Flexible recommender systems based on graphs

Joseph Pellegrino

To cite this version:

Joseph Pellegrino. Flexible recommender systems based on graphs. AISR2017, May 2017, Paris, France. hal-01640313

HAL Id: hal-01640313
https://hal.science/hal-01640313

Submitted on 20 Nov 2017

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Flexible recommender systems based on graphs

J. Pellegrino, *Kernix Software*

Abstract— **The objective of this paper is to share Kernix's approach to build flexible recommender systems based on graph oriented databases. The integration of entities, and relationships between them, into a unique graph structure allows to design recommendations as graph traversals. This approach offers a flexible framework allowing to handle the variety of entities of interest and enabling to design rich strategies in order to compute recommendations for various use cases.**

Index Terms—**Graph database, Recommender systems, Data Management**

I. Introduction

Kernix is a company of forty-five people whose activity rely on a digital factory, crafting custom websites, and a data lab, providing data oriented solutions for business use cases. These solutions span from predictive maintenance to credit risk scoring but also fraud detection and recommender systems. In the latter context, Kernix has designed and built several custom recommender engines aiming at fulfilling its customers' needs. In order to cope with the variety of these needs and with the data available, Kernix has decided to adopt a flexible approach essentially based on the use of a graph oriented database.

Basic approaches to recommendation [1] consist in building separate models for content-based (CB) and collaborative filtering (CF) strategies, respectively relying on the exploitation of items-items similarities and users-items interactions. In general, these approaches lack some kind of flexibility in the integration of complex interactions between entities. As an illustration, in the case of movie recommendation, it is not easy to merge in one approach the explicit taste of a user about its favorite actors and genres with its ratings for specific movies.

A flexible approach would ease the exploitation of the multiple kinds of possible interactions between different entities (users, actors, genres and movies in our example). Graphs are natural mathematical structures allowing to encode these interactions, and, as we will show in what follows, recommendations can be computed thanks to graph traversals.
In the following, we present how Kernix is leveraging on a graph oriented database technology in order to build different so-called flexible recommendation engines. Part II briefly describes the use of a graph oriented database, allowing us to introduce the vocabulary further used. Part III deals with a toy example of a CF recommendation based on the MovieLens dataset. Part 2 refers to a hybrid recommender engine that Kernix has built for one of its client. Part IV summarizes the stack of technologies on which our solutions are based, and the workflow associated to it. In the conclusion, we share some advantages and caveats of our approach and draw some perspectives.

II. Brief description of the use of a graph database

A graph database models and stores data as *nodes* and *edges* of a graph structure [2]. These elements can bear types allowing their categorization and can embed additional information specifying their *properties*. For instance, in the case of movie recommendation, node types could be *User, Actor*, *Genre* and *Movie* and edge types could be *Has_rated*, *Has_actor*, *Has_genre* for edges respectively linking users to movies, movies to actors and movies to genres; while properties of a *User* node could be the name, age and sex of the user and a property of the *Has_rated* edge could be the rating itself.

Graph databases allow efficient and fast retrieval of complex hierarchical structures that are difficult to model in relational systems. Information retrieval is performed thanks to specific query languages [3] based on the semantic of graph traversals, generally proposing optimized routines for computing shortest paths for instance.

Within this graph framework, for a given dataset, several kinds of recommendations can be elaborated by combining and scoring paths between entities. In our example, similarity between users can be evaluated through the paths that connect one user to movies, actors and genres, that are themselves connected to other users, each type of path being potentially weighted by different coefficients. The two following parts will concretely illustrate the way these concepts can be implemented.

III. Example of a CF recommender system

To illustrate the case of a graph-based CF recommender system, we have built a movie recommender engine based on the ML-100k dataset [12]. This dataset has been gathered by GroupLens Research on the MovieLens website and consists in 100,000 ratings (1-5) from 943 users on 1682 movies. The

dataset has been cleaned up such that each user has rated at least 20 movies. The model chosen for the graph consists in:

- two kinds of nodes: *User* and *Movie*.
- one kind of edge: *Has_rated*, relating users to movies. The rating $r(m_j,u_i)$ given by user u_i to movie m_j is recorded as a property of the edge.

For a given user u_1, we have chosen to recommend movies that are mostly appreciated by users giving similar ratings as u_1. For reasoning, this strategy implies the following conceptual steps.

Step 1: Selection of the users who are the most "similar" to u_1.

The similarity between u_1 and u_i is here defined as:

$$sim(u_1, u_i) = card(S_{u_1,u_i})/card(R_{u_1})$$

where $S_{u_1,u_2} = \bigcup_{|r(m_j,m_2)-r(m_j,m_1)|\leq 1} m_j$, is the set of movies commonly rated by u_1 and u_2 with a difference of rating less are equal to 1; and R_{u1} is the set of all movie rated by u_1.

Thanks to this similarity measure, we form the subset U_{sim} of users u_i with similarity with u_1 above a certain threshold t (here arbitrarily set to 0.5).

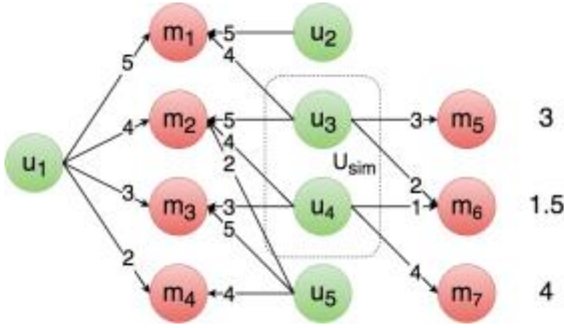

Fig. 1. Schematic representation of the method

Step 2: Ranking of movies seen by these users.

The recommendation score of each movie m_j rated by the users belonging to U_{sim} is then computed as the mean rating. In this case, movies will be ranked from « most appreciated by similar users » to « least appreciated by similar users ». If V_m is the subset of users who have rated movie m, and W_m its intersection with U_{sim}, then the score of movie m can be expressed as:

$$s(m) = \frac{\sum_{u_i \in W_m} r(m, u_i)}{card(W_m)}$$

Despite the fact that the above description of the method is mainly set-theoretical, the approach is in fact really natural in terms of graph traversal as illustrated by Fig.1 and can be easily implemented as a request to the graph database.

Although this toy recommender system shows reasonable performances for this case, an assessment on a more significant set of users must performed to draw any solid performance metrics. One has to mention that this model doesn't involve any training phase as machine-learning based approaches requires so it is really well suited for cases of fast addition/deletion of entities and evolution of their connections. The following part extends the approach to a richer use case.

IV. Example of a Hybrid Recommender System

In this part, we will share the design of the approach we took for building a recommender system for one of our client. This company proposes to ease interactions between individuals and professionals through "do it yourself" workshops. Our goal was to integrate to the website of this company an engine recommending workshops to users. The recommendation has been built on connections between different "entities", namely *User*, *Craftsman*, *Workshop*, *Session* and *Category*. The types of these entities and the relationships we decided to implement in the graph database are summarized in the Fig.2.

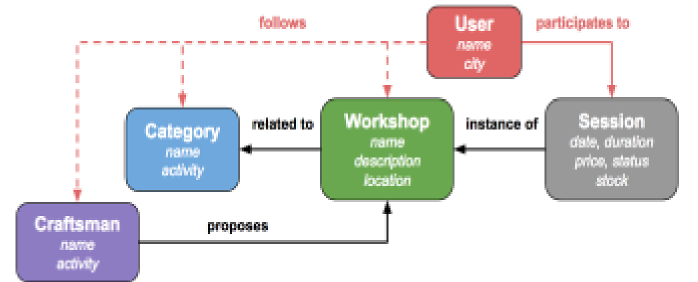

Fig. 2. Types of data and edges defined for the recommender system

In order to compute the recommendation of workshops, we decided to combine the three following strategies based on the same graph structure.

Strategy 1: Scoring through categories

For a given user, the ranking of workshops to recommend is based on the number of times they are linked to categories for which the user has shown an interest. This mark of interest, as illustrated in Fig.3, take into account several paths between the user and the categories:

- direct edges of type "follow" from the user to the categories
- paths through workshops followed by the user
- paths through workshops to which the user attend a session

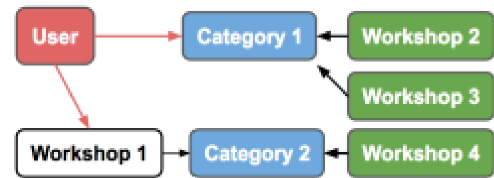

Fig. 3. Example of paths taken into account for the scoring through categories (deeper paths passing by sessions are not shown for compactness)

Strategy 2: Scoring through users similarity

This strategy is analogous to the CF recommendation example of the previous part, except that paths taken into account between users can be deeper than just one node. Indeed, to select the subset U_{sim} of users similar to the target user, we exploit the paths through workshops for which the user has shown a mark of interest by:

- following the workshop
- participating to a session of the workshop
- following the craftsman that proposes the workshop

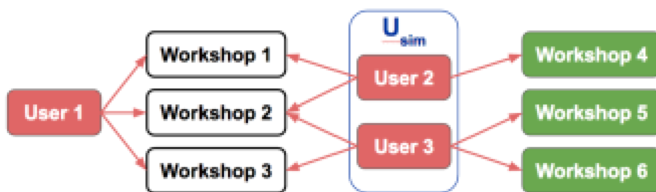

Fig. 4. Example of paths taken into account for the scoring through users similarity (deeper paths passing by sessions and craftsmen are not shown for compactness)

The users defined as similar to the target user are those counting a sufficient number of workshops in common with him. Finally, workshops are ranked based on the number of times they are linked to the subset U_{sim}, through the same kinds of paths as described above.

Startegy3: Scoring through workshops similarity

This last strategy leverages on the computation of semantic similarity between the descriptions of the workshops. In order to compute these similarities, we have implemented a NLP pipeline performing the vectorization of the descriptions (through tokenization, stemming and tf-idf weighting) and the fit of these data points by LSI model (in order to merge frequently co-occurring terms into main *concepts*). The similarity between descriptions is obtained by the cosine of the angle formed by their corresponding vectors. The similarity scores are then materialized in the graph database as the property of "similarity" edges drawn between corresponding workshops as illustrated in Fig.5:

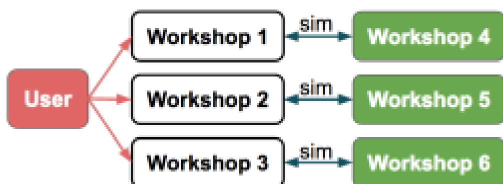

Fig. 5. Example of paths taken into account for the scoring through workshops similarity (deeper paths passing by sessions and craftsmen are not shown for compactness)

Recommended workshops are defined as the most similar to the workshops for which the target user has shown an interest, as described in the previous strategy.

Analogously to the above toy example, these three strategies are implemented as requests to the graph database. Their results are subsequently combined to give a list of workshops that is further filtered by the geographical region of interest of the user and sorted by remaining days before available session.

One advantage of this approach is that it is very flexible schemes of computation and combination of recommendation based on the same graph structure. Another subtle advantage is that a small amount of data is sufficient to start serving reasonably adapted recommendations. This rely on the fact that recommendations are based on local request to the graph structure such that they can start making sense as soon as portions of the graph are sufficiently well connected.

V. Technological Stack and Workflow

This part draws the main lines of the architecture we use in order to build an engine serving the kind of flexible recommendation described above.

Basically, the engine lies on the synchronized exploitation of a Neo4j database [4] and a MongoDB database [7]. The Neo4j technology is made for storing data structured as graph and requesting it with the Cypher language. MongoDB, a document oriented database, is used in order to store potentially large content associated with each entity (the descriptions of the workshops in the previous use case for instance), whereas the graph database is not really suited for this functionality.

The overall application is built with the Node.js [9] framework. Packages of this framework (mainly async.js) allow to design asynchronous workflows that are useful for handling real-time addition/deletion of entities and relationships, and managing complex execution schemes. Other packages (mainly express.js) allow to expose web services useful if the engine has to communicate with websites (as needed by a majority of our use cases). Finally, connectors to Neo4j and MongoDB are also available. All these features allow the application to orchestrate data collection and processing, writing to the databases and requesting them to serve recommendations.

NLP and machine-learning tools we are using for computing similarity between texts for instance, mainly belong to the python library Gensim [6]. Python processes are integrated to the overall Javascript platform thanks to communication via a RabbitMQ queuing system [8].

Finally, to ease the deployment of the whole application, each of its micro-services is containerize thanks to the Docker technology [10] .

VI. Conclusion and Perspective

We have presented in this paper the way Kernix is building recommender engine based on a graph database. To conclude, we share some advantages and caveats of this kind of flexible recommender system.

We have mainly adopted the approach described above because of the ease of addition/deletion of entities and their connections and the adaptability of the recommendation

computation to the various client's use cases we have encountered over time. This flexibility of the system also relies on the possible enrichment of the graph of entities thanks to data processing as mentioned above concerning semantic similarity. Beside these motivating points, two other points remain challenging. Firstly, we are still investigating ways to assess recommendations quality either by in production A/B testing and by offline evaluation. Secondly, we are studying the potential scalability of these techniques to high data volume and denser stream of information.

REFERENCES

[1] Francesco Ricci and Lior Rokach and Bracha Shapira, Introduction to Recommender Systems Handbook, Recommender Systems Handbook, Springer, 2011, pp. 1-35

[2] Angles, Renzo; Gutierrez, Claudio, "Survey of graph database models" *ACM Computing Surveys*. Association for Computing Machinery. **40** (1)

[3] https://developer.ibm.com/dwblog/2017/overview-graph-database-query-languages/

[4] https://neo4j.com/

[5] https://neo4j.com/developer/cypher-query-language/

[6] https://radimrehurek.com/gensim/

[7] https://www.mongodb.com/

[8] https://www.rabbitmq.com/

[9] https://nodejs.org/en/

[10] https://www.docker.com/

[11] F. Maxwell Harper and Joseph A. Konstan. 2015. The MovieLens Datasets: History and Context. ACM Transactions on Interactive Intelligent Systems (TiiS) 5, 4, Article 19 (December 2015), 19 pages. DOI=http://dx.doi.org/10.1145/2827872

[12] https://movielens.org

[13] A Graph-based Recommender System for Digital Library Zan Huang, Wingyan Chung, Thian-Huat Ong, Hsinchun Chen (https://pdfs.semanticscholar.org/859b/e8c9a179cb0d231e62ca07b9f2569035487f.pdf)