REQUEST: A Query Language for Customizing Recommendations

Gediminas Adomavicius
Department of Information and Decision Sciences
Carlson School of Management
University of Minnesota
gedas@umn.edu

Alexander Tuzhilin
Information, Operations & Management Sciences Department
Stern School of Business
New York University
atuzhili@stern.nyu.edu

Rong Zheng
Information, Operations & Management Sciences Department
Stern School of Business
New York University
rzheng@stern.nyu.edu

Abstract

Initially popularized by Amazon.com, recommendation technologies have become widespread over the past several years. However, the types of recommendations available to the users in these recommender systems are typically determined by the vendor and therefore are not flexible. In this paper we address this problem by presenting the recommendation query language REQUEST that allows users to customize recommendations by formulating them in the ways satisfying personalized needs of the users. REQUEST is based on the multidimensional model of recommender systems that supports additional contextual dimensions besides classical *User* and *Item* dimensions and also OLAP-type aggregation and filtering capabilities. The paper also presents a recommendation algebra, shows how REQUEST recommendations can be mapped into this algebra, and analyzes the expressive power of the query language and the algebra. The paper also shows how users can customize their recommendations using REQUEST queries through a series of examples.

Keywords: personalization, recommender systems, recommendation query language, recommendation algebra.

1. Introduction

Recommender systems represent an important class of personalization technologies that help users to deal with information overload in e-commerce and numerous other applications. There has been much work done in the area of recommender systems over the past decade since the introduction of the first papers on the subject [12, 21, 22], especially after these technologies were popularized by Amazon and Netflix, as well as after the establishment of the $1,000,000 Netflix Prize Competition that attracted over 43,000 contestants from 180 countries [6]. A recent survey of the rapidly growing field of recommender systems can be found in [3].

Most of the work in recommender systems focuses on a two-dimensional paradigm of recommending items to users or users to items (e.g., books to customers or customers for books). Although there are different types of approaches to deriving recommendations, including the ranking- [10] and market-basket-analysis-based [18], the majority of the academic work in recommender systems and implementations of commercial systems, including Amazon and Netflix, focuses on the *rating-based* approach [3], where recommendations use explicit or implicit ratings provided by the end-users.

Rating-based approaches are usually classified into *content-based*, *collaborative*, and *hybrid* [7]. In *content-based* recommendation methods, rating $R(u,i)$ of item i for user u is typically estimated based on the ratings $R(u,i')$ assigned by the same user u to other items i' that are "similar" to item i in terms of their *content*. For example, in order to recommend movies to user u, the content-based approach tries to understand user preferences by analyzing commonalities among the content of the movies user u has rated highly before. Then, only the movies that have a high degree of similarity with customer's past preferences are recommended.

Collaborative recommender systems try to predict rating $R(u,i)$ of item i for user u based on how other "similar" users u' previously rated item i. Here "user similarity" is defined in terms of the distance between the ratings users u and u' assigned to the items that both of them rated, the most popular types of distance metrics being correlation- and cosine-based measures between two rating vectors [3]. Then collaborative filtering methods recommend those items to the user that she has not rated yet and that were

highly rated by the similar users.

Content and collaborative methods can be combined into a *hybrid* approach in several different ways [7]. One popular way to combine them is by learning and maintaining user profiles based on the content analysis of the items preferred by the users, and then directly comparing the resulting profiles to determine similar users in order to make collaborative recommendations. Other types of hybrid methods are also possible and are described in [3].

Although the traditional two-dimensional user/item paradigm described above is suitable for some applications, such as recommending books and music CDs, it is significantly less suitable for the "context-rich" applications, such as traveling or shopping applications. For example, when recommending vacations to travelers, one would likely recommend a different vacation to a customer in the winter than in the summer, i.e., the time-of-travel context is clearly important when making recommendations. Similarly, when recommending groceries, a "smart" shopping cart [26] needs to take into account not only information about products and customers, but also such information as shopping date/time, store, who accompanies the primary shopper, products already placed into the shopping cart, and its location in the store. Clearly, the two-dimensional paradigm of classical recommender systems is less suitable for these applications.

To provide better recommendations in such contextually rich applications, one may need to consider other dimensions besides Item and User. For example, when Netflix or any on-demand movie provider recommends movies, it may consider such additional dimensions as Time when the movie was seen, Company in which the movie was seen (e.g., alone, with friends, parents, etc.), and Place in which it was seen (e.g., in the theater or at home). A completely different movie may be recommended by Netflix to a student when he wants to see it on a Saturday night with his girlfriend in a movie theater than when he wants to see it on Thursday evening with his parents at home. In [4, 5] we proposed a new *multidimensional* approach to recommender systems where we incorporated multiple dimensions and the OLAP-based cubes of ratings into the recommendation model. To estimate missing ratings in multidimensional cubes, we proposed the *reduction-based* method in [4] and the *heuristic-based* and

model-based methods in [2].

However, the multidimensional approach described in [4] and the classical two-dimensional recommendation methods have one significant limitation in common. These methods are *hard-wired* by the developers into the recommender systems, are inflexible and limited in their expressiveness, and, therefore, neglect some possible needs of the users. For example, a typical recommender system would recommend the top *k* items to a user, or the best *k* users for a product. This situation is quite limited, especially in multidimensional settings, where the number of possible recommendations increases significantly with the number of dimensions [5]. Therefore, there is a need to empower end-users and other stakeholders by providing them with the tools for expressing recommendations *that are of interest to them* [3, 15]. For example, Jane Doe may need a recommendation for the best two *dates* to go on vacation to Jamaica with her boyfriend. Also, Netflix or an on-demand movie service, such as provided by the Time Warner Cable, can envision a web-based interface to a multidimensional cube of ratings that lets the users express the recommendations that are of interest to them or automatically tailors recommendations based on a given context, such as the time of day or the day of week. For example, a certain user (e.g., Tom) may seek recommendations for him and his girlfriend of top 3 movies and the best times to see them over the weekend, and he enters this request into the recommender system via the web-based interface. Such query-based recommendation applications are not limited to on-demand movies but are relevant to a broad range of recommendation applications, including retailing, financial, travel and other applications. Furthermore, we believe that flexible recommendation capabilities would be appealing to a variety of different users, and not just to the end-users who are direct recipients of recommendations. For example, such functionality would be useful to the analysts of a company providing recommendation services, who may want to take advantage of all the knowledge that their recommender system holds and analyze it from a variety of different perspectives (“show me the top 2 movie genres for each user age bracket”, etc.).

One tool for expressing such requests is a recommendation language that is similar to how database users use query languages to retrieve information from databases. In fact, one may try to use SQL for this

purpose, and the above recommendation for Tom can be specified in SQL as

```
SELECT R.MovieId, R.TimeId, R.UserId, R.CompanionId, AVG(R.PersonalRating)
FROM MovieRecommender R, User U, Time T, Companion C
WHERE R.UserId = U.UserId AND R.TimeId = T.TimeId AND R.CompanionId = C.CompanionId
        AND U.Name = "Tom" AND T.TimeOfWeek = "weekend" AND C.Type = "Girlfriend"
GROUP BY R.MovieId, R.TimeId, R.UserId, R.CompanionId
```

where User and Companion are the relations storing information about customers and different types of companions, MovieRecommender is the ratings table, and Time is the temporal dimension table. Although "doable," this SQL query and, more generally, SQL at large would have the following problems when used for recommendation purposes. First, notice that SQL does not *exactly* provide the requested recommendation: it returns the list of tuples (movies, times to see them, users, etc.), but *does not* specify *what* is recommended to *whom* and does not provide the top 3 recommended movie/time pairs. More generally, as it will be shown in the paper, there are certain recommendations that cannot be expressed in SQL (but can in the language proposed in this paper). Second, SQL is a comprehensive, general-purpose query language and, therefore, many of the possible SQL queries do not represent recommendations. Therefore, in order to help the end-user formulate recommendations correctly and meaningfully, one may want to impose elaborate constraints on SQL to be able to restrict the language for the recommendation task. We have studied this issue and realized that it is very hard to develop a simple, elegant and intuitive system of such constraints for SQL. A better alternative would be to introduce a language that is directly defined on top of the "native" multidimensional recommendation model. Third, the above SQL query is fairly cumbersome: it constitutes a join of four relational tables, has 6 conditions in the WHERE clause, has the GROUP BY statement and the aggregation function AVG. Clearly, there should be a better and a more intuitive way to express this simple type of recommendation, and this observation served as a direct motivation for developing a special-purpose recommendation language. This necessity to replace cumbersome SQL queries with more elegant and intuitive formulations grows substantially for the significantly more complex recommendations, such as the ones presented in Section 3. Fourth, this cumbersomeness may have not only a cognitive effect on the users writing queries, but could possibly also affect query performance in some cases, since processing multiple join queries can be a very time-

consuming operation. In summary, the above issues can be attributed to the *task and model mismatch.* SQL is a general-purpose query language, which makes it a less intuitive and a less useful tool for users in the "vertical" application domain of recommender systems, where SQL may not have some specialized capabilities important for recommender systems. Also, SQL is based on the relational data model, and multidimensional recommendations on the multidimensional model [4] would need to be mapped into the relational model to support SQL queries, which leads to various translation problems. To avoid these issues, it is advantageous to develop a specialized recommendation language based on the characteristics of the application domain that also supports the multidimensional recommendation model, as we do in the paper.

To provide flexible and user-driven recommendations and to address the previously specified issues with using SQL as a recommendation language, we designed a new *recommendation query language REQUEST*,[1] which allows its users to express in a flexible manner a broad range of recommendations that are tailored to their own individual needs and, therefore, more accurately reflect their interests. For example, the earlier recommendation for Tom can be expressed in REQUEST as

```
RECOMMEND  Movie, Time TO User, Companion
USING  MovieRecommender
RESTRICT  User.Name = "Tom" AND Time.TimeOfWeek="weekend" AND Companion.Type = "Girlfriend"
BASED ON  PersonalRating
SHOW TOP 3
```

where MovieRecommender is a 5-dimensional cube of ratings having dimensions User, Movie, Time, Companion, and Theater; also, PersonalRating represents the ratings measure for the cube.

The above REQUEST query is based on the OLAP paradigm [9], which is a natural choice for querying multidimensional recommender systems, since the data model of REQUEST matches the multidimensional data model of the ratings cube. Besides REQUEST, we also present a multidimensional

[1] REQUEST is an acronym for REcommendation QUEry STatements. The initial version of our recommendation query language, called RQL, was introduced in an earlier workshop paper [5], where only the preliminary ideas of how to define the query language were presented. In this paper, we systematically redesigned the language by formally introducing its syntax, semantics, and the corresponding recommendation algebra. This allowed us to significantly extend capabilities of the language over its preliminary version [5]. To reflect these major changes, we renamed the language from RQL to REQUEST.

recommendation algebra that is used for defining certain "core" parts of REQUEST queries. We also describe how these core REQUEST queries can be processed by mapping them into this algebra.

This paper makes the following contributions. It proposes language REQUEST for expressing flexible user-driven recommendations and presents its syntax and semantics. It also presents recommendation algebra RA, which enhances the systematic definition of REQUEST. We also show how the core REQUEST queries can be mapped into RA, thus providing a way to process these queries, and compare the expressive power of REQUEST and RA.

2. Background: Multidimensional Recommender Systems

A multidimensional *ratings cube* is defined as a tuple (D, M, H, E, L) as follows.

Dimensions (D). $D = \{d_1, d_2, \ldots, d_n\}$ is a set of n dimensions, where each d_i is a dimension name. For example, in addition to the standard *User* and *Movie* dimensions of the traditional movie recommender systems, such as MovieLens [19], we consider other *contextual* dimensions [4, 5], such as Time, Theater and Companion., i.e., D = {User, Movie, Time, Theater, Companion}.

Attribute Hierarchies (H). Each dimension d_i is represented by a set of attributes $A_i=\{a_{i1}, \ldots, a_{it}\}$ where each a_{ij} is an attribute name; e.g. A_{time}={Date, DayOfWeek, TimeOfWeek, Month, Quarter, Year}. The domain of attribute x of dimension d is denoted as *dom*($d.x$), e.g., *dom*(Time.DayOfWeek) = { Mon, Tue, Wed, Thu, Fri, Sat, Sun } and *dom*(Time.TimeOfWeek) = { weekday, weekend }.

The multidimensional recommendation model allows for OLAP-based aggregation hierarchies [4, 5] that help aggregate ratings according to the methods described in [4]. In particular, attributes A_i of dimension d_i form a directed acyclic graph (i.e., a hierarchy) $H_i = (A_i, E_i)$ with set of nodes A_i (i.e., each node corresponds to an attribute) and set of edges E_i. There exists a directed edge in H_i from attribute $x \in A_i$ to attribute $y \in A_i$, iff every value of x uniquely determines the value of y, i.e., if attribute y is functionally dependent on attribute x. Such an edge will be denoted (x, y) or $x \rightarrow y$. We will assume that H_i has a single root node, *Root*(H_i), which we will call the *key* dimension attribute, consistent with the standard database terminology. Let $H = \{ H_1, \ldots, H_n \}$.

Given hierarchy H_i and attribute $d_i.x \in A_i$, we define *SubGraph*(H_i, $d_i.x$) to be a subgraph of H_i rooted

at $d_i.x$, i.e., it defines the graph containing all the nodes and edges reachable from $d_i.x$.

Elements (E). Each dimension d_i in a cube is represented by a set of elements E_i. For instance, dimension Movie in our example is represented by all the movies available for the users to rate. For simplicity and without loss of generality, we use the domain of the key dimension attribute to represent the set of elements of d_i, i.e., $E_i := dom(Root(H_i))$. An example of the elements' set for the User dimension would be a set of all user IDs available in the data. Let $E = \{ E_1, \ldots, E_n \}$.

Measures (M). $M = \{m_1, m_2, \ldots, m_k\}$ represents a set of measures, where each m_i is a different type of a rating from domain $dom(m_i)$. The measures can either be numeric or Boolean. A numeric measure usually represents a discrete finite ordered value, e.g., a movie rating on the scale of $\{1, \ldots, N\}$. A Boolean measure can be used to represent a "status flag" denoting the *state* of a rating or its specific characteristic, e.g., indicating whether a given movie has been seen by a given user.

> **Example 1.** Consider the application for recommending movies to users that has the following dimensions, each dimension defined by the attributes specified in parentheses:
>
> - *Movie:* the set of all the movies that can be recommended; it is defined as Movie(MovieID, Title, Length, ReleaseYear, Director, Genre).
> - *User:* the people to whom movies are recommended; it is defined as User(UserID, Name, Address, Age, Gender, Profession).
> - *Theater:* the movie theaters showing the movies; it is defined as Theater(TheaterID, Name, Address, Capacity, City, State, Country).
> - *Time:* the time when the movie can be or has been seen; it is defined as Time(Date, DayOfWeek, TimeOfWeek, Month, Quarter, Year).
> - *Companion:* represents a person or a group of persons with whom one can see the movie. It is defined as Companion(companionType), where attribute companionType has values "alone", "friends", "girlfriend/boyfriend", "family", "co-workers", and "others".
>
> We also use *three* rating measures in this example: PublicRating, a numeric measure specifying how much the general public liked the movie; PersonalRating, a numeric measure specifying how much a particular person liked or is predicted to like the movie in the settings specified by the Time, Theater, and Companion dimensions; and Consumed, a Boolean measure specifying whether or not a given user has actually seen a given movie in a given context. The PersonalRating assigned to a movie by a person depends on where and how the movie has been

seen, with whom and at what time. Finally, we consider the following aggregation hierarchies: *Movie:* MovieID → Genre; *User:* UserID → Age, UserID → Gender, UserID → Profession; *Theater:* TheaterID → City → State → Country; *Time:* Date → DayOfWeek → TimeOfWeek, Date → Month→ Quarter→ Year. ■

Cube Cells (L). Each cube is a *partially defined* rating function R from an n-dimensional space of $E_1 \times \ldots \times E_n$ to a k-dimensional space of measures, i.e., R: $E_1 \times \ldots \times E_n \rightarrow dom(m_1) \times \ldots \times dom(m_k)$. Alternatively, a cube can be perceived as a set of cells L, each cell $l \in L$ consisting of the tuple (*address*, *content*), i.e., l = (*address*, *content*), where *address* = $(\alpha_1, \ldots, \alpha_n)$, $\alpha_i \in E_i$, and *content* = $(\beta_1, \ldots, \beta_k)$, $\beta_i \in$ dom(m_i). Since the mapping R is partial, *content* can also have value *NULL* for some cells. We also use the notation L[*address*] = *content* to refer to a specific cell, and L[*address*].m_j to refer to a specific measure within a cell. Furthermore, the ratings $R(\alpha_1, \ldots, \alpha_n)$ of the recommendation space $S = E_1 \times E_2 \times \ldots \times E_n$ are either explicitly provided by the users or are implicitly inferred by the system as described below. For example, R(Aviator, Jane, theater5, 2/19/2005, boyfriend) = (6, 8, True) means that Jane gave rating 6 (i.e., PersonalRating = 6) to "Aviator" that she actually saw (i.e., Consumed = True) with her boyfriend on February 19, 2005 in movie theater 5, but the general public gave the movie the rating of 8 (i.e., PublicRating = 8).

Given these preliminaries, the recommendation problem is defined as follows. First, the system needs to estimate the unknown ratings and make the rating function R total [4]. Second, to make a recommendation, one needs to select certain *non-overlapping* "what" dimensions $d_{i1}, \ldots, d_{ik}$ ($k < n$) and certain "for whom" dimensions $d_{j1}, \ldots, d_{jl}$ ($l < n$) and, accordingly, recommend for each tuple $(\alpha_{j1}, \ldots, \alpha_{jl}) \in E_{j1} \times \ldots \times E_{jl}$ tuple $(\alpha_{i1}, \ldots, \alpha_{ik}) \in E_{i1} \times \ldots \times E_{ik}$ maximizing the rating $R(\alpha_1, \ldots, \alpha_n)$ across all the tuples $(\alpha_1, \ldots, \alpha_n)$ coinciding with $(\alpha_{j1}, \ldots, \alpha_{jl}) \in E_{j1} \times \ldots \times E_{jl}$ on corresponding dimensions $d_{j1}, \ldots, d_{jl}$.

Since the rating cube is only partially filled, it is important to estimate the unspecified ratings for recommendation purposes. This multidimensional rating estimation problem is addressed in [4], where the *reduction-based* method of estimating unknown ratings in terms of the known ratings is presented. To

understand how it works, assume that we want to recommend a movie to Jane Doe who wants to see it with her boyfriend on Saturday in a movie theater. If the Time dimension is partitioned into weekend and weekday components and since Saturday falls on a weekend, the reduction-based approach uses *only* the ratings for the movies seen on weekends by customers with their boyfriends/girlfriends in the movie theaters in order to provide recommendations for Jane Doe. It was shown that this approach outperforms the standard collaborative filtering in multidimensional settings under certain conditions [4]. Alternative multidimensional rating estimation methods include *heuristic*- and *model-based* approaches [2].

In this paper we focus on the *querying* capabilities of the REQUEST language and, therefore, we assume that the multidimensional ratings cube is fully pre-computed *before* users start issuing recommendation queries. In other words, we assume that all the unknown ratings have been estimated using any of the aforementioned rating estimation techniques. How to perform rating estimation "on demand" based on the query that was issued on a partially filled ratings cube constitutes an interesting future research problem, as we mention in Section 6.

The work described in [4] focuses on presenting the multidimensional recommendation model and does not specify how to *express* a wide variety of recommendations that are possible in multidimensional settings. In the next section we address this limitation by presenting the query language REQUEST for expressing such recommendations.

3. Recommendation Query Language REQUEST

In this section, we describe the language by providing various examples of REQUEST queries in Section 3.1, then present its syntax in Section 3.2 and semantics in Section 3.3.

3.1. Introducing REQUEST via Examples

All the examples presented in this section are based on the 5-dimensional MovieRecommender schema from Example 1. The first example presents the most basic and traditional recommendation request.

Query 1: Recommend the best movies to users:

```
RECOMMEND  Movie TO User
USING  MovieRecommender
BASED ON  PersonalRating
```

In this query, the RECOMMEND and TO clauses specify that movies will be recommended to users. The USING clause specifies the name of the multidimensional rating cube. The BASED ON clause specifies that personal ratings are used for recommendation purposes. The movies in this query are ordered separately for each user based on the PersonalRating measure that is either provided by the user or estimated from the set of known ratings as mentioned in Section 2. The query returns the highest-rated movie for each user. Query 1 actually uses some defaults, and the equivalent query with the *explicitly* specified parameters is:

```
RECOMMEND  Movie (MovieID) TO User (UserID)
USING  MovieRecommender
BASED ON  PersonalRating(AVG)
SHOW TOP 1 BY PersonalRating
```

Qualifier AVG specifies that, when the MovieRecommender cube is reduced to two dimensions Movie and User, all the ratings of a movie seen by a user on different occasions are aggregated by averaging their values (note that the user could see or rate the same movie more than once, i.e., in different contexts). Each measure can have its own default aggregation function (e.g., AVG in this case). The SHOW TOP *k* clause returns *k* best movies for the user ordered by aggregated PersonalRating measure (by default, $k = 1$). MovieID and UserID represent the dimensional attributes that will be used when displaying the results.

Next we introduce the restrictions on the recommendation criteria.

Query 2: Recommend, using personal ratings, top 5 action movies to users older than 18.

```
RECOMMEND  Movie TO User
USING  MovieRecommender
RESTRICT  Movie.Genre = "Action" AND User.Age >= 18
BASED ON  PersonalRating(AVG)
SHOW  TOP 5  BY PersonalRating
```

The RESTRICT clause is used to select the movies and the users satisfying the selection criteria. Then only the selected movies are ordered for each selected user based on the instructions specified in the BASED ON and the SHOW clauses, as discussed above. As this and other examples show, the syntax of REQUEST differs from that of SQL. This is done on purpose to reflect significant differences between

the application domains these languages are meant for. We discuss this further in Section 3.2.

We next show how ratings are filtered using the POSTFILTER clause.

Query 3: Recommend top 5 movies to the user for the weekend but only when personal ratings of the movies are higher than 7 (if fewer than 5 movies satisfy these criteria, then show only those satisfying them).

```
RECOMMEND  Movie TO User
USING  MovieRecommender
RESTRICT  Time. TimeOfWeek =”weekend”
BASED ON  PersonalRating(AVG)
POSTFILTER  PersonalRating > 7
SHOW  TOP 5
```

Query 3 demonstrates that different clauses (RESTRICT and POSTFILTER) are used for the selections of attributes and ratings. First, only the weekend ratings are selected with the RESTRICT clause. Then they are aggregated using the “BASED ON PersonalRating (AVG)” clause. Only then the POSTFILTER clause is applied to these *aggregated* PersonalRatings and only those greater than 7 are selected. If we want to restrict non-aggregated ratings, we should use the PREFILTER clause, as will be shown in Query 5. The reasons for using separate RESTRICT and POSTFILTER clauses when restricting attributes and ratings are discussed in Section 3.2.

The next example shows that more than one dimension can be used in recommendations, i.e., Movie and Time are recommended to User and Companion.

Query 4: Recommend to Tom and his girlfriend top 3 movies and the best times to see them over the weekend.

```
RECOMMEND  Movie, Time TO User, Companion
USING  MovieRecommender
RESTRICT User.Name = “Tom” AND Time.TimeOfWeek =”weekend” AND Companion.Type = “Girlfriend”
BASED ON  PersonalRating
SHOW  TOP 3
```

Sometimes, a certain *group* of people may be interested in a certain *type* of movies. For example, there has been work done on the topic of recommending to groups of users [13] as well as using aggregate ratings in the recommendation process [25]. The next example shows how this type of aggregation can be done in REQUEST.

Query 5: Recommend movie genre to various professions using only the movies with personal ratings bigger than 6:

```
RECOMMEND  Movie.Genre TO User.Profession
USING  MovieRecommender
PREFILTER  PersonalRating > 6
BASED ON  PersonalRating(AVG)
```

This query aggregates rating scores for individual movies into averaged rating scores for different genres of movies. Also, individual users are aggregated by profession, and each profession becomes a new target for a recommendation. Before the ratings are aggregated, the PREFILTER operator selects the ratings bigger than 6, and only these ratings are aggregated. It differs from the POSTFILTER operator in Query 3 since it deals with *non-aggregated* ratings, whereas POSTFILTER is applied to the aggregate ratings. This distinction is crucial in some recommendation settings.

The next example demonstrates that recommendations are not restricted to the User dimension; in general, different things can be recommended to various objects.

Query 6: Identify the top two professions that appreciate the movie "Beautiful Mind" the most.

```
RECOMMEND  User.Profession TO Movie
USING  MovieRecommender
RESTRICT  Movie.Title = "Beautiful Mind"
BASED ON PersonalRating(AVG)
SHOW  TOP 2
```

Remember that a rating score for a movie is either *explicitly specified* by the user or is *estimated* from the existing user-specified ratings using one of the rating estimation methods described in [2, 4]. The next query is a modification of Query 1 that makes use of this fact.

Query 7: Recommend best movies to users *that they have not seen yet.*

```
RECOMMEND  Movie TO User
USING  MovieRecommender
BASED ON  PersonalRating(AVG), Consumed(DISJ)
POSTFILTER NOT(Consumed)
SHOW TOP 1 BY  PersonalRating
```

This query collects all the ratings given to a movie by a user (note that the user can provide multiple ratings to a movie, seen in different contexts). *Consumed* is a Boolean flag related to PersonalRating

measure specifying whether a movie was seen ("consumed") by a user on some occasion. *Consumed*(*DISJ*) is the disjunction of the values of all of these flags for a movie/user pair. If this disjunction is true (the cumulative Consumed flag is True), this means that on at least one occasion the user has seen the movie. The POSTFILTER NOT(Consumed) statement removes these cases. Thus, only the movies that the user has not seen before are recommended.

The next query shows how recommendations based on multiple ratings are used.

Query 8: Show top 5 movies with both public ratings and personal ratings bigger than 8 to students based only on the movies they have seen.

```
RECOMMEND  Movie To User
USING  MovieRecommender
RESTRICT  User.Profession = "Student"
PREFILTER  Consumed
BASED ON PersonalRating(AVG), PublicRating(AVG)
POSTFILTER  PublicRating > 8 AND PersonalRating > 8
SHOW TOP 5  BY PersonalRating, PublicRating
```

This query first selects the ratings of movies provided by students that they have previously seen (i.e., prefilters them based on the Consumed flag). Then it aggregates them based on personal and public ratings and selects only PublicRating and PersonalRating that on average are greater than 8. Finally, it sorts the movies for each user based on these two ratings in a standard lexicographic manner and selects the top five of them for each user.

Since ratings in a recommender system can be provided by users or estimated by software, note that we have an option of differentiating between actual, estimated, and other types of ratings for any rating measure. The REQUEST language supports this functionality via binary flags (implemented as separate Boolean measures) that can be used in PRE- and POSTFILTER clauses, as well as be aggregated using specific Boolean aggregation functions. For example, Queries 7 and 8 use the Consumed flag specifying if the rating is based on the movie that the user has actually seen. This is possible because, as explained earlier, all the estimated ratings and the related flags are *precomputed*, and thus can be used conceptually as additional measures.

After introducing REQUEST via examples, we next define the syntax of the language.

3.2. Syntactic Definition of REQUEST

The BNF specification of REQUEST syntax is presented in Figure 1. First, note that we do not mimic the syntax of SQL for REQUEST because, if we tried to do so, this would likely cause many false assumptions on behalf of the users who may assume that properties of SQL operators automatically extend to REQUEST simply because the names of the operators are the same. For example, the RESTRICT clause of REQUEST is significantly more restrictive than the WHERE clause of SQL, as will be explained below. This observation is also applicable to various other REQUEST clauses that will be discussed later in this section.

We have developed REQUEST as a domain-specific ("vertical") query language (i.e., a language for a specialized application domain of recommender systems) as opposed to a general language for querying data. Following this approach, we tried to make sure that every construct of the language has a well-defined and intuitive meaning *pertaining to recommender systems*, while at the same time trying to maintain expressiveness and rigor of the language.

In particular, as Figure 1 shows, the USING clause allows only a single cube, thus restricting recommendations to a single cube of ratings and prohibiting *joins* between cubes in REQUEST. This is the case, because multi-cube recommendations seldom have meaningful and practically important applications and also can lead to various complications and side-effects. For example, in order to join two cubes on a certain dimension, such as Time, the two dimensions should be identical for *all* the levels of the aggregation hierarchy, e.g., across the entire Time hierarchy, which is often impractical and also difficult to enforce. Also, if multiple cubes are used in queries, then there is a dilemma of whether the PUSH and PULL operators of the standard OLAP querying paradigm [1], that can "push" one of the dimensions to become a measure and also "pull" a measure as a new dimension, should be supported at the algebraic level. Without such operators incorporated into REQUEST, certain multi-cube queries either cannot be expressed or can be done only in a very convoluted manner. At the same time, incorporating the PUSH and PULL operators into the language creates numerous complications for REQUEST due to the inherent semantic differences between dimensions and measures in the

multidimensional recommendation model. Therefore, having multiple cubes creates various problems both with *and* without the PUSH and PULL operators in REQUEST. Finally, when joining cubes, estimated ratings need to be re-evaluated for the joined cubes, often in significantly higher-dimensional spaces. This can lead to the rating estimation problem due to the rating sparsity in the joined cube. Instead of supporting joins in REQUEST, a much better alternative is for the domain expert to manually build a single cube that is the join of two or more individual cubes. In the rest of this section, we describe syntax of REQUEST based on Figure 1.

The RESTRICT clause contains *dimension_restrictions* that constitute the standard restrictions of the "slice-and-dice" operator of the OLAP systems. Each individual restriction is limited to the numeric and textual comparison of a dimension attribute to a constant value (or a set of values), as specified in the BNF grammar above, and these dimensions and attributes have to be present in the schema of the *cube_name* cube. Moreover, multiple restrictions in a single RESTRICT clause are permitted, but only if combined by logical operator AND. Disjunctions (OR) are *not* allowed because the result of such restrictions would no longer be a multidimensional cube, as illustrated in Figure 2.

We also would like to note that, although the RESTRICT clause is somewhat similar to the WHERE clause of SQL, they also have the following key differences mostly stemming from our need to restrict REQUEST to make it more suitable for the recommendation applications. First, the WHERE clause of SQL is not limited to conjunctions, as RESTRICT is, but can also have disjunctions. Second, each individual restriction (conjunct) in the RESTRICT clause can involve only one dimension (i.e., a comparison of some dimension attribute to a constant, as mentioned earlier) in order to ensure that the result of the restriction is still a proper multidimensional cube. For this reason, for example, the restriction "RESTRICT User.Age > Movie.Length" is not allowed in REQUEST. In contrast, the WHERE clause of SQL allows having attributes from multiple relations in a single condition. Third, the WHERE clause of SQL supports nested queries, whereas RESTRICT does not. Besides these major differences between the two clauses, there are also minor differences apparent from the BNF grammars of the two languages.

// **general syntax of a REQUEST query**
REQUEST_query ::=
RECOMMEND *recommend_dim_list* TO *recipient_dim_list*
USING *cube_name*
[RESTRICT *dimension_restrictions*]
[PREFILTER *preaggregation_measure_restrictions*]
BASED ON *aggr_measure_list*
[POSTFILTER *postaggregation_measure_restrictions*]
[SHOW *measure_rank_restriction*]

// **RECOMMEND and TO clauses**
recommend_dim_list ::= *dimension_list*
recipient_dim_list ::= *dimension_list*
dimension_list ::= *single_dimension* { **,** *single_dimension* }*
single_dimension ::= { *dimension_name* [*output_attribute_list*] | *dimension_attribute* }
output_attribute_list ::= **(** *attribute_name* { **,** *attribute_name* }* **)**

// **USING clause**
cube_name ::= *variable*

// **RESTRICT clause**
dimension_restrictions ::= *single_dimension_restriction* { AND *single_dimension_restriction* }*
single_dimension_restriction ::=
dimension_attribute { *numeric_comparison* | *textual_comparison* | *set_membership_test* }

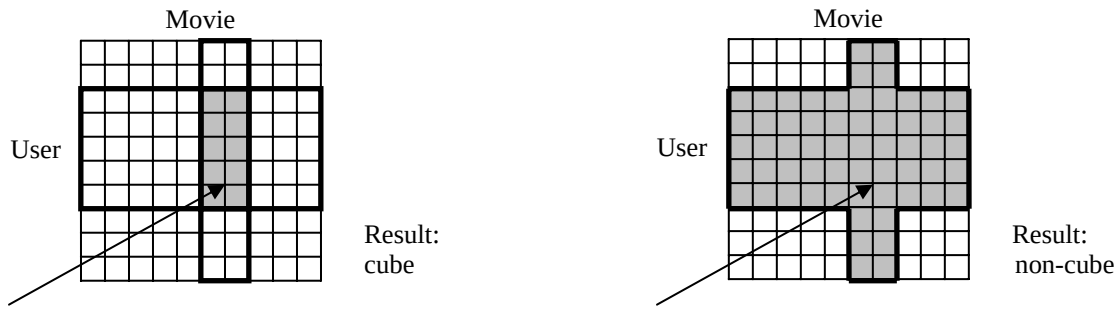
// **BASED ON clause**
aggr_measure_list ::= *single_aggr_measure* { **,** *single_aggr_measure* }*
single_aggr_measure ::= *measure_name* [**(** *rating_aggr_function***)**]
rating_aggr_function ::= *numeric_aggr_function* | *boolean_aggr_function*
numeric_aggr_function ::= MIN | MAX | SUM | AVG
boolean_aggr_function ::= DISJ | CONJ | MAJORITY

// **PREFILTER and POSTFILTER clauses**
preaggregation_measure_restrictions ::= *measure_restrictions*
postaggregation_measure_restrictions ::= *measure_restrictions*
measure_restrictions ::= *single_measure_restriction* { *logical_op single_measure_restriction* }*
single_measure_restriction ::= *numeric_measure_restriction* | *boolean_measure_restriction*
logical_op ::= AND | OR
numeric_measure_restriction ::= *measure_name numeric_comparison*
boolean_measure_restriction ::= *measure_name* | NOT **(** *measure_name* **)** | *measure_name* **=** *boolean_value*

// **SHOW clause**
measure_rank_restriction ::= { TOP | BOTTOM } *number* [BY *measure_list*]
measure_list ::= *measure_name* { **,** *measure_name* }*

// **common expressions**
dimension_attribute ::= *dimension_name* **.** *attribute_name*
dimension_name ::= *variable*
attribute_name ::= *variable*
measure_name ::= *variable*
numeric_comparison ::= { **=** | **<>** | **>** | **<** | **<=** | **>=** } *number*
textual_comparison ::= { **=** | LIKE } **'***string***'**
set_membership_test ::= { IN | NOT IN } **(** *value_list* **)**
value_list ::= *numeric_value_list* | *textual_value_list*
numeric_value_list ::= *number* { **,** *number* }*
textual_value_list ::= **'***string***'** { **,** **'***string***'** }*
boolean_value ::= true | false

Figure 1. BNF Specification of REQUEST Syntax.



(a) User.Age>=18 **AND** Movie.Genre='action' **(b)** User.Age>=18 **OR** Movie.Genre='action'

Figure 2. Combining dimension restrictions using (a) AND and (b) OR operators.

The PREFILTER and POSTFILTER clauses contain *measure_restrictions* that constitute a set of restrictions on various types of measures used in *cube_name.* Note that, unlike *dimension_restrictions*, both AND and OR operators are allowed in *measure_restrictions* according to Figure 1. REQUEST uses separate RESTRICT and PRE-/POSTFILTER clauses when restricting attributes on dimensions and rating measures for the following reasons. First, these two types of restrictions are semantically very different: the first one restricts the contextual information by imposing conditions on dimensional attributes, while the second does it on the measures. Second, the POSTFILTER clause *must* be kept separately because, unlike RESTRICT, it is applicable to the aggregate ratings, which are semantically different from the un-aggregated ratings. Although this point is not applicable to the PREFILTER clause, it is better to keep both the PRE- and the POSTFILTER clauses (as they are symmetric), which makes it impossible to merge PREFILTER and RESTRICT clauses. Third, as pointed out before, only disjunctions are allowed in the RESTRICT, but both conjunctions and disjunctions are allowed in the PRE- and POSTFILTER clauses, making it even more important to treat them separately. Fourth, to keep the semantics of recommendations clear, it is important not to mix the rating measures and dimensional restrictions by prohibiting expressions of the form "PersonalRating > Time.DayOfWeek." These were the reasons for keeping the two types of restrictions separate. Note that this situation is not unlike the case in temporal databases, where separate WHERE and WHEN clauses are used for regular and temporal dimensions [23].

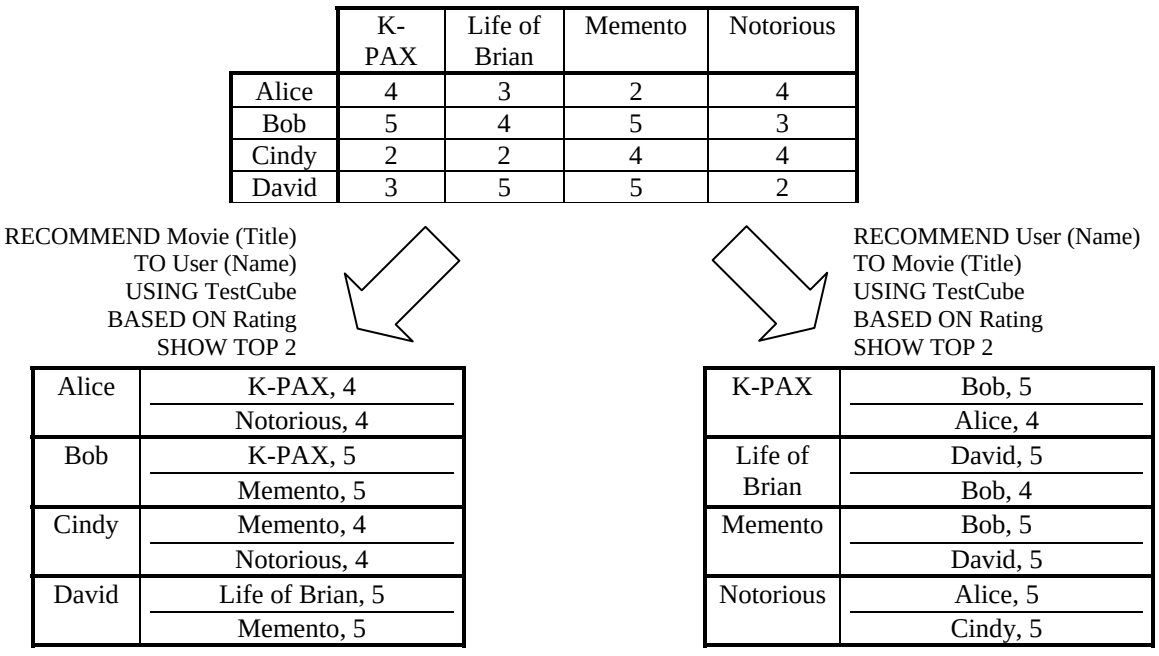
Formally, the output of a recommendation query is a set of tuples $\{ (t,L_t) \mid t \in T \}$, where t is a

recommendation recipient and L_t is a list of recommendations for recipient *t*. For example, in a movie recommender system, a simple example of a recommendation tuple would be: (JohnDoe, <(Titanic,10),(Gladiator,9),(StarWars,8)>). In other words, *T* represents the element combinations of dimensions from *recipient_dim_list* specified in the TO clause of the query. L_t consists of an *ordered set* of *k* recommendations, where *k* is specified by the SHOW clause of the query. More precisely, $L_t = < (r_{t1},m_{t1}), \ldots, (r_{tk},m_{tk}) >$, i.e., each recommendation is represented by a tuple (r_{tj},m_{tj}), where $r_{tj} \in R_q$ and $m_{tj} \in M_q$. Here R_q represents the element combinations of dimensions from *recommend_dim_list* specified in the RECOMMEND clause of the query (note that *recommend_dim_list* and *recipient_dim_list* must be *mutually exclusive*), and M_q represents the combinations of possible values for one or more measures that are specified in *measure_list* in the BY subclause (of the SHOW clause). Specific tuples (r_{tj},m_{tj}) are obtained from the *processed* ratings cube (i.e., after restrictions and aggregations specified in the query are done) by sorting all cells belonging to a given recipient *t* based on their measure values; these measure values m_{tj} and the corresponding element combinations of "RECOMMEND" dimensions r_{tj} constitute the contents of each recommendation in L_t. L_t is further truncated according to the SHOW clause that limits the results to the top or bottom *k* recommendations. If more than one measure is specified in *measure_list*, then the ordering is lexicographic. For example, in Query 8, Movies are first ordered based on the PersonalRating measure; if some records have the same value of PersonalRating, then those are further sorted based on PublicRating. Also, if the optional BY subclause is not specified, the results are sorted according to the first measure in the BASED ON clause.

The output of a recommendation query can intuitively be represented as a matrix (cube) of the "TO" dimensions with the entries consisting of the lists of the elements representing the "RECOMMEND" dimensions. In other words, one row in a recommendation matrix directly corresponds to one recommendation tuple described earlier. For example, Figure 3 shows the output matrices for two recommendations (movies to users and vice versa). The answer to the left query shows top two movies for each user, and the right one – the top two users for each movie (as specified in the SHOW clause).

The output matrix produced for the left query in Figure 3 is based on users (as specified in the TO clause), and its cells contain movies (as specified in the RECOMMEND clause) and the corresponding rating measures (which were also used for sorting).

Note that, if the end-users want to use actual user names and movie titles in the output matrix, they should specify the *output_attribute_list* parameters from Figure 1 in the RECOMMEND and TO clauses. For example, the left recommendation in Figure 3 is stated as "RECOMMEND Movie(Title) TO User(Name)…" Also note that the recommendation results can be more complex in the sense that multiple dimensions can be used in the RECOMMEND and TO clauses, as Query 4 from Section 3.1 demonstrates. In such cases, each row in the output matrix can contain multiple dimensions and complex multidimensional entries in the cells.



	K-PAX	Life of Brian	Memento	Notorious
Alice	4	3	2	4
Bob	5	4	5	3
Cindy	2	2	4	4
David	3	5	5	2

Alice	K-PAX, 4
	Notorious, 4
Bob	K-PAX, 5
	Memento, 5
Cindy	Memento, 4
	Notorious, 4
David	Life of Brian, 5
	Memento, 5

K-PAX	Bob, 5
	Alice, 4
Life of Brian	David, 5
	Bob, 4
Memento	Bob, 5
	David, 5
Notorious	Alice, 5
	Cindy, 5

Figure 3. Generating recommendations from a multidimensional ratings cube.

Although we used the term "REQUEST queries" throughout the paper, recommendations are really not queries according to the standard meaning of the term, since they return a very idiosyncratic output of a recommendation matrix, which prevents the whole recommendation operation from being *closed*. To address this issue, we distinguish between the *Core-REQUEST query* containing RECOMMEND, TO,

USING, RESTRICT, PREFILTER, POSTFILTER, and BASED ON clauses, and the recommendation *wrapper* containing RECOMMEND, TO, and SHOW clauses (different aspects of RECOMMEND and TO are used in core and wrapper parts of the query). The Core-REQUEST query operates on a multidimensional cube of ratings and *always* returns the same type of an object – a cube of ratings. In contrast to this, the recommendation wrapper takes a multidimensional cube of ratings and transforms it to a different type of object – the recommendation matrix that is subsequently returned as an output to the end-user. When processing REQUEST queries, the core query is evaluated first, and then the wrapper is applied to the output of the Core-REQUEST query.

Although REQUEST is related to the OLAP query languages, it has certain distinctive characteristics pertaining to recommendations that make it different from these languages. First, as explained above, REQUEST queries are divided into the "core" and "wrapper" components, each component requiring separate evaluation methods. Second, ratings can be actual (specified by the user) and inferred (from the actual ratings). Therefore, REQUEST supports mechanisms for distinguishing between different types of ratings, as Query 7 demonstrated. Third, the language provides various other recommendation-specific properties, such as using a single cube of ratings, the PREFILTER and POSTFILTER clauses, and recommendation-specific types of aggregations. All this differentiates REQUEST from the general-purpose OLAP-based query languages and makes it a uniquely suited *vertically-targeted* language for recommender systems. This approach of developing a special-purpose vertically-targeted query language to meet the idiosyncratic needs of a particular class of applications (recommender systems, in this case) is in line with the development of other types of special-purpose query languages for different classes of vertical database applications, such as temporal, spatial, and multimedia applications.

3.3. Semantics of REQUEST Queries

Operational semantics of REQUEST is defined as the following sequence of operations over the cube *cube_name* from the USING clause of the query. Note that this operational semantics is only *conceptual*, i.e., the real query processing may be performed differently, but would result in the same exact outcome as explained below.

1. Dimension restrictions. First, the dimension restrictions specified in the RESTRICT clause, if present, produce a sub-cube of *cube_name* by restricting some of its dimensions to include only a subset of their elements specified in the RESTRICT clause. For example, "RESTRICT User.Age ≥ 18 AND Movie.Genre = 'action' " produces a smaller cube having only the users with ages 18 and above and only the action movies. Section 3.2 lists the limits to the syntax of these restrictions (e.g., only comparisons of dimensional attributes to constants are allowed, and no disjunctions in the RESTRICT clause). This amounts to applying restrictions in the RESTRICT clause one dimension at a time and also restricting one attribute at a time. The order in which these dimensions are restricted is unimportant since the final result does not depend on it.

2. Measure-based cube filtering (before aggregation). In this step, the cells of the restricted cube produced in Step 1 are further filtered based on measure restrictions specified in the PREFILTER clause. Since the measures supported by our multidimensional recommendation model can be either numeric or Boolean, the measure-based filtering capabilities of REQUEST include comparisons of both numeric and Boolean measures to specific values, as specified in Figure 1. Also note that this step filters *individual* cube cells based on their measure values using multiple restrictions combined with AND and OR logical operators. Again, all measures mentioned in the PREFILTER clause have to be present in the schema of the *cube_name* cube.

3. Cube aggregation. After performing restricting and prefiltering operations in Steps 1 and 2, the remaining cells of the obtained cube are aggregated according to the dimensions and their granularity levels specified in the mutually exclusive lists *recommend_dim_list* and *recipient_dim_list* from the RECOMMEND and TO clauses and according to the following three rules: (i) if a dimension is specified in either RECOMMEND or TO clause by itself, i.e., without providing an aggregation attribute (e.g., Movie), then the cube is not aggregated along this dimension; (ii) if a dimension is specified in either RECOMMEND or TO clause with a corresponding aggregation attribute (e.g., Movie.Genre), then the cube is aggregated along this dimension based on the specified attribute (e.g., all individual movies are aggregated into their genres); (iii) if a dimension is omitted from both RECOMMEND and TO clauses,

then the cube is aggregated fully along this dimension (i.e., in the resulting cube this dimension essentially disappears). Furthermore, the aggregation is done for the measures specified in the BASED ON clause. This clause also specifies the aggregation functions to be used for each measure. The currently supported numeric aggregation functions include AVG, SUM, MIN, and MAX, and the supported Boolean aggregation functions include CONJ (i.e., conjunction), DISJ (i.e., disjunction), and MAJORITY; however, the REQUEST language can be easily extended to support additional aggregation functions, such as AVG-in-TOP-*n*. If an aggregation function is not specified for the measures in the BASED ON clause, a default aggregation function for that measure is used (e.g., AVG). Essentially, this step represents a typical "roll-up" operation in OLAP systems. Figure 4 provides further illustration of the cube aggregation operation, where the Time dimension is collapsed and the User dimension is aggregated based on the gender attribute. Then, for each movie, all the ratings across different occasions provided by users of a particular gender (e.g., see the shaded area in Figure 4) are averaged using the AVG function.

4. Measure-based cube filtering (after aggregation). In this step, the cells of the resulting aggregated cube can further be filtered based on the aggregated measure restrictions specified in the POSTFILTER clause. As it is specified in the BNF grammar in Figure 1, the syntax of the POSTFILTER clause is the same as for PREFILTER. Thus, if a given REQUEST query contains no aggregation, then PREFILTER and POSTFILTER, if both present, can be combined as a conjunction into one measure-based filtering operation at the query processing stage. Also, all the measures mentioned in the POSTFILTER clause must appear among the measures mentioned in the BASED ON clause (because only these measures are aggregated). Note that the POSTFILTER clause is somewhat similar to the HAVING clause in SQL since both of them provide additional restrictions based on the aggregated data. However, one significant difference is that SQL allows creating an arbitrary number of aggregated attributes from the same un-aggregated one, e.g., "SELECT MIN(Rating), MAX(Rating), SUM(Rating), COUNT(Rating), AVG(Rating) FROM Table". In contrast, each measure can lead to just one aggregated version of itself in REQUEST. Another difference is that REQUEST supports not only numeric, but also Boolean

aggregation functions (that are not available in standard SQL).

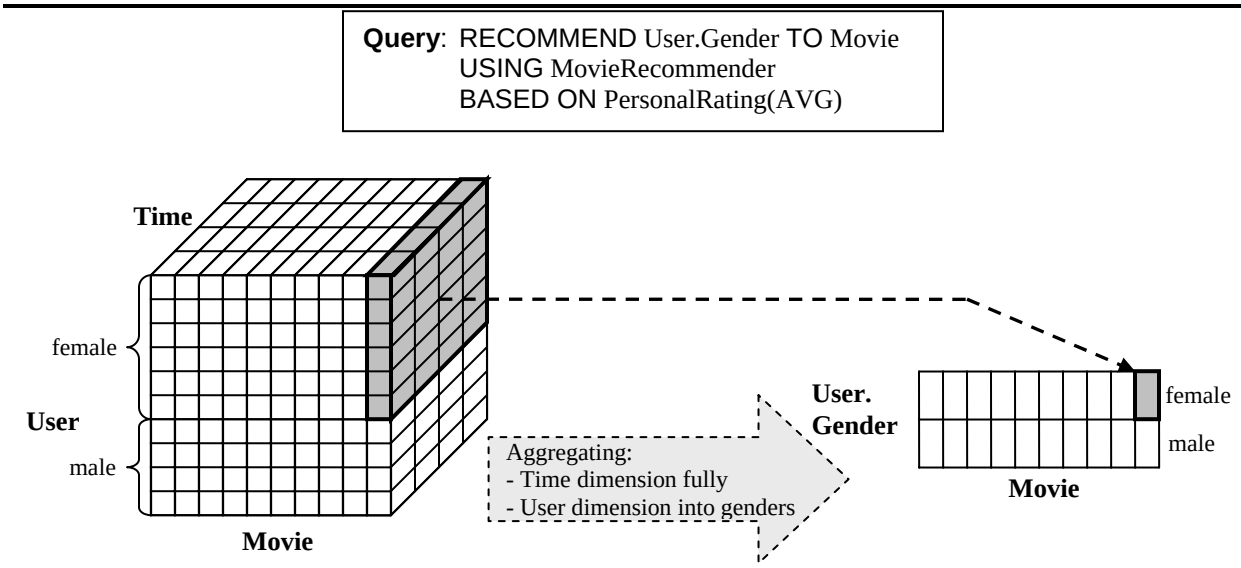


Figure 4. Cube aggregation operation.

5. Generating recommendations. In this step, the cube obtained in Step 4 is transformed into a specialized recommendation matrix, as was described in detail in Section 3.2. The rows of this matrix are determined by the TO clause of the REQUEST query. Each row of the matrix also contains the list of the records specified in the RECOMMEND clause and the measures are used to sort the results. These lists of records are sorted and truncated based on the SHOW clause.

This completes the description of semantics of the REQUEST language. This description was provided in a semi-formal manner in the sense that we did not use mathematics to define semantics of each of the 5 operations formally for the sake of readability. However, we provided enough details for the interested reader to easily understand and reconstruct formal semantic procedures defining each of these five steps. Moreover, we provide a formal definition of the recommendation algebra in Section 4 which will make this reconstruction process easier.

We next present a recommendation algebra that more formally defines how REQUEST queries are processed. Since algebraic operators should return objects of the same type as their inputs, we will target the recommendation algebra *only* to the Core-REQUEST queries (corresponding to Steps 1-4 above). To process a full REQUEST query, we construct an algebraic expression equivalent to the Core-REQUEST

query, evaluate it, and then "feed" the results into the REQUEST wrapper to produce the final output.

4. Recommendation Algebra (RA)

Since multidimensional recommendations are based on the OLAP paradigm, we use the OLAP algebras introduced in the database community [1, 11, 16, 17, 24] to define the recommendation algebra. However, since the REQUEST language is tailored specifically for the domain of recommendations, only a subset of the standard OLAP operators is needed to process REQUEST queries. For example, we do not use a JOIN operator in the recommendation algebra because REQUEST works only on one cube, and we do not use the PUSH and PULL operators for the reasons explained in Section 3. In the rest of this section, we describe the recommendation algebra RA. We will follow the definitions of the OLAP operators introduced in [24].

The general syntax of recommendation algebra operators is:

$$C_O = OP_{parameters}(C_I),$$

where $C_I = (D, M, H, E, L)$ denotes the input cube, $C_O = (D^*, M^*, H^*, E^*, L^*)$ – the resulting output cube, *OP* – a recommendation algebra operator, and *parameters* – the parameters of operator *OP*. The ratings cube and its components *D*, *M*, *H*, *E* and *L* are defined in Section 2.

We next introduce individual operators using this general syntax.

Dimension restriction (DRSTR) operator. This operator defines the "slice and dice" operation on the cube by putting restrictions on the dimensions. The simplest form of DRSTR operator is:

$$C_O = DRSTR_{Psimple}(C_I),$$

where P_{simple} is a domain restriction based on a single dimension d_i, e.g., "User.Age > 21". In other words, P_{simple} is a Boolean function (a predicate) of the form P_{simple}: $E_i \rightarrow$ { true, false }. Given an arbitrary input ratings cube, as a result of this operator, only those cells that satisfy the given predicate are retained in the resulting cube. The calculation of C_O is formally defined as:

- $D^* = D$, $M^* = M$, and $H^* = H$.
- $E_i^* = \{e_i \in E_i \mid P_{simple}(e_i)\}$. Also, $E_j^* = E_j$, if $j \neq i$.
- $L^* = \{(address, content) \in L \mid address \in E_1^* \times \ldots \times E_n^*\}$.

In addition to the aforementioned simple predicates, this operator can also support more complex predicates. For example, $P_{complex}$ could be represented by a compound predicate of the form

$$P_{complex} = p_1 \text{ AND } p_2 \text{ AND } \ldots \text{ AND } p_x$$

where each p_j is a domain restriction involving a single dimension. Because the result of a simple predicate-based restriction is always a cube, the compound restriction operator is defined as a composition of simple restriction operators, i.e.,

$$C_O = DRSTR_{Pcomplex}(C_I) = DRSTR_{p1 \text{ AND } \ldots \text{ AND } px}(C_I) = DRSTR_{p1}(DRSTR_{p2}(\ldots(DRSTR_{px}(C_I))\ldots)).$$

Note that, while DRSTR operator can support predicates with conjunctions (logical AND operations), it does not support arbitrary disjunctions (logical OR operations) because the result of such operations is no longer guaranteed to be a cube, as mentioned earlier.

Measure restriction (MRSTR) operator. This operator defines the "cell filtering" operation on the ratings cube by putting restrictions on measures. The simplest form of MRSTR operator is:

$$C_O = MRSTR_P(C_I),$$

where P is a measure restriction based on a single measure m_j. The restrictions can be based on a numeric measure ("PersonalRating > 7") and on a Boolean measure ("Consumed = false").

Unlike the DRSTR operator (which is a "slice and dice" operator), the MRSTR operator performs simple filtering of cube cells and, therefore, can support more complex predicates, e.g.,

$$P = p_1 <op> p_2 <op> \ldots <op> p_x ,$$

where <*op*> represents a logical operator AND or OR. Given an arbitrary input cube, as a result of this operator, only the content of cells that satisfy the given predicate are retained in the resulting cube. The content of all other cells is assigned to NULL (these cells are retained and not deleted in order to maintain the proper cube structure).

The calculation of C_O is formally defined as follows:

- $D^* = D$, $M^* = M$, $H^* = H$, and $E^* = E$.
- Assign $L^* = L$. Then, $\forall(address, content) \in L^*$: if $\neg P(content)$ then $L^*[address]$ = NULL.

Metric projection (MRPJ) operator. This operator restricts the output of a ratings cube to include only a subset of the original set of measures. The simple form of MRPJ operator is:

$$C_O = MRPJ_{mj}(C_I),$$

where m_j is a measure to be projected out. The calculation of C_O is formally defined as follows:

- $D^* = D$, $H^* = H$, and $E^* = E$.
- $M^* = M - \{ m_j \}$.
- Assign $L^* = L$. Then, $\forall(address, content) \in L^*$: remove the j^{th} measure from $L^*[address]$. If, as a result, $L^*[address]$ has no more measures left, assign $L^*[address]$ = NULL.

If a set of metrics $M' = \{m'_1, \ldots, m'_x\}$ is needed to be removed at once, a more complex MRPJ operator can be implemented as a composition of simple MRPJ operators. In other words,

$$C_O = MRPJ_{M'}(C_I) = MRPJ_{m'1}(\ldots(MRPJ_{m'x}(C_I))\ldots).$$

Destroy dimension (DTDM) operator. This operator reduces dimensions of the resulting ratings cube by including only a subset of the original set of dimensions. The simplest form of DTDM is

$$C_O = DTDM_{di}(C_I),$$

where d_i is a dimension to be destroyed. Note that, if we destroyed dimension d_i by just removing its component from all cube cell addresses, we would have a number of cells in the cube with the same exact addresses, which leads to ambiguous results and, therefore, is undesirable. One way to deal with this situation is to aggregate all cells with the same address into a single cell in a resulting cube. Since, as described below, we already have an operator for cell aggregation (i.e., AGGR), we do not introduce the aggregation capability into DTDM. Therefore, in order to properly destroy dimension d_i, we restrict the use of the DTDM operator only to the situations in which there is no ambiguity and no loss of information in the resulting cube. Thus, dimension d_i can be destroyed only when it has been maximally aggregated: $|E_i| = 1$.

The calculation of C_O is formally defined as follows:

- If $|E_i| > 1$, abort processing and return the same ratings cube, i.e., $C_O = C_I$. Otherwise, continue as specified below.
- $M^* = M$, $D^* = D - \{ d_i \}$, $H^* = H - \{ H_i \}$, and $E^* = E - \{ E_i \}$.
- Assign $L^* = L$. Then, $\forall(address, content) \in L^*$: remove the i^{th} dimension from *address*.

If a set of dimensions $D' = \{d'_1, \ldots, d'_x\}$ is needed to be removed at once (assuming they are all maximally aggregated), a more complex DTDM operator can be easily implemented as a combination of simpler DTDM statements. In other words,

$$C_O = DTDM_{D'}(C_I) = DTDM_{d'1}(\ldots(DTDM_{d'x}(C_I))\ldots).$$

Aggregation (AGGR) operator. The aggregation operator performs aggregation on one or more dimensions and applies aggregation functions, such as SUM, AVG, etc., to each of the measures of the cube based on dimensions specified as grouping attributes. The general form of AGGR is:

$$C_O = AGGR_{(d_1.x_1,\ldots,d_l.x_l),(m_1.f_1,\ldots,m_k.f_k)}(C_I)$$

where $d_i.x_i$ represents a grouping attribute for dimension d_i (specified only for dimensions that need to be grouped). Also, $m_j.f_j$ is an aggregation function specified for each cube measure m_j. If the aggregation function is not specified for some measure, a default aggregation function for that measure is used. After aggregation, $d_i.x_i$ becomes a dimension (with its own attributes, whichever appropriate) by replacing d_i. Our model also provides a special option to aggregate the dimension completely by specifying $d_i.x_i$ as d_i.ALL (instead of using some attribute name). Furthermore, $m_j.f_j$ can be one of the standard numeric aggregation functions (e.g., MAX, MIN, AVG, SUM) or Boolean aggregation functions, including:

$$f_j(B) = \mathrm{DISJ}(B) = \vee_{b \in B} b, \qquad f_j(B) = \mathrm{CONJ}(B) = \wedge_{b \in B} b, \text{ and}$$

$$f_j(B) = \mathrm{MAJORITY}(B) = \begin{cases} \text{true}, & \text{if } |\{b \in B \mid b = \text{true}\}| \geq |\{b \in B \mid b = \text{false}\}| \\ \text{false}, & \text{otherwise} \end{cases}.$$

As an example, consider the movie recommender system having three dimensions: User, Movie, and Time. Suppose user John Doe wants to know which movies *that he has not seen yet* are most relevant to him, regardless of when he is planning to watch them. Suppose the system has the following four ratings for John Doe: [(John Doe, Gladiator, weekday), (Rating=8, Consumed=true)], [(John Doe, Gladiator, weekend), (Rating=9, Consumed=false)], [(John Doe, Titanic, weekday), (Rating=7, Consumed=false)], [(John Doe, Titanic, weekend), (Rating=8, Consumed=false)]. In this case, the aggregation operator would look like:

$$C_O = AGGR_{(Time.ALL),(Rating.AVG,Consumed.DISJ)}(C_I).$$

Since the time dimension has to be aggregated completely, we use AVG function to aggregate the ratings for the same movie; we also use DISJ Boolean aggregation function to make sure that movies with at least one Consumed rating would not get recommended (since the user has seen it already). In this case, the results of aggregation would be: [(John Doe, Gladiator, ALL), (Rating=8.5, Consumed=true)], [(John Doe, Titanic, ALL), (Rating=7.5, Consumed=false)], and the user, by filtering on the "Consumed=false" status flag, would be able to receive a correct recommendation of "Titanic", since the user has already seen "Gladiator".

Now consider the same four ratings but a different scenario, where John Doe wants to know which times of week seem to be best for him in terms of movie watching, regardless of what kind of movie he is planning on watching. In this case, the aggregation operator would be:

$$C_O = AGGR_{(Movie.ALL,Time.TimeOfWeek),(Rating.AVG,Consumed.CONJ)}(C_I).$$

Since the movie dimension has to be aggregated completely, we again use AVG function to aggregate the ratings for the same time values. However, this time it may make more sense to use the CONJ Boolean aggregation function to make sure that only time periods with no unseen movies (i.e., with no Consumed=false flags) would get labeled as Consumed=true (since only in that case there would not be anything for the user to watch during that time period). Note that, while in the current model we use CONJ, DISJ, and MAJORITY functions, other Boolean aggregation functions are also possible.

The calculation of C_O is then formally defined as follows:

- $D^* = D - \cup_i \{ d_i \} + \cup_i \{ d_i.x_i \}$. Note that, if $d_i.x_i = Root(H_i)$ then dimension d_i remains unchanged, i.e., there is no aggregation on d_i. As a result, E_i, and H_i (see below) would also remain unchanged.
- $M^* = M$.
- $\forall i = 1, \ldots, n$: $H_i^* = SubGraph(H_i, d_i.x_i)$. In other words, the attributes for the newly aggregated dimension are the ones that are uniquely determined by the new key attribute $d_i.x_i$ (i.e., that are reachable from $d_i.x_i$ in the attribute hierarchy for dimension d_i). Furthermore, after aggregation, only the hierarchy structure rooted in node $d_i.x_i$ is needed for further processing. For example, based on Time dimension attribute hierarchy (i.e., Time → DayOfWeek → TimeOfWeek), after aggregating Time dimension based on DayOfWeek the new set of attributes would be {DayOfWeek, TimeOfWeek}.

- $\forall i = 1, \ldots, n$: $E_i^* = dom(d_i.x_i)$. The cube cells along the newly aggregated dimension become labelled with the values of the new key attribute. For example, after aggregating Time dimension based on DayOfWeek, the cube cells for this dimension would be labeled as { Mon, Tue, Wed, Thu, Fri, Sat, Sun }.
- $\left(\forall address^* \in E_1^* \times \ldots \times E_n^*\right)\ (\forall j = 1...k)\ L^*[address^*].m_j = \underset{\substack{address \in L,\\ address \prec address^*}}{aggr_{f_j}} \left(L[address].m_j\right)$.

  In other words, each metric m_j is computed for each cell of the new cube using aggregation function f_j and based on cells from input cube that were replaced by (or aggregated into) a given cell. More precisely, given $address = (e_1, ..., e_n) \in E_1 \times \ldots \times E_n$ and $address^* = (e_1^*, ..., e_n^*) \in E_1^* \times \ldots \times E_n^*$, we say that $address \prec address^*$ if and only if $d_i = e_i \Rightarrow d_i.x_i = e_i^*$. Finally, note that cube cells that have NULL values are ignored during the aggregation. However, if *all* underlying cells have NULL values for a specific aggregation, then that aggregated cell will be assigned the NULL value as well.

Composition of RA operators. The recommendation algebra RA is formed by the composition of these five operators. Since each of these operators takes a rating cube and produces another rating cube, the RA algebra is closed. For example, Query 3 (recommend top 5 movies to the user to see over the weekend, but only when the personal ratings of the movies are higher than 7) can be expressed in RA as:

MRSTR$_{\text{(PersonalRating > 7)}}$(DTDM$_{\text{(Theater, Time, Companion)}}$(
AGGR$_{\text{(Theater.ALL, Time.ALL, Companion.ALL),(PersonalRating.AVG)}}$(
MRPJ$_{\text{(PublicRating, Consumed)}}$(DRSTR $_{\text{(Time.TimeOfWeek="weekend")}}$(MovieRecommender)))))

As explained before, this algebraic expression specifies only the *core* part of the REQUEST query. The actual recommendation results are generated by the REQUEST *wrapper* from the results of core query. Therefore, this algebraic expression destroys all other dimensions towards the end, leaving only the User and Movie dimensions for the wrapper to work with. Also, this example shows how MRPJ and DTDM operators *remove* measures and dimensions from the cube; e.g., PublicRating and Consumed measures as well as Theater, Time, and Companion dimensions are removed from the MovieRecommender cube.

5. Mapping REQUEST Queries into Recommendation Algebra RA

The translation of the "core" part of the REQUEST query is based on the underlying algebra RA. In particular, the mapping is performed by parsing the query and generating corresponding algebraic

operators. The MAP algorithm, presented in Figure 5, shows how to translate an arbitrary Core-REQUEST query with its specific parameters, such as various aggregations as well as measure or dimension restrictions, into an algebraic expression in RA.

As mentioned earlier, the most general form of the REQUEST query is:

REQUEST_query ::= RECOMMEND *recommend_dim_list* TO *recipient_dim_list*
USING *cube_name*
[RESTRICT *dimension_restrictions*]
[PREFILTER *preaggregation_measure_restrictions*]
BASED ON *aggr_measure_list*
[POSTFILTER *postaggregation_measure_restrictions*]
[SHOW *measure_rank_restriction*]

Based on the input query *REQUEST_query*, the MAP algorithm produces a corresponding algebraic expression *RA_op* in RA. By default, initially *RA_op* is assigned the identity operator ***ID*** (Line 1), i.e., ***ID***(*cube*) ≡ *cube* for any *cube* instance. Then, MAP continuously "grows" this initial algebraic expression *RA_op* by composing it with newly generated operators in the following way. For notational purposes, we use the ⊕ symbol to represent the composition of two algebraic operators, i.e., $op_1 \oplus op_2$ (*cube*) = $op_2(op_1(cube))$ for any *cube* and any algebraic operators op_1, op_2. In particular, first, MAP checks whether *REQUEST_query* has any restrictions on dimensions (Line 2) and, if so, MAP then generates a dimension restriction operator ***DRSTR*** with corresponding parameters (Line 3). Second, MAP checks whether *REQUEST_query* has any restrictions on measures (Line 4) and, if so, it then generates a measure restriction operator ***MRSTR*** with corresponding parameters (Line 5). Third, once dimension and measure restrictions are applied, the aggregation is performed next. The measures to be aggregated and their aggregation functions are specified by the user in the BASED ON clause of the query, but first the unused measures (measures that do not appear in this clause) are projected out using operator ***MRPJ*** (Lines 6-7).

Subsequently, operator ***AGGR*** is generated (Line 17) with parameters *dimension_aggregations* and *measure_aggregations*, where the former specifies the granularity (or aggregation) levels for all dimensions that need to be grouped (Lines 8-13) and the latter specifies aggregation functions for all measures (Lines 14-16). Fourth, all the irrelevant (and fully aggregated) dimensions, i.e., the dimensions that do not appear in RECOMMEND and TO clauses, are destroyed using operator ***DTDM*** (Lines 18-19).

Fifth, MAP checks whether *REQUEST_query* has any post-aggregation restrictions on measures (Line 20) and if so, it then generates a measure restriction operator ***MRSTR*** with corresponding parameters (Line 21). Finally, Line 22 returns the resulting algebraic expression *RA_op*, and the query results can be obtained by applying *RA_op* to the input cube *cube_name* specified in the USING clause.

MAP(*REQUEST_query*) **{**

(1) *RA_op* := ***ID***

(2) if (∃ RESTRICT clause in *REQUEST_query*) then

(3) *RA_op* := *RA_op* ⊕ $\boldsymbol{DRSTR}_{(dimension\_restrictions)}$

(4) if (∃ PREFILTER clause in *REQUEST_query*) then

(5) *RA_op* := *RA_op* ⊕ $\boldsymbol{MRSTR}_{(preaggregation\_measure\_restrictions)}$

(6) foreach $m \notin aggr\_measure\_list$ in BASED ON clause

(7) *RA_op* := *RA_op* ⊕ $\boldsymbol{MRPJ}_m$

(8) *dimension_aggregations* = ∅

(9) foreach $d_i \in cube\_name$

(10) if $d_i \notin recommend\_dim\_list \cup recipient\_dim\_list$ then

(11) *dimension_aggregations* := *dimension_aggregations* ∪ { d_i.ALL }

(12) else if (∃x) $d_i.x \in recommend\_dim\_list \cup recipient\_dim\_list$ then

(13) *dimension_aggregations* := *dimension_aggregations* ∪ { $d_i.x$ }

(14) *measure_aggregations* = ∅

(15) foreach (m_j, $aggr_j$) ∈ *aggr_measure_list* in BASED ON clause

(16) *measure_aggregations* := *measure_aggregations* ∪ { $m_j.aggr_j$ }

(17) *RA_op* := *RA_op* ⊕ $\boldsymbol{AGGR}_{(dimension\_aggregations),(measure\_aggregations)}$

(18) foreach $d \notin recommend\_dim\_list \cup recipient\_dim\_list$

(19) *RA_op* := *RA_op* ⊕ $\boldsymbol{DTDM}_d$

(20) if (∃ POSTFILTER clause in *REQUEST_query*) then

(21) *RA_op* := *RA_op* ⊕ $\boldsymbol{MRSTR}_{(postaggregation\_measure\_restrictions)}$

(22) return *RA_op*;

}

Figure 5. Mapping Core-REQUEST queries into RA expressions.

We next explore a formal relationship between the Core-REQUEST queries and RA. To do this, we first introduce some preliminary concepts. Let *o* be a specific instance of any of the five RA operators, for example, *o* = $\text{DRSTR}_{(\text{Movie.Genre} = \text{“comedy”})}$. Given recommendation cube *C*, we say that *o* is a *well-defined operation* for *C* if *o*(*C*) can be successfully performed based on the schema of cube *C* as well as the dimensions, attributes, and measures specified in operator *o*. For example, operator $\text{DRSTR}_{(\text{Movie.Genre} = \text{“comedy”})}$ is well-defined for any cube that has dimension *Movie* with an attribute *Genre* and is not well-defined for any other cube.

The notion of a well-defined operation can be directly extended from a single algebraic operator to

sequences of operators. Let s be a sequence of RA operators, i.e., $s = \langle o_1, \ldots, o_n \rangle$, where each o_i is a specific instance of any of the five RA operators. We say that s is a *well-defined operation sequence* for cube C if operation $o_n(o_{n-1}(\ldots(o_2(o_1(C)))))$ can be successfully performed in the sense that each operator o_i in the sequence is well-defined for its input cube: operator o_1 is well-defined for cube C, o_2 is well-defined for cube $o_1(C)$, etc., i.e., o_i is well-defined for cube $o_{i-1}(\ldots(o_1(C)))$ for each $i = 2, \ldots, n$.

Lemma 1 [safe swap of DRSTR forward]. Let o_{DRSTR} be an instance of the DRTSR operator, o_{ANY} be an instance of *any* of the five recommendation algebra operators (i.e., DRSTR, MRSTR, MRPJ, DTDM, AGGR), and C be a recommendation cube where $o_{\text{DRSTR}}(o_{\text{ANY}}(C))$ is well-defined. Then, $o_{\text{ANY}}(o_{\text{DRSTR}}(C))$ is also well-defined and $o_{\text{DRSTR}}(o_{\text{ANY}}(C)) = o_{\text{ANY}}(o_{\text{DRSTR}}(C))$.

Proof. Immediate from the definitions of RA operators. ■

Lemma 2 [safe swap of DTDM back]. Let o_{DTDM} be an instance of the DTDM operator, o_{ANY} be an instance of *any* of the five RA operators, and C be a recommendation cube where $o_{\text{ANY}}(o_{\text{DTDM}}(C))$ is well-defined. Then, $o_{\text{DTDM}}(o_{\text{ANY}}(C))$ is also well-defined and $o_{\text{ANY}}(o_{\text{DTDM}}(C)) = o_{\text{DTDM}}(o_{\text{ANY}}(C))$.

Proof. Immediate from the definitions of RA operators. ■

Lemma 3 [safe swap of MRPJ back]. Let o_{MRPJ} be an instance of the MRPJ operator, o_{ANY} be an instance of *any* of the five RA operators, and C be a recommendation cube where $o_{\text{ANY}}(o_{\text{MRPJ}}(C))$ is well-defined. Then, $o_{\text{MRPJ}}(o_{\text{ANY}}(C))$ is also well-defined and $o_{\text{ANY}}(o_{\text{MRPJ}}(C)) = o_{\text{MRPJ}}(o_{\text{ANY}}(C))$.

Proof. Immediate from the definitions of RA operators. ■

Theorem 1 [canonical form of recommendation algebra]. Let C be a recommendation cube. Then, for *any* sequence s of recommendation algebra operators, such that $s(C)$ is a well-defined operation, there exists a corresponding canonical sequence s' of the form:[2]

$$s' = \langle [\text{DRSTR}],[\text{MRSTR}],(\text{AGGR},[\text{MRSTR}])^*,[\text{DTDM}],[\text{MRPJ}] \rangle \qquad (1)$$

that is equivalent to $s(C)$, i.e., $s'(C) = s(C)$, and where the number of AGGR operators in s' is equal to the number of AGGR operators in s.

[2] Using traditional notation, the square brackets denote that a particular operator is optional, and the star symbol (*) in (AGGR,[MRSTR])* denotes zero, one, or more repetitions of AGGR and, possibly, MRSTR operators.

Proof. The proof is provided in the Appendix. ■

We next establish the relationship between Core-REQUEST and RA.

Theorem 2. RA is strictly more expressive than Core-REQUEST.

Proof. Obviously, RA is at least as expressive as the Core-REQUEST language, because every Core-REQUEST statement can be expressed in RA using the MAP algorithm. Furthermore, directly from the MAP algorithm we have that all Core-REQUEST queries are of the following algebraic form: <[DRSTR],[MRSTR],[MRPJ],[AGGR],[DTDM],[MRSTR]>. Based on Lemmas 2 and 3 and also on the fact that, in the absence of AGGR operator, PREFILTER and POSTFILTER can be represented as one measure restriction operation (as mentioned in Section 3), all Core-REQUEST queries can also be expressed by the following equivalent sequence: <[DRSTR],[MRSTR],[AGGR,[MRSTR]],[DTDM], [MRPJ]>. Based on Theorem 1, it is clear that RA can produce the expressions of a strictly more general form, i.e., <[DRSTR], [MRSTR],(AGGR,[MRSTR])*,[DTDM],[MRPJ]>, where the precise difference in expressive power lies in the RA's ability to specify multiple <AGGR, [MRSTR]> operator sequences (as opposed to only 0 or 1 such sequences in Core-REQUEST). ■

Theorems 1 and 2 explain the difference between expressive powers of RA and Core-REQUEST at the theoretical level: Core-REQUEST allows *at most one* aggregation operation in a query, while RA supports *multiple* aggregations because the algebra is *closed.* For example, in a 2-dimensional recommendation application with User and Movie dimensions and one measure, Rating, the following RA expression

$$\text{AGGR}_{(\text{User.ALL, Movie.ALL}),(\text{Rating.AVG})}(\text{MRSTR}_{(\text{Rating} >7)}(\text{AGGR}_{(\text{User.Profession, Movie.Genre}),(\text{Rating.AVG})}(C)))$$

cannot be expressed in Core-REQUEST.

One way to address the issue that Core-REQUEST is strictly less expressive than RA is to extend REQUEST in such a way that their expressive powers would become equal. According to Theorems 1 and 2, this would mean providing support for multiple aggregations (and optional measure restriction capabilities after each aggregation) in REQUEST. One way to achieve this is by supporting an arbitrary

number of Core-REQUEST query compositions (e.g., that could be implemented via nested queries). It immediately follows from Theorems 1 and 2 and Lemmas 1-3 that such an extended language would have the same expressive power as algebra RA. Upon a careful consideration, however, we decided against this extension when designing REQUEST because multiple aggregations (a) do not occur naturally in recommendation applications and, therefore, have a very limited need in the real-world applications, and (b) unnecessarily complicate the language design by adding extra complexity needed for allowing an arbitrary number of aggregations. These extensions would make REQUEST significantly less user-friendly without providing any tangible benefits. Note also that, aside from these query compositions with two or more aggregation operators (which cannot be expressed in Core-REQUEST), composition of any other Core-REQUEST queries (i.e., having zero or one aggregations among them) can always be expressed in Core-REQUEST, according to Theorem 1, Lemmas 1-3, and the definitions of RA operators.

6. Conclusions

In this paper we introduced language REQUEST for specifying user-driven recommendations. REQUEST queries are formulated on multidimensional cubes of ratings, support OLAP-based aggregation capabilities, are expressed in a simple declarative language capturing idiosyncrasies of recommender systems, and thus provide several advantages to the users of recommender systems. In particular, REQUEST empowers the end-users by letting them customize recommendations by formulating them in the ways that satisfy their individual needs in a flexible and user-friendly manner. Also, unlike SQL, which constitutes a general-purpose database query language, REQUEST is designed specifically for multidimensional recommender systems. Therefore, its constructs are developed exclusively for specific recommendation contexts, and every REQUEST query can be directly interpreted as a recommendation. As a result, REQUEST can express complex recommendations in a concise and clear manner. Finally, REQUEST design follows the multidimensional data model and does not depend on any of its particular implementations (one can use ROLAP, MOLAP, hybrid OLAP, or even non-OLAP-based approaches to implement multidimensional recommender systems).

We also presented an OLAP-based recommendation algebra, showed how REQUEST

recommendations can be expressed in it, and provided analysis of its expressiveness. Therefore, REQUEST queries can be processed using this mapping similarly to how SQL queries are processed in relational DBMSes. One query processing problem pertaining to recommender systems deals with the determination of which new ratings need to be evaluated in order to answer a particular REQUEST query, in case the entire cube of ratings cannot be pre-computed ahead of time. For example, in order to answer the query "which movies to recommend to Jane Doe to see on March 5 on Saturday night with her boyfriend in a movie theatre," the system may not need to estimate *all* the ratings on-the-fly in the recommendation cube described in Example 1. Since rating estimation in such cases becomes query-dependent, an interesting and challenging problem is to determine the subset of ratings that needs to be estimated to answer a given query. We plan to study this problem in the future.

Finally, it is important to develop a good GUI-based front-end to REQUEST so that naïve end-users would be able to express their user-driven recommendations using this interface. Designing such interface constitutes another topic of our future research.

References

[1] Agrawal, R., A. Gupta, and S. Sarawagi S. Modeling multidimensional databases. In *Proc. of the International Conf. on Data Engineering*, pp. 232-243, 1997.

[2] Adomavicius, G. and Tuzhilin, A. Incorporating Context into Recommender Systems Using Multidimensional Rating Estimation Methods. *Proc. of WPRSIUI Workshop*, 2005.

[3] Adomavicius, G. and Tuzhilin, A. Towards the next generation of recommender systems: A survey of the state-of-the-art and possible extensions. *IEEE Transactions on Knowledge and Data Engineering*, 17(6):734-749, June 2005.

[4] Adomavicius, G., R. Sankaranarayanan, S. Sen, and Tuzhilin, A. Incorporating Contextual Information in Recommender Systems Using a Multidimensional Approach. *ACM Trans. on Information Systems*, 23(1):103-145, January 2005.

[5] Adomavicius, G. and Tuzhilin, A. Multidimensional Recommender Systems: A Data Warehousing Approach. L. Fiege, G. Mühl, and U. Wilhelm (Eds.), *Electronic Commerce: Second International Workshop, WELCOM 2001*, LNCS, vol. 2232, pp. 180-192, 2001.

[6] Bennet, J. and Lanning, S. The Netflix Prize. In *SIGKDD Conference*, 2007.

[7] Balabanovic, M. and Y. Shoham. Fab: Content-based, collaborative recommendation. *Communications of the ACM*, 40(3):66-72, 1997

[8] Ceri, S. and G. Gottlob. Translating SQL into relational algebra: Optimization, semantics and equivalence of SQL queries. *IEEE Trans. on Soft. Engineering*, 11(4):324-345, 1985.

[9] Chaudhuri, S. and U. Dayal. An overview of data warehousing and OLAP technology. *ACM SIGMOD Record*, 26(1):65-74, 1997.

[10] Cohen, W. W., R. E. Schapire, and Y. Singer. Learning to order things. *Journal of Articial Intelligence Research*, 10:243-270, 1999.

[11] Gyssens, M and L.V.S. Lakshmanan. A foundation for multi-dimensional databases. In *Proc. of the International Conf. on Very Large Data Bases (VLDB-97)*, pp. 106-115, 1997.

[12] Hill, W, L. Stead, M. Rosenstein, and G. Furnas. Recommending and evaluating choices in a virtual community of use. In *Proc. of the Conf. on Human Factors in Comp. Syst.*, 1995.

[13] Jameson, A. and B. Smyth. Recommendation to Groups. In Brusilovsky, P., Kobsa, A. and Nejdl, W. eds. *Adaptive Web: Methods and Strategies of Web Personalization*, Springer, Berlin, 2007.

[14] Kimball, R. *The Data Warehouse Toolkit*. John Wiley & Sons, Inc. 1996.

[15] Koutrika, G., R. Ikeda, B. Bercovitz, H. Garcia-Molina. Flexible Recommendations over Rich Data. In *Proceedings of the 2008 ACM Conference on Recommender Systems (RecSys'08)*, pp. 203-210, Lausanne, Switzerland, 2008.

[16] Li, C. and X. Sean Wang. A Data Model for Supporting On-Line Analytical Processing. In *Proc. of the Conf. on Information and Knowledge Management (CIKM-1996)*, 1996.

[17] Marcel, P. Modeling and querying multidimensional databases: an overview. *Networking and Information Systems Journal*, 2(5):515--548, 1999.

[18] Mild, A and Reutterer, T. Collaborative Filtering Methods for Binary Market Basket Data Analysis. *Lecture Notes in Computer Science*, 2252: 302, 2001.

[19] Miller, B. N., I. Albert, S. K. Lam, J. A. Konstan, and J. Riedl. MovieLens Unplugged: Experiences with an Occasionally Connected Recommender System. In *Proc. of the International Conference on Intelligent User Interfaces*, 2003.

[20] Ramakrishnan, R. and J. Gehrke. *Database Management Systems*, McGraw-Hill, 2000.

[21] Resnick, P., N. Iakovou, M. Sushak, P. Bergstrom, and J. Riedl. GroupLens: An open architecture for collaborative filtering of netnews. In *Proc. of CSCW*, 1994.

[22] Shardanand, U. and P. Maes. Social information filtering: Algorithms for automating 'word of mouth'. In *Proc. of the Conf. on Human Factors in Computing Systems*, 1995.

[23] Snodgrass, R. The temporal query language TQuel. *ACM Transactions on Database Systems*, 12(2):247-298, 1987.

[24] Thomas, H. and A. Datta. A conceptual model and algebra for On-Line analytical processing in decision support databases, *Information Sys. Research*, 12(1), 2001.

[25] Umyarov, A. and A. Tuzhilin. Improving Collaborative Filtering Recommendations Using External Data, In *Proceedings of the IEEE International Conference on Data Mining (ICDM-2008)*, Pisa, Italy, 2008.

[26] Wade, W. A grocery cart that holds bread, butter and preferences. *NY Times*, Jan 16, 2003.

Appendix. Canonical Form of Recommendation Algebra: Proof of Theorem 1.

Theorem 1 [canonical form of recommendation algebra]. Let C be a recommendation cube. Then, for *any* sequence s of recommendation algebra operators, such that $s(C)$ is a well-defined operation, there exists a corresponding canonical sequence s' of the form:

$$s' = \text{<[DRSTR],[MRSTR],(AGGR,[MRSTR])*,[DTDM],[MRPJ]>}$$

that is equivalent to $s(C)$, i.e., $s'(C) = s(C)$, and where the number of AGGR operators in s' is equal to the number of AGGR operators in s.

Proof. The proof is by construction and follows from Lemmas 1-3. First, let $s' := s$. Based on Lemma 1, all DRSTR operators (if any present in sequence s) can be moved to the front of the sequence by repeatedly swapping these operators with preceding ones, without affecting the result of the overall algebraic operation. Similarly, based on Lemmas 2 and 3, all DTDM and MRPJ operators (if any) can be safely moved to the end of the sequence. Furthermore, if there are several DRSTR operators in s, after being moved to the beginning of the sequence, they can all be combined into a single DRSTR operator (as discussed in Section 4): $\text{DRSTR}_{p1}(\text{DRSTR}_{p2}(\ \ldots(\text{DRSTR}_{px}(C))\ldots)) = \text{DRSTR}_{p1\ AND\ \ldots\ AND\ px}(C)$. Analogous statements can also be made for the multiple DTDM as well as multiple MRPJ operators (as discussed in Section 4). And if there are no DRSTR (or DTDM, or MRPJ) operators in s, they will be omitted in s' as well. Thus, at this point we have that s' is of the form: <[DRSTR], $s_{\text{AGGR,MRTSR}}$, [DTDM],[MRPJ]>, where $s_{\text{AGGR,MRTSR}}$ is the sub-sequence of only AGGR and MRSTR operators present in s (in their original order).

Because generally one cannot safely swap two instances of AGGR operator (as illustrated in Figure A1), the exact sequencing of AGGR operator instances in s must be preserved in s'. Thus, s' will have the same exact AGGR operators as in s (and in the same sequence). Furthermore, because generally one

cannot safely swap instances of AGGR and MRSTR operators (as illustrated in Figure A2), the relative sequencing between AGGR and MRSTR also is preserved in s'. As a result, $s_{AGGR,MRTSR}$ can always be expressed with the following equivalent pattern: <[MRSTR], (AGGR,[MRSTR])*>, since adjacent MRSTR operators (if any present) can always be combined into one MRSTR operator.

We have constructed sequence s' of the form <[DRSTR],[MRSTR],(AGGR,[MRSTR])*, [DTDM],[MRPJ]> only by safely swapping adjacent operators. Thus, $s'(C) = s(C)$, and the number of AGGR operators in s' is equal to the number of AGGR operators in s. ■

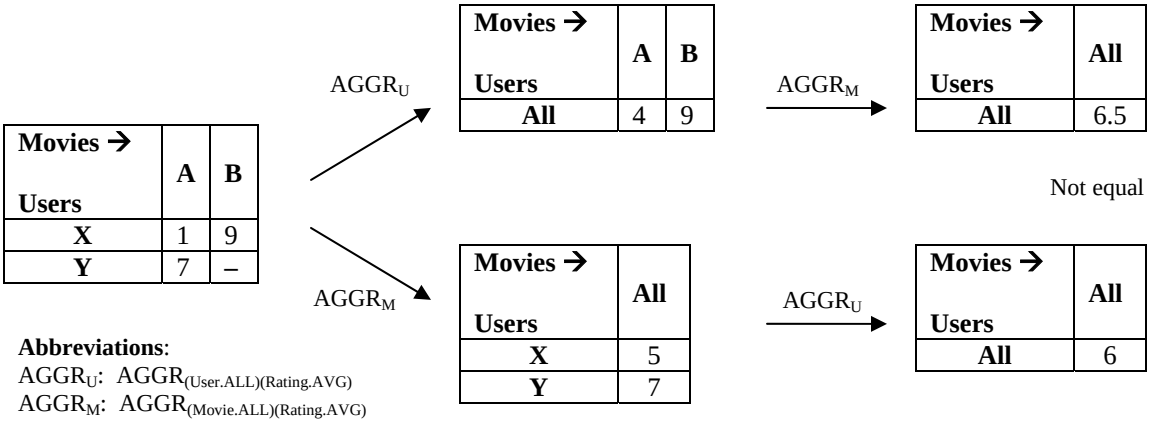


Figure A1. Non-safe swapping of two AGGR instances.

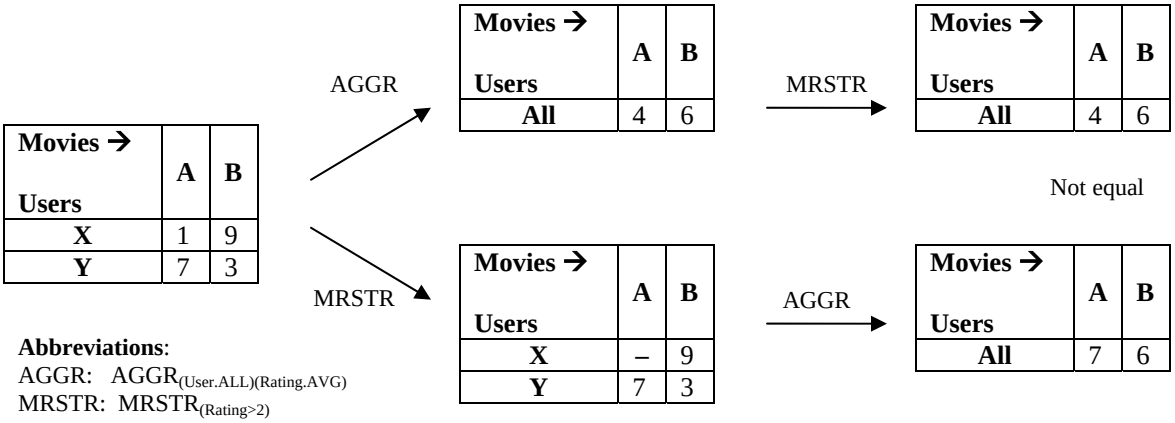


Figure A2. Non-safe swapping of the AGGR and MRSTR instances.