Implementing a Recommender system with graph database

Prototype

Seminar

Author:
Hoang-Qu-Cung – 07-803-133
hoang-qui.cung@unifr.ch
Malek Jedidi – 09-214-719
malek.jedidi@unifr.ch

Course Name:
eBusiness

Examiner:
Prof. Andreas Meier

Supervisor:
Dr. Luis Terán

Date of submission:
Fribourg, 7th May 2014
Contents
1. Introduction.................................................................................................................. 3
2. Recommendation systems in eCommerce ................................................................. 4
   2.1. Content-based system ............................................................................................. 4
   2.2. Collaborative filtering ............................................................................................ 5
3. Graph data modelling .................................................................................................... 7
   3.1. Graph theory ......................................................................................................... 8
   3.2. Fuzzy sets and fuzzy clustering ............................................................................. 8
4. Graph algorithms and theorems .................................................................................. 10
   4.1. Minimum spanning tree ....................................................................................... 10
   4.2. The Hamiltonian Cycle ....................................................................................... 10
   4.3. Cosine similarity .................................................................................................. 11
5. Graph Database and Neo4j ........................................................................................ 12
   5.1. Graph database .................................................................................................... 12
   5.2. Query performance .............................................................................................. 12
6. Implementation ............................................................................................................ 13
   6.1. Design .................................................................................................................. 13
   6.2. Cypher and creation of the database .................................................................... 15
   6.3. The algorithms and theorems .............................................................................. 16
7. Use cases ..................................................................................................................... 18
   7.1. Find the nearest user ........................................................................................... 18
   7.2. Recommend a movie with a given genre ............................................................... 21
8. Conclusion .................................................................................................................... 24
References ....................................................................................................................... 25
1. Introduction

Nowadays, we are facing an overload of information in several environments, including the media content. Indeed, a huge amount of movie and TV shows are produced every day and the user can have difficulties to find the correct content when browsing a movie in electronic shops. The offer is large and choosing the correct movie to watch among millions of possibilities is a very difficult task if the shop doesn’t provide a recommendation system to filter them.

Recommender systems have become a crucial service for any electronic shop in the recent years and they are particularly used for movies or books recommendation. Having an efficient system can generate significant end-user value, by allowing the shop to personalize relations with the client.

A graph data model, although being fairly new has many solutions to improve business applications that have a huge amount of data with a high degree of correlation. A graph is a set of nodes, which represent entities, and relationships that connect them. It’s the main advantage is that connectedness between elements is the core of the design. It has also a high degree of scalability and flexibility. It means that, in an era, where the amount of data is so big and where the Internet-world is facing an overload, graph databases could propose some solutions to manage and store this data.

While working with graph data model, it is useful to know how to use effectively some graph algorithms and theorems. Some of them will be selected among the huge available panel of possibilities and explained.

Another motivation behind this work is to show the advantages of using a graph database system for a heavily connected data and provide some insights of how to build it. Neo4j, the most used graph database will be introduced. It is a rather new and immature technology and, even if it has shown some impressive performance, his potential is still being tested and improved. An illustration of Neo4j will be presented at the end of this paper.
2. Recommendation systems in eCommerce

In eCommerce, Shafer et al. assert that recommendation systems are a key to perform mass customization. It means finding the appropriate product or service among a morass of information for multiple customers with multiple needs. In other words, each customer would have personalized recommendations and the website would be able to adapt itself to each user. [Sch99]

Personalization is a practice that improves largely the interaction of a customer with the vendor. More than ever, the shopping experience on the web must provide such systems in order to establish a long-term relationship with the customers. [Pra01]

There are two main categories of recommendation systems: content-based (or item-to-item) and collaborative filtering (user-to-user). In the first category, the recommendation is based on the products and their properties, whereas the second consider the similarities between end-users. Both of them will be described below and the advantages will be explained. [Dem13] [Mei09]

2.1. Content-based system

The content-based system is a model, where the recommendations are made on the basis of the several properties of a product. In other words, the system compares the similarities or the complementarity of the elements.

For example, a search system can use this category of recommendation. Given the previous purchases, the algorithm would search in the catalog related items and suggest them to the user. If a customer has bought a great number of western movies or has bought many books from the same author, the system would recommend other popular western movies or books from this author. [Lin03]

This system is also often used for directly scan the shopping cart of a customer instead of his purchase history and match the items with complementary ones. For example, the figure below shows, with a shopping cart containing a Bluetooth device, the system recommends Bluetooth dongles. [Lin03]
A content-based system can also use the similarity between elements to provide a recommendation. In this case, a user isn’t asked to put attributes, but only a movie he liked and the system compares this item with similar ones in the base. This technique can also be performed by analyzing the purchase history of a user: it will try to match the movies the user watch and find similar ones among his store. [Dem13]

There is a main limitation to the content-based model: the scalability. Since it can work very well when the purchase history is rather small (or with the shopping basket where the collection is limited), the quality of the recommendation can drop very fast when dealing with a larger set of elements. If the user buys a lot of items, the system wouldn’t be able to return pertinent recommendations. It could either be too general, like the most popular movies of a category, or too narrow like the books of the same author. [Lin03]

With only an “is similar to” relationship between products, the recommendation can have a rather poor. It would be more interesting to know how a product is similar to another one and the level of the similarity.

2.2. Collaborative filtering

In this category, the system builds profiles of users based on their behaviors and compares the different user profiles to provide a recommendation. For example, the system find correlation between users’ purchases and recommend a product to a customer based on the items bought by similar users.

A common practice to improve this technique is to ask the user to rate the products, so it becomes much easier to find common pattern among users. By this way, the model is transformed into a segmentation model. The customer’s preferences are analyzed and the recommendations are made based on the preferences of other users of the same segment. So, the users collaborate to build their “global” profile and profit from others’ ratings. [Pra01]

To consolidate this model, not only explicit ratings can be considered, but also implicit ratings such as navigating behavior like time spent on a page or bookmarks. [Pra01]
However, defining segments of users could lead to poor quality recommendations, since all the users of a same segment are considered to have exactly the same preferences. And doing some tweaking on the segments, like using granularity, would become expensive to just determine those segments. [Lin03]

Also, collaborative filtering system requires time: Since it is based on past behaviors, the user has to build slowly his profile and provide data that can be analyzed. A new user preferences are difficult to determine and the strategy would be rather asking to provide inputs by himself. The participation of the user is crucial to determine the accuracy of the recommendations. [Dem13]
3. Graph data modelling

The last section has shown several techniques to compute a recommendation. There is a common point among all those techniques: the importance to relate the items which each other. Indeed, in either user-to-user or item-to-item system, the core of the technique is to find connection between elements. The users are compared to each other, as well as products.

To hold all those connections, graph data models will be used. Actually, graphs are well suitable to store such data, especially when the complexity of relationships can’t be handled by relational databases anymore. Even if the data set is small, the architecture of graph databases permit us to query it rapidly and commute complex operations on relationships.

A graph model is composed of nodes and directed relationships (or vertices and edges). Both of them can have properties, which are key-values that define the elements they are attached to. The thing that comes directly in mind when seeing data modelled with graphs is that it is really intuitive to understand and it seems very easy to express various scenarios. [Rob13]

Although being simple, it can also be used to model more complex scenarios. The nodes and the relationships can represent multiple and different types of elements. [Rob13]
3.1. Graph theory

The first theorem of graph theory was first presented by Leonhard Euler to find a solution known as the Seven Bridges. Since then, many other graph theorems appeared and graphs have proved that they can model a lot of different-looking. They have also been used to understand problems in many fields such as biology, transportation or social networks. [Hür14]

This has led to the development of graph algorithms, a class of algorithms to compute operations on graphs.

![A graph](image1)

Figure 1: A graph

3.2. Fuzzy sets and fuzzy clustering

The previous section has shown some weaknesses in content-based and collaborative filtering problems. One redundant problem is that the relations between nodes are one way-relations: it means that a relationship is only an “is similar to” one. In reality, things can have shades of grey. A product can be “more or less” similar to another one. A user can maybe have a little bit the same preferences with another one or can share strong similarity. Thus, the user can be a member of several segments.

In fuzzy logic, the relationships are determined in a range of 0 to 1. For example, in a binary world (non-fuzzy) a movie can belong to the “Western” genre and to “Action”
genre, and not belong to the “Romance” genre. In a fuzzy world, the same movie can belong to the “Western” genre at 45% and to the Romance genre at 15%. [Bou13] [Ter10]

In graph theory, the fuzzy logic could be translated as weighted graphs. In other words, the vertices are labeled with numbers between 1 and 0 and the weight of a path would be the sum of all traversed vertices. [Ruo13]

Figure 2: Fuzzy sets
4. Graph algorithms and theorems

There are plenty of existing graph algorithms. Some widely used algorithms will be explained below, those algorithms will be later used to implement the system and have been wisely chosen. In order to solve our initial statement, several modification will be made to those algorithms.

4.1. Minimum spanning tree

When searching through the literature for a clustering solution, one redundant theorem explained as an introduction is the minimum spanning tree (MST). A MST is a tree that connects all the vertices together and has the minimal length.

There are two commonly used algorithms for this model: Prim’s Algorithm and Kruskal’s Algorithm. They have both easy logic, but Prim’s seems to show better performance when computing with a very large number of nodes. [Hür14]

4.2. The Hamiltonian Cycle

Unlike the MST that doesn’t permit to have cycles, The Hamiltonian cycle is a well-known general algorithm that can be used to determine the shortest path of a cycle. Of course, the weights of the edges have to be determined. Many others algorithms are based on this general theorem. [Ruo13]
4.3. Cosine similarity

The cosine similarity is a measure of similarity between two vectors. It is the cosine of the angle between two n-dimensional vectors. The cosine is also defined as the dot product of the two vectors divided by the product of the two vectors lengths. The result ranges between -1 and 1. 1 meaning perfectly similar.

\[
\text{similarity} = \cos(\theta) = \frac{A \cdot B}{\|A\| \|B\|} = \frac{\sum_{i=1}^{n} A_i \times B_i}{\sqrt{\sum_{i=1}^{n} (A_i)^2} \times \sqrt{\sum_{i=1}^{n} (B_i)^2}}
\]

The cosine similarity is commonly used in recommendation systems for collaborative filtering in order to determine the similarity between two users and to find the nearest neighbors. [Ter10]
5. Graph Database and Neo4j

In the last chapter, the benefits of using correlation search between data were shown. To compute those operations, it is important to find and use the right tools and technology. Since, graph models and graph algorithms are used in this paper, this chapter will explain how to use graph databases to store our data and how to query it by algorithms computations.

5.1. Graph database

Graph database is a noSQL data model which focuses on entities and the relation between them. There are several available graph databases, but, in this paper Neo4j will be presented and used.

Neo4j is an open-source graph database written by Java programming language and is said to be fully ACID, in contrary of most of the database models.

5.2. Query performance

The architecture of Neo4j is built in such a way to be scalable with Big Data. Hongchang Huang and Ziyu Dong tested the query performance of Neo4j and came with some interesting results. First, the architecture is defined in such a way to be traversed very rapidly. The data is not only physically stored in a way to be read rapidly by allowing fixed size spaces for tuples, but also by using multiple cache layers. [Hua13]

Their result are quite interesting: as the complexity rises, the stability of a query remains rather at the same level. Thus, it isn’t fool to think that graph databases could bring some solutions to today’s internet overload of information.
6. Implementation

6.1. Design

Like most of noSQL database systems, there is no specific rules to design a database. From the developer point of view, the flexibility is one of the main advantages, of those advanced database systems. There is no need to normalize the database like in a rational environment. Therefore, the flexibility does not mean that it is right to put whatever wherever. A good practice is to design the database depending on the query patterns that are most used. [Cud14]

It is also important to split the databases for OLAP and OLTP operations. In an OLAP system, it isn’t necessary to have a fully consistent database the whole time. Ideally, the database for OLAP operation should be separated from OLTP. Analysis operation should be fast. The design below has only sufficient and enough data to perform our patterns. [Cud14]

In the case of this paper, the database of a fictional online movie store will be designed. Here are the nodes, the relationships that will be created with their properties. Those are defined to match our algorithms and methods explained in the next sub-chapter.

Nodes

The nodes of the graph database will represent entities:

- The movies
- The persons
- The genres
- The users

A total of 10 movies and 10 users were create for this Implementation.

Relationships

There will be initially four be relationships:

- RATED (from User to Movie)
- ACTED_IN (from Person to Movie)
• DIRECTED (from Person to Movie
• HAS_GENRE (from Movie to Genre)

Properties:
The Movies nodes will have following properties:

• Title
• Year of release
• Imdb rate

The Persons, the Genres and the Users nodes will have one property:

• Name

The Rated relationships will have one property:

• Rate

The Acted_in, Has_genre and Directed nodes will have one property:

• Weight
6.2. Cypher and creation of the database

Cypher is the Neo4j query language. It is a declarative language that allows to perform reading and writing operations on the graph database. One of the main advantages of this language that is highlighted by his creators is the fact that it is designed to be human readable.

Here is a sample of queries for creating the nodes and the relationships.

To perform the write action on several nodes, one common method avoiding scripting is the use of a spreadsheet.

Figure 4: A sample of the created database
Figure 5: Using a spreadsheet to import the data into Neo4j

Query example for creating a Movie node:

```cypher
MERGE (:Movie {name: 'Blood Diamond', rate:'8', year:'2006'})
```

Query example for creating a User node:

```cypher
MERGE (:User {name: 'Dan'})
```

Query example for creating a Relationship between User and Movie nodes:

```cypher
MATCH (u1:User),(m1:Movie) WHERE u1.name='Alfred' AND m1.name='Body of Lies'
CREATE (u1)-[:RATED {rate:2}]->(m1)
```

Note that since March 2014, with the release of Neo4j 2.1, it is possible to import directly *.csv files in Cypher. The direction taken by Neo4j is clearly to be used for large amount of data. And this update will provide more performance to deal with a big set of data.

6.3. The algorithms and theorems

Now that the database is designed and implemented. The canvas of the algorithms will be written in Cypher and will be tested.

A fuzzy logic will be implemented such following: Like explained in the precedent section, a movie can have different levels of genres. It is neither only a “Comedy” movie nor a “Sci-Fi” movie. It can be both but it has always a stronger “Comedy” or “Sci-Fi” side. To implement it, a property “weight” will be set on the “HAS_GENRE” relationship. The term weight will be used according to the graph theorems. Indeed, it is more a
value to weight or graph and determine the distance between nodes than a fully fuzzy logic. The weights will have a range between 0 and 9.

The cosine similarity will be implemented on the relationships between users, depending on how the users rated the movies watched. Each user will be related to the other ones by a relationship called “SIMILARITY” with a property “value”.

```
User 1 ─── Similarity {value: [0..1]} ─── User 2

Rated {rate: 7} ─── Movie ─── Rated {rate: 4}
```

Figure 6: Cosine similarity

Finally, a variation of the minimum spanning tree will be implemented. Unlike the precedent design, it will not be written onto disk but will by computed via queries. The goal here is to determine the longest path from a node to another, by traversing the paths and calculating the total weight.

All of those implementations will be illustrated with use cases in the next chapter.
7. Use cases

7.1. Find the nearest user

The goal of the use case is to perform a recommendation of a movie that a user hasn’t rated yet, using collaborative filtering. In other words, the system will find the users nearest neighbors and match his rating behavior with them. Thus, it will be possible to predict what he wants to watch next.

First of all, the cosine similarity will be added to the database. As said before, relationships will be created between each single user.

The following Cyper query is ran.

```
MATCH (u1:User)-[x:RATED]->(m:Movie)<-[y:RATED]-(u2:User)
WITH SUM(x.rate * y.rate) AS DotProduct,
    SQRT(REDUCE(xDot = 0, i IN COLLECT(x.rate) | xDot + toInt(i^2))) AS xLength,
    SQRT(REDUCE(yDot = 0, j IN COLLECT(y.rate) | yDot + toInt(j^2))) AS yLength,
    u1, u2
CREATE UNIQUE (u1)-[s:SIMILARITY]-(u2)
SET s.value = DotProduct / (xLength * yLength)
```

Now, the users are linked to each other with a “SIMILARITY” relationship. Here is a sample of the values set on this relationship:

<table>
<thead>
<tr>
<th>DotProduct / (xLength * yLength)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.934331459958054</td>
</tr>
<tr>
<td>0.471870105616409</td>
</tr>
<tr>
<td>0.6114706739826228</td>
</tr>
<tr>
<td>0.38930477498252253</td>
</tr>
<tr>
<td>0.6109208450770314</td>
</tr>
<tr>
<td>0.8455510310544155</td>
</tr>
<tr>
<td>0.9449978885786016</td>
</tr>
</tbody>
</table>

*Figure 7: Sample of Cosine similarity values*
The next step testing this implementation is to run a query to find the nearest neighbor of a given user. Let’s find the nearest neighbor of Igor.

```
MATCH (u1:User {name:'Igor'})-[s:SIMILARITY]-(u2:User)
WITH u2, s.value AS sim
ORDER BY sim DESC
LIMIT 3
RETURN u2.name AS Neighbor, sim AS Similarity
```

Here is the result:

<table>
<thead>
<tr>
<th>Neighbor</th>
<th>Similarity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jasmin</td>
<td>0.934331459958054</td>
</tr>
<tr>
<td>Alfred</td>
<td>0.8422177437542384</td>
</tr>
<tr>
<td>Ethan</td>
<td>0.8422177437542384</td>
</tr>
</tbody>
</table>

The system has found the 3 nearest neighbors of Igor. The next query will return the movies that those neighbors have watched and that Igor didn’t yet. It would also display
the rates given by the neighbors. This step improves a little bit more the recommendation.

MATCH (u2:User)-[r:RATED]->(m:Movie), (u2:User)-[s:SIMILARITY]-(u1:User {name:'Igor'})
WHERE NOT((u1)-[:RATED]->(m))
WITH m, u2, s.value AS similarity
ORDER BY similarity DESC
return m, u2

The result is the following:

<table>
<thead>
<tr>
<th>Movie</th>
<th>User</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>Caspe</td>
</tr>
<tr>
<td>rate</td>
<td>8.2</td>
</tr>
<tr>
<td>year</td>
<td>1995</td>
</tr>
<tr>
<td>name</td>
<td>Shutter Island</td>
</tr>
<tr>
<td>rate</td>
<td>8.1</td>
</tr>
<tr>
<td>year</td>
<td>2010</td>
</tr>
</tbody>
</table>

Figure 10: Nearest neighbors’ favorite movies

It shows the movies watched by every user ordered by proximity. But it would be more interesting to group the movies together, since this information is split among all users. That is why the next step will introduce a Group By Movie. This query will pick the 3 nearest neighbors and displays their ratings.

MATCH (u2:User)-[r:RATED]->(m:Movie), (u2:User)-[s:SIMILARITY]-(u1:User {name:'Igor'})
WHERE NOT((u1)-[:RATED]->(m))
WITH m, u2, s.value AS similarity, r.rate AS rate
ORDER BY similarity DESC
WITH m, COLLECT(rate)[0..3] AS rating
return m, rating

The final step would be to determine naively the mean of those ratings and ordering it.
MATCH (u2:User)-[r:RATED]->(m:Movie), (u2:User)-[s:SIMILARITY]-(u1:User {name:'Igor'})

WHERE NOT((u1)-[:RATED]->(m))

WITH m.name AS title, u2, s.value AS similarity, r.rate as rate

ORDER BY similarity DESC

WITH title, COLLECT(rate)[0..3] AS rating

WITH title, REDUCE(s = 0, i IN rating | s + i) / LENGTH(rating) AS recommendation

ORDER BY recommendation DESC

return title, recommendation

Now, the result is the following:

<table>
<thead>
<tr>
<th>title</th>
<th>recommendation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Revolutionary Road</td>
<td>8</td>
</tr>
<tr>
<td>Titanic</td>
<td>6</td>
</tr>
<tr>
<td>Shutter Island</td>
<td>6</td>
</tr>
<tr>
<td>Casino</td>
<td>6</td>
</tr>
<tr>
<td>The Departed</td>
<td>6</td>
</tr>
<tr>
<td>Gangs of New York</td>
<td>4</td>
</tr>
<tr>
<td>Catch Me If You Can</td>
<td>4</td>
</tr>
</tbody>
</table>

Figure 11: Recommended movies

It seems that the nearest neighbors liked the movie “Revolutionary Road” and rated it with the score 8. This movie would really likely please to Igor.

7.2. Recommend a movie with a given genre

This Use Case will focus more on the item-to-item recommendation. The goal of this Use Case is to provide a recommendation to a User who hasn’t rated enough Movies in order to compute a collaboration filtering recommendation.

When seeing this Use Case, it may normal to think at using the cosine similarity like in the precedent one. This is creating “SIMILARITY” relationships between Movie nodes. However, it doesn’t work well in this case. The reason is that it is necessary to have great amount of vector coordinates to determine a pertinent Cosine similarity. The nodes must have a great amount of mutual direct neighbors. It isn’t the case for the
Movie, because the array of Actors and Directors is really big and to find a movie which have the same director AND the same actors AND the same genres. To calculate their similarity seems irrelevant here.

That is why, a variant of the minimum spanning tree will be used in this case. This method can, for instance, be used to propose another similar movie to a user by analyzing his current shopping cart. It is also widely done, but this example shows how to do it, using the same graph database as before. The strategy here is to traverse the graph given a mandatory set of nodes and finding a height total weight (instead of the minimum).

The statement is the following:

Given a movie, find another which share the same genre. The recommended movie must belong at a high rate to the main genre of the given movie.

Here is the query:

```sql
START m1=node(*)
MATCH (m1:Movie)-[r1:HAS_GENRE]->(g:Genre)<-[r2:HAS_GENRE]-(m2:Movie)
WHERE m1.name = 'Inception'
WITH g, m2, r2, max(toInt(r1.weight) AS maximum
WHERE r2.weight > 4
RETURN m2
```

In this example, the movie Inception is the starting node. The algorithm is:

```
START Node: Name = 'Inception'
    FOR ALL “HAS_GENRE” relationship r1 DO
        SELECT max(r1.weight) AS maximum
    END
    SELECT Genre g s.t. (g) <- [maximum:HAS_GENRE] - (Inception)
    FOR ALL “HAS_GENRE” relationship r2 DO
        SELECT r2.weight > 4
```
In other words, it looks at the dominant genre of a selected movie and recommends other movies which has that genre as a high rate.

Here is the result:

![Figure 12: Similar movies](image-url)
8. Conclusion

Although not being fully mature yet, Neo4j already shows some interesting lookouts. It can be used in combination of graph algorithms and theorems like the minimum spanning tree or the cosine similarity to compute recommendations. And with his natural way to connect data and his ability to perform well with larger sets, it could provide some solutions to today’s issues about Big Data.

Since the data set used in this paper is rather small, a further lookout is to push his possibilities with a much larger set and implement it on a distributed system. Neo4j is great to play around with small sets, researches were made to test his performance. Now, it would be interesting to know how it works in a company environment, and if the promises made are kept.
References

Articles


Literature


Website


Courses