Recommendations as Graph Explorations

Marialena Kyriakidi*
marilou@di.uoa.gr
University of Athens and
Athena Research Center
Athens, Greece

Georgia Koutrika
gkoutrika@gmail.com
Athena Research Center
Athens, Greece

Yannis Ioannidis
yannis@di.uoa.gr
University of Athens and
Athena Research Center
Athens, Greece

ABSTRACT

We argue that most recommendation approaches can be abstracted as a graph exploration problem. In particular, we describe a graph-theoretic framework with two primary parts: (a) a recommendation graph, modeling all the elements of an (application) domain from a recommendation perspective, including the subjects and objects of recommendations as well as the relationships between them; (b) a set of path operations, inferring new edges, i.e., implicit or unknown relationships, by traversing and combining paths on the graph. The resulting path algebra model provides an abstraction and a common foundation that is beneficial to three aspects of recommendations: (a) expressive power - expression and subsequent use of several significantly different, existing but also novel recommendation approaches is reduced to parameterizing a unique model; (b) usability - by capturing part of the recommendation mechanisms in the underlying path algebra semantics, specification of recommendation approaches becomes easier and less tedious; (c) processing speed - implementing recommender systems on top of graph engines opens up the door for several optimizations that speed up execution. We demonstrate the above benefits by expressing several categories of recommendation approaches in the path algebra model and benchmarking some of them in a recommender system implemented on top of Neo4J, a widely used graph system.

CCS CONCEPTS

• **Information systems → Data management systems**.

KEYWORDS

recommender systems, modeling, path algebra

ACM Reference Format:
Marialena Kyriakidi, Georgia Koutrika, and Yannis Ioannidis. 2020. Recommendations as Graph Explorations. In *Fourteenth ACM Conference on Recommender Systems (RecSys '20), September 22–26, 2020, Virtual Event, Brazil.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3383313.3412269

*Currently at Google

RecSys '20, September 22–26, 2020, Virtual Event, Brazil

ACM ISBN 978-1-4503-7583-2/20/09…$15.00
https://doi.org/10.1145/3383313.3412269

1 INTRODUCTION

As recommendations become core components of many industrial systems to personalize/customize many experiences and their use continues its upward trend, the landscape of recommender systems is evolving very rapidly and becoming increasingly more diverse in three dimensions:

- *Information space*: A wealth of diverse types of individual or complex entities and relationships between them may be used as reference information to generate recommendations within an application. Users leave a great variety of online traces and the available data is heterogeneous, interconnected and growing in size significantly.
- *Goal*: Even on the same information space, different types of entities may play the role of recommendation *subjects* (receiving recommendations) and recommendation *objects* (being recommended) for different purposes. One may recommend a hotel to a traveler group or a flight-hotel-car package to a traveler (sales), users to each other (social networking), a potential member to a gym (targeted advertising), or an actor to a movie (casting).
- *Mechanism*: A rich set of old and emerging algorithms may be used to generate recommendations, from conventional content-based and collaborative filtering [31] to matrix factorization [21] and deep learning [24].

The increasing diversity in these dimensions has led to the development of an enormous number of recommenders that address seemingly different problems with seemingly different solutions. Every case is treated as distinct and handled from scratch, with a specific, possibly ad hoc algorithm providing a tailored solution for a specific task within a specific domain and information space. Hence, the resulting systems offer constrained solutions that cannot be re-used, generalized, or adapted to different or novel problem settings easily. This fragmentary methodology (or lack thereof) towards recommendations is problematic on several grounds: Much effort is replicated unnecessarily time and again; Any possible performance optimizations must be handcrafted in the code developed for the specific case at hand; Experimental comparison and evaluation of different algorithms is hindered by the sheer number of options and the ensuing uncertainty on the results' quality.

In this paper, we aim for a uniform model with the appropriate small set of abstractions to capture different recommendation elements effortlessly, explore connections in heterogeneous information spaces, facilitate algorithm design and evaluation, and lead to recommendation engines that scale well. Such a model is inspired by the success of Data Management Systems and is the first major step on the way towards the ultimate goal of the development of *Recommendation Management Systems*.

In particular, we propose *RecGraph*, a *graph-theoretic* approach to recommendations, consisting of a data model and a computation model. *The data model is a graph*: The nodes are entities relevant to the domain, including those that can be recommended or receive recommendations. The edges capture diverse relationships, such as domain-specific connections (part-of, associated-with, ...), states (friend-of, similar-to, ...), actions (rated, visited, ...), and attitudes and preferences (likes, trusts, matters, believes, ...). *The computation model is based on a path algebra:* Inspired by earlier works on path algebras [8], all processing is expressed using a very small set of *generic path operators*, which are instantiated to particular functions depending on the recommendation strategy and the domain at hand. Their nature follows a series-parallel paradigm, where some operators compose consecutive edges along a path and some aggregate or fuse edges between the same nodes. Associated with the path algebra is a *declarative Datalog-like query language* that uses these operators to write recommendation algorithms uniformly, as path-based derivations of new recommendation-related edges. Path operators can exhibit a desired *set of properties*, that can lead to performance optimization opportunities.

The fundamental benefit of *RecGraph* is that one can develop a single, fine-tuned, generic recommender system on top of a graph engine and then instantiate it into numerous specific recommenders through parameterization. The uniformity in the conception of recommendations thus afforded has several advantages. (a) *Expressive power*: The graph data model abstractions can capture, explore, and follow the evolution of diverse heterogeneous information spaces. The path computation model abstractions can capture different recommendation goals and a great variety of algorithms using the same uniform logic and a few operators. (b) *Usability:* The *RecGraph* language helps designers to focus more on the recommendation logic rather than the implementation details, which potentially leads to better and/or cleaner algorithms and faster development. (c) *Processing speed*: Using a state-of-the-art graph engine offers reliability and promises significant performance improvements, since graph databases are designed for data scalability. Furthermore, the optimizer of a graph engine can take advantage of the mathematical properties of the *RecGraph* operators to choose optimal executions independent of how they may be written.

In this paper, we demonstrate these benefits by expressing several recommendation methods in *RecGraph* and benchmarking some of them in a recommender system implemented on top of Neo4J, a widely used graph system.

2 RELATED WORK

Recommendation Systems. Given the focus of this paper, we limit our exposition to graph-related recommendation algorithms. In [14], a random walk model on a bipartite user-item graph is used to simulate a user's random selection on reaching target item nodes. In [19], a social approach computes recommendation scores for items based on direct and indirect relations between users, items and tags. In [28], a graph solution for group recommendation is modeled as an optimization problem over item affinities. In [35], heterogeneous information networks are used for rating prediction. Although these approaches use graphs for specific recommendation issues, our work differs substantially as we aim to build a *generic* framework that captures any recommendation task.

Graph Databases. Interest on traditional databases has shifted to graph databases, resulting in many graph processing systems similar, due to their common data modeling, but different on the graph features they aim to improve, e.g. reachability questions, graph patterns, other graph metrics. In [6], an algebraic framework is used for social analysis. We provide similar functionality for recommendations, but with a more concise operator set. In [18], the property graph model is extended and applied to transport networks. In [13], a model and SQL-like query language is given for semi-structured data, but it does not deal with data role transformations. In [20], a graph-oriented object model and language is proposed for end-user graphical interfaces. In [23], an object-oriented model and language is presented that uses hypergraph formalisms to model complex objects. In [11], the proposed graph data model and visual query language (a piecewise-linear, stratified Datalog) emphasize structural queries. The work in [33] extends [11] for social networks, mostly handling data management operations and graph transformations. In [16], a schema-less graph model extends property graphs and provides algebraic operators for online graph querying and analysis. Work in [27] offers a high-level declarative graph query language which includes extensions on the property graph model and relational graph algebra. In [32], a multi-relational graph model and formal algebra is proposed. Lastly, [22], provides a framework targeted towards the recommendation task where workflows are defined over structured, relational data, providing operators that can be compiled into standard SQL. Our work differs considerably from the above in that it focuses exclusively on recommendations, the types of nodes and edges in our graph data model carry semantics related to recommendations, and processing is expressed in a novel generic path algebra (and an associated Datalog-like edge-inference query language) that is based on a very small set of operator types that can be instantiated into any specific operators appropriate for the recommendation problem at hand, with great benefits on expressiveness, usability, and performance.

3 RECGRAPH FOR RECOMMENDATIONS

Given an application setting, we map all entities and interactions between them, that are interesting from a recommendation perspective, as nodes and edges on a graph, respectively. Next, we provide a formal definition of the graph model and its core elements.

3.1 RecGraph Model

A $RecGraph(V, E, L_V, L_E, A_V, A_E)$ is a graph where

- V is the set of nodes and E is the set of edges. Any entity directly or indirectly affecting the recommendation, including any recommendable entity, is a node, and any ontological (e.g., reviewed, part-of) or attitudinal (e.g., likes) connection between them is an edge.
- L_V and L_E are the label sets for nodes and edges and A_V and A_E the attribute sets for nodes and edges, respectively. Each node or edge label l is associated with a set of attributes $A_V^l = \{a_v | a_v \in A_V\}$ or $A_E^l = \{a_e | a_e \in A_E\}$ respectively.

- Nodes and edges represent either specific objects (instances), e.g., a specific user review, or object classes (schema), e.g., the class of all user reviews.

A node v

- has a label $l \in L_V$ denoting its type and is identified by a unique identifier id,
- has a value w for each attribute $a_v \in A_V^l$, which is depicted as a $\langle a_v : w \rangle$ pair, and is drawn from its domain $dom(a_v)$,
- is denoted as $l(id, \{a_i : w_i\})$ or $l(id, a_1 : w_1, ..., a_k : w_k)$ or simply $l(id)$ depending on the context or recommendation focus.

For instance, a *rating* attribute can have a domain $dom(rating) = \{1, 2, 3, 4, 5\}$, while a *review* node with two attributes can be written as $review(1, rating : 3, review : sometext)$ or $review(1)$, if its attributes are omitted.

A directed edge $e(v, u)$, where v is the start and u is the end node,

- has a label $l \in L_E$ denoting its type,
- has a value w for each attribute $a_e \in A_E^l$, which is depicted as a $\langle a_e : w \rangle$ pair, and is drawn from its domain $dom(a_e)$,
- is denoted as $l(u, v, \{a_i : w_i\})$ or $l(u, v, a_1 : w_1, ..., a_k : w_k)$ or simply $l(u, v)$, depending on the context or recommendation focus.

As an example, a *likes* edge from *user* to *business* with two attributes can be written as *likes*$(user(i), business(j), score:$ 0.9, *uncertainty:* 0.3), while a *reviewed* edge from *user* to *business* with its attributes omitted as *reviewed* $(user(i), business(j))$. Note that we can traverse a directed edge $l(u, v)$ in the inverse direction. We will denote this as $-l(v, u)$.

Yelp Recommendation Example Graph. We use the Yelp dataset to create a recommendation graph. A Yelp recommendation graph can include user, business, review, photo and tip nodes. Interactions between these entities result in action (e.g., wrote, reviewed), state (e.g., friends) and attitude (e.g., likes) relations. There is no single correct way to represent a domain, consequently, we present one possible model. For schema simplicity, we focus on the part of the graph where a user reviews a business.

The Yelp recommendation graph is presented in Figure 1. Nodes $U(ser)$, $R(eview)$, $RA(ting)$, $B(usiness)$, $L(ocation)$, $T(ag)$, $C(ontext)$ are schema nodes. Edges that appear dotted are schema edges, i.e. *reviews*, *wrote*. Every other node and edge in the figure is an instance one. Schema and instance nodes are connected with *in* relationships. Node/edge ids are omitted and label types (e.g. $User$, $Review$, etc) appear in bold only on the schema level. Attributes are also generally omitted, but we keep a few to use as examples in the instance level, e.g. $rating : 4$, $name : Kenko$, as text inside brackets. According to our definition, every node in the graph can be recommended to other nodes. In that sense, specific businesses (i.e $Kenko$ business node) are nodes, but other information types can be as well (i.e. *Japanese* tag node). Notice that in the same graph, *context* nodes also appear. Generally, we model and treat context the same way as any other node. However, we will not focus on contextual recommendations as they are out of the paper's scope.

We observe that *answering a recommendation question* (e.g "Does a user u likes a business b") *transforms into a reachability question between two nodes in the graph* (i.e. "Is there a path between nodes u and b?"). Consequently, by changing the start and end nodes, we change the recommendation question. In addition, expressing both schema and instance nodes in the same graph allows inference not only on instance queries, but also on more generic ones (i.e. "Does a user u writes any reviews R?"). Accommodating other recommendation types, such as group and package recommendations, can be treated the same way. Reachability questions can be answered by traversing more than one paths. To choose which paths to traverse, what types of node and edge information to include in the computations, and how to combine them, we propose a set of path operators. We kept the operator set small and concise: we create paths and aggregate them as we implicitly link the start (subject) and end (object) nodes.

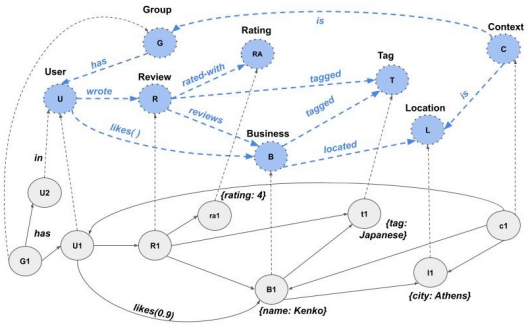

Figure 1: Part of Yelp Graph.

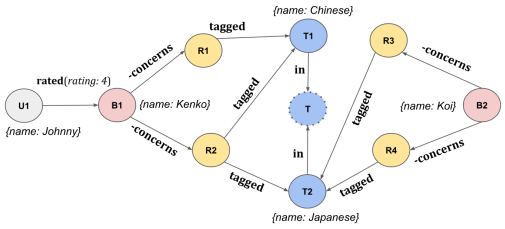

Figure 2: Example graph for CB recommendation.

3.2 RecGraph Operators

A recommendation on the graph is the result of path traversal computations. The operators are used for edge inference. We concluded on three, namely *CON*, *AGG*, and *FUSE*, (figures 3, 4, 5) that are inspired from path algebra [8] but are tailored to the recommendation problem. *CON* is used for traversing consecutive edges, while *AGG* and *FUSE* are used for combining multiple edges between the same nodes. While navigating the graph we perform concatenations of edges and aggregations of paths, and perform computations using information on node and edge labels. More formally:

Edge Concatenation (CON).
$CON_f(l_1(s, u_1, \{a_1 : w_1\}), ..., l_n(u_{n-1}, t, \{a_n : w_n\})) = l(s, t, \{a_j : w_j\})$: It operates on n consecutive edges $l_1(s, u_1, \{a_1 : w_1\}), ..., l_n(u_{n-1}, t, \{a_n : w_n\})$ and produces an edge $l(s, t, \{a_j : w_j\})$. It uses the attributes and the labels of the input edges to produce a new edge based on a function f, where $CON_f : L_E \times L_E \times ... \times L_E \rightarrow L_E$, $2^{A_E} \times 2^{A_E} ... \times 2^{A_E} \rightarrow 2^{A_E}$.

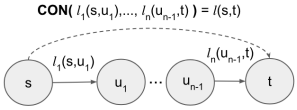

Figure 3: CON Operator Type

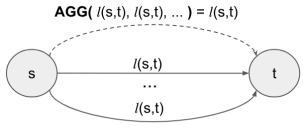

Figure 4: AGG Operator Type

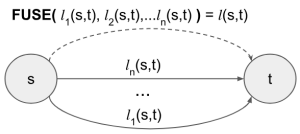

Figure 5: FUSE Operator Type

Path Aggregation (AGG).
$AGG_f(\ \{l(s,t,\{a_l : w_l\})\}\ ) = l(s,t,\{a_l : w_l\})$: it operates on a set of edges $\{l(s,t,\{a_l : w_l\})\}$ of the same label $l \in L_E$ between nodes s and t, and produces an edge $l(s,t,\{a_l : w_l\})$ of the same type. It uses the attributes and labels of the input edges to produce a new edge based on a function f, where $AGG_f : 2^{L_E} \rightarrow L_E, 2^{A_E} \rightarrow A_E$.

Path Fusion (FUSE).
$FUSE_f(\ l_1(s,t,\{a_1 : w_1\}), .., l_n(s,t,\{a_n : w_n\})\ ) = l(s,t,\{a_j : w_j\})$: it operates on an n-tuple of edges between nodes s and t with different labels $l_i \in L_E$, and produces an edge $l(s,t,\{a_j : w_j\})$ of a label $l \in L_E$. It handles the attributes and labels of the input edges to produce a new edge based on a function f, where $FUSE_f : L_{E_1} \times L_{E_2} ... \times L_{E_n} \rightarrow L_E, 2^{A_{E_1}} \times 2^{A_{E_2}} ... \times 2^{A_{E_n}} \rightarrow A_E$.

3.3 Recommendations as a Path Problem

Every time we want to make a recommendation, we are trying to infer an edge between the *subject node* (i.e. the node that receives recommendations) and the *object* node (i.e. the object that can be recommended). Intuitively, we define all reachable paths between subject and object, and combine them. More formally:

Given two nodes u, v in graph G, a recommendation of v to u depends on (a) deriving an implicit edge from u to v through some combination of the paths connecting them and (b) assessing the values of the edge labels so predicted.

$$predict(u,v) = (AGG_f | FUSE_f)\{path_{u,v} = CON_f\}_i \qquad (1)$$

where CON_f operators traverse paths from u to v and compose them in edges, while AGG_f and $FUSE_f$ operators aggregate the resulting homogeneous or heterogeneous edges, respectively, to derive the final edge predicted. Each operator may be instantiated by its own function f

3.4 RecGraph Operator Properties

In traditional path problems, operators can exhibit a set of properties. Below we present the ideal setting for our model. However, which properties hold, depends on the implemented algorithm, i.e. the edge labels and operator functions. Generally, the more properties, the better the optimization opportunities.

- CON_f is *associative*. $\forall s,u_1,u_2,t. : CON_f(CON_f(l_1(s,u_1), l_2(u_1,u_2)), l_3(u_2,u_3))) = CON_f(l_1(s,u_1), CON_f(l_2(u_1,u_2), l_3(u_2,u_3))))$
- AGG_f is *commutative*. $\forall s,t. : AGG_f(l^{(1)}(s,t), l^{(2)}(s,t)) = AGG_f(l^{(2)}(s,t), l^{(1)}(s,t))$ where $l^{(1)} = l^{(2)} = l$
- AGG_f is associative. $\forall s,t. : AGG_f(\ AGG_f(l^{(1)}(s,t), l^{(2)}(s,t)), l^{(3)}(s,t)) = AGG_f(l^{(1)}(s,t), AGG_f(l^{(2)}(s,t), l^{(3)}(s,t)))$, where $l^{(1)} = l^{(2)} = l^{(3)} = l$
- $FUSE_f$ is associative. $\forall s,t. : FUSE_f(FUSE_f(l_1(s,t), l_2(s,t)), l_3(s,t)\ ) = FUSE_f(\ l_1(s,t),\ FUSE_f(l_2(s,t), l_3(s,t))\ )$
- $FUSE_f$ is commutative. $\forall s,t. : FUSE_f(\ l_1(s,t), l_2(s,t)\ ) = FUSE_f(\ l_2(s,t), l_1(s,t)\ )$
- CON_f is *distributive* over $AGG_{f'}$. $\forall s,u,t. : CON_f(\ l_1(s,u), AGG_{f'}(l^{(2)}(u,t), l^{(3)}(u,t))) = AGG_{f'}(CON_f(l_1(s,u), l^{(2)}(u,t)), CON_f(l_1(s,u), l^{(3)}(u,t))\ )\ CON_f(\ AGG_{f'}(l^{(2)}(u,t), l^{(3)}(u,t))\ , l_1(s,u)) = AGG_{f'}(CON_f(l^{(2)}(u,t), l_1(s,u)), CON_f(l^{(3)}(u,t), l_1(s,u))\ )$, where $l^{(2)} = l^{(3)} = l$
- CON_f is *distributive* over $FUSE_{f'}$. $\forall s,u,t. : CON_f(\ l_1(s,u), FUSE_{f'}(l_2(u,t), l_3(u,t))\ ) = FUSE_{f'}(\ CON_f(l_1(s,u), l_2(u,t)), CON_f(l_1(s,u), l_3(u,t))\ )CON_f(\ FUSE_{f'}(l_2(u,t), l_3(u,t)), l_1(s,u)\ ) = FUSE_{f'}(\ CON_f(l_2(u,t), l_1(s,u)),\ CON_f(l_3(u,t), l_1(s,u))\ )$

3.5 RecGraph Optimization Rules

An algorithm in RecGraph is a sequence of operators, that a graph engine translates to an execution plan. The optimizer of the engine creates alternative candidate plans and executes the most efficient. Given the operator properties and database statistics, the optimizer can apply *rules to transform a plan to equivalent* ones. We formally present the rules below. We will see them in action in Section 6.

- *Rule 1.* An *n-ary CON* operator could be decomposed in *CON* operators of smaller sizes, and vice versa, i.e. consecutive *CON* operators can be composed in a single *n-ary CON* operator. The decomposition can result in at most $n-1$ 2-*ary CON* operators. The newly created operators implement their own functions that collectively result in the same output that the function of the *n-ary CON* produced, depending on the semantics of the labels that the operator affects. $CON_f(\ l_1(s,u_1), ..., l_n(u_{n-1},t)\ ) = CON_{f'}(CON_{f1}(l_1(s,u_1), l_2(u_1,u_2)), \ldots, CON_{fn}(l_{n-1}(u_{n-2},u_{n-1}), l_n(u_{n-1},t)))$.
- *Rule 2.* An *AGG/FUSE* operator that is imposed on a *n-ary CON* could be decomposed in multiple *AGG/ FUSE* operators over each created *CON* and vice versa, i.e. consecutive pairs of *CON* and *AGG/FUSE* operators can be composed in a single pair of an *n-ary CON* and *AGG/FUSE* operators. The newly created operators collectively implement the same function that the initial operators produced. $AGG_f(\{CON_{f'}(l_1(s,u_1), ..., l_n(u_{n-1},t))\}) = CONf''(AGG_{f1}(\{CON_{f'1}(l_1(s,u_1), l_2(u_1,u_2))\}), \ldots, AGG_{fn}(\{CON_{f'n}(l_{n-1}(u_{n-2},u_{n-1}), l_n(u_{n-1},t)\ )\}))$
- *Rule 3.* A *FUSE* operator could be reduced in *AGG* operators of different functions, if the underlying *CON* operator and its output changes. $FUSE_f(l_1(s,t), .., l_n(s,t))= AGG_{f1}(AGG_{f2}(...(AGG_{fi}\{l_i(s,t)\})))$.

Intuitively, the more uniform the graph (i.e. same semantics) and the more properties the functions have (e.g., addition, multiplication, etc, exhibit algebraic properties), the better for optimization purposes. In Recgraph normally, imposing *AGG* and *FUSE* operators as early as possible is a good strategy, as it reduces the number of created paths. However, it may make more sense to postpone aggregations when the output degree of the nodes is not very large; graph engines can perform path traversals very fast, so if the paths are long but few, it is faster to traverse first by using *CON* operators and aggregate later with *AGG*/*FUSE*.

3.6 RecGraph Query Language

In our framework, an algorithm is expressed as a combination of operators and is processed as high-level, extended Datalog-like rules. We introduce our query language below. Note that we use lowercase letters as variables in our rules.

```
% CON rule.
label(node_1,node_n;⟨attribute_j⟩) :=
 label_1(node_1,node_2;⟨attribute_i1⟩),...,label_n(node_n-1,node_n;⟨attribute_in⟩);
 attribute_j = function_CON(⟨attribute_i1⟩,..., ⟨attribute_in⟩).
% AGG rule.
label(node_1,node_2;⟨attribute_k⟩) :=
 [ label(node_1,node_2;⟨attribute_ki⟩); attribute_k = function_AGG
       (⟨attribute_k1⟩)].
% FUSE rule.
label(node_1,node_2;⟨attribute_k⟩) :=
 [ label_1(node_1,node_2;⟨attribute_i⟩),..., label_n(node_1,node_2;⟨attribute_j⟩);
   attribute_k = function_FUSE(⟨attribute_i⟩,..., ⟨attribute_j⟩) ].
```

4 RECOMMENDATION METHODS WITH RECGRAPH

In this section, we demonstrate the expressive power of *RecGraph* by using its primitives to model two important and distinctly different families of recommendation methods: similarity-based and matrix factorization methods. We also hint on the same for neural network-based methods, but leave a detailed account of their unification into the common model as future work, partly because their effectiveness for recommendations in comparison to other methods is still being studied, if not actually challenged, as pointed out recently [12].

Similarity-based Methods. Classical recommendations involve methods based on user/item similarities and some form of user feedback (e.g., ratings). In these methods, first a similarity function is defined and computed, and afterwards, a method to estimate user preference, which range from a simple weighted average to a regression model. These models rely on detecting strong associations of closely related objects within small sets. In our model, such associations are equivalent to paths in a graph. Depending on the type of the explored edges, different approaches are captured: user similarity edges lead to collaborative filtering, item similarity edges to content-based filtering, user interaction edges to social recommendations, etc. Representative methods that can be thus expressed include [34] , [25], [31], [10], [30], [29] and [5], some of which are also benchmarked in section 7.

Below, we choose a classical and easy to follow content-based method to demonstrate modeling of this category. We map our data onto the recommendation graph of Fig. 2. We want to derive a preference score from user *Johnny* $U1$ to business *Koi* $B2$ by using ratings and reviews of businesses that are most similar to $B2$ and *Johnny* visited in the past. A business is connected to review nodes and a review to tag nodes. The schema can be richer, but we omit irrelevant information and focus on the elements we need for the computation. In this example, we have one user, two businesses with two reviews each, and a total of two keywords, namely *Japanese* and *Chinese*. We describe an algorithm that derives *similarity* edges between business nodes and predicts a preference edge *likes* from the user node towards this business taking the weighted average of user ratings and business similarities. We provide the rules in separate boxes based on their functionality:

```
% AGG on 'likes' from CON on 'rated','similarity'
likes(U(i),B(j);preference):= [ (
      rated(U(i),B(k);rating=r1),
similar(B(k),B(j); similarity=s1)
      );preference=weighted_avg(r1, s1) ].
```

Next, we derive business similarity as the Jaccard coefficient between tags. For this, we take the union and the intersection of the tag sets for the two business nodes. We compute the common tags by traversing path *-concerns,tagged,-tagged,concerns* and their union through path *-concerns,tagged,in,-in,-tagged,concerns*.

```
% FUSE for calculating Jaccard similarity
similar(U(i),B(j);similarity):= [
      common(B(i),B(j);commontags=ct),
      all(B(i),B(j);unionTags=ut);
      similarity=jaccard(ct,ut)].
% CON for common tags and CON for all tags
common(B(i),B(j);commontags):=
      -concerns(B(i),R(k)),tagged(R(k),T(p)),
      -tagged(T(p),R(l)),concerns(R(l),B(j));
      commontags=union(T(p,name)).
all(B(i),B(j);unionTags):=
      -concerns(B(i),R(k)),tagged(R(k),T(p)),
      in(T(p),T(m)),-in(T(m),T(n)),
-tagged(T(n),R(l)),concerns(R(l),B(j));
      unionTags=union(T(t,name)).
```

Matrix Factorization Methods. Matrix factorization (MF) leads to a product of matrices and includes a variety of matrix decompositions. In recommender systems, it corresponds to a class of collaborative filtering algorithms that decompose the user-item matrix into the product of two lower dimensionality matrices. Interactions between users and items various explicit and implicit types, with ratings being the most classical. We capture such interactions as edges in our model. Representative methods that can be thus expressed include [15], [21],[26] [9], [17], [36]. To demonstrate modeling in this category we choose an algorithm popularized by Simon Funk [15] during the Netflix Prize competition [1], commonly referred to as *Funk SVD*. Typically, MF computes the low-rank approximation by minimizing the squared error loss. Among various possibilities, the most common way is Stochastic Gradient Descent. We decompose the problem into two phases, prediction and training, and show that both phases can be expressed as path problems. Assume a user-ratings matrix (2x3 here) on businesses with several missing entries. We want to predict the preference score of user $U1$ for business $B2$. After applying MF, the resulting matrices are presented in Figure 6. The same information in our model is shown in Figure 7. We consider the product of the two matrices as edges that denote affinity (i.e. labeled *has*) towards the latent factors,

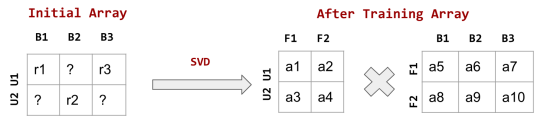

Figure 6: Traditional SVD decomposition in arrays

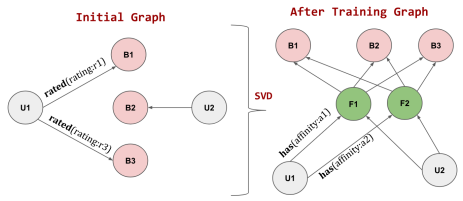

Figure 7: SVD decomposition in graphs

which are newly created nodes in the graph. The operators for the *prediction part* are quite straightforward and follow the graph in Figure 8. Without loss of generality, if we ignore the bias terms, the predicted rating is calculated as $prediction(u,b) = \sum_f a_{uf} a_{fb}$, i.e., as matrix multiplication that can be represented with the following rules.

```
% AGG and CON rules for rating prediction
prediction(U(i),B(j);score):=[ (
     has(U(i),F(k);affinity=a1),
     -has(F(k),B(j);affinity=a2); rating=a1*a2);
     score=addition(rating)].
```

For the *training part* we have two options. We can either make calls to an external library to first train the graph and apply the result in our model, or we can implement our own training operators. Showcasing the latter, we create the graph in Figure 9. In the schema, we only show edges relevant to a single user-item training example. We again ignore the regularization term. Learning rate g and affinity edges are initialized to some selected values. We have an *error* edge to derive the prediction error, and use it with the corresponding affinity edges to update them. The update rules are those of stochastic gradient descent (SGD). We present the rules for updating the first array, i.e. user-feature, as the rules for updating the second one are similar.

```
% Rules for training User-Feature edges
% FUSE for computing error
error(U(i),B(j);err):= [ ( rated(U(i),B(j);rating=r1),
     prediction(U(i),B(j);score=s1 ); err=r1-s1 ].
% Update AGG and CON rules for edges has(u,f)
has(U(i),F;affinity):= [
( error(U(i),B(j);err=e), has(B(j),F;affinity=a);
     affinity=g*e*a );affinity=addition(affinity) ].
```

Neural Network Methods. As mentioned earlier, unification of these methods under RecGraph is part of our current and future work. The graph-based nature of neural networks makes capturing the inference part of these methods straightforward. Initial results on capturing the training part are quite promising, with several examples having been successfully unified (similarly to the Funk SVD example), but the overall effort is still in progress.

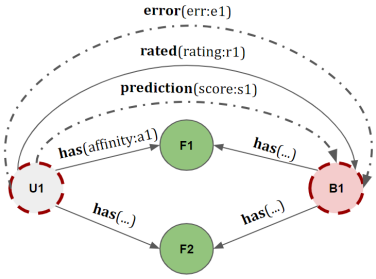

Figure 8: Prediction graph

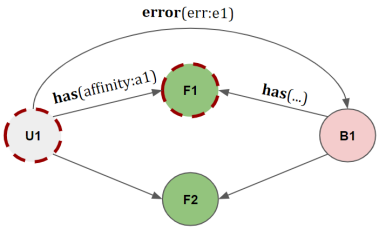

Figure 9: Training graph

5 RECGRAPH SYSTEM

An overview of the basic components of the first version of Recgraph in Figure10. RecGraph is written in Java and runs on top of Neo4j[4], a graph database management system with native graph storage and processing. Any graph database system could play the role of the graph engine, but we chose Neo4j because it is widely used and well documented. The input of the RecGraph system is a program in the RecGraph language. This is sent to the Parser, which reads the operators as a sequence and generates an AST (abstract syntax tree). This is passed on to the Optimizer, which exploits any algebraic properties the operators may have to modify their execution sequence and generate an equivalent AST with potentially much better or even optimal performance. The final AST is passed on to the Execution Engine, which interprets it and communicates with the graph engine underneath. Querying a Neo4j database can be written either in Cypher, a declarative SQL-like language built specifically for it, or in several different programming languages at a lower level. We chose Java to have more control over how to best execute queries using the logic of our operators.

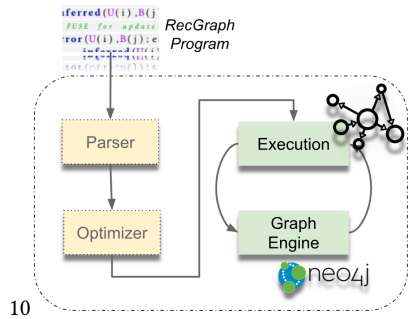

Figure 10: RecGraph Overview

6 BENEFITS OF THE RECGRAPH APPROACH

In this section, we elaborate more on the advantages of the *RecGraph* approach towards a unified recommendations model. Since a recommendation process consists of user-system interactions, we aim to improve on both fronts. Expressivity and usability benefit the user side, while processing speed the system side.

(a) *Expressivity*: The abstraction of the graph data model can capture the heterogeneous recommendation space and adapt to its changes (addition/removal of entities does not impact the whole schema). The path computation model abstractions can capture different, even non-traditional recommendation goals and a great variety of algorithms using the same uniform logic with a few operators in a single framework. For example, one may recommend a hotel to a traveler group or a flight-hotel-car package to a traveler (sales), users to each other (social networking), a potential member to a gym (targeted advertising), or an actor to a movie (casting).

(b) *Usability*: The uniform approach enables the easy experimentation and comparison of different ad-hoc approaches, which is a critical aspect of the recommendation development process and currently can be a tedious task, with the results of such efforts questionable. The *RecGraph* language is declarative and helps designers to focus more on the recommendation logic rather than the implementation details, which can potentially lead to faster development and to better and/or cleaner algorithms. In addition, the amount of code needed for a solution is shorter (as it happens with logic-based languages) and easier to understand. To demonstrate this, we present an example of solutions for the same problem with different frameworks. Assume we are simply calculating business similarity, where *business* nodes are connected to *category* nodes that represent their tags. Our approach is presented below.

```
% RecGraph computing business similarities
similar(U(i),B(j);score):= [ common(B(i),B(j);
     commontags=ct), all(B(i),B(j);uniontags=ut);
     score=jaccard(ct,ut)].
common(B(i),B(j);commontags):= tagged(B(i),C(p)),
     -tagged(C(p),B(j)); commontags=union(C(p,name)).
all(B(i),B(j);uniontags):= tagged(B(i),T(p)),
     in(C(p),C(m)),-in(C(m),C(n)), -tagged(C(n),B(j));
     uniontags=union(T(t,name)).
```

As a direct comparison, we provide a solution from a non-graph model, assuming the original Json format of the Yelp dataset. We code in Madis[3], an open-source tool that extends relational systems with User Defined Functions, because it allows easy manipulation of json files, and uses SQL with Python extensions, which are both popular tools.

```
% Madis for computing business similarities
output 'businesscategories.csv' select
     '"'||c1||'"','"'||comprspaces(c4)||'"' from (select
     c1,jsplitv(t2j(regexpr("\,",c2,"\t"))) as c4 from
     (select jsonpath(c1,"business_id","categories")
     from (file 'business.json')) where c2 is not null);
% Create vectors and compute similarities
create table b_vector as select C1 as business,
     jgroup(C2) as categories from (file
     'businesscategories.csv') group by C1;
select bid1,bid2, jaccard(c1,c2) from (
select t.business as bid1, s.business as bid2,
     t.categories as c1, s.categories as c2 from
     b_vector as t, b_vector as s where s.business !=
     t.business);
```

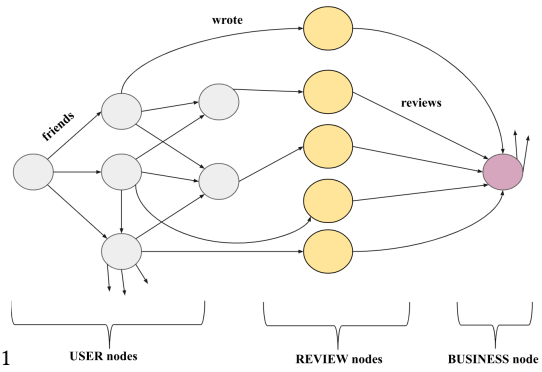

Figure 11: Social restaurant recommendation subgraph

Additionally, we have performed an initial user study on 10 users to test systematically the above claims. The results are encouraging and we are currently designing an extensive experiment with a much larger user set.

(c) *Processing speed*: Using a state-of-the-art graph engine offers reliability and promises significant performance improvements, since graph databases are designed for interlinked data scalability. Furthermore, the optimizer of a graph engine can take advantage of the mathematical properties of the *RecGraph* operators to produce alternative plans, independently of how they may be written, and select the optimal one for execution. Note that most graph engines are designed for generic graph processing and handle deep traversals well. However, occasionally recommendation problems can result in more segmented, shallower traversals, due to the heavy semantics on the graph. Hence, we have designed our operators specifically for the recommendation field, allowing tailored optimizations whenever possible. To illustrate how an optimizer can create an alternative plan, we use a Yelp subgraph (Figure 11) that contains *users*, *reviews*, *businesses* and *categories*. Users are *friends*, *write* reviews that *review* businesses in one or more categories. We want to *"recommend restaurants that my friends or the friends of my friends have reviewed"*. For the solution we assume that: (a) business preference is calculated by averaging scores from each friend or friend-of-friend, (b) scores are computed by reducing the original ratings of other users, and (c) a score from a direct friend is stronger than that of a friend of friend.

One way to solve this, is to traverse and combine the paths: (a) *friends*, *wrote*, *reviews* and (b) *friends*, *friends*, *wrote*, *reviews*. This approach is depicted in the execution plan of Figure 12. In the annotations next to the operators in the figure, we describe the implemented function of the current operator, and on the arrows the input/output edges of each one. However, due to the structure of the current problem (the *friends*-connected subgraph is denser than the *reviews* subgraph), when the above plan is executed, a great number of paths is explored, many of them visiting the same node through different paths. This could have been avoided if we looked at the available information: i.e. the specific *business* node has a small *reviews* in-degree, i.e. few people rated it, and the schema of the graph, i.e. intermediate *AGG* operations is possible on *user* and *review* nodes. We will not produce the rules that result in the

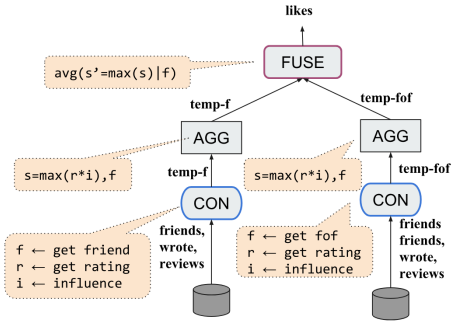

Figure 12: Initial execution plan

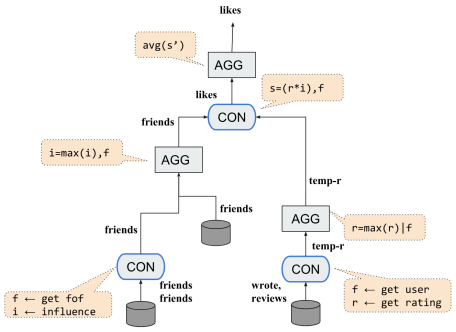

Figure 13: Alternative execution plan

optimized plan of Fig.13, but note that by using statistical information (e.g, node in-degree, subgraph density), operator properties, and transformation rules, we (and subsequently the optimizer) can push *FUSE*/*AGG* operators down the execution plan to reduce the amount of produced paths. We benchmark these plans in the next section.

7 EXPERIMENTAL EVALUATION

Experimental Setup. We evaluate RecGraph on the Yelp Open Dataset[2]. The dataset contains a total of 1,637m users, 193k businesses, 6,686m reviews, with additional descriptive information on users and businesses. We chose Yelp because it is semantically rich, and can be used to showcase different use cases that involve complex paths. We implemented four algorithms; each one answers a different recommendation question. We are interested in measuring how much performance can improve if optimizations are imposed, and as the underlying graph increases in complexity. For every algorithm there is a figure measuring performance that shows three clusters, namely *RecGraph 1*, *RecGraph 2*, and *Cypher*. Each cluster is an implementation of the algorithm. *RecGraph 1* and *RecGraph 2* are solutions using with our language. *RecGraph 1* corresponds to an initial solution, while *RecGraph 2* to the one after optimization. *Cypher* provides a solution using cypher queries. We considered using other systems, but concluded that cypher was a more appropriate language to use for comparison, since it is Neo4j's original language and since we built our framework on top of Neo4j. Each one of the clusters *Recgraph 1*, *Recgraph 2*, and *Cypher*, includes

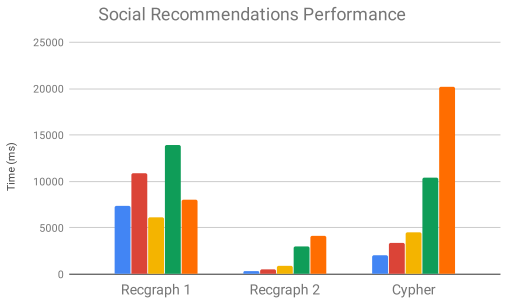

Figure 14: Running times for Social

five bars that measure performance using the same implementation. Moving from left to right (i.e. from the blue bar to the orange one), each bar measures performance in subgraphs of increasing complexity, i.e. longer paths, larger number of paths, nodes, etc. For every algorithm we measure the average time that the system will take to make an estimation on the preference.

Social Recommendation Case: Our goal is to *"recommend restaurants that my friends or the friends of my friends have reviewed"*. We will use the problem setting of Section 6. In Figure 14, we view the running times of the algorithms. Performance depends on the density of the friends network and on the average number of reviews. The imbalance between the number of friends and the number of reviews favors *Recgraph 2* that calculates fewer, smaller paths. It takes advantage of the intermediate aggregations of the paths and thus performs better. The Social Recommendation use case is an excellent example of how to exploit heavy aggregations on a partially homogeneous dense network (such as the friend subgraph).

Content-Based Case: We now want to *"recommend restaurants that are similar to the ones that the user rated highly in the past"*. A business is described by a set of tags and similarity is calculated with Jaccard. Complexity in this setting arises from the number of different tags that describe a business. Since a business is described by a small set of keywords (that does not vary too much from business to business), intermediate aggregations in the paths are not as critical as in the Social Recommendation case. *Recgraph 2* now skips intermediate aggregations, computes longer paths and performs better (see Figure 15), as it takes advantage of the fast traversals that graph engines offer.

Collaborative Filtering Case: Our goal is to *"recommend restaurants that users, similar to our user, have rated highly in the past"*. We compute user similarity by measuring the cosine similarity between review ratings. Complexity in this setting arises from the number of reachable businesses and users, since it involves the exploration of many different paths that do not necessarily lead to the business we intend to evaluate. As seen in Figure 16, *Recgraph 2*, performs better because it changes the original plan and imposes intermediate aggregations earlier.

Even though both content-based and the collaborative filtering cases are conceptually the same, i.e. first calculate similarities and then prediction scores, the execution changes because of the differences in the graph they traverse. In one case, the longer paths are exploited first, while in the other case intermediate aggregations are exploited first.

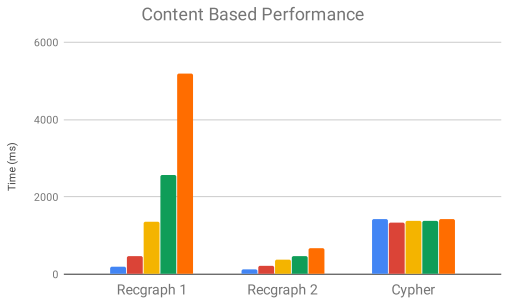

Figure 15: Running times for CB

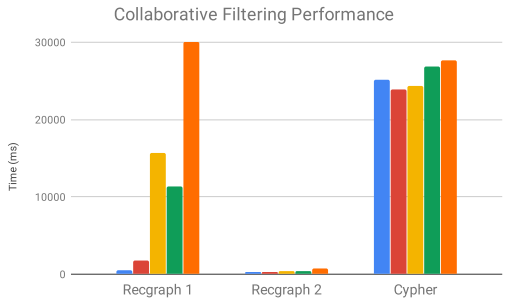

Figure 16: Running times for CF

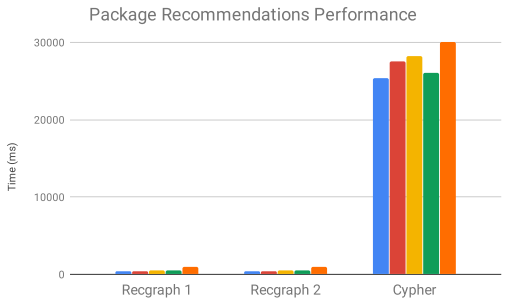

Figure 17: Running times for Package

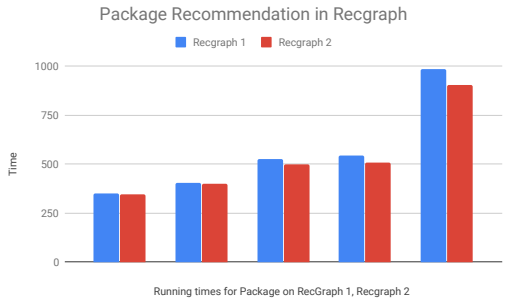

Figure 18: Running times for Package on RecGraph 1, Recraph 2

Package Recommendation Case: Following the generic description of a package recommendation problem, we want to recommend a package, i.e. a set of items, to a user. An item is usually associated with a value and a cost, and the combination of the items determines the value and cost of the package. We consider a simplified version of [7], where items come with an associated value and the overall package value is measured as the average of the items that form the package. In this case, we want to *"recommend a package of restaurants that users, similar to our user, have rated highly in the past"*. A common approach is to measure user preference for each business and then calculate the total score for the package. Here, we use the collaborative filtering algorithm of the previous section to estimate individual preferences. Complexity now arises from two factors: (i) the collaborative filtering bottlenecks that we addressed before, and (ii) re-visiting several parts of the graph as we calculate preferences for businesses. This time *Recgraph 1* works better because its collaborative filtering part is built using the optimized algorithm of Figure 16, but still worse than *Recgraph 2*. We scale Figure 17 and isolate *Recgraph 1* and *Recgraph 2* clusters in Figure 18 to show this. Even though we made sure both algorithms use the same optimized version for collaborative filtering, *Recgraph 2* is more well-behaved due to the plan transformations that avoid the calculations of commonalities in that data, e.g. when computing similarities. The package recommendation use case is very interesting mostly due to the difference in the recommendation goal. Similarly we can capture other 'non-traditional' recommendation tasks, e.g. groups, package-to-group recommendations, etc.

8 CONCLUSIONS AND FUTURE WORK

In this paper, we presented a graph-theoretic approach that introduces recommendations as a graph exploration problem. We proposed a uniform methodology that has (i) a data model, i.e. a graph to model elements of a domain, and (ii) a computational model, i.e. a small set of path operators to express different recommendations. We presented a desired set of operator properties that provide performance opportunities, and the query language to write recommendation algorithms with these operators on top of a graph engine. We provided modeling examples and benchmarked some of them. We are currently working on our own prototype recommendation engine to expand and support even more algorithmic methods in the future.

REFERENCES

[1] 2006. *Netflix Prize.* https://www.netflixprize.com/
[2] 2018. *Yelp Open Dataset.* https://www.yelp.com/dataset
[3] 2019. *Madis.* https://github.com/madgik/madis
[4] 2019. *Neo4j.* https://neo4j.com/docs/
[5] Charu C Aggarwal et al. 2016. *Recommender systems.* Vol. 1. Springer.
[6] Sihem Amer-Yahia, Laks Lakshmanan, and Cong Yu. 2009. Socialscope: Enabling information discovery on social content sites. *arXiv preprint arXiv:0909.2058* (2009).
[7] Idir Benouaret and Dominique Lenne. 2016. A package recommendation framework for trip planning activities. In *Proceedings of the 10th ACM Conference on Recommender Systems.* ACM, 203–206.
[8] Carré Bernard. 1979. *Graphs and Networks.* Clarendon Press, Oxford, UK.
[9] Yang Bo, Lei Yu, Liu Dayou, and L Jiming. 2013. Social collaborative filtering by trust. In *International Joint Conference on Artificial Intelligence AAAI Press.* 2747–2753.
[10] John S Breese, David Heckerman, and Carl Kadie. 2013. Empirical analysis of predictive algorithms for collaborative filtering. *arXiv preprint arXiv:1301.7363*

(2013).
[11] Mariano P Consens and Alberto O Mendelzon. 1989. Expressing structural hypertext queries in GraphLog. In *Proceedings of the second annual ACM conference on Hypertext*. ACM, 269–292.
[12] Maurizio Ferrari Dacrema, Paolo Cremonesi, and Dietmar Jannach. 2019. Are we really making much progress? A worrying analysis of recent neural recommendation approaches. In *Proceedings of the 13th ACM Conference on Recommender Systems*. 101–109.
[13] Anton Dries, Siegfried Nijssen, and Luc De Raedt. 2009. A query language for analyzing networks. In *Proceedings of the 18th ACM conference on Information and knowledge management*. ACM, 485–494.
[14] Francois Fouss, Alain Pirotte, Jean-Michel Renders, and Marco Saerens. 2007. Random-walk computation of similarities between nodes of a graph with application to collaborative recommendation. *IEEE Transactions on knowledge and data engineering* 19, 3 (2007), 355–369.
[15] Simon Funk. 2006. *Try this at home.* https://sifter.org/~simon/journal/20061211.html
[16] Amine Ghrab, Oscar Romero, Sabri Skhiri, Alejandro Vaisman, and Esteban Zimányi. 2016. Grad: On graph database modeling. *arXiv preprint arXiv:1602.00503* (2016).
[17] Guibing Guo, Jie Zhang, and Neil Yorke-Smith. 2015. Trustsvd: Collaborative filtering with both the explicit and implicit influence of user trust and of item ratings. In *Twenty-Ninth AAAI Conference on Artificial Intelligence*.
[18] Ralf Hartmut Güting. 1994. GraphDB: Modeling and querying graphs in databases. In *VLDB*, Vol. 94. 12–15.
[19] Ido Guy, Naama Zwerdling, Inbal Ronen, David Carmel, and Erel Uziel. 2010. Social media recommendation based on people and tags. In *Proceedings of the 33rd international ACM SIGIR conference on Research and development in information retrieval*. ACM, 194–201.
[20] Marc Gyssens, Jan Paredaens, Jan Van den Bussche, and Dirk Van Gucht. 1994. A graph-oriented object database model. *IEEE Transactions on knowledge and Data Engineering* 6, 4 (1994), 572–586.
[21] Yehuda Koren. 2008. Factorization meets the neighborhood: a multifaceted collaborative filtering model. In *Proceedings of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 426–434.
[22] Georgia Koutrika, Benjamin Bercovitz, and Hector Garcia-Molina. 2009. FlexRecs: expressing and combining flexible recommendations. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*. ACM, 745–758.
[23] Mark Levene and Alexandra Poulovassilis. 1991. An object-oriented data model formalised through hypergraphs. *Data & Knowledge Engineering* 6, 3 (1991), 205–224.
[24] Huizhi Liang and Timothy Baldwin. 2015. A Probabilistic Rating Auto-encoder for Personalized Recommender Systems. In *Proceedings of the 24th ACM International Conference on Information and Knowledge Management, CIKM 2015, Melbourne, VIC, Australia, October 19 - 23, 2015*. 1863–1866. https://doi.org/10.1145/2806416.2806633
[25] Greg Linden, Brent Smith, and Jeremy York. 2003. Amazon. com recommendations: Item-to-item collaborative filtering. *IEEE Internet computing* 7, 1 (2003), 76–80.
[26] Hao Ma, Dengyong Zhou, Chao Liu, Michael R Lyu, and Irwin King. 2011. Recommender systems with social regularization. In *Proceedings of the fourth ACM international conference on Web search and data mining*. 287–296.
[27] József Marton, Gábor Szárnyas, and Dániel Varró. 2017. Formalising openCypher graph queries in relational algebra. In *Advances in Databases and Information Systems*. Springer, 182–196.
[28] Shameem A Puthiya Parambath, Nishant Vijayakumar, and Sanjay Chawla. 2017. SAGA: A Submodular Greedy Algorithm for Group Recommendation. *arXiv preprint arXiv:1712.09123* (2017).
[29] Simon Philip, P Shola, and A Ovye. 2014. Application of content-based approach in research paper recommendation system for a digital library. *International Journal of Advanced Computer Science and Applications* 5, 10 (2014).
[30] Paul Resnick, Neophytos Iacovou, Mitesh Suchak, Peter Bergstrom, and John Riedl. 1994. GroupLens: an open architecture for collaborative filtering of netnews. In *Proceedings of the 1994 ACM conference on Computer supported cooperative work*. 175–186.
[31] Francesco Ricci, Lior Rokach, and Bracha Shapira. 2011. Introduction to recommender systems handbook. In *Recommender systems handbook*. Springer, 1–35.
[32] Marko A Rodriguez and Peter Neubauer. 2011. A path algebra for multi-relational graphs. In *Data Engineering Workshops (ICDEW), 2011 IEEE 27th International Conference on*. IEEE, 128–131.
[33] Mauro San Martın, Claudio Gutierrez, and Peter T Wood. 2011. SNQL: A social networks query and transformation language. *cities* 5 (2011), r5.
[34] Badrul Sarwar, George Karypis, Joseph Konstan, and John Riedl. 2001. Item-based collaborative filtering recommendation algorithms. In *Proceedings of the 10th international conference on World Wide Web*. 285–295.
[35] Chuan Shi, Zhiqiang Zhang, Ping Luo, Philip S Yu, Yading Yue, and Bin Wu. 2015. Semantic path based personalized recommendation on weighted heterogeneous information networks. In *Proceedings of the 24th ACM International on Conference on Information and Knowledge Management*. ACM, 453–462.
[36] Yufei Wen, Lei Guo, Zhumin Chen, and Jun Ma. 2018. Network embedding based recommendation method in social networks. In *Companion Proceedings of the The Web Conference 2018*. 11–12.