# RecDB: Towards DBMS Support for Online Recommender Systems

Mohamed Sarwat
Supervised by: Mohamed F. Mokbel
Dept. of Computer Science and Engineering, University of Minnesota, Minneapolis, MN 55455
{sarwat,mokbel}@cs.umn.edu

## ABSTRACT

Recommender systems have become popular in both commercial and academic settings. The main purpose of recommender systems is to suggest to users useful and interesting items or content (data) from a considerably large set of items. Traditional recommender systems do not take into account system issues (i.e., scalability and query efficiency). In an age of staggering web use growth and ever-popular social media applications (e.g., Facebook, Google Reader), users are expressing their opinions over a diverse set of data (e.g., news stories, Facebook posts, retail purchases) faster than ever. In this paper, we propose *RecDB*; a fully fledged database system that provides online recommendation to users. We implement *RecDB* using existing open source database system *Apache Derby*, and we use showcase the effectiveness of *RecDB* by adopting inside Sindbad; a Location-Based Social Networking system developed at University of Minnesota.

## Categories and Subject Descriptors

H.2.8 [**Database Applications**]: Spatial databases and GIS; Data mining; Statistical databases

## General Terms

Design, Experimentation, Human Factors, Performance

## Keywords

Social Networking, Recommender Systems, Query Processing, Model Maintenance, Filtered Recommendation

## 1. INTRODUCTION

Recommender systems have become popular in both commercial [6, 20] and academic settings [1, 5, 21]. The main purpose of recommender systems is to suggest to users useful and interesting items or content (data) from a considerably large set of items. For instance, recommender systems have successfully been leveraged to help users find interesting books and media from a massive inventory base (Amazon [20]), news items from the Internet (Google News [6]), and movies from a large catalog (Netflix, Movielens [21]). The technique used by many of these systems is collaborative filtering (CF) [24], which analyzes past community opinions to find correlations of similar users and items to suggest $k$ personalized items (e.g., movies) to a querying user $u$. Community opinions are expressed through explicit ratings represented by the triple (*user, rating, item*) that represents a *user* providing a numeric *rating* for an *item*.

Traditional recommender systems do not take into account system issues (i.e., scalability and query efficiency) for two main reasons: (1) Due to the nature of the recommended items (e.g., books, movies, cloths), traditional systems were built with the implicit assumption that the recommendation model changes slowly, which tolerates using an *offline* process that builds a fresh model daily or weekly in order to adapt to changes in the underlying content [14, 22, 26]. (2) The recommender system community put more focus on recommendation result quality in order to increase user satisfaction.

Such traditional practices are no longer valid in an increasingly dynamic online world. In an age of staggering web use growth and ever-popular social media applications (e.g., Facebook [9], Google Reader [13]), users are expressing their opinions over a diverse set of data (e.g., news stories, Facebook posts, retail purchases) faster than ever. In such an environment, the system must adapt quickly to its diverse and ever-changing content. Recommender systems cannot wait weeks, days, or even hours to rebuild their models [6]. The rate that new items or users enter the system (e.g., Facebook updates, news posts), and the rate at which users express opinions over items (e.g., Diggs [7], Facebook “likes" [10]), requires recommender models to change in minutes or seconds, implying models be updated *online* (i.e., in real time).

In this paper, we propose *RecDB*; an efficient and scalable system that provides online recommendation to users. Unlike existing implementation approaches, *RecDB* pushes the recommender system functionality inside the database engine in order to provide online recommendations for users. In the rest of the paper, we show the different components of *RecDB* and how it modifies different layers in the database stack.

## 2. RECOMMENDER SYSTEMS

The basic functionality of a *Recommender System* is to take a set of ratings (i.e., triplet user, item, rating) as input, build a recommendation model, and then use the generated model to retrieve a set of recommended items for each user [15, 25]. Nowadays, the high rate that new items or users enter the system, and the high rate that users express opinions over items requires the set of recommended items for each user to change in minutes or even seconds. The process consists of two phases (Figure 1): (1) Model Building Phase: in which the recommendation model is generated


*SIGMOD '12 PhD Symposium*, May 20, 2012, Scottsdale, AZ, USA.

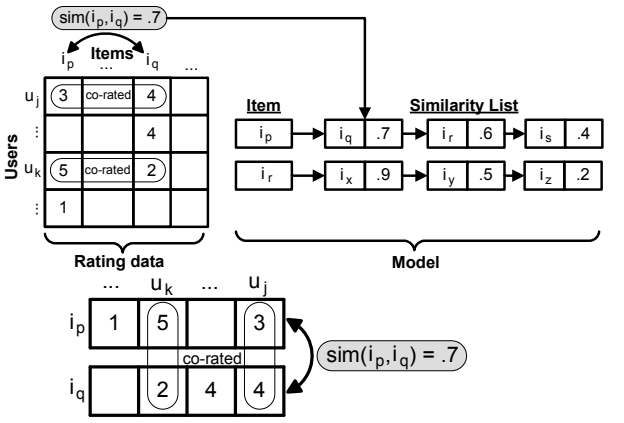



**Figure 1: Item-based CF model generation and item/item similarity calculation.**

using the user/item ratings data, and (2) Recommendation Generation Phase: in which the recommendation model is used to create a set of recommended items for each user. Conceptually, ratings are represented as a matrix with users and items as dimensions, as depicted in Figure 1. Given a querying user $u$, CF produces a set of $k$ recommended items $\mathcal{I}_r \subset \mathcal{I}$ that $u$ is predicted to like the most.

**Phase I: Model Building**. This phase computes a similarity score *sim*($i_p$,$i_q$) for each pair of objects $i_p$ and $i_q$ that have at least one common rating by the same user (i.e., co-rated dimensions). Similarity computation is covered below. Using these scores, a model is built that stores for each item $i \in \mathcal{I}$, a list $\mathcal{L}$ of similar items ordered by a similarity score *sim*($i_p$,$i_q$), as depicted in Figure 1). Building this model is an $O(\frac{R^2}{U})$ process, where $R$ and $U$ are the number of ratings and users, respectively. It is common to truncate the model by storing, for each list $\mathcal{L}$, only the $n$ most similar items with the highest similarity scores [26]. The value of $n$ is referred to as the *model size* and is usually much less than $|\mathcal{I}|$.

**Phase II: Recommendation Generation**. Given a querying user $u$, recommendations are produced by computing $u$'s predicted rating $P_{(u,i)}$ for each item $i$ not rated by $u$ [26]:

$$P_{(u,i)} = \frac{\sum_{l \in \mathcal{L}} sim(i,l) * r_{u,l}}{\sum_{l \in \mathcal{L}} |sim(i,l)|} \quad (1)$$

Before this computation, we reduce each similarity list $\mathcal{L}$ to contain only items *rated* by user $u$. The prediction is the sum of $r_{u,l}$, the user's rating for a related item $l \in \mathcal{L}$ weighted by *sim(i,l)*, the similarity of $l$ to candidate item $i$, then normalized by the sum of similarity scores between $i$ and $l$. The user receives as recommendations the top-$k$ items ranked by $P_{(u,i)}$.

**Computing Similarity**. To compute *sim*($i_p$, $i_q$), we represent each item as a vector in the user-rating space of the rating matrix. For instance, Figure 1 depicts vectors for items $i_p$ and $i_q$ from the matrix in Figure 1. Many similarity functions have been proposed (e.g., Pearson Correlation, Cosine); we use the Cosine similarity in *LARS* due to its popularity:

$$sim(i_p, i_q) = \frac{\vec{i_p} \cdot \vec{i_q}}{\|\vec{i_p}\| \|\vec{i_q}\|} \quad (2)$$

This score is calculated using the vectors' co-rated dimensions, e.g., the Cosine similarity between $i_p$ and $i_q$ in Figure 1 is .7 calculated using the circled co-rated dimensions. Cosine distance is useful for numeric ratings (e.g., on a scale [1,5]). For unary ratings, other similarity functions are used (e.g., absolute sum [4]).

# 3. RECOMMENDER SYSTEM IMPLEMENTATION APPROACHES

Recommender system functionality has been always taken care of in the application layer as depicted in Figure 2. In other words, the application developer implements all the logic behind the recommendation functionality, and uses the DBMS only for storage; we call that the *traditional approach*. The good news about the traditional approach is that it gives high freedom to the application developer to write its own recommendation techniques that fits the end-user of the application Nonetheless, the *traditional approach* suffers from the following drawbacks: (1) Implementation Complexity: As the application developer is responsible for the whole recommender system logic, the application development process might end up being tedious. (2) Lack of System Expertise: All the application developer cares about is the application functionality, hence s/he might not be able to handle the system performance and scalability issues.

On the other hand, the *Built-in* approach pushes both steps (i.e., model building and recommendation generation) of the recommender system inside the DBMS. Hence, the application developer just focuses on the application logic and relies on the DBMS to take care of the system performance and scalability issues. However, the *Built-in* approach is sort of rigid as it mandates the usage of specific recommendation techniques that are implemented a-priori inside the DBMS. In case the application developer wants to employ a different recommendation technique, s/he might either implement the new recommendation technique inside the DBMS or alternatively use the *traditional* approach.

The *Extensible* approach is similar to the *Built-in* approach, with the exception that the DBMS is extensible to new recommendation techniques, which could be declared by the application developer. The *Extensible* approach combines the advantages of both the *traditional* approach and the *Built-in* approach in such a way that it isolates the application developer from the system issues and at the same time allow her/him to define new recommendation techniques. For the aforementioned reasons, we set the *Extensible* approach as our system design goal when building *RecDB*.

# 4. RECDB OVERVIEW

We propose *RecDB*; a system that implements the recommender system functionality inside the DBMS engine. RecDB is built on three main system design pillars:

- Low Latency: That is necessary in order to feed recommendations to users in an online/real time manner.
- High Extensibility: That means that the system is extensible to as many recommendation techniques as possible.
- High Flexibility: That means the system provides flexible recommendation based-upon application requirements.

To this end, RecDB adopts an *extensible* approach (see Figure 4), as described in section 3. RecDB pushes both recommender system phases inside the core engine of an DBMS To achieve that, RecDB needs to modify almost all layers of the database stack; ranging from the query parser to the storage engine. In the rest of the paper, we will highlight the three main *RecDB* components, namely: *(1) RecStore:* It is a module built inside the database storage engine that incrementally maintains the recommendation model aiming at increasing the system efficiency. *(2) Rec-tree:* it is an efficient tree structure that provides flexible recommendation based upon the indexed users/items attributes. *(3) RecQuery:* It is a module built inside the database query processor, whose role is to minimize the

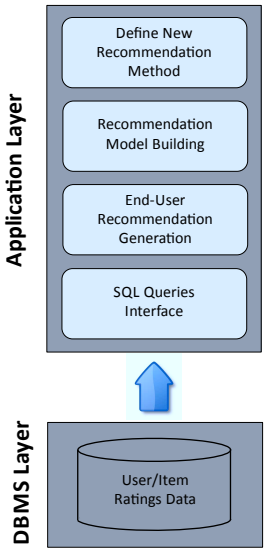



**Figure 2: Traditional Approach**

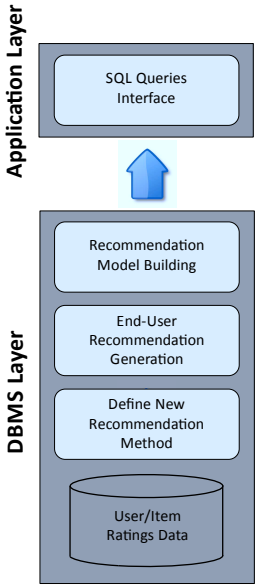



**Figure 3: Built-In Approach**

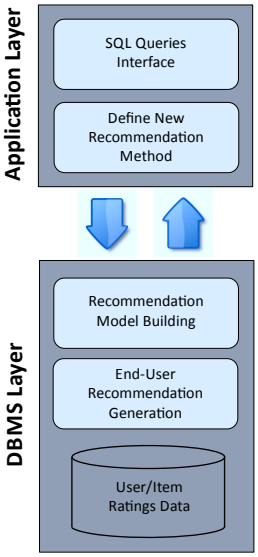



**Figure 4: Extensible Approach**

recommendation query latency, in order to recommend a high number of items to a huge number of system users in an online manner. In the rest of this section, we will give a little bit of details of the aforementioned modules.

## 4.1 RecStore: Incremental Model Maintenance

To be effective, recommender systems must evolve with their content. For example, new users enter the system changing the collective opinions of items, the system adds new items widening the recommendation pool, or user tastes change. These actions affect the recommender model, that in turn affect the system's recommendation quality. Traditionally, most systems have been able to tolerate using an *offline* process that builds a fresh model daily or weekly in order to adapt to changes in the underlying content [14, 22, 26]. However, the rate that new items or users enter the system in nowadays application (e.g., Facebook updates, news posts), and the rate which users express opinions over items (e.g., Diggs [7], Facebook "likes" [10]), requires recommender models to change in minutes or seconds, implying models be updated *online*.

Recent work from the data management community has shown that many popular recommendation methods (including collaborative filtering) can be expressed with conventional SQL, effectively pushing the core logic of recommender systems within the DBMS [16]. However, the approach does nothing to address the pressing problem of *online model maintenance*, as collaborative filtering still requires a computationally intense offline model generation phase when implemented with a DBMS.

*RecStore* [19] is a module built to complement the storage engine of an DBMS, which enhances the recommendation model generation step by proposing a method that *incrementally maintains* the recommendation model when new user/item ratings enter the system. The basic idea behind *RecStore* is to separate the logical and internal representations of the recommender model. *RecStore* receives updates to the user/item ratings data (i.e., the base data for a collaborative filtering models) and maintains its internal representation based on these updates. As *RecStore* is built into the DBMS storage engine, it outputs tuples to the query processor through access methods that transform data from the internal representation

```
CREATE REC-TREE INDEX
ON users_Table (UserAge, UserCity)
USERS FROM users_Table KEY userID -- (userID, name, . . . )
ITEMS FROM items_Table KEY itemID -- (itemID, itemDetails)
RATINGS FROM ratings_Table KEY (userID,itemID) --(userID,itemID,rating)
```

**Figure 5: SQL Example 1 to Create Rec-tree Index**

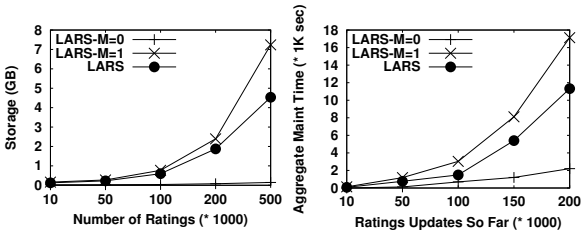
```
CREATE REC-TREE INDEX
ON items_Table (ItemType)
USERS FROM users_Table KEY userID -- (userID, name, . . . )
ITEMS FROM items_Table KEY itemID -- (itemID, itemDetails)
RATINGS FROM ratings_Table KEY (userID,itemID) --(userID,itemID,rating)
```

**Figure 6: SQL Example 2 to Create Rec-tree Index**

into the logical representation expected by the query processor. *RecStore* is designed with extensibility in mind. *RecStore*'s architecture is generic, and thus the logic for a number of different recommendation methods can easily be "plugged into" the *RecStore* framework, making it a one-stop solution to support a number of popular recommender models within the DBMS. *RecStore* is also adaptive to system workloads, tunable to realize a trade-off that makes query processing more efficient at the cost of update overhead, and vice versa.

## 4.2 Rec-tree: An Efficient Index Structure for Processing Online Recommender Queries

Recommender systems need to provide flexible recommendation to the end-user. For instance, Amazon.com has a huge number of users, items (i.e., products), and user/item ratings. Among all Amazon.com items, a user might be interested to get recommended "Books" only. Also, a user might want to get recommended items that are bought only by users living in Minnesota. Moreover, a user might need to get recommended only "Books" that were bought by users who lives in Minnesota and their age is between 21 and 35. The main challenge is that we need to maintain a recommendation model for all attributes ranges defining the index structure. Notice that existing database index structures (e.g., $B^+$-tree and




(a) Storage (b) Maintenance

**Figure 7: Scalability of LARS**

$R$-tree) can be naively employed to solve the problem. However, that approach suffers from scalability issues as it needs to maintain a recommendation model for all values indexed by the tree. Moreover, as we maintain a recommendation model for all values in each level of the tree, the query and update (e.g., insertion and deletion) performance becomes a severe system bottleneck.

To solve the problem, we propose *Rec-tree*, a multi-dimensional tree index structure that is built specifically to index recommendation models and provides flexible and online recommendation to users. As in traditional database index structures, the user can define which users/items attributes (i.e., dimensions) needed to be indexed by *Rec-tree* (see Examples in Figures 5 and 6 ). Rec-tree partitions the user/items ratings space based upon the index attribute and maintain a recommendation model for each partition. Rec-tree is an adaptive structure that decides whether to merge or split a tree node by employing a tradeoff between recommendation quality and system scalability.

As a proof of concept, we implemented LARS [18]; a Location-Aware Recommender System that provides recommendation for users base on their locations as well as the items locations. LARS employs a partial pyramid structure [3] (equivalent to a partial quad-tree [11]). In each pyramid cell, we store an item-based collaborative filtering model built using *only* the spatial ratings with user locations contained in the cell's spatial region. rating may contribute to up to at most a single cell at each pyramid level: starting from the lowest maintained grid cell containing the embedded user location up to the root level. Levels in the pyramid can be incomplete, as LARS will periodically merge or split cells based on trade-offs of locality and scalability. When deciding to merge, we define a system parameter $\mathcal{M}$, a real number in the range [0,1] that defines a tradeoff between scalability gain and locality loss. LARS merges (i.e., discards quadrant $q$) if:

$$(1 - \mathcal{M}) * scalability\_gain > \mathcal{M} * locality\_loss \quad (3)$$

**Initial Experiments.** Figure 7 depicts the storage and aggregate maintenance overhead required for an increasing number of ratings. We again LARS-M=0 and LARS-M=1 to indicate the extreme cases for LARS. Figure 7(a) depicts the impact of increasing the number of ratings from 10K to 500K on storage overhead. LARS-M=0 requires the lowest amount of storage since it only maintains a single collaborative filtering model. LARS-M=1 requires the highest amount of storage since it requires storage of a collaborative filtering model for all cells (in all levels) of a complete pyramid. The storage requirement of LARS is in between the two extremes since it merges cells to save storage.

## 4.3 RecQuery: Low-Latency Recommendation Generation

In traditional recommender systems, when a user logs in, a complex SQL query is issued at the application layer, while the user only cares about the set of recommended items. To remedy this situation, *RecDB* pushes the second step (i.e., *Recommendation Generation*) inside the DBMS, by creating a RECOMMENDATION VIEW as in Figure 8. In Figure 8, the application developer declares a view called *Rec_View* which has two fields *userID* and *itemID* (representing the recommended items for each user). All users are defined in *users_Table* that has *userID* as a primary key. All items are stored in *items_Table* that has *itemID* as a primary key. The ratings are represented by *ratings_Table* which is defined by the triplet *(userID,itemID,rating)*. The *USING* keyword determines the recommendation model (e.g., Item-Based Collaborative filtering) used to generate recommendation for users. Once having the recommendation view, the application developer can issue regular SQL queries on that view to select appropriate recommended items for end-users. *RecQuery* modifies the query processor by making use of the recommendation view; mentioned before. Recall that the main goal is to provide recommendations for users in real time and online manner while the recommendation model is updated frequently. To this end, *RecQuery* has to answer two main research questions: (1) *What to materialize ?* – What items (of which users) should be stored in the recommendation view and which not, and (2) *How to materialize ?* – For materialized items (users), how to incrementally maintain the view.

```
CREATE RECOMMENDATION VIEW
Rec_View KEY (userID,itemID) -- (userID,itemID)
USERS FROM users_Table KEY userID -- (userID, name, . . . )
ITEMS FROM items_Table KEY itemID -- (itemID, itemDetails)
RATINGS FROM ratings_Table KEY (userID,itemID)  -- (userID,itemID,rating)
USING ItemBasedModel
```

**Figure 8: RECOMMENDATION VIEW**

# 5. DATA SETS

In order to evaluate *RecDB* system quality and performance, we make use three main data sets as follows:

- MovieLens data: The MovieLens data used in is a real movie rating data taken from the popular MovieLens recommendation system at the University of Minnesota [23]. This data consists of 10 million ratings for 10,000 movies from 72,000 users. Users' ratings to items takes values between zero and five.

- Foursquare data: Foursquare [12] is a mobile location-based social network application. Users are associated with a home city, and alert friends when visiting a venue (e.g., restaurant) by **"checking-in"** on their mobile phones. During a "check-in", users can also leave **"tips"**, which are free text notes describing what that they liked about the venue. Any other user can add the "tip" to her **"to-do list"** if interested in visiting the venue. Once a user visits a venue in the "to-do list" , she marks it as "done". Also, users who check into a venue the most are considered the **"mayor"** of that venue. We crawled Foursquare and collected data for 1,010,192 users and 642,990 venues across the United States. Foursquare does not publish each "check-in" for a user, however, we were able to collect the following pieces of data: (1) user tips for a venue, (2) the venues for which the user is the mayor, and (3) the completed to-do list items for a user. In addition, we extracted each user's friend list. To extract user/items ratings from foursquare data, we use a numeric *rating* value range of [1, 3], translated as follows: (a) 3 represents the user is the "mayor" of the venue, (b) 2 represents that the user left a "tip" at the venue, and (c) 1 represents the user visited the


**Figure 9: Sindbad Recommendation Service**

venue as a completed "to-do" list item. Using this scheme, a user may have multiple ratings for a venue, in this case we use the highest rating value.

- Synthetic data: The synthetic data set is generated using a synthetic data generator that takes the number of users, number of items, and number of ratings as input and randomly generates a set of user/item ratings. Users' ratings to items are assigned random values between zero and five. Notice that the synthetic data is only used to test the system performance and is never user to test the recommendation quality.

## 6. SYSTEM PROTOTYPE

To implement the recommendation functionality RecDB employs the Lenskit recommender framework built at the University of Minnesota. LensKit [8, 17] is an open source toolkit for building, researching, and studying recommender systems. To support *RecDB* system functionalities, we modify the different layers of *Apache Derby* [2], an open source relational database system built in Java and available under Apache License.

We plan to build a RecDB prototype inside *Sindbad* [18, 27]; a location-based social networking system built at the University of Minnesota. Sindbad users can request recommendations of either spatial items (e.g., restaurants, stores) or non-spatial items (e.g., movies) by explicitly issuing a location-aware recommendation query. The *location-aware recommender* module (LARS) suggests a set of items based on: (a) the user location (if available), (b) the item location (if available), and (c) ratings previously posted by either the user or the user's friends. Figure 9 depicts an Android Phone Application that is built on top of Sindbad. As shown in the figure, End-users may ask Sindbad for recommendations (e.g., Restaurants) by clicking on the recommendation link on the left-hand side of the web interface or by clicking on the location-aware recommendation button in the mobile app. The user then enters the type of object he is interested in (e.g., restaurant, theaters, stores). a spatial range in miles, and also the number of recommended items to be returned to him and then presses the *Recommend* button. The recommended items are then shown on the map. Sindbad is used to demonstrate the effectiveness of *RecDB* in a real application setting; which gives high credibility to the system.

## 7. CONCLUSION

In this paper, we propose *RecDB*; a fully fledged database system that provides recommendation functionality in an efficient and scalable way. To this end, RecDB uses an extensible approach which pushes the recommender systems phases inside the database engine. RecDB has three main component: (1) RecStore that efficiently maintains the recommendation model in order to serve online recommendations for end-users, (2) Rec-tree is an index structure that provides flexible recommendation functionality filtered by user/item attributes, and (3) RecQuery that efficiently maintains a recommendation view in order to serve low-latency recommendation to end-users. RecDB is implemented as part of existing DBMS (i.e., Apache Derby) and is demonstrated by Sindbad; a Location-based Social Networking system.

## 8. REFERENCES


[1] 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.

[2] Apache Derby: http://db.apache.org/derby.

[3] W. G. Aref and H. Samet. Efficient Processing of Window Queries in the Pyramid Data Structure. In *PODS*, 1990.

[4] J. S. Breese, D. Heckerman, and C. Kadie. Epirical Analysis of Predictive Algorithms for Collaborative Filtering. In *UAI*, 1998.

[5] CoFE Recommender System: http://eecs.oregonstate.edu/iis/CoFE.

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

[7] Digg: http://digg.com.

[8] M. D. Ekstrand, M. Ludwig, J. A. Konstan, and J. T. Riedl. Rethinking the Recommender Research Ecosystem: Reproducibility, Openness, and LensKit. In *RecSys*, 2011.

[9] Facebook: http://www.facebook.com.

[10] Facebook turns on its 'Like' button: http://news.cnet.com/8301-1023_3-10160112-93.html.

[11] R. A. Finkel and J. L. Bentley. Quad trees: A data structure for retrieval on composite keys. *Acta Inf.*, 4:1–9, 1974.

[12] Foursquare: http://foursquare.com.

[13] Google Reader: www.google.com/reader.

[14] G. Karypis. Evaluation of Item-Based Top-$N$ Recommendation Algorithms. In *CIKM*, 2001.

[15] J. A. Konstan, B. N. Miller, D. Maltz, J. L. Herlocker, L. R. Gordon, and J. Riedl. GroupLens: Applying Collaborative Filtering to Usenet News. *Commun. ACM*, 40(3), 1997.

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

[17] LensKit: http://lenskit.grouplens.org/.

[18] J. J. Levandoski, M. Sarwat, A. Eldawy, and M. F. Mokbel. LARS: A Location-Aware Social Networking System. In *ICDE*, 2012.

[19] J. J. Levandoski, M. Sarwat, M. F. Mokbel, and M. D. Ekstrand. RecStore: An Extensible and Adaptive Framework for Online Recommender Queries inside the Database Engine. In *EDBT*, 2012.

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

[21] B. N. Miller, I. Alber, S. K. Lam, J. A. Konstan, and J. Riedl. MovieLens Unplugged: Experiences with an Occasionally Connected Recommender System. In *IUI*, 2002.

[22] B. N. Miller, J. A. Konstan, and J. Riedl. PocketLens: Toward a Personal Recommender System. *TOIS*, 22(3), 2004.

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

[24] P. Resnick, N. Iacovou, M. Suchak, P. Bergstrom, and J. Riedl. GroupLens: An Open Architecture for Collaborative Filtering of Netnews. In *CSWC*, 1994.

[25] P. Resnick and H. R. Varian. Recommender Systems. *Commun. ACM*, 40(3), 1997.

[26] B. Sarwar, G. Karypis, J. Konstan, and J. Riedl. Item-Based Collaborative Filtering Recommendation Algorithms. In *WWW*, 2001.

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