2017 IEEE 33rd International Conference on Data Engineering

Database System Support for Personalized Recommendation Applications

Mohamed Sarwat[1], Raha Moraffah[2], Mohamed F. Mokbel[3], James L. Avery[4]
[1,2]Arizona State University, Tempe, AZ 85287
[3]University of Minnesota, Minneapolis, MN 55455
[4]IBM, Austin, TX 78758
[1]msarwat@asu.edu, [2]rmoraffa@asu.edu, [3]mokbel@cs.umn.edu, [4]jlavery@us.ibm.com

***Abstract*—Personalized recommendation has become popular in modern web services. For instance, Amazon recommends new items to shoppers. Also, Netflix recommends shows to viewers, and Facebook recommends friends to its users. Despite the ubiquity of recommendation applications, classic database management systems still do not provide in-house support for recommending data stored in the database. In this paper, we present the anatomy of RecDB an open source PostgreSQL-based system that provides a unified approach for declarative data recommendation inside the database engine. RecDB realizes the personalized recommendation functionality as query operators inside the database kernel. That facilitates applying the recommendation functionality and typical database operations (e.g., Selection, Join, Top-k) side-by-side. To further reduce the application latency, RecDB pre-computes and caches the generated recommendation in the database. In the paper, we present extensive experiments that study the performance of personalized recommendation applications based on an actual implementation inside PostgreSQL 9.2 using real Movie recommendation and location-aware recommendation scenarios. The results show that a recommendation-aware database engine, i.e., RecDB, outperforms the classic approach that implements the recommendation logic on-top of the database engine in various recommendation applications.**

***Keywords*-Database, Recommendation, Analytics, Personalization, Machine Learning, Join, Indexing**

I. Introduction

A classic database management system (DBMS) expects its users to know exactly what data they need to query in advance. In many cases, the user (data scientist/analyst) does not know exactly what kind of information she needs to extract from the database. A user might need a database management system that allows her to explore, and not only to query data using classic query processing techniques. Recently, recommender systems, as an exploration method, have grabbed attention in both industry [5], [7], [18] and academia [1], [2], [14], [13], [21], [27], [28]. Recommender systems are employed on a daily basis to help users explore interesting movies (e.g., Netflix), books/products (e.g., Amazon), friends (e.g., Facebook), and news articles (e.g., Google News). A recommender system exploits a large history of user's preferences (e.g., movie ratings) and/or behavior (e.g., watching, reading) to extract a set of interesting data items for each user [1], [22], [14], [13], [21], [26], [27], [28]. The idea is to take as input a set of users U, items I, and ratings R to build a recommendation model *M*. Upon receiving a recommendation query, the recommender model *M* is used to compute a utility function $PredictRating(u, i)$ that predicts how much a user $u \in U$ would like an item $i \in I$ [2], [9], [11].

Classic DBMSs do not provide in-house support for collaborative recommendation. A straightforward solution implements the recommendation functionality *on-top* of the database system, abbr. *OnTopDB*. However, the *OnTopDB* approach suffers from the following: (1) Tremendous overhead of extracting the data from the database, loading it to a specialized recommendation engine [9], and then loading the produced recommendation back to the database. (2) The *OnTopDB* approach, e.g., [6] does not harness the full power of the database kernel that includes query optimization, indexing, and materialized views. That may lead to query execution plans that perform unnecessary work, incurring high latency, especially when only a subset of the recommendation answer is required. On the other hand, incorporating the collaborative recommendation functionality inside the DBMS kernel is beneficial for the following reasons: (1) Collaborative recommendation algorithms take as input structured data (users, items, and user historical preferences) that could be adequately stored and accessed using a relational database system. (2) The In-DBMS approach facilitates applying the recommendation functionality and typical database operations (e.g., Selection, Join) side-by-side. That allows application developers to go beyond traditional recommendation applications, e.g., "Recommend to Alice ten movies" to define recommendation scenarios like "Recommend ten nearby (filter based on location) restaurants to Alice" and "Recommend to Bob ten movies watched by her friends (filter based on friends)".

In this paper, we introduce RecDB, a system approach to incorporating a collaborative recommender system completely inside a database management system [24]. RecDB provides an intuitive interface for application developers to build custom-made recommenders. To achieve that, we extend SQL with new statements to create and/or drop recommenders, namely `CREATE/DROP RECOMMENDER`. RecDB initializes and maintains each created recommender that consists of a recommendation model *M* that is queried to generate recommendations to end-users [16].

RecDB proposes a novel querying paradigm that allows DBMS users to express recommendation as part of their

James L. Avery worked on the project while at the University of Minnesota

2375-026X/17 $31.00 © 2017 IEEE
DOI 10.1109/ICDE.2017.174

uid	name	City	Age	Gender
1	Alice	'Minneapolis, MN'	18	Female
2	Bob	'Austin, TX'	27	Male
3	Carol	'Minneapolis, MN'	45	Female
4	Eve	'San Diego, MN'	34	Female
...				

(a) **Users**

mid	name	Director	Genre
1	'Spartacus'	'Stanley Kubrick'	'Action'
2	'Inception'	'Christopher Nolan'	'Suspense'
3	'The Matrix'	'Lana Wachowski'	'Sci-Fi'
...			

(b) **Movies**

uid	iid	ratingval
1	1	1.5
2	2	3.5
2	1	4.5
2	3	2
3	2	1
3	1	2
4	2	1
...		

(c) **Ratings**

Fig. 1: Recommender Input Data. The Ratings table contains a set of opinions that users (uid) expressed over items (iid).

issued SQL queries. To achieve that, the system allows users to specify the ratings table in the `FROM` clause and invokes a `RECOMMEND` clause; a SQL extension to denote the recommendation functionality. The system then optimizes the recommendation-aware SQL query through a set of newly introduced recommendation-aware relational operators. Finally, the query executor produces a set of recommended data items, along with their predicted recommendation scores. In summary, the contributions of this paper are:

1) We present RECDB[1] – a unified approach for declarative data recommendation inside the database engine. We extend SQL with new clauses that create, drop, and query recommenders inside a database engine (Section III).
2) We provide recommendation-aware relational operators that incorporates the recommendation operation into the core functionality of traditional relational operators (*selection*, *join*, and *ranking*) to realize a variety of popular data recommendation algorithms inside the database query processor.
3) We perform preliminary experiments (Section VI), based on actual system implementation inside PostgreSQL. Using real data extracted from movie recommendation applications shows that RECDB exhibits up to two orders of magnitude better query performance than the straightforward OnTopDB approach for a myriad recommendation scenarios.

Scope. This paper assumes a shared-memory/shared-disk architecture. However, the ideas presented in RECDB could be extended to a shared-nothing distributed database architecture. Moreover, RECDB does not introduce a novel recommendation model with higher accuracy. It instead focuses on performance issues that include query execution latency.

II. Background and Related Work

Related work to RECDB spans various areas, which include recommendation algorithms, recommender systems in database systems, and context-aware recommendations.

Recommendation Algorithms. A Recommender system takes as input a set of users U, items I, and ratings (history of users opinions over items) R and estimates a utility function $\mathcal{F}(u, i)$ that predicts how much a certain user $u \in U$ will like an item $i \in I$ such that i has not been already seen by u [2]. To estimate such utility function, many recommendation algorithms have been proposed in the literature [2] that can be classified as follows: (1) Non-Personalized: this class of algorithms leverages statistics and/or summary information to recommend the same interesting (e.g., the most highly rated) items to all users. (2) Content-based Filtering: analyzes the item's content information and recommends to a user a set of items similar (in content) to those she liked before. (3) Collaborative Filtering: harnesses the historical preferences (tastes) of many users to predict how much a specific user would like a certain item. Collaborative filtering recommenders falls into two main categories: (a) Neighborhood-based [2]: that leverages the similarity between system users or items to estimate how much a user like an item. (b) Matrix Factorization [23]: that trains a machine learning or a probabilistic model that predicts how much a user would like an unseen item. In this paper, we focus more on collaborative recommendation algorithms.

Collaborative Filtering 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) [2] 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 $l \in \mathcal{L}$

[1] https://github.com/DataSystemsLab/recdb-postgresql

weighted by *sim(i,l)*, the similarity of l to candidate item i, then normalized by the sum of similarity between i and l.

Contextual Recommendation. Existing context-aware recommendation algorithms [4] 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 [4]. Several frameworks have proposed defining context-aware recommendation services over the web using either client/server architecture, 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 [25] 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.

Recommender System Architectures. (I) Offline systems pre-compute recommendation offline for all users, store them on disk, and returns the pre-computed recommendation to a user when she logs on to the system. Such offline systems include: software libraries [9], [10] that perform the recommendation process in-memory, e.g., LensKit, as well as large-scale offline systems, e.g., Mahout, that are built on-top of Hadoop and run the recommendation generation process as a batch processing task. (II) Online systems produce recommendation online for each user, when she logs on to the system, based on the recent snapshot of the user/item ratings data. Online architectures have the advantage of producing fresher recommendation than their offline counterparts. Moreover, online systems are capable of generating arbitrary recommendation to end-users. Such arbitrary recommendation queries require integrating the recommendation logic with other data access operations at query time which cannot be generated using offline recommender systems. On the other side, online systems incur more recommendation generation latency from an end-user perspective, as opposed to offline systems that delivers the pre-computed recommendation fast to end-users at query time. In this paper, we focus on online recommender systems.

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], algorithms for answering recommendation requests with complex constraints [19], [20], a query language for recommendation [3], and extensible frameworks to define new recommendation algorithms [9], [17], leveraging recommendation for database exploration [6], [8]. Unlike RECDB, the aforementioned work lacks one or more of the following features: (1) Executing online arbitrary recommendation queries, (2) Efficiently initializing and maintaining multiple recommendation algorithms, (3) Native support for recommendation inside the database engine.

III. SYSTEM OVERVIEW

RECDB takes as input a user/item *Ratings* table that contains a set of users U, a set of data items I, and a set of ratings that each tuple represents a rating $ratingval$ that a user $u \in U$ assigned to a data item $i \in I$. Ratings represent 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).

Running Example. Figure 1 gives an example of movie recommendation data. The users table represents the set of system users (e.g., Alice). Each user has a user ID (primary key) and other attributes (e.g. home town (city)). The Items table represents a set of Movies (data items) such that each movie has an ID (primary key) and other attributes (e.g., movie director, genre). The ratings table contains the ratings that some users (uid is a foreign key that references uid in the users table) have assigned to some movies (iid is a foreign key that references mid in the movies table).

RECDB provides a tool to the system users to freely decide which recommendation algorithm to be used in building a recommender. The system allows users to employ a SQL-like clause to declare a new recommender by specifying the recommender input data source (i.e., ratings table) and recommendation algorithm. 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 recommendation-aware SQL query. Internals of RECDB 's recommendation-aware query executor that enable such interface are described in later sections.

A. Creating a Recommender

RECDB allows users to create recommenders inside the database engine. A created recommender will be utilized by the query execution engine to recommend data items to querying users. To allow creating a new recommender, RECDB employs a new SQL statement (similar to creating a table or a view), called `CREATE RECOMMENDER`, as follows:

```
CREATE RECOMMENDER [Recommender Name]
ON [Ratings Table]
USERS FROM [Users ID Column]
ITEMS FROM [Items ID Column]
RATINGS FROM [Ratings Value Column]
USING [Recommendation algorithm]
```

Semantics. The recommender creation SQL, presented above, takes the following parameters: (1) `Recommender name` is a unique name assigned to the created Recommender. (2) `Ratings Table` is the table that contains the input user/item ratings data (e.g., see Figure 1). (3) `Users ID Column`, `Items ID Column`, and `Ratings Value Column` are the columns containing the users, items, and ratings data in the ratings table. (4) `Recommendation algorithm` is the algorithm used to build the recommender. Currently, RECDB supports three main recommendation algorithms (along with their variants): (a) Item-Item Collaborative Filtering with Cosine (abbr. ItemCosCF) or Pearson Correlation (abbr. ItemPearCF) similarity functions, (b) User-User Collaborative filtering with Cosine or Pearson Correlation similarity functions

(abbr. UserCosCF / UserPearCF), and (c) Regularized Gradient Descent Singular Value Decomposition (abbr. SVD). If no recommendation algorithm is specified, RECDB employs by default the ItemCosCF algorithm. Examples are given below:

Recommender 1. GeneralRec*: a ItemCosCF recommender created on the input data stored in the Ratings table.*

```
Create Recommender GeneralRec On Ratings
Users From uid Item From iid Ratings From ratingval
Using ItemCosCF
```

This SQL creates a recommender, named *GeneralRec* inside RECDB. *GeneralRec* is a traditional recommender that can be queried to recommend a set of movies for a certain user, e.g., *recommend me five movies*.

Recommender Initialization. In this step, RECDB trains 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 collaborative filtering (cosine similarity measure) model (ItemCosCF) [2] 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 $SimScore(i_p, i_q) = \frac{\vec{i_p} \cdot \vec{i_q}}{\|\vec{i_p}\| \|\vec{i_q}\|}$. The score is calculated using the vector's co-rated dimensions. Cosine distance is useful for numeric ratings (e.g., scale [1,5]). For unary ratings, other functions are used (e.g., absolute sum).

Maintaining a Recommender. To get the most accurate result, *RecModel* should be updated with every newly inserted rating by a user u assigned to an item i. However, doing so is infeasible as collaborative 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* only if the number of new updates reaches to a certain percentage ratio $N\%$ (a system parameter) from the number of entries used to build the current model. 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.

B. Recommendation Query

Once a recommender is created and initialized using the `CREATE RECOMMENDER` statement, users can issue SQL queries that harnesses the created recommender to produce data recommendation, as follows:

```
SELECT     [Select Clause]
FROM       [Rating Table]
RECOMMEND  <UserID> TO <ItemID> ON <RatingVal>
USING      [Recommendation Algorithm]
WHERE      [Where Clause]
```

Query Semantics. The `SELECT` and `WHERE` clauses are typical as in any SQL query. The `FROM` clause may directly accept a `[Ratings]` table with the same schema passed to the `CREATE RECOMMENDER` statement. The `RECOMMEND` clause is responsible for predicting how much the system users would like the unseen items. The application developer also needs to specify the Item ID, User ID (i.e., `TO <ItemID>`), and Rating Value (i.e., `ON <RatingVal>`) Columns. RECDB then returns a set of tuples S that each tuple $s \in S$; $s = \langle UserID, ItemID, RatingVal\rangle$ represents the predicted rating score *RatingVal* for each user and item pair based on the recommendation algorithm specified in the `USING` clause. An example of a query is given below:

Query 1. *Return ten movies to user with ID 1 using the Item-Item Collaborative Filtering algorithm.*

```
Select R.uid, R.iid, R.ratingval From Ratings as R
Recommend R.iid To R.uid On R.ratingVal Using ItemCosCF Where R.uid=1
Order By R.ratingVal Desc Limit 10
```

In this case, the system uses the `GeneralRec` recommender, which was created before using a `CREATE RECOMMENDER`. Since this recommender was created based on the age attribute, Query 1 will predict the ratings based on the algorithm passed to `ItemCosCF` algorithm. The query finally returns the Top-10 movies to user 1 in a descending order of the predicted rating value (`ratingval`).

IV. QUERY PROCESSING

This section discusses RECDB internal processing of recommender queries. RECDB encapsulates the recommendation functionality into a new family of recommendation-aware query operators. Being a query operator allows the recommendation functionality to be a part of a larger query plan that includes other query operators, e.g., selection, projection, and join. It also means that the recommendation functionality will be treated as a first class citizen operation, allowing a myriad of query optimization techniques that can be integrated to speed up recommendation queries.

A. Recommendation Operator

RECDB employs three main versions of the RECOMMEND operator; one for each recommendation algorithm. Each operator take as input a ratings table and return a set of tuples S such that each tuple $s \in S$; $s = \langle$*uid,iid,ratingval*$\rangle$ represents a predicted rating score *ratingval* for each item iid unseen by user uid. This section first describes the recommendation operators and then explains how they are integrated in the SQL query pipeline.

1) Item-Item Collaborative Filtering Operator: RECDB stores the item-item similarity list as a table, called the *Item Neighborhood Table*. This table consists of two columns: (1) *ItemID*: a unique item identifier and (2) *ItemNeighbors*: a set of Key-Value pairs $\langle$*iid,simscore*$\rangle$ that contains every item iid that belongs to *ItemID* neighborhood. The Item Neighborhood Table is indexed by a primary key index created on the *ItemID* field. In case the neighborhood

Algorithm 1 ITEMCF-RECOMMEND

```
1:  /* load User Vector Table block by block in Memory */
2:  for each user u ∈ UserVector do
3:      UserItems ← List of User u rated items in ItemNeighborhood
4:      /* load Item Neighborhood Table block by block in Memory */
5:      for each item i ∈ ItemNeighborhood do
6:          ItemNeighbors ← List of item similar to item i in ItemNeighborhood
7:          if item i ∈ UserItems then
8:              r_{u,i} ← Rating that u gave to i
9:          else
10:             CandItems ← ItemNeighbors ∩ UserItems
11:             if CandItems not equal φ then
12:                 r_{u,i} ← Predict(u,i, UserItems, ItemNeighbors)
13:             else
14:                 r_{u,i} ← 0
15:         EMIT ⟨u, i, r_{u,i}⟩
```

size is large, RECDB employs the *Horizontal matrix representation* given the fact that it incurs less storage space for dense matrices. RECDB adopts the same storage layout for the user-user collaborative filtering algorithm, but in this case stores the *User Neighborhood table* instead of items.

For an incoming query, the query planner invokes the ITEMCF-RECOMMEND operator when the `USING` clause specifies the Item-Item Collaborative Filtering Algorithm (i.e., `ItemCosCF` or `ItemPearCF`). Algorithm 1 gives the pseudocode of the ITEMCF-RECOMMEND operator. The ITEMCF-RECOMMEND operator accesses both the *user vector table* (*UserVector*) and an *item neighborhood table* (*ItemNeighborhood*). The operator then returns a set of tuples S such that each tuple $s \in S$; $s = \langle u, i, r_{u,i} \rangle$ represents a user u, item i (unseen by user uid), and a rating $r_{u,i}$. ITEMCF fetches *UserVector* block by block to retrieve each tuple $\langle u; \{(i_1, r_{u,i_1}), ..., (i_m, r_{u,i_m})\}\rangle$. In a nested-loop fashion, the algorithm scans *ItemNeighborhood* block by block and saves the currently retrieved block in an in-memory buffer. If an item i is already rated by u, we set $r_{u,i}$ to the rating that u already assigned to i. In case u did not rate i, we first determine whether the set of similar items to i *ItemNeighbors* intersects the set of items rated by u (i.e., *UserItems*). If there is no overlap, the algorithm sets $r_{u,i}$ to 0. Otherwise, ITEMCF invokes `Predict()` to estimate the rating value $r_{u,i}$. Finally, ITEMCF emits the tuple $\langle u, i, r_{u,i} \rangle$ up in the query pipeline. The `Predict` procedure takes as input the user u, item i, the items rated by u *UserItems*, and the set of items similar to i *ItemNeighbors*. It then employs an aggregate function to estimate how much user u would like item i and returns a predicted rating accordingly. Example is given below:

Query 2. *Predict the rating that users would give to unseen items based on the Item-Based Collaborative Filtering Algorithm.*

```
Select R.uid,R.iid, R.ratingval From Ratings as R
Recommend R.iid To R.uid On R.ratingval Using ItemCosCF
```

By specifying the `Recommend` clause and the `Ratings` table in the `From` clause, RECDB figures that an ItemCosCF recommender, i.e., GeneralRec, is already created and initialized. Hence, the system accesses `GeneralRec` via the ITEMCF operator to perform the recommendation functionality.

Cost. The cost of ITEMCF-RECOMMEND is determined by the I/O cost incurred by the GetRecScore() function that is responsible for calculating the recommendation score for each item. The I/O cost of GetRecScore() function $||\mathcal{F}(U, I, R)||$ depends upon the recommendation algorithm defined for the underlying recommender R. Let α_u denotes the percentage of pages in the items table that consists of items unseen by the querying user u and $||I||$ represents the number of pages occupied by the items table I. Hence, the total scores evaluation and tuples reporting cost is $\alpha_u \times ||I|| \times ||\mathcal{F}(U, I, R)||$.

Finally, the output cost $\alpha_u \times ||I||$ represents the number of tuples reported as answer by the RECOMMEND operator.

Item	ItemFeatures
'Spartacus'	{⟨Feature1,0.5⟩;⟨Feature2,-0.7⟩;⟨Feature3,0.1⟩}
'Inception'	{⟨Feature1,0.4⟩;⟨Feature2,0.8⟩;⟨Feature3,-0.1⟩}
'The Matrix'	{⟨Feature1,0.5⟩;⟨Feature2,0.5⟩;⟨Feature3,0.6⟩}

(a) **Item Factor Table**

User	UserFeatures
'Alice'	{⟨Feature1,0.5⟩;⟨Feature2,-0.1⟩;⟨Feature3,0.1⟩}
'Bob'	{⟨Feature1,0.3⟩;⟨Feature2,0.7⟩;⟨Feature3,0.1⟩}
'Carol'	{⟨Feature1,0.5⟩;⟨Feature2,0.6⟩;⟨Feature3,-0.3⟩}
'Eve'	{⟨Feature1,-0.4⟩;⟨Feature2,0.1⟩;⟨Feature3,-0.1⟩}

(b) **User Factor Table**

Fig. 2: Matrix Factorization Model

2) User-User Collaborative Filtering Operator: To generate recommendation using user-user collaborative filtering, RECDB employs a variant of the RECOMMEND operator called USERCF. USERCF is similar to ITEMCF except that it accesses the following data structures: the *item vector table* (*ItemVector*) and the *user neighborhood table* (*UserNeighborhood*). The operator finally returns a set of tuples S such that each tuple $s \in S$; $s = \langle u, i, r_{u,i} \rangle$ represents a user u, item i (unseen by user uid), and a rating $r_{u,i}$.

3) Matrix Factorization Operator: To access matrix factorization models, RECDB is equipped with a variant of the RECOMMEND operator called MATRIXFACT. The MATRIXFACT operator accesses the following data structures: (1) *user factor table* (*UserFactor*): a table that contains the set of user vectors such that each user vector $p_u \in p$ denotes the weights that each user would assign to a set of item features (latent factors) and (2) an *item factor table* (*ItemFactor*): a table that consists of a set of item vectors such that each item vector $q_i \in q$ denotes the weights that qualifies how much each item belongs to a set of features (latent factors) (see Figure 2).

$$\min_{q^*, p^*} \sum_{(u,i) \in k} (r_{ui} - q_i^T . p_u)^2 + \lambda(||q_i||^2 + ||p_u||^2) \qquad (3)$$

To learn the matrix factorization model, the system uses techniques like singular value decomposition (SVD), stochastic gradient descent, alternating least square to minimize the regularized squared error (see Equation 3).

Similar to previous operators, MATRIXFACT also returns a set of tuples S such that each tuple $s \in S$; $s = \langle u, i, _{u,i} \rangle$ represents a user u, item i and a rating r. Algorithm 2 gives the pseudocode of the MATRIXFACT operator. In a block nested loop manner, MATRIXFACT scans *UserFactor* block by block to fetch the feature vector of each user u. Then, MATRIXFACT scans *ItemNeighborhood* block by block to retrieve the feature

Algorithm 2 MATRIXFACT-RECOMMEND

```
1: /* load User Features Table block by block in Memory */
2: for each user u ∈ UserFactorVector do
3:     uFeatures ← List of latent factors (features) learned for user u
4:     /* load Item Features Table block by block in Memory */
5:     for each item i ∈ ItemFactorVector do
6:         iFeatures ← List of latent factors (features) learned for item i
7:         r_{u,i} ← DotProduct(iFeatures, uFeatures)
8:         EMIT ⟨u, i, r_{u,i}⟩
```

vector for each item i. If an item i is already rated by u, we set $r_{u,i}$ to the rating that u already assigned to i. The algorithm calculates the dot product of both *uFeatures* and *iFeature* and that represents the value of the predicted rating $r_{u,i}$. ITEMCF emits the tuple $\langle u, i, r_{u,i} \rangle$ up in the query pipeline.

B. Query Pipeline Integration

The RECOMMEND operators are non-blocking (pipeline-able) database operators that follows the iterator model adopted by almost all existing relational database engines (i.e., PostgreSQL in our case). The non-blocking nature means that other operators in the query pipeline can receive results from the RECOMMEND operator before it is done with all of its predictions. Being a pipeline-able operator allows a seamless integration with other query operators in a database query processor. However, as the RECOMMEND operator only applies to a recommender and does not apply to normal database relations, it should always be pushed down to the bottom of the query pipeline. In this section, we discuss the integration of the RECOMMEND operator in the bottom of a query pipeline with selection, join, and ranking operators.

1) Selection: Two cases might happen when a selection predicate is applied to the recommendation answer: (1) *Case 1: uid or iid selection predicate*, where the selection predicate is applied to the user or item identifiers, and (2) *Case 2: ratingval selection predicate*, where the selection predicate is applied to the predicted rating value $ratingval$. RECDB deals with these two cases as follows: The following query gives an example of an *iid* selection predicate query over a recommendation result.

Query 3. *Predict the ratings that user* $uid = 1$ *would give to items 1 to 5 using the ItemCosCF algorithm.*

```
Select R.iid, R.ratingval From Ratings as R
Recommend R.iid To R.uid On R.ratingval Using ItemCosCF
Where R.uid=1 And R.iid In (1,2,3,4,5)
```

This kind of query is very frequent, where in many cases a user would like to only know the recommendation score for a specific item (e.g., a movie in Netflix) or for a set of few items (e.g., a set of few books in Amazon). A straightforward execution of such queries uses the query plan in Figure 3, which performs well only if the predictive selectivity is very low. For highly selective predicates (e.g., Query 3) above), the RECOMMEND operator performs lots of unnecessary work fetching all items data from disk and calculating their predicted rating scores, while only few items are needed.

Since in many cases, the predicate selectivity is very high; It is very common to generate recommendation for a single

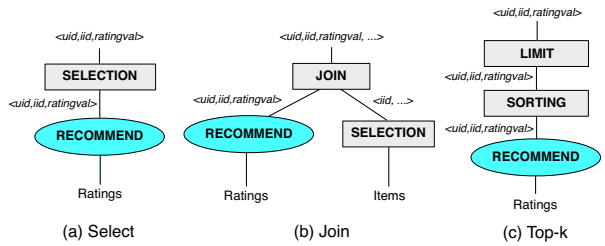


Fig. 3: Recommend Query Plans

user or predict the rating for only one item, RECDB employs a variant of the RECOMMEND operator, called FILTERRECOMMEND. Instead of calculating the predicted rating scores for all user/item pairs, the family of FILTERRECOMMEND operators takes the filtering predicate as input and prunes the predicted rating score calculation for those items that do not satisfy the filtering predicate.

2) Join: Query 4 in Section IV-B gives an example of a recommendation query, where the output of the recommender operator needs to be joined with another query to get the movie names instead of their identifiers and to only retrieve Action movies. This is a very common query in any recommender system. For example, Netflix and Amazon always return the item information, not the item identifiers. Also, a Netflix user may opt to receive movie recommendation for a certain movie genre.

Query 4. *Predict the rating that user* $(uid = 1)$ *would give to* action *movies.*

```
Select R.uid, M.name, R.ratingval From Ratings as R, Movies as M
Recommend R.iid To R.uid On R.ratingval Using ItemCosCF
Where R.uid=1 And M.iid = R.iid And M.genre='Action'
```

Figure 3(b) gives the query plan of the above query, where the ITEMCF-RECOMMEND operator is applied directly to the *GeneralRec* recommender. In the meantime, the `Movies` table goes to a selection operator (i.e., filter) with the predicate *"genre='Action'"* to pass only action movies. The output of the RECOMMEND operator is joined with the output of the selection operator using a traditional join operator with the predicate *"Movies.iid = GeneralRec.iid"* to come up with the movie names for action movies along with ratings produced from *GeneralRec*.

The straightforward plan for executing queries that combine recommendation and join (Figure 3b) may be acceptable only if there is no filter over the joined relation, or the filter has very low selectivity. Otherwise, if the filter is highly selective, the RECOMMEND operators will end up doing redundant work predicting the rating scores for all user/item pairs, 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, RECDB employs the JOINRECOMMEND operator. Besides the user u and a recommender R, the JOINRECOMMEND operator takes a joined database relation rel (e.g., `Movies`) as input, combines their tuples, and returns the joined result. Analogous to index nested

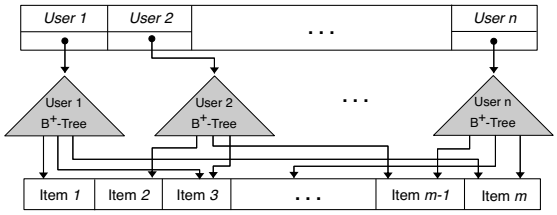


Fig. 4: RecScore Index Structure

Algorithm 3 INDEXRECOMMEND

```
1:  /* Phase I: Retrieve Users One by One in RecScoreIndex*/
2:  for each user u ∈ RecScoreIndex that satisfies uPred do
3:      RecTree_u ← Fetch user u RecTree pointer
4:      /* Phase II: Traverse RecTree to satisfy the ratingval predicate rPred */
5:      TreeNode←TRAVERSE(RecTree_u,rPred)
6:      /* Phase III: Fetch Items One by One at the leaf nodes of RecTree_u */
7:      for each item i ∈ FetchNextItem(TreeNode) do
8:          if item i satisfies iPred then
9:              r_{u,i} ← the predicted Rating of i stored in the node
10:             EMIT ⟨u, i, r_{u,i}⟩
```

loop join, JOINRECOMMEND employs the input relation rel as the outer relation. For each retrieved tuple $tup \in rel$, the algorithm calculates the predicted rating score for item i with iid equal to tup.iid in the same way it was calculated in the RECOMMEND algorithm. The algorithm then concatenates $\langle uid,iid,ratingval \rangle$ and tup and the resulting joined tuple $\langle tup,iid,ratingval \rangle$ is finally added to the join answer S. The algorithm terminates when there are no more tuples left in rel.

C. Pre-Computation

Recommendation applications are rather interactive and real time in nature which necessitates mitigating the recommendation generation latency. To further optimize query execution performance, RECDB pre-computes the predicted ratings for user/item pairs and save them in a data structure, named *RecScoreIndex*.

Data Structure. *RecScoreIndex* (see Figure 4) is a hash table where each entry is represented as $\langle u,RecTree_u \rangle$, as the user identifier and a pointer to a B$^+$-tree that indexes the pre-computed predicted rating scores of all items unseen by the user. The predicted rating scores are pre-computed based on the recommendation model *RecModel*. $Rectree_u$ is built for each user u, where the predicted rating score *ratingval* is the key $Rectree_u$ leaf nodes contain pointers to the corresponding items. That means that items in the leaf nodes of $Rectree_u$ are sorted in a descending order of their predicted rating value.

Operator. RECDB accesses *RecScoreIndex* using an optimized recommendation operator, named INDEXRECOMMEND. The optimized operator reduces the amount of work performed by the recommendation operators by directly accessing the pre-computed predicted rating scores saved in *RecScoreIndex*. Algorithm 3 gives the pseudocode of the INDEXRECOMMEND query operator. Besides the ratings table, the algorithm takes as input a user predicate (*uPred*), item predicate (*iPred*), and *ratingval* predicate (*rPred*). The algorithm runs in three main phases: (1) *Phase I: User ID Filtering*: In this phase, INDEXRECOMMEND fetches each user u that satisfies *uPred* by looking up the given user IDs in the *RecScoreIndex* hash table. (2) *Phase II: Rating Value Filtering*: In this phase, INDEXRECOMMEND traverses the $RecTree_u$ corresponding to each user u retrieved in the first phase to satisfy the ratingVal predicate $rPred$ (3) *Phase III: Item ID Filtering*: In this phase, the algorithm fetches items one-by-one in the leaf level of $RecTree_u$. The algorithm filters out those items that do not satisfy the item predicate (*iPred*). Finally, INDEXRECOMMEND emits each tuple $\langle u,i,r_{u,i} \rangle$ that passes the three phases up in the query pipeline.

Query 5 needs to join the `GeneralRec` 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. The following two plans are deemed correct: (1) Apply the JOINRECOMMEND operator on recommender `GeneralRec` and table `Movies` first and then perform a traditional top-k operation on the join result. (2) If a *RecScoreIndex* is maintained, the system may apply the INDEXRECOMMEND operator to retrieve items in sorted order and then perform the join operation on the returned items to retrieve *Action* movies.

Query 5. *Recommend the top* 5 *Action movies to user* $uid = 1$ *using the SVD Algorithm.*

```
Select M.name, R.ratingval From Ratings as R, Movies M
Recommend R.iid To R.uid On R.ratingval Using SVD
Where R.uid=1 And M.iid=R.iid And M.genre='Action'
Order By R.ratingval Desc Limit 5
```

D. Caching Recommendation

Most recommendation applications (e.g., Amazon, Google News) deal with large user base and large pool of items. For the system to scale up, i.e., accommodate more users and items, it must minimize the maintenance cost and reduce the overall storage occupied by the recommender data structure. Maintaining the *RecScoreIndex* every time maintenance is triggered for *RecModel* exhibits the lowest query response time because each incoming recommendation query may directly access the pre-computed recommendation scores stored in *RecScoreIndex*. However, the maintenance cost as well as the storage overhead incurred by materializing all *RecScoreIndex* entries may lead to a severe scalability bottleneck, especially with large amounts of user/item pairs.

Main idea. Since recommendation queries are personalized for each user, the system collects statistics about the demand of each user, and leverages these statistics to take the materialization decision. RECDB stores those ⟨user,item,ratingval⟩ triplets in *RecScoreIndex* that correspond to highly demanding users. That mitigates the overall recommendation generation latency as queries directly access the pre-computed predicted ratings in *RecScoreIndex*. Since the generated recommendation represents a set of items, the system also collects statistics that quantifies each item's consumption rate. The system then determines whether an item is highly consumed, i.e., frequently rated/updated, and only maintains entries in *RecScoreIndex* that correspond to those highly-consumed items.

Algorithm 4 Caching Algorithm

```
1:  U' ← All users in recommender T with TS_u larger than TS_mat
2:  I' ← All items in recommender T with TS_i larger than TS_mat
    /* STEP 1: Statistics Maintenance */
3:  for each item i ∈ I' do
4:      Maintain P_i ← UC_i / (TS_now − TS_init)
5:      If P_i > P_MAX then Maintain P_MAX ← P_i
6:  for each user u ∈ U' do
7:      Maintain P_i ← QC_u / (TS_now − TS_init)
8:      If D_u > D_MAX then Maintain D_MAX ← D_u
    /* STEP 2: Materialization Decision */
9:  for each user/item pair {u,i} ∈ {U' × I'} do
10:     if i is unseen by u then
11:         Hot_{u,i} ← (D_u / D_Max) × (P_i / P_Max)
12:         if Hotness Ration Hot_{u,i} ≥ HOTNESS-THRESHOLD then
13:             Append user/item pair {u,i} to Admission List
14:         else
15:             Append user/item pair {u,i} to Eviction List
```

User	QC_u	TS_u	D_u
Alice	100	10	100/(15-10)= 20
Bob	10	12	10/(15-10)≈ 2

(a) **Users Histogram**

Item	UC_i	TS_i	P_i
Spartacus	1000	12	1000/(15-10)≈ 200
Inception	10	12	10/(15-10)≈ 2
The Matrix	100	10	100/(15-10)= 20

(b) **Items Histogram**

User u	Item i	$Hot_{u,i}$
Alice	Spartacus	(20/20) × (200/200) = 1
Alice	Inception	(20/20) × (2/200) = 0.01
Alice	The Matrix	20/20 × (20/200) 0.01
Bob	Spartacus	(2/20) × (200/200) 0.1
Bob	Inception	(2/20) × (2/200) = 0.001
Bob	The Matrix	(2/20) × (20/200) ≈ 0.01

(c) **User/Item Pair Hotness Ratio**

TABLE I: Materialization Manager Example

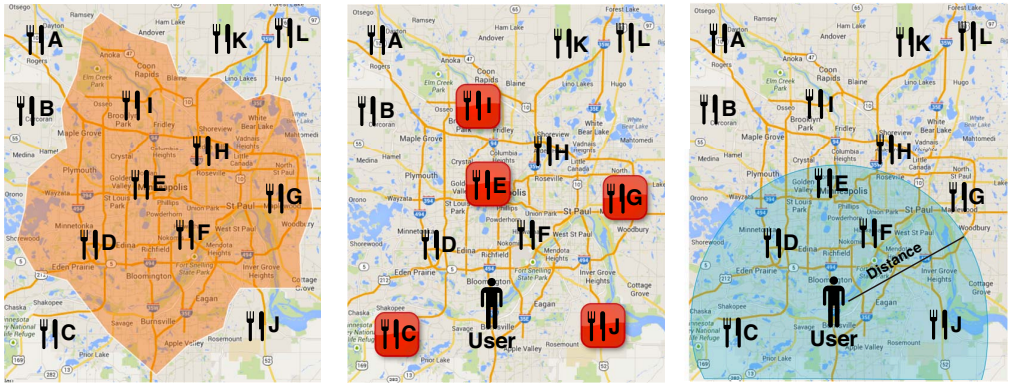
1) Statistics: To take the materialization decision, RECDB maintains the following statistics for each created recommender T: (1) *Users Histogram:* A hash table indexed by the user ID that contains the following fields for each user $u \in U$: (a) *Queries Count (QC_u)*: represents the number of issued recommendation queries by u since T is created. (b) *Query TimeStamp TS_u*: time stamp of the last recommendation query issued by u. (c) *User Demand Rate (D_u)*: represents the rate of queries issued by user u. (2) *Items Histogram:* A table hashed by the item ID and contains the following fields for each item i: (a) *Updates Count UC_i*: represents the number of updates applied, i.e., ratings insertion, to item i in recommender T since the recommender is created. (b) *Update TimeStamp TS_i*: time stamp of the last update transaction performed over item i, (c) *Item Consumption rate (P_i)*: represents the rate of updates performed on item i. (3) *Maximum User Demand (D_{MAX})*: the maximum user demand rate D_u of a user u among all users in recommender T. (4) *Maximum Item Consumption (P_{MAX})*: the maximum item consumption rate P_i of an item i among all items in recommender T.

2) Caching Algorithm: Using the statistics maintained for a recommender T, the main role of the cache manager is determining which user/item/rating triplets need to be cached in *RecScoreIndex*. To achieve this goal, the cache manager runs asynchronously every fixed period of time (e.g., 5 mins) in the background and performs the following steps:

STEP 1: Statistics Maintenance. It first retrieves the set of users U' such that each user $u \in U'$ has (TS_u) larger than the last time the cache manager was invoked. We also retrieve the set of items I' such that each item $i \in I'$ has (TS_i) larger than the last time the cache manager was executed. The cache manager then calculates *User Demand Rate* D_u for each user $u \in U'$, as follows: $D_u = \frac{QC_u}{TS_{now} - TS_{init}}$; such that TS represents the current system timestamp. In addition, RECDB maintains *Maximum Demand* by setting D_{MAX} to D_u in case D_u value is larger than current D_{MAX} value. Therefore, the materialization manager calculates *Item Consumption Rate* P_i for each item $i \in I'$, as follows: $P_i = \frac{UC_i}{TS_{now} - TS_{init}}$; such that TS represents the current system timestamp. The system also maintains *Maximum Demand* by setting P_{MAX} to P_i if P_i value is larger than current P_{MAX}.

STEP 2: Decision Making. This step leverages the maintained statistics to decide whether the entry corresponding to a user/item pair needs to be materialized. To this end, the cache manager maintains two in-memory lists: (1) *Admission List*: a list that contains the user/item pairs that requires materialization, and (2) *Eviction List*: a list that contains those user/item pairs that needs to be dematerialized. For each user/item pair $\{u,i\}$ such that $u \in U'$ and $i \in I'$, we calculate the hotness ratio $Hot_{u,i}$ ($0 \geq Hot_{u,i} \leq 1$), as follows: $Hot_{u,i} = \frac{D_u}{D_{Max}} \times \frac{P_i}{P_{Max}}$; such that $\frac{D_u}{D_{Max}}$ and $\frac{P_i}{P_{Max}}$ represent the normalized user demand rate and item consumption rate, respectively. The hotness ratio $Hot_{u,i}$ determines whether the entry, corresponding to $\{u,i\}$, is eligible for materialization. If $Hot_{u,i}$ is greater than or equal to a system parameter HOTNESS-THRESHOLD (value between 0 and 1), we add the user/item pair $\{u,i\}$ to the admission list, otherwise we append them to the eviction list. The HOTNESS-THRESHOLD exhibits a tradeoff between: (1) Query latency, and (2) System Scalability (Storage Overhead and Maintenance Cost). In other words, when HOTNESS-THRESHOLD is equal to 0, RECDB tends to fully materialize all *RecScoreIndex* entries, and when set to 1, *RecScoreIndex* is not materialized at all.

Example. Table I depicts an example that illustrated the materialization manager dynamics triggered for a cell C. The table gives Cell C *Users Histogram* and *Items Histograms* at the time the materialization manager is invoked at timestamp 15. As it turns out from the table, *Users Histogram* contains two users: *Alice* and *Bob*, and *Items Histogram* contains three items (i.e., Movies): *Spartacus*, *Inception*, and *The Matrix*. The *User Demand* for *Alice* is calculated as D_{Alice} = QC_{Alice} / (15-TS_{Alice}) = 20. Similarly, D_{Bob} is equal to $\approx$ 3.33. The item consumption rate $P_{Spartacus}$ for *Spartacus* movie is evaluated as $UC_{Spartacus}$ / (15-10) = $\approx$ 200, and the same calculation is applied to other movies. Table I also manifests the hotness ratio calculated by the materialization manager for the user/item pairs. Assume that all movies are unseen by both *Alice* and *Bob* and *RecScoreIndex* only contains the entry t_1 that corresponds to user *Bob* and movie *Inception*. Let HOTNESS-THRESHOLD be set to 0.5, in this case the entry t_1 is added to cell C *Eviction List* as $Hot_{Bob,Inception} = \approx$ 0.001 is less than HOTNESS-THRESHOLD. In contrast, entry t_2,



(a) ST_Contains() with Recommendation (b) ST_Distance() with Recommendation (b) ST_DWithin() with Recommendation

Fig. 5: POI Recommendation Examples

corresponding to user *Alice* and movie *Bob*, is added to the *Admission List*.

When maintenance is triggered for *RecModel*, the system retrieves all user/item pairs in the *Eviction List* and batch deletes all corresponding entries in *RecScoreIndex*. The system also retrieves the user/item pairs in the *Admission List* and batch inserts them in *RecScoreIndex*. Finally, RECDB maintains the recommendation score $RecScore$ for all materialized entries.

V. CASE STUDY

Applications like Yelp and Google maps provide tools for their users to express their opinions over visited items, e.g., restaurants (see Figure 5). That motivated the use of recommender systems to suggest Point-Of-Interests (POIs) to end-users. Consider the following scenarios:

Scenario 1. *Alice plans to visit San Diego to attend the IEEE ICDE 2017 conference. To plan her trip, she searches for Hotel recommendation in the San Diego area that are liked (highly rated) by other users who have similar taste to her.*

In Scenario 1, the system first retrieves hotels that lie within the San Diego area. Therefore, it predicts the rating that Alice would give to such hotels based on the opinions of other users similar to her. In Scenario 2 (below), the system calculates the distance between Alice's current location and all hotels. It also predicts a rating for each hotel based on its similarity to restaurants already seen by Alice and finally ranks hotels based on both the spatial proximity and predicted rating.

Scenario 2. *When Alice arrives to her booked hotel in San Diego, she looks for nearby restaurant recommendation that are close to her current location and are also similar to the restaurants she liked before.*

By integrating RECDB with PostGIS, a geospatial extension of PostgreSQL, users can spatially filter the recommended POIs to only return those POIs that resides in a specified urban area. That also allows users to rank POIs based on both its personalized recommendation score and spatial proximity to the querying user.

A. POI Recommenders

The following SQL creates a recommender, named `POI-ItemCosCF-Rec`, on the data stored in the HotelRatings table. `POI-ItemCosCF-Rec` can predict the rating that users would give to POIs based on the ItemCosCF algorithm.

Recommender 2. POI-ItemCosCF-Rec*: an ItemCosCF recommender created on the HotelRatings table.*

```
Create Recommender POI-ItemCosCF-Rec On HotelRatings
Users From uid Item From iid Ratings From ratingval Using ItemCosCF
```

The following SQL creates another recommender, named `POI-UserPearCF-Rec`, on the the RestaurantRatings table. It can predict how much users would like POIs based on the SVD recommendation algorithm.

Recommender 3. POI-UserPearCF-Rec*: a UserPearCF recommender created on the RestaurantRatings table.*

```
Create Recommender POI-UserPearCF-Rec On RestRatings
Users From uid Item From iid Ratings From ratingval Using SVD
```

B. Generating POI Recommendation

After initializing the POI recommenders, users may issue location-aware recommendation queries. For instance, to produce POI recommendation as given in Scenario 1, users may issue the following SQL queries:

Query 6. *Predict the rating that user 1 would give to Hotels that exist in the 'San Diego' urban area.*

```
Select H.name, R.ratingval
From HotelRatings as R, Hotels as H, City as C
Recommend R.iid To R.uid On R.ratingVal Using ItemCosCF
Where R.uid=1 AND R.iid=H.vid AND C.name = 'San Diego'
            AND ST_Contains(C.geom, H.geom)
```

In this case, RECDB uses the `POI-ItemCosCF-Rec` recommender, which was created before using a `CREATE RECOMMENDER`. Query 6 predicts the ratings that user 1 would

TABLE II: Recommender model building time

Init. Time	ItemCosCF	ItemPearCF	SVD
MovieLens	2.24 sec	2.12 sec	15.62 sec
LDOS-CoMoDa	0.17 sec	0.07 sec	0.4 sec
Yelp	6.26 sec	8.03 sec	32.01 sec

give to unseen hotels using the RECOMMEND operator. However, the query leverages the ST_Contains() function (provided by PostGIS) to predict a rating only for those hotels that lie within the extent of the 'San Diego' urban area.

Query 7. *Recommend 10 restaurants to user 1 that lie within 500 miles of her location based on the UserPearCF algorithm.*

```
Select V.name, V.address From Ratings as R, Restaurants as V
Recommend R.iid To R.uid On R.ratingVal Using UserPearCF
Where R.uid=1 AND R.iid=V.vid AND ST_DWithin(ULoc, V.geom, 500)
Order By R.ratingVal Desc Limit 10
```

Query 7 harnesses the POI-UserPearCF-Rec recommender, created earlier and initialized inside RECDB, to predict the ratings that user 1 would give to restaurants that lie within 500 meters range from the user current spatial location. To this end, Query 7 invokes the ST_DWithin() geometry function to filter out restaurants that are not spatially within 500 meters from the user location.

Query 8. *Recommend top-10 restaurants that are close to user 1 current location based on the UserPrearCF algorithm.*

```
Select V.name, V.address From Ratings as R, Restaurants as V
Recommend R.iid To R.uid On R.ratingVal Using UserPearCF
Where R.uid=1 AND R.iid=V.vid
Order By CScore(R.ratingVal, ST_Distance(V.geom, ULoc)) Desc Limit 3
```

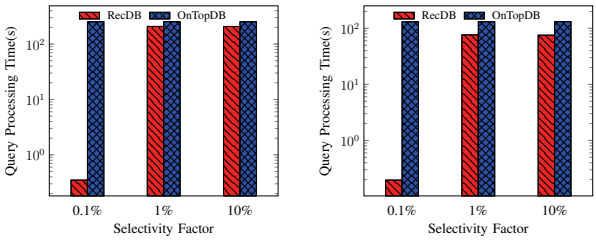
Query 8 combines both the predicted rating score calculated using the UserPearCF algorithm and the spatial proximity score using the PostGIS ST_Distance() function. The query finally returns the Top-3 restaurants.

VI. EXPERIMENTS

This section presents preliminary experimental results that compares the query execution performance of: (1) RecDB: A PostgreSQL 9.2 extension that implements a prototype of RECDB. (2) OnTopDB: a system provides personalized recommendation by building the recommendation functional layer on top of PostgreSQL 9.2. All experiments run on a machine with 8 CPUs (3.5 GHz per core), 32 GB memory, and 2 TB magnetic disk with PostgreSQL 9.5 installed.

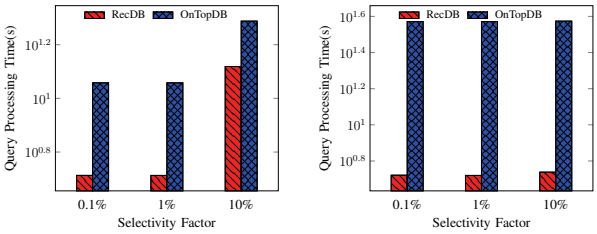
Datasets. We use three real datasets i.e. MovieLens, LDOS-CoMoDa, and Yelp to analyze the performance of both systems. MovieLens [2] contains 100K ratings for 1,682 movies by 943 users. The data set consists of three tables: *users* (userid, name) with each tuple consisting of a userid and name attributes. *movies* (movieid, name, genre) in which information about movies are stored, and *ratings* (userid, itemid, rating) with each tuple representing how much a user liked a movie.

[2]http://www.grouplens.org/node/73



(a) ItemCosCF (b) SVD

Fig. 6: Query time (MovieLens)



(a) ItemCosCF (b) SVD

Fig. 7: Query time (Yelp)

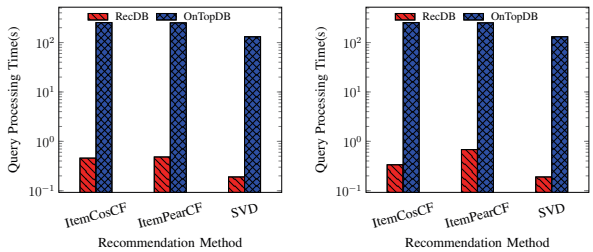
LDOS-CoMoDa is also a movie recommender dataset that contains 2297 ratings for 785 movies by 185 users. The dataset consists of three tables: *itemTiltes* (Moiveid, Name, genre, director, actor), *users* (userid, age, city, location), and *ratings* (userid, itemId, rating) that contains the users' reviews for a movie. The Yelp dataset represents a subset of the Yelp dataset challenge[3]. It consists of 1,446 businesses (aka, POI in Section V) reviewed by 3,403 users and the total number of reviews is 126,747. For each dataset, we created three different recommenders using three different recommendation algorithms, Item-based Collaborative Filtering Pearson Correlation (ItemPearCF) and Cosine Similarity (ItemCosCF) as well as singular value decomposition (SVD). Table II summarizes the recommender model building time.

A. Impact of Query Selectivity

This section compares the performance of RecDB against OnTopDB for queries that combine the recommendation operator and selection operator. The experiment evaluates the two approaches using queries with 0.1%, 1%, and 10% query selectivity factors. The selectivity factor represents the ratio of selected recommended items over the total number items in the dataset.

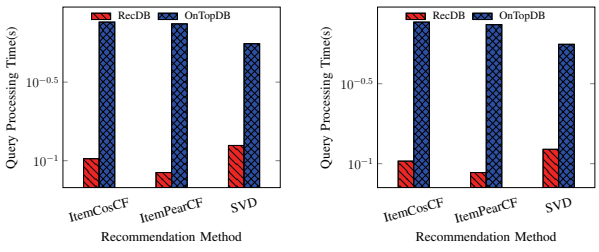
Figure 7 depicts the query processing time of both systems while varying selectivity factor of the issued query. As it turns out from the figures, RecDB outperforms OnTopDB in processing recommendation queries that involves a selection predicate. However, the performance of RecDB is not quite different than that of OnTopDB for high selectivity factors, i.e., 10%. This behavior is more obvious in the MovieLens dataset given the fact that the number of movies that the user never rated before is way higher. That leads to lot of time spent

[3]https://www.yelp.com/dataset_challenge



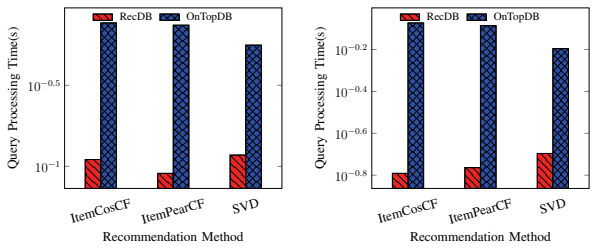
(a) One-way join (b) Two-way join

Fig. 8: Join query time (MovieLens)



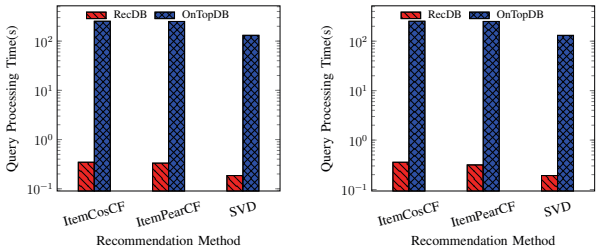
(a) K = 10 (b) K = 100

Fig. 10: Top-K recommendation query time (MovieLens)



(a) One-way join (b) Two-way join

Fig. 9: Join query time (Ldos-CoModa)



(a) K = 10 (b) K = 100

Fig. 11: Top-K recommendation query time (Ldos-CoModa)

on predicting the recommendation score for a high number of items.

On the other hand, the experimental results unequivocally shows that RecDB outperforms OnTopDB by more than two orders of magnitude especially for queries with high selectivity factor (0.1%). RecDB is more efficient due to the fact that items are recommended to only those users which are specified in the where clause of recommended query. On the other hand, OnTopDB returns the ratings to a limited set of items after predicting the ratings for all items that exist in the items table. In other words, one can observe that the query processing time is directly proportional to the query selectivity factor. The rationale is that with large numbers of users and/or items, the number of rating prediction operation increases and hence the query processing time also increases.

B. Studying the Join + Recommendation case

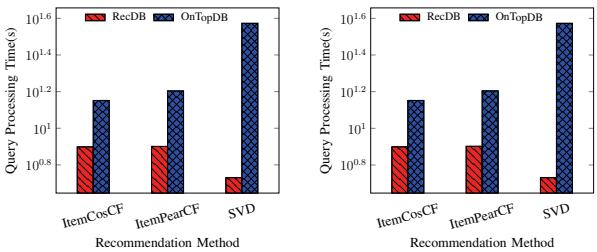
This section compares the performance of RecDB against OnTopDB for queries that combin the join operator and the recommendation operator. For profound understanding, we performed experiments on datasets with different number of joins. The query processing time taken by both applications for joins queries on movielens and LDOS-CoMoDa dataset are given in figure 8 and 9 respectively.

Figure 8 and 9 compares the processing time of both RecDB and OnTopDB using different recommendation algorithms. As it turns out in the figures, the results clearly shows that RecDB achieves up to two-orders of magnitude less query response time than OnTopDB in processing join/recommend queries. Unlike RecDB, OnTopDB processes a recommendation query for all the users before recommending the items to a particular user and due to this it takes more time to execute against RecDB. Furthermore, RecDB still achieves the same gain over OnTopDB even when the number of joins in the query increases (i.e., Two-Way join). That happens due to the fact that RecDB filters out items way before it performs the recommendation score prediction step or applies any of the join operators. Furthermore, RecDB only predicts the recommendation score for those tuples that are guaranteed to satisfy the join predicate.

C. Impact of Pre-Computation and Caching

This section studies the impact of pre-computing the recommendation scores on the system performance. To achieve that, we compare the performance of both applications for top-k recommendation queries while varying k (k=10 and k=100). In these experiments, we issue a top-k recommendation query for a set of randomly selected users. The query recommends top-k items based on ratings using different recommendation algorithms. Figure 10 and 12 show the processing time taken by RecDB and OnTopDB to process top-k recommendation queries on MovieLens, LDOS-CoMoDa, and yelp datasets, respectively.

The results show that RecDB consistently achieves up to two orders of magnitude better performance compared to OnTopDB. Similarly, results from LDOS-CoMoDa and Yelp datasets show that RecDB outperforms OnTopDB. This is due to the fact that OnTopDB needs to perform the recommendation score prediction step for all items, sorts them and then select the top-k. On the other hand, RecDB utilizes accesses the pre-computed predicted rating in the RecScoreIndex using the IndexRecommend operator. Furthermore, RecDB leverages the items saved in the recommendation cache to further reduce the query response time. That way, Top-k recommendation query processing becomes more efficient even when increasing the value of k.



(a) K = 10 (b) K = 100

Fig. 12: Top-K recommendation query time (Yelp)

VII. Conclusion and Future work

RecDB adopts an *In-Database* approach that pushes the recommendation functionality inside a relational database engine to achieve the following properties: (1) *Usability*: Crafted inside a relational DBMS, the system is easily used and configured so that a novice application developer can declaratively define a variety of recommenders that fits the application needs in few lines of SQL code. (2) *Seamless Integration*: The system is able to seamlessly integrate the recommendation functionality in the traditional `SPJ`, i.e., `SELECT`, `PROJECT`, `JOIN`, query pipeline to execute interactive recommendation queries. RecDB encapsulates the recommendation functionality into new query operators. Being a query operator allows the recommendation functionality to be a part of a larger query plan that includes other database query operators, e.g., selection, join. That also allows for a myriad query optimization techniques that can be integrated to speed up the recommendation generation process. (3) *Efficiency*: RecDB provides online recommendation to a high number of users over a large pool of items. It pre-computes the predicted ratings and save them in data structure, that is leveraged by the query planner, to reduce the recommendation generation latency. In the future, we plan to consider more recommendation methods that include tensor factorization and deep learning.

VIII. Acknowledgement

Dr. Sarwat' s research is supported by the National Science Foundation under Grant IIS 1654861. Dr. Mokbel' s research is supported by NSF grants IIS-0952977, IIS-1218168, IIS-1525953, CNS-1512877.

References

[1] Z. Abbassi and L. V. S. Lakshmanan. On Efficient Recommendations for Online Exchange Markets. In *Proceedings of the IEEE International Conference on Data Engineering, ICDE*, 2009.

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

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

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

[5] 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.

[6] G. Chatzopoulou, M. Eirinaki, S. Koshy, S. Mittal, N. Polyzotis, and J. S. V. Varman. The querie system for personalized query recommendations. *IEEE Data Eng. Bull.*, 34(2):55–60, 2011.

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

[8] 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.

[9] M. D. Ekstrand, M. Ludwig, J. A. Konstan, and J. T. Riedl. Rethinking the recommender research ecosystem: reproducibility, openness, and lenskit. In *Proceedings of the ACM Conference on Recommender Systems, RecSYS*, 2011.

[10] Z. Gantner, S. Rendle, C. Freudenthaler, and L. Schmidt-Thieme. MyMediaLite: a free recommender system library. In *Proceedings of the ACM Conference on Recommender Systems, RecSYS*, 2011.

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

[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. *Proceedings of the International Conference on Very Large Data Bases, VLDB*, 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, 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.

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

[19] 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.

[20] 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.

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

[22] S. B. Roy, S. Thirumuruganathan, G. Das, S. Amer-Yahia, and C. Yu. Exploiting Group Recommendation Functions for Flexible Preferences. In *Proceedings of the IEEE International Conference on Data Engineering, ICDE*, 2014.

[23] 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.

[24] M. Sarwat, J. Avery, and M. F. Mokbel. RecDB in Action: Recommendation Made Easy in Relational Databses. In *Proceedings of the International Conference on Very Large Data Bases, VLDB*, 2013.

[25] 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*, 2013.

[26] H. Su, K. Zheng, J. Huang, H. Jeung, L. Chen, and X. Zhou. CrowdPlanner: A Crowd-Based Route Recommendation System. In *Proceedings of the IEEE International Conference on Data Engineering, ICDE*, 2014.

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

[28] H. Yin, B. Cui, J. Li, J. Yao, and C. Chen. Challenging the Long Tail Recommendation. *Proceedings of the International Conference on Very Large Data Bases, VLDB*, 5(9):896–907, 2012.