Methods for Speeding up Recommender System Computations Using a Graph Database

Patricia E.N. Lutu

Abstract—**Recommender systems are commonly used for Internet-based activities to assist users in making decisions on what items to select. One very common use of recommender systems is in electronic commerce purchases. The need for recommender systems in electronic commerce is due to the vast amounts of items to choose from. Due to this vast amount of items, generation of recommendations for recommender systems is a computationally intensive activity. This paper reports on studies that were conducted to investigate methods for speeding up the computations for generating recommendations when the data that is used to generate recommendations is stored in a graph database. The proposed methods involve the pre-computation and storage of values that are used in the generation of recommendations. This leads to a speed-up of the computations for generating recommendations.**

Index Terms–**collaborative filtering, Cypher, graph database, Neo4j, recommender system**

I. Introduction

Recommender systems are commonly used for Internet-based activities to assist users in making decisions on what items to select. One very common use of recommender systems is in electronic commerce (e-commerce) purchases. Many e-commerce businesses, e.g. Amazon.com, use recommender systems to provide customers with lists of automatically generated recommendations of items to purchase. Examples of items are: books, music, movies. The recommendations are based on historical data on item purchases. The historical data typically consists of previous customers purchases and product ratings by other customers [1]. The need for recommender systems in e-commerce is due to the vast amounts of items to choose from, which leads to information overload. E-commerce traders, e.g. Amazon.com may have millions of item categories and thousands of item types in each category. This makes it extremely difficult for customers to examine all items in a given category in order to make informed decisions on which items to purchase. Various approaches exist for implementing recommender systems. Jannach, et al. [2] and Felfernig et al. [3] have observed that collaborative-filtering systems, content-based systems, and knowledge-based systems are the three basic approaches to recommender systems implementation. The research reported in this paper is based on collaborative-filtering recommender systems. Generation of recommendations for collaborative-filtering systems is a computationally intensive activity.

Experiments were conducted on methods for speeding up the computations for generating recommendations for the user-based collaborative-filtering method, when the data is stored in a graph database. The experimental results demonstrate that, for a graph database, it is easy to pre-compute and store various values that are needed for generating recommendations. These values are subsequently used to generate recommendations for active users at a much faster rate. The rest of the paper is organised as follows: Section II provides the background to the reported research. Section III provides a discussion of the implementation of user-based collaborative-filtering recommender systems. Section IV presents the methods that were studied for speeding up computations. Section V provides a discussion of the experimental methods that were used. Section VI presents the experimental results. Section VII concludes the paper.

II. Background

A. *Collaborative Filtering Recommender Systems*

According to Ricci, et al. [1], collaborative filtering is the most commonly used approach for recommender systems. Collaborative filtering consists of two categories of recommender systems. These are: neighbourhood-based and model-based systems. Felfernig et al. [3] and Jannach et al. [2] have reported two different approaches to neighbourhood-based collaborative filtering. These are user-based systems and item-based systems. Neighbourhood-based collaborative filtering recommender systems rely on users' past behaviour to be able to recommend items [4]. The past behaviour is in the form of purchases and ratings of items by users. The two categories of collaborative filtering are user-based and item-based collaborative filtering. For user-based collaborative filtering, the data for users with similar preferences is used to suggest items to the active user [3]. Given an active user *U*, who is interacting with the system and requires recommendations, the first step of user-based collaborative-filtering is to identity users with similar ratings to *U*. This step is done using the *k*-nearest neighbour algorithm, which identifies the *k* most similar users to *U* [5]. The ratings and purchases for these *k* users are then used to determine the recommendations which are presented to the user *U*. For item-based collaborative filtering, the rating of the active user *U* for an item *I* is predicted based on the ratings of *U* for items similar to *I* [5]. Two items are considered to be similar if several users of the system have rated these items in a similar fashion.

Manuscript received March 23, 2020; revised July 08, 2020.

Patricia. E. N. Lutu is a Senior Lecturer in the Department of Computer Science, University of Pretoria, Pretoria 0002, Republic of South Africa, phone: +27124204116; fax:+27123625188; e-mail: Patricia.Lutu@up.ac.za; web: http://www.cs.up.ac.za/~plutu;

B. Computations for user-based collaborative filtering

The user-based approach takes the ratings of other users for an item, and uses this information to make a recommendation for the active user [5]. A common measure that is used for this approach is Pearson correlation coefficient of *similarity* [5]. Given two users U_a and U_b, a set of items Is_{et}, and item I, ($I \in I_{set}$), that is rated by both U_a and U_b, the Pearson correlation coefficient of *similarity* $sim(U_a, U_b)$ between users U_a and U_b is given by

$$sim(U_a, U_b) = \frac{\sum_{I \epsilon I_{set}} (rU_{a,I} - \overline{rU_a})(rU_{b,I} - \overline{rU_b})}{\sqrt{\sum_{I \epsilon I_{set}} (rU_{a,I} - \overline{rU_a})^2} \sqrt{\sum_{I \epsilon I_{set}} (rU_{b,I} - \overline{rU_b})^2}} \quad (1)$$

In equation (1), rUa,I denotes the rating for item I by user Ua and rUb,I denotes the rating for item I by user U_b. $\overline{rU_a}$ and $\overline{rU_b}$ denote the mean ratings by U_a and U_b respectively. The possible values of *similarity* range between -1 and 1, where 1 represents perfect *similarity*, and -1 represents maximum *dis-similarity*.

The users with *similarity* values closest to 1 are then selected. These are the k-nearest neighbours (e.g. $k = 20$) for the active user U_a. The final task of the recommender system is to predict item ratings for the active user U_a. These ratings are for items that were purchased and rated by a similar user U_b but not yet purchased or rated by user U_a. Given the set of k-nearest users U_{set}, the rating prediction of an item I, not yet purchased by U_a is computed as

$$pred(U_a, I) = \overline{rU_a} + \frac{\sum_{U_b \epsilon U_{set}} sim(U_a, U_b).(rU_{b,I} - \overline{rU_b})}{\sum_{U_b \epsilon U_{set}} sim(U_a, U_b)} \quad (2)$$

After the rating predictions are computed, the last step is to select the items to be recommended to the active user. These are the top n items (e.g. $n = 20$) with the highest predicted ratings.

III. IMPLEMENTATION OF NEIGHBOURHOOB-BASED COLLABORATIVE FILTERING RECOMMENDER SYSTEMS

A. Data storage for recommender systems

Recommender system data may be stored in a relational database. As an example, for the Movies recommender system, three tables may be used for users, items, and ratings data. However, several researchers (e.g. [6]) have observed that when there are very large numbers of users and items (e.g. millions), and a large number of ratings in the relational database tables, it is impossible or prohibitively expensive to perform the necessary computations for recommendations in real time. Alternatively, a ratings matrix is commonly used to store the ratings of items by users [5]. The matrix has one row for each user and one column for each item. Complex matrix operations are performed on the ratings matrix [5] in order to generate recommendations for the active user. There are two graph-based approaches for the implementation of a recommender system. The first approach uses the graph data structure which is implemented as adjacency matrices [1], [5]. The second approach uses a graph database [7], [8], [9].

B. Graph databases

Recommender systems data for millions of user item ratings and millions of items is categorised as Big Data. Two current solutions for storing and querying Big Data are NoSQL databases and New SQL databases [10]. Graph databases are one category of NoSQL databases. These databases have been used to store data for recommender systems [8], [9]. A graph database can store data entities and relationships using simple concepts derived from mathematical graph theory [11]. Nodes represent the graph data entities. Relationships are implemented as edges which connect the nodes (data entities). A graph database is ideal for storing data for a neighbourhood-based collaborative filtering recommender system because entities and relationships of interest are users, items to recommend, 'rating' relationships between users and items, 'similar' relationships between users, and 'similar' relationships between items [5].

Examples of graph database systems are Neo4j and Apache Spark [7], [11]. Neo4j is an open source NoSQL database system which supports ACID and database transactions [7], [11]. This database system was used for the research reported in this paper. Neo4j use labels and properties. Labels are used to categorise graph nodes. For example, the Movies database has two categories of nodes, namely user nodes and movie nodes. So, the node labels are 'User' and 'Movie'. Properties are attributes that are associated with data entities (nodes) and relationships, and are expressed as key-value pairs, e.g. (movieID: 1432). Neo4j provides a query language called Cypher, which is specifically designed for working with graph data. Cypher is a declarative language like SQL. It uses patterns to describe graph data, and it uses familiar SQL-like clauses.

C. Methods for speeding up computations for neighbourhood-based collaborative filtering systems

The main computations for user-based collaborative filtering were presented in Section II-B as equations (1) and (2). The computation of *similarity* suffers from two major problems. Firstly, the mean values for ratings for each user in equation (1) need to be computed each time the equation is computed. For the user-based approach, when there are many users (thousands or hundreds of thousands) who have rated the same items as the active user, then the computations of the mean values will take a significant amount of time. Secondly, the computation of equation (1) involves the computation of products, squares, and square roots. For the user-based approach, when there are many users who have rated the same items as the active user, then the computations of these terms will take a significant amount of time. The computation of predictions for equation (2) suffers from the following problem: the values of *similarity* have to be computed whenever equation (2) is evaluated. Several authors have reported that for user-based and item-based recommendation, various values can be pre-computed and stored in the recommender system database. Linden et al. [12] have reported that, for the item-based recommender system that Amazon.com was using in 2003, a similar-items table holding the similarity values for item pairs was created off-line. This resulted in major speed-ups for generating recommendations online.

IV. METHODS THAT WERE STUDIED FOR SPEEDING UP COMPUTATIONS

Graph databases were used for user-based collaborative recommender systems for item recommendations. The solutions for speeding up recommendations that were studied and are reported in this paper are as follows: (1) Each graph database consisted of two types of nodes: users and items, and two types of relationships: *RATED* and *SIMILAR*. For user-based recommendation, the *SIMILAR* relationship was between two user nodes. (2) For user-based recommendation the mean rating for each user was computed and stored in the user's graph node so that it did not have to be computed repeatedly. (3) The *similarity* values for user node pairs were computed based on equation (1). *SIMILAR* relationships were then created, and the *similarity* values were attached to the relationships. The *similarity* values that were stored were limited to be very high values (correlation >= 0.8). It should be noted that, in practice, the values of the mean ratings and *similarity* should be periodically updated. (4) Demographic data [13] was also used to determine its effect on the speed of computation for recommendations.

V. EXPERIMENTAL METHODS

A. *Objectives of the experiment*

The first objective of the experiments was to demonstrate that the methods presented in Section IV are easy to implement when a graph database is used to store the data for a neighbourhood-based collaborative recommender system. The second objective was to demonstrate that the methods provide a significant speed up in the computation of recommendations.

B. *Data and algorithms for the experiments*

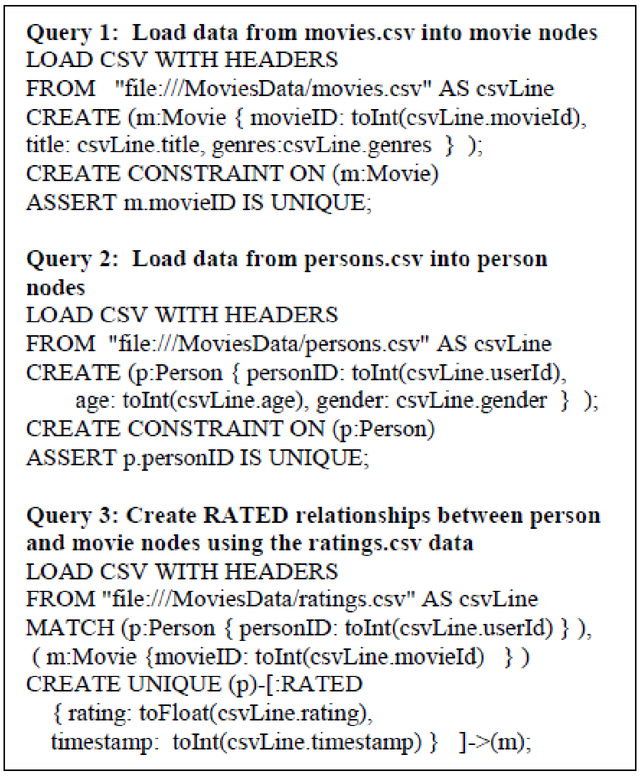
The MovieLens dataset (ml-latest-small) which was generated on 26 September, 2018, was used for the experiments [14]. The dataset consists of four files: movies.csv, ratings.csv, tags.csv and links.csv. The movies.csv file contains 9,742 records and has the attributes: movieId, title, genres. The ratings.csv file contains 100,836 records and has the attributes: userId, movieId, rating, timestamp. The range for the ratings is [0.5, 1, 1.5, 2,…, 4.5, 5]. The tags.csv file consists of 3,683 records, and the links.csv consists of 9,742 records. The files that were used for the experiments are: movies.csv, ratings.csv and persons.csv. The persons.csv file consists of 610 records and has the attributes: userId, age, gender. The userId values were obtained from the ratings.csv file. The age and gender attribute values were randomly generated. The data was stored in Neo4j Community Edition databases and the Neo4j Cypher query language was used for processing the data.

VI. EXPERIMENTS

A. *Creation of the Neo4j graph databases for the movies data*

For purposes of studying recommendation performance, two Neo4j graph databases were created. The databases are referred to as database 1 and database 2 in this discussion. Neo4j Cypher queries were used to load the data into the databases. For the two databases, nodes were created for each person (user) and each movie (item), and relationships of the form *p-RATED-m* were created between the person (*p*) nodes and the movie (*m*) nodes based on the data in the ratings.csv file. For database 2, mean values for ratings by each person were pre-computed and stored in the person nodes. *Similarity* values between person nodes were also pre-computed and relationships of the form *p1-SIMILAR-p2* were created between the person nodes (*p1*, *p2*) based on the data in the ratings.csv file. The computed *similarity* values were then attached to the *SIMILAR* relationships as properties. The *p1-SIMILAR-p2* relationships were only created for person node pairs with a *similarity* value of at least 0.8.

The Cypher queries for loading the data into the two databases are given in Fig. 1.



Query 1: Load data from movies.csv into movie nodes
```
LOAD CSV WITH HEADERS
FROM  "file:///MoviesData/movies.csv" AS csvLine
CREATE (m:Movie { movieID: toInt(csvLine.movieId),
title: csvLine.title, genres:csvLine.genres  }  );
CREATE CONSTRAINT ON (m:Movie)
ASSERT m.movieID IS UNIQUE;
```

Query 2: Load data from persons.csv into person nodes
```
LOAD CSV WITH HEADERS
FROM  "file:///MoviesData/persons.csv" AS csvLine
CREATE (p:Person { personID: toInt(csvLine.userId),
      age: toInt(csvLine.age), gender: csvLine.gender  }  );
CREATE CONSTRAINT ON (p:Person)
ASSERT p.personID IS UNIQUE;
```

Query 3: Create RATED relationships between person and movie nodes using the ratings.csv data
```
LOAD CSV WITH HEADERS
FROM "file:///MoviesData/ratings.csv" AS csvLine
MATCH (p:Person { personID: toInt(csvLine.userId) } ),
 ( m:Movie {movieID: toInt(csvLine.movieId)   } )
CREATE UNIQUE (p)-[:RATED
   { rating: toFloat(csvLine.rating),
   timestamp:  toInt(csvLine.timestamp) }   ]->(m);
```

Fig. 1: Cypher queries for loading data into the graph databases

Query 1 was used to read data from the movies.csv file, create movie nodes in the graph database, and then store the movies data in the nodes. Query 2 was used to read data from the persons.csv file, create person nodes in the graph database, and then store the persons data in the nodes. Query 3 was used to read data from the ratings.csv file, locate the person node and movie node for each data record, and create the *RATED* relationship in the graph database, for the (person, movie) pair.

The Cypher queries for storing pre-computed values in database 2 are given in Fig. 2. Query 4 was used to compute the average rating values and store them in the person nodes of database 2. Query 5 was used to compute *similarity* values between pairs of person nodes, create the *SIMILAR* relationships and store the *similarity* values as properties for the *SIMILAR* relationships in database 2.

```
Query 4: Compute and store the average ratings
            for database 2
MATCH (person1:Person)-[rel:RATED] ->(movie1:Movie)
WITH person1, SUM(rel.rating) AS sum_ratings,
         COUNT(movie1) AS count_ratings
SET person1.avg_rating =  sum_ratings / count_ratings
RETURN  person1.personID,person1.avg_rating

Query 5: Create SIMILAR relationships and store
           similarity values for database 2
MATCH (Ua:Person),(Ub:Person) WHERE Ua <> Ub
MATCH (Ua)-[r:RATED]->(movie)
    WITH AVG(r.rating) AS RUa, Ua, Ub
MATCH (Ub)-[r:RATED]->(movie)
     WITH AVG(r.rating) AS RUb, RUa, Ua, Ub
MATCH (Ua)-[Ra:RATED]->(movie)<-[Rb:RATED]-(Ub)
    WITH SUM((Ra.rating - RUa) * (Rb.rating - RUb))
           AS numerator,
           SQRT(SUM((Ra.rating – RUa)^2) *
                    SUM((Rb.rating RUb)^2))
           AS denominator,Ua, Ub
WHERE (denominator <> 0)  AND
            (numerator/denominator >= 0.8)
CREATE (Ua)-[:SIMILAR
           {sim:(numerator/denominator)}]->(Ub);
```

Fig. 2: Cypher queries for pre-computations for database

B. Computation of recommendations

Recall that the second objective of the experiments was to determine whether the methods proposed in Section IV provide a speed up in the computations for recommendations. To this end, experiments for generating recommendations were conducted on the two graph databases described in Section VI-A. Fig. 3 shows the query for generating recommendations from database 1. Fig. 4 shows the query for generating recommendations from database 2.

The main difference between Query 6 and Query 7 is that Query 6 has to compute the average ratings and *similarity* values before generating the recommendations, while Query 7 uses pre-computed average ratings and *similarity* values. Due to variability in execution times, each of the queries of Fig. 3 and Fig. 4 was executed 10 times for each of 10 selected person nodes. The <personID value> that were used for the queries were: 50, 100, 150, 200, 250, 300, 350, 400, 450, 500. These are the users personIDs for whom recommendations were generated.

Table I gives the mean values for the execution times of the queries for each of the 10 person nodes. Column 2 shows the mean execution times in milliseconds, for each of the 10 users for the execution of Query 6 on database 1 (method A). This is the database for which no pre-computed values are stored. On average, this method takes 2,102.6ms to generate 40 recommendations for a user.

```
Query 6: Compute recommendations from database 1
MATCH (Ua:Person),(Ub:Person)  WHERE (Ua <> Ub)  AND
          (Ua.personID =  <personID value>)
MATCH (Ua)-[r:RATED]->(movie)
        WITH AVG(r.rating) AS RUa, Ua, Ub
MATCH (Ub)-[r:RATED]->(movie)
     WITH AVG(r.rating) AS RUb, RUa, Ua, Ub
MATCH (Ua)-[Ra:RATED]->(movie)<-[Rb:RATED]-(Ub)
        WITH SUM((Ra.rating - RUa) * (Rb.rating - RUb))
              AS numerator,
              SQRT(SUM((Ra.rating - RUa)^2) *
                    SUM((Rb.rating - RUb)^2))
                    AS denominator,
                 Ua, Ub, RUa, Rub   WHERE denominator <> 0
      WITH  (numerator / denominator)
             AS similarity,  RUa, Rub, Ua, Ub
            ORDER BY similarity DESC  LIMIT 20
MATCH (Ub)-[Rb:RATED]->(movie)
WHERE (Ua <> Ub)
            WITH SUM(similarity * (Rb.rating - RUb) )
            AS numerat, SUM(similarity)  AS denominat,
               RUa, Rub, Ua, Ub, movie
RETURN movie.title AS Recommended_movie,
             RUa + (numerat / denominat)  AS Predicted_Rating
ORDER BY Predicted_Rating DESC LIMIT 40;
```

Fig. 3: Cypher query for generating recommendations from database 1

```
Query 7: Compute  recommendations from database 2
MATCH (Ua)-[S1:SIMILAR]->(Ub)
WHERE (Ua.personID = 50)
        WITH Ua, Ub, S1 ORDER BY S1.sim
                        DESC  LIMIT 20
MATCH (Ub)-[Rb:RATED]->(movie)
WHERE NOT (Ua)-[:RATED]->(movie)
     WITH SUM(S1.sim * (Rb.rating - Ub.avg_rating) )
            AS numerator, SUM(S1.sim)
            AS denominator, Ua.avg_rating
            AS Ua_avg_r, Ua, Ub, S1, movie
RETURN movie.title AS Recommended_movie,
            Ua_avg_r + (numerator / denominator)
            AS Predicted_Rating
ORDER BY Predicted_Rating DESC LIMIT 40;
```

Fig. 4: Cypher query for generating recommendations from database 2

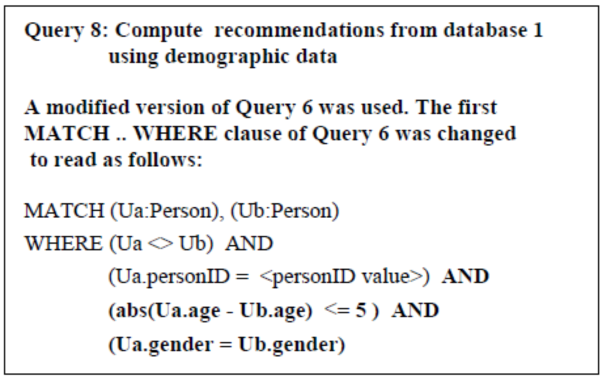
Column 3 shows the mean execution times in milliseconds, for each of the 10 users for the execution of Query 7 on database 2 (method B). This is the database for which the average rating and *similarity* values for person nodes were pre-computed and stored. On average, this method takes 137.6ms to generate 40 recommendations for a user. Column 4 shows the ratio of method A mean time to method B mean time (TMA/TMB) for each user. The values clearly indicate that it is much faster to generate recommendations using method B (database 2) compared to the use of method A (database 1). On average, method B is

15.3 times faster than method A.

TABLE I
MEAN EXECUTION TIMES FOR THE GENERATION OF RECOMMENDATIONS

PersonID of user U	Mean times for generation of 40 recommendations (milliseconds)		Method B is n times faster than Method A n = TMA/TMB
	Method A: Database 1 TMA	Method B: Database 2 TMB	
50	3406	132	25.8
100	2502	122	20.5
150	1708	136	12.6
200	3229	110	29.4
250	1511	147	10.3
300	1608	179	9.0
350	1682	182	9.2
400	1721	117	14.7
450	1712	134	12.8
500	1947	117	16.6
Mean	2102.6	137.6	15.3

A second experiment was conducted to assess the benefits of using demographic data in the generation of recommendations. Database 1 was used for this experiment. The query that was used to generate recommendations used criteria based on gender and age difference. The rationale here is that persons who are of the same gender and same age group as the active user are more likely to have the same tastes as the active user. Query 8 of Fig. 5 was used for the computation of recommendations.



Query 8: Compute recommendations from database 1 using demographic data

A modified version of Query 6 was used. The first MATCH .. WHERE clause of Query 6 was changed to read as follows:

```
MATCH (Ua:Person), (Ub:Person)
WHERE (Ua <> Ub)  AND
        (Ua.personID =  <personID value>)  AND
        (abs(Ua.age - Ub.age)  <= 5 )  AND
        (Ua.gender = Ub.gender)
```

Fig. 5: Cypher query for generating recommendations from database 1 using demographic data

Table II gives the mean values for the execution times of Query 6 and Query 8 for each of the 10 person nodes. Column 2 shows the mean execution times in milliseconds, for each of the 10 users for the execution of Query 6 on database 1 (method A). Column 3 shows the mean execution times in milliseconds, for each of the 10 users for the execution of Query 8 on database 1 (method C). On average, this method takes 568ms to generate 40 recommendations for a user. Column 4 shows the ratio of method A mean time to method C mean time (TMA/TMC) for each user. The values clearly indicate that it is faster to generate recommendations using method C compared to the use of method A. On average, method C is 4 times faster than method A.

TABLE II
MEAN EXECUTION TIMES FOR THE GENERATION OF RECOMMENDATIONS FOR METHODS A AND C

PersonID of user U	Mean times for generation of 40 recommendations (milliseconds)		Method C is R times faster than Method A by R times R = TMA / TMC
	Method A: Database 1 No limit to nodes: TMA	Method C: Database 1 Use demographic info: TMC	
50	3406	1117	3
100	2502	682	4
150	1708	350	5
200	3229	1007	3
250	1511	415	4
300	1608	364	4
350	1682	325	5
400	1721	463	4
450	1712	439	4
500	1947	520	4
Mean	2103	568	4

VII. CONCLUSIONS

The objectives of the experiments reported in this paper were to investigate methods for speeding up the computations for generating recommendations for the user-based collaborative filtering method. The data for generating recommendations was stored in a Neo4j graph database. Values that are required for the computations were pre-computed and also stored in the graph database. The results of the experiments have demonstrated that the methods proposed in Section IV, result in a faster computations for the generation of recommendations. The experimental results for the comparison of method A (database 1 with only *RATED* relationships) and database 2 (which additionally has *SIMILAR* relationships, *similarity* values and average rating values) demonstrated that when average ratings, *SIMILAR* relationships and *similarity* values are pre-computed, the speed of execution to compute recommendations is significantly increased. The experimental results for the comparison of method A (database 1 with only *RATED* relationships) and database 1 (using demographic data) demonstrated that the use of demographic data results in faster computation of recommendations. However, the computation is not as fast as method B which uses pre-computed values. The final conclusion is that, when data for generating recommendations is stored in a graph database, it is easy to pre-compute and store values that are needed for the generation of recommendations. This speeds up the real-time generation of recommendations.

REFERENCES

[1] F. Ricci, L. Rokach and B. Shapira, “Introduction to recommender systems”, in: *Recommender Systems Handbook*, F. Ricci, L. Rokach, B. Shapira, P.B. Kantor eds. New York: Springer New York, Dordrecht, Heidelberg, London, 2011, pp. 1-29.

[2] D. Jannach, M. Zanker, A. Felfernig and G. Friedrich, *Recommender Systems : An Introduction.* 1st ed. New York: Cambridge University Press, 2010.

[3] A. Felfernig, G. Friedrich, D. Jannach and M. Zanker, "Developing constraint-based recommenders", in: *Recommender Systems Handbook*, F. Ricci, L. Rokach, B. Shapira, P.B. Kantor eds. New York: Springer New York, Dordrecht, Heidelberg, London, 2011, pp. 187-212.

[4] M. Jones, "Recommender systems, Part 1: introduction to approaches and algorithms", *IBM DeveloperWorks*, 2013, pp. 1-8.

[5] C. Desrosiers and G. Karypis, "A comprehensive survey of neighborhood-based recommendation methods", in: *Recommender Systems Handbook*, F. Ricci, L. Rokach, B. Shapira, P.B. Kantor eds. New York: Springer New York, Dordrecht, Heidelberg, London, 2011, pp. 107-144.

[6] H. Lee and J. Kwon, "Efficient recommender system based on graph data for multimedia application", *International Journal of Multimedia and Ubiquitous Engineering*, vol. 8, no. 4, July 2013.

[7] I. Robinson, J. Webber, E. Eifrem, *Graph Databases*, 2nd Edition, O'Reilly Media Inc., 2015.

[8] F. Zi, N. Jin, L. Bi and J. Shen, "Design of movie recommender system based on graph database", Journal of Software Guide, vol. 15, pp. 144-146, 2016.

[9] N. Yi, C. Li, X. Feng and M. Shi, "Design and implementation of movies recommender system based on graph database", *Proceedings of the IEEE 14th Web Information Systems and Applications Conference*, 2017, pp. 132-135.

[10] A. B. M. Moniruzzaman and S. A. Hossain, "NoSQL Database: New era of databases for big data analytics - classification, characteristics and comparison", *International Journal of Database Theory and Application*, vol. 6, no. 4. 2013.

[11] M. Needham and A.E. Hodler, *Graph Algorithms: Practical Examples in Apache Spark and Neo4j*, O'Reilly Media Inc., California, USA, 2019.

[12] G. Linden, B.Smith and J. York, "Amazon.com recommendations: Item-to-item collaborative filtering", *IEEE Internet Computing*, Jan.Feb 2003, pp. 76-80.

[13] M.J. Pazzani, "A framework for collaborative, content-based and demographic filtering". *Artificial Intelligence Review* vol. 13, pp. 393–408, 1999.

[14] F. Maxwell Harper and J.A. Konstan, "The MovieLens Datasets: History and Context". *ACM Transactions on Interactive Intelligent Systems (TiiS)* vol. 5, no. 4, 2015 <https://doi.org/10.1145/2827872>