RECATHON: A Middleware for Context-Aware Recommendation in Database Systems

[1]Mohamed Sarwat, [2]James L. Avery, [3]Mohamed F. Mokbel

[1]*School of Computing, Informatics, and Decision Systems Engineering, Arizona State University, Tempe, AZ*
[2]*IBM Mobile Innovation Lab, Austin, 11501 Burnet Road Austin, TX*
[3]*Department of Computer Science and Engineering, University of Minnesota, Minneapolis, MN*
[1]msarwat@asu.edu, [2]jlavery@us.ibm.com, [3]mokbel@cs.umn.edu

***Abstract*—This paper presents RECATHON; a context-aware recommender system built entirely inside a database system. Unlike traditional recommender systems that are context-free where they support the general query of *Recommend movies for a certain user*, RECATHON users can request recommendations based on their age, location, gender, or any other contextual/demographical/preferential user attribute. A main challenge of supporting such kind of recommenders is the difficulty of deciding what attributes to build recommenders on. RECATHON addresses this challenge as it supports building recommenders in database systems in an analogous way to building index structures. Users can decide to create recommenders on selected attributes, e.g., age and/or gender, and then entertain efficient support of multidimensional recommenders on the selected attributes. RECATHON employs a multi-dimensional index structure for each built recommender that can be accessed using novel query execution algorithms to support efficient retrieval for recommender queries. Experimental results based on an actual prototype of RECATHON, built inside PostgreSQL, using real MovieLens and Foursquare data show that RECATHON exhibits real time performance for large-scale multidimensional recommendation.**

I. INTRODUCTION

Recently, recommender systems have grabbed researchers' attention in both industry [2], [7], [9], [19] and academia [14], [13], [24], [28], [29]. The main goal of a recommender system is to suggest new and interesting items to users from a large pool of items. Recommender systems are implicitly employed on a daily basis to recommend movies (e.g., Netflix), books/products (e.g., Amazon), friends (e.g., Facebook), and news articles (e.g., Google News). A recommender system exploits the users' opinions (e.g., movie ratings) and/or purchasing (e.g., watching, reading) history in order to extract a set of interesting items for each user. In technical terms, a recommendation algorithm takes as input a set of users U, items I, and ratings (history of users' opinions) R and estimates a utility function $RecScore(u, i)$ that predicts how much a certain user $u \in U$ would like an item $i \in I$ such that i has not been already seen (or watched, consumed, etc) by u [4]. To estimate such a utility function, many recommendation algorithms have been proposed in the literature [4], [11] (e.g., Collaborative Filtering).

Classical recommender systems answer the traditional *context-free recommendation* query like, *Recommend me 10 movies*, where we estimate the recommendation utility function based on the whole users' opinions history and hence generates recommendation to the querying user regardless of the user attributes (e.g., user age, job, and gender). On the other side, a multidimensional recommender would be able to estimate the recommendation utility function based on a subset of the users' opinions data that corresponds to specific attributes' range and hence would be able to support *context-aware*, *preference-aware*, or *user demographic-aware* recommendations queries such as: *Recommend me 10 movies that people in my age would like*, *Recommend me 10 restaurants that are located in Tempe, Arizona*, or *Recommend me 5 movies that people of my gender would like*. From a modeling perspective, it has been shown that such queries would exhibit higher accuracy than context-free recommendations [3], [6].

Building a multidimensional (context-aware) recommender poses the following challenges: (1) *Challenge I:* Deciding the contextual attributes to build a recommender on and the recommendation algorithm to estimate the utility function. For example, if a user has two attributes, *age* and *gender*, then there is a possibility of four recommenders, a context-free, an *age*-aware, a *gender*-aware, and an (*age*, *gender*)-aware recommender. Also, each recommender may have different versions based on the employed recommendation algorithm. (2) *Challenge II:* For a multidimensional recommender, how to efficiently store and maintain the underlying recommendation models. A basic solution would be to materialize all recommendation models that correspond to all attributes combinations. Nonetheless, this approach incurs tremendous storage and maintenance overhead. (3) *Challenge III:* How to execute context-aware recommendation queries expressed by users over a recommender. A straightforward solution implements the recommendation functionality *on-top* of a database system. However, this approach does not harness the full power of the database engine.

In this paper, we introduce RECATHON : a middleware for building context-aware recommendation applications in a database management system. RECATHON tackles *Challenge I* by providing a declarative interface for users to build custom-made multidimensional recommenders. The main idea is to deal with building recommenders inside the database engine in an analogous way to creating indexes (or views). Database users can build indexes on tables, based on the query workload. Similarly, if there is a significant number of recommendation

queries that use *age* and *salary*, a RECATHON designer might decide to build a recommender over the fields ⟨*age*, *salary*⟩. To achieve that, RECATHON extends SQL with new statements to create/drop multidimensional recommenders, namely `CREATE/DROP RECOMMENDER`. When creating a context-aware recommender, the user specifies the recommender name, the sets of users, items, and ratings that will be used to build the recommender. In addition, the user specifies the set of contextual attributes that the recommender will support and selects one of RECATHON built-in recommendation algorithms.

To address *Challenge II*, we initialize and maintain a multidimensional grid data structure for each recommender created using the `CREATE RECOMMENDER` statement, where each dimension corresponds to one of the attributes. Each grid cell includes a recommendation model that is used to efficiently generate recommendations to users. Materializing a large number of grid cells incurs more recommendation models being stored and maintained, which may preclude system scalability. To remedy this issue, RECATHON adaptively decides, based upon the query/update workload, which recommender cells to maintain in order to reduce the overall recommender storage and maintenance overheard without largely compromising the query execution performance. To query an existing multidimensional recommender (*Challenge III*), RECATHON users specify the recommender in the `FROM` clause and the user identifier as a predicate to the `WHERE` clause of a SQL query. RECATHON executes the query and produces the set of recommended items, along with their scores with respect to the querying user. To reduce query latency, RECATHON optimizes incoming recommendation queries through a set of query execution algorithms that embed the recommendation functionality within other database operations, namely, *select*, *join*, and *ranking*.

Experiments, based on actual system implementation inside PostgreSQL, using real data extracted from MovieLens [20] and Foursquare [27], show that RECATHON exhibits high performance for large-scale multidimensional recommendation. The rest of the paper is organized as follows: Preliminaries are given in Section II. Challenges I to III are addressed in Sections III to V. Experiments are presented in Section VI. Related work is outlined in Section VII. Finally, Section VIII concludes the paper.

II. PRELIMINARIES

Data Model. RECATHON assumes the following input data: (1) *Users*: a set of users and their attributes. (2) *Items*: a set of items and their attributes. (3) *Ratings*: users expressing their opinions over items. Opinions can be a numeric rating (e.g., one to five stars), or unary (e.g., Facebook "check-ins"). Also, ratings may represent purchasing behavior (e.g., Amazon). Figure 1 gives an example of movie recommendation data.

Recommendation Algorithms. Most recommendation algorithms produce recommendations in two steps, as follows:

Step I: Recommendation Model Building: This step consists of building a recommendation model *RecModel* using the input data. The format of the model depends on the underlying recommendation algorithm. For example, a recommendation model for the item-item cosine-similarity model (ItemCosCF) [4] is a similarity list of the tuples $\langle i_p, i_q, SimScore \rangle$, where $SimScore$ is the similarity score between items i_p and i_q. To compute *SimScore*(i_p, i_q), we represent each item as a vector in the user-rating space of the user/item ratings matrix. The Cosine similarity is then calculated as follows:

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

Step II: Recommendation Generation: This step utilizes the *RecModel* (e.g., items similarity list) created in *Step I* to predict a recommendation score, $RecScore(u, i)$, for each user/item pair. $RecScore(u, i)$ reflects how much each user u likes the unseen item i. The recommendation score $RecScore$ depends on the recommendation algorithm defined for the underlying recommender. For the ItemCosCF recommendation algorithm, $RecScore(u, i)$ for each item i not rated by u is calculated as follows:

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

Before this computation, we reduce each similarity list $\mathcal{L}$ to contain only items *rated* by user u. The recommendation score is the sum of $r_{u,l}$, a user u'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 between i and l.

uid	name	age	gender	city
1	Alice	20	Female	Minneapolis
2	Carol	22	Female	Minneapolis
3	Eve	18	Female	Minneapolis
4	Erin	19	Female	Minneapolis
5	Bob	34	Male	Minneapolis
6	Dan	40	Male	St. Paul
7	Frank	65	Female	Edina

(a) Users Table (*Users Data*)

iid	name
1	'The Lord of the Rings'
2	'I Love New Work'

(b) Movies Table (*Items Data*)

uid	iid	Rating
1	1	3.5
2	2	4.5
1	2	2.0
3	1	5.0
4	1	1.5
3	2	3.0
5	2	3.0
6	1	5.0
7	1	3.5
6	2	1.0
7	2	1.5
5	1	2.5

(c) Ratings Table

Fig. 1. Recommender Input Data.

III. MULTI-DIMENSIONAL RECOMMENDER

Figure 2 depicts RECATHON architecture. To support multidimensional recommenders, the first challenge (*Challenge I*) is to provide a tool to the system users to freely decide which attributes and recommendation algorithm to be used in building the recommender. RECATHON addresses this challenge by allowing users to use a SQL-like clause to define new multidimensional recommenders by specifying the recommender data and attributes (i.e., dimensions). RECATHON exposes the newly created recommender to its users as a virtual schema, where users can issue SQL queries to obtain a set of recommended items (e.g., movies) based on the specified attributes

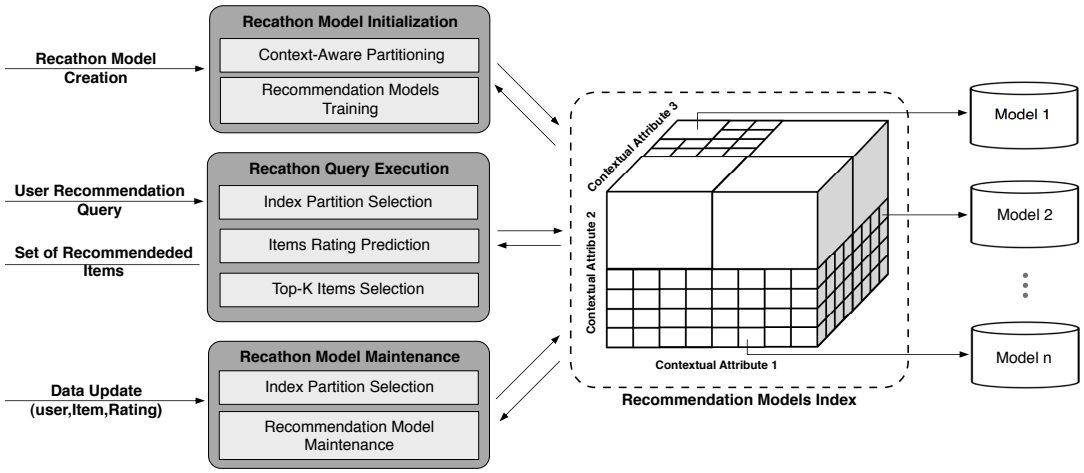


Fig. 2. Recathon Architecture.

(e.g., age and city). This section focuses on how users interact with the system. In particular, Section III-A explains the SQL clause for creating a new recommender, while Section III-B explains the SQL for querying a recommender.

A. Creating a New Recommender

To allow creating a new recommender-aware system, RECATHON employs a new SQL statement, called `CREATE RECOMMENDER`, as follows:

```
CREATE RECOMMENDER [Recommender Name]
USERS FROM [Users Table]
ITEMS FROM [Items Table]
RATINGS FROM [Ratings Table]
ATTRIBUTES [Attributes Set]
USING [Recommendation algorithm]
```

The recommender creation SQL, presented above, has the following parameters: (1) `Recommender name` is a unique name assigned to the created Recommender. (2) `Users Table`, `Items Table`, and `Ratings Table` are the names of relational tables containing the users, items, and ratings input data (e.g., the tables in Figure 1). (3) `Attributes Set` is a set of dimensions that the recommender will be built on (e.g., age). (4) `Recommendation algorithm` is the algorithm used to build the recommender. Currently, RECATHON supports three main recommendation algorithms (with their variants): (a) Item-Item Collaborative Filtering with Cosine (abbr. ItemCosCF) or Pearson Correlation (abbr. ItemPearCF) similarity functions, (b) User-User Collaborative filtering (abbr. UserCosCF / UserPearCF), and (c) Regularized Gradient Descent Singular Value Decomposition (abbr. SVD). If no recommendation algorithm is specified, RECATHON employs by default the ItemCosCF algorithm. Examples are given below:

Example 1: AgeRec: an age-aware recommender created on the input data stored in the Users, Movies, and Ratings tables of Figure 1, using the ItemCosCF recommendation algorithm:

```
CREATE RECOMMENDER AgeRec  USERS FROM Users U
ITEMS FROM Movies  RATINGS FROM Ratings
ATTRIBUTES U.age  USING ItemCosCF
```

With this SQL, a new recommender, named *AgeRec*, is added to the database system, and can be queried later to return a set of recommended movies based on the user age, e.g., *recommend me five movies that people in my age like*.

Example 2: AgeCityGenderRec: an (age, city, gender)-aware recommender created on the input in Figure 1, using the SVD recommendation algorithm:

```
CREATE RECOMMENDER AgeCityGenderRec
USERS FROM Users U  ITEMS FROM Movies
RATINGS FROM Ratings
ATTRIBUTES U.age, U.city, U.gender USING SVD
```

With this SQL clause, a new recommender, named *AgeCityGenderRec*, is added to the database system, and can be queried later to return to a querying users a set of recommended movies based on the user age, city, and gender, e.g., *recommend me five movies that people in my age, living in my city, and of my gender, would like*.

Notice that in the above two examples, if we had no `ATTRIBUTES` clause, we would create a traditional recommender that can be queried to recommend a set of movies for a certain user, regardless of the user attributes, e.g., *recommend me five movies*.

B. Querying a Recommender

Once a recommender is created using the `CREATE RECOMMENDER` statement, it is exposed to querying users as a virtual table with the schema: (`uid`,`iid`,`RecScore`), where `uid` is a user identifier, `iid` is an item identifier, and `RecScore` is the recommendation score that predicts how much the user `uid` would like the item `iid` according to the underlying recommender function, specified in the `USING` clause of the `CREATE RECOMMENDER` statement. Then, RECATHON users can issue SQL queries over the created multidimensional recommender schema as follows:

```
SELECT  [Select Clause]
FROM    [Recommender], [Tables]
WHERE   [Where Clause]
```

$Query\ 1$	**Select** * **From** AgeCityGenderRec R **Where** R.uid=2
$Query\ 2$	**Select** R.iid **From** AgeCityGenderRec R **Where** R.uid=1 **Order By** R.RecScore **Desc Limit** 10
$Query\ 3$	**Select** R.iid, R.RecScore **From** AgeRec R **Where** R.uid=1 AND R.iid IN (1,2,3,4,5)
$Query\ 4$	**Select** M.name, R.RecScore **From** AgeRec R, Movies M **Where** R.uid=1 AND M.iid = R.iid AND M.genre='Action'
$Query\ 5$	**Select** R.iid, R.RecScore **From** AgeRec R, Movies M **Where** R.uid AND R.iid = M.iid AND M.genre = 'Action' **Order By** R.RecScore **Desc Limit** 5

TABLE I
RECOMMENDATION QUERY EXAMPLES

The SELECT and WHERE clauses are typical as in any SQL query. The FROM clause may accept, in addition to relational tables, a [Recommender] schema, which is created by a CREATE RECOMMENDER statement. In the WHERE clause, the querying user may specify the *UserID*, the user identifier for whom the recommendation needs to be generated. RECATHON then returns a set of tuples $\langle ItemID, RecScore \rangle$ that represents the predicted recommendation score *RecScore* for each item *ItemID* based on the recommender specified in the FROM clause. Examples are given in table I. For instance, $Query\ 2$ returns the top-10 recommended items to user with ID 1, based on the user age, city, and gender. In this case, the query uses the AgeCityGenderRec, which was created before using a CREATE RECOMMENDER. Since this recommender was created based on the age, city, and gender attributes, it will return an answer to the user based on the values of these attributes in the USERS table for user ID 1.

IV. RECATHON INDEXING

After creating a recommender, a main challenge (*challenge II*) is how to internally represent, store, and maintain the underlying recommendation models in a scalable manner. To address this issue, we introduce the following data structures to represent user-created multidimensional recommenders: (1) We maintain one global structure, namely *RecCatalog*, for all created recommenders (section IV-A). (2) For each recommender, we maintain one multidimensional grid and we explain how we reduce both storage and maintenance overhead to achieve scalability (section IV-B).

A. Recommender Catalog

RECATHON maintains a relational table, termed *RecCatalog*, that includes metadata about all created recommenders, and is stored as part of the main database catalog that includes information about tables, index structures, etc. A row in *RecCatalog* has seven attributes: (1) *RecName*; the recommender name, (2) *Users*; the input users table, (3) *Items*; the input items table, (4) *Ratings*; the input ratings table, (5) *Attributes*; a vector where each element corresponds to an attribute in the *users* table that contributes to the recommender model, (6) *Algorithm*; the algorithm used to generate predicted scores, and (7) *RecIndex*; a pointer to the multi-dimensional grid index for this particular recommender. A new row is added/deleted to/from *RecCatalog* with each successful CREATE RECOMMENDER / DROP RECOMMENDER SQL statement.

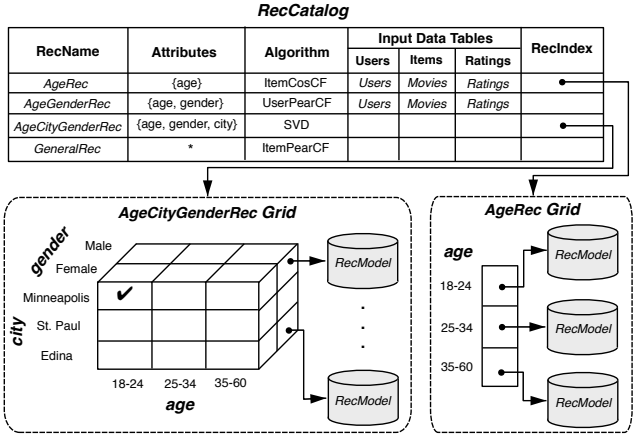


Fig. 3. Recathon data structure.

Figure 3 gives an example of *RecCatalog*, where it has four entries for four recommenders, *AgeRec*, *AgeGenderRec*, *AgeCityGenderRec*, and *GeneralRec*. With each recommender, the corresponding attributes are listed. Notice that in the case of *GeneralRec*, *Attributes* is empty, which corresponds to a general recommender system regardless of any attributes.

B. Multi-dimensional Grid

For each created recommender, RECATHON maintains a *Multi-dimensional Grid* G, where each dimension corresponds to one of the recommender attributes. A grid cell C in G represents a subdomain of the space created by the multiple attributes. The subdomain could be a certain value for categorical attributes or range of values for continuous domain attributes. For example, as *AgeRec* recommender in Figure 3 is defined based on only one attribute (*age*), its index is a one-dimensional grid based on the age attribute. As this is a continuous domain attribute, each cell represents a range of age values, i.e., [18-24], [25-34], and [35-60]. In the meantime, the *AgeCityGenderRec* recommender index is a three-dimensional grid based on three attributes (*age*, *city*, and *gender*). The *age* dimension is divided into three categories based on range of values. The *city* attribute has three values as {Minneapolis, St. Paul, Edina}, while the *gender* attribute is divided into two categorical values as {Male, Female}. The top left outer cell (check marked in Figure 3) in *AgeCityGenderRec* represents the values ⟨18-24, Minneapolis, Female⟩ that correspond to its values of the ⟨*age*, *city*, *gender*⟩ dimensions.

Each cell in the multi-dimensional grid points to a table, *RecModel*, that maintains auxiliary precomputed information to speed up the generation of the recommendation query result. The precomputed information may have different schema based on the underlying recommendation algorithm. For example, for the Item-Item collaborative filtering algorithm, *RecModel* represents an items similarity list with the schema (*ItemID1*, *ItemID2*, *SimScore*), where *SimScore* is computed per equation 1 (Section II).

Initialization. The multi-dimensional grid G is initialized upon issuing a CREATE RECOMMENDER statement, through two main steps: (1) *Grid Construction*, where we allocate the memory space for the grid, and decide on each cell size in terms of

the values it represents. In case of categorical attributes (e.g., Gender, Job, and City), we allocate one cell per attribute. For continuous domain attributes (e.g., age and salary), we divide the space into N parts, where parts have almost equal number of ratings. More sophisticated techniques can be used to divide the space. Yet, we opt for a simple division here as a proof of concept for RECATHON functionality. (2) *RecModel Building*, where the *RecModel* table for each cell C in G is built by running the specified recommender algorithm in the `CREATE RECOMMENDER` statement on the set of users U whose attributes correspond to the subdomain covered by C. For instance, in case of *ItemCosCF* recommendation algorithm, we scan the *ratings* table and run a nested loop algorithm over all items to calculate the cosine similarity score between every item pair in each cell C using equation 1. After the initialization procedure terminates, a pointer to the newly created grid structure G is added to the *RecIndex* field corresponding to the appropriate recommender entry in *RecCatalog*.

Maintenance. To get the most accurate result, the *RecModel* at each cell C should be updated with every single new rating for a user u whose attributes correspond to the subdomain covered by C. However, this is infeasible as most recommendation algorithms employ complex computational techniques that are very costly to update. The update maintenance procedure differs based on the underlying recommender algorithm, specified in the `CREATE RECOMMENDER` statement. Yet, most of the algorithms may call for a complete model rebuilding to incorporate any new update. To avoid such prohibitive cost, we decide to update the *RecModel* in a cell C only if the number of new updates in C reaches to a certain percentage ratio α (a system parameter) from the number of entries used to build the current model in C. We do so because an appealing quality of most supported recommendation algorithms is that as *RecModel* matures (i.e., more data is used to build it), more updates are needed to significantly change the recommendations produced from it. The smaller the value of α, the more up-to-date is the *RecModel*, yet, the larger the cost in frequently updating *RecModel*. A typical value of α is 0.5.

To realize this maintenance procedure, we maintain two counters in each cell C: (1) C_{total} as the number of entries used to build the current *RecModel* in C, and (2) C_{new} as the number of updates received in C since the last build of *RecModel*. Any update for a user u located in C will increase C_{new} by one, yet it would not affect C_{total}. Once $\frac{C_{new}}{C_{total}} > \alpha$, we rebuild the model in C, reset C_{new} to 0 and update C_{total} to be the current number of entries located in C.

Storage and Maintenance Optimization. Since storing and maintaining a recommendation model for every cell is quite expensive and may preclude system scalability, RECATHON automatically determines which recommender cells to materialize based on (per-cell) statistics. RECATHON only materializes *RecModel* only for *hot* cells to mitigate the *recommendation query latency* as well as reduce both the overall *maintenance cost* and the *storage overhead* occupied by the multidimensional grid. A *hot* cell is defined as the cell that receives more recommendation queries than data updates.

Algorithm 1 Cell Maintenance

```
1:  Function Maintenance (Cell C)
2:  if C_new/C_total > α then
3:      UpdateRate ← UC/(TS_U − TS_init)
4:      QueryRate ← QC/(TS_Q − TS_init)
5:      if (1 − β)×QueryRate ≥ ( β× UpdateRate ) then
6:          Create RecModel on disk if not already maintained
7:          Train RecModel using data lying within C
8:      else
9:          Delete RecModel from disk if already maintained
10:     C_new ← 0
11:     C_total ← the current number of entries located in C
```

To take the materialization decision, RECATHON maintains the following statistics for each grid cell C: (1) *Queries Count* QC: represents the number of issued recommendation queries over the recommendation model *RecModel* in cell C since the cell was created. (2) *Updates Count* UC: keeps track of the number of updates performed over the user/item/ratings data lying within cell C since the cell was created. (3) *Query TimeStamp* TS_Q: time stamp of the last query issued over cell C. (4) *Update TimeStamp* TS_U: time stamp of the last update transaction performed over cell C.

Algorithm 1 gives the pseudocode of the maintenance process. Using cell C statistics, the algorithm (pseudocode omitted for brevity) determines whether to keep and train the underlying (already existing) *RecModel* or drop it to save storage and maintenance overhead. The algorithm first leverages the maintained statistics to calculate the update rate *UpdateRate* and query rate *QueryRate* in cell C. If $(1-\beta)\times$ *QueryRate* is larger than or equal $\beta\times$ *UpdateRate*, that means cell C is hot, such that β denotes a system parameter specified by the user. In such case, the algorithm creates a recommendation model entry *RecModel* for cell C on disk and train *RecModel* using up-to-date data lying within C. Otherwise, we delete *RecModel* from disk and all incoming queries over C will have to first train *RecModel* on-the-fly.

Note that the value of β exhibits a tradeoff between (1) scalability measured in terms of storage and maintenance overhead on one hand and (2) query processing performance on the other hand. The larger the value of β the less the number of recommendation models maintained in the multidimensional grid and hence the higher the system scalability and the lower the query execution performance. On the contrary, a low β leads to more models being maintained and hence higher query execution performance for the price of less scalability.

V. QUERY PROCESSING

Giving an initialized multidimensional recommender, a main challenge (*Challenge III*) is how to efficiently execute recommendation queries over such recommender. The recommendation query takes two input parameters, a created recommender R as a virtual table and a querying user id uid. The algorithm then returns a set of tuples S such that each tuple $s \in S$; $s = \langle i, RecScore \rangle$ represents for each item i (unseen by uid), a recommendation score *RecScore* using R.

$Query\ 1$ in Table I is a very simple example of a recommendation SQL query. $Query\ 1$ lists all items that are not rated

Algorithm 2 RECOMMEND (uid, R)

```
1: Cat ← RecCatalog entry that corresponds to recommender R
2: G ← Cat.RecIndex (The Multi-dimensional Grid Index)
3: Attr ← The set of attributes in Cat.Attributes
4: AttrV ← Values of attributes Attr in table Cat.Users for uid
5: C ← The cell in G that corresponds to AttrV
6: I ← Set of items in table Cat.Ratings that are not rated by user uid
7: AnswerSet ← φ
8: for each item i ∈ I do
9:     RecScore ← Cat.Algorithm.GetRecScore(uid, i,
                                  Cat.Ratings,C.RecModel)
10:    AnswerSet ← AnswerSet ∪ ⟨i.iid, RecScore⟩
11: return AnswerSet
```

by users $uid = 2$ along with their predicted recommendation score, based on the user age, city, and gender. By specifying the *AgeCityGenderRec* recommender in the `FROM` clause, the RECOMMEND algorithm will look at the recommender catalog *RecCatalog* to find out that this recommender needs to know the age, city, and gender of user u with uid=2, and is built for the users table, depicted in Figure 1. Retrieving data from this table, the RECOMMEND algorithm finds that this query is for user *Carol*; a 22 year old female, living in 'Minneapolis'. With this data, we follow the index pointer of the *RecCatalog* entry to the three-dimensional grid index, and retrieve the *RecModel* stored in the grid cell that corresponds to the entry: ⟨22, Minneapolis, Female⟩. As the recommender functionality aims to provide a predicted score for those items that are not rated by the querying user, we need to scan the ratings table (obtained from the *RecCatalog*) to find those items I that are not rated by uid, but have some ratings from other users. For each of these items, we compute its predicted score using the underlying recommendation algorithm, also obtained from *RecCatalog*. Finally, the RECOMMEND algorithm returns all items in I along with their computed scores as the result.

Algorithm 2 gives the pseudocode of RECOMMEND. Given a user ID uid and a recommender R, the algorithm starts by retrieving the catalog entry Cat that corresponds to the recommender R. Following the information on the Cat entry, we get pointers to the corresponding multi-dimensional grid index G. Then, we retrieve the values for the attributes specified in R for the querying user uid. Based on these values, we locate the cell C in G that represents the user attributes. Since we need to compute the recommender score for only those items that are not rated by the user uid, yet are rated by others, we scan the table of ratings to get the set of such items I. For each of these items, we call the underlying recommendation algorithm $Cat.Algorithm$.GetRecScore, which is specific to R, as specified in the `USING` clause in the `CREATE RECOMMENDER` statement. Finally, the final answer set is composed of all the items in T, with the score of each item computed from the GetRecScore function.

For example, the GetRecScore function (Pseudocode omitted for space constraints) for the Item-Item Collaborative Filtering takes four parameters: the user u, item i, the table of ratings, and the *RecModel* table that is maintained at the corresponding grid cell of user u in the multi-dimensional grid index. Then, for each tuple t in the rating table that includes the user u, we retrieve the similarity score between the item i we want to score and the rated item $t.iid$, based on the table *RecModel*. We accumulate such scores, each weighted by the user rating $t.rating$. The final score ends up to be the average total weighted score divided by the total similarity score, as was described earlier in Section II.

A. Selection

In many cases, a user may want to only know the recommendation score for a specific item (e.g., a movie in Netflix) or a set of few items (e.g., a set of few books in Amazon). A straightforward execution of such queries would perform the recommendation generation algorithm first and then filter out the unneeded items. This plan performs well only if the predictive selectivity is very low. For highly selective predicates, the RECOMMEND algorithm performs lots of unnecessary work fetching all items data from disk and calculating their recommendation scores, while only few items are needed. Since in many cases, the predicate selectivity is very high (it is common to select only one item), RECATHON employs a variant of RECOMMEND, called FILTERRECOMMEND. Instead of calculating the recommendation scores for all items lying within cell C, FILTERRECOMMEND takes the filtering predicate as input and prunes the predicted recommendation score calculation for those items that do not satisfy the filtering predicate. To achieve that, we modify the loop (lines 8 to 10) in Algorithm 2 to iterate and calculate the recommendation only for items that satisfy the iid selection predicate.

B. Join

$Query\ 4$ in Table I gives an example of a recommendation query, where the output of the recommender generation algorithm needs to be joined with another table to get the movie names instead of their identifiers. This is a very common query in any recommender system. For example, Netflix and Amazon always return the item information, not the item identifiers. As the RECOMMEND algorithm only returns the item identifier, the result has to be joined with the item table to return the item information details, e.g., name, price, specs, etc. The straightforward plan for executing such join queries may be acceptable only if there is no filter over the items table, or the filter has very low selectivity. Otherwise, if the filter is highly selective, RECOMMEND ends up doing redundant work computing the scores for all items, while only few of them are needed. It is very common to have a very selective filter over the items table, e.g., only *Action* movies.

To efficiently support such queries, RECATHON employs the JOINRECOMMEND algorithm. Besides the user u and a recommender R, JOINRECOMMEND takes a joined database relation rel (e.g., `Movies`) as input, combines their tuples, and returns the joined result. Analogous to index nested loop join, JOINRECOMMEND employs the input relation rel as the outer relation. For each retrieved tuple $tup \in rel$, the algorithm calculates the score for item i with iid equal to

tup.iid in the same way it was calculated in the RECOMMEND algorithm (Algorithm 2). The algorithm then concatenates (iid,$RecScore$) and tup and the resulting joined tuple (tup,iid,$Recscore$) is finally added to the join answer S. The algorithm terminates when there are no more tuples left in rel.

C. Ranking (Top-k)

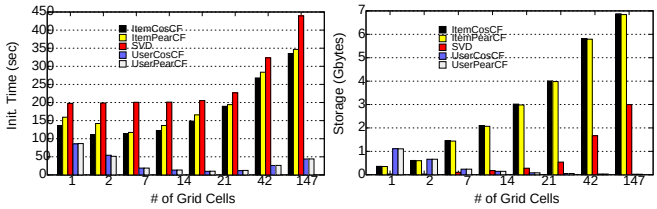
$Query$ 2 in Table I gives an example of a ranking query where we need to return only the top-10 recommended items. This is a very important query as it is very common to return to the user a limited number of recommended items rather than all items with their scores. The straightforward query plan for this query would incur too much overhead in computing the score for all items, and then return only the top 10 ones.

Since top-k recommendation is by far the most commonly used query in existing commercial applications (e.g., Amazon and Netflix), we further optimize its query execution performance by pre-computing a set of top-$|L|$ items for each user u and caching these items in a sorted list, named L_u. When a querying user u issues a top-k recommendation query, RECATHON just needs to fetch the first k items in L_u ($k << |L|$). Note that this approach functions correctly only for pure top-k recommendation. If the recommendation query consists of a top-k operation accompanied with selection or join, using the pre-computed list will not necessarily return a correct answer. For example, consider $Query$ 5 in Table I. This query recommends the top 5 Action movies to user $uid = 1$, based on the user age (i.e., use the AgeRec recommender).

$Query$ 5 needs to join the `AgeRec` recommender with the `Movies` table to only select *Action* movies and then return the top-5 *Action* movies to user 1. In that case, L_u may not contain any *Action* movie whereas the `Movies` table might contain some. Hence, RECATHON would apply JOINRECOMMEND first on recommender `AgeRec` and table `Movies` and then perform a traditional top-k operation on the join result.

VI. EXPERIMENTAL EVALUATION

This section presents a comprehensive experimental evaluation of RECATHON based on an actual system implementation inside RecDB [26] integrated with PostgreSQL 9.2. All proposed techniques are implemented using the iterator model adopted by PostgreSQL for operator implementations. We evaluate RECATHON using two real datasets described as follows: (1) **MovieLens**: a real movie recommendation dataset [20] that consists of 6040 users, 3883 movies, and one million (1M) ratings. Each user has *gender*, *age*, and *job* attributes. The user *gender* attribute consists of two values {Female, Male}, the *age* attribute is partitioned into seven ranges, and the *job* attribute has 21 job categories. Each movie has name and genre attributes. The ratings data contains historical ratings (in a scale from 1 to 5) that users have assigned to movies. (2) **Foursquare**: a real venue (e.g., restaurant) recommendation dataset extracted [27] from Foursquare website, and consists of $150K$ users, $90K$ venues, and $1M$ ratings. Each user has a spatial location attribute that represents where that user lives. Each item is also assigned a spatial location that represents where this item is located.



(a) Initialization Time (b) Storage Overhead

Fig. 4. Initialization Time and Storage Overhead.

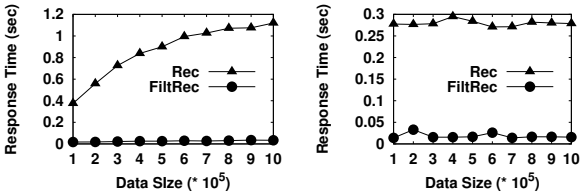
Section VI-A studies the storage and initialization overhead while Section VI-B studies queries performance. All experiments were performed on a machine with 3.6 Ghz Quad-Core processor, 16 GB RAM, 500 GB storage, and running Ubuntu Linux 12.04. The default dataset is MovieLens.

A. Recommender Creation

We evaluate the context-aware recommender creation and initialization process performance using the following two metrics: (1) *Recommender Initialization Time*: the total runtime (in seconds) taken by the system to process a `CREATE RECOMMENDER` statement, and (2) *Recommender Storage Overhead*: the amount of storage (in Gbytes) occupied by the multidimensional grid and recommendation models created upon recommender creation. We run our experiments for the following five popular recommendation algorithms, all supported and built into RECATHON:

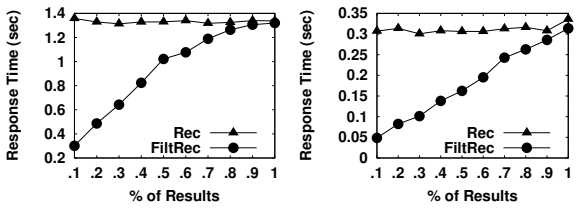
1) **ItemCosCF**: Item-Item collaborative filtering with cosine distance used to measure similarity among items (explained before in section II).
2) **ItemPearCF**: Item-Item Collaborative filtering with Pearson correlation used to measure similarity among items.
3) **UserCosCF**: User-User Collaborative Filtering with cosine distance used to measure similarity among users.
4) **UserPearCF**: User-User Collaborative Filtering with Pearson correlation used to measure similarity among users.
5) **SVD**: Regularized Gradient Descent Singular Value Decomposition.

Figure 4 gives the effect of varying the number of grid cells for the multi-dimensional grid structure on the context-aware recommender initialization time and storage overhead. Since we have three contextual attributes, *age*, *gender*, and *job*, we have the possibility of eight context-aware recommenders with 1, 2, 7, 14, 21, 42, 147, and 294 three-dimensional grid cells. A context-aware recommender with one grid cell corresponds to context-free recommenders that do not take care of any attributes. On the other side, a context-aware recommender of 294 cells corresponds to the combination of three attributes: *age*, *gender*, and *job*. The number 294 is the product of 21, 7, and 2, as the number of categories of *job*, *age*, and *gender*



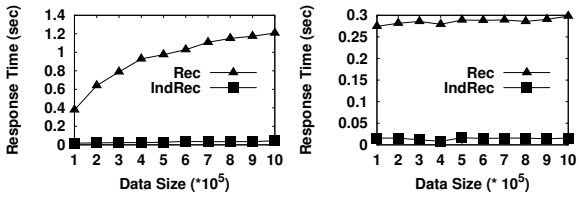
(a) ItemCosCF (b) SVD

Fig. 5. iid Selection Predicate: Varying Data Size (MovieLens)



(a) ItemCosCF (b) SVD

Fig. 6. iid Selection Predicate: Varying Selectivity (MovieLens)



(a) ItemCosCF (b) SVD

Fig. 7. Ranking (Top-k): Varying Data Size (MovieLens)

attributes, respectively. Similarly, 147 grid cells correspond to an $\langle age, job \rangle$-aware recommender, and so on.

Figure 4 shows that the higher the number of grid cells, the higher the initialization time and storage overhead for the *ItemCosCF*, *ItemPearCF*, and *SVD* recommendation algorithms. The main reason is that more grid cells lead to building more recommendation models, as we build one recommendation model per grid cell. This consumes both storage and computation time. However, the opposite scenario happens for *UserCosCF* and *UserPearCF* algorithms. This counter intuitive behavior is mainly because these two recommender algorithms partition the user ratings based on the user attributes and hence, the number of users is small in each cell and building a user similarity list for several small cells is faster (and requires less storage) than building the user similarity list for one fat cell. Overall, for all recommender algorithms and number of grid cells, RECATHON is able to provide a reasonable computational and storage overhead.

Comparing various recommender algorithms to each other shows that *UserCosCF* and *UserPearCF* are mostly faster and occupy less storage on disk than *ItemCosCF* and *ItemPearCF*. That happens due to the fact that the number of ratings per user in the dataset is less than the number of ratings per item. For all created recommenders, *SVD* consistently takes more time than other algorithms since *SVD* is an iterative algorithm that takes several iterations to build the recommendation model.

B. Recommender Queries

In this section, we evaluate the query execution performance in terms of the query response time. We consider the following four different approaches for query execution: **(1) Rec**: a query plan that only relies on the RECOMMEND algorithm to execute recommendation queries. **(2) FiltRec**: a query plan that leverages the FILTERRECOMMEND algorithm to optimize recommendation queries with a predicate over iid. **(3) IndRec**: a plan that exploits the pre-computed sorted list of recommended items to execute a ranking (Top-k) query. **(4) JoinRec**: a plan that employs the JOINRECOMMEND algorithm to join a recommendation with a database table. For space constraints, we plot only the results of three recommendation algorithms, namely, *ItemCosCF*, *UserPearCF*, and *SVD*. Experiments run for the $\langle age, gender \rangle$-aware recommender.

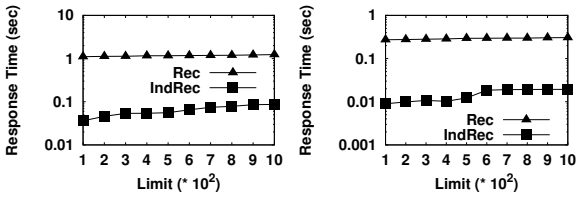
1) Selection Predicate: In this section, we study the performance of recommendation queries with selection predicate applied the item. Unless mentioned otherwise, the query selectivity is set to 25% and the data size is 1M ratings.

Varying data size. Figure 5 studies the impact of data size on the query execution performance. We vary the data size from 100K to 1M ratings, executing a set of 100 synthetically generated queries using *ItemCosCF* and *SVD* recommendation algorithms. For all recommender algorithms, *FiltRec* outperforms *Rec* by at least an order of magnitude. This is obvious since *FiltRec* applies the *iid* filtering predicate before calculating the predicted recommendation score for an item, which saves a huge amount of effort wasted by *Rec* in applying the RECOMMEND algorithm first on all items and then performing the predicate filtering step. In the meantime, the response time increases as the data size gets bigger since we need to retrieve more recommendation models in *ItemCosCF*. In the *SVD* case, the response time remains unchanged.

Varying Selectivity. Figure 6 gives the impact of *iid* predicate selectivity on the average query response time. As it turns out from the figure, as we increase the percentage of reported items (decrease selectivity), the average query response time in *Rec* remains constant since the RECOMMEND algorithm predicts the recommendation score for all items anyway before applying the *iid* predicate. On the other hand, *FiltRec* query response time increases linearly with the percentage of reported items until it reaches the same performance as *Rec* when 100% of items are reported. That happens because *FiltRec*, when 100% of items are returned, performs the same amount of recommendation score calculation as *Rec*. However, for higher selectivity (i.e., lower % of results), *FiltRec* outperforms *Rec* by more than an order of magnitude. With *iid* selection, a typical selectivity would be very high, i.e., less than 1%, which shows better performance for *FiltRec* over *Rec*.

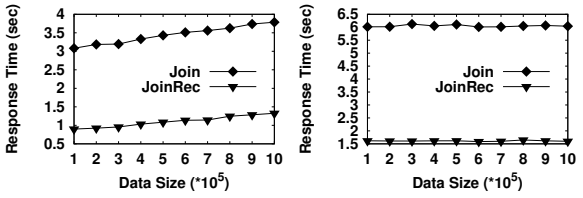
2) Ranking (Top-k): This section studies the performance of top-k recommendation queries. Unless mentioned otherwise, k is set to 1000 and the data size is 1M ratings.

Varying data size. Figure 7 depicts the effect of data size on top-k recommendation query performance. As given in the figure, *IndRec* exhibits more than an order of magnitude performance over *Rec*. That is justified by the fact that *IndRec*



(a) ItemCosCF (b) SVD

Fig. 8. Ranking (Top-k): Varying Limit (k) Size (MovieLens)



(a) ItemCosCF (b) SVD

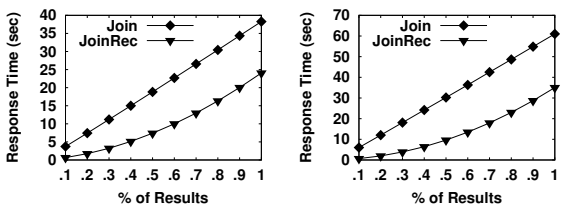
Fig. 9. Join: Varying Joined Recommender Data Size (Foursquare).

leverages the pre-computed recommendation scores, created for active users, in the *RecScore Index* to retrieve items in sorted order. Hence, the limit operator terminates early when the required number of items is retrieved. On the other hand, *Rec* has to first calculate recommendation scores, then sort items based on their score, and finally pick the top-k items. Note that the query response time in *Rec* increases as the data gets bigger (except for *SVD*) because *Rec* takes more time accessing bigger models. However, the response time in *IndRec* slightly increases since *IndRec* retrieve pre-computed recommendation scored in sorted order.

Varying k. Figure 8 gives the impact of k on top-k recommendation query performance. We vary k from 100 to 1000, generate a workload of 100 top-k recommendation queries, and measure the response time for *ItemCosCF* and *SVD* algorithms. In all algorithms, *IndRec* achieves more than an order of magnitude better performance than *Rec* for all values of k. Moreover, *Rec* performance is constant for different k values, which is explained by the fact that sorting in *Rec* is dominating the query execution performance. On the other side, the response time in *IndRec* slightly increases for larger k values as more items are accessed in *RecScore Index*.

3) Join: This section studies the performance of recommendation queries that involve joining the recommendation answer with other database tables in the *Foursquare* dataset. The joined table has a selection predicate that filters out unwanted items. Unless mentioned otherwise, the predicate selectivity is set to 25% and the data size is 1M ratings.

Varying data size. In this experiment, we build recommenders with different data sizes (100K to 1M ratings), and we generate a workload of 100 join queries that join the created recommender and the movies table. Figure 9 shows that `JoinRec` scales about an order of magnitude better than `Join` for all recommendation algorithms. The main reason is that JOINRECOMMEND efficiently calculates the predicted score only for filtered items. In the meantime, the bigger the data size, the worse the query execution performance for both `Join` and `JoinRec`, since more data are joined. That happens for all recommendation algorithms, except for *SVD*.



(a) ItemCosCF (b) SVD

Fig. 10. Join: Varying Joined Table (rel) Selectivity (Foursquare).

Varying Selectivity. In these experiments, we vary the selectivity of the joined table with the input recommender. We generate a workload of 100 random join queries of the same selectivity, and execute such queries for each recommendation algorithm. We simulate the selectivity change in terms of the ratio of output tuples (filtered by a selection predicate) over the original number of tuples in the joined table rel. Figure 10 shows that *JoinRec* exhibits more than an order of magnitude better performance than *Join* for high selectivity (small % of rel). However, when the selectivity decreases (% of rel increases), *JoinRec* performance becomes closer to *Join* since *JoinRec* has to compute the predicted score for more items.

VII. RELATED WORK

Recommendation Algorithms. A Recommendation algorithm speculates how much a user would like an item she has never seen (bought, watched) before. *Collaborative Filtering* [23] is considered the most popular amongst several recommendation algorithms proposed in the literature [4], [23]. There are several methods to perform collaborative filtering including item-item [25], user-user [23], regression-based [25], or approaches that use more sophisticated probabilistic models (e.g., Bayesian Networks [8]). Collaborative filtering techniques analyze past community opinions to find similar users (or items) to suggest k personalized items (e.g., movies) to a querying user u. RECATHON does not come up with a novel recommendation algorithm. However, it adapts existing algorithms to generate multidimensional recommendation.

Recommender systems in databases. Few, and recent, works have studied the problem of integrating the recommender system functionality with database systems. This includes a framework for expressing flexible recommendation by separating the logical representation of a recommender system from its physical execution [15], [16], algorithms for answering recommendation requests with complex constraints [21], [22], a query language for recommendation [5], and extensible frameworks to define new recommendation algorithms [11], [18], leveraging recommendation for database exploration [10]. Unlike RECATHON, the aforementioned work lacks one or more of the following features: (1) Producing multidimensional recommendation to users, (2) Executing online recommendation queries in a near real time manner, (3) Efficiently initializing and maintaining multiple recommendation algorithms.

Context-Aware Recommendation. Existing context-aware recommendation algorithms [6] focus on leveraging contextual information to improve recommendation accuracy over classical recommendation techniques. Conceptual models for context-aware recommendation have also been proposed for better representation of multidimensional attributes in recommender systems [6]. Several frameworks have proposed defining context-aware recommendation services over the web using either client/server architecture [1], or by mimicking successful web development paradigms [12]. Such techniques, though they provide support for context-aware recommendation, do not consider system performance issues (e.g., efficiency and scalability). Location-aware recommender systems [17], [27] present a special case of context-aware recommender systems, where efficiency and scalability are main concerns. However, the proposed techniques for location-aware recommender systems are strongly geared towards the spatial attribute, with no direct applicability to other attributes.

VIII. Conclusion

We have presented RECATHON; a multidimensional recommender system. In an analogous way to creating indexes on table attributes that are most likely to appear in incoming queries, RECATHON users can create a recommender over a certain set of attributes. The created recommender can be then queried to provide recommended items based on a certain set of attributes. Examples of recommender queries include "*Recommend me 10 movies that people in my age and gender would like*". In this case, we issue an $\langle age, gender \rangle$-aware recommender. If such recommender is created a-priori, this query will entertain a real-time response. RECATHON employs a multi-dimensional index for each built recommender to support efficient recommender queries. The system is able to seamlessly integrate the recommendation functionality with traditional database operations to execute a variety of context-aware recommendation queries. Extensive experiments show that RECATHON exhibits real time performance for large-scale context-aware recommendation scenarios.

IX. Acknowledgement

This work is supported by the National Science Foundation, USA, under Grants IIS-0952977 and IIS-1218168.

References

[1] S. Abbar, M. Bouzeghoub, and S. Lopez. Context-aware recommender systems: A service-oriented approach. In *PersDB*, 2009.

[2] F. Abel, Q. Gao, G.-J. Houben, and K. Tao. Analyzing user modeling on twitter for personalized news recommendations. In *Proceedings of the International Conference on User Modeling, Adaption and Personalization, UMAP*, 2011.

[3] G. Adomavicius, B. Mobasher, F. Ricci, and A. Tuzhilin. Context-aware recommender systems. *AI Magazine*, 32(3), 2011.

[4] G. Adomavicius and A. Tuzhilin. Toward the Next Generation of Recommender Systems: A Survey of the State-of-the-Art and Possible Extensions. *IEEE Transactions on Knowledge and Data Engineering, TKDE*, 17(6), 2005.

[5] G. Adomavicius, A. Tuzhilin, and R. Zheng. Request: A query language for customizing recommendations. *Information Systems Research*, 22(1):99–117, 2011.

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

[7] S. Amer-Yahia, A. Galland, J. Stoyanovich, and C. Yu. From del.icio.us to x.qui.site: recommendations in social tagging sites. In *Proceedings of the ACM International Conference on Management of Data, SIGMOD*, 2008.

[8] J. S. Breese, D. Heckerman, and C. Kadie. Empirical Analysis of Predictive Algorithms for Collaborative Filtering. In *Proceedings of the Conference on Uncertainty in Artificial Intelligence, UAI*, 1998.

[9] A. Das et al. Google News Personalization: Scalable Online Collaborative Filtering. In *Proceedings of the International World Wide Web Conference, WWW*, 2007.

[10] M. Drosou and E. Pitoura. YMALDB: A Result-Driven Recommendation System for Databases. In *Proceedings of the International Conference on Extending Database Technology, EDBT*, 2013.

[11] M. D. Ekstrand, M. Ludwig, J. A. Konstan, and J. T. Riedl. Rethinking the recommender research ecosystem: reproducibility, openness, and lenskit. In *RecSys*, 2011.

[12] T. Hussein, T. Linder, W. Gaulke, and J. Ziegler. Context-aware Recommendations on Rails. In *International Workshop on Context-aware Recommender Systems, CARS*, 2009.

[13] H. Kailun, W. Hsu, and M. L. Lee. Utilizing Social Pressure in Recommender Systems. In *Proceedings of the IEEE International Conference on Data Engineering, ICDE*, 2013.

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

[15] G. Koutrika, B. Bercovitz, and H. Garcia-Molina. FlexRecs: Expressing and Combining Flexible Recommendations. In *Proceedings of the ACM International Conference on Management of Data, SIGMOD*, 2009.

[16] J. J. Levandoski, M. Sarwat, M. D. Ekstrand, and M. F. Mokbel. RecStore: An Extensible and Adaptive Framework for Online Recommender Queries inside the Database Engine. In *Proceedings of the International Conference on Extending Database Technology, EDBT*, 2012.

[17] J. J. Levandoski, M. Sarwat, A. Eldawy, and M. F. Mokbel. LARS: A Location-Aware Recommender System. In *Proceedings of the IEEE International Conference on Data Engineering, ICDE*, 2012.

[18] J. J. Levandoski, M. Sarwat, M. F. Mokbel, and M. D. Ekstrand. RecStore: An Extensible and Adaptive Framework for Online Recommender Queries inside the Database Engine. In *Proceedings of the International Conference on Extending Database Technology, EDBT*, 2012.

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

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

[21] A. G. Parameswaran, H. Garcia-Molina, and J. D. Ullman. Evaluating, combining and generalizing recommendations with prerequisites. In *Proceedings of the International Conference on Information and Knowledge Managemen, CIKM*, 2010.

[22] A. G. Parameswaran, P. Venetis, and H. Garcia-Molina. Recommendation systems with complex constraints: A course recommendation perspective. *ACM Transactions on Information Systems, TOIS*, 29(4):20, 2011.

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

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

[25] B. Sarwar, G. Karypis, J. Konstan, and J. Riedl. Item-Based Collaborative Filtering Recommendation Algorithms. In *Proceedings of the International World Wide Web Conference, WWW*, 2001.

[26] M. Sarwat, J. Avery, and M. F. Mokbel. RecDB in Action: Recommendation Made Easy in Relational Databases. *Proceedings of the Very Large DataBases Endowment, PVLDB*, 6(12):1242–1245, 2013.

[27] M. Sarwat, J. J. Levandoski, A. Eldawy, and M. F. Mokbel. LARS*: A Scalable and Efficient Location-Aware Recommender System. In *IEEE Transactions on Knowledge and Data Engineering, TKDE*, 2014.

[28] M. Vartak and S. Madden. CHIC: a combination-based recommendation system. In *Proceedings of the ACM International Conference on Management of Data, SIGMOD*, 2013.

[29] H. Yin, B. Cui, J. Li, J. Yao, and C. Chen. Challenging the Long Tail Recommendation. *PVLDB*, 5(9):896–907, 2012.