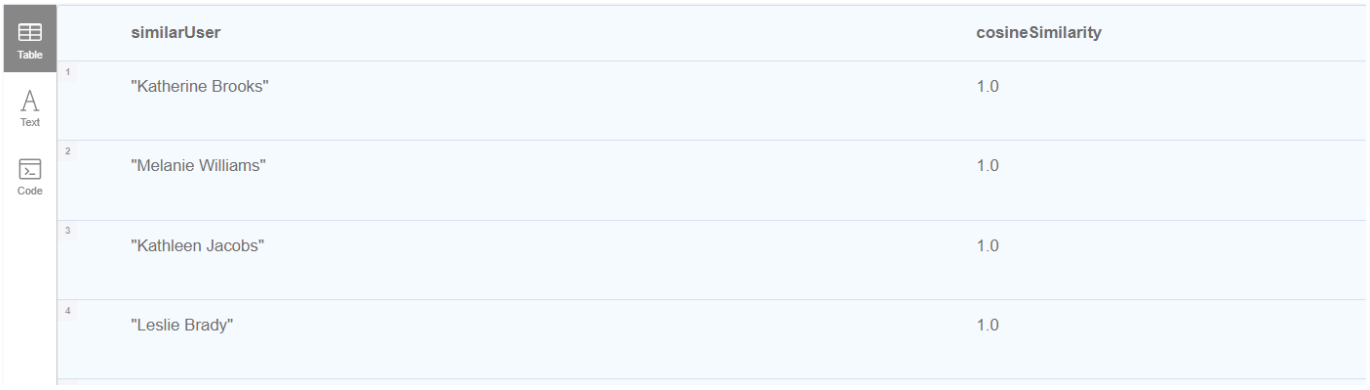
$$similarity(A, B) = \frac{A \cdot B}{\|A\| \times \|B\|} = \frac{\sum_{i=1}^{n} A_i \times B_i}{\sqrt{\sum_{i=1}^{n} A_i^2} \times \sqrt{\sum_{i=1}^{n} B_i^2}}$$

The cosine similarity of two users will tell us how similar two users' preferences for movies are. Users with a high cosine similarity will have similar preferences.

Find the users with the most similar preferences to Cynthia Freeman, according to cosine similarity



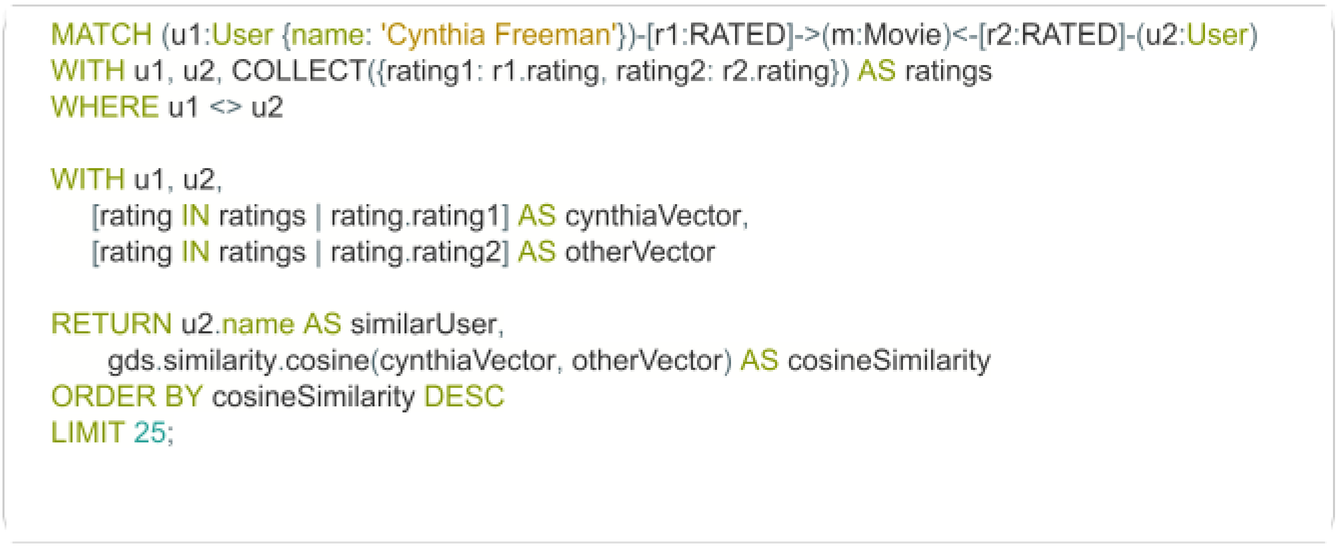
```
MATCH (u1:User {name: 'Cynthia
Freeman'})-[r1:RATED]->(m:Movie)<-[r2:RATED]-(u2:User)

WITH u1, u2, COLLECT({rating1: r1.rating, rating2: r2.rating}) AS ratings

WHERE u1 <> u2

WITH u1, u2,

    REDUCE(s = 0, r IN ratings | s + r.rating1 * r.rating2) AS dotProduct,

    SQRT(REDUCE(s = 0, r IN ratings | s + r.rating1 * r.rating1)) AS magnitude1,

    SQRT(REDUCE(s = 0, r IN ratings | s + r.rating2 * r.rating2)) AS magnitude2

WITH u1, u2, round(dotProduct / (magnitude1 * magnitude2), 2) AS cosineSimilarity

RETURN u2.name AS similarUser, cosineSimilarity

ORDER BY cosineSimilarity DESC

LIMIT 5
```



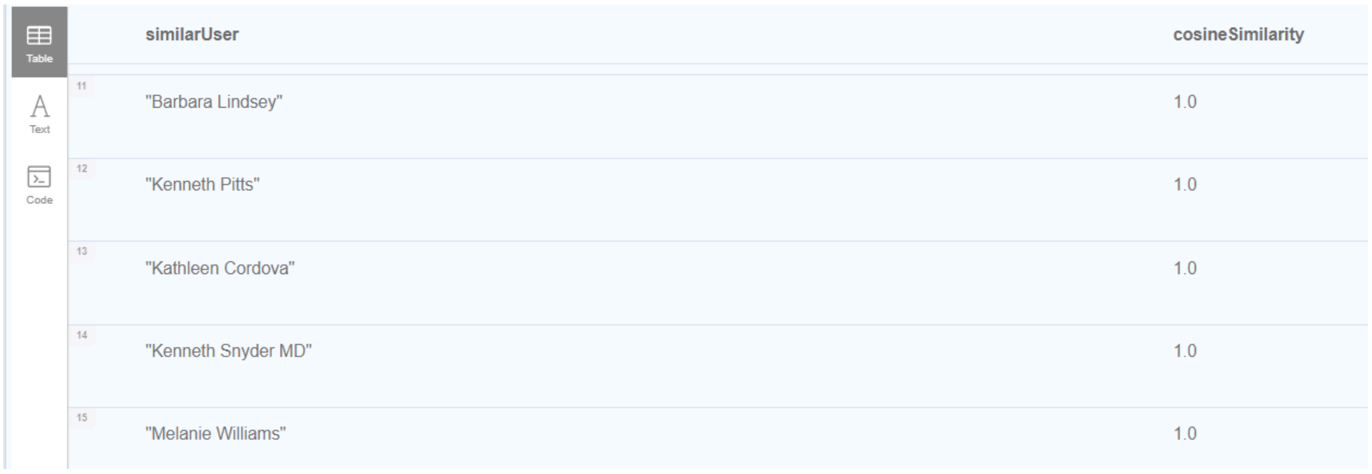
	similarUser	cosineSimilarity
1	"Katherine Brooks"	1.0
2	"Melanie Williams"	1.0
3	"Kathleen Jacobs"	1.0
4	"Leslie Brady"	1.0

We can also compute this measure using the Cosine Similarity algorithm in the Neo4j Graph Data Science Library.

Find the users with the most similar preferences to Cynthia Freeman, according to cosine similarity function



```
MATCH (u1:User {name: 'Cynthia Freeman'})-[r1:RATED]->(m:Movie)<-[r2:RATED]-(u2:User)
WITH u1, u2, COLLECT({rating1: r1.rating, rating2: r2.rating}) AS ratings
WHERE u1 <> u2

WITH u1, u2,
    [rating IN ratings | rating.rating1] AS cynthiaVector,
    [rating IN ratings | rating.rating2] AS otherVector

RETURN u2.name AS similarUser,
      gds.similarity.cosine(cynthiaVector, otherVector) AS cosineSimilarity
ORDER BY cosineSimilarity DESC
LIMIT 25;
```



	similarUser	cosineSimilarity
11	"Barbara Lindsey"	1.0
12	"Kenneth Pitts"	1.0
13	"Kathleen Cordova"	1.0
14	"Kenneth Snyder MD"	1.0
15	"Melanie Williams"	1.0

Collaborative Filtering – Similarity Metrics

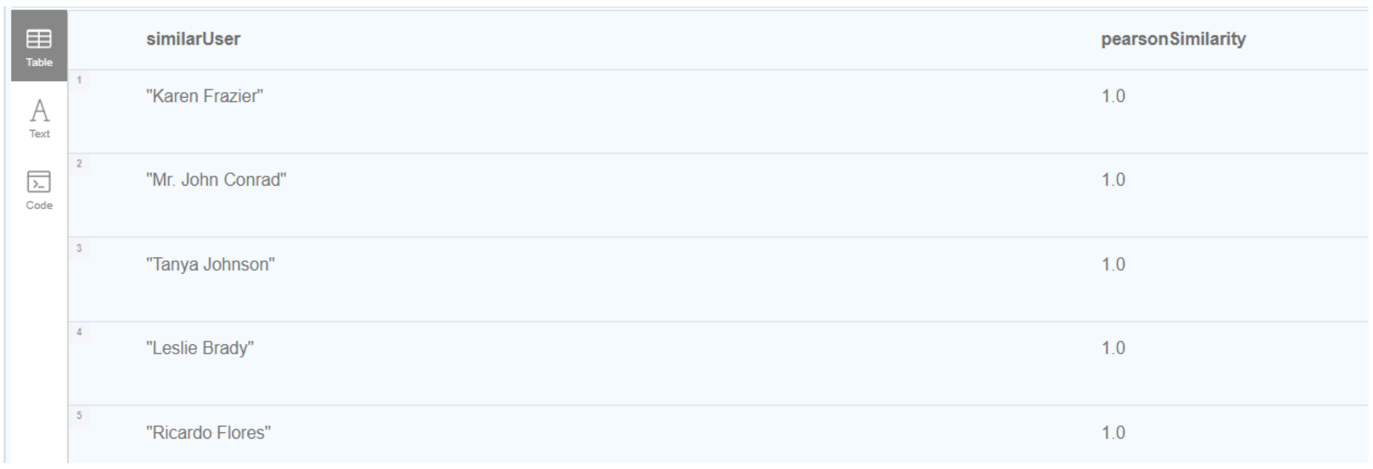
Pearson Similarity

Pearson similarity, or Pearson correlation, is another similarity metric we can use. This is particularly well-suited for product recommendations because it takes into account the fact that different users will have different **mean ratings**: on average some users will tend to give higher ratings than others. Since Pearson similarity considers differences about the mean, this metric will account for these discrepancies.

$$\frac{\sum_{i=1}^{n}(A_i - \bar{A})(B_i - \bar{B})}{\sqrt{\sum_{i=1}^{n}(A_i - \bar{A})^2 \sum_{i=1}^{n}(B_i - \bar{B})^2}}$$

Find users most similar to Cynthia Freeman, according to Pearson similarity

```
MATCH (u1)-[r1:RATED]->(m:Movie)<-[r2:RATED]-(u2)
WITH u1, u2, avgRating1, avgRating2,
    collect((r1.rating - avgRating1) * (r2.rating - avgRating2)) AS products,
    collect((r1.rating - avgRating1)^2) AS squares1,
    collect((r2.rating - avgRating2)^2) AS squares2
WITH u1, u2,
    reduce(sumProd = 0.0, p IN products | sumProd + p) AS sumProduct,
    reduce(sumSquare1 = 0.0, s IN squares1 | sumSquare1 + s) AS sumSquare1,
    reduce(sumSquare2 = 0.0, s IN squares2 | sumSquare2 + s) AS sumSquare2

// Calculer la similarité de Pearson
WITH u1, u2,
    CASE
        WHEN sqrt(sumSquare1) * sqrt(sumSquare2) = 0 THEN 0.0
        ELSE round(sumProduct / (sqrt(sumSquare1) * sqrt(sumSquare2)), 2)
    END AS pearsonSimilarity
RETURN u2.name AS similarUser, pearsonSimilarity
ORDER BY pearsonSimilarity DESC
LIMIT 10
```

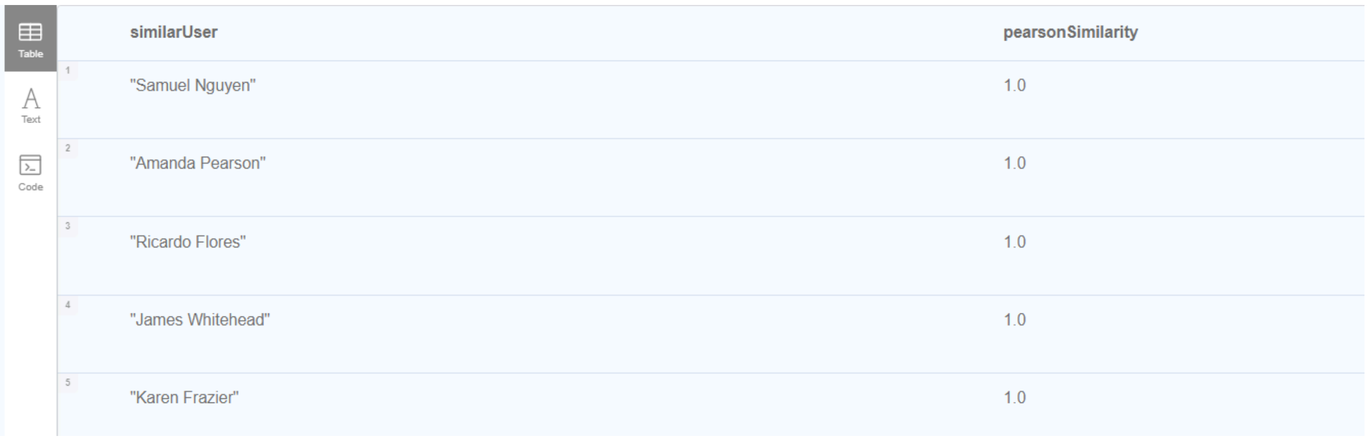


similarUser	pearsonSimilarity
"Karen Frazier"	1.0
"Mr. John Conrad"	1.0
"Tanya Johnson"	1.0
"Leslie Brady"	1.0
"Ricardo Flores"	1.0

We can also compute this measure using the Pearson Similarity algorithm in the Neo4j Graph Data Science Library.

Find users most similar to Cynthia Freeman, according to the Pearson similarity function

```
MATCH (u1:User {name: 'Cynthia Freeman'})-[r1:RATED]->(m:Movie)<-[r2:RATED]-(u2:User)
WITH u1, u2, COLLECT({rating1: r1.rating, rating2: r2.rating}) AS ratings
WHERE u1 <> u2

WITH u1, u2,
   [rating IN ratings | rating.rating1] AS cynthiaVector,
   [rating IN ratings | rating.rating2] AS otherVector

RETURN u2.name AS similarUser,
     gds.similarity.pearson(cynthiaVector, otherVector) AS pearsonSimilarity
ORDER BY pearsonSimilarity DESC
LIMIT 10;
```



	similarUser	pearsonSimilarity
1	"Samuel Nguyen"	1.0
2	"Amanda Pearson"	1.0
3	"Ricardo Flores"	1.0
4	"James Whitehead"	1.0
5	"Karen Frazier"	1.0

Collaborative Filtering – Neighborhood-Based Recommendations

kNN – K-Nearest Neighbors

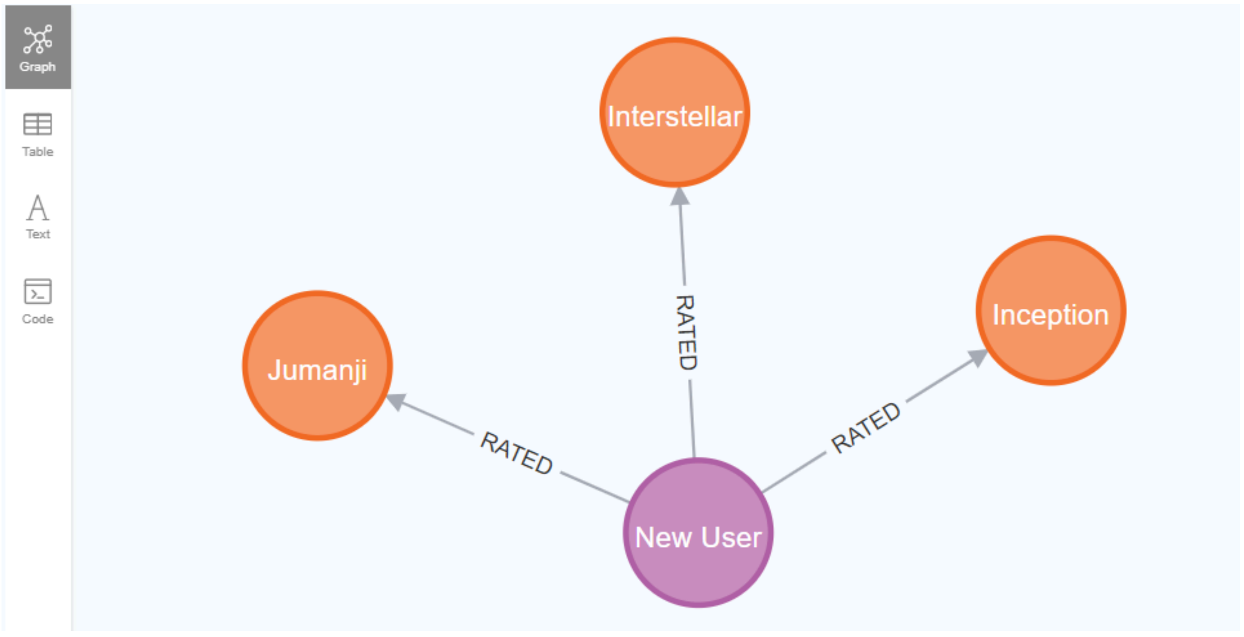
Now that we have a method for finding similar users based on preferences, the next step is to allow each of the **k** most similar users to vote for what items should be recommended.

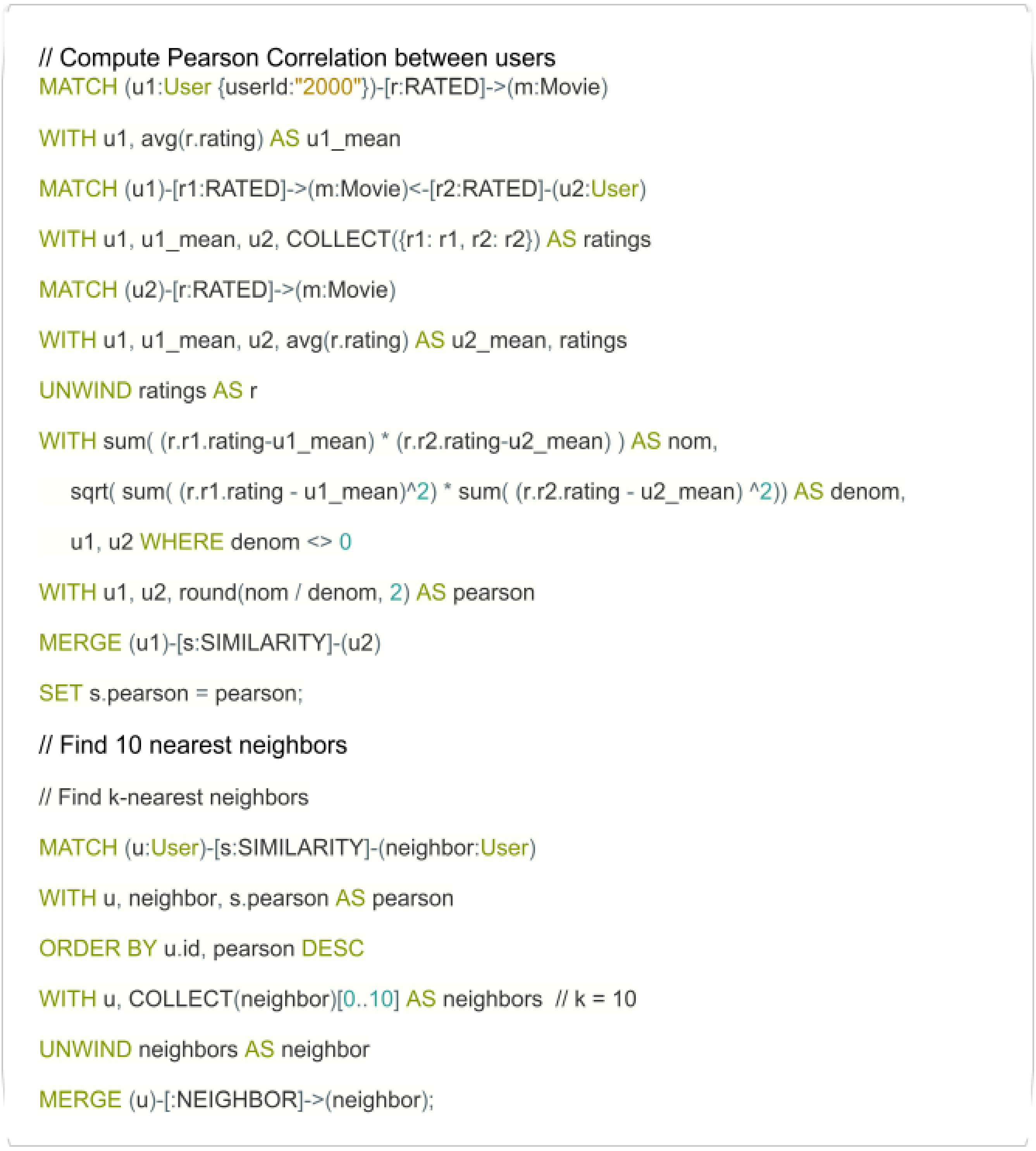
Essentially:

"Who are the 10 users with tastes in movies most similar to mine? What movies have they rated highly that I haven't seen yet?"

kNN movie recommendation using Pearson similarity

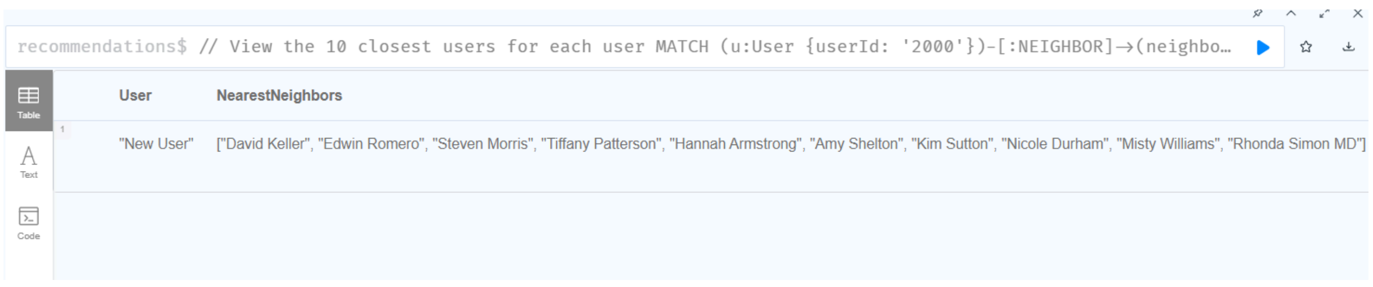
```
// Create our new user
CREATE (:User {userId: 2000, name: 'New User'});
----------------
MATCH (u:User {userId: 2000}), (m1:Movie {title: Inception})
CREATE (u)-[:RATED {rating: 4}]->(m1);
----------------
MATCH (u:User {userId: 2000}), (m2:Movie {title: Interstellar})
CREATE (u)-[:RATED {rating: 5}]->(m2);
----------------
MATCH (u:User {userId: 2000}), (m3:Movie {title: Jumanji})
CREATE (u)-[:RATED {rating: 3}]->(m3);
```





```
// Compute Pearson Correlation between users
MATCH (u1:User {userId:"2000"})-[r:RATED]->(m:Movie)

WITH u1, avg(r.rating) AS u1_mean

MATCH (u1)-[r1:RATED]->(m:Movie)<-[r2:RATED]-(u2:User)

WITH u1, u1_mean, u2, COLLECT({r1: r1, r2: r2}) AS ratings

MATCH (u2)-[r:RATED]->(m:Movie)

WITH u1, u1_mean, u2, avg(r.rating) AS u2_mean, ratings

UNWIND ratings AS r

WITH sum( (r.r1.rating-u1_mean) * (r.r2.rating-u2_mean) ) AS nom,

   sqrt( sum( (r.r1.rating - u1_mean)^2) * sum( (r.r2.rating - u2_mean) ^2)) AS denom,

   u1, u2 WHERE denom <> 0

WITH u1, u2, round(nom / denom, 2) AS pearson

MERGE (u1)-[s:SIMILARITY]-(u2)

SET s.pearson = pearson;

// Find 10 nearest neighbors

// Find k-nearest neighbors

MATCH (u:User)-[s:SIMILARITY]-(neighbor:User)

WITH u, neighbor, s.pearson AS pearson

ORDER BY u.id, pearson DESC

WITH u, COLLECT(neighbor)[0..10] AS neighbors  // k = 10

UNWIND neighbors AS neighbor

MERGE (u)-[:NEIGHBOR]->(neighbor);
```

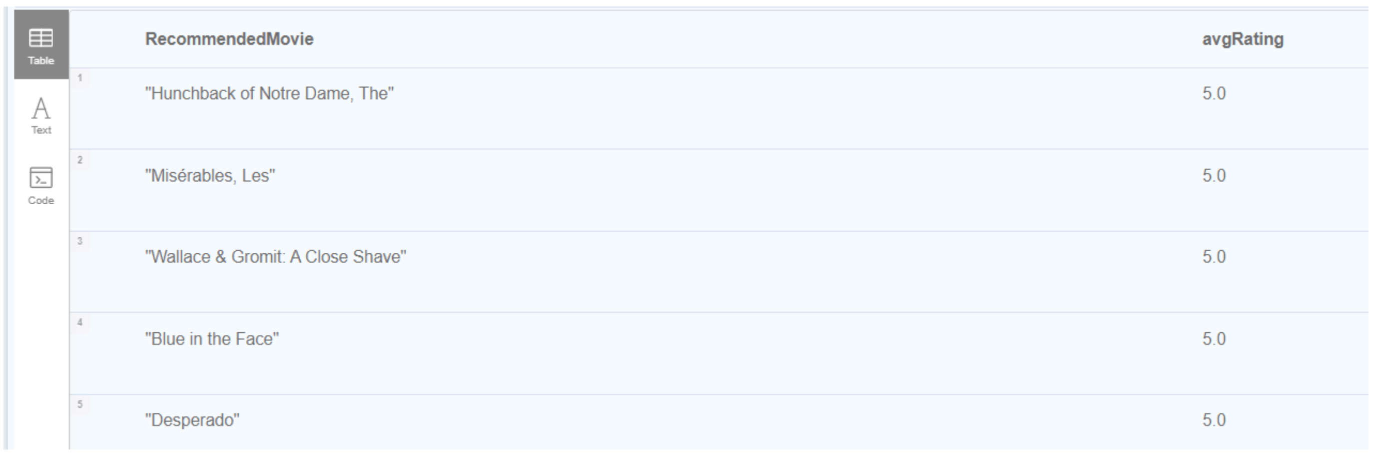
```
// View the 10 closest users for  user with userId = 2000
MATCH (u:User {userId: '2000'})-[:NEIGHBOR]->(neighbor:User)
RETURN u.name AS User, COLLECT(neighbor.name) AS NearestNeighbors
ORDER BY User;
```



```
recommendations$ // View the 10 closest users for each user MATCH (u:User {userId: '2000'})-[:NEIGHBOR]→(neighbo…
```

	User	NearestNeighbors
1	"New User"	["David Keller", "Edwin Romero", "Steven Morris", "Tiffany Patterson", "Hannah Armstrong", "Amy Shelton", "Kim Sutton", "Nicole Durham", "Misty Williams", "Rhonda Simon MD"]

```
//What movies have they rated highly that I haven't seen yet?"
MATCH (u:User {userId: '2000'})-[:NEIGHBOR]->(neighbor)-[r:RATED]->(m:Movie)
WHERE NOT (u)-[:RATED]->(m)
WITH m, AVG(r.rating) AS avgRating
ORDER BY avgRating DESC
RETURN m.title AS RecommendedMovie, avgRating
LIMIT 5;
```



	RecommendedMovie	avgRating
1	"Hunchback of Notre Dame, The"	5.0
2	"Misérables, Les"	5.0
3	"Wallace & Gromit: A Close Shave"	5.0
4	"Blue in the Face"	5.0
5	"Desperado"	5.0

Further Work

Optional Exercises

Extend these queries:

Temporal component

Preferences change over time, use the rating timestamp to consider how more recent ratings might be used to find more relevant recommendations.

Keyword extraction

Enhance the traits available using the plot description. How would you model extracted keywords for movies?

Image recognition using posters

There are several libraries and APIs that offer image recognition and tagging.

Since we have movie poster images for each movie, how could we use these to enhance our recomendations?