RecBench: Benchmarks for Evaluating Performance of Recommender System Architectures

Justin J. Levandoski Michael D. Ekstrand Michael J. Ludwig
Ahmed Eldawy Mohamed F. Mokbel John T. Riedl

Department of Computer Science and Engineering, University of Minnesota, Minneapolis, MN
{justin,ekstrand,mludwig,eldawy,mokbel,riedl}@cs.umn.edu

ABSTRACT

Traditionally, recommender systems have been "hand-built", implemented as custom applications hard-wired to a particular recommendation task. Recently, the database community has begun exploring alternative DBMS-based recommender system architectures, whereby a database both *stores* the recommender system data (e.g., ratings data and the derived recommender models) and *generates* recommendations using SQL queries. In this paper, we present a comprehensive experimental comparison of both architectures. We define a set of benchmark tasks based on the needs of a typical recommender-powered e-commerce site. We then evaluate the performance of the "hand-built" MultiLens recommender application against two DBMS-based implementations: an unmodified DBMS and *RecStore*, a DBMS modified to improve efficiency in incremental recommender model updates. We employ two non-trivial data sets in our study: the 10 million rating MovieLens data, and the 100 million rating data set used in the Netflix Challenge. This study is the first of its kind, and our findings reveal an interesting trade-off: "hand-built" recommenders exhibit superior performance in model-building and pure recommendation tasks, while DBMS-based recommenders are superior at more complex recommendation tasks such as providing filtered recommendations and blending text-search with recommendation prediction scores.

1. INTRODUCTION

Research and development of recommender systems has been an exciting and vibrant field for over a decade, having produced proven methods for "preference-aware" computing. The purpose of a recommender system is to help users identify interesting, personalized items or content from a large search space. For example, recommenders have successfully helped users find books and media of interest from a massive inventory base (Amazon [20]), news items from the Internet (Google News [12]), and personalized movie suggestions (Netflix [25], Movielens [22]). Traditionally, recommender systems have been "hand-built", implemented as custom software stacks hard-wired to the particular recommendation task at hand. In terms of scalability, these systems assume all data fits in memory, sometimes sacrificing accuracy (i.e., recommendation quality) for speed if data sizes require out-of-memory computation [29].

Researchers from the data management community have recently begun exploring alternative architectures for recommender systems that leverage the capabilities of a database management system (DBMS). In these approaches, the DBMS both *stores* the recommender system data and *generates* recommendations. Many popular recommendation methods can be implemented using an unmodified DBMS using only SQL [18], and by modifying the database storage engine, the DBMS adapts well to both the update *and* query-intensive online environments that deploy recommender systems [19]. Furthermore, implementation of a recommender within the DBMS instantly leverages indexing, query optimization, and query execution constructs that have proven to be an efficient and scalable solution for many data-driven tasks. This "DBMS-based" approach runs counter to traditional recommender system architectures that implement recommendation logic outside a persistent data storage system such as a DBMS.

Given these disparate but capable approaches to implementing recommender systems, i.e., the "hand-built", the unmodified DBMS, and modified DBMS architectures, it is natural to wonder how each system performs under realistic workloads. Our paper addresses this topic by presenting an experimental study of approaches to building recommender systems. Our study involves the comparison of three approaches: (1) a *hand-built* architecture using the MultiLens recommendation engine [21, 24], (2) an *unmodified* DBMS using the PostgreSQL database, and (3) *RecStore* [19], a PostgreSQL database engine modified to perform efficient incremental updates to the recommender model (used to generate recommendations).

To measure the useful performance of each architecture, we define a rich set of six benchmark tasks based on the needs of modern recommender-powered e-commerce scenarios. Our tasks test many recommender performance aspects, and were created based on our experience with real-world recommender systems such as MovieLens [22]. Since our goal is to test relative performance of system architectures, we run our benchmark tasks on each architecture using a single recommendation technique: item-based collaborative filtering [29], a popular recommendation method used widely in both academic and commercial systems (e.g., Amazon [20]). Our tasks, however, are in no way limited to item-based CF, and easily apply to a wide array other recommendation techniques. Furthermore, our study employs two real, large-scale data sets: the MovieLens [23] and Netflix prize [26] data sets.

No previous work has performed such a rigorous study of recommender system performance. Most evaluations have focused on recommendation *quality* rather than system performance [1, 10,

Proceedings of the VLDB Endowment, Vol. 4, No. 11

9, 16]. In their 2004 detailed evaluation of recommender systems [16], Herlocker et al. state:

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

We have yet to see a comprehensive performance evaluation of recommender system architectures to date[1]. This paper fills that void.

The goal of this work is to provide an objective understanding of the performance implications of each measured architecture when run on a query workload and data set likely to be found in real-world e-commerce contexts. The aim is for our results to spur discussion of interesting topics that bridge both the database and recommender system communities. Our results reveal an interesting trade-off. Both the unmodified DBMS and *RecStore* are faster than the hand-built architecture at performing complex recommendation tasks, such as producing filtered recommendations (e.g., recommend movies released before a given year) and blending text-search and recommendation ranking scores. The hand-built approach is more efficient at performing pure recommendation tasks, such as producing unfiltered recommendation results and predicting a user's interest in a given item. Meanwhile, the hand-built approach is very efficient at building a collaborative filtering model. The unmodified DBMS perform poorly at this build task, while *RecStore* exhibits tolerable performance between the hand-built and DBMS approaches.

The rest of this paper is organized as follows. Section 2 provides an overview of recommender systems and the details of item-based collaborative filtering. Section 3 introduces our benchmark tasks. Section 4 details the system architectures used in this study. Section 5 provides the results of our experimental benchmark, followed by a summary discussion in Section 6. Finally, Section 7 concludes this paper. The appendix contains further discussion of architectural comparisons, as well as implementation details for our experiments.

2. OVERVIEW OF THE RECOMMENDATION PROCESS

In this section, we first describe the general purpose of recommender systems, followed by a presentation of item-based collaborative filtering, the recommendation technique used in our experiments.

2.1 Recommender Systems

A recommender system is an information agent that provides suggestions for items likely to be of interest to the user [10]. This broad definition is due to the large number of disparate strategies for recommending items. For instance, Burke et al. classifies recommender system into five classes based on their recommendation strategy [10]. The traditional and most popular strategy is collaborative filtering (CF) [20, 28].

The CF approach 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 $\mathcal{I}_{u_j} \subset \mathcal{I}$. We assume users express opinions through a numeric ranking (e.g., an integer on scale of 1 through 5), however unary/binary ranking is also possible (e.g., hyperlink clicks, Facebook "likes" [14]). Given a querying user u_q, CF produces a set of recommended items $\mathcal{I}_r \subset \mathcal{I}$ that u_q is predicted to like the most.

[1]Sarwar, et al. did analyze recommender throughput [29], though to determine parameter settings for a hand-built recommender.

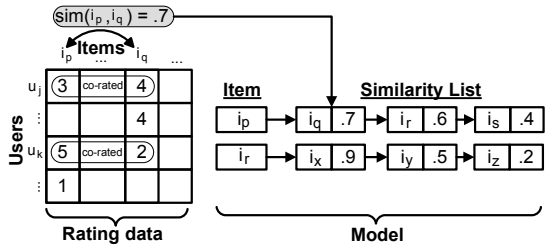


Figure 1: Item-based model generation.

2.2 Item-Based Collaborative Filtering

This section describes the details of item-based CF. We chose item-based CF for our experiments due to its popularity and widespread adoption in commercial systems (e.g., Amazon [20]). Thus, we believe it most accurately reflects a real-life e-commerce recommendation scenario for our benchmarks (though our benchmark tasks are generic enough to support *any* recommendation technique). Item-based CF that consists of two phases: *model-building* and *recommendation generation*. We describe the details of each phase below.

2.2.1 Model Building

Item-based collaborative filtering builds a model that stores, 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-based 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 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.

Many similarity scoring measures have been proposed [29] (e.g., Pearson correlation, cosine distance). We use the cosine distance as our similarity measure in the experiments due to its widespread adoption. Given two item vectors i_p and i_q, the cosine distance is defined as:

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

It is common practice to truncate the item-based model by storing, for each similarity list $\mathcal{L}$, only the k items with highest similarity score, where k is much smaller than the total number of items. Truncation has been observed to have minimal impact on the quality of recommendations [29].

2.2.2 Recommendation Generation

Given a querying user u_q, recommendations are produced by computing u_q's predicted rating $P_{(u_q,i)}$ for an item i [29]:

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

Before this computation, we reduce each similarity list $\mathcal{L}$ to contain only items *rated* by user u_q. The prediction is the sum of $r_{u_q,l}$, 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-n items ranked by $P_{(u_q,i)}$.

3. BENCHMARK TASKS

It is important in systems-related fields to objectively evaluate architectural approaches to both understand their limitations and benefits and advance the field in general. In this section we describe a set of six rich benchmark tasks to objectively evaluate the performance of recommender system architectures. In proposing these tasks, we aim to capture a wide array of common functionality typical in recommender-powered e-commerce scenarios.

We have designed these tasks to be independent of the underlying recommendation technique. They are therefore suitable for end-to-end performance evaluations of any recommender algorithm implementation. In this section, we describe each task generically in terms of its basic functionality and objective. Later, in our experiments (Section 5), we describe each task's implementation specific to our data sets (MovieLens and Netflix), recommendation technique (item-based CF), and architectural approaches.

Initialization. The objective of initialization is to prepare the recommender system for "live" use. For most systems, initialization means building a recommender model based on previous ratings or purchase history of a user base.

Pure recommend. This task simulates a user navigating to an e-commerce site (e.g., Netflix), where they receive a set of recommendations from the system on a home page. The objective of this task is to produce for the user a set of n recommended items. This tasks tests the system efficiency in executing the common top-n recommendation process.

Filtered recommend. This task models the user requesting recommendations for only a specific set of items (e.g., comedy movies released after 1990). The objective of this task is to produce a set of n recommended items that match one or more constraints placed on the item metadata (e.g., movie year). This task tests the system efficiency in querying item metadata and producing a set of n filtered recommendations.

Blended recommend. This task models our system user requesting recommendations based on a free-text search. The objective of this task is to produce a set of n recommended items using the average of the item recommendation score and text-search score. This task tests the system's ability to efficiently incorporate IR-based scoring techniques into its recommendation process.

Item prediction. This task models our system user navigating to a specific target item (e.g., movie, book), whereby the recommender system guages the user's interest in the item. The objective of this task is to generate the user's predicted rating for the target item. This task tests the system performance in producing predictions for a single item.

Item update. This task models a new item being added to the system. The objective of this task is to get the item into circulation as a recommendation candidate as soon as possible after users start rating it or customers start buying it. For most systems, this task will require incorporating the item's ratings or purchase history into the recommender model built originally by the *initialization* task.

4. RECOMMENDER SYSTEM ARCHITECTURES

This section provides an overview of the three different recommender system architectures tested in this paper, namely an unmodified DBMS, RecStore (a modified DBMS), and the hand-built MultiLens system. It is important to note that each system provides the same end result, as each implements the same recommendation method (item-based CF from Section 2). For further discussion and a non-experimental comparison of the advantages and disadvantages of each architecture, please see Appendix A.

```
SELECT R1.mid as item,
   R2.mid as rel_item,
   SUM(R1.rating * R2.rating)/
   (sqrt(sum(R1.rating*R1.rating))*
   sqrt(sum(R2.rating*R2.rating))
   +100) as sim
FROM Ratings R1, Ratings R2
WHERE R1.itemId <> R2.itemId AND
      R1.userId = R2.userId
GROUP BY R1.itemId,R2.itemId;
```

(a) Item-item cosine model generation

```
--Find movies rated by REC_USER_X
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
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
LIMIT N;
```

(b) Recommendation generation

Figure 2: Collaborative filtering SQL examples.

4.1 An Unmodified DBMS

A standard database management system is perfectly capable of implementing a very wide spectrum of recommendation methods [18]. This section provides an overview of how to use a DBMS to implement an item-based recommender system, labeled ***DBMS*** in our experiments. We chose PostgreSQL as our DBMS due to its superior performance in initial tests versus other open-source solutions, though the techniques we discuss apply to any relational database.

Data storage. Ratings data is stored in a relation Ratings(userId, itemId, rating), where userId and itemId represent unique ids of users and items, respectively. We store the model using a three-column table model(item, rel_itm, score). Any item or user metadata (e.g., movie titles, user information) are stored in separate relations.

Model building. Standard SQL is sufficient to build collaborative filtering models within the DBMS. For example, Figure 2a depicts an SQL query that creates the item-based cosine model. This query computes the cosine score for item pairs (i.e., Equation 1). Note that this query involves a self-join over the Ratings relation, which can be rather large.

Recommendation generation. Recommendation generation can also be implemented using standard SQL, thus leveraging the full power of the query optimizer. Figure 2b provides the SQL for generating movie predictions for a user X using the weighted sum method from Equation 2. For presentation clarity, we provide two separate queries, though the entire query can be nested. The first query finds movies the user has rated (which can also be precomputed or maintained by the system), while the second query performs a weighted sum prediction over the reduced item-based cosine model to produce recommendations.

4.2 RecStore: A Modified DBMS

Typically in collaborative filtering, the model-building phase is performed *offline*, while the recommendation generation phase is performed *online*. In previous work, we built *RecStore* [19], a database storage engine module that supports a completely *online* collaborative filtering process. *RecStore* enables fast incremental updates to the recommender model, while still supporting efficient recommendation queries. The basic idea behind RecStore is to maintain an intermediate store containing intermediate statistics sufficient to compute similarity scores in the collaborative filtering model. RecStore also maintains a model store, which is the materialized collaborative filtering model itself. Upon receiving a ratings update, RecStore uses the intermediate store to quickly update the affected scores in the model store.

From a query-writing perspective, *RecStore* hides details of its internal maintenance strategies, and exposes the model to the query

processor as a relational table. Thus, the SQL queries introduced in Section 4.1 can remain unchanged in a *RecStore*-enabled DBMS.

Through tuning, *RecStore* adapts to different system workloads (e.g., update or query-intensive workloads) to realize an efficiency trade-off between updates and query processing. For instance, *RecStore* can store only intermediate statistics, and not the full materialized model (the model store). This configuration has the advantage of lower storage and update overhead and fast updates, since the materialized model is not maintained or updated. However, query processing in this configuration is at a disadvantage, since the model similarity scores must be generated at query runtime from the intermediate statistics. It was experimentally shown that various RecStore strategies outperform existing DBMS approaches (using regular and materialized views) in both query and update efficiency on a real recommender workload [19].

In this paper, our experiments contain an implementation of *RecStore* using two internal maintenance strategies. (1) The *maintain-all* strategy maintains both the intermediate and model store, labeled ***RecStore MA*** in our experiments. (2) The *intermediate-only* strategy maintains the intermediate store but not the model store, labeled ***RecStore MI*** in our experiments. We don't present all details of *RecStore* here. However, our extended technical report [19] provides the technical details of *RecStore*.

4.3 MultiLens: The Hand-Built Approach

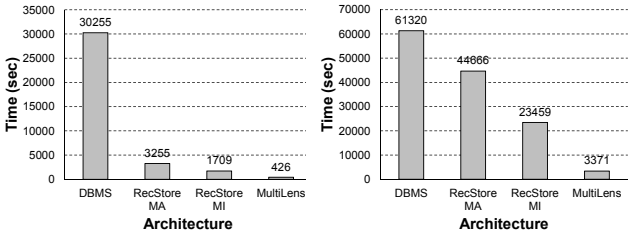
For our hand-built recommender, we used a customized version of the MultiLens collaborative filtering implementation [21, 24], labeled ***MultiLens*** in our experiments. While the MultiLens engine implements a suite of different recommendation methods, our version has been fine tuned to focus on the needs of item-based collaborative filtering using cosine similarity. In this vein, MultiLens implements a multi-threaded model builder, as well as a hand-coded matrix library for storing the ratings data and item-based cosine model in memory. This library creates an array of pointers for each row in a matrix. Each pointer references a compressed array representing the non-null data for that row. Each entry in the compressed array is stored as a primitive `float` type. We chose MultiLens due to its superior performance in initial tests when compared to other general-purpose hand-built systems (e.g., Apache Mahout [3]).

MultiLens also interfaces with an underlying DBMS (the same PostgreSQL database used in the ***DBMS*** approach) through a JDBC layer. The DBMS and MultiLens interact for two main reasons: (1) *Ratings storage*. MultiLens loads ratings from the database to build its in-memory model. (2) *Metadata Queries*. MultiLens queries item metadata stored in the database to produce filtered recommendations (e.g., recommend only movies released after 1990) as well as recommendations blended with text-search scores.

5. PERFORMANCE EXPERIMENTS

This section presents our performance experiments results for our benchmark tasks tested on three recommender architectures presented in Section 4. We first describe our experimental environment. We then present performance results along with discussion for each benchmark task.

As RecBench aims to measure the relative performance of different system architectures, we selected a single algorithm (item-based CF) which is widely used in both academic and commercial recommender systems (based on our experience and conversations with practitioners) and varied only the system architecture and implementation platform. Our tasks are in no way limited to item-based CF, as they can apply to other recommendation techniques; we consider evaluating performance of different recommendation techniques on varying architectures important future work.



(a) MovieLens Data (b) Netflix Data

Figure 3: Initialization task.

5.1 Experimental Environment

Data sets. The data sets we employ in our experiments come from two real movie recommendation applications. (1) The MovieLens [23] data set consisting of 10M movie ratings and (2) The data set from the Netflix Challenge [26] consisting of 100M movie ratings. Appendix B provides detailed statistics and schemas for both data sets.

Implementations. Both DBMS-based approaches were implemented in PostgreSQL 8.4, while MultiLens was built and run using Java 1.6.0. Appendix B provides in-depth implementation details for each architecture.

System details. All experiments were done on a 4-way 3.0GHz Intel Xeon system with 48GB of RAM and a 300GB SCSI disk running Ubuntu 8.04. Our data was stored on a 236GB ext3 file system. Our performance metric is elapsed wall clock time necessary to perform each benchmark task. We perform three runs of each task. The times reported for each task are the average for a series of runs over a randomly selected set of users and/or items from the data; we describe the details of user/item selection within the writeup for each task.

5.2 Benchmark Experiments

This section provides the results of our experimental evaluation for our three architectures implementing item-based collaborative filtering [29]. We use the following parameter values for item-based collaborative filtering, which have been experimentally shown to produce quality recommendations [29]. (1) *Similarity list size*, the amount of similar items stored for each item in the model, is set to 75. (2) *Neighborhood size*, the number of similar items used in calculating a prediction (Equation 2), is set to 30.

The workload we use consists of the benchmark tasks described in Section 3. For each task, we briefly describe how it is implemented in each architecture. We then discuss and compare the results for each architecture. For all tasks except the first and final tasks, we omit the results of the *RecStore MA* approach, as its performance is exactly the same as the unmodified DBMS approach since both implementations perform query processing over a fully materialized model table.

5.2.1 Task 1: Initialization

This task represents the necessary initialization process to get the recommender system running. For item-based collaborative filtering, initialization involves building the item-based model.

SQL Commands: The DBMS implementation generates the initial item-based cosine model using the SQL given in Figure 2a. Meanwhile, both *RecStore* approaches build the model by incrementally updating their intermediate statistical representation of the model. For the *RecStore MA* approach, the materialized model is also incrementally updated using the intermediate statistics.

Hand-built implementation: MultiLens builds an item-based cosine model using a parallelized algorithm that builds the similarity matrix in row blocks, allowing multiple blocks to be computed

in parallel. For each row (an item), it retrieves all users who rated that item and builds the similarity list from the other items they have rated. The model builder uses three compute threads, leaving a processor free for the database to serve ratings requests. The JVM is limited to 32GB of heap size; this is sufficient to build the Netflix model.

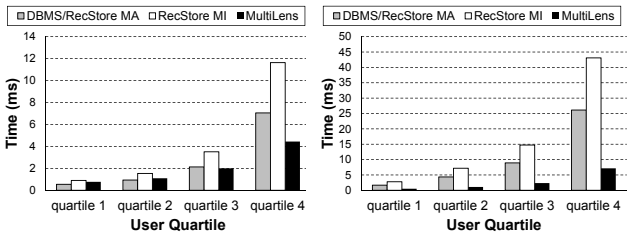
Results and Discussion: The results for the initialization task for the MovieLens and Netflix data sets are given in Figures 3(a) and 3(b), respectively. The hand-built architecture clearly outperforms the DBMS-based approach. For the MultiLens data, the hand-built approach is 71 times faster than the DBMS, representing almost two orders of magnitude. For the Netflix data, the hand-built approach is 4.5 times faster than the DBMS. Meanwhile, both *RecStore* approaches improve upon the unmodified DBMS. The intermediate statistics maintained by *RecStore* help similarity score generation immensely. The *RecStore MA* strategy is less efficient than the *RecStore MI* strategy due to the extra step in updating the materialized model.

The large disparity between the unmodified DBMS and MultiLens architectures reveals the databases drawback in performing matrix/array computations. For this task, building the item-based model essentially requires a series of vector distance computations over all items in the user rating space (Equation 1). Databases are ill-equipped to perform matrix/array computations efficiently. As affirmed by [30], databases and the relational model are an unnatural fit for applications requiring a matrix or array data model. The only way to perform this task using SQL is to perform a self-join involving both an anti and equi-join over the massive ratings table, followed by a group-by aggregation to perform the cosine distance computation, as depicted in Figure 2a. Of course, this task could be implemented as a database user-defined function (UDF), providing more control over the model building algorithm. However, our objective for these experiments is to adhere strictly to an SQL-based implementation platform for the DBMS implementation. On the other hand, MultiLens is well-equipped to build the item-based model for two reasons: (1) The model-build algorithm is tuned specifically for vector distance computations, and (2) The algorithm is built to compute different sections of the item-based model in parallel.

An interesting factor in this result, however, is the relative performance degradation of MultiLens (i.e., 71 to 18 times improvement) as the rating data set size increases by an order of magnitude. Examination of the DBMS query plan for building the model indicates that it is primarily performing merge joins and sorts over the ratings table. Our benchmark results suggest that while the DBMS model build is using an algorithm asymptotically superior to the MultiLens algorithm. For the database the build is O(R lg R) compared to O(R^2/U) for MultiLens, where R, I, and U are the number of ratings, items, and users, respectively. The DBMS performance constants render it slower than the hand-built recommender in practice. This asymptotic benefit, however, could account for the dramatic change in relative performance between the MovieLens and NetFlix data sets. While we are unsure what is causing this effect, one possible explanation is that the simple caching MultiLens performs in its database access layer is not scaling as well as the database to Netflix-scale loads.

5.2.2 Task 2: Pure Recommend

This task assumes we have a user U logging into our movie recommendation system to find movies to add to her rental list. Upon logging in, U sees on her homepage a set of N recommendations. The goal of this task is to perform a straightforward top-N recommendation for U.



(a) MovieLens Data (b) Netflix Data

Figure 4: Pure recommend results.

The computational overhead for this task is correlated with the number of items a user has rated (see Section 2.2.2). Thus, we select users for this experiment as follows. We first break the users into quartiles based on the number of movies each user has rated. We then randomly choose k users from each quartile, where k equals approximately 2.5% of all users in the data. The times reported in this experiment are an average over each user quartile. We omit the results for the *RecStore MA* strategy, as these results are the same as the unmodified *DBMS* approach (i.e., the DBMS performs query processing over a fully materialized model table).

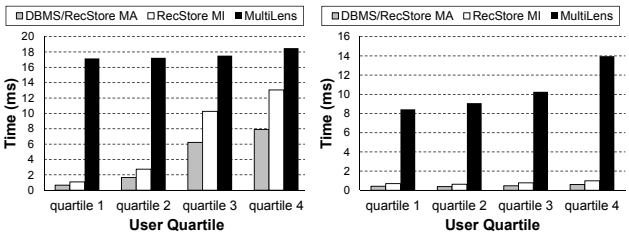
SQL Commands: Both DBMS implementations generate recommendations using the SQL given in Figure 2b, where the user rating table has been pre-computed. For the *RecStore MI* approach, model values must be generated at run-time using the model's intermediate statistical representation.

Hand-built implementation: MultiLens performs the recommendation generation steps described in Section 2.2.2, using its custom matrix library to store the model. MultiLens computes all recommendations for a user; picking only the top N merely truncates this list and does not save computation time.

Results and Discussion: Figures 4(a) and 4(b) give the results for this pure recommend task for the MovieLens and Netflix data sets, respectively. As expected, recommendation efficiency is positively correlated to the size of the quartile, as larger quartiles require processing of more user-rated items to generate recommendations.

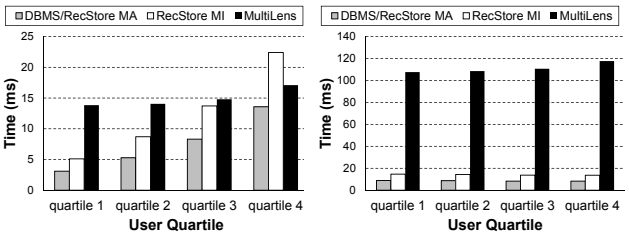
The hand-built architecture is more efficient than all DBMS-based approaches across the board. There are two main reasons for the performance difference between the DBMS and MultiLens. First, and most prominent, is that the DBMS-based execution strategy that produces recommendations is simply less efficient than the hand-built approach. For this task, PostgreSQL chooses an optimal index-only query plan to perform selection over the Ratings and model tables. However, the DBMS is forced to join the results of both operations and then perform a group-by to generate prediction scores for the recommendation candidate items. MultiLens, on the other hand, performs prediction much more efficiently due to its custom in-memory storage of the recommender model. Model reduction is performed quickly, while similarity score computation is a streamlined weighted sum operation using two arrays: one storing the item similarity list and the storing the user's previous ratings for the item.

The second reason for the better relative performance of MultiLens is overhead in data size. For the DBMS, the interface between operators executing the query plan is the tuple. These tuples require small, but non-trivial, book-keeping overhead. When matched against the "lean and mean" MultiLens implementation that stores primitive float entries in its model, the processing time required to handle tuple overhead becomes a factor.



(a) MovieLens Data (b) Netflix Data

Figure 5: Filtered recommend results.



(a) MovieLens Data (b) Netflix Data

Figure 6: Blend recommend results.

5.2.3 Task 3: Filtered Recommend

Assume now that our system user U was not inspired by any of the recommendations produced by the previous recommend task (Section 5.2.2). Upon further thought, user U decides that she only wants recommendations for recent comedies. This prompts the user to tell the system to only recommend comedy movies released after 1990. This *filter task* is so named as it requires the system to produce the top-N recommendations results for only items filtered by boolean constraints given by the user.

SQL Commands: This task requires a slight modification to the SQL used in the pure recommend task (Section 5.2.2), where we must add selection constraints over the *movies* relation to find comedies released after 1990, and join these results with the model relation to ensure only these movies are returned as recommendations. Appendix B provides the SQL for this task.

Hand-built implementation: MultiLens supports filtering recommendations with a client-provided candidate list. The operation is the same as for the pure recommend task, except that the similarity lists for the user's rated items are pruned to only contain items matched by the filter. We build the candidate list with a database query that generates a list of movie IDs using the same movie selection constraints as the SQL for the DBMS-based approach and provide this list to MultiLens.

Results and Discussion: Figure 5 provides the results for the filtered recommend task for both the MovieLens and Netflix data. We report the average time to produce a single recommendation for each user quartile, and again see a correlation between larger quartiles and higher processing time.

The DBMS-based architecture exhibits superior performance to the hand-built architecture. The reason for this relative performance is that the DBMS-based approach can *integrate* the filter as selection in its query execution plan, while the hand-built approach cannot. The database is able to take advantage of the given constraints by performing selection over the *movies* relation, and pipelining these results into a join against the model table. Since the selection factor is high, this operation "pushed down" in the query plan, and effectively limits the amount of candidate items processed by the rest of the query. Meanwhile, MultiLens executes this task in two parts. It first asks the database for the movies matching the constraints, which involves both query processing and data transfer overhead. It then performs recommendation generation and intersects these two answer sets.

Comparing the DBMS-based approaches, we see that the *RecStore MI* approach is less efficient than the unmodified DBMS and *RecStore MA* approaches, as was also observed in the *pure recommend* task. As we will see throughout the rest of the experiments, the *RecStore MI* is always outperformed in *query processing* by the unmodified DBMS and *RecStore MA* approaches. This is because *RecStore MI* must always calculate model similarity scores *on demand* from its intermediate statistics, while the other approaches keep the model scores materialized.

5.2.4 Task 4: Blended Recommend

Our system user U now realizes that in addition to recent comedies, she also wants to watch a sci-fi movie. Instead of using boolean constraints, however, she wants to receive recommendations based on a free text search for "Alien".

The difference between this task and the filtered recommend task (Section 5.2.3) is that text search is fuzzy, producing a similarity score representing the relevance of the text data to the query. For this task, the text search score is blended with the recommendation score to produce a final aggregate score. The movies returned to the user are the Top-N recommendations with the highest blended score. The blend function we use is the average of both scores. In practice, the choice of the blend function makes little performance difference; we could have used other similar functions without affecting performance results.

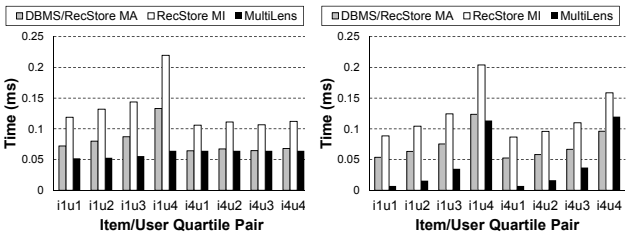
Our text data is a set of free-text tags concatenated with the "director" and "starring" attributes of the movies data. For the Netflix data, the text search is performed over the title attribute, as this is the only text data available. Details of our data sets are covered in Appendix B.

SQL Commands: The SQL commands for this task involves a join on the movie id attribute between the result of two subqueries: one for text search and the other to produce recommendation candidates. The scores from these joined results are then blended (i.e., averaged in this case). Appendix B provides the SQL for this task.

Hand-built implementation: MultiLens queries PostgreSQL to produce text-search scores using the same SQL-based text-search constraints used in the DBMS-based approach. It then generates a recommendation list as in the pure recommend task. It then walks through the results list and computes the final N recommendations using the average of the recommendation score and text search score.

Results and Discussion: The results for this task are given in Figure 6 for the MovieLens and Netflix data sets, again broken down by user quartile. The unmodified DBMS and *RecStore MA* approaches exhibit superior performance over the hand-built approach. Again, text search *integration* in query execution is the key to the efficiency of the DBMS-based architecture.

For the DBMS-based approach, the text-search query is literally "built into" its query execution, and the DBMS takes full advantage of this fact. In PosgreSQL, as with most databases, text-search results with a score of 0 (i.e., do not match the query) are not returned. Thus, for the DBMS-based approach, the text-search subquery acts as a reduction factor in the join against the subquery producing the recommendation prediction scores, meaning the amount of data actually used in the averaging (i.e., blend) computation by the database remains small. On the other hand, the MultiLens architecture must wait for the database to both perform text-search and send the answer back before performing the blend computation. This overhead is the main reason for the inferior performance of MultiLens.



(a) MovieLens Data (b) Netflix Data

Figure 7: Item prediction results.

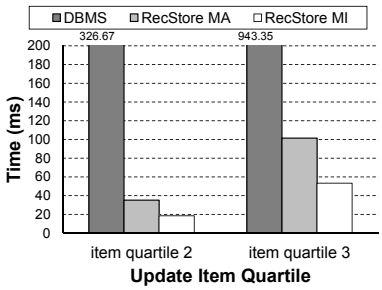


Figure 8: New item update results.

5.2.5 Task 5: Item Prediction

After being presented with a list of recommendations, our system user U is ready to start looking into the details of specific movies. This task simulates U navigating to a specific item I to view movie details. The job of the recommender system is to tell the U how much she will like the movie by predicting her rating for I (using Equation 2).

For this task, we created item quartiles based on rating count in the same manner as the user quartiles (Section 5.2.2). The higher quartiles imply more "popular" (i.e., highly rated) items. We ran this experiment performing prediction for each user quartile against each item quartile, averaging each user-item quartile pair. We report results for each user quartile matched against the first and last item quartiles, as these results represent the two extremes in terms of computational complexity for the prediction task.

SQL Commands: The SQL commands for this task is similar to the SQL used in the pure recommend task (Section 5.2.2), except we must only perform rating prediction for a single target item. Appendix B provides the SQL for this task.

Hand-built implementation: MultiLens computes a prediction by walking over U's rating list and computing the weighted average using the items U has rated that have item I in their similarity lists.

Results and Discussion: Figure 4(a) and 7(b) give the results for the predict task run against the MovieLens and Netflix data, respectively. For the Netflix data, the hand-built approach is more efficient at prediction for most user-item quartile combinations. There is a difference, however, between the runs that match the user quartiles against the smallest item quartile i1, and the runs that match the user quartiles against the largest item quartile i4. In the latter set of runs, the DBMS-based approach exhibits better performance than the hand-built approach for the final run matching the largest user quartile u4 against the largest item quartile i4. This is because, for the larger item quartiles with popular items, it is more likely that a user will have already rated a target item I. Thus, the DBMS-based approach will not execute the majority of its query due to the NOT EXIST clause returning false in the prediction SQL query. MultiLens performs prediction regardless of the user having already rated the target item. However, MultiLens can easily be changed to shortcut its predictions upon finding I is already rated, and we expect that MultiLens will be more efficient than the DBMS when this is the case. Experiments for the MovieLens data exhibit similar results, with MultiLens outperforming the DBMS-based approach for all user quartiles matched with item quartile $i1$, while both approaches exhibit similar performance for item quartile $i4$.

5.2.6 Task 6: Update

This task involves introducing a new item to the recommender system. The goal is to quickly inject the item into circulation as a recommendation candidate. For item-based collaborative filtering, we achieve this goal by incorporating user ratings for this new item into the model as soon as possible. To simulate updates, this task first builds a model, holding out all ratings for 20 *items* from each of the second and third item-quartiles computed for the *item prediction* task. We then add the ratings for these missing items into the system one-by-one. Results are reported per quartile as the average time to incorporate each item into the recommender model.

SQL Commands: For the unmodified DBMS approach, we implement a trigger to update the stored recommender model according to the SQL given in Figure 2a. Meanwhile, the *RecStore* approach uses customized code to incrementally update the recommender model. Upon receiving an update, both the *RecStore MI* and *RecStore MA* approaches update stored intermediate statistics for the item-based cosine model. The *RecStore MA* approach then uses the update to the intermediate statistics to incrementally update its *model store*.

Hand-built implementation: MultiLens currently does not support introduction of new items to an existing recommender model. Therefore, a complete model rebuild is necessary to bootstrap the new item. After the item is added, the similarities can be incrementally updated, but this will be dominated by the rebuild time.

Results and Discussion: Figure 8 gives the results for the update task for the MultiLens data set (relative performance is similar for the Netflix data). The unmodified DBMS approach incurs the most overhead in updating the model. Upon receiving an update, this approach must recalculate all affected model entries *from scratch* using the rating data. Both the *Recstore MA* and *RecStore MI* approaches improve the update performance of the DBMS by at least an order of magnitude. The intermediate statistics maintained by *RecStore* clearly help in incrementally updating the model. Comparatively, *RecStore MI* is more efficient than *RecStore MA* due to not having to perform the extra step in updating the model store. However, as we observed in previous experiments, *RecStore MI* is less efficient than *RecStore MA* in query processing. MultiLens currently does not support incremental updates to the item-based recommender model when an item does not previously exist in the system. Since MultiLens must completely rebuild the model to incorporate new items, its performance on this task will be equivalent that of the *initialize* task. Recomputing the model incorporates all new items, however, so its cost can be amortized over all items added to the inventory since the last build.

6. DISCUSSION

Our experimental results reveal an interesting trade-off. The "hand-built" recommender architecture is more efficient at straightforward recommendation tasks (i.e., pure recommend and single-item prediction). For these tasks, the hand-built architecture had the obvious advantage of an implementation hard-wired to the specific recommendation technique, e.g., optimized in-memory storage, state-of-the-art algorithmic implementation. Meanwhile, the DBMS-based architecture is at a disadvantage for these tasks due to its general-purpose nature, e.g., book-keeping overhead for tuples, query execution using generic operators.

The DBMS recommender architecture, however, outperforms the hand-built approach at more complex recommender tasks involving filtered recommendations and blending text-search scores with recommendation scores. In our experiments, this performance benefit was due to the DBMS supporting the advanced search features (e.g., text search, selection predicates) as built-in operators, thus being able to self-optimize for the advanced recommendation queries. Meanwhile, the hand-built recommender exists at the middleware layer and requires the overhead of communicating with a third-party application (e.g., PostgreSQL through JDBC) in order to perform the advanced search and filter operations. Building and maintaining the recommender model poses a substantial challenge for DBMS-based recommenders; this problem can be greatly reduced by extending the DBMS with direct support for recommendation tasks (e.g., update support from *RecStore*). Appendix A provides further discussion comparing these architectures.

Our results also suggest that both architectures stand for further improvement. An interesting direction for DBMS-based recommenders could explore the creation of a "recommend" operator to improve performance of simple recommendation tasks. Semantically, this operator could take as input a model relation, and produce ecommendations based on a given recommendation method. In fact, work in this area has already begun with the FlexRecs framework [18] that proposes three operators capable of evaluating myriad different recommendation techniques. In general, the operator approach has proven beneficial for other preference methods. For instance, the skyline [8] and top-k [11] operators outperform their respective SQL implementations by orders of magnitude.

Meanwhile, hand-built architectures implemented at the middleware layer fall short in more complex recommendation tasks since they rely on an outside systems (e.g., database, text-search engines) to help produce results. The data transfer and missed optimization opportunities for such tasks introduces non-trivial overhead. Thus, an interesting research direction could explore pushing frequently-used filtering tasks into the hand-built recommender architecture.

In addition to improving each individual architecture, another interesting line of work is to build a performance-optimized hybrid recommender architecture. This work would combine principals from both hand-built and DBMS-based recommenders to investigate the optimal level (e.g., middleware, internal DBMS) to place certain architectural components of the recommender system (e.g., model building, recommendation generation, filtering). We equate such an approach to past work that explored building hybrid architectures that coupled a DBMS with both text-search [2, 17] and XML [4, 6, 13] functionality.

The benchmarks in this paper open up an interesting line of future work evaluating recommender system performance. We highlight two interesting directions below that can make use of our benchmark tasks. Though not an exhaustive list of directions, these proposals elucidate the interesting research space in evaluating recommender architectures. (1) *Multi-user experimental workloads*. Our experiments only evaluated single-user queries as a first step in understanding performance characteristics of different recommender architectures. An interesting line of future work would be to evaluate multi-user workloads, e.g., measuring throughput as the level of concurrency changes. We equate such an extension to prior work in single-user database benchmarks [5] extended to multi-user environments [7]. (2) *Different recommendation techniques*. In this work, we chose to experimentally evaluate recommender system architectures using a single recommendation technique (item-based collaborative filtering). An interesting line of future experimental work could test the performance of different recommendation techniques on varying system architectures.

7. CONCLUSION

This paper provides the first benchmark for evaluating performance of recommender system architectures. Our benchmark tasks model the needs of a typical recommender-powered e-commerce scenario. In our experiments, we evaluated the following three recommender system architectures using our benchmark tasks with the MovieLens and Netflix data sets: (1) an unmodified DBMS, (2) *RecStore*, a DBMS customized to efficiently handle recommender model updates, and (3) *MultiLens*, a hand-built recommender system. Our experimental results reveal that "hand-built" systems exhibit superior performance in model-building and pure recommendation tasks, while the DBMS-based approaches are superior at more complex recommendation tasks such providing filtered recommendations. In light of our benchmarks and results, we highlight interesting research directions at the intersection of recommender systems and data management.

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):734–749, 2005.

[2] S. Agrawal, S. Chaudhuri, and G. Das. DBXplorer: A System for Keyword-Based Search Over Relational Databases. In *ICDE*, pages 5–16, 2002.

[3] Apache Mahout: http://mahout.apache.org/.

[4] K. Beyer et al. System RX: One Part Relational, One Part XML. In *SIGMOD*, pages 347–358, 2005.

[5] D. Bitton, D. J. Dewitt, and C. Turbyfill. Benchmarking Database Systems: A Systematic Approach. In *VLDB*, pages 8–19, 1983.

[6] P. Bohannon et al. LegoDB: Customizing Relational Storage for XML Documents. In *VLDB*, pages 1091–1094, 2002.

[7] H. Boral and D. J. DeWitt. A methodology for database system performance evaluation. In *SIGMOD*, pages 176–185, 1984.

[8] S. Börzsönyi et al. The Skyline Operator. In *ICDE*, pages 421–430, 2001.

[9] J. S. Breese, D. Heckerman, and C. Kadie. Empirical Analysis of Predictive Algorithms for Collaborative Filtering. In *UAI*, pages 43–52, 1998.

[10] R. Burke. Hybrid Recommender Systems: Survey and Experiments. *User Modeling and User-Adapted Interaction*, 12(4):331–370, 2002.

[11] M. J. Carey and D. Kossmann. On Saying "Enough Already!" in SQL. In *SIGMOD*, pages 219–230, 1997.

[12] A. Das, M. Datar, A. Garg, and S. Rajaram. Google News Personalization: Scalable Online Collaborative Filtering. In *WWW*, pages 271–280, 2007.

[13] D. DeHaan et al. A Comprehensive XQuery to SQL Translation using Dynamic Interval Encoding. In *SIGMOD*, pages 623–634, 2003.

[14] The Facebook Blog: "I like this": http://blog.facebook.com/blog.php?post=53024537130.

[15] Hadoop HBase: http://hadoop.apache.org/hbase/.

[16] J. L. Herlocker, J. A. Konstan, L. G. Terveen, and J. T. Riedl. Evaluating Collaborative Filtering Recommender Systems. *TOIS*, 22(1):5–53, 2004.

[17] V. Hristidis and Y. Papakonstantinou. DISCOVER: Keyword Search in Relational Databases. In *VLDB*, pages 670–681, 2002.

[18] G. Koutrika, B. Bercovitz, and H. Garcia-Molina. FlexRecs: Expressing and Combining Flexible Recommendations. In *SIGMOD*, pages 745–758, 2009.

[19] J. J. Levandoski, M. F. Mokbel, M. Sarwat, and M. D. Ekstrand. RecStore: DBMS Engine Support for Efficient Online Recommender Systems. Technical Report UM-CS-TR, University of Minnesota, 2010.

[20] G. Linden et al. Amazon.com Recommendations: Item-to-Item Collaborative Filtering. *IEEE Internet Computing*, 7(1):76–80, 2003.

[21] B. N. Miller. *Toward a personal recommender system*. PhD thesis, University of Minnesota, 2003.

[22] B. N. Miller et al. MovieLens Unplugged: Experiences with an Occasionally Connected Recommender System. In *IUI*, pages 263–266, 2002.

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

[24] MultiLens Recommender System: http://www.cs.luther.edu/ bmiller/dynahome.php?page=multilens.

[25] Netflix: http://www.netflix.com.

[26] Netflix Prize Dataset: http://www.netflixprize.com.

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

[28] P. Resnick et al. GroupLens: An Open Architecture for Collaborative Filtering of Netnews. In *CSWC*, pages 175–186, 1994.

[29] B. Sarwar, G. Karypis, J. Konstan, and J. Riedl. Item-Based Collaborative Filtering Recommendation Algorithms. In *WWW*, pages 285–295, 2001.

[30] M. Stonebraker et al. Requirements for Science Data Bases and SciDB. In *CIDR*, 2009.

APPENDIX

This appendix supplements the content of the main body of the paper as follows. Appendix A provides a comparison of the recommender system architectures tested in this paper. Appendix B provides further implementation details for our experiments.

A. ARCHITECTURAL COMPARISONS

In this section, we compare and contrast the recommender system architectures tested in this paper. We aim to flesh out the advantages and disadvantages of architectural features that cannot be measured by experimental numbers.

A.1 Flexibility

Flexibility refers to the ability of the system to adapt to non-trivial changes in the requirements of a recommender application. For example, a book/music website may decide to use a different recommendation method, or make their current method more interactive by allowing users to specify constraints on which books or music they receive as recommendations. Having a more flexible architecture implies such changes are easier to implement.

Recent work has shown that the DBMS approach is at an advantage to the hand-build approach in terms of flexibility [8]. Due to its declarative interface and general query execution infrastructure, adding constraints or changing the recommendation method is relatively straightforward. For example, using our SQL example in Figure 2b, filtering item-based recommendations requires simply adding constraints to the WHERE clause of the SQL query. Examples of real-world constraints are "only recommend movies from a given genre" or "for items with same prediction scores, recommend those with higher profit margins". Furthermore, changing the recommendation method from item-based to user-based collaborative filtering [28] requires us to only reverse the item Ids (itmId) and user Ids (usrId) in Figure 2a in order to find neighborhoods of similar users. These changes in the hand-built approach would require non-trivial system modifications.

Hand-built recommender systems are usually "lean-and mean", using only the ids of the user-item matrix for fast in-memory recommendation computation. These systems usually do not store extra metadata on items or users (e.g., titles, names), thus adding constraints would require both importing the metadata as well as programming custom filters for each necessary constraint. Also, creating a new recommendation method within the hand-built architecture will usually require re-writing the recommendation logic from scratch.

A.2 Query Processing

The difference in query processing strategies between each recommender architecture is vast. The DBMS, being a general data management platform, can take advantage of decades of research on query processing and optimization. The DBMS composes its execution strategy using simple relational operators. Multiple execution strategies are possible, thus the DBMS is capable of configuring an optimal (or close-to-optimal) query execution plan based on available indexes and statistical summaries of attributes in the underlying relations. The DBMS is also scalable for out-of-memory operations, and has been a proven solution for scalable processing of large data sets. However, if most query processing is memory-bound, the DBMS is at a disadvantage as it organizes data into pages and tuples, which both require book-keeping overhead not present in a hand-built application.

Meanwhile, hand-built recommender systems are hard-wired to a single execution strategy. The hand-built architecture takes full advantage of domain-specific optimizations, such as custom in-memory storage techniques that work in concert with specialized recommendation generation algorithms. Furthermore, these custom storage techniques and algorithms can easily port to more exotic architectures such as map-reduce nodes to increase scalability; a task more difficult for the well-entrenched DBMS architecture. The hand-built approach is also harder to tune for data sets which do not fit in main memory, as it must either re-implement disk-based algorithms or rely on the operating system's virtual memory paging algorithm. In some cases this pitfall may be aided by data access facilities provided by a computational framework such as Hadoop's Hbase [15].

A.3 Architectural Integration

DBMS-based and hand-built recommender implementations integrate differently with varying application infrastructures. The availability of both strategies is itself beneficial, as it allows system architects to implement the recommendation logic where it makes most sense for their application. The DBMS architecture places the recommendation logic in the data storage layer, while a hand-built recommender resides at the middleware layer. Developers and architects can therefore place the recommender in various layers based on system load, hardware capabilities, or other application demands and constraints.

B. EXPERIMENT DETAILS

In this section, we provide details of the data sets used in our experiments. We also discuss the implementation details for each architecture we evaluated in our experiments.

B.1 Data Sets

This section provides details of the data sets used in the experimental study presented in Section 5

MovieLens. The MovieLens data consists of three parts:

- *Movies*(*movieId* [integer], *title* [text], *genres* [text], *releaseDate* [date], *directedBy* [text], *starring* [text]): contains data on 10681 movies. The *genres* field is a comma-delimited string containing standard genre descriptions (e.g., Comedy, Drama), while the *starring* field contains a comma-delimited list of actor names starring in the film.
- *Ratings*(*userId* [int], *movieId* [int], *rating* [int], *timestamp* [date/time]): contains 10M ratings for 69878 users over 10681 movies. Each entry is marked with a timestamp representing when the user rated the movie.
- *Tags*(*userId* [int], *movieId* [int], *tag* [text], *timestamp* [date/time]): contains 10K movie tags. The *tag* field contains a single user-defined string for a movie, where a user is allowed to tag a movie multiple times.

Netflix. We also use the data set from the Netflix Challenge [26] consisting of 100M movie ratings. The Netflix data consists of two parts:

- *Movies*(*movieId* [integer], *title* [text], *releaseYear* [date]): contains data on 17,770 movies from the Netflix rental site.
- *Ratings*(*userId* [int], *movieId* [int], *rating* [int], *timestamp* [date/time]): contains 100M ratings for 480,189 users over 17,770 movies with a schema similar to the MovieLens data.

B.2 Architecture Implementation Details

This section discusses the implementation details for each architecture evaluated in our experiments. We first discuss the MultiLens hand-built implementation. We then discuss the unmodified DBMS and RecStore implementations, respectively.

B.2.1 MultiLens Implementation

We built and ran the MultiLens implementation in Java 1.6.0p14 for x86_64. The model builder loads data on-demand from the rating and metadata tables in the same PostgreSQL database as we used for the DBMS tests. The Java-PostgreSQL JDBC interface implements some caching to reduce database re-fetches, similar to that done by any standard JDBC-based application.

B.2.2 DBMS Implementation

We implemented the database approach using the PostgreSQL 8.4 open-source DBMS [27]. Data is stored in relations according to the schemas discussed previously in Section B.1. For the *Movies* relation, a B+ tree index is built over the attributes *movieId*. The *Ratings* contains two B+ tree indices: one clustered on (*movieId*,*rating*) and one unclustered on (*userId*,*rating*). For the MovieLens data we built two B+ tree indices: one clustered over (*movieId*,*tags*) and one unclustered over (*userId*,*tags*). Postgres shared memory is set to 2GB for all tasks, and *fsync* is disabled to avoid forward syncing for write-ahead logging. We used the default settings for all other Postgres tuning parameters.

In addition to the SQL provided for benchmark tasks 1 and 2 in Figure 2, we provide below the SQL used for benchmark tasks 3 through 5:

Task 3 - Filtered Recommend:

```
SELECT M.itm as Candidate Item,
       SUM(R.sim * U.rating)/ SUM(R.sim) as Prediction
FROM  Model M, usrXMovies U, movies MV
WHERE  MV.releaseDate > 1990 AND
       MV.genre SIMILAR TO '%Comedy%' AND
       M.itm = MV.mid AND
       M.rel_itm = U.itemId
GROUP BY M.itm ORDER BY Prediction DESC
LIMIT N;
```

Task 4 - Blended Recommend:

```
SELECT T.mid, (T.rank + R.rank)/2 as combRank
FROM (SELECT mid, max(ts_rank_cd(to_tsvector(tag), query))
      as rank
      FROM tags, to_tsquery('Alien') query
      WHERE mid NOT IN (select itemId from usrXMovies) AND
         query @@ to_tsvector(tag)
      GROUP BY mid) T,
     (SELECT M.itm as mid,
      SUM(R.sim * U.rating)/SUM(R.sim)
      FROM Model M, userXMovies U
      WHERE M.rel_itm = U.itmId AND
      M.itm NOT IN (select itmId FROM usrXMovies)
      GROUP BY M.itm) R
WHERE T.mid = R.mid
ORDER BY combRank DESC
LIMIT 10;
```

Task 5 - Item Prediction: While similar to the SQL for the pure recommend task (Section 5.2.2), this query only performs a rating prediction for a single target item. We also add a NOT EXISTS clause in the first query that creates a NULL user-movies table if we find that our user U has already rated a target item I. This ensures that the query will not compute unnecessary predictions in the second SQL query.

```
CREATE TEMP TABLE usrXMovies AS
SELECT R.mid as itemId, R.rating as rating
FROM   Ratings R
WHERE  R.userId = U AND
       NOT EXISTS (SELECT mid FROM Ratings
                   WHERE userId=U AND movieId=I);

SELECT M.itm as Candidate Item,
       SUM(R.sim * U.rating)/ SUM(R.sim) as Prediction
FROM   Model M, usrXMovies U, movies MV
WHERE  M.rel_itm = U.itemId AND
       M.itm = I
GROUP BY M.itm ORDER BY Prediction DESC
LIMIT N;
```

B.2.3 RecStore Implementation

The *RecStore* prototype is implemented between the storage engine and query processor of the PostgreSQL database. As mentioned in Section 4.2, our experiments test two variants of the *RecStore* framework, the *maintain-all* (abbr. *RecStore MA*) and *maintain-intermediate* (abbr. *RecStore MI*) strategy. For both strategies, an *intermediate store* is maintained that contains sufficient statistics to help incrementally update the recommender model. These statistics represent a "deconstructed" cosine score (Equation 1), whereby vector lengths and dot-products are stored as separate pieces. The *RecStore MA* strategy also maintains a *model store*, which is a materialized table storing the item-based model.