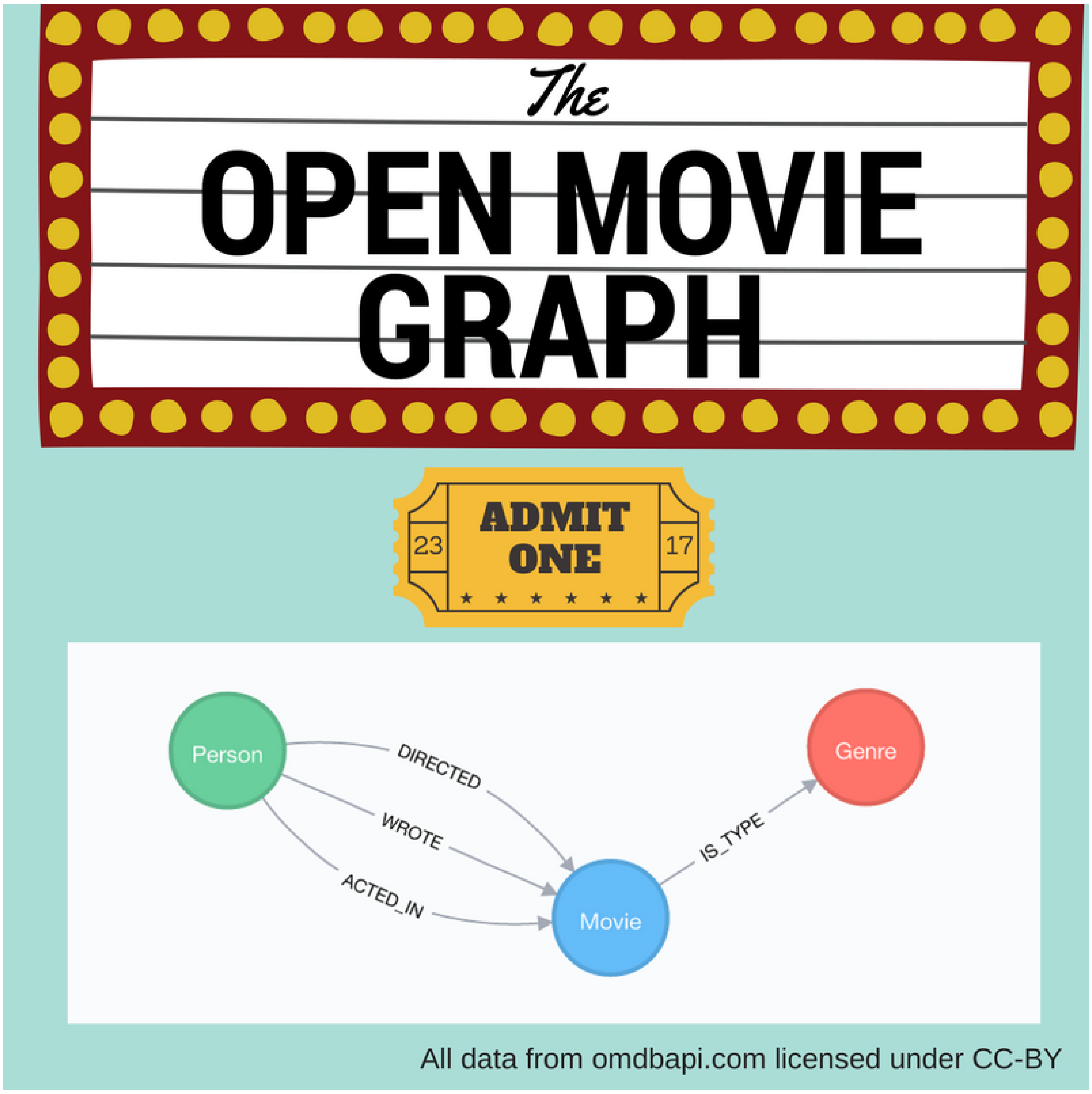
Recommendations

```
<style type="text/css">
* {
  margin-bottom: 0.5em;
}
</style>
```

Personalized Product Recommendations with Neo4j



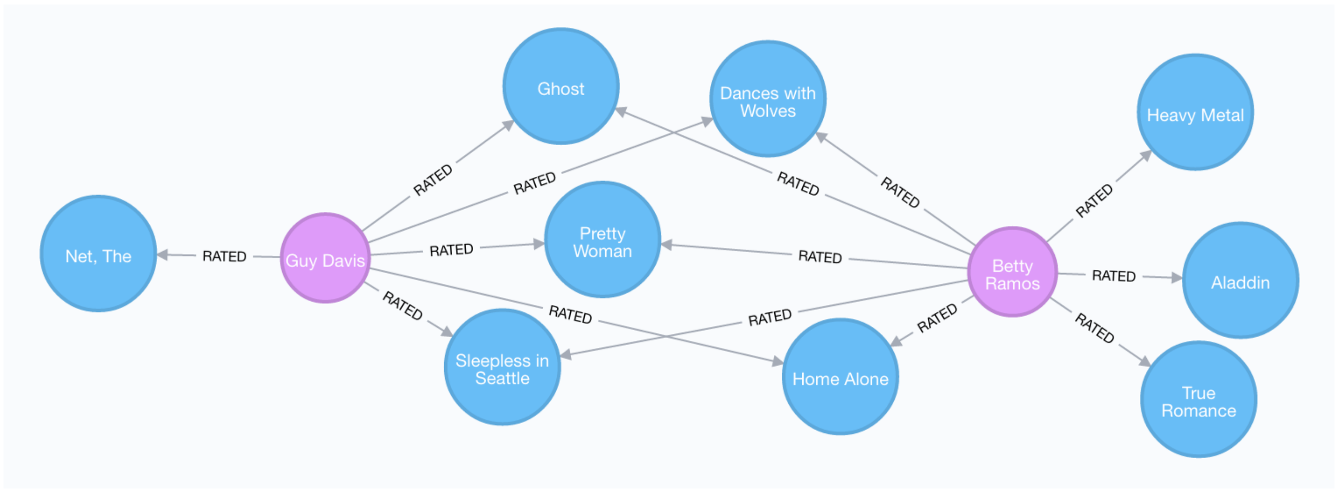
Recommendations

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:

- Open Movie Database
- MovieLens

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.

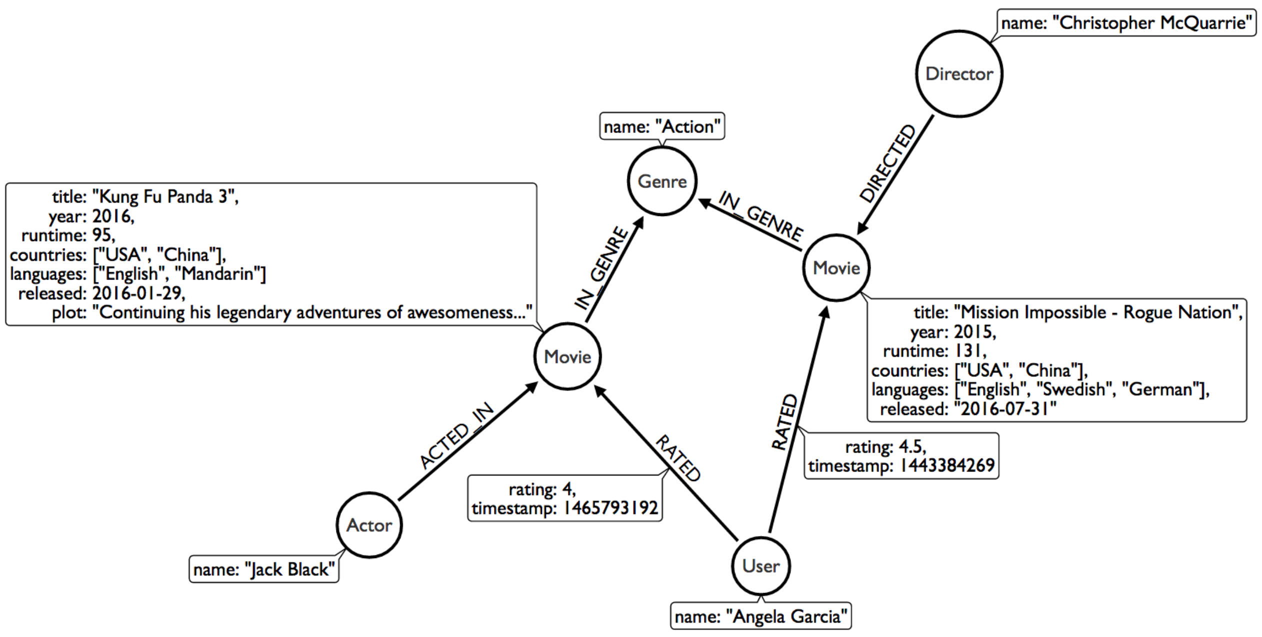
Eliminate Data Silos

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
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..."
IN_GENRE
Movie
title: "Mission Impossible - Rogue Nation",
year: 2015,
runtime: 131,
countries: ["USA", "China"],
languages: ["English", "Swedish", "German"],
released: "2016-07-31"
ACTED_IN
RATED
RATED
rating: 4.5,
timestamp: 1443384269
rating: 4,
timestamp: 1465793192
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.

Intro To 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`.

Take the Cypher Graphacademy courses to learn more. Use the Cypher Refcard as a syntax reference.

Dissecting a Cypher Statement

Let’s look at 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.

How many reviews does each Matrix movie have? Click on the block to put the query in the top-most window on the query editor. Hit the triangular [play circle] button or press `Ctrl` + `Enter` *to run it and see the resulting visualization.*

```
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;
```

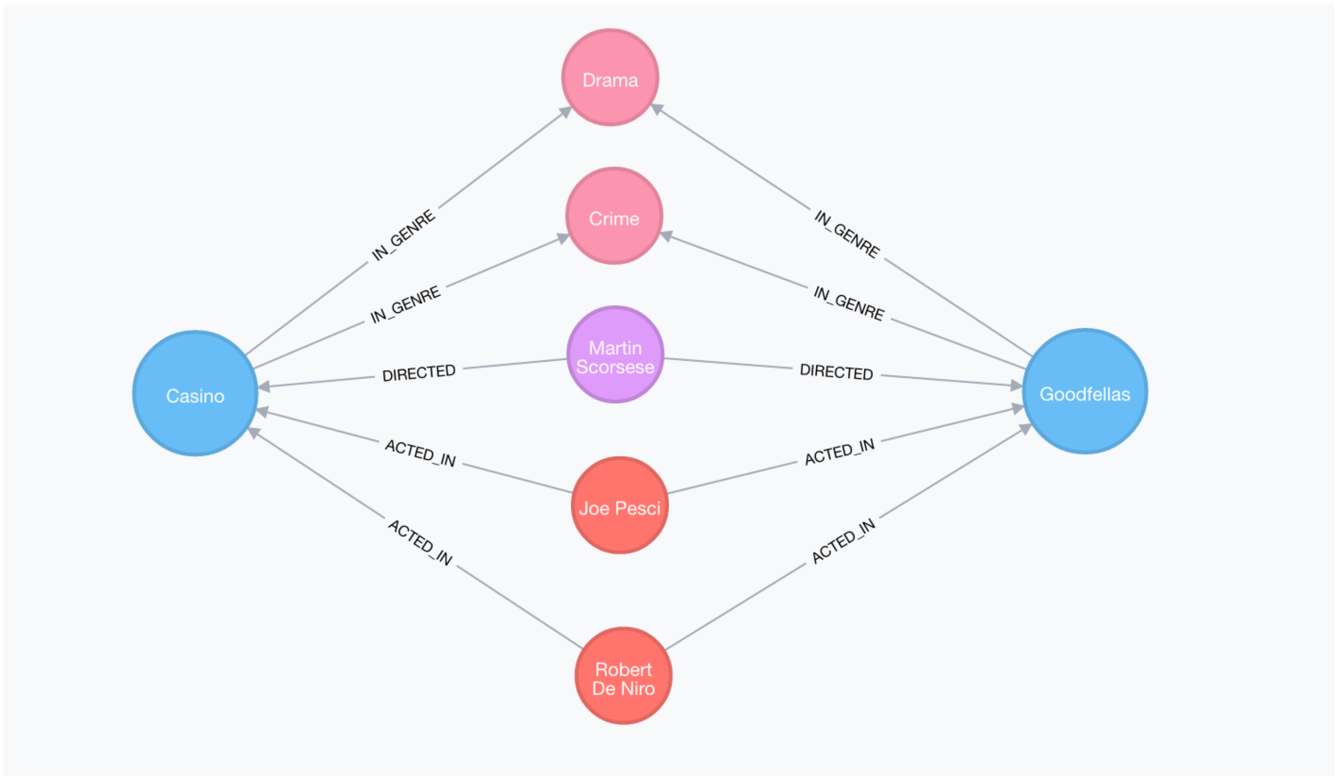
find	`MATCH (m:Movie)<-[:RATED]-(u:User)`	Search for an existing graph pattern
filter	`WHERE m.title CONTAINS "Matrix"`	Filter matching paths to only those matching a predicate
aggregate	`WITH m, count(*) AS reviews`	Count number of paths matched for each movie
return	`RETURN m.title as movie, reviews`	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

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.

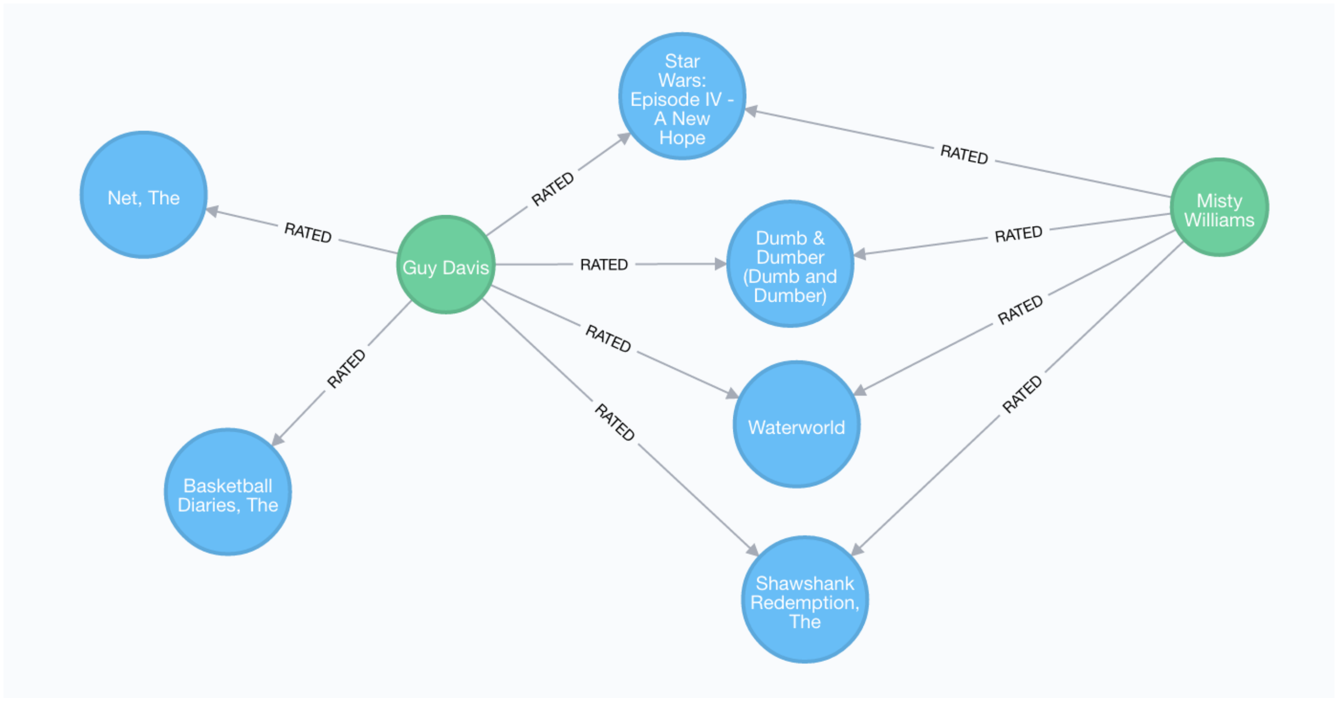


"Items similar to the item you're looking at now"

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

Collaborative Filtering

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

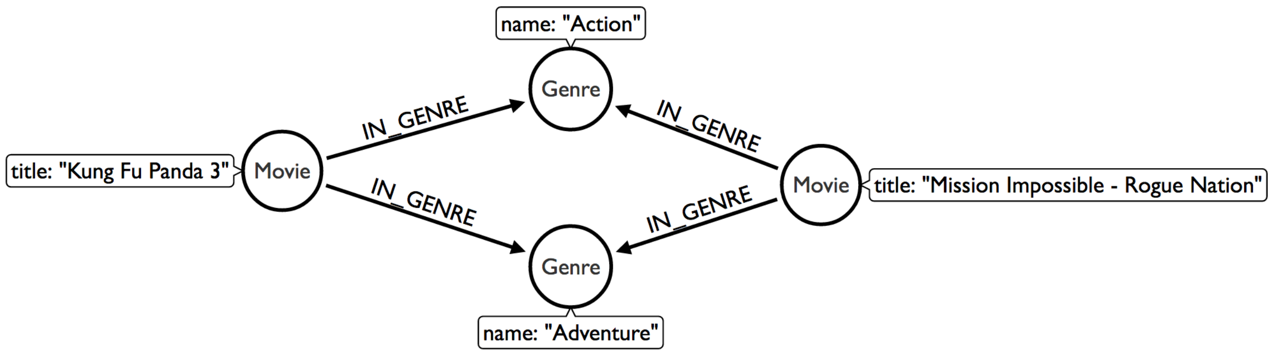

Star Wars: Episode IV - A New Hope
Net, The
Guy Davis
Dumb & Dumber (Dumb and Dumber)
Misty Williams
Waterworld
Basketball Diaries, The
Shawshank Redemption, The
RATED

"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
```

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

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;
```

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 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
```

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

```
// 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)-[: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
```

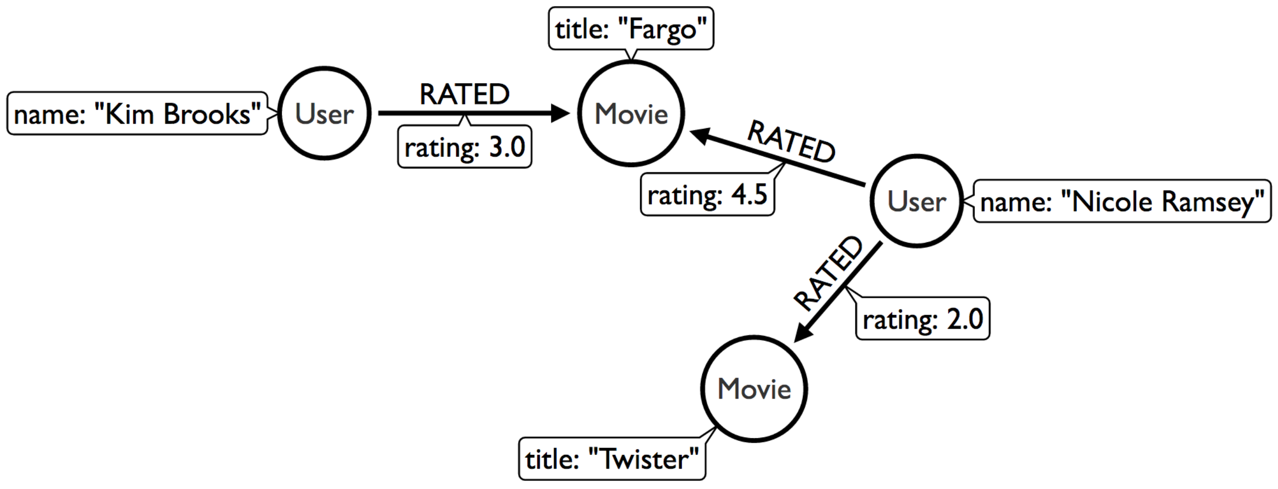
We can apply this same approach to all "traits" of the movie (genre, actors, directors, etc.):

```
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 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

```
// 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

```
// Show average ratings by Misty Williams
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?

```
// 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

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

```
// find genres with higher than average rating and their number of rated movies
MATCH (u)-[r:RATED]->(m:Movie)
       -[:IN_GENRE]->(g:Genre)
WHERE r.rating > mean

WITH u, g, count(*) AS score

// find movies in those genres, that have not been watched yet
MATCH (g)<-[:IN_GENRE]-(rec:Movie)
WHERE NOT EXISTS { (u)-[:RATED]->(rec) }

// order by sum of scores
RETURN rec.title AS recommendation, rec.year AS year,
       sum(score) AS sscore,
       collect(DISTINCT g.name) AS genres
ORDER BY sscore 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

```
// Most similar users using Cosine similarity
```

```
MATCH (p1:User {name: "Cynthia Freeman"})-[x:RATED]->
      (m:Movie)<-[y:RATED]-(p2:User)
WITH p1, p2, count(m) AS numbermovies,
     sum(x.rating * y.rating) AS xyDotProduct,
     collect(x.rating) as xRatings, collect(y.rating) as yRatings
WHERE numbermovies > 10
WITH p1, p2, xyDotProduct,
sqrt(reduce(xDot = 0.0, a IN xRatings | xDot + a^2)) AS xLength,
sqrt(reduce(yDot = 0.0, b IN yRatings | yDot + b^2)) AS yLength
RETURN p1.name, p2.name, xyDotProduct / (xLength * yLength) AS sim
ORDER BY sim DESC
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 (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.similarity.cosine(p1Ratings, p2Ratings) AS similarity
ORDER BY similarity DESC
```

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
```

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 (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.similarity.pearson(p1Ratings, p2Ratings) AS similarity
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_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
```

Further Work

Resources

- Web Cypher Refcard
- Web Neo4j Documentation
- Blog Post Collaborative Filtering: Creating the Best Teams Ever
- Video Data Science and Recommendations
- Web Use-Case: Real-Time Recommendation Engines
- Article: Exploring Practical Recommendation Systems In Neo4j
- Book (free download) Graph Data Science For Dummies

Exercises

Extend these queries:

Temporal component

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

Keyword extraction

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

Image recognition using posters

There are several libraries and APIs that offer image recognition and tagging.
Since we have movie poster images for each movie, how could we use these to enhance our recomendations?