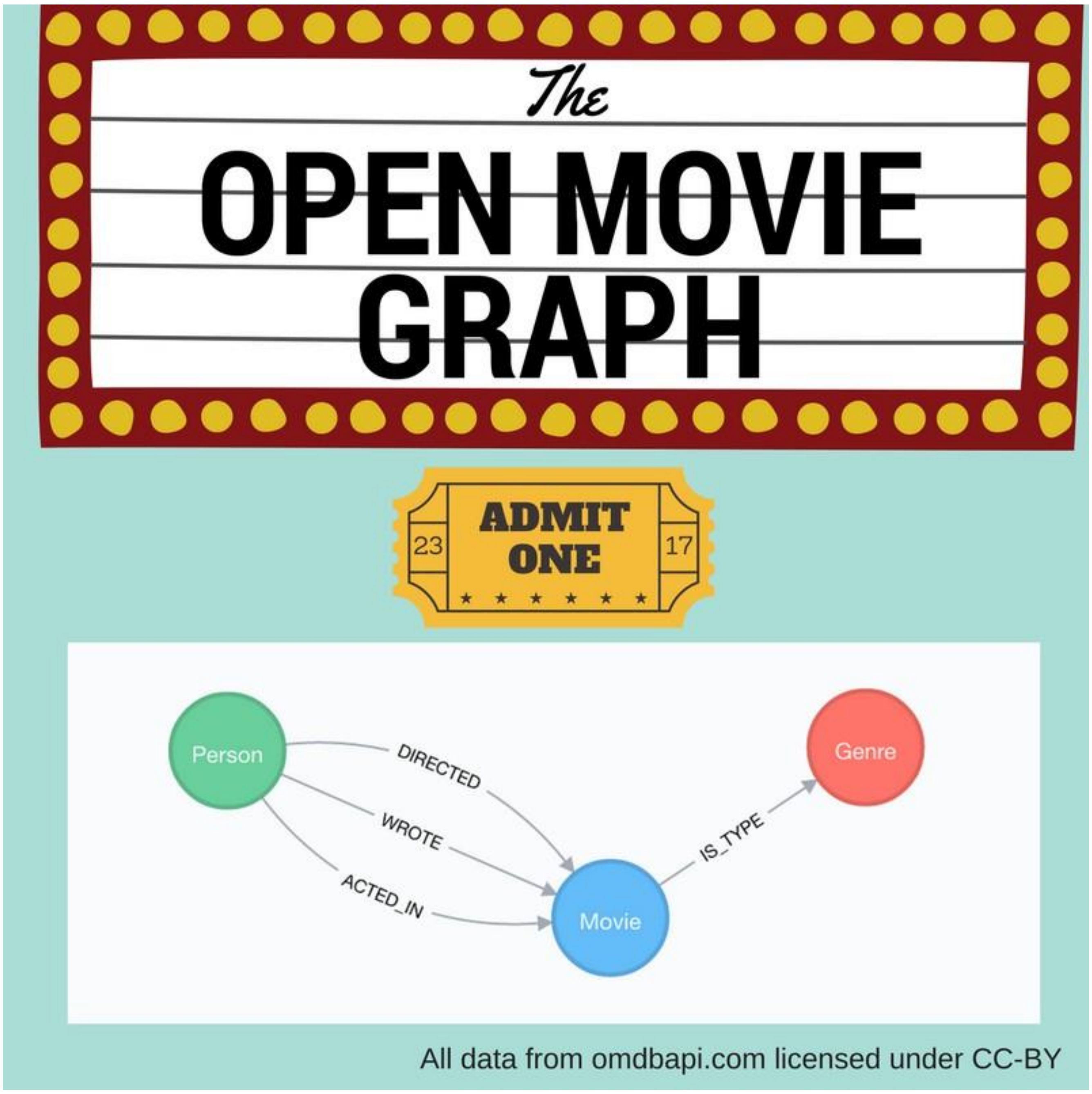
Final Project : Build Recommendations system using movies data

Personalized Product Recommendations with Neo4j



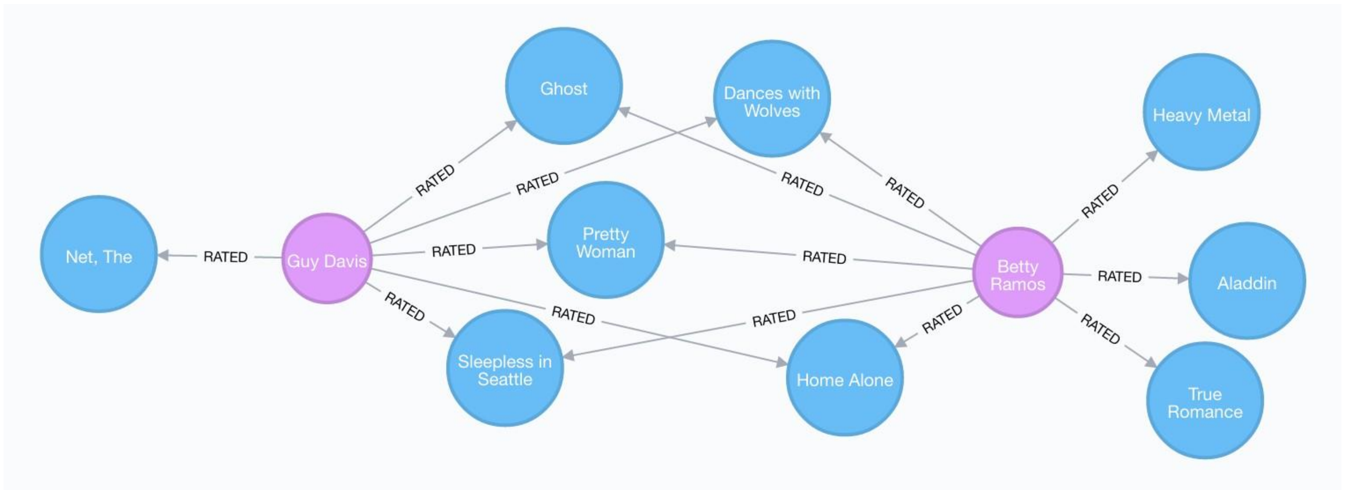
Recommendations

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
  1. Create empty database named recommendations
  2. use recommendations
  3. 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

```
public void ensureDatabaseExists() {
    try (Session session = driver.session()) {
        String createDatabaseQuery = String.format("CREATE DATABASE %s IF NOT EXISTS;",
database);
        session.writeTransaction(tx -> {
            tx.run(createDatabaseQuery).consume();
            return null;
        });
        System.out.println("Database '" + database + "' ensured to exist.");
    } catch (Exception e) {
        System.err.println("Error ensuring database exists: " + e.getMessage());
    }
}
public void executeCypherFile(String filePath) {
    try (Session session = driver.session(SessionConfig.forDatabase(database))) {
        String apocQuery = String.format("CALL apoc.cypher.runFile('%s')", filePath);
        session.writeTransaction(tx -> {
            tx.run(apocQuery).consume();
            return null;
        });
        System.out.println("Cypher queries executed successfully on database '" + database + "'
using apoc.cypher.runFile.");
    } catch (Exception e) {
        System.err.println("Error executing Cypher queries: " + e.getMessage());
    }
}


public static void main(String[] args) {
    String uri = "bolt://localhost:7687";
    String user = "neo4j";
    String password = "admin12345"; // replace with your actual password
    String database = "recommendations";
    String filePath = "/all-plain.cypher"; // replace with the actual path to your Cypher file

    Neo4jCypherExecutor executor = new Neo4jCypherExecutor(uri, user, password, database);
    executor.ensureDatabaseExists();
    executor.executeCypherFile(filePath);
    //executor.reviewsCount();
    //executor.recommendItems();
    //executor.separations();
    //executor.collaborativeFiltering();
    //executor.contentBasedFiltering();
    //executor.personalizedRecommendations();
    //executor.recommendationWeightedContent();
    executor.jaccardSimilarityOfGenres();
    executor.close();
}
```

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.

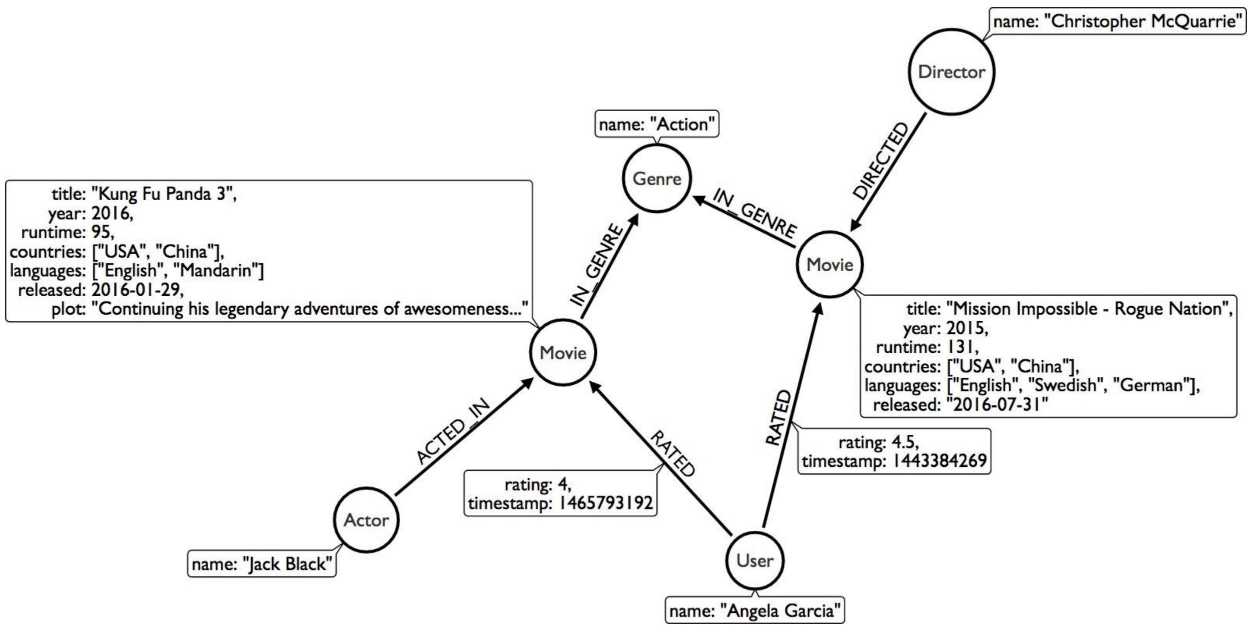
Eliminate Data Silos

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.

Int: Replace ??? by the corrects values

```
MATCH (m:Movie)<-[:RATED]-(u:User)
WHERE toLower(m.title) CONTAINS 'matrix'
WITH m, count(*) AS reviews
RETURN m.title AS movie, reviews
ORDER BY reviews DESC LIMIT 5
```

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

to do the same work.

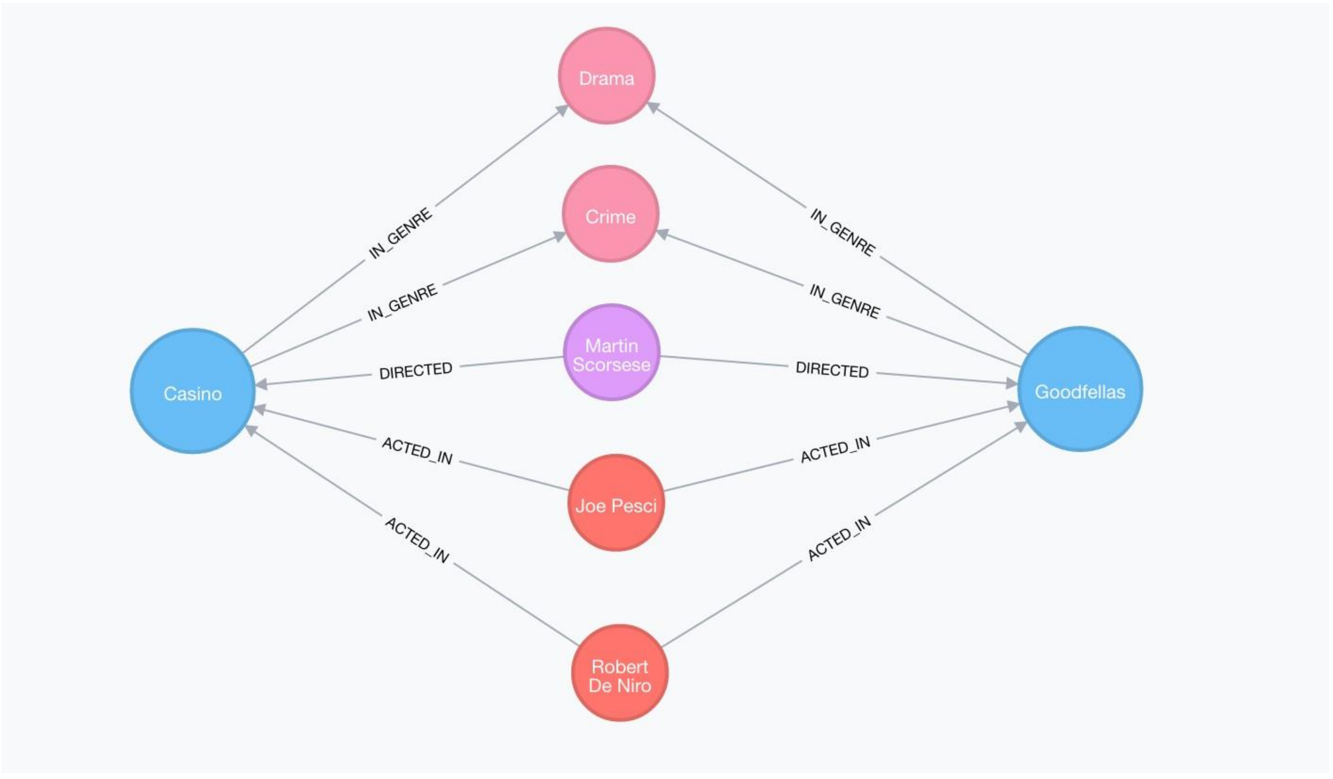
```
public class Question1 {

    @Context
    public Transaction tx;

    public static class EntityContainer {
        public String title;
        public long reviews; // Use long because the database stores counts as long

        public EntityContainer(String title, long reviews) {
            this.title = title;
            this.reviews = reviews;
        }
    }

    @Procedure(name = "recommend.howManyReview", mode = Mode.READ)
    public Stream<EntityContainer> howManyReview() {
        String query = "MATCH (m:Movie)<-[:RATED]-(u:User) " +
                "WHERE toLower(m.title) CONTAINS 'matrix' " +
                "WITH m, count(*) AS reviews " +
                "RETURN m.title AS movie, reviews " +
                "ORDER BY reviews DESC LIMIT 5";


            Result result = tx.execute(query);
            Iterable<Map<String, Object>> iterable = () -> result;
            return StreamSupport.stream(iterable.spliterator(), false)
                    .map(row -> new EntityContainer((String) row.get("movie"), (Long)
row.get("reviews")));

    }
}
```

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 (joe:Actor {name: "Joe Pesci"})-[r1:ACTED_IN]->(m:Movie)
MATCH (robert:Actor {name: "Robert De Niro"})-[r2:ACTED_IN]->(m)
MATCH (martin:Director {name: "Martin Scorsese"})-[r4:DIRECTED]->(m)
MATCH (m)-[r3:IN_GENRE]->(genre:Genre)
RETURN joe.name AS Actor1, robert.name AS Actor2, martin.name AS Director, m.title AS Movie, genre.name AS
Genre
```

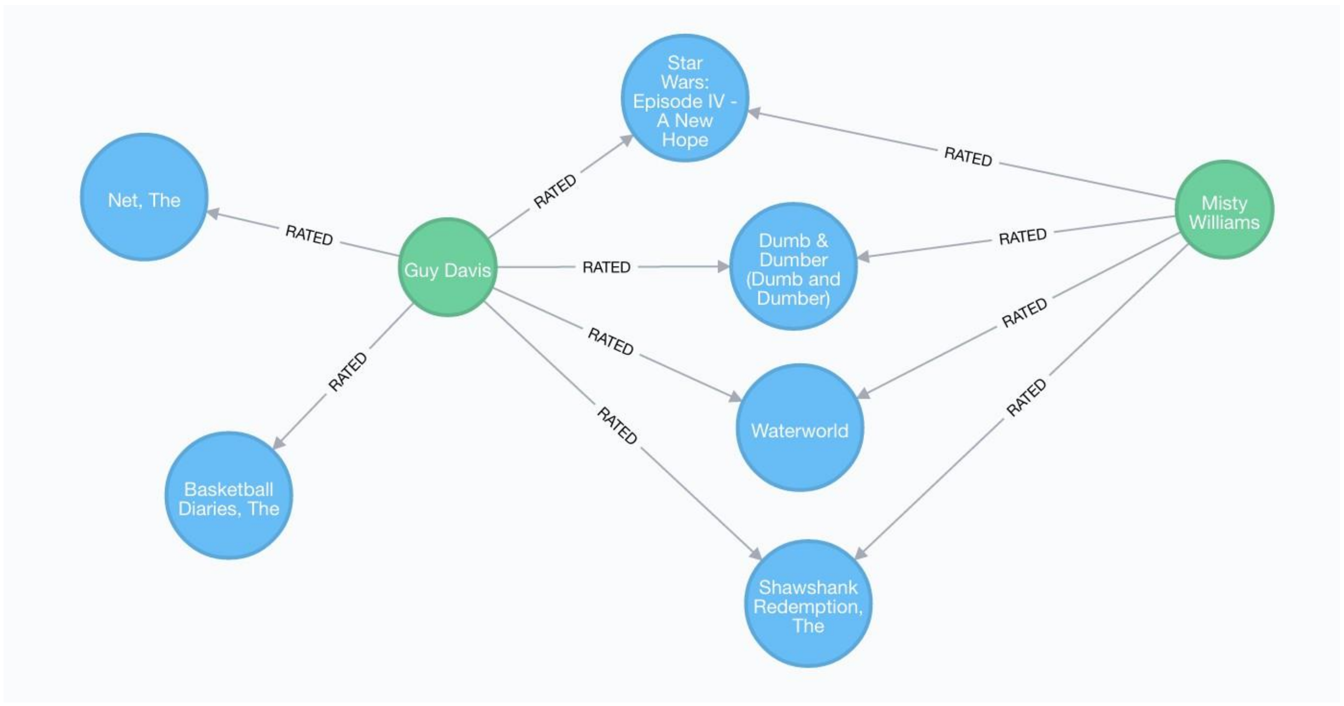
2/ After you completed previous request and tested it, create your own User defined procedure

to do the same work.

```
package example;

import org.neo4j.graphdb.Result;
import org.neo4j.graphdb.Transaction;
import org.neo4j.procedure.Context;
import org.neo4j.procedure.Mode;
import org.neo4j.procedure.Procedure;

import java.util.Map;
import java.util.stream.Stream;
import java.util.stream.StreamSupport;

public class Question2 {

    @Context
    public Transaction tx;

    public static class EntityContainer {
        public String Actor1;
        public   String Actor2;
        public String Director;
        public String Movie;
        public String Genre;


        public EntityContainer(String Actor1,  String Actor2, String Director, String Movie,
String Genre ) {
            this.Actor1=Actor1;
            this.Actor2=Actor2;
            this.Director=Director;
            this.Movie=Movie;
            this.Genre=Genre;
        }
    }
////works!
    @Procedure(name = "recommend.recommendItems", mode = Mode.READ)
    public Stream<EntityContainer> recommendItems() {
        String apocQuery = "MATCH (joe:Actor {name: \"Joe Pesci\"})-[r1:ACTED_IN]->(m:Movie)\n" +
                "MATCH (robert:Actor {name: \"Robert De Niro\"})-[r2:ACTED_IN]->(m)\n" +
                "MATCH (martin:Director {name: \"Martin Scorsese\"})-[r4:DIRECTED]->(m)\n" +
                "MATCH (m)-[r3:IN_GENRE]->(genre:Genre)\n" +
                "RETURN joe.name AS Actor1, robert.name AS Actor2, martin.name AS Director,
m.title AS Movie, genre.name AS Genre";

        Result result = tx.execute(apocQuery);
        Iterable<Map<String, Object>> iterable = () -> result;
        return StreamSupport.stream(iterable.spliterator(), false)
                .map(row -> new EntityContainer(
                        (String) row.get("Actor1"),
                        (String) row.get("Actor2"),
                        (String) row.get("Director"),
                        (String) row.get("Movie"),
                        (String) row.get("Genre")));



    }
}
```

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
Misty Williams
Dumb & Dumber (Dumb and Dumber)
Waterworld
Basketball Diaries, The
Shawshank Redemption, The
RATED

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

```
MATCH (u:User)-[: RATED]->(m1:Movie {title: "Waiting to Exhale"})
MATCH (u)-[:RATED]->(m2:Movie {title: "Jumanji"})
MATCH (u)-[:RATED]->(m:Movie )
RETURN u, m1, m2, m
```

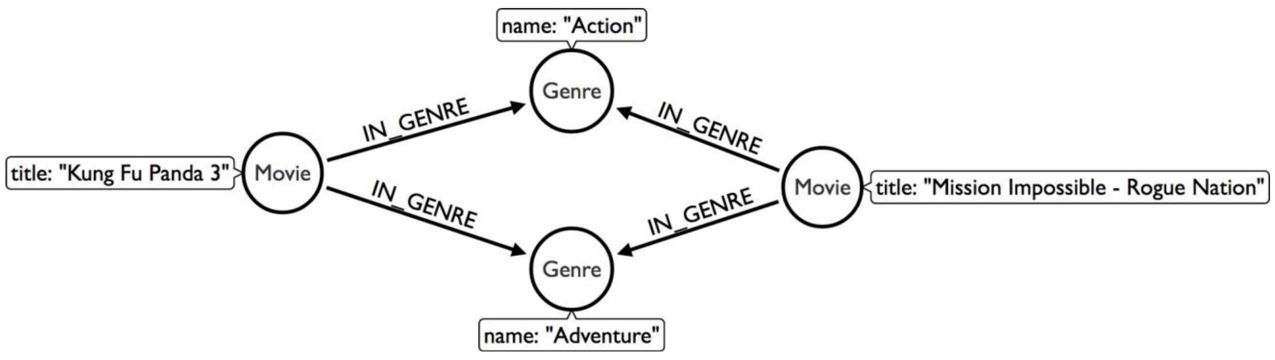
2/ After you completed previous request and tested it, create your own User defined procedure

to do the same work.

```
public class Question3 {

    @Context
    public Transaction tx;


    public static class EntityContainer {
        public Node u ;
        public Node m1 ;
        public Node m2 ;
        public Node m ;

        public EntityContainer(Node u ,Node m1 , Node m2 ,Node m  ) {
            this.u=u;
            this.m1=m1;
            this.m2=m2;
            this.m=m;
        }
    }
    ////Works
//  "Toy Story"  "Jumanji"  "Grumpier Old Men"  "Waiting to Exhale"
    @Procedure(name = "recommend.collaborativeFiltering", mode = Mode.READ)
    public Stream<EntityContainer> collaborativeFiltering () {
        String apocQuery = "MATCH (u:User)-[: RATED]->(m1:Movie {title: \"Waiting to
Exhale\"})\n" +
                "MATCH (u)-[:RATED]->(m2:Movie {title: \"Jumanji\"})\n" +
                "MATCH (u)-[:RATED]->(m:Movie )\n" +
                "RETURN u, m1, m2, m";

        Result result = tx.execute(apocQuery);
        Iterable<Map<String, Object>> iterable = () -> result;
        return StreamSupport.stream(iterable.spliterator(), false)
                .map(row -> new EntityContainer(
                        (Node) row.get("u"),
                        (Node) row.get("m1"),
                        (Node) row.get("m2"),
                        (Node) row.get("m")));

    }
}
```

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 (inception:Movie {title: "Inception"})-[r1:IN_GENRE]->(g:Genre)
MATCH (m:Movie)-[r2:IN_GENRE]->(g)
        WITH m, COLLECT(g.name) AS genres
RETURN  m.title AS movie, genres AS Genres
```

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

to do the same work.

```
public class Question4 {

    @Context
    public Transaction tx;


    public static class EntityContainer {
        public List<String> Genres ;

        public String movie ;

        public EntityContainer(String movie, List<String> Genres ) {
            this.movie=movie;
            this.Genres=Genres;

        }
    }
    ////works
    // "Toy Story"  "Jumanji"  "Grumpier Old Men"  "Waiting to Exhale"
    @Procedure(name = "recommend.similarityBasedOnCommonGenres", mode = Mode.READ)
    public Stream<EntityContainer> similarityBasedOnCommonGenres () {
        String apocQuery = "MATCH (inception:Movie {title: \"Inception\"})-[r1:IN_GENRE]-
>(g:Genre)\n" +
                "MATCH (m:Movie)-[r2:IN_GENRE]->(g)\n" +
                "WITH m, COLLECT(g.name) AS genres\n" +
                "RETURN  m.title AS movie, genres AS Genres";

        Result result = tx.execute(apocQuery);
        Iterable<Map<String, Object>> iterable = () -> result;
        return StreamSupport.stream(iterable.spliterator(), false)
                .map(row -> new EntityContainer(
                        (String) row.get("movie"),
                        (List<String>) row.get("Genres")));

    }
}
```

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{name: "Jessica Sherman"})-[r:RATED]->(watched_movies:Movie)-[:IN_GENRE]->(g:Genre)
WITH u, COLLECT(DISTINCT g.name) AS genres, COLLECT(watched_movies) AS watched_movies_list
MATCH (all_movies:Movie)-[:IN_GENRE]->(gm:Genre)
WHERE gm.name IN genres AND NOT all_movies IN watched_movies_list
RETURN DISTINCT all_movies.title AS suggested_movies
```

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

to do the same work.

```
public class Question5 {

    @Context
    public Transaction tx;


    public static class EntityContainer {
        public String suggested_movies ;

        public EntityContainer(String suggested_movies ) {
            this.suggested_movies=suggested_movies;


        }
    }
    // "Omar Huffman" "Mr. Jason Love" "Angela Thompson"
    ////Works
    @Procedure(name = "recommend.personalizedRecommendationsBasedOnGenres", mode = Mode.READ)
    public Stream<EntityContainer> personalizedRecommendationsBasedOnGenres () {
        String apocQuery = "MATCH (u:User{name: \"Jessica Sherman\"})-[r:RATED]-
>(watched_movies:Movie)-[:IN_GENRE]->(g:Genre)\n" +
                "WITH u, COLLECT(DISTINCT g.name) AS genres, COLLECT(watched_movies) AS
watched_movies_list\n" +
                "MATCH (all_movies:Movie)-[:IN_GENRE]->(gm:Genre)\n" +
                "WHERE gm.name IN genres AND NOT all_movies IN watched_movies_list\n" +
                "RETURN DISTINCT all_movies.title AS suggested_movies";

        Result result = tx.execute(apocQuery);
        Iterable<Map<String, Object>> iterable = () -> result;
        return StreamSupport.stream(iterable.spliterator(), false)
                .map(row -> new EntityContainer((String) row.get("suggested_movies")));

    }
}
```

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

```
// Collect the genres, actors, and directors for movies rated above 3 by Omar Huffman
MATCH (u:User {name: "Omar Huffman"})-[r:RATED]->(m:Movie)-[:IN_GENRE]->(g:Genre)
WHERE r.rating > 3
WITH COLLECT(DISTINCT g.name) AS allGenres, m
MATCH (m)<-[:ACTED_IN]-(A:Person)
WITH allGenres, COLLECT(DISTINCT A.name) AS allActors, m
MATCH (m)<-[:DIRECTED]-(D:Person)
WITH allGenres, allActors, COLLECT(DISTINCT D.name) AS allDirectors, m

// Calculate scores for other movies based on the collected genres, actors, and directors
MATCH (gm:Movie)
WITH gm, allGenres, allActors, allDirectors,
   size([a IN allActors WHERE (gm)<-[:ACTED_IN]-(:Person {name: a})]) * 3 AS actorScore,
   size([g IN allGenres WHERE (gm)-[:IN_GENRE]->(:Genre {name: g})]) * 5 AS genreScore,
   size([d IN allDirectors WHERE (gm)<-[:DIRECTED]-(:Person {name: d})]) * 4 AS directorScore
WITH gm.title AS movie_name, (actorScore + genreScore + directorScore) AS score
RETURN movie_name, score
ORDER BY score DESC
```

```
public class Question6 {

    @Context
    public Transaction tx;


    public static class EntityContainer {
        public String movie_name;
        public long score;

        public EntityContainer(String movie_name, long score) {
            this.movie_name = movie_name;
            this.score = score;
        }
    }
    ////works
    // "Omar Huffman" "Mr. Jason Love" "Angela Thompson"
    @Procedure(name = "recommend.weightedContentAlgorithm", mode = Mode.READ)
    public Stream<EntityContainer> weightedContentAlgorithm() {
        String apocQuery =
                // Collect the genres, actors, and directors for movies rated above 3 by Omar
Huffman
                        "MATCH (u:User {name: \"Omar Huffman\"})-[r:RATED]->(m:Movie)-
[:IN_GENRE]->(g:Genre) \n" +
                        "WHERE r.rating > 3 \n" +
                        "WITH COLLECT(DISTINCT g.name) AS allGenres, m \n" +
                        "MATCH (m)<-[:ACTED_IN]-(A:Person) \n" +
                        "WITH allGenres, COLLECT(DISTINCT A.name) AS allActors, m \n" +
                        "MATCH (m)<-[:DIRECTED]-(D:Person) \n" +
                        "WITH allGenres, allActors, COLLECT(DISTINCT D.name) AS allDirectors, m
\n" +
                        "// Calculate scores for other movies based on the collected genres,
actors, and directors \n" +
                        "MATCH (gm:Movie) \n" +
                        "WITH gm, allGenres, allActors, allDirectors, \n" +
                        "      size([a IN allActors WHERE (gm)<-[:ACTED_IN]-(:Person {name: a})])
* 3 AS actorScore, \n" +
                        "      size([g IN allGenres WHERE (gm)-[:IN_GENRE]->(:Genre {name: g})]) *
5 AS genreScore, \n" +
                        "      size([d IN allDirectors WHERE (gm)<-[:DIRECTED]-(:Person {name:
d})]) * 4 AS directorScore \n" +
                        "WITH gm.title AS movie_name, (actorScore + genreScore + directorScore)
AS score \n" +
                        //"ORDER BY score DESC \n"+
                        "RETURN movie_name, score ";

        Result result = tx.execute(apocQuery);
        Iterable<Map<String, Object>> iterable = () -> result;
        return StreamSupport.stream(iterable.spliterator(), false)
                .map(row -> new EntityContainer(
                        (String) row.get("suggested_movies"),
                        (long) row.get("score")));

    }
}
```

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.

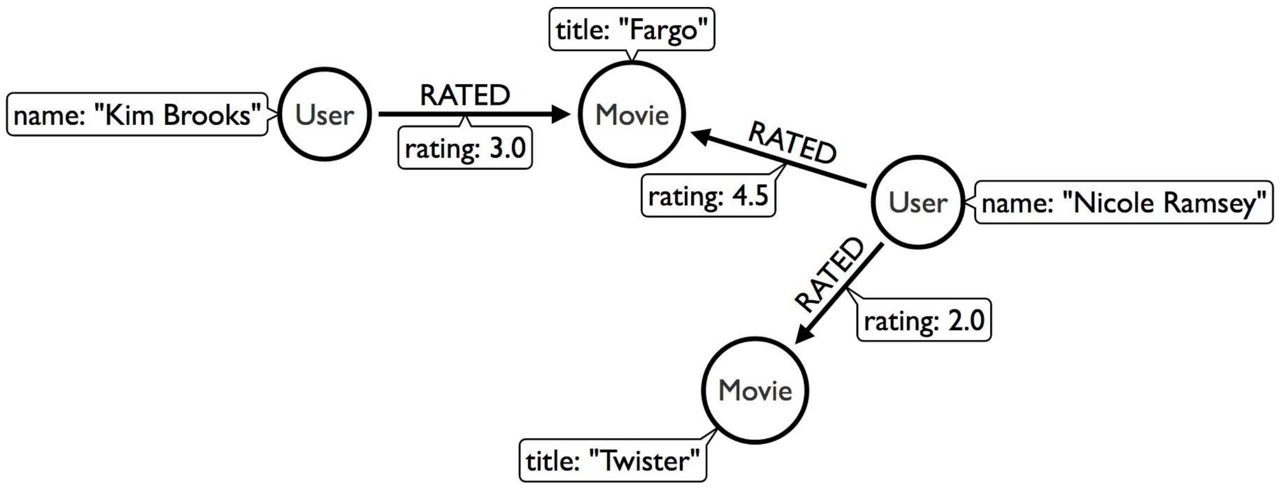
We can calculate the Jaccard index for sets of movie genres to determine how similar two movies are.

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

```
// Step 1: Retrieve genres for all movies
MATCH (m:Movie)-[:IN_GENRE]->(g:Genre)
WITH m.title AS movie, collect(g.name) AS genres

// Step 2: Retrieve genres for the movie "Inception"
MATCH (inception:Movie {title: 'Inception'})-[:IN_GENRE]->(inceptionGenre:Genre)
WITH inception, collect(inceptionGenre.name) AS inceptionGenres, movie, genres

// Step 3: Calculate Jaccard similarity for genres and find movies similar to "Inception"
WHERE movie <> 'Inception' // Exclude the movie "Inception" itself
WITH movie, genres, inceptionGenres,
   apoc.coll.intersection(genres, inceptionGenres) AS genreIntersection,
   apoc.coll.union(genres, inceptionGenres) AS genreUnion
WITH movie, (toFloat(size(genreIntersection)) / size(genreUnion)) AS genreJaccard
ORDER BY genreJaccard DESC
LIMIT 10
RETURN movie, genreJaccard
```

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

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 r.rating,m.title
```

Find Misty's average rating

```
// Match the ratings given by Misty Williams to movies
MATCH (u:User {name: "Misty Williams"})-[r:RATED]->(m:Movie)
// Return the average rating
RETURN avg(r.rating) AS averageRating;
```

What are the movies that Misty liked more than average?

```
// Step 1: Calculate Misty Williams' average rating
MATCH (u:User {name: "Misty Williams"})-[r:RATED]->(m:Movie)
WITH u, avg(r.rating) AS averageRating

// Step 2: Find movies that Misty Williams rated higher than her average rating
MATCH (u)-[r:RATED]->(m:Movie)
WHERE u.name = "Misty Williams" AND r.rating > averageRating
RETURN m.title AS movie, r.rating AS rating, averageRating
ORDER BY r.rating DESC;
```

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.

```
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

```
MATCH (peer:User)-[r:RATED]->(movie:Movie)
WITH AVG(r.rating) AS avgRating
RETURN avgRating AS OverallAverageRating
```

```
MATCH (peer:User)-[r:RATED]->(movie:Movie)
WITH AVG(r.rating) AS  OverallAverageRating

// Step 2: Query to find genres with higher than average rating
MATCH (u:User  )-[r1:RATED]->(m:Movie)-[:IN_GENRE]->(g:Genre)
WITH g.name AS Genre, AVG(r1.rating) AS AverageRating, OverallAverageRating
WHERE AverageRating > OverallAverageRating
RETURN COLLECT(Genre)
```

```

// Step 1: Calculate Overall Average Rating
MATCH (peer:User)-[r:RATED]->(movie:Movie)
WITH AVG(r.rating) AS OverallAverageRating

// Step 2: Query to find genres with higher than average rating
MATCH (u:User {name: 'Cynthia Freeman'})-[r1:RATED]->(m:Movie)-[:IN_GENRE]->(g:Genre)
WITH g.name AS Genre, AVG(r1.rating) AS AverageRating, OverallAverageRating ,u
WHERE AverageRating > OverallAverageRating

// Step 3: Retrieve movies in these genres that Cynthia Freeman has not rated
MATCH (allM:Movie)-[:IN_GENRE]->(g:Genre)
WHERE g.name IN [Genre] AND NOT (u)-[:RATED]->(allM)

RETURN allM.title
```

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 (cynthia:User {name: 'Cynthia Freeman'})-[r:RATED]->(m:Movie)
WITH cynthia, COLLECT({movieId: ID(m), rating: r.rating}) AS cynthiaRatings

// Match all other users who have rated the same movies as Cynthia
MATCH (other:User)-[r2:RATED]->(m)
WHERE other.name <> 'Cynthia Freeman'

// Collect ratings for each user
WITH cynthia, cynthiaRatings, other, COLLECT({movieId: ID(m), rating: r2.rating}) AS otherRatings

// Filter out cases where ratings are empty or of different sizes
WHERE SIZE(cynthiaRatings) > 0 AND SIZE(otherRatings) > 0
 AND SIZE(cynthiaRatings) = SIZE(otherRatings)

// Calculate dot product, magnitude of ratings for Cynthia and other user
WITH cynthia, other,
    REDUCE(dotProduct = 0.0, idx IN RANGE(0, SIZE(cynthiaRatings)-1) |
      dotProduct + cynthiaRatings[idx].rating * otherRatings[idx].rating) AS dotProduct,
    SQRT(REDUCE(csq = 0.0, idx IN RANGE(0, SIZE(cynthiaRatings)-1) |
      csq + toFloat(cynthiaRatings[idx].rating)^2)) AS cynthiaMagnitude,
    SQRT(REDUCE(isq = 0.0, idx IN RANGE(0, SIZE(otherRatings)-1) |
      isq + toFloat(otherRatings[idx].rating)^2)) AS otherMagnitude

// Calculate cosine similarity, handling division by zero
WITH cynthia, other,
    CASE
     WHEN cynthiaMagnitude > 0 AND otherMagnitude > 0 THEN dotProduct / (cynthiaMagnitude *
otherMagnitude)
     ELSE 0.0 // Handle division by zero case
    END AS cosineSimilarity

// Return the results
RETURN other.name AS otherUser, cosineSimilarity
ORDER BY cosineSimilarity DESC;
```

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

```
MATCH (cynthia:User {name: 'Cynthia Freeman'})-[r:RATED]->(:Movie)
WITH cynthia, COLLECT(r.rating) as cynthiaRating
```

```
// Match all other users who have rated the same movies as Cynthia
MATCH (other:User)-[r2:RATED]->(:Movie)
WHERE other.name <> 'Cynthia Freeman'

// Collect ratings for each user
WITH cynthia, cynthiaRating, other, COLLECT(r2.rating) as otherRating

// Filter out cases where ratings are empty or of different sizes
WHERE size(cynthiaRating) > 0 AND size(otherRating) > 0 AND size(cynthiaRating) = size(otherRating)

// Calculate cosine similarity for each pair
WITH cynthia, cynthiaRating, other, otherRating,
    gds.similarity.cosine(cynthiaRating, otherRating) AS cosineSimilarity

// Return the results
RETURN other.name AS otherUser, cosineSimilarity
ORDER BY cosineSimilarity DESC;
```

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

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 (cynthia:User {name: 'Cynthia Freeman'})-[r:RATED]->(m:Movie)
WITH cynthia, COLLECT({movieId: ID(m), rating: r.rating}) AS cynthiaRatings

// Match all other users who have rated the same movies as Cynthia
MATCH (other:User)-[r2:RATED]->(m)
WHERE other.name <> 'Cynthia Freeman'

// Collect ratings for each user
WITH cynthia, cynthiaRatings, other, COLLECT({movieId: ID(m), rating: r2.rating}) AS otherRatings

// Filter out cases where ratings are empty or of different sizes
WHERE SIZE(cynthiaRatings) > 0 AND SIZE(otherRatings) > 0
 AND SIZE(cynthiaRatings) = SIZE(otherRatings)

// Calculate average ratings for Cynthia and other user
WITH cynthia, cynthiaRatings, other, otherRatings,
   REDUCE(cynthiaSum = 0.0, rating IN cynthiaRatings | cynthiaSum + rating.rating) / SIZE(cynthiaRatings)
AS cynthiaAvgRating,
   REDUCE(otherSum = 0.0, rating IN otherRatings | otherSum + rating.rating) / SIZE(otherRatings) AS
otherAvgRating

// Calculate numerator and denominators for Pearson correlation
WITH cynthia, other, cynthiaAvgRating, otherAvgRating,
   REDUCE(dotProduct = 0.0, idx IN RANGE(0, SIZE(cynthiaRatings)-1) |
     dotProduct + (cynthiaRatings[idx].rating - cynthiaAvgRating) * (otherRatings[idx].rating -
otherAvgRating)) AS numerator,
   SQRT(REDUCE(csq = 0.0, idx IN RANGE(0, SIZE(cynthiaRatings)-1) |
     csq + (cynthiaRatings[idx].rating - cynthiaAvgRating)^2)) AS cynthiaStdDev,
   SQRT(REDUCE(isq = 0.0, idx IN RANGE(0, SIZE(otherRatings)-1) |
     isq + (otherRatings[idx].rating - otherAvgRating)^2)) AS otherStdDev

// Calculate Pearson correlation, handling division by zero
WITH cynthia, other,
   CASE
    WHEN cynthiaStdDev > 0 AND otherStdDev > 0 THEN numerator / (cynthiaStdDev * otherStdDev)
    ELSE 0.0 // Handle division by zero case
   END AS pearsonSimilarity

// Return the results
RETURN other.name AS otherUser, pearsonSimilarity
ORDER BY pearsonSimilarity DESC;
```

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 (cynthia:User {name: 'Cynthia Freeman'})-[r:RATED]->(:Movie)
WITH cynthia, COLLECT(r.rating) as cynthiaRating

// Match all other users who have rated the same movies as Cynthia
MATCH (other:User)-[r2:RATED]->(:Movie)
WHERE other.name <> 'Cynthia Freeman'
```

```
// Collect ratings for each user
WITH cynthia, cynthiaRating, other, COLLECT(r2.rating) as otherRating

// Filter out cases where ratings are empty or of different sizes
WHERE size(cynthiaRating) > 0 AND size(otherRating) > 0 AND size(cynthiaRating) = size(otherRating)

// Calculate pearson similarity for each pair
WITH cynthia, cynthiaRating, other, otherRating,
   gds.similarity.pearson(cynthiaRating, otherRating) AS cosineSimilarity

// Return the results
RETURN other.name AS otherUser, cosineSimilarity
ORDER BY cosineSimilarity DESC;
```

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

```
MATCH (cynthia:User {name: 'Cynthia Freeman'})-[r:RATED]->(m:Movie)
WITH cynthia, COLLECT({movieId: ID(m), rating: r.rating}) AS cynthiaRatings

// Match all other users who have rated the same movies as Cynthia
MATCH (other:User)-[r2:RATED]->(m)
WHERE other.name <> 'Cynthia Freeman'

// Collect ratings for each user
WITH cynthia, cynthiaRatings, other, COLLECT({movieId: ID(m), rating: r2.rating}) AS otherRatings

// Filter out cases where ratings are empty or of different sizes
WHERE SIZE(cynthiaRatings) > 0 AND SIZE(otherRatings) > 0
 AND SIZE(cynthiaRatings) = SIZE(otherRatings)

// Calculate Pearson similarity between Cynthia and other users
WITH cynthia, other,
   REDUCE(dotProduct = 0.0, idx IN RANGE(0, SIZE(cynthiaRatings)-1) |
     dotProduct + (cynthiaRatings[idx].rating * otherRatings[idx].rating)) AS dotProduct,
   SQRT(REDUCE(csq = 0.0, idx IN RANGE(0, SIZE(cynthiaRatings)-1) |
     csq + (cynthiaRatings[idx].rating)^2)) AS cynthiaMagnitude,
   SQRT(REDUCE(isq = 0.0, idx IN RANGE(0, SIZE(otherRatings)-1) |
     isq + (otherRatings[idx].rating)^2)) AS otherMagnitude

// Calculate Pearson correlation coefficient
WITH cynthia, other,
   CASE
     WHEN cynthiaMagnitude > 0 AND otherMagnitude > 0 THEN dotProduct / (cynthiaMagnitude *
otherMagnitude)
     ELSE 0.0
   END AS pearsonSimilarity

// Select top 10 most similar users based on Pearson similarity
WITH cynthia, other, pearsonSimilarity
ORDER BY pearsonSimilarity DESC
LIMIT 10

// Find movies highly rated by these similar users that Cynthia hasn't seen
MATCH (other)-[r3:RATED]->(movie:Movie)
WHERE NOT EXISTS {
   MATCH (cynthia)-[:RATED]->(movie)
}
AND r3.rating >= 4.0 // Consider only highly rated movies (adjust rating threshold as needed)

// Return recommended movies along with the ratings from similar users
RETURN movie.title AS recommendedMovie, r3.rating AS ratingBySimilarUser
ORDER BY ratingBySimilarUser DESC;
```

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.

```
// Collect the genres, actors, and directors for movies rated above 3 by Omar Huffman
MATCH (u:User {name: "Omar Huffman"})-[r:RATED]->(m:Movie)-[:IN_GENRE]->(g:Genre)
WHERE r.rating > 3
WITH COLLECT(DISTINCT g.name) AS allGenres, m

MATCH (m)<-[:ACTED_IN]-(A:Person)
WITH allGenres, COLLECT(DISTINCT A.name) AS allActors, m

MATCH (m)<-[:DIRECTED]-(D:Person)
WITH allGenres, allActors, COLLECT(DISTINCT D.name) AS allDirectors, m

// Calculate scores for other movies based on the collected genres, actors, and directors
MATCH (gm:Movie)
WITH gm, allGenres, allActors, allDirectors,
   size([a IN allActors WHERE (gm)<-[:ACTED_IN]-(:Person {name: a})]) * 3 AS actorScore,
   size([g IN allGenres WHERE (gm)-[:IN_GENRE]->(:Genre {name: g})]) * 5 AS genreScore,
   size([d IN allDirectors WHERE (gm)<-[:DIRECTED]-(:Person {name: d})]) * 4 AS directorScore

// Calculate watch date-based scores
OPTIONAL MATCH (u:User)-[r:WATCHED]->(gm)
WITH gm.title AS movie_name, (actorScore + genreScore + directorScore) AS baseScore, r.watchDate AS
watchDate, datetime() AS now
```

```
WITH movie_name, baseScore,
   CASE
    WHEN watchDate IS NOT NULL AND duration.inDays(watchDate, now).days <= 7 THEN 3
    WHEN watchDate IS NOT NULL AND duration.inDays(watchDate, now).days <= 30 THEN 2
    ELSE 1
   END AS timeScore

RETURN movie_name, (baseScore * timeScore) AS totalScore
ORDER BY totalScore DESC
```

Keyword extraction

Enhance the traits available using the plot description.

How would you model extracted keywords for movies?