RecStore: An Extensible and Adaptive Framework for Online Recommender Queries inside the Database Engine

Justin J. Levandoski[1§] Mohamed Sarwat[2] Mohamed F. Mokbel[2] Michael D. Ekstrand[2]

[1]Microsoft Research, Redmond, WA, justin.levandoski@microsoft.com
[2]University of Minnesota, Minneapolis, MN, {sarwat,mokbel,ekstrand}@cs.umn.edu

ABSTRACT

Most recommendation methods (e.g., collaborative filtering) consist of (1) a computationally intense *offline* phase that computes a recommender model based on users' opinions of items, and (2) an *online* phase consisting of SQL-based queries that use the model (generated offline) to derive user preferences and provide recommendations for interesting items. Current application usage trends require a completely *online* recommender process, meaning the recommender model must update in *real time* as new opinions enter the system. To tackle this problem, we propose *RecStore*, a DBMS storage engine module capable of efficient *online model maintenance*. Externally, models managed by *RecStore* behave as relational tables, thus *existing* SQL-based recommendation queries remain *unchanged* while gaining online model support. *RecStore* maintains internal statistics and data structures aimed at providing efficient incremental updates to the recommender model, while employing an *adaptive* strategy for internal maintenance and load shedding to realize a balance between efficiency in updates or query processing based on system workloads. *RecStore* is also *extensible*, supporting a declarative syntax for defining recommender models. The efficacy of *RecStore* is demonstrated by providing the implementation details of three state-of-the-art collaborative filtering models. We provide an extensive experimental evaluation of a prototype of *RecStore*, built inside the storage engine of PostgreSQL, using a real-life recommender system workload.

1. INTRODUCTION

Recommender systems have grown popular in both commercial [6, 21] and academic settings [1, 5, 23]. The purpose of recommender systems is to help users identify useful, interesting items or content (data) from a considerably large search space. For example, recommender systems have successfully been used to help users find interesting books and media from a massive inventory base (Amazon [21]), news items from the Internet (Google News [6]), and movies from a large catalog (Netflix, Movielens [23]). By far, the most popular recommendation method used is collaborative filtering [18, 30], which consists of two phases: (1) A computationally expensive offline *model generation* phase that uses community opinions (e.g., user ratings) of data items in order to derive meaningful correlations between users and/or items. (2) An online *recommendation generation* phase that uses the model to produce recommendations. From a database perspective, the recommendation process is simply a set of SQL-based *recommender queries* built to provide answers according to a particular recommendation method [19]. Examples of recommendation methods include user-based [29] or item-based [31] collaborative filtering.

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 [17, 24, 31]. However, these 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 [12]), 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, forcing recommender systems to use an *offline* model building phase is *unacceptable*, as 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 that which users express opinions over items (e.g., Diggs [8], 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 [19]. 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.

In this paper, we address the problem of providing online recommender model maintenance for DBMS-based recommender systems. We present *RecStore*, a module built *inside* the storage engine

This work is supported in part by the National Science Foundation under Grants IIS-0811998, IIS-0811935, CNS-0708604, IIS-0952977 and by a Microsoft Research gift

§Work done while at the University of Minnesota

EDBT 2012, March 26–30, 2012, Berlin, Germany.

of a relational database system. *RecStore* enables online model support for DBMS-based recommender systems (e.g., [19]) through efficient incremental updates to *only* parts of the model affected by a rating update. Thus, updating the recommender model does not involve significant overhead, nor regeneration of the model from scratch. *RecStore* exposes the model to the query processor as a standard relational table, meaning that *existing* recommender queries can remain *unchanged*.

The basic idea behind *RecStore* is to separate the logical and internal representations of the recommender model. *RecStore* receives updates to the user/item rating 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 though access methods that transform data from the internal representation 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. We provide a generic definition syntax for *RecStore*, and provide implementation case studies for various memory-based [1, 3] collaborative filtering methods (e.g., item-based [31] and user-based [29]). We also discuss support for other non-trivial recommendation methods (e.g., [3, 20]).

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. At one extreme, *RecStore* has lowest query latency by making update costs more expensive; appropriate for query-intense workloads. At the other extreme, *RecStore* minimizes update costs by pushing computation into query processing; appropriate for update-intense workloads. For particularly update-intense workloads, *RecStore* also performs load-shedding to process *only* important updates that significantly alter the recommender model and change the answers to recommender queries.

RecStore requires a small code footprint, which is advantageous to implementation in existing database engines. Our prototype of *RecStore*, built *inside* PostgreSQL [28], between the storage engine and query processor, requires approximately 600 lines of either modified or new code. Rigorous experimental study of our *RecStore* prototype using a *real* workload from the popular MovieLens [26] recommender system shows that *RecStore* exhibits desirable performance in *both* updates and query processing compared to existing DBMS approaches that support online recommender models using regular and materialized views.

The rest of this paper is organized as follows. Section 2 provides a preliminary background. Related work is covered in Section 7. Section 3 introduces the *RecStore* architecture. Section 4 describes the functionality of *RecStore*. Section 6 provides an experimental evaluation of *RecStore*. Finally, Section 8 concludes the paper.

2. COLLABORATIVE FILTERING AND THE DBMS

This section provides an overview of collaborative filtering, the primary recommendation method we are concerned with in this paper; support for other methods is discussed later in Section 5. Collaborative filtering assumes a set of n users $\mathcal{U} = \{u_1, ..., u_n\}$ and a set of m items $\mathcal{I} = \{i_1, ..., i_m\}$. Each user u_j expresses opinions about a set of items $\mathcal{I}_{u_j} \subseteq \mathcal{I}$. In this paper, we assume opinions are expressed through an explicit numeric rating (e.g., one through five stars), but other methods are possible (e.g., hyperlink clicks, Facebook "likes" [10], Diggs [8]). An active user u_a is given a set of recommendations $\mathcal{I}_r$ such that $\mathcal{I}_{u_a} \cap \mathcal{I}_r = \emptyset$, i.e., the user has not rated the recommended items. The recommendation process is usually broken into two phases: (1) an offline *model generation* phase that creates a model storing correlations between items and/or users, and (2) an online *recommendation generation* phase that uses the model to generate recommended items. There are several methods to perform collaborative filtering including item-based [31], user-based [29], regression-based [31], or approaches that use more sophisticated probabilistic models (e.g., Bayesian Networks [3]).

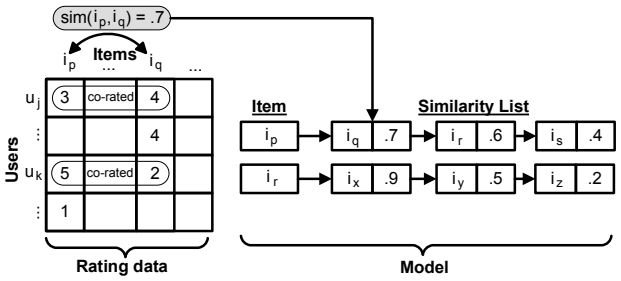


Figure 1: Item-based Model Generation

Below we describe the details of item-based [31] and user-based [29] collaborative filtering, by far two of the most popular recommendation methods in use today (e.g., Amazon [21]). These methods are classified as "memory-based" recommendation approaches [3, 1], so called because they "remember" the opinion history of the entire user base in order to provide recommendations. Details of other methods are discussed in Section 5.

2.1 Offline Model Generation

The offline model generation phase analyzes the entire user/item rating space, and uses statistical techniques to find correlated items and/or users. These correlations are measured by a *score*, or weight, that defines the strength of the relation.

2.1.1 Item-based collaborative filtering

The item-item model builds, for each of the m items $\mathcal{I}$ in the database, a list $\mathcal{L}$ of *similar* items. Given two items i_p and i_q, we can derive their similarity score $sim(i_p, i_q)$ by representing each as a vector in the user-rating space, and then use a similarity function over the two vectors to compute a numeric value representing the strength of their relationship. Figure 1 depicts this item-item model-building process. Conceptually, we can represent the ratings data as a matrix, with users and items each representing a dimension, as depicted on the left side of Figure 1. The similarity function, $sim(i_p, i_q)$, computes the similarity of vectors i_p and i_q using *only* their co-rated dimensions. In our example u_j and u_k represent the co-rated dimensions. Finally, we store i_p, i_q, and $sim(i_p, i_q)$ in our model, as depicted on the right side of Figure 1. The similarity measure need not be symmetric, i.e., it is possible that $sim(i_p, i_q) \neq sim(i_q, i_p)$.

Many similarity measures have been proposed in the literature [17,

31]. On of the most popular measures used is the cosine distance, calculated as:

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

Here, items i_p and i_q are represented as vectors in the user-rating space, and k represents a dampening factor that discounts the influence of item pairs having high scores, but only a *few* common ratings [14]; given the co-rating count between two items as $corate(i_p, i_q)$, k is defined as:

$$k = \begin{cases} 1 & corate(i_p, i_q) \geq 50 \\ corate(i_q, i_q)/50 & otherwise \end{cases} \quad (2)$$

2.1.2 User-based collaborative filtering

The user-user model is similar in nature to the item-item paradigm, except that the model calculates similarity between users (instead of items). This calculation is performed by comparing user vectors in the item-rating space. For example, in Figure 1, focusing on the user/item matrix, users u_j and u_k can be represented as vectors in item space, and compared based on the items they have co-rated (i.e., i_p and i_q). The user-user model primarily uses cosine distance and Pearson correlation as similarity measures [3], much like that of the item-item paradigm with the exception that similarity is measured in item space rather than user space.

2.2 Online recommendation generation

The online recommendation generation phase employs the ability to predict ratings for items that a user u_a has not yet rated. Rating predictions are produced by performing aggregation over the recommender models. These predictions can be used to (1) give the user their predicted score for a specific item on request, or (2) produce a set of top-N recommended items based on highest predicted scores.

2.2.1 Item-based collaborative filtering

Recommendation generation for the item-based cosine method produces the top-n items based on predicted score using two steps. (1) *Reduction*: cut down the model such that each item i left in the model is an item *not* rated by user u_a, while i's similarity list $\mathcal{L}$ contains only items l already rated by u_a. (2) *Compute*: the predicted rating $P_{(u_a,i)}$ for an item i and user u_a is calculated as a weighted sum [31]:

$$P_{(u_a,i)} = \frac{\sum_{l \in \mathcal{L}} sim(i,l) * r_{u_a,l}}{\sum_{l \in \mathcal{L}} sim(i,l)} \quad (3)$$

The prediction is the sum of the user's rating for a related item l, $r_{u_a,l}$, weighted by the similarity to the candidate item i. The prediction is normalized by the sum of scores between i and l.

2.2.2 User-based Collaborative Filtering

Rating prediction in the user-based recommender paradigm is similar in spirit to the item-based method. Recall that the similarity list $\mathcal{L}$ in the user-user paradigm is a list of similar users to a particular user u. A prediction $P_{(u_a,i)}$ for an item i given user u_a is calculated as [18]:

```
/* Find movies rated by REC_USER_X,
 * store in temp table usrXMovies */
CREATE TEMP TABLE usrXMovies AS
SELECT  R.mid as itemId, R.rating as rating
FROM    ratings R
WHERE   R.uid = REC_USER_X;

/* Generate predictions using weighted sum */
SELECT  M.itm as Candidate Item,
        SUM(M.sim * U.rating)/ SUM(M.sim) as Prediction
FROM    Model M, usrXMovies U
WHERE   M.rel_itm = U.itmId AND
        M.itm NOT IN (select itmId FROM usrXMovies)
GROUP BY M.itm ORDER BY Prediction DESC;
```

Figure 2: Item-based recommender query

$$P_{(u_a,i)} = \overline{r}_{u_a} + \frac{\sum_{l \in \mathcal{L}} (r_{u_l,i} - \overline{r}_{u_l}) * sim(u_a, u_l)}{\sum_{l \in \mathcal{L}} |sim(u_a, u_l)|} \quad (4)$$

This value is the weighted average of deviations from a related user u_l's mean. In this equation, $r_{u_l,i}$ represents a user u_l's (non-zero) rating for item i, while $\overline{r}_{u_a}$ and $\overline{r}_{u_l}$ represent the average rating values for users u_a and u_l, respectively.

2.3 DBMS-based Collaborative Filtering

A DBMS can be used to implement the recommendation process just described. Ratings data can be stored in a relation *Ratings(userId,itemId,rating)*, where *userId* and *itemId* represent unique ids of users and items, respectively.

2.3.1 Model representation

The model can be represented by a three-column table *Model(item,rel_itm,score)* for the item-item model, or *Model(user,rel_user,score)* for the user-user model (different schemas may be necessary for other methods).

2.3.2 Recommender queries

A DBMS-based recommenders will use SQL to produce recommendations. Figure 2 provides an SQL example of the process discussed in Section 2.2 (listed in two parts for readability). The first query finds all movies rated by a user X. The second query uses these results to produce recommendations for user X using Equation 3. The WHERE clause represents the *reduction* step, while the SELECT clause represents the *computation* step. The query assumes the *model* relation *M(itm,rel_itm,sim)* is already generated offline.

3. RecStore ARCHITECTURE

Figure 3 depicts the high-level architecture of *RecStore*, built inside the storage engine of a DBMS. *RecStore* consists of the following main components:

- **Intermediate store and filter**. The *intermediate store* contains a set of statistics, functions, and/or data structures that are efficient to update, and can be used to quickly generate part of the recommender model. The data maintained in the intermediate store is specific to the recommendation method. Whenever *RecStore* receives ratings updates (i.e., insertions,

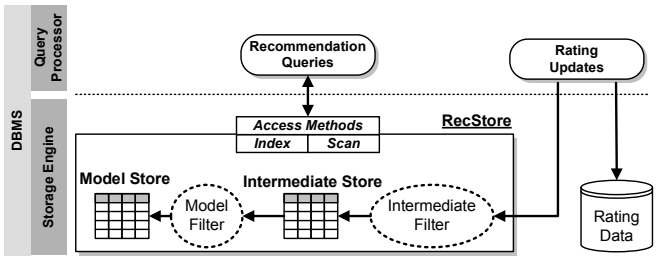


Figure 3: RecStore Architecture

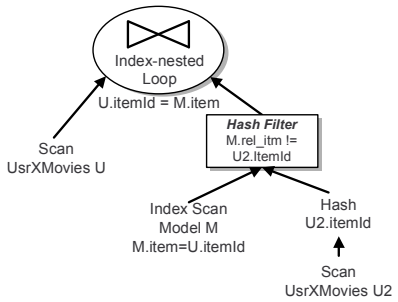


Figure 4: Query plan for 2^{nd} query in Figure 2

deletions, or changes to the ratings table), it applies an *intermediate filter* that determines whether the update will affect the contents of the *intermediate store* (Section 4.1.1).

- **Model store and filter**. The *model store* represents the materialized model that matches the exact storage schema needed by the recommender method (e.g., *(itm, rel_itm, sim)* for the item-based model covered in Section 2). Any changes to the intermediate store goes through a *model filter* that determines whether it affects the contents of the *model store* (Section 4.1.2).

The DBMS query processor requests data from *RecStore* while executing recommender queries. *RecStore* employs two standard DBMS access methods to interface with the query processor: *scan*, i.e., return all model data, and *index*, i.e., return only model data satisfying a given condition (e.g., item id = x). The access methods can produce tuples (i.e., model values) either directly from the model store, or on demand from the intermediate store, or the base ratings data; these query processing details are covered in Section 4.2. As an example, consider query plan given in Figure 4 that retrieves all tuples in the *Model* relation with item ids equal to those rated by user X (this operation is performed by query two in Figure 2). This plan performs an index scan over the model to perform the join between *usrXMovies*. In our experience, most access to *RecStore* will be index-based, as recommendation generation queries require only a portion of the model (similar to Figure 4).

4. RecStore: BUILT-IN ONLINE DBMS-BASED RECOMMENDERS

The main objective of *RecStore* is to bring online model support to existing recommender queries for various workloads and recommendation methods. This objective presents three main challenges that we address in the rest of this paper: (1) Efficient online incremental maintenance of the recommender model, i.e., avoiding expensive model regeneration with each update (Section 4.1). (2) The ability to adapt the system to various workloads, e.g., query or update-intensive workloads (Section 4.2). (3) The ability to support various existing recommender methods (Section 5).

4.1 Online Model Maintenance

This section describes the framework for online model maintenance within *RecStore*. The framework is extensible, and its specific functionality is determined by the underlying recommendation method. While this approach may seem overly-tailored to each specific method, we note that many methods, especially collaborative filtering, share commonalities in model structure. We defer such discussion until later in Section 5. For now, we use the example of the item-based cosine model to illustrate *RecStore*'s approach to providing online model maintenance, consisting of two steps.

4.1.1 Step 1: Intermediate Filter

We describe the functionality of the intermediate filter with an example using the item-based cosine method described in Section 2. For this method, the intermediate store contains a "deconstructed" cosine score (Equation 3), where we store for each item pair (i_p,i_q) that share at least *one* co-rated dimension (1) $pdot(i_p, i_q)$, their partial dot product, (2) $len_p(i_p, i_q)$ and $len_q(i_q, i_p)$, the partial length of each vector for only the co-rated dimensions, and (3) $co(i_p, i_q)$, the number of users who have co-rated items i_p and i_q. This data is stored as a six-column relation, where the first two columns store the item id pairs, while the last four columns store the four statistics just described.

RecStore employs an *intermediate filter* upon receiving a rating update $\mathcal{R}$. The intermediate filter performs three tasks in the following order. (1) *Filter*. This task determines whether $\mathcal{R}$ will be used to update entries in the intermediate store. If not, $\mathcal{R}$ is immediately dropped (but still stored in the ratings data). This step is required by the adaptive maintenance and load shedding techniques discussed later Section 4.2. In the general case this step will not drop any updates. (2) *Enumeration*. This task determines all intermediate store entries $\mathcal{E}$ that will change due to $\mathcal{R}$. For our item-based cosine example with a new rating for item i_p, $\mathcal{E}$ would contain all entries (i_p,i_q) for which items i_p and i_q are co-rated by the user u. (3) *Updates*. Finally, all statistics, functions, or data structures in the intermediate store associated with an entry $e \in \mathcal{E}$ are updated. These updates are then forwarded to the model filter. For our item-based cosine example, the stored statistics are updated as follows, assuming a new rating for item i_p with value s_p: $pdot(i_p, i_q) = pdot(i_p, i_q) + s_p \times s_q$, $len_p(i_p, i_q) = len_p(i_p, i_q) + s_p$, $len_q(i_q, i_p) = len_q(i_q, i_p) + s_q$, and $co(i_p, i_q) = co(i_p, i_q) + 1$.

Together, the intermediate filter and store are the keys to efficient online model maintenance in *RecStore*. The filter reduces update processing overhead by allowing *RecStore* to only process the updates necessary to maintain an accurate intermediate representation. The contents of the intermediate store keep computational overhead low for online maintenance by allowing *RecStore* to quickly update the intermediate store and, once updated, quickly derive a final model score from the intermediate representation.

4.1.2 Step 2: Model Filter

Upon receiving updates from the intermediate filter, the *model filter* executes the same three tasks as the intermediate filter (i.e., filter, enumeration, and updates), except applied to the model store instead of the intermediate store. Continuing our item-based cosine example, its model store contains entries of the form (i_p,i_q,sim(i_p,i_q)), i.e., the item-based model schema discussed in Section 2. The model filter uses the statistical updates from the intermediate store for item pairs (i_p,i_q) to update the similarity score in the model store entry (i_p,i_q, sim(i_p,i_q)) as follows per Equation 2: (1) If statistic $co(i_p, i_q) < 50$, then $sim(i_p, i_q)$ is updated as:

$$sim(i_p, i_q) = \frac{co(i_p, i_q) * pdot(i_p, i_q)}{50 * \sqrt{len_p(i_p, i_q)}\sqrt{len_q(i_p, i_q)}}$$

(2) If statistic $co(i_p, i_q) \geq 50$, we update $sim(i_p, i_q)$ as:

$$sim(i_p, i_q) = \frac{pdot(i_p, i_q)}{\sqrt{len_p(i_p, i_q)}\sqrt{len_q(i_p, i_q)}}$$

Updating the similarity score is the final step in the *RecStore* online maintenance process.

4.2 Adaptive Strategies for System Workloads

This section discusses how *RecStore* adapts to different workload characteristics. We first discuss generic maintenance strategies that help realize an update and query efficiency trade-off. We then discuss load-shedding for update-intensive workloads.

4.2.1 Update vs. Query Efficiency Trade-off

While the intermediate and model store are beneficial to *RecStore*, their sizes may lead to non-trivial maintenance costs. For instance, in item-item or user-user collaborative filtering, the size of the model can reach O(n^2), where n is the number of items (or users). In this case, *RecStore* could be responsible for updating and maintaining data for O(n^2) items (or users) in its intermediate and model store, leading to burdensome maintenance costs. In this section, we explore a trade-off: reducing the storage and maintenance of data in the intermediate store and model store (i.e., the *internal maintenance* approach) in return for sacrificing query processing (i.e., recommendation generation) efficiency.

RecStore can be tuned to realize an efficiency trade-off between updates and query processing. The basic idea is to maintain α entries in the intermediate store, β entries in the model store, and require the invariant that $\alpha \geq \beta$, i.e., all entries in the model store are also maintained in the intermediate store. Both values cannot be greater than $\mathcal{M}$: the total possible number of entries, a model-specific value (e.g., for item-based models $\mathcal{M} = \mathcal{I}^2$).

Low values of α and β imply low incremental update latency as the filters update fewer entries in the intermediate and model stores. On the other hand, during query processing, the access methods must service requests from the query processor by producing model values in the following order of efficiency: (1) directly from the model store if the entry is maintained there. (2) If the entry is not maintained in the model store but maintained in the intermediate store, the model value is produced *on-demand* from the intermediate store (e.g., from the intermediate statistics covered in Section 4.1.1 for the item-based cosine method). (3) If the entry is not maintained in the intermediate nor the model store, the model value must be produced *on-demand* using the base ratings data (e.g., using Equation 1 for the item-based cosine method). Thus, as α and β decrease, query processing latency increases as more model values must be produced *on demand*. Larger values of α and β have a reverse effect on update and query processing efficiency.

Using the maintenance parameters α and β allows *RecStore* to be tuned for a wide range of workloads. More update-intense workloads can lower values of α and β at the cost of increasing recommender query latency. Meanwhile, query-intense workloads can use larger values of α and β at the cost of increasing update overhead. We now explore several strategies for α and β settings; experimental analysis for these strategies is given in Section 6.

- **Extreme Approaches**. Two extreme approaches can be taken by *RecStore*: (1) *Materialize all*. In this approach $\alpha = \beta = \mathcal{M}$, meaning *RecStore*'s intermediate and model stores maintain all required model information. RecStore filters just apply the conditions imposed by the specific similarity functions upon receiving a rating update. Recommendation generation, i.e., the query processing functionality that generates recommended items, is most efficient at this extreme. However, storage and maintenance costs are at their highest with this approach. (2) *Materialize none*. In this approach $\alpha = \beta = 0$, and basically mimics the use of regular DBMS views that we recompute model values *on demand*. In this approach there is no need for the intermediate store, model store, nor filters. Recommendation generation for this approach is very expensive, but incurs *no* storage and maintenance costs as nothing is maintained.

- **Intermediate Store Only**. In this approach $\alpha = \mathcal{M}$ and $\beta = 0$. This approach (abbr. *Intermediate Only*) represents a middle ground between *Materialize All* and *Materialize None*, where we materialize the intermediate store in full for all required model information, while not maintaining the model store. This means that the initial filter will be applied on all incoming updates as described in Section 4.1, while there is no filter for the model store. The recommendation generation process for a requested object *o* (e.g., item or user) needs to rebuild part of the model store that includes *o* using the fully maintained intermediate store. This rebuilding process makes this approach incur higher query processing cost compared to the *Materialize All* approach, but much lower query processing cost than the *Materialize None* approach. On the other hand, storage and maintenance costs are lower than the *materialize all* approach, as the model store is nonexistent.

- **Full Intermediate Store and Partial Model Store**. This approach (abbr. *Partial Model*) sets $\alpha = \mathcal{M}$ and $\beta = N$, and represents a middle ground between the *Materialize all* and *Intermediate Only* approaches. This approach materializes only a *portion* of the model store, i.e., only N objects (e.g., items or users), while materializing the intermediate store in full. We employ *hotspot detection* (described in Section 4.2.2) to select the N items in the model store. This approach directs the initial filter will be applied to all incoming updates. All updates made to the intermediate store are still forwarded to the model filter as described in Section 4.1,

however, the model filter only accepts updates for the qualifying N objects, and their related objects, that are maintained in the model store. The query processing and storage/maintenance overhead for this approach lies between the *Materialize all* and *Intermediate Only* approaches.

- **Partial Intermediate Store and Partial Model Store**. This approach sets $\alpha = K$ and $\beta = N$, and is similar to the *Partial Model* approach, except that we also partially materialize the intermediate store. The model store still maintains data for N objects, while the intermediate store maintains data for K objects. These K and N objects are derived using *hotspot detection* (described next in Section 4.2.2). This approach directs the initial filter only accepts incoming updates for the K objects (items of users), and their related objects, that qualify for storage in the intermediate store. The model filter remains unchanged from the *Partial Model* approach. The query processing and storage/maintenance overhead for this approach lies between the *Partial Model* and *Intermediate Only* approaches.

4.2.2 HotSpot Detection

For the approaches that use partial materialization, α and β should ideally be set to ensure the maintenance of model *hotspots*, i.e., popular or frequently accessed entries. This setting assures efficient query processing over popular model entries, while sacrificing higher query latency for less popular model entries. We use two methods to detect hotspots. (1) *Most accessed*. Keep the α and β most accessed entries from the model determined by simple usage statistics from the access methods. (2) *Most rated*. Keep the α and β most popular entries in the model determined by association with the *most ratings* (e.g., most-rated movies, users who rate the most movies).

4.2.3 Load Shedding

For the special case of update-intense workloads where the system is incapable of processing all ratings updates, *RecStore* is capable of load-shedding. The goal of load-shedding is to process *only* updates that significantly alter the recommender model, thus changing the answer to recommender queries. Load-shedding techniques are model-specific, and *RecStore* executes these techniques in a special filter before the intermediate filter.

As an example, consider the item-based cosine method, where an update should only be processed if it changes the order in the model similarity lists. In this case, altered order in any similarity list can potentially change the answer to a recommender query per Equation 3. An effective heuristic approach to achieve this goal is to process updates that change intermediate store entries with a co-rating count (i.e., the statistic $co(i_p, i_q)$) below a pre-set threshold $\mathcal{T}$. The intuition here is that low co-rated items have less terms defining their cosine distance (Equation (1)), thus an update will likely alter the score significantly compared to more highly co-rated items. Of course, more sophisticated statistical techniques can apply. However, any load-shedding approach should remain simple to evaluate and maintain due to its mission-critical purpose.

5. RECSTORE EXTENSIBILITY

RecStore provides a generic extensible architecture capable of supporting different recommendation methods. This section first demonstrates how to register a preference method with *RecStore*. We then provide various case studies demonstrating how *RecStore* accommodates other item-based collaborative filtering methods.

```
DEFINE RECSTORE MODEL ItmItmCosine
FROM    Ratings R1, Ratings R2
WHERE   R1.itemId <> R2.ItemId AND R1.userId = R2.userId
WITH INTERMEDIATE STORE:
   (R1.itemId as item, R2.itemId as rel_itm, vector_lenp,
    vector_lenq, dot_prod, co_rate)
WITH INTERMEDIATE FILTER:
  ALLOW UPDATE WITH My_IntFilterLogic(),
  UPDATE vector_lengthp AS R1.rating*R1.rating,
  UPDATE vector_lengthq AS R2.rating*R2.rating,
  UPDATE dot_prod AS R1.rating*R2.rating,
  UPDATE co_rate AS 1
WITH MODEL STORE:
   (R1.itemId as item, R2.itemId as rel_itm, COMPUTED sim)
WITH MODEL FILTER:
  ALLOW UPDATE WITH My_ModFilterLogic(),
  UPDATE sim AS
    if (co_rate < 50)
       co_rate*dot_prod/50*sqrt(vector_len1)* sqrt(vector_len2);
    else
       co_rate/sqrt(vector_len1)* sqrt(vector_len2);
```

Figure 5: Registering a recommendation method

Finally, we discuss how *RecStore* supports recommendation methods beyond "memory-based" collaborative filtering.

5.1 Registering a Recommender Method

We provide a syntax for registering a new recommender method model within *RecStore*. Figure 5 gives an example for registering the item-based cosine method. Registration begins by first defining the model name, and then providing a *from* and optional *where* clause to specify the base data used in the model. For the item-based cosine model, the base data comes from the *Ratings* relation, and the where-clause defines a relational constraint (in the form of a self-join) declaring that model entries are (non-equal) items that are co-rated by the same user. The major clauses are:

- **WITH INTERMEDIATE STORE:** defines the data in the intermediate store; in this case the intermediate statistics for the item-based cosine method.
- **WITH INTERMEDIATE FILTER:** defines the intermediate filter in two parts. (1) *Allow Updates With* defines the logic for filtering incoming updates (task 1 discussed in Section 4.1.1), currently contained in a user-defined function. (2) *Update* defines how to compute data in the intermediate store when given a rating update that is *not filtered*; the logic can be given directly or contained in a user-defined function.
- **WITH MODEL STORE:** defines the name and schema of the model store, this schema is exposed to the rest of the DBMS and used by the recommender queries. Any attributes computed from data in the intermediate store are given the *COMPUTED* prefix. Our example item-based cosine follows the schema discussed in Section 4.1.2, where the value *sim* is a computed attribute.
- **WITH MODEL FILTER:** follows the same syntax as the intermediate filter, with the exception that the *compute* clause defines how to update the model store values using data from the intermediate store.

5.2 Item-Based Collaborative Filtering

We now discuss *RecStore* registration for two other item-based collaborative filtering methods [31], namely probabilistic and Pearson

	Probabilistic	**Pearson**
Intermediate Store	len(i_p): partial vector length of for i_p freq(i_p): no. ratings for i_p sum(i_p,i_q): sum of scores for i_p given co-rated item i_q	mean(i_p): mean rating score for i_p stddev(i_p): standard dev. for i_p freq(i_p): no. ratings for i_p sum(i_p): sum of ratings for i_p sumsq(i_p): sum or ratings squared for i_p coprodsum(i_p,i_q): sum of product deviation from mean for i_p given co-rated dimension i_q
Intermediate Filter	Update sum$_q$(i_p,i_q) only where user u co-rated i_p and i_q, always update other statistics. Update logic sum(i_p,i_q)=sum(i_p,i_q)+s_p; len(i_p)=len(i_p)+s_p^2; freq(i_p)=freq(i_p)+1	Always update mean(i_p), stddev(i_p), freq(i_p), sum(i_p), sumsq(i_p).Only update coprodsum(i_p,i_q) if user u co-rated i_p and i_q, *and*mean(i_p) has not changed greater than Δ since last coprodsum(i_p,i_q)recalculation. Update logic freq(i_p)=freq(i_p)+1; mean(i_p)=$\frac{s_p}{freq(i_p)} + \frac{(freq(i_p)-1)mean(i_p)}{freq(i_p)}$; sum($i_p$)=sum($i_p$)+$s_p$; sumsq($i_p$)=sumsq($i_p$)+$s_p^2$; stddev($i_p$)=$\frac{\sqrt{freq(i_p)*sumsq(i_p)-sum(i_p)^2}}{freq(i_p)}$; coprodsum($i_p$,$i_q$)=coprodsum($i_p$,$i_q$)+$(s_p - mean(i_p))(s_q - mean(i_p))$
Model Store	(i_p,i_q,$sim(i_p, i_q)$)	(i_p,i_q,$sim(i_p, i_q)$)
Model Filter	Update entry (i_p,i_q,$sim(i_p, i_q)$) for each statistical update for pair (i_p,i_q) Update Logic $sim(i_p, i_q)$=$\frac{sum_q(i_p,i_q)}{\sqrt{len(i_q)}*freq(i_p)*(freq(i_q))^\alpha}$	*Update* entry for each (i_p,i_q,$sim(i_p, i_q)$) for each statistical update affecting pair (i_p,i_q). Completely *recalculate* coprodsum(i_p,i_q) if mean(i_p) has changed greater than threshold Δ. Update Logic If mean(i_p) has changed less than Δ, $sim(i_p, i_q)$=$\frac{coprodsum(i_p,i_q)}{stddev(i_p)stddev(i_q)}$, otherwise $sim(i_p, i_q)$=$\frac{coprodsum(i_p,i_q)=\sum_{u\in\mathcal{U}_c}(s_p-mean(i_p))(s_q-mean(i_p))}{stddev(i_p)stddev(i_q)}$

Table 1: Realizing probabilistic and Pearson item-based collaborative filtering methods in *RecStore*, summary of implementation approach assuming new rating by user u for item i_p with score s_p

item-based recommenders. We demonstrate each use case assuming a user u has provided a new rating value s_p for an item i_p.

Item-based probabilistic recommender. This method is similar to our running example of the item-based cosine recommender, except the similarity score $sim(i_p, i_q)$ is measured as the conditional probability between two items i_p and i_q as follows.

$$sim(i_p, i_q) = \frac{\sum_{u \in \mathcal{U}_c} r_{u,i_q}}{Freq(i_p) \times (Freq(i_q))^\alpha} \quad (5)$$

Here, r_{u,i_q} represents a rating for item i_q normalized to unit-length, $Freq(i)$ represents the number of non-zero ratings for item i, and α is a scaling factor [17].

The second column of Table 1 provides an approach to implementing the item-based probabilistic method in *RecStore*. The intermediate store contains (1) the partial vector length for item i_q (len(i_q)), (2) the total number of ratings for i_p (freq(i_p)), and (3) the item-pair statistic maintains the running sum ratings for item i_p given that it is co-rated with an item i_q (sum(i_p,i_q)). The intermediate filter updates all single-item statistics, while only updating the pair statistic for which items i_p and i_q are both rated by user u. Each statistic update requires constant time. The *model filter*, upon receiving changes to the intermediate statistics, updates the similarity score $sim(i_p,i_q)$ for pairs i_p,i_q in constant time using the intermediate statistics (equation given in the last row, second column of Table 1).

Item-based Pearson recommender. This method is similar to the item-based cosine method, except it measures the similarity between objects using their Pearson correlation coefficient as follows.

$$sim(i_p, i_q) = \frac{\sum_{u \in \mathcal{U}_c}(R_{u,i_p} - \overline{R}_{i_p})(R_{u,i_q} - \overline{R}_{i_q})}{\sigma_{i_p}\sigma_{i_q}} \quad (6)$$

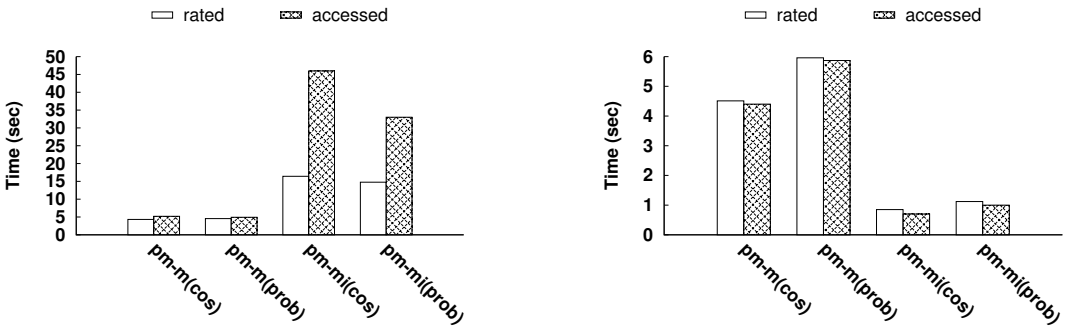
$\mathcal{U}_c$ represents users who co-rated items i_p and i_q, R_{u,i_p} and R_{u,i_q} represent a user's ratings, and $\overline{R}_{i_p}$ and $\overline{R}_{i_q}$ represent the average rating for items i_p and i_q, respectively. σ_{i_p} and σ_{i_q} are the standard deviations for i_p and i_q

The third column of Table 1 provides an approach to implementing the Pearson method in *RecStore*. The intermediate store maintains for an item i_p its mean rating value for an item (mean(i_p)), its standard deviation of rating values (stddev(i_p)), the total number of ratings for i_p (freq(i_p)), the sum of ratings for i_p (sum(i_p)), and the sum of the squared rating values for i_p (sumsq(i_p)). The intermediate store also maintains *coprodsum*(i_p,i_q): the sum of the product of deviations from the mean (i.e., the numerator in Equation 6) for an item pair (i_p,i_q) given that they share at least one co-rated dimension. The *intermediate filter* updates all single-item statistics (those maintained for i_p only). The statistic *coprodsum*(i_p,i_q) is incremented by the product of the deviation of user u's score for i_p (i.e., s_p) from the *newly* calculated *mean*(i_p), and the deviation of i_q (i.e., s_q) from the stored mean for item i_q (*mean*(i_q)). Note that previous rating scores for i_p in the sum deviated from different means, since *mean*(i_p) changed with this update. In essence, we are willing to forgo this difference in *accuracy* as long as *mean*(i_p) has not changed by at least a value Δ since the last calculation of *coprodsum*(i_p,i_q). What we gain in this trade-off is efficiency, since updating *coprodsum*(i_p,i_q) is more efficient than recalculating the sum from scratch.

The *model filter* updates the similarity score $sim(i_p,i_q)$ for pairs i_p,i_q in the model store using one of two methods (both given in the last row, third column of Table 1). (1) If the value *mean*(i_p) had not changed by Δ since the last recalculation of *coprodsum*(i_p,i_q), then we can update *sim*(i_p,i_q) efficiently by dividing *coprodsum*(i_p,i_q) by the product of the standard deviations. Otherwise, we must *recalculate* the value of *coprodsum*(i_p,i_q) from scratch for each entry using the current value of *mean*(i_p).

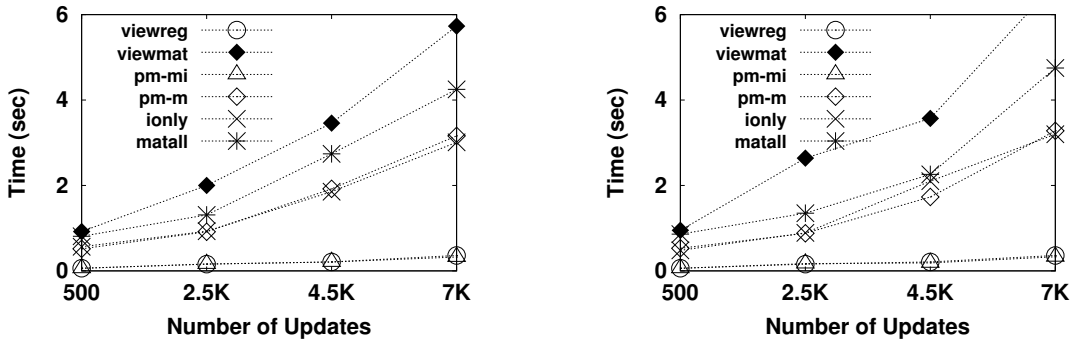
5.3 User-based Collaborative Filtering

The model for user-based collaborative filtering [29] is similar to the item-based approach, except that the model stores groups of similar users (as described in Section 2). Thus, the use cases previously discussed for the item-based approach can apply *directly* to the user-based approach, with the exception that similarity is measured over user vectors in the item rating space.



(a) Query (b) Update

Figure 6: Hotspot Detection



(a) Cosine (b) Update

Figure 7: Update Efficiency

5.4 Non-"Memory-Based" Collaborative Filtering within RecStore

Many other recommendation methods use models that are *not* similarity-based lists, as is the case with the "memory-based" collaborative filtering techniques we have explored. In general, *RecStore* can support these different recommendation techniques as long as their models can be represented by sufficient statistics to update the model incrementally. For instance, recommendation methods that use sophisticated probabilistic models (e.g., Bayesian Networks [3], Markov decision processes [32]) *do not* lend themselves well to incremental updates, due to the computationally intense optimization process used to learn their parameters. On the other hand, methods that use linear regression to learn a rating prediction model [31] can fit easily within *RecStore*. In this case, the intermediate store can maintain the general linear model statistics: X (the regression design matrix), X^T (X transposed) and f (the regressand). It is known that these statistics are *incrementally updatable* and *sufficient* to learn unknown regression coefficients by solving the system of equations [11]: $X^T X\beta = X^T f$, where β represents the learned regression coefficients. The source for these statistics depends on the recommendation method. Examples include ratings vectors [31], a multi-dimensional ratings base (e.g., multi-dimensional recommenders [2]), or item attributes (e.g., content-based recommenders [7]).

6. EXPERIMENTAL EVALUATION

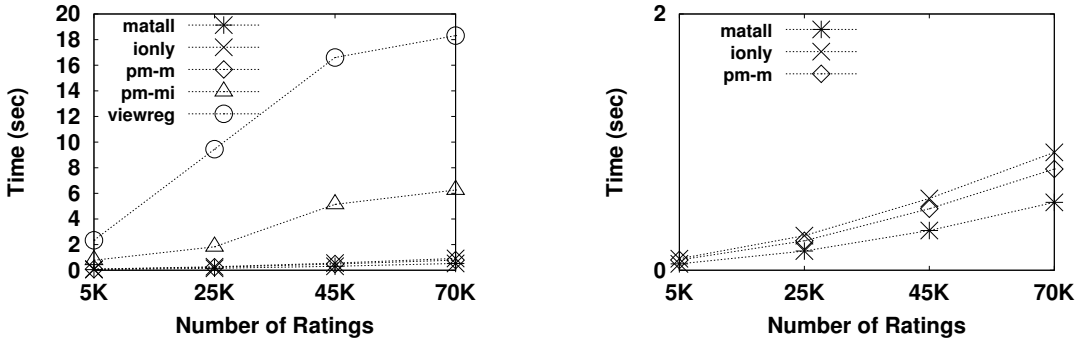
This section experimentally evaluates the performance of a prototype of *RecStore* implemented in between the storage engine and query processor of the PostgreSQL 8.4 database system [28] using the real-world Movielens 10M rating data set [27]. We test various *RecStore* adaptive maintenance strategies based on α and β proposed in Section 4.2.1: materialize all (abbr. ***matall***) where α = $\beta = \mathcal{M}$, intermediate only (***ionly***) where α=$\mathcal{M}$ and $\beta = 0$, partial model hotspot maintenance where α=$\mathcal{M}$ and β is set to 20% of all movies (***pm-m***), and partial intermediate and model hotspot maintenance (***pm-mi***) where α and β are set to 40% and 20% of all movies. We also compare against regular (***viewreg***) and materialized DBMS views (***viewmat***). The ***viewreg*** approach is implemented using a regular PostgreSQL view, but since PostgreSQL does not support materialized views, we provide a fair simulation of ***viewmat*** within *RecStore* by maintaining a materialized *Model* store without the use of an intermediate store.

We provide experiments for: (1) Partial maintenance strategies (Section 6.1), (2) update efficiency (Section 6.2), (3) query efficiency using the query given in Figure 2 (Section 6.3), and (4) a *real* recommender system workload trace consisting of interleaved queries and updates (Section 6.4). Each experiment is run for both the *cosine* and *probabilistic* item-based recommendation method (details of both methods given in Section 5).

The experiment machine is an Intel Core2 8400 at 3Ghz with 4GB of RAM running Ubuntu Linux 8.04. Our performance metric is the *elapsed time* over an average of five runs reported by the PostgreSQL EXPLAIN ANALYZE command.

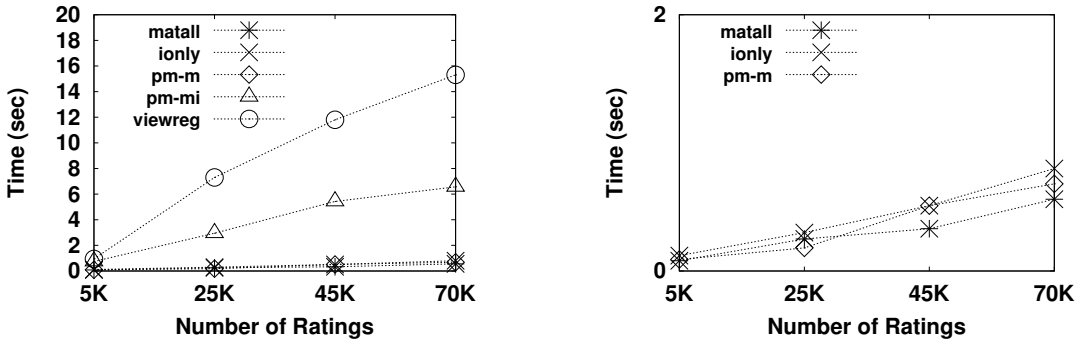
6.1 Hotspot Detection Strategies

This experiment studies the effectiveness of our two hotspot detection strategies covered in Section 4.2: *most-rated* (abbr. ***rated***) and *most-accessed* (abbr. ***accessed***). We use a real workload trace consisting of the continuous arrival of both ratings updates and recommender queries against the MovieLens system [23, 26]. We start with a *Ratings* table that already contains 950K ratings, and report the *total* time necessary to process 1K ratings updates in-



(a) Cosine (b) Cosine Zoomed

Figure 8: Query Efficiency



(a) Probabilistic (b) Probabilistic Zoomed

Figure 9: Query Efficiency

terleaved with 40 recommendation generation queries for different users. Figures 6(a) and 6(b) report performance for ***rated*** and ***accessed*** using both the ***pm-mi*** and ***pm-m*** approaches implementing the *cosine* and *probabilistic* methods. The *update* performance is relatively similar between the ***rated*** and ***accessed*** strategies for all cases. However, the *query* performance of ***rated*** over ***accessed*** exhibits a 50% speedup, as ***rated*** is able to keep model data in the intermediate and model store requested by the recommendation generation queries. Thus, in the rest of this section, we employ the ***rated*** strategy for both ***pm-mi*** and ***pm-m***.

6.2 Update Efficiency

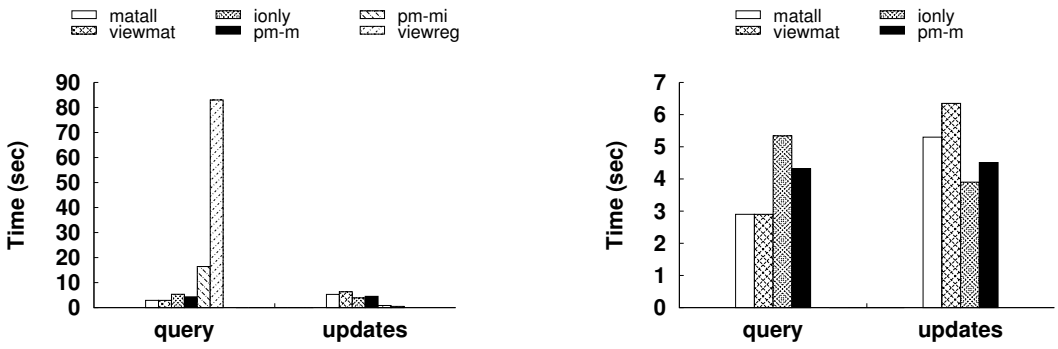
This experiment studies update efficiency and scalability. We start with a *Ratings* table already containing 950K rating tuples, and measure the total time it takes to process 500, 2.5K, 4.5K, and 7K updates, respectively. Figures 7(a) and 7(b) give the results for the *cosine* and *probabilistic* methods, respectively. For both methods, all approaches exhibit the same relative performance. The materialized view (***viewmat***) incurs the most overhead of all approaches. This performance is due to the need, on every update, to recalculate the model score from scratch using the ratings data. The *RecStore* ***matall*** strategy, on the other hand, incurs less update overhead compared to ***viewmat*** due to its intermediate store, that helps it to efficiently update the model store. This experiment confirms that *RecStore* overcomes the update efficiency drawback of materialized views. Both ***ionly*** and ***pm-mi*** exhibit better performance, with ***ionly*** doing slightly better due to not having to maintain a partial model store. Both ***matnone*** and ***pm-mi*** exhibit the best performance due to the low storage and maintenance costs.

6.3 Query Efficiency

This experiment studies query efficiency and scalability. We measure the time to perform the recommender query given in Figure 2 for a user X as the number of tuples in the *Ratings* table increases from 5K to 70K. We choose user X as the user that has rated the *most* movies. Figures 8(a) and 9(a) give the results for the *cosine* and *probabilistic* recommendation methods, respectively. The ***viewreg*** approach (a regular DBMS view) performs very poorly, as it must calculate all requested model scores *from scratch* from the ratings relation. The ***pm-mi*** approach exhibits performance between ***matnone*** and the rest of the approaches, as it must service a fraction of its requests from the ratings data, similar to ***viewreg***. Figures 8(b) and 9(b) zoom in on the ***matall***, ***ionly***, and ***pm-m*** approaches for the *cosine* and *probabilistic* models, respectively. As expected, the ***matall*** approach exhibits the best query processing performance as it must only retrieve values from the model store. Both ***ionly*** and ***pm-m*** exhibit close performance to ***matall***. We do not plot the ***viewmat***, since it exhibits the *same* performance as ***matall***, as the query operates over a completely materialized model relation for both approach. Due to both query and update performance, we can conclude that *RecStore* provides better support for *online* recommender systems compared to existing DBMS approaches (***viewmat*** and ***viewreg***).

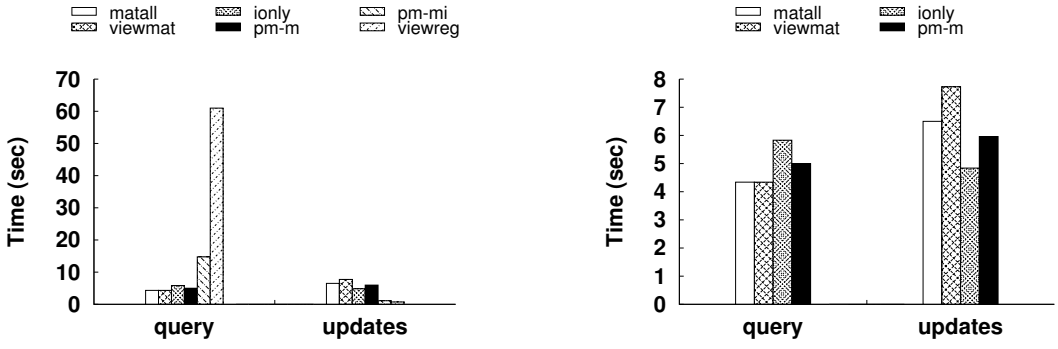
6.4 Update + Query Workload

This experiment uses our real recommender system workload trace (described in Section 6.1) to test *comprehensive* update and query processing performance. Figures 10(a) and 11(a) give the results of both query and updates for the *cosine* and *probabilistic* methods, respectively. Both ***viewreg*** and ***pm-mi*** exhibit poor query pro-



(a) Cosine (b) Cosine Zoomed

Figure 10: Real Workload



(a) Probabilistic (b) Probabilistic Zoomed

Figure 11: Real Workload

cessing performance for the workload, with ***viewreg*** performing almost an order magnitude worse than other approaches. While the ***viewreg*** performance is expected, the ***pm-mi*** performance is more surprising. Both ***viewreg*** and ***pm-mi*** exhibit the best update performance out of all approaches, as confirmed by our previous experiments (Section 6.2). However, the query processing performance of ***viewreg*** makes it an unattractive alternative, while the update/query processing tradeoff for ***pm-mi*** is a borderline choice due to its high query processing penalty. Figures 10(b) and 11(b) remove the ***viewreg*** and ***pm-mi*** numbers to zoom in on the other approaches for the *cosine* and *probabilistic* models, respectively. Both the ***matall*** and ***viewmat*** approaches exhibit the same query processing performance that is superior to ***ionly*** and ***pm-m***. As for updates, we note again that ***matall*** (*RecStore*) provides more efficient update performance over ***viewmat*** (materialized views). Meanwhile, ***pm-m*** and ***ionly*** show superior update performance to ***matall*** and ***viewmat***, with ***ionly*** providing the best performance.

In this experiment, we can observe the update/query processing trade-off discussed in Section 4.2.1 for high values of α and β (***matall***) compared to lower values of α and β (***ionly*** and ***pm-mi***). Thus, for slightly more update-heavy recommender systems, the ***ionly*** or ***pm-mi*** is preferable due to efficient updates with little query processing penalty. Meanwhile, for more query-heavy systems, the ***matall*** approach is preferable with tolerable update penalty.

7. RELATED WORK

Collaborative Filtering. The term collaborative filtering has a broad definition [1, 3]. We mainly focus on the original *memory-based* collaborative filtering approach [18, 29], so called because it "remembers" the ratings history of the entire user/item spectrum to provide recommendations [1, 3]. The scope of most work within collaborative filtering systems has been from a *user-centric* perspective, e.g., providing users with quality [15, 22] and trustworthy recommendations [25]. Other work has explored high-level approaches to memory-based collaborative filtering (e.g., user-based [18, 29], item-based [17, 31], hybrid [4]) and their effect on recommendation quality. There is a *scarcity* of work that studies recommenders from a systems perspective, i.e., measuring query processing efficiency of different architectures. Herlocker et al. in their 2004 detailed evaluation of recommender systems state [15]:

"We have chosen not to discuss computation performance of recommender algorithms. Such performance is certainly important, and in the future we expect there to be work on the quality of time-limited and memory-limited recommendations."

To date, the research community is *still* lacking such important work on recommender system performance. Further, very little work has suggested a systems solution to online recommender systems. An exception is the Google News recommender [6]. However, this work is specific to the recommendation methods created specifically for Google's click logs. In contrast, our work takes a more *generic* approach to online recommender systems by offering a generic and extensible framework within the DBMS engine that accommodates various recommendation methods.

DBMS and Recommender Systems. Little systems research has addressed the intersection of database and recommender systems (as asserted by [16]). The AWESOME system [33] suggests recommendation methods to use based on the characteristics of the data stored in a database. Closer to our work is FlexRecs [19],

that studies a flexible model and workflow for expressing a number of recommendation methods. FlexRecs compiles its workflow into a series of conventional SQL queries to execute the recommendation process. This work shows that implementation of many different recommendations methods is possible within a DBMS. While FlexRecs addresses the implementation of recommendation logic using a DBMS, it still assumes the model building phase for many recommendation methods is performed *offline*. Our work addresses *online* model support for DBMS-based recommender queries.

Database Views. We can employ DBMS views as a solution to online collaborative filtering. Views are a fundamental topic within the data management research community, with a rich volume of research addressing various view aspects, including, but not limited to, view composition, materialized view maintenance, and query processing using views [13]. Views provide a *general* solution to a wide range of data management problems, including security (i.e., data access restriction), transparency from a physical schema, and ease-of-use. In this paper, we study the *specific* data management problem of online maintenance of recommender models, for which DBMS views incur serious efficiency drawbacks.

8. CONCLUSION

This paper presented *RecStore*, an extensible and adaptive DBMS storage engine module that provides *online* support for recommender queries. We fist presented the generic architecture of *RecStore*, and then described how *RecStore* supports online recommender model maintenance by enabling fast incremental updates to the model by implementing an intermediate and model store. We described how *RecStore* adapts to various system workloads, and provides load-shedding support for update-intense workloads. We then demonstrated the extensibility of *RecStore* by presenting a declarative model registration language, and provided case-studies showing how *RecStore* accommodates various recommendation methods. Using a real recommender system workload and a system prototype of *RecStore* inside PostgreSQL, our experimental results show that *RecStore* provides superior performance to existing DBMS view approaches to support online recommender systems. Further, the experiments also confirm that *RecStore* is indeed adaptive to update-heavy or query-heavy recommender system workloads.

9. 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] G. Adomavicius et al. Incorporating Contextual Information in Recommender Systems Using a Multidimensional Approach. *TOIS*, 23(1), 2005.

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

[4] R. Burke. Hybrid Recommender Systems: Survey and Experiments. *User Modeling and User-Adapted Interaction*, 12(4), 1997.

[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] S. Debnath, N. Ganguly, and P. Mitra. Feature Weighting in Content Based Recommendation System Using Social Network Analysis. In *WWW*, 2008.

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

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

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

[11] G. H. Golub and C. F. V. Loan. *Matrix Computations*. Johns Hopkins, 1989.

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

[13] A. Gupta and I. S. Mumick. *Materialized Views: Techniques, Implementations, and Applications*. MIT Press, 1999.

[14] J. L. Herlocker, J. A. Konstan, A. Borchers, and J. Riedl. An Algorithmic Framework for Performing Collaborative Filtering. In *SIGIR*, 1999.

[15] J. L. Herlocker, J. A. Konstan, L. G. Terveen, and J. T. Riedl. Evaluating Collaborative Filtering Recommender Systems. *TOIS*, 22(1), 2004.

[16] Y. E. Ioannidis and G. Koutrika. Personalized Systems: Models and Methods from an IR and DB Perspective. In *VLDB*, 2005.

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

[18] 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.

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

[20] B. Krulwich. Lifestyle Finder: Intelligent User Profiling Using Large-Scale Demographic Data. *Artificial Intelligence Magazing*, 18(2), 1997.

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

[22] S. M. McNee, J. Riedl, and J. A. Konstan. Making recommendations better: an analytic model for human-recommender interaction. In *CHI*, 2006.

[23] 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.

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

[25] B. Mobasher et al. Toward trustworthy recommender systems: An analysis of attack models and algorithm robustness. *ACM TOIT*, 7(4), 2007.

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

[27] Movielens Datasets: http://www.grouplens.org/node/73.

[28] PostgreSQL: http://www.postgresql.org.

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

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

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

[32] G. Shani, R. I. Brafman, and D. Heckerman. An MDP-Based Recommender System. In *UAI*, 2002.

[33] A. Thor and E. Rahm. AWESOME - A Data Warehouse-based System for Adaptive Website Recommendations. In *VLDB*, 2004.