RecDB in Action: Recommendation Made Easy in Relational Databases *

Mohamed Sarwat
University of Minnesota, Twin Cities
Minneapolis, MN 55455
sarwat@cs.umn.edu

James Avery
University of Minnesota, Twin Cities
Minneapolis, MN 55455
javery@cs.umn.edu

Mohamed F. Mokbel
University of Minnesota, Twin Cities
Minneapolis, MN 55455
mokbel@cs.umn.edu

ABSTRACT

In this paper, we demonstrate *RecDB*; a full-fledged database system that provides personalized recommendation to users. We implemented *RecDB* using an existing open source database system *PostgreSQL*, and we demonstrate the effectiveness of *RecDB* using two existing recommendation applications (1) Restaurant Recommendation, (2) Movie Recommendation. To make the demo even more interactive, we showcase a novel application that recommends research papers presented at VLDB 2013 to the conference attendees based on their publication history in DBLP.

1. INTRODUCTION

Recommender systems have grabbed significant attention in both commercial [3, 4, 7] and academic [1, 2, 5, 6, 9] settings. The main objective of a recommender system is to suggest new interesting items to users from a large pool of items. Recommender systems are implicitly employed on a daily basis to recommend movies (e.g., NetFlix), friends (e.g., Facebook), news articles (e.g., Google News) [4], books/products (e.g., Amazon) [7], and Microblog posts (e.g., Twitter). For instance, Netflix reported that 75% of Movies users watch on Netflix is from recommendation.

Recommendation techniques exploit the history of events performed by the system users in order to extract a set of interesting items for each user. These events might be users clicks (i.e., website links visited), users opinions (e.g., movie ratings), or users purchases (e.g., buying a product on Amazon). In technical terms, a recommender system takes as input a set of users U, items I, and user/item events R and estimates a utility function $\mathcal{F}(u, i)$ that predicts how much a certain user $u \in U$ will like an item $i \in I$ such that i has not been already seen (or watched, consumed...) by u [2].

Currently, to support the recommendation functionality in any application, a developer must implement the recommendation utility estimation as well as the recommendation query execution algorithms at the application layer. That is considered a hassle for a novice application developer who might not be quite familiar with efficient recommender system implementations. An application developer would prefer to declaratively create and query recommenders and save the effort to focus on the main application logic. Moreover, optimizing recommendation queries might be tedious especially that most applications integrate (e.g., `JOIN`) generated recommendation with other data to enrich the end-user experience.

In this paper, we demonstrate *RecDB* a full-fledged database system that produces personalized recommendations to the system users. Integrated with an open source relational DBMS (i.e., PostgreSQL), *RecDB* uses SQL to seamlessly integrate the recommendation functionality with traditional relational operators, i.e., `SELECT`, `PROJECT`, `JOIN`. To this end, *RecDB* introduces a new SQL statement, `CREATE RECOMMENDER`, that takes the input user/item events data, internally runs the recommendation algorithm, and creates the data structures necessary to generate recommendations. *RecDB* therefore employs a new SQL operator, `RECOMMEND`, that leverages the initialized data structures to generate recommendations to the querying user. We summarize the main features of *RecDB*, as follows.

- **Usability**: The system is easily used and configured so that a novice application developer can define a variety of recommenders that fits the application needs in few lines of code. *RecDB* helps the community building an *out-of-the-box* tool to implement a myriad of recommendation applications.
- **Flexibility**: *RecDB* is flexible in terms of defining a recommender using a wide range of popular recommendation algorithms (e.g., item-based/user-based collaborative filtering, singular value decomposition), presented in the literature and implemented inside *RecDB*.
- **Seamless Integration**: The system is able to seamlessly integrate the recommendation functionality in the traditional `SPJ`, i.e., `SELECT`, `PROJECT`, `JOIN`, query pipeline to execute rich recommendation queries.
- **Efficiency**: *RecDB* provides near real-time personalized recommendation to a high number of users over a large pool of items, and enormous user/item events matrix.

To prove *RecDB* effectiveness, we demonstrate the system using two (existing) real life applications: (1) *MovieLens* [8]: a system developed at University of Minnesota that delivers movie recommendation to $\approx$ 72K end-users world wide and (2) *Sindbad* [10]: A location-aware social networking system developed at University of Minnesota and provides a restaurant recommendation service [11] to its users. We replace the underlying recommender system, already-deployed for both applications, with *RecDB* to show the effectiveness of our system and its applicability to real life recommendation scenarios. Moreover, we build a new application that leverages the publication history (i.e., retrieved from DBLP) of VLDB 2013 conference attendees and recommends papers to them

*This work is supported in part by the National Science Foundation under Grants IIS-0952977 and IIS-1218168.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Articles from this volume were invited to present their results at The 39th International Conference on Very Large Data Bases, August 26th - 30th 2013, Riva del Garda, Trento, Italy.
Proceedings of the VLDB Endowment, Vol. 6, No. 12
Copyright 2013 VLDB Endowment 2150-8097/13/10... $ 10.00.

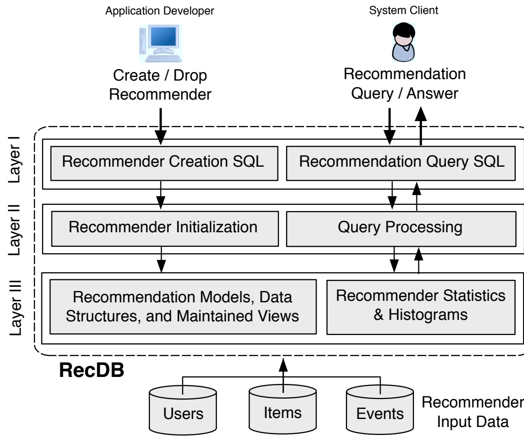

Figure 1: RecDB System Overview.

accordingly. For the paper recommendation application, we furthermore allow the demo attendee to issue ad-hoc recommendation queries using *psql*, i.e., PostgreSQL client, to show the simplicity of integrating recommendation functionality with other relational operators in PostgreSQL.

2. SYSTEM OVERVIEW

Figure 1 highlights the layered architecture of *RecDB*.

Input Data. *RecDB* assumes the following data, as input: (1) *Users*: a set of users $\mathcal{U} = \{u_1, ..., u_n\}$. (2) *Items*: a set of items $\mathcal{I} = \{i_1, ..., i_m\}$. (3) *Events*: Each user u_j performs actions or expresses opinions about a set of items $\mathcal{I}_{u_j} \subseteq \mathcal{I}$. Events can be a numeric rating (e.g., the Netflix scale of one to five stars), or unary (e.g., Facebook "likes", Foursquare "check-ins", or Amazon purchases). *RecDB* consists of three layers, as follows:

Layer I: SQL Layer: This layer supports two new SQL clauses: (1) *Recommender Creation SQL*, and (2) *Recommender Query SQL*. These new SQL clauses are leveraged by the application developer and the clients in declaring and querying personalized recommenders. The parsed SQL statements are then passed to the relevant component in the *processing layer*.

Layer II: Processing Layer: As given in Figure 1, this layer consists of two main components, namely (1) *Recommender Initialization*: This component creates the necessary recommendation models, data structures, and views for a created recommender. (2) *Query processing*: This component efficiently executes recommendation queries over a created recommender and returns the recommendation answer back to the user.

Layer III: Indexing and Storage Layer: This layer stores a set of data structures and recommendation models necessary to produce recommendations. For efficient query execution, *RecDB* also stores a set of views that contains the final recommendation scores generated using the recommendation model. The system also saves statistics about created recommenders and query/update workloads that are harnessed by the *query processing* component.

2.1 Recommender Creation

The application developer creates a new recommender using the `CREATE RECOMMENDER` SQL statement, as follows:

```
CREATE RECOMMENDER [Recommender Name]
USERS FROM [Users Table] KEY Users.uid
ITEMS FROM [Items Table] KEY Items.iid
EVENTS FROM [Events Table] KEY Events.eid
USING [Recommendation Algorithm]
```

The application developer specifies the following parameters: (1) `Recommender name`: A unique name assigned to the newly declared Recommender. (2) `Users Table`, `Items Table`, and `Events Table`: names of SQL tables that contains the users, items, and user/item events information. The users, items, and events data tables are specified in the `USERS FROM`, `ITEMS FROM`, and `EVENTS FROM` SQL clauses. (4) `Recommendation Algorithm`: the application developer may choose to build the recommender using several recommendation algorithms supported by *RecDB* (e.g., Item-Item collaborative filtering, User-User collaborative filtering, singular value decomposition), by specifying the recommendation algorithm in the `USING` clause.

2.2 Recommendation Algorithms

Most recommendation algorithms perform two main steps:

Step I: Model Building: That step is performed by the recommender initialization component when the application developer issues a `CREATE RECOMMENDER` statement to *RecDB*. That step consists of building a recommendation model *RecModel* using the recommender input data. For instance, for the Item-Item Collaborative Filtering algorithm, we generate an items similarity list. To compute the similarity *SimScore*(i_p, i_q), we represent each item as a vector in the user-events space of the user/item events matrix. Many similarity functions have been proposed (e.g., Cosine); the Cosine similarity is calculated as given in equation 1.

$$SimScore(i_p, i_q) = \frac{\vec{i_p} \cdot \vec{i_q}}{\|\vec{i_p}\|\|\vec{i_q}\|} \quad (1)$$

Step II: Recommendation Generation: This step is performed by the query processing component when a user issues a recommendation query to *RecDB*. This step utilizes the *RecModel* (e.g., items similarity list) created in *Step I* to predict a recommendation score, $RecScore(u, i)$ (equation 2), for each user/item pair. $RecScore(u, i)$ reflects how much each user u likes item i.

$$RecScore(u, i) = \frac{\sum_{l \in \mathcal{L}} sim(i, l) * r_{u,l}}{\sum_{l \in \mathcal{L}} |sim(i, l)|} \quad (2)$$

RecDB users may select an algorithm among a variety of recommendation algorithms that fits their application needs. Examples are as follows: (1) Item-Item Collaborative Filtering with Cosine Similarity Function (abbr. ItemCosCF), and its variants (2) User-User Collaborative filtering (abbr. UserCosCF), and its variants (3) Regularized Gradient Descent Singular Value Decomposition (abbr. SVD). (4) Content-based Filtering (abbr. ContentFilter).

2.3 Recommendation Query

Once a recommender is initialized, users can issue recommendation queries over that initialized recommender. A recommender is exposed to the querying user as a virtual SQL table that has a virtual schema, (<u>`uid,iid`</u>,`RecScore`), explained as follows: (1) `uid`: ID of a user who exists in the users table, (2) `iid`: ID of an item in the items table, (3) `RecScore`: a recommendation score (values between 0 and 1) that predicts, i.e., based on the underlying recommendation algorithm, how much the user would like the item.

In *RecDB*, we define a new clause named `RECOMMEND` that is integrated with traditional SQL clauses, e.g., `SELECT`, `FROM`, and `WHERE` clauses, as follows:

```
SELECT [Select Clause]
FROM [Recommender], [Tables]
WHERE [Where Clause]
RECOMMEND(k) User_ID
```

The `RECOMMEND` clause is responsible for generating k recommendations using an initialized recommender. In the `FROM` clause,

Recommender	Recommender Declaration SQL	Recommendation Query SQL

MovieRec
Movie recommender built using the item-item CF (ItemCF) recommendation algorithm
	`CREATE RECOMMENDER MovieRec`
	`USERS FROM Users KEY uid`
	`Items FROM Movies KEY mid`
	`EVENTS FROM Ratings KEY uid,mid`
	`USING ItemCosCF`
		Q$_1$: `SELECT A.mid FROM MovieRec A RECOMMEND(5) A.uid = 1`
		
		Q$_2$: `SELECT E.name FROM MovieRec A, Movies E`
		`WHERE A.mid = E.mid AND E.genre = 'Comedy'`
		`RECOMMEND(5) A.uid = 10`

RestaurantRec
Restaurant recommender built using the singular value decomposition (SVD) recommendation algorithm
	`CREATE RECOMMENDER RestaurantRec`
	`USERS FROM Users KEY uid`
	`Items FROM Restaurants KEY rid`
	`EVENTS FROM CheckIns KEY uid,rid`
	`USING SVD`
		Q$_3$: `SELECT C.name FROM RestaurantRec1 B, Restaurants C`
		`WHERE B.rid = C.rid AND C.location = 'New York City'`
		`RECOMMEND(10) B.uid = 1`
		
		Q$_4$: `SELECT C.name FROM RestaurantRec1 B, Restaurants C`
		`WHERE B.rid = C.rid AND C.location = 'Riva Del Garda'`
		`RECOMMEND(10) B.uid = 10`

PapersRec
VLDB 2013 paper recommender built with content-based filtering recommendation algorithm
	`CREATE RECOMMENDER PapersRec`
	`USERS FROM Authors KEY aid`
	`Items FROM Papers KEY pid`
	`EVENTS FROM Citations KEY aid,pid`
	`USING ContentFilter`
		Q$_5$: `SELECT F.title FROM PapersRec D, Papers F`
		`WHERE F.pid = D.pid AND F.venue='VLDB2013'`
		`RECOMMEND(10) D.aid = 100`
		
		Q$_6$: `SELECT F.title, G.session, G.time`
		`FROM PapersRec D, Papers F, VLDB2013Program G`
		`WHERE D.pid = F.pid AND G.pid = F.pid AND`
		`F.venue='VLDB2013' AND G.Day = 2`
		`RECOMMEND(10) D.aid = 100`

Table 1: *RecDB* Applications SQL.

Figure 2: MovieLens: Movie recommendation website.

the user specifies a recommender [Recommender] that is harnessed by the system to produce k recommended items for user USER_ID. To execute a recommendation query, *RecDB* invokes an operator, named `Recommend`, that is responsible for evaluating the user/item recommendation scores for all items unseen by the querying user. When a user asks for recommendation, the `Recommend` operator calculates the recommendation score $RecScore$, based on the selected recommendation algorithm (see Equation 2), for all candidates items and selects the top-k items with the highest $RecScore$ value and returns them to the user. `Recommend` is integrated with other relational operators (e.g., `Select`, `Project`, `Join`) in the query pipeline.

3. DEMONSTRATION SCENARIOS

In this section, we present two existing real life applications for which we employ *RecDB* as the underlying system for demonstration purpose: (1) Movilens: Movie Recommendation Application, and (2) Sindbad: Restaurant Recommendation Application. Furthermore, we developed an application that recommends papers presented in VLDB 2013 to the conference attendees based on their publication history. Table 1 shows how to create and query recommenders, in *RecDB*, for the three aforementioned applications.

3.1 MovieLens: Movie Recommendation

Figure 3 depicts a screenshot from MovieLens—movie recommendation application. The data set leveraged by this application consists of three tables: (1) `Users (uid, name)`: the set of users that contains information about all users registered with MyRest. Each user tuple consists of a user ID and name. (2) `Movies (mid, title, genre)`: the set of movies saved in the database; each movie has a unique ID and name. (3) `Ratings (uid, mid, rating, timestamp)`: The history of ratings such that each rating represents how much a user liked a movie she/he watched.

The first row in Table 1 gives the details of the `CREATE RECOMMENDER` SQL statement used to declare `MovieRec`, a recommender that is created on top of the Users, Movies, and Ratings database tables. We specify the item-item collaborative filtering method to be applied to the declared recommender. Query Q$_1$ retrieves five movie recommendations using `MovieRec`. `MovieRec` is placed in the `FROM` statement of the issued query. The user, for whom the recommendation needs to be generated (`A.uid = 1`), is passed in `RECOMMEND(5)` clause. Q$_2$ recommends five *Comedy* movies to user (`A.uid = 10`) and returns the title (`title`) of each movie.

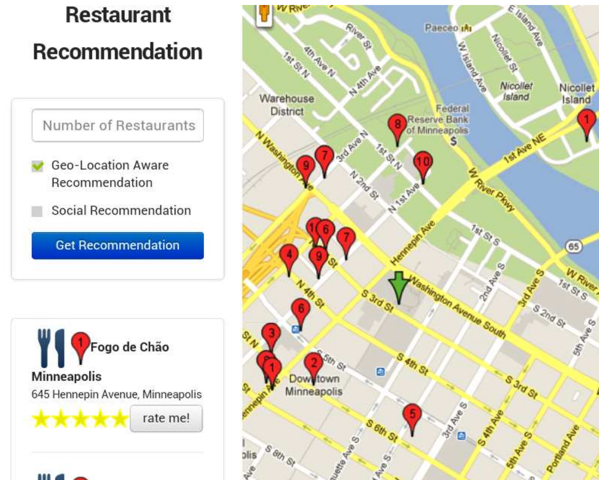

Figure 3: Sindbad: Restaurant recommendation website.

3.2 Sindbad: Restaurant Recommendation

Figure 3 shows a screenshot of *Sindbad* restaurant recommendation service. The data set leveraged by this application consists of three tables: (1) `Users (uid)`: that con-

39th International Conference on Very Large Data Bases 2013
Riva del Garda, Trento, August 26th - 30th

Please enter the following information:

User DBLP Name

Number of Papers

Paper Track: Research Track

VLDB 2013 Day: Day 1: August, 26th 2013

Presented later this day.

Find Papers

VLDB 2013 Papers Recommendation

#	Paper tile	Paper authors	Session	Day	Time	Location
1	DisC Diversity: Result Diversification based on Dissimilarity and Coverage	Marina Drosou and Evaggelia Pitoura	TBA	TBA	TBA	TBA
2	On Differentially Private Frequent Itemset Mining	Chen Zeng, Jeffrey F. Naughton, and Jin-Yi Cai	TBA	TBA	TBA	TBA
3	Lightweight Locking for Main Memory Database Systems	Kun Ren, Alexander Thomson, and Daniel J. Abadi	TBA	TBA	TBA	TBA
4	ClouDiA: A Deployment Advisor for Public Clouds	Tao Zou, Ronan Le Bras, Marcos Vaz Salles, Alan Demers, and Johannes Gehrke	TBA	TBA	TBA	TBA
5	An In-depth Comparison of Subgraph Isomorphism Algorithms in Graph Databases	Jinsoo Lee, Wook-Shin Han, Romans Kasperovics, and Jeong-Hoon Lee	TBA	TBA	TBA	TBA

Figure 4: VLDB 2013 Papers Recommendation Application.

tains IDs of all registered users, (2) `Restaurants (rid, name, location)`: the set of restaurants saved in the database such that each restaurant has a name and a spatial location (i.e., city), and (3) `CheckIns (uid, rid, visited, timestamp)`: The history of check-ins that represents whether a user has visited a restaurants before. In such case, the `visited` field is set one if the user visited the restaurant, and zero otherwise.

The application generates restaurant recommendation to users based upon their spatial locations. We create `RestaurantRec`; a recommender that builds a singular value decomposition (SVD) recommendation model using `CheckIns` table as the user/item events matrix. A user visiting `'New York City'` asks for restaurant recommendation by issuing query Q_3. For Q_3, the user states the current user location using traditional SQL operators (`WHERE B.iid = C.iid AND C.location = 'New York City'`). *RecDB* therefore produces a set of ten restaurants by passing the user ID (`B.uid = 1`) to the `RECOMMEND(10)` clause. Similarly, Q_4 recommends ten restaurants in `'Riva Del Garda'` to the user (`uid = 10`).

3.3 VLDB 2013 Papers Recommendation

Figure 3 exhibits a screenshot of the paper recommendation application. We leverage the DBLP citation database to build an application that recommends papers to VLDB 2013 attendees such that the recommended papers are presented in the conference. The database schema is as follows: (1) `Authors (aid, dblp_name)`: a table that contains a set of 500 authors that publish papers in database venues (i.e., VLDB, SIGMOD, ICDE, EDBT). Each user tuple consists of an author identifier (`aid`), and the author name as it appears in DBLP. (2) `Papers (pid, title, abstract, venue)`: the set of papers published by any author in the `Authors` table in database venues (including VLDB 2013). Each tuple contains a paper identifier (`pid`), title of the paper (`title`), the abstract content, and the venue in which the paper is published (`venue`). (3) `Citations (aid, pid, cited, timestamp)`: The history of citations such that each citation represents whether an author has cited a paper. `cited` is a boolean field; it is set to one if the author `aid` has cited paper `pid`, and zero otherwise. (4) `VLDB2013Program (pid, aid, session,Day,time,location)`: that contains the VLDB 2013 conference schedule. Each entry represents a paper along with the session name, the day/time in which the paper is presented, as well as the presentation location (e.g., hall name).

The last row in Table 1 gives the SQL used for building the paper recommendation application. We create a content-filtering recommender (`ContentFilter` [2]) that leverages the papers abstracts content and the `Citations` table to recommend users new papers (in VLDB 2013) that are similar (in content) to other papers they cited before. Using this application, the demo attendee may ask for papers recommendation by issuing queries similar to Q_5 and Q_6 in table 1. For instance, Q_5 recommends VLDB 2013 papers that correspond to the top-k papers for which the content is similar to the papers cited by the querying user before. Q_6 performs the same functionality with the extra feature of recommending only papers that are scheduled to be presented in the second day of the conference. The idea is to get real time paper recommendation for the conference attendees. We also allow the user to choose a specific conference day to get paper recommendation accordingly. For more interactivity, we allow the demo attendee to issue ad-hoc queries using *psql*.

4. REFERENCES

[1] Z. Abbassi and L. V. S. Lakshmanan. On Efficient Recommendations for Online Exchange Markets. In *ICDE*, 2009.

[2] G. Adomavicius and A. Tuzhilin. Toward the Next Generation of Recommender Systems: A Survey of the State-of-the-Art and Possible Extensions. *TKDE*, 17(6), 2005.

[3] S. Amer-Yahia, A. Galland, J. Stoyanovich, and C. Yu. From del.icio.us to x.qui.site: recommendations in social tagging sites. In *SIGMOD*, 2008.

[4] A. Das et al. Google News Personalization: Scalable Online Collaborative Filtering. In *WWW*, 2007.

[5] B. Kanagal, A. Ahmed, S. Pandey, V. Josifovski, J. Yuan, and L. G. Pueyo. Supercharging Recommender Systems using Taxonomies for Learning User Purchase Behavior. *PVLDB*, 5(10):956–967, 2012.

[6] G. Koutrika, B. Bercovitz, and H. Garcia-Molina. FlexRecs: Expressing and Combining Flexible Recommendations. In *SIGMOD*, 2009.

[7] G. Linden, B. Smith, and J. York. Amazon.com Recommendations: Item-to-Item Collaborative Filtering. *IEEE Internet Computing*, 7(1), 2003.

[8] MovieLens: http://www.movielens.org/.

[9] S. B. Roy, S. Amer-Yahia, A. Chawla, G. Das, and C. Yu. Space efficiency in group recommendation. *VLDB Journal*, 19(6), 2010.

[10] M. Sarwat, J. Bao, A. Eldawy, J. J. Levandoski, A. Magdy, and M. F. Mokbel. Sindbad: A Location-based Social Networking System. In *SIGMOD*, 2012.

[11] M. Sarwat, J. J. Levandoski, A. Eldawy, and M. F. Mokbel. LARS*: A Scalable and Efficient Location-Aware Recommender System. In *TKDE*, 2013.