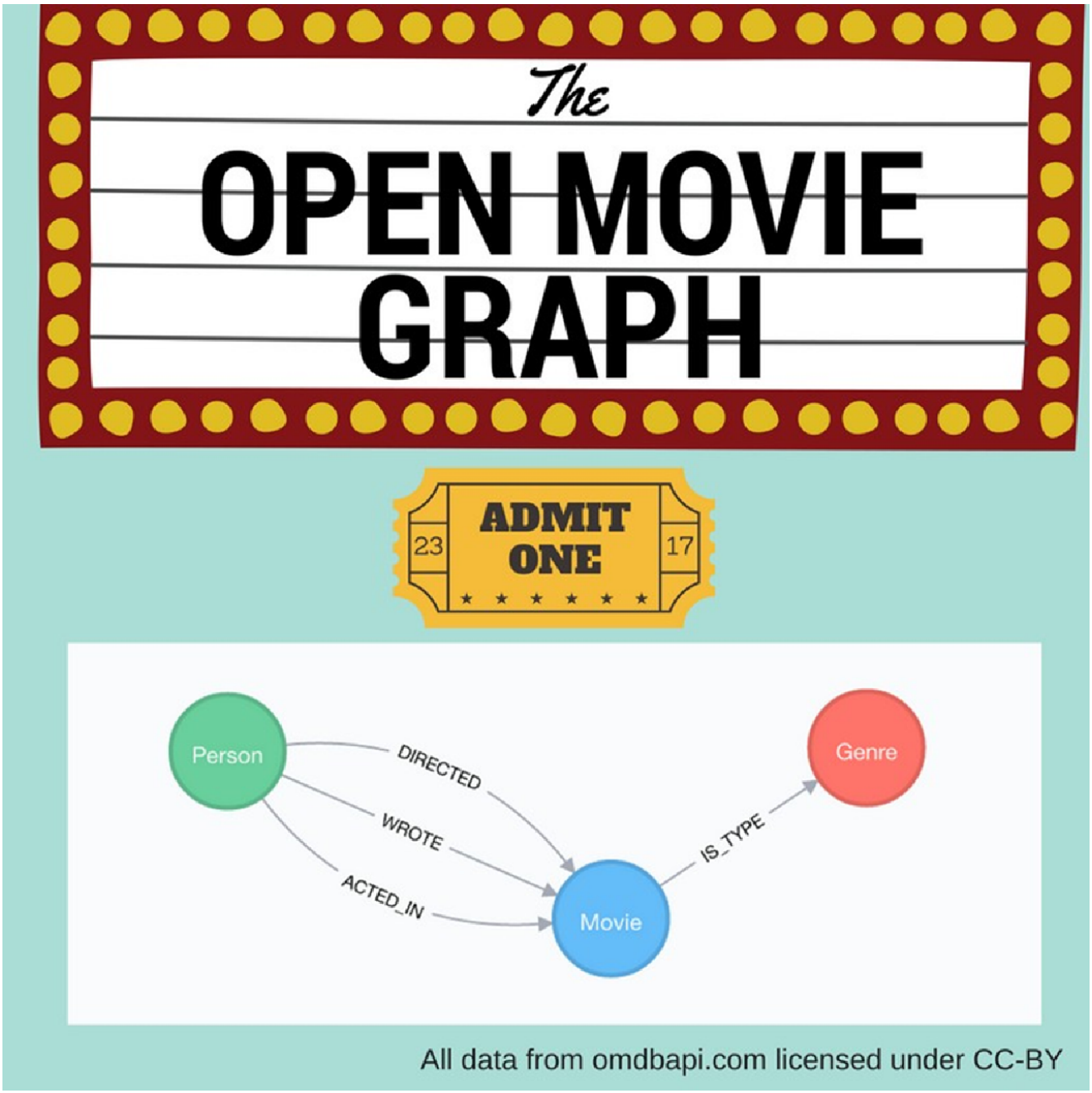
```
 Projet NEO4J :
Réalisé(e)s par :
    - M MEJRI Salam
    - Mme BENAMOR Nesrine
Encadré par :
    - M CISSE Ansoumana
```

Final Project :Build Recommendations system using movies data

Personalized Product Recommendations with Neo4j



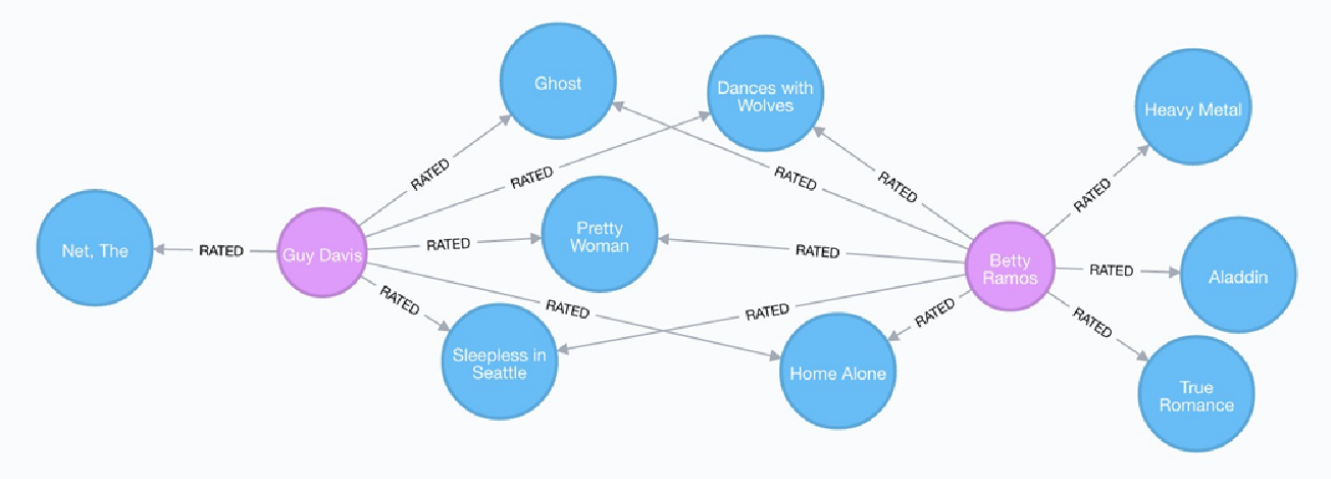
Recommendations

Personalized product recommendations can increase conversions, improve sales rates and provide a better experience 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
1.  Create empty database named recommendations
     CREATE DATABASE recommendations
2.  Use recommendations
     :use recommendations
3.  Import data from file data/all-plain.cypher
call apoc.cypher.runFile('file:///C:/Users/TUF-GAMING-
SALAM/Desktop/RenduNeo4j/data/all-plain.cypher',{statistics:true})
```

The Open Movie Graph Data Model

The Property Graph Model

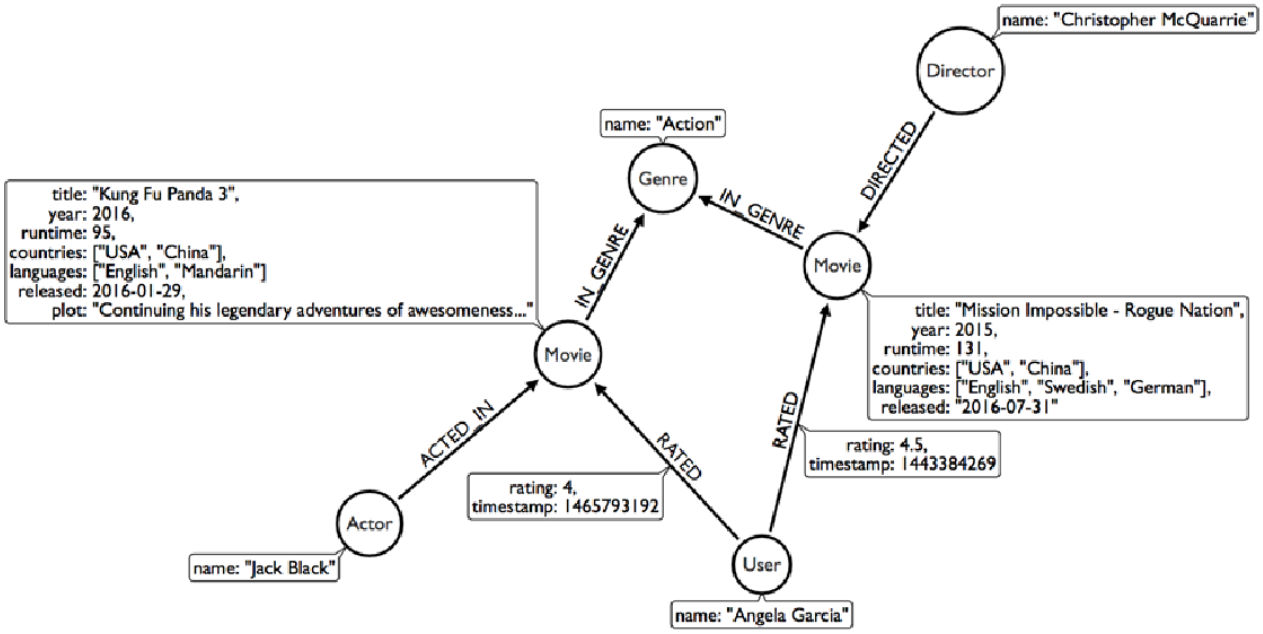
The data model of graph databases is called the labeled property graph model. Nodes: The entities

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.



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)←[:RATED]-(:User)</code>

Variables

Graph elements can be bound to variables that can be referred to later in the query: <code>(m:Movie)←[r:RATED]-(u: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 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.

```
package com.example.project;

import org.neo4j.graphdb.Node;
import org.neo4j.graphdb.Result;
import org.neo4j.graphdb.Transaction;
import org.neo4j.procedure.*;
import java.util.stream.Collectors;
import java.util.stream.Stream;

/**
 * This is an example returning {@link org.neo4j.graphdb.Entity Entities} from stored
procedures.
 * {@link Node Nodes} and {@link org.neo4j.graphdb.Relationship relationships} are
both entities
 * and can only be accessed in their transaction. So it is important that you use the
injected one
 * and not open a new one; otherwise you can access them from the outside.
 */
public class MovieRecommendation {

    @Context
    public Transaction tx;

    @Procedure(name = "recommend.howManyReview", mode = Mode.READ)
    @Description("recommend.howManyReview(title)- returns number of rated per title")
    public Stream<ReviewsByMovie> howManyReview(@Name(value = "title",defaultValue =
"Matrix") String title) {

        String query = "MATCH(m :Movie)<-[ :RATED]-(u:User) WHERE m.title CONTAINS '"
+title+"' WITH m, count(*) AS reviews RETURN m.title AS movie, reviews ORDER BY
reviews DESC ";

        Result result =tx.execute(query);
        return result.stream().map(obj->{
            return new  ReviewsByMovie((String) obj.get("movie"),(Long) obj.get(
"reviews"));
        }).collect(Collectors.toList()).stream();

    }

    public static class ReviewsByMovie {
        // These records contain two lists of distinct relationship types going in and
out of a Node.
        public String movie;
        public Long reviews;

        public ReviewsByMovie(String movie, Long reviews) {
            this.movie = movie;
            this.reviews = reviews;
        }
```

```
    }
}
```

Pour appeler la procédure `howManyReview` dans Neo4j, nous utilisons :

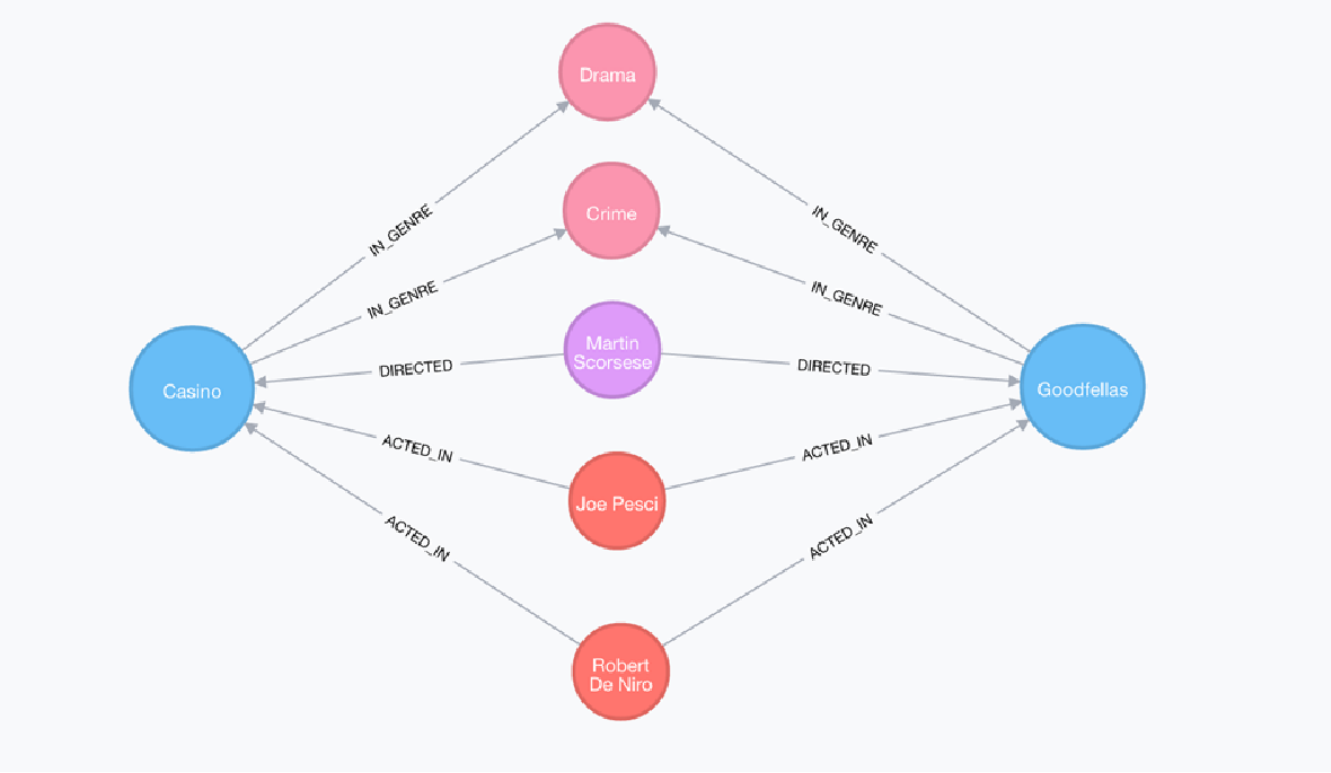
```
call recommend.howManyReview() yield movie, reviews  return movie,reviews
```

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 p=(m:Movie {title: 'Net, The'})-[:ACTED_IN|IN_GENRE|DIRECTED*2]-()
RETURN p LIMIT 25;
```

2/ After you completed previous request and tested it, create your own User defined procedure to do the same work.

```
package com.example.project;
```

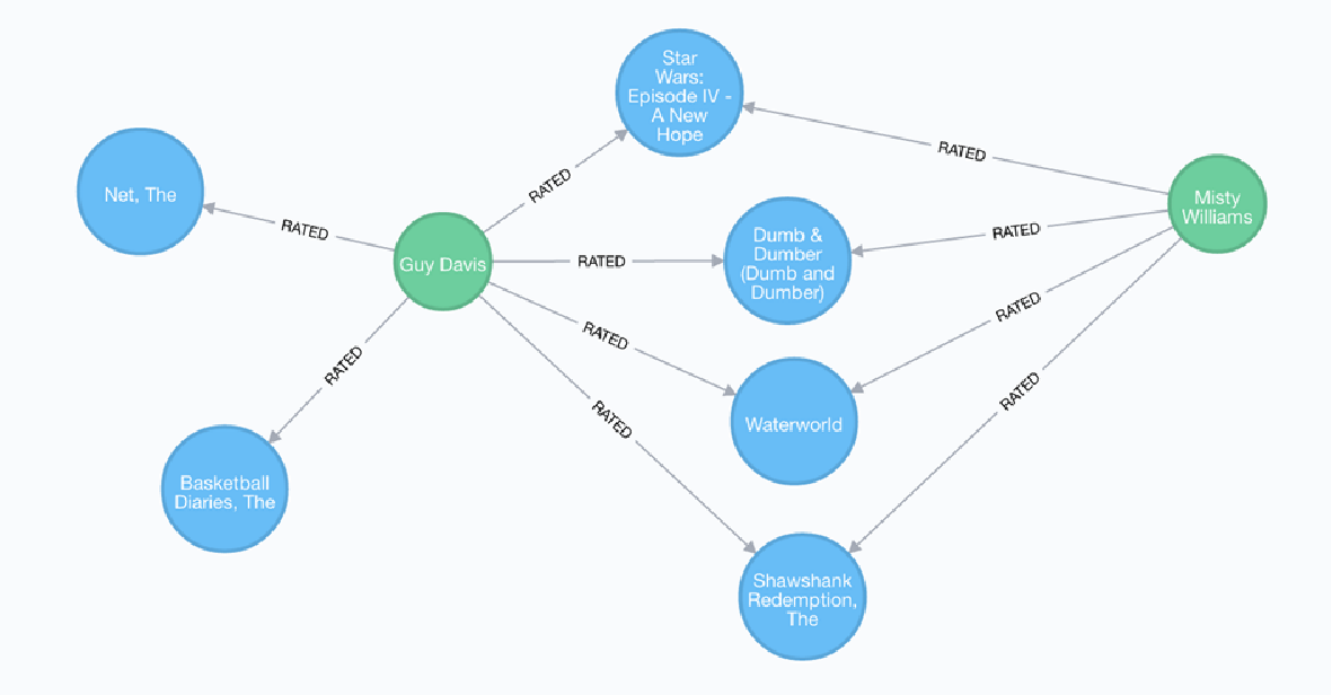
```
import org.neo4j.graphdb.*;
import org.neo4j.procedure.*;
import java.util.stream.Stream;

public class MoviePaths {

    @Context
    public GraphDatabaseService db;

    public static class MoviePath {
        public Path path;

        public MoviePath(Path path) {
            this.path = path;
        }
    }

    @Procedure(name = "recommend.moviePaths", mode = Mode.READ)
    @Description("RETURN paths connected to movie 'Net, The'")
    public Stream<MoviePath> moviePaths() {
        try (Transaction tx = db.beginTx()) {
            String query = "MATCH p=(m:Movie {title: 'Net, The'})-
[:ACTED_IN|IN_GENRE|DIRECTED*2]-() " +
                    "RETURN p LIMIT 25";
            Result result = tx.execute(query);

            return result.stream().map(row -> new MoviePath((Path) row.get("p")))
.onClose(tx::close);
        }
    }
}
```

Pour appeler la procédure `moviePaths` dans Neo4j, nous utilisons :

```
call recommend.moviePaths() yield path  return path
```

Collaborative Filtering

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



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

```
MATCH (m:Movie {title: 'Crimson Tide'})<-[:RATED]- (u:User)-[:RATED]->(rec:Movie)
WITH rec, COUNT(*) AS usersWhoAlsoWatched
ORDER BY usersWhoAlsoWatched DESC LIMIT 25
RETURN rec.title AS recommendation, usersWhoAlsoWatched
```

2/ After you completed previous request and tested it, create your own User defined procedure to do the same work.

```
package com.example.project;

import org.neo4j.graphdb.*;
import org.neo4j.procedure.*;
import java.util.stream.Stream;

public class Q3Recommendations {

    @Context
    public GraphDatabaseService db;

    public static class Recommendation {
        public String recommendation;
        public long usersWhoAlsoWatched;

        public Recommendation(String recommendation, long usersWhoAlsoWatched) {
            this.recommendation = recommendation;
            this.usersWhoAlsoWatched = usersWhoAlsoWatched;
        }
    }
```

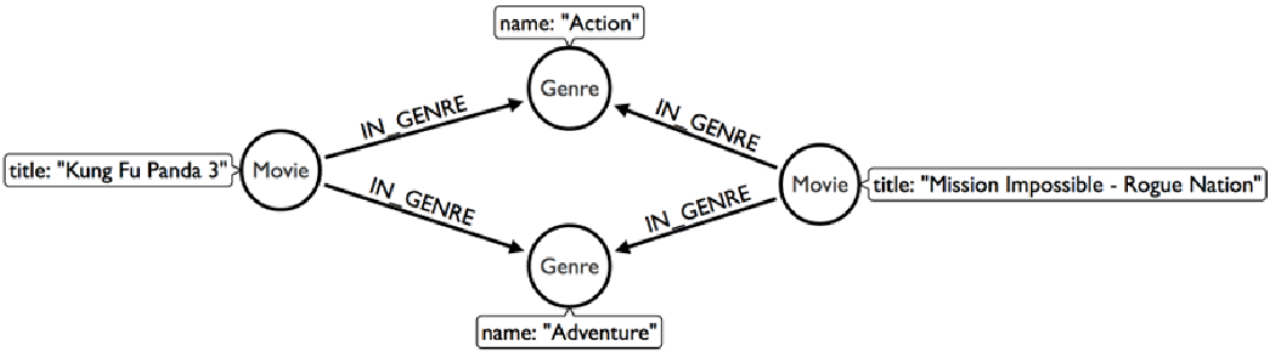
```
    @Procedure(name = "recommend.recommendations", mode = Mode.READ)
    @Description("RETURN movie recommendations for users who watched 'Crimson Tide'")
    public Stream<Recommendation> recommendations() {
        try (Transaction tx = db.beginTx()) {
            String query = "MATCH (m:Movie {title: 'Crimson Tide'})<-[:RATED]-
(u:User)-[:RATED]->(rec:Movie) " +
                    "WITH rec, COUNT(*) AS usersWhoAlsoWatched " +
                    "ORDER BY usersWhoAlsoWatched DESC LIMIT 25 " +
                    "RETURN rec.title AS recommendation, usersWhoAlsoWatched";
            Result result = tx.execute(query);

            return result.stream().map(row -> new Recommendation((String) row.get(
"recommendation"), (Long) row.get("usersWhoAlsoWatched"))).onClose(tx::close);
        }
    }
}
```

Pour appeler la procédure `recommendations` dans Neo4j, nous utilisons :

```
call recommend.recommendations() yield path  return path
```

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]-(rec:Movie)
WHERE m.title = 'Inception'
WITH rec, collect(g.name) AS genres, count(*) AS commonGenres
RETURN rec.title, genres, commonGenres
```

2/ After you completed previous request and tested it, create your own User defined procedure to do the same work.

```
package com.example.project;

import org.neo4j.graphdb.*;
import org.neo4j.procedure.*;
import java.util.List;
import java.util.stream.Stream;

public class Q4GenreRecommendations {

    @Context
    public GraphDatabaseService db;

    public static class GenreRecommendation {
        public String title;
        public List<String> genres;
        public long commonGenres;

        public GenreRecommendation(String title, List<String> genres, long
commonGenres) {
            this.title = title;
            this.genres = genres;
            this.commonGenres = commonGenres;
        }
    }

    @Procedure(name = "recommend.genreRecommendations", mode = Mode.READ)
    @Description("RETURN movie recommendations based on genres for movie 'Inception'")
    public Stream<GenreRecommendation> genreRecommendations() {
        try (Transaction tx = db.beginTx()) {
            String query = "MATCH (m:Movie)-[:IN_GENRE]->(g:Genre)<-[:IN_GENRE]-
(rec:Movie) " +
                    "WHERE m.title = 'Inception' " +
                    "WITH rec, collect(g.name) AS genres, count(*) AS commonGenres " +
                    "RETURN rec.title, genres, commonGenres";
            Result result = tx.execute(query);

            return result.stream().map(row -> new GenreRecommendation((String) row.
get("rec.title"), (List<String>) row.get("genres"), (Long) row.get("commonGenres")))
.onClose(tx::close);
        }
    }
}
```

Pour appeler la procédure `genreRecommendations` dans Neo4j, nous utilisons :

```
call recommend.genreRecommendations() yield title,genres,commonGenres  return
```

```
title,genres,commonGenres
```

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: 'Angelica Rodriguez'})-[r:RATED]->(m:Movie),(m)-[:IN_GENRE]-
>(g:Genre)<-[:IN_GENRE]-(rec:Movie)
WHERE NOT EXISTS{ (u)-[:RATED]->(rec) }
WITH rec, g.name as genre, count(*) AS count
WITH rec, collect([genre, count]) AS scoreComponents
RETURN rec.title AS recommendation, rec.year AS year, scoreComponents,reduce(s=0,x in
scoreComponents | s+x[1]) AS score
ORDER BY score DESC LIMIT 10;
```

2/ After you completed previous request and tested it, create your own User defined procedure to do the same work.

```
package com.example.project;

import org.neo4j.graphdb.*;
import org.neo4j.procedure.*;
import java.util.List;
import java.util.stream.Stream;

public class Q5UserRecommendations {

    @Context
    public GraphDatabaseService db;

    public static class UserRecommendation {
        public String recommendation;
        public long year;
        public List<List<Object>> scoreComponents;
        public long score;

        public UserRecommendation(String recommendation, long year, List<List<Object>>
scoreComponents, long score) {
            this.recommendation = recommendation;
            this.year = year;
            this.scoreComponents = scoreComponents;
            this.score = score;
        }
    }
```

```
    @Procedure(name = "recommend.userRecommendations", mode = Mode.READ)
    @Description("RETURN movie recommendations for user 'Angelica Rodriguez'")
    public Stream<UserRecommendation> userRecommendations() {
        try (Transaction tx = db.beginTx()) {
            String query = "MATCH (u:User {name: 'Angelica Rodriguez'})-[r:RATED]-
>(m:Movie),(m)-[:IN_GENRE]->(g:Genre)<-[:IN_GENRE]-(rec:Movie) " +
                    "WHERE NOT EXISTS{ (u)-[:RATED]->(rec) } " +
                    "WITH rec, g.name as genre, count(*) AS count " +
                    "WITH rec, collect([genre, count]) AS scoreComponents " +
                    "RETURN rec.title AS recommendation, rec.year AS year,
scoreComponents,reduce(s=0,x in scoreComponents | s+x[1]) AS score " +
                    "ORDER BY score DESC LIMIT 10";
            Result result = tx.execute(query);

            return result.stream().map(row -> new UserRecommendation((String) row.get
("recommendation"), (Long) row.get("year"), (List<List<Object>>) row.get(
"scoreComponents"), (Long) row.get("score"))).onClose(tx::close);
        }
    }
}
```

Pour appeler la procédure `userRecommendations` dans Neo4j, nous utilisons :

```
call recommend.userRecommendations() yield recommendation,year,scoreComponents,score
return recommendation,year,scoreComponents,score
```

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 (m:Movie) WHERE m.title = 'Wizard of Oz, The'
MATCH (m)-[:IN_GENRE]->(g:Genre)<-[:IN_GENRE]-(rec:Movie)
WITH m, rec, count(*) AS gs
OPTIONAL MATCH (m)<-[:ACTED_IN]-(a)-[:ACTED_IN]->(rec)
WITH m, rec, gs, count(a) AS as
OPTIONAL MATCH (m)<-[:DIRECTED]-(d)-[:DIRECTED]->(rec)
WITH m, rec, gs, as, count(d) AS ds
RETURN rec.title AS recommendation,(5*gs)+(3*as)+(4*ds) AS score
ORDER BY score DESC LIMIT 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.

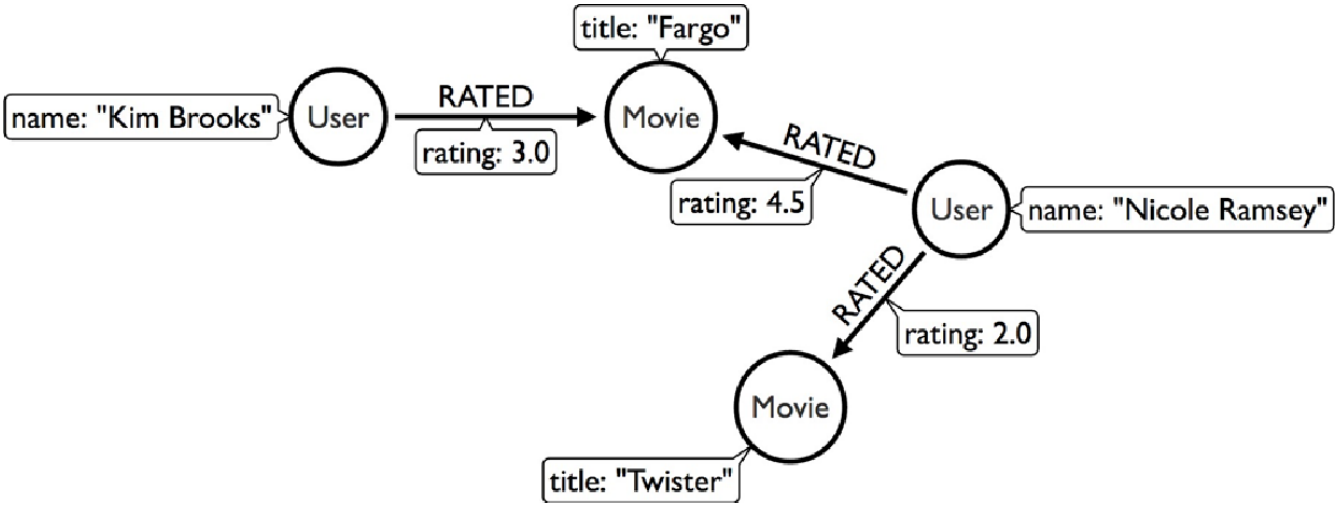
What movies are most similar to Inception based on Jaccard similarity of genres?

```
MATCH (m:Movie {title:'Inception'})-[:IN_GENRE]->(g:Genre)<-[:IN_GENRE]-(other:Movie)
WITH m, other, count(g) AS intersection, collect(g.name) as common
WITH m,other, intersection, common,[(m)-[:IN_GENRE]->(mg) | mg.name] AS set1,[(other)-
[:IN_GENRE]->(og) | og.name] AS set2
WITH m,other,intersection, common, set1, set2,set1+[x IN set2 WHERE NOT x IN set1] AS
union
RETURN m.title, other.title, common, set1,set2,((1.0*intersection)/size(union)) AS
jaccard
ORDER BY jaccard DESC LIMIT 25;
```

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

```
MATCH (m:Movie {title: 'Inception'})-[:IN_GENRE|ACTED_IN|DIRECTED]-(t)<-
[:IN_GENRE|ACTED_IN|DIRECTED]-(other:Movie)
WITH m, other, count(t) AS intersection, collect(t.name) AS common,[(m)-
[:IN_GENRE|ACTED_IN|DIRECTED]-(mt) | mt.name] AS set1, [(other)-
[:IN_GENRE|ACTED_IN|DIRECTED]-(ot) | ot.name] AS set2
WITH m,other,intersection, common, set1, set2,set1 + [x IN set2 WHERE NOT x IN set1]
AS union
RETURN m.title, other.title, common, set1,set2,((1.0*intersection)/size(union)) AS
jaccard
ORDER BY jaccard DESC LIMIT 25
```

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'})
MATCH (u)-[r:RATED]->(m:Movie)
RETURN *
LIMIT 100;
```

Find Misty's average rating

```
MATCH (u:User {name: 'Misty Williams'})
MATCH (u)-[r:RATED]->(m:Movie)
RETURN avg(r.rating) AS average;
```

What are the movies that Misty liked more than average?

```
MATCH (u:User {name: 'Misty Williams'})
MATCH (u)-[r:RATED]->(m:Movie)
WITH u, avg(r.rating) AS average
MATCH (u)-[r:RATED]->(m:Movie)
WHERE r.rating > average
RETURN *
LIMIT 100;
```

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

```
MATCH (u:User {name: 'Andrew Freeman'})-[r:RATED]->(m:Movie)
WITH u, avg(r.rating) AS mean

MATCH (u)-[r:RATED]->(m:Movie)-[:IN_GENRE]->(g:Genre)
WHERE r.rating > mean
WITH u, g, count(*) AS score

MATCH (g)<-[:IN_GENRE]-(rec:Movie)
WHERE NOT EXISTS { (u)-[:RATED]->(rec) }

RETURN rec.title AS recommendation, rec.year AS year,
```

```
       sum(score) AS scor,
       collect(DISTINCT g.name) AS genres
ORDER BY scor 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, count(m) AS numbermovies, sum(r1.rating * r2.rating) AS r1r2DotProduct,
     collect(r1.rating) as r1Ratings, collect(r2.rating) as r2Ratings
WHERE numbermovies > 10
WITH u1, u2, r1r2DotProduct, sqrt(reduce(r1Dot = 0.0, a IN r1Ratings | r1Dot + a^2))
AS r1Length,
sqrt(reduce(r2Dot = 0.0, b IN r2Ratings | r2Dot + b^2)) AS r2Length
RETURN u1.name, u2.name, r1r2DotProduct / (r1Length * r2Length) AS sim
ORDER BY sim DESC
LIMIT 100;
```

Explication

```
Cette requête Cypher est utilisée pour calculer la similarité cosinus entre
```

```
l'utilisateur 'Cynthia Freeman' et d'autres utilisateurs basée sur les films qu'ils
ont notés en commun. Voici une explication détaillée :

1. Première partie de la requête:

MATCH (p1:User {name: "Cynthia Freeman"})-[x:RATED]->(m:Movie)<-[y:RATED]-(p2:User)

- On commence par trouver l'utilisateur 'Cynthia Freeman' (u1) et un autre utilisateur
(u2) qui ont tous les deux noté le même film (m).
- [r1:RATED] et [r2:RATED] représentent les relations où u1 a noté le film m avec la
note r1.rating et u2 a noté le même film m avec la note r2.rating.

2. Deuxième partie de la requête:

WITH u1, u2, count(m) AS numbermovies, sum(r1.rating * r2.rating) AS r1r2DotProduct,
collect(r1.rating) as r1Ratings, collect(r2.rating) as r2Ratings
WHERE numbermovies > 10

- On regroupe les résultats pour u1 et u2, calculant le nombre de films notés en
commun (numbermovies), le produit scalaire des notes données par u1 et u2
(r1r2DotProduct).
- collect(r1.rating) as r1Ratings et collect(r2.rating) as r2Ratings collectent toutes
les notes données par u1 et u2 respectivement.

3. Troisième partie de la requête:

WITH u1, u2, r1r2DotProduct,
     sqrt(reduce(r1Dot = 0.0, a IN r1Ratings | r1Dot + a^2)) AS r1Length,
     sqrt(reduce(r2Dot = 0.0, b IN r2Ratings | r2Dot + b^2)) AS r2Length

- On calcule la longueur (norme) des vecteurs de notes de u1 (r1Length) et u2
(r2Length) en utilisant la formule mathématique de la norme euclidienne : la racine
carrée de la somme des carrés des éléments du vecteur.

4. Quatrième partie de la requête:

RETURN u1.name, u2.name, r1r2DotProduct / (r1Length * r2Length) AS sim
ORDER BY sim DESC
LIMIT 100;

- On retourne les noms de u1 et u2, ainsi que la similarité cosinus entre eux calculée
comme le quotient du produit scalaire (r1r2DotProduct) et du produit des normes des
vecteurs (r1Length * r2Length).
- Les résultats sont ordonnés par similarité cosinus décroissante (ORDER BY sim DESC)
et limités aux 100 premiers résultats (LIMIT 100).
```

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]->(movie)<-[r2:RATED]-(u2:User)
WHERE u2 <> u1
WITH u1, u2, collect(r1.rating) AS u1Ratings, collect(r2.rating) AS u2Ratings
WHERE size(u1Ratings) > 10
RETURN u1.name AS from, u2.name AS to, gds.similarity.cosine(u1Ratings, u2Ratings) AS
similarity
ORDER BY similarity DESC
```

Explication

```
Cette requête Cypher est utilisée pour calculer la similarité cosinus entre les
évaluations de films de l'utilisateur 'Cynthia Freeman' et celles des autres
utilisateurs. Voici une explication détaillée :

1. Première partie de la requête:

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

- On commence par sélectionner l'utilisateur 'Cynthia Freeman' (u1) et tous les
utilisateurs (u2) qui ont noté les mêmes films (movie).
- [r1:RATED]->(movie)<-[r2:RATED]- représente les relations où u1 a noté le film (r1)
et u2 a noté le même film (r2).

2. Deuxième partie de la requête:

WITH u1, u2, collect(r1.rating) AS u1Ratings, collect(r2.rating) AS u2Ratings
WHERE size(u1Ratings) > 10

- On regroupe les évaluations données par u1 dans u1Ratings et celles données par u2
dans u2Ratings.
- WHERE size(u1Ratings) > 10 filtre pour ne garder que les utilisateurs qui ont noté
plus de 10 films en commun avec u1.

3. Troisième partie de la requête:

RETURN u1.name AS from, u2.name AS to, gds.similarity.cosine(u1Ratings, u2Ratings) AS
similarity
ORDER BY similarity DESC

- On retourne le nom de u1 (from), le nom de u2 (to), et la similarité cosinus entre
les évaluations de u1 et u2 calculée à l'aide de la fonction gds.similarity.cosine.
- La similarité cosinus mesure la similarité entre deux vecteurs de notes (dans ce
cas, les évaluations de films), en prenant en compte l'angle entre eux dans l'espace
vectoriel.
- Les résultats sont ordonnés par similarité cosinus décroissante (ORDER BY similarity
```

```
DESC), ce qui place les utilisateurs les plus similaires à u1 en premier.
```

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:User {name:"Cynthia Freeman"})-[r:RATED]->(m:Movie)
WITH u1, avg(r.rating) AS u1_mean
MATCH (u1)-[r1:RATED]->(m:Movie)<-[r2:RATED]-(u2)
WITH u1, u1_mean, u2, collect({r1: r1, r2: r2}) AS ratings
WHERE size(ratings) > 10
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
RETURN u1.name, u2.name, nom/denom AS pearson
ORDER BY pearson DESC LIMIT 100;
```

Explication

```
Cette requête Cypher est utilisée pour calculer la similarité de Pearson entre
l'utilisateur 'Cynthia Freeman' et d'autres utilisateurs, basée sur les films qu'ils
ont notés en commun. Voici une explication étape par étape :

1. Première partie de la requête:
```

```
MATCH (u1:User {name:"Cynthia Freeman"})-[r:RATED]->(m:Movie)
WITH u1, avg(r.rating) AS u1_mean

- On commence par trouver l'utilisateur 'Cynthia Freeman' (u1) et tous les films (m)
qu'elle a notés, avec les évaluations (r.rating) qu'elle a données à ces films.
- avg(r.rating) AS u1_mean calcule la moyenne des notes que u1 a données aux films
notés.

2. Deuxième partie de la requête:

MATCH (u1)-[r1:RATED]->(m:Movie)<-[r2:RATED]-(u2)
WITH u1, u1_mean, u2, collect({r1: r1, r2: r2}) AS ratings
WHERE size(ratings) > 10

- On cherche maintenant d'autres utilisateurs (u2) qui ont noté les mêmes films (m)
que u1. Les évaluations sont collectées dans ratings, qui comprend à la fois les
évaluations de u1 (r1) et de u2 (r2).
- WHERE size(ratings) > 10 filtre pour ne garder que les utilisateurs qui ont noté
plus de 10 films en commun avec u1.

3. Troisième partie de la requête:

MATCH (u2)-[r:RATED]->(m:Movie)
WITH u1, u1_mean, u2, avg(r.rating) AS u2_mean, ratings

- On récupère les évaluations des films notés par u2 avec les relations [r:RATED].
- avg(r.rating) AS u2_mean calcule la moyenne des notes que u2 a données aux films
notés.

4. Quatrième partie de la requête:

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

- On déroule les évaluations stockées dans ratings.
- On calcule le numérateur (nom) et le dénominateur (denom) de la formule de
similarité de Pearson entre u1 et u2.
- WHERE denom <> 0 s'assure que le dénominateur n'est pas nul pour éviter une division
par zéro.

5. Dernière partie de la requête:

RETURN u1.name, u2.name, nom/denom AS pearson
ORDER BY pearson DESC LIMIT 100;

-On retourne les noms de u1 et u2 ainsi que le coefficient de similarité de Pearson
entre eux (nom/denom).
- Les résultats sont ordonnés par similarité décroissante (ORDER BY pearson DESC) et
```

```
limités aux 100 premiers résultats (LIMIT 100).
```

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'})-[x:RATED]->(movie)<-[x2:RATED]-(u2:User)
WHERE u2 <> u1
WITH u1, u2, collect(x.rating) AS u1Ratings, collect(x2.rating) AS u2Ratings
WHERE size(u1Ratings) > 10
RETURN u1.name AS from, u2.name AS to, gds.similarity.pearson(u1Ratings, u2Ratings) AS
similarity
ORDER BY similarity DESC
```

Explication :

```
Cette requête Cypher permet de calculer la similarité de Pearson entre les évaluations
données par l'utilisateur 'Cynthia Freeman' et les autres utilisateurs, basées sur les
films qu'ils ont notés en commun.

Voici une explication étape par étape :

MATCH (u1:User {name: 'Cynthia Freeman'})-[x:RATED]->(movie)<-[x2:RATED]-(u2:User)

On sélectionne l'utilisateur nommé 'Cynthia Freeman' (u1) et tous les utilisateurs
(u2) qui ont noté les mêmes films (movie).
Les relations [x:RATED] et [x2:RATED] représentent les évaluations des films par u1 et
u2 respectivement.
WHERE u2 <> u1

On s'assure que u2 n'est pas le même que u1 pour éviter de comparer un utilisateur
avec lui-même.
WITH u1, u2, collect(x.rating) AS u1Ratings, collect(x2.rating) AS p2Ratings

On regroupe les évaluations données par u1 dans p1Ratings et celles données par u2
dans p2Ratings.
WHERE size(p1Ratings) > 10

On filtre les paires d'utilisateurs pour ne garder que celles où u1 a noté plus de 10
films en commun avec p2.
RETURN u1.name AS from, u2.name AS to, gds.similarity.pearson(u1Ratings, u2Ratings) AS
similarity

On retourne le nom de u1 (from), le nom de u2 (to), et la similarité de Pearson entre
les évaluations de u1 et u2 calculée à l'aide de la fonction gds.similarity.pearson.
ORDER BY similarity 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 (u1:User {name:"Cynthia Freeman"})-[r:RATED]->(m:Movie)
WITH u1, avg(r.rating) AS u1_moyen
MATCH (u1)-[r1:RATED]->(m:Movie)<-[r2:RATED]-(u2)
WITH u1, u1_moyen, u2, COLLECT({r1: r1, r2: r2}) AS ratings WHERE size(ratings) > 10
MATCH (u2)-[r:RATED]->(m:Movie)
WITH u1, u1_moyen, u2, avg(r.rating) AS u2_moyen, ratings
UNWIND ratings AS r
WITH sum( (r.r1.rating-u1_moyen) * (r.r2.rating-u2_moyen) ) AS nom, sqrt( sum(
(r.r1.rating - u1_moyen)^2) * sum( (r.r2.rating - u2_moyen) ^2)) AS denom, u1, u2
WHERE denom <> 0
WITH u1, u2, nom/denom AS pearson
ORDER BY pearson DESC LIMIT 10
MATCH (u2)-[r:RATED]->(m:Movie) WHERE NOT EXISTS( (u1)-[:RATED]->(m) )
RETURN m.title, SUM( pearson * r.rating) AS score
ORDER BY score DESC LIMIT 25
```

Explication:

```
Cette requête Cypher vise à recommander des films à l'utilisateur 'Cynthia Freeman' en
se basant sur la similarité de goûts avec d'autres utilisateurs. Voici une explication
détaillée :

MATCH (u1:User {name:"Cynthia Freeman"})-[r:RATED]->(m:Movie)

On commence par sélectionner l'utilisateur 'Cynthia Freeman' (u1) et tous les films
(m) qu'elle a notés, avec les relations [r:RATED] représentant les évaluations données
par u1 à ces films.
WITH u1, avg(r.rating) AS u1_moyen

On calcule la moyenne des notes données par u1 à tous les films, qu'on stocke dans
u1_moyen.
MATCH (u1)-[r1:RATED]->(m:Movie)<-[r2:RATED]-(u2)

On cherche d'autres utilisateurs (u2) qui ont noté les mêmes films (m) que u1, et on
```

```
collecte les évaluations données par u1 (r1) et par u2 (r2).
WITH u1, u1_moyen, u2, COLLECT({r1: r1, r2: r2}) AS ratings WHERE size(ratings) > 10

On regroupe ces évaluations dans ratings et on filtre pour ne garder que les paires
d'utilisateurs qui ont noté plus de 10 films en commun.
MATCH (u2)-[r:RATED]->(m:Movie)

On récupère les évaluations des films notés par u2, avec les relations [r:RATED].
WITH u1, u1_moyen, u2, avg(r.rating) AS u2_moyen, ratings

On calcule la moyenne des notes données par u2 à ses films et on garde les
informations précédentes dans ratings.
UNWIND ratings AS r

On déroule les évaluations stockées dans ratings pour les utiliser individuellement.
WITH sum( (r.r1.rating-u1_moyen) * (r.r2.rating-u2_moyen) ) AS nom, sqrt( sum(
(r.r1.rating - u1_moyen)^2) * sum( (r.r2.rating - u2_moyen) ^2)) AS denom, u1, u2
WHERE denom <> 0

On calcule les composants numérateur (nom) et dénominateur (denom) de la formule de
similarité de Pearson entre u1 et u2, en s'assurant que denom n'est pas nul pour
éviter une division par zéro.
WITH u1, u2, nom/denom AS pearson

On calcule la similarité de Pearson entre u1 et u2, qu'on stocke dans pearson.
ORDER BY pearson DESC LIMIT 10

On trie les résultats par ordre décroissant de similarité de Pearson et on limite à 10
résultats pour obtenir les utilisateurs les plus similaires à u1.
MATCH (u2)-[r:RATED]->(m:Movie) WHERE NOT EXISTS( (u1)-[:RATED]->(m) )

On sélectionne les films notés par u2 mais pas encore notés par u1.
RETURN m.title, SUM( pearson * r.rating) AS score

On retourne le titre des films et on calcule un score pondéré (SUM( pearson * r.rating
)) en multipliant la similarité de Pearson (pearson) par la note donnée par u2
(r.rating).
ORDER BY score DESC LIMIT 25

On ordonne les résultats par score décroissant et on limite à 25 films recommandés à
u1.
```

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.

Réponse

Pour étendre les requêtes Neo4j et Cypher en tenant compte du composant temporel, on peut utiliser des propriétés de type `timestamp` pour les relations `RATED` L'idée est d'incorporer les informations temporelles pour ajuster les recommandations en fonction des évaluations les plus récentes. Voici comment on peut aborder cela :

- Ajout de Timestamp aux Évaluations

Lors de l'ajout d'une évaluation par un utilisateur, on inclue un timestamp :

```
MATCH (u:User {name: 'Angelica Rodriguez'}), (m:Movie {title: 'Inception'})
CREATE (u)-[r:RATED {rating: 5, timestamp: timestamp()}]->(m)
```

timestamp() : Retourne le timestamp actuel en millisecondes.

- Recherche des Films Évalués Récemment

Pour trouver les films évalués par un utilisateur avec les évaluations les plus récentes :

```
MATCH (u:User {name: 'Angelica Rodriguez'})-[r:RATED]->(m:Movie)
RETURN m.title, r.rating, r.timestamp
ORDER BY r.timestamp DESC
LIMIT 5
```

- Recommandations Basées sur les Évaluations Récentes

Pour recommander des films en se basant sur les évaluations récentes d'un utilisateur :

```
MATCH (u:User {name: 'Angelica Rodriguez'})-[r:RATED]->(m:Movie)<-[:ACTED_IN]-
(a:Actor)-[:ACTED_IN]->(rec:Movie)
WHERE r.timestamp > timestamp() - 604800000  // Les 7 derniers jours
AND NOT (u)-[:RATED]->(rec)
RETURN rec.title, COUNT(a) AS commonActors, AVG(r.rating) AS avgRating
ORDER BY avgRating DESC, commonActors DESC
LIMIT 5
```

- Utilisation des Périodes pour Pondérer les Évaluations

Pour pondérer les évaluations en fonction de leur récence :

```
MATCH (u:User {name: 'Angelica Rodriguez'})-[r:RATED]->(m:Movie)<-[:ACTED_IN]-
(a:Actor)-[:ACTED_IN]->(rec:Movie)
WHERE NOT (u)-[:RATED]->(rec)
WITH rec, r.rating, r.timestamp, (timestamp() - r.timestamp) AS age
RETURN rec.title, SUM(r.rating / age) AS weightedRating
```

```
ORDER BY weightedRating DESC
LIMIT 5
```

- Filtrage Basé sur une Période Spécifique

Pour récupérer des évaluations faites dans une période spécifique, par exemple, le dernier mois :

```
MATCH (u:User {name: 'Angelica Rodriguez'})-[r:RATED]->(m:Movie)
WHERE r.timestamp > timestamp() - 2592000000  // Les 30 derniers jours
RETURN m.title, r.rating, r.timestamp
ORDER BY r.timestamp DESC
```

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

Extraction des Mots-Clés

Outils et Techniques :

- on utilise des techniques de traitement du langage naturel (NLP) telles que TF-IDF, l'algorithme RAKE, ou des modèles basés sur des réseaux neuronaux comme BERT pour extraire les mots-clés pertinents des descriptions de films.

Exemple de Code : Voici un exemple d'extraction de mots-clés en utilisant Python et la bibliothèque RAKE :

```
import RAKE

rake = RAKE.Rake('stopwords.txt')
description = "Inception is a science fiction movie that explores the concept of dream
invasion and manipulation."
keywords = rake.run(description)

print(keywords)
```

Modélisation des Mots-Clés dans Neo4j

Création du Schéma:

- Ajoutons une nouvelle étiquette Keyword pour modéliser les mots-clés.
- Créons une relation HAS_KEYWORD entre les nœuds Movie et les nœuds Keyword.

Exemple de Requêtes Cypher:

- Ajoutons des Mots-Clés à un Film
- Créons des Mots-Clés et les Relier à des Films :

```
// Supposons que nous avons les mots-clés extraits sous forme de liste
WITH ["science fiction", "dream invasion", "manipulation"] AS keywords
MATCH (m:Movie {title: 'Inception'})
UNWIND keywords AS keyword
MERGE (k:Keyword {name: keyword})
MERGE (m)-[:HAS_KEYWORD]->(k)
```

Requêtes pour Utiliser les Mots-Clés Rechercher des Films par Mots-Clés :

```
MATCH (m:Movie)-[:HAS_KEYWORD]->(k:Keyword)
WHERE k.name IN ["science fiction", "manipulation"]
RETURN m.title, COLLECT(k.name) AS keywords
```

Recommander des Films Basés sur des Mots-Clés Similaires :

```
MATCH (u:User {name: 'Angelica Rodriguez'})-[:RATED]->(m:Movie)-[:HAS_KEYWORD]-
>(k:Keyword)<-[:HAS_KEYWORD]-(rec:Movie)
WHERE NOT (u)-[:RATED]->(rec)
RETURN rec.title, COUNT(k) AS commonKeywords
ORDER BY commonKeywords DESC
LIMIT 5
```

Enrichissement des Propriétés des Films

Enrichir les nœuds Movie avec des propriétés supplémentaires dérivées des mots-clés :

- Genres additionnels : En ajoutant des mots-clés comme propriétés, cela permet de préciser davantage le genre du film.
- Thèmes principaux : Identifier les thèmes principaux à partir des mots-clés et les ajouter comme propriétés au nœud Movie.

Gestion et Mise à Jour des Mots-Clés

- Mettre en place des mécanismes pour mettre à jour régulièrement les mots-clés des films en fonction des nouvelles descriptions ou des modifications apportées aux anciennes descriptions :

Requêtes pour Mettre à Jour les Mots-Clés :

```
MATCH (m:Movie {title: 'Inception'})-[r:HAS_KEYWORD]->(k:Keyword)
DELETE r
WITH m
// Ajouter les nouveaux mots-clés après suppression des anciens
WITH ["nouveau mot-clé 1", "nouveau mot-clé 2"] AS newKeywords, m
UNWIND newKeywords AS newKeyword
MERGE (k:Keyword {name: newKeyword})
MERGE (m)-[:HAS_KEYWORD]->(k)
```

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

Pour enrichir notre projet Neo4j et Cypher avec la reconnaissance d'image, on peut utiliser diverses bibliothèques et APIs pour extraire des informations des affiches de films et les intégrer dans notre base de données Neo4j. Voici comment procéder.

Exemple de bibliothèques et APIs Populaires pour la Reconnaissance d'Image

- TensorFlow et Keras (Python) :Pour la reconnaissance d'image avec des modèles pré-entraînés.
- OpenCV (Python) : Pour la détection de caractéristiques et le traitement d'image.
- API Google Cloud Vision :Pour une reconnaissance d'image avancée et des annotations automatiques.
- API Microsoft Azure Computer Vision : Pour des services similaires à Google Cloud Vision.
- API Amazon Rekognition : Pour une reconnaissance d'image puissante et flexible.

Exemple d'Utilisation avec Google Cloud Vision API

Pour utiliser l'API Google Cloud Vision pour analyser des affiches de films, voici les étapes à suivre :

1. Configuration de Google Cloud Vision
    - Créons un projet sur Plateforme Google Cloud
    - Activons l'API Cloud Vision.
    - Configurons l'authentification en téléchargeant un fichier JSON de clé de service.
2. Installation de la Bibliothèque Google Cloud Installons la bibliothèque Google Cloud pour Python :

```
!pip install google-cloud-vision
```

```
import io
from google.cloud import vision

# Initialiser le client Google Cloud Vision
client = vision.ImageAnnotatorClient()

# Charger l'image d'une affiche
with io.open('path_to_your_movie_poster.jpg', 'rb') as image_file:
    content = image_file.read()
    image = vision.Image(content=content)

# Envoyer l'image à l'API et recevoir les étiquettes
response = client.label_detection(image=image)
labels = response.label_annotations

# Afficher les résultats
for label in labels:
```

```
    print(f"{label.description}: {label.score*100:.2f}%")
```

Intégration des Résultats dans Neo4j

Une fois qu'on a extrait les informations des affiches de films, on peut les intégrer dans la base de données `recommendations` en utilisant Cypher.

a.Ajoutons des Mots-Clés Extraits comme Nœuds et Relations Supposons qu'on a extrait des mots-clés tels que "science fiction", "action", et "thriller" pour un film. On peut les ajouter à Neo4j comme suit :

```
// Créer un nœud pour le film
MERGE (m:Movie {title: 'Inception'})

// Créer des nœuds pour les mots-clés et les relier au film
WITH ['science fiction', 'action', 'thriller'] AS keywords, m
UNWIND keywords AS keyword
MERGE (k:Keyword {name: keyword})
MERGE (m)-[:HAS_KEYWORD]->(k)
```

b.Requêtes Cypher pour Utiliser les Mots-Clés On peut maintenant utiliser ces mots-clés pour enrichir nos requêtes et recommandations.

Exemple de Requête pour Trouver des Films par Mots-Clés :

```
MATCH (m:Movie)-[:HAS_KEYWORD]->(k:Keyword)
WHERE k.name IN ['science fiction', 'action']
RETURN m.title, COLLECT(k.name) AS keywords
```

References

- https://neo4j.com/deployment-center/#gds-tab
- https://neo4j.com/docs/graph-data-science/current/installation/neo4j-server/
- https://neo4j.com/docs/graph-data-science/current/algorithms/knn/
- https://neo4j.com/docs/graph-data-science/current/algorithms/similarity-functions/
- https://neo4j.com/docs/cypher-cheat-sheet/5/neo4j-enterprise
- https://github.com/neo4j/graph-data-science/releases
- https://github.com/neo4j-contrib/neo4j-apoc-procedures/releases

[Discourse users]

[Discord]