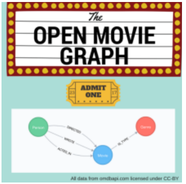
Advanced Data & Network Mining

Recommenders 2023-24

terri.hoare@dbs.ie

Graph-Based Recommenders

Personalised Product Recommendations

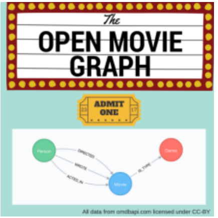


Recommendations

Personalised product recommendations can increase conversions, improve sales rates and provide a better experience for users. We will explorehow 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 Recommenders

Personalised Product Recommendations



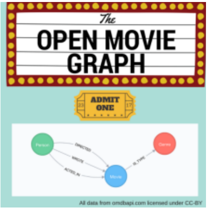
Graph-Based Recommendations

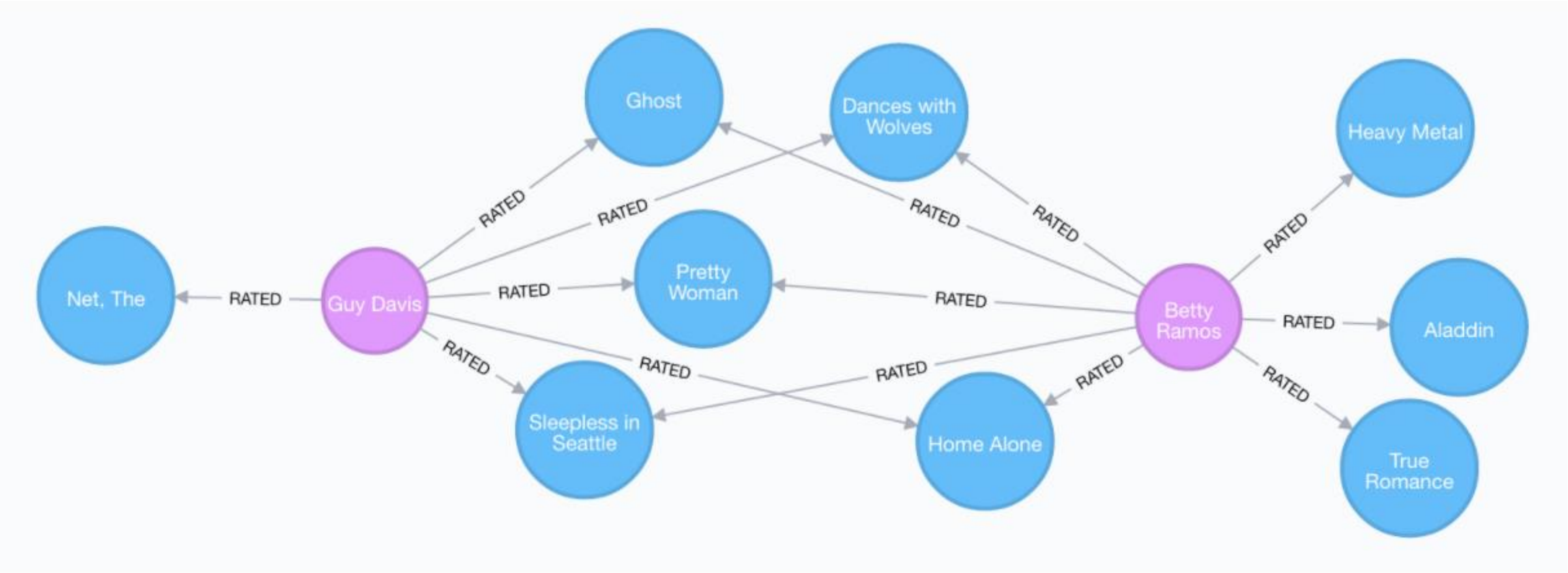
Generating **personalised recommendations** is one of the most common use cases for a graph database. Some of the main benefits of using graphs to generate recommendations include:

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 labelled property graph model allows for easily combining datasets from multiple sources, allowing enterprises to **unlock value from previously separated data silos.**

Graph-Based Recommenders

Personalised Product Recommendations

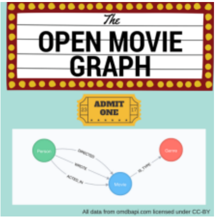




Sources: - Open Movie Database http://www.omdbapi.com/ and MovieLens dataset https://grouplens.org/datasets/movielens/

Graph-Based Recommenders

Personalised Product Recommendations



Eliminate Data Silos

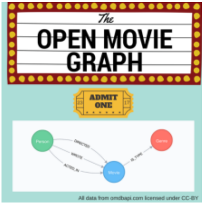
In this use case, we are using graphs to combine data from multiple silos.

- **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 silo.

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

Graph-Based Recommenders

Personalised Product Recommendations



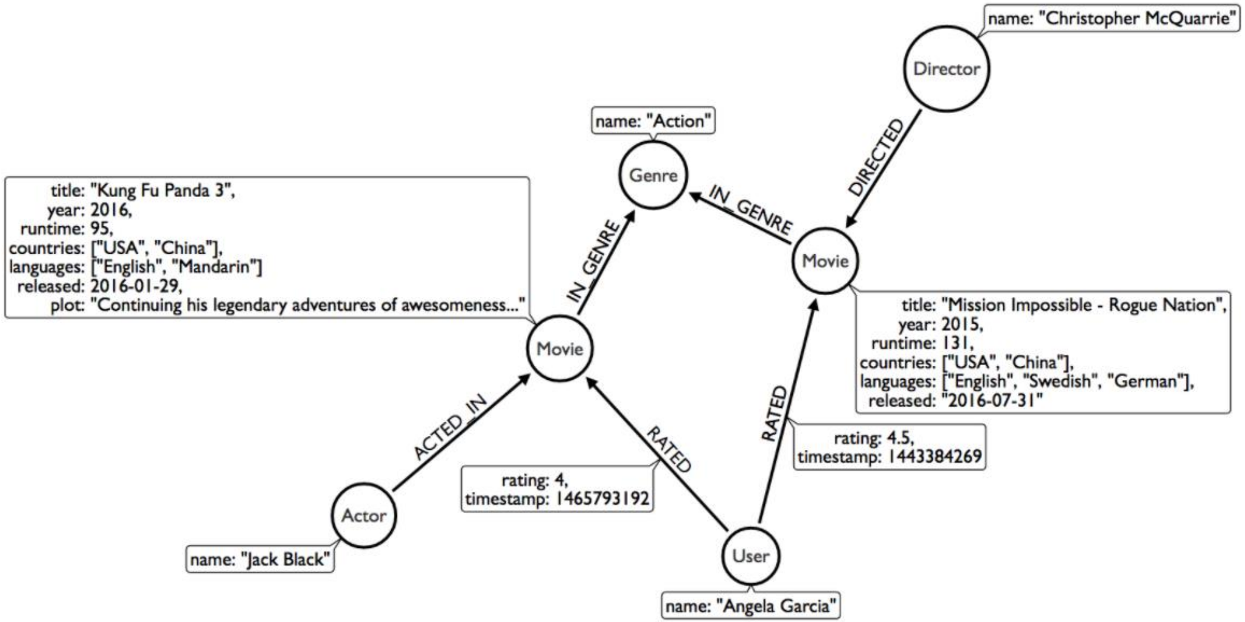
The Property Graph Model

The data model of graph databases is called the labelled 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.

Personalised Product Recommendations

Data Model



Nodes

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

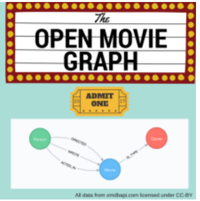
Relationships

`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.

Personalised Product Recommendations



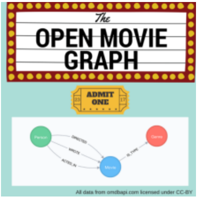
Intro to Cypher

Graph Patterns
Cypher is the query language for graphs and is centred 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: **(:Movie)<-[:RATED]-(:User)**

Personalised Product Recommendations



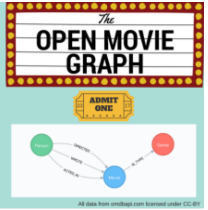
Intro to Cypher

Graph Patterns
Cypher is the query language for graphs and is centred 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: **(:Movie)<-[:RATED]-(:User)**

Personalised Product Recommendations



Intro to Cypher

Aliases

Graph elements can be bound to aliases that can be referred to later in the query: **(m:Movie)<-[r:RATED]-(u:User)**

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.

Aggregations

There is an implicit group when using aggregation functions such as **COUNT**

Ref https://neo4j.com/docs/cypher-refcard/current/?ref=browser-guide

Personalised Product Recommendations

Intro to Cypher

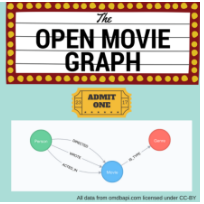
Aliases
Graph elements can be bound to aliases that can be referred to later in the query: **(m:Movie)<-[r:RATED]-(u:User)**

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.

Aggregations
There is an implicit group when using aggregation functions such as **COUNT**

Ref **https://neo4j.com/docs/cypher-refcard/current/?ref=browser-guide**

Personalised Product Recommendations



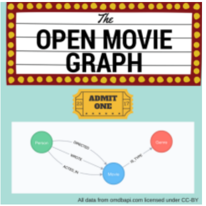
Intro to Cypher

Dissecting a Cypher Statement

Let’s look at a Cypher query that answers the question "How many reviews does each Matrix movie have?"

```
MATCH (m:Movie)←[:RATED]-(u:User)
WHERE m.title CONTAINS "Matrix"
WITH m.title AS movie, COUNT(*) AS reviews
RETURN movie, reviews
ORDER BY reviews DESC
LIMIT 5;
```

Personalised Product Recommendations



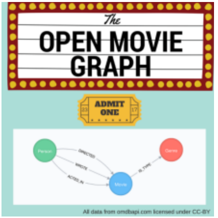
DBS Dublin Business School

Intro to Cypher

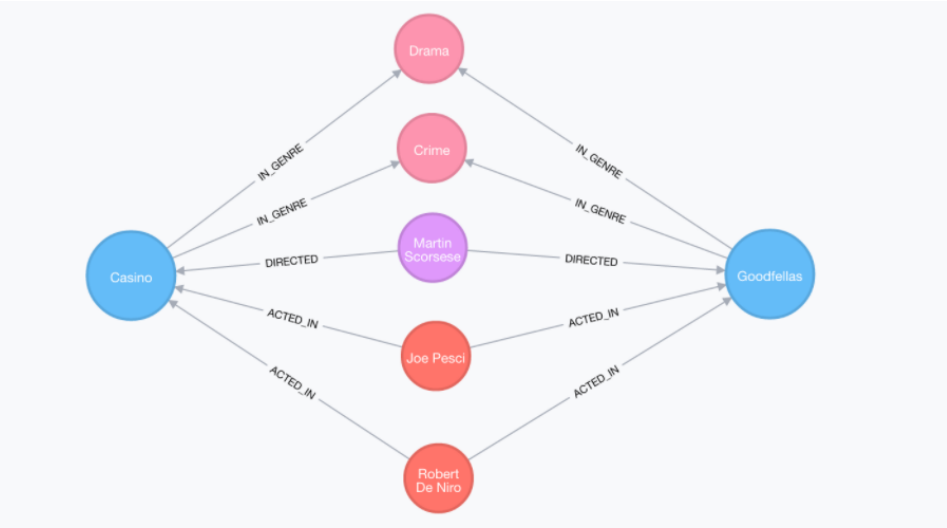
```
MATCH (m:Movie)←[:RATED]-(u:User)
WHERE m.title CONTAINS "Matrix"
WITH m.title AS movie, COUNT(*) AS reviews
RETURN movie, reviews
ORDER BY reviews DESC
LIMIT 5;
```

find	`MATCH (m:Movie)←[:RATED]-(u:User)`	Search for an existing graph pattern
filter	`WHERE m.title CONTAINS "Matrix"`	Filter matching paths to only those matching a predicate
aggregate	`WITH m.title AS movie, COUNT(*) AS reviews`	Count number of paths matched for each movie
return	`RETURN movie, reviews`	Specify columns to be returned by the statement
order	`ORDER BY reviews DESC`	Order by number of reviews, in descending order
limit	`LIMIT 5;`	Only return first five records

Personalised Product Recommendations



Content-Based Filtering



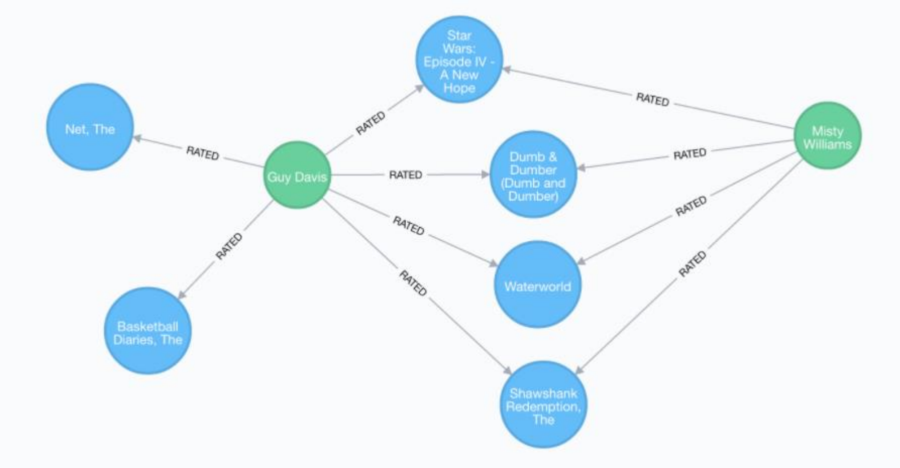
Recommend items that are similar to those that a user is viewing, rated highly or purchased previously. "Products similar to the product you're looking at now."

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

Neo4j Sandbox Case Study

Personalised Product Recommendations

Collaborative Filtering



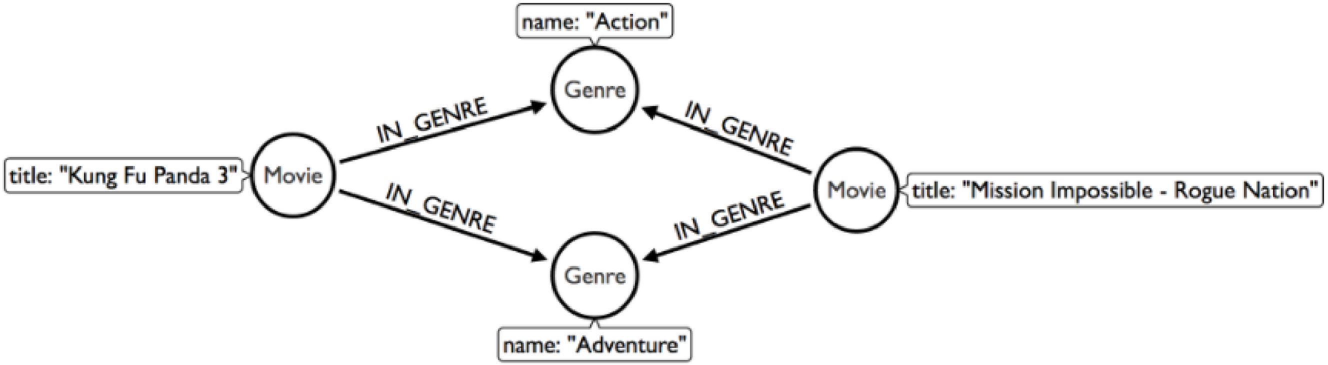
Use the preferences, ratings and actions of other users in the network to find items to recommend. "Users who bought this thing, also bought that other thing."

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

Personalised Product Recommendations

Content-Based Filtering

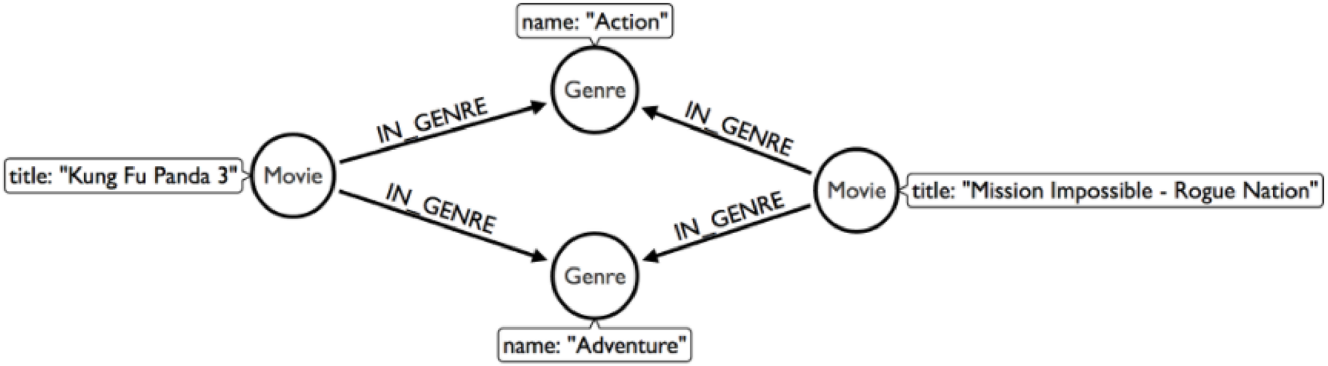
The goal of **content-based** filtering is to find similar items, using attributes (or traits) of the item. Using our movie data, one way we could define similarity is movies that have common genres.



Personalised Product Recommendations

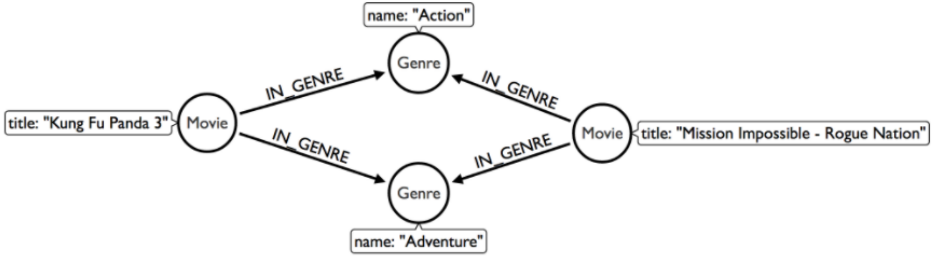
Content-Based Filtering

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 similarity is movies that have common genres.



Personalised Product Recommendations

Content-Based Filtering



Similarity Based on Common Genres

Find movies most similar to Inception based on shared genres.

```
// Find similar movies by common 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
ORDER BY commonGenres DESC LIMIT 10;
```

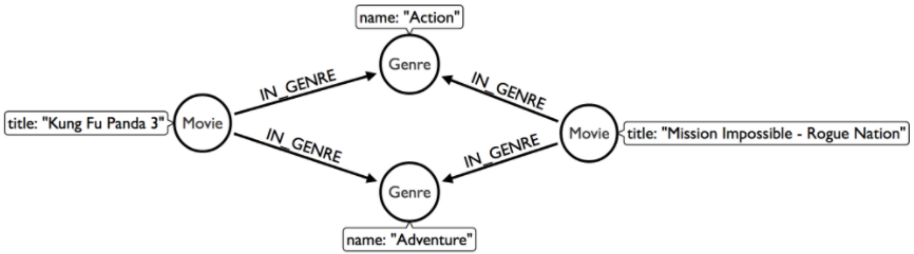
Personalised Product Recommendations

Content-Based Filtering

Personalized Recommendations Based on Genres

If we know what movies a user has watched, we can use this information to recommend similar movies: Recommend movies similar to those the user has already watched.

```
// Content recommendation by overlapping genres
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, COUNT(*)] AS scores
RETURN rec.title AS recommendation, rec.year AS year,
COLLECT(scores) AS scoreComponents,
REDUCE (s=0,x in COLLECT(scores) | s+x[1]) AS score
ORDER BY score DESC LIMIT 10
```



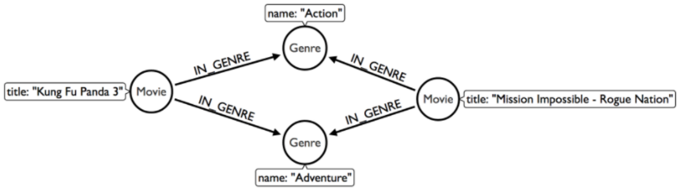
Personalised Product Recommendations

Content-Based Filtering

Weighted Content Algorithm

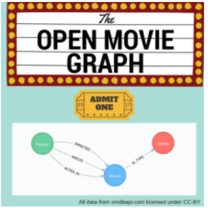
There are many more traits in addition to just genre that we can consider to compute similarity, such actors and directors. Use a weighted sum to score the recommendations based on the number of actors, genres and directors they have in common to boost the score: Compute a weighted sum based on the number and types of overlapping traits

```
// Find similar movies by common genres
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:Actor)-[:ACTED_IN]→(rec)
WITH m, rec, gs, COUNT(a) AS as

OPTIONAL MATCH (m)←[:DIRECTED]-(d:Director)-[: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 100
```



Neo4j Sandbox Case Study

Personalised Product Recommendations



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.

Personalised Product Recommendations

Content-Based Similarity Metrics

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.

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|}$$

Personalised Product Recommendations

Content-Based Similarity Metrics

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

```
MATCH (m:Movie {title: "Inception"})-[:IN_GENRE]->(g:Genre)<-[:IN_GENRE]-(other:Movie)
WITH m, other, COUNT(g) AS intersection, COLLECT(g.name) AS i
MATCH (m)-[:IN_GENRE]->(mg:Genre)
WITH m,other, intersection,i, COLLECT(mg.name) AS s1
MATCH (other)-[:IN_GENRE]->(og:Genre)
WITH m,other,intersection,i, s1, COLLECT(og.name) AS s2

WITH m,other,intersection,s1,s2

WITH m,other,intersection,s1+[x IN s2 WHERE NOT x IN s1] AS union, s1, s2

RETURN m.title, other.title, s1,s2,((1.0*intersection)/SIZE(union)) AS jaccard ORDER BY jaccard DESC
LIMIT 100
```

Personalised Product Recommendations

Content-Based Similarity Metrics

We can 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 i
MATCH (m)-[:IN_GENRE|ACTED_IN|DIRECTED]-(mt)
WITH m,other, intersection,i, COLLECT(mt.name) AS s1
MATCH (other)-[:IN_GENRE|ACTED_IN|DIRECTED]-(ot)
WITH m,other,intersection,i, s1, COLLECT(ot.name) AS s2

WITH m,other,intersection,s1,s2

WITH m,other,intersection,s1+[x IN s2 WHERE NOT x IN s1] AS union, s1, s2
RETURN m.title, other.title, s1,s2,((1.0*intersection)/SIZE(union)) AS jaccard ORDER BY jaccard DESC
LIMIT 100
```

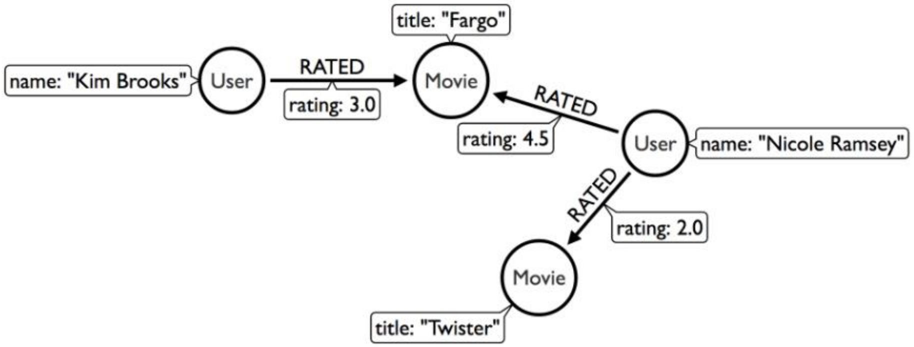
Personalised Product Recommendations

Collaborative Filtering – leveraging 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.
2. Assuming that similar users have similar preferences, what are the movies those similar users like?

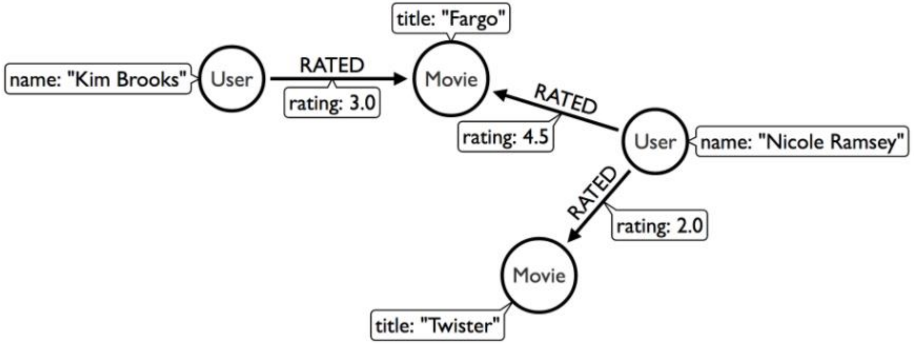


Personalised Product Recommendations

Collaborative Filtering – leveraging Ratings

Show all ratings by Misty Williams:

```
// Show all ratings by Misty Williams
MATCH (u:User {name: "Misty Williams"})
MATCH (u)-[r:RATED]->(m:Movie)
RETURN *;
```

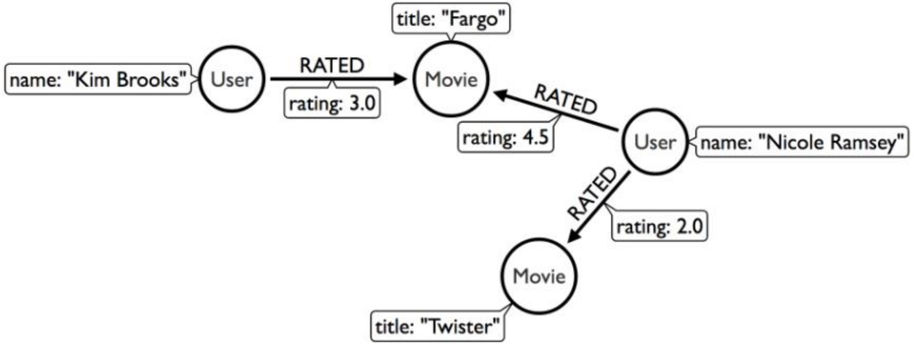


Personalised Product Recommendations

Collaborative Filtering – leveraging Ratings

Find Misty's average rating:

```
// Show all ratings by Misty Williams
MATCH (u:User {name: "Misty Williams"})
MATCH (u)-[r:RATED]→(m:Movie)
RETURN avg(r.rating) AS average;
```

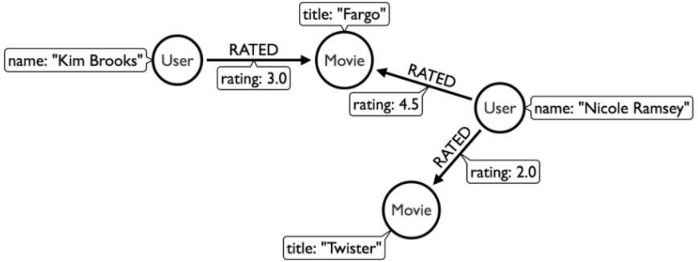


Personalised Product Recommendations

Collaborative Filtering – Wisdom of Crowds

Simple Collaborative Filtering. Limitations: not normalising based on popularity or taking ratings into consideration. This approach can be improved using the **kNN method**.

```
MATCH (u:User {name: "Cynthia Freeman"})-[:RATED]→(:Movie)←[:RATED]-(o:User)
MATCH (o)-[:RATED]→(rec:Movie)
WHERE NOT EXISTS( (u)-[:RATED]→(rec) )
RETURN rec.title, rec.year, rec.plot
LIMIT 25
```



Neo4j Sandbox Case Study

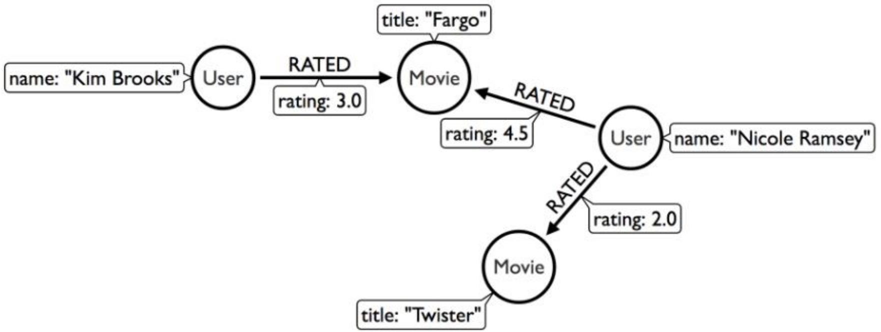
Personalised Product Recommendations

Collaborative Filtering – Wisdom of Crowds

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, COLLECT(DISTINCT g.name) AS genres, SUM(score) AS sscore
ORDER BY sscore DESC LIMIT 10
```



Personalised Product Recommendations

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 explore 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. 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.

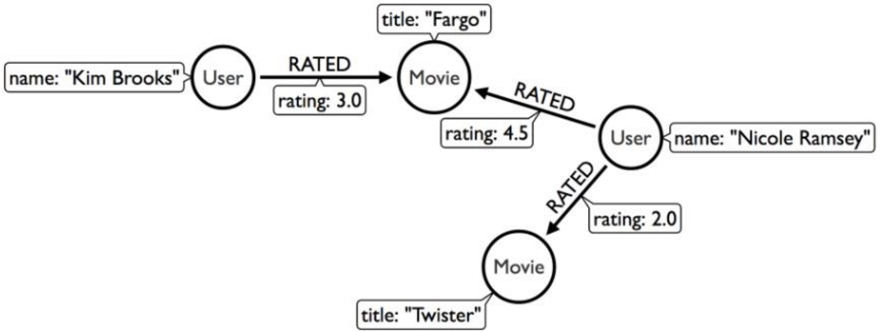
$$similarity(A,B) = \frac{A \cdot B}{\|A\| \times \|B\|} = \frac{\sum_{i=1}^{n} A_i \times B_i}{\sqrt{\sum_{i=1}^{n} A_i^2} \times \sqrt{\sum_{i=1}^{n} B_i^2}}$$

Personalised Product Recommendations

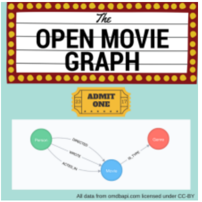
Collaborative Filtering – Similarity Metrics

```
// Most similar users using Cosine similarity
MATCH (p1:User {name: "Cynthia Freeman"})-[x:RATED]->(m:Movie)<-[y:RATED]-(p2:User)
WITH COUNT(m) AS numbermovies, SUM(x.rating * y.rating) AS xyDotProduct,
SQRT(REDUCE(xDot = 0.0, a IN COLLECT(x.rating) | xDot + a^2)) AS xLength,
SQRT(REDUCE(yDot = 0.0, b IN COLLECT(y.rating) | yDot + b^2)) AS yLength,
p1, p2 WHERE numbermovies > 10
RETURN p1.name, p2.name, xyDotProduct / (xLength * yLength) AS sim
ORDER BY sim DESC
LIMIT 100;
```

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

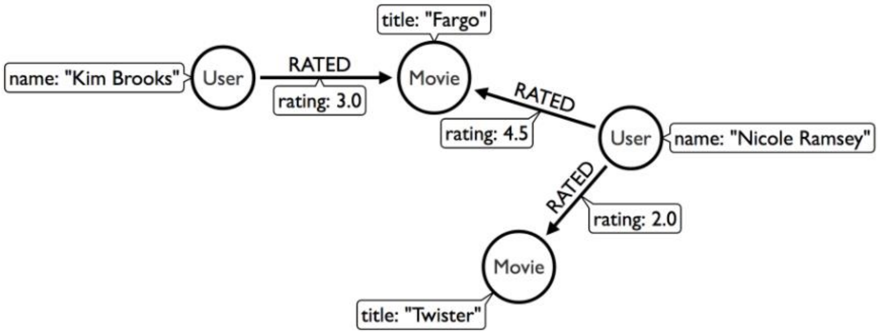


Personalised Product Recommendations



Collaborative Filtering – Similarity Metrics

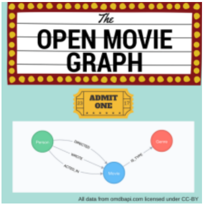
```
MATCH (p1:User {name: 'Cynthia Freeman'})-[x:RATED]->(movie)<-[x2:RATED]-(p2:User)
WHERE p2 <> p1
WITH p1, p2, collect(x.rating) AS p1Ratings, collect(x2.rating) AS p2Ratings
WHERE size(p1Ratings) > 10
RETURN p1.name AS from,
       p2.name AS to,
       gds.alpha.similarity.cosine(p1Ratings, p2Ratings) AS similarity
ORDER BY similarity DESC
```



We can also compute this measure using the [Cosine Similarity algorithm] in the Neo4j Graph Algorithms Library. Find the users with the most similar preferences to Cynthia Freeman, according to cosine similarity function.

Personalised Product Recommendations

Collaborative Filtering – Similarity Metrics



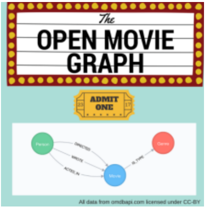
Pearson Similarity

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.

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

Personalised Product Recommendations

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.

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

Personalised Product Recommendations

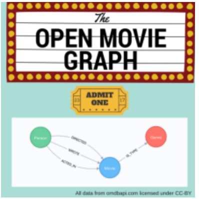
Collaborative Filtering – Similarity Metrics

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
```

Personalised Product Recommendations

Collaborative Filtering – Similarity Metrics

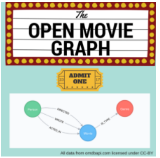


We can also compute this measure using the [Pearson Similarity algorithm] in the Neo4j Graph Algorithms Library. Find users most similar to Cynthia Freeman, according to the Pearson similarity function.

```
MATCH (p1:User {name: 'Cynthia Freeman'})-[x:RATED]->(movie:Movie)
WITH p1, gds.alpha.similarity.asVector(movie, x.rating) AS p1Vector
MATCH (p2:User)-[x2:RATED]->(movie:Movie) WHERE p2 <> p1
WITH p1, p2, p1Vector, gds.alpha.similarity.asVector(movie, x2.rating) AS p2Vector
WHERE size(apoc.coll.intersection([v in p1Vector | v.category], [v in p2Vector | v.category])) > 10
RETURN p1.name AS from,
       p2.name AS to,
       gds.alpha.similarity.pearson(p1Vector, p2Vector, {vectorType: "maps"}) AS similarity
ORDER BY similarity DESC
LIMIT 100
```

Personalised Product Recommendations

Collaborative Filtering – Neighbourhood-Based



kNN – k Nearest Neighbours

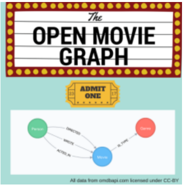
Once we have a method for finding similar users based on preferences, the next step is to allow each of the **k** most similar users to vote for what items should be recommended.

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?"

Personalised Product Recommendations

Collaborative Filtering – Neighbourhood-Based



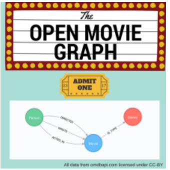
kNN movie recommendation using Pearson similarity.

```
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

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
```

Neo4j Sandbox Case Study

Personalised Product Recommendations

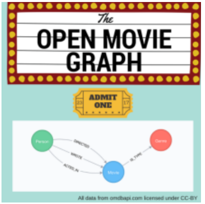
Collaborative Filtering – Neighbourhood-Based



kNN movie recommendation using Pearson similarity function.

```
MATCH (u1:User {name: 'Cynthia Freeman'})-[x:RATED]->(movie:Movie)
WITH u1, gds.alpha.similarity.asVector(movie, x.rating) AS u1Vector
MATCH (u2:User)-[x2:RATED]->(movie:Movie) WHERE u2 <> u1

WITH u1, u2, u1Vector, gds.alpha.similarity.asVector(movie, x2.rating) AS u2Vector
WHERE size(apoc.coll.intersection([v in u1Vector | v.category], [v in u2Vector | v.category])) > 10

WITH u1, u2,  gds.alpha.similarity.pearson(u1Vector, u2Vector, {vectorType: "maps"}) AS similarity
ORDER BY similarity DESC
LIMIT 10

MATCH (u2)-[r:RATED]->(m:Movie) WHERE NOT EXISTS( (u1)-[:RATED]->(m) )
RETURN m.title, SUM( similarity * r.rating) AS score
ORDER BY score DESC LIMIT 25
```

Personalised Product Recommendations



Further Work

- **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 recommendations?