RQL: A Query Language for Recommender Systems

Gediminas Adomavicius[1], Alexander Tuzhilin[2], and Rong Zheng[2]

[1] Information and Decision Sciences
Carlson School of Management
University of Minnesota
gedas@umn.edu

[2] Information, Operations, and Management Sci.
Stern School of Business
New York University
{atuzhili,rzheng}@stern.nyu.edu

Abstract

Initially popularized by Amazon.com, recommendation technologies have become widespread over the past several years, both in the industry and academia. The traditional two-dimensional approach to recommender systems, involving the dimensions of Users and Items, has been subsequently extended to the multidimensional approach supporting additional contextual dimensions and OLAP-type aggregation capabilities. Furthermore, the class of all possible recommendations available to the users in traditional recommender systems is typically determined by the vendor and is quite limited. In this paper we address this limitation by proposing a query language RQL that allows the users to formulate various types of recommendation requests on their own. RQL adapts OLAP queries to the domain of recommender systems and, therefore, is able to support both the traditional two-dimensional and the more complex multidimensional recommender systems. The paper also presents a recommendation algebra that allows mapping RQL queries into the algebraic expressions for the query processing purposes. Finally, the paper presents a method for executing RQL queries.

1. Introduction

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, 7, 13], especially after these technologies were popularized by Amazon.com, Netflix, and other companies. A recent comprehensive survey of the rapidly growing field of recommender systems can be found in [2].

Most of the work in recommender systems focuses on a two-dimensional paradigm of recommending items to users or users to items (e.g., finding the most relevant books to a specific customer or the most likely buyers for a specific book). Although the two-dimensional paradigm is suitable for some applications, such as recommending books and music CDs, it is significantly less suitable for other applications, such as recommending vacations to travellers or recommending groceries to purchase by a "smart" shopping cart using wireless location-based technologies [16]. For example, one would recommend a different vacation to a customer in the winter than in the summer. Similarly, a "smart" shopping cart, providing real-time recommendations to shoppers, needs to take into account not only information about products and customers but also such contextual information as shopping date/time, store, who accompanies the primary shopper, products already placed into the shopping cart, and its location within the store.

To provide recommendations in such "contextually rich" applications, one needs to consider other dimensions, besides Items and Users, such as Time, Location and Companion (i.e., accompanying person). In [3], we proposed a new *multidimensional* approach to recommender systems, in which we incorporated multiple dimensions and the OLAP-based multidimensional cubes of ratings into the recommendation model. We also proposed a novel *reduction-based* method for estimating missing ratings in multidimensional cubes and showed that this method outperforms a traditional collaborative filtering method [3].

However, the approach described in [3] 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 the needs of the users. For example a typical recommender system would recommend top k items to a user or a set of users, or would recommend the best k users who should be the primary targets for a certain product. This situation is quite limited, especially in the multidimensional settings, where the number of possible recommendations increases significantly with the number of dimensions.

To address the problem of limited and restricted recommendation capabilities in the current generation of recommender systems, we present *query language RQL* that allows end-users to express a broad range of recommendations. RQL empowers the users and gives them the flexibility to request recommendations that are of interest to them. For example, we may want to recommend the best two *times* to go on vacation to Jamaica to Jane Doe and her boyfriend. This recommendation can be expressed in RQL as

```
RECOMMEND  Time TO Customer
FROM  VacationCube
BASED ON  AVG(Ratings)
WHERE  Customer.Name = "Jane Doe" AND
       Companion.Type = "Boyfriend" AND
       TravelDestination.Region = "Jamaica"
AGGR BY  TravelDestination.Region
SHOW  TOP 2
```

As this example shows, RQL has an SQL-like syntax that is adapted to the context of recommender systems and, specifically, to the OLAP-based multidimensional recommendation model (based on the data cube). In particular, this query deals with the Customer, Time, TravelDestination, and Companion dimensions, with a set of ratings stored in ratings cube VacationCube, and shows that RQL can support multidimensional recommender systems. One benefit of RQL being based on the declarative paradigms of SQL and OLAP is that it hides many implementation-related details from the end-user and allows the user to express recommendations in a declarative and, therefore, more intuitive manner. For example, since there are many resorts in Jamaica, in order to recommend the best two travel times to Jane to Jamaica, in the implementation of the above query it is necessary to average the resort ratings at a given time and select the best two times based on these average ratings.

Besides RQL, we also present a multidimensional recommendation algebra that is used for processing certain "core" parts of RQL queries. Finally, we describe how these core parts of RQL queries can be processed by mapping them into this algebra, then into the relational algebra and, finally, into SQL. We will also demonstrate that this mapping can be inefficient in some cases. Therefore, good query optimization methods need to be deployed along the way to speed the performance of the resulting SQL queries.

Although RQL provides a general method for expressing recommendations by the end users, in this paper we followed the relational OLAP (ROLAP) approach for implementing RQL by mapping RQL queries into SQL. However, query processing implementations of RQL are not limited to ROLAP and can also include multidimensional OLAP (MOLAP) approaches when RQL queries are directly evaluated on the rating cubes.

As with most query languages, including SQL, the end-users of RQL can be divided into two main groups: the naïve- and the power-users. As with SQL, the power-users can express their recommendations directly in RQL. However, this task can be difficult for the naïve users. Therefore, it is necessary to provide them with a form-based user interface (UI) that would simplify the process of formulating RQL queries. Moreover, as in the case of classical query languages and their form-based UIs (such as Microsoft Access UI and underlying SQL), there should be a tight coupling between the semantics of the graphical recommendation interface and RQL so that the simple and direct translation between the two is possible. This means that RQL is necessary not only for the power users, but also as a guide for designing proper semantics of the form-based UI used by the naïve users.[1]

The rest of the paper is organized as follows. Section 2 provides background information on multidimensional recommendation systems. Section 3 presents the RQL language, and Section 4 describes the corresponding recommendation algebra RA. Section 5 describes how RQL queries are processed, and the conclusions and the future research directions are presented in Section 6.

2. Background: Multidimensional Recommender Systems

In addition to the standard *User* and *Item* dimensions of the traditional recommender systems, the multidimensional approach considers other contextual dimensions [3]. Formally, given a *recommendation space* $S = D_1 \times D_2 \times \ldots \times D_n$ consisting of dimensions $D_1, D_2, \ldots, D_n$, where D_i is a subset of a Cartesian product of some attributes A_{ij}, $(j = 1,\ldots,k_i)$, i.e., $D_i \subseteq A_{i1} \times A_{i2} \times \ldots \times A_{ik_i}$, and a *partially defined* rating function R: $D_1 \times \ldots \times D_n \rightarrow$ *Rating* mapping dimensions into an ordered discrete finite set of ratings, the recommendation problem is defined as follows. First, the system needs to estimate the unknown ratings and make the rating function R total [3]. Second, in order to make some recommendation, one needs to select certain "what" dimensions $D_{i1}, \ldots, D_{ik}$ $(k < n)$ and certain "for whom" dimensions $D_{j1}, \ldots, D_{jl}$ $(l < n)$ that do not overlap, i.e., $\{D_{i1}, \ldots, D_{ik}\} \cap \{D_{j1}, \ldots, D_{jl}\} = \varnothing$, and recommend for each tuple $(d_{j1}, \ldots, d_{jl}) \in D_{j1} \times \ldots \times D_{jl}$ the tuple $(d_{i1}, \ldots, d_{ik}) \in D_{i1} \times \ldots \times D_{ik}$ that maximizes the rating $R(d_1,\ldots,d_n)$ across all the tuples $(d_1,\ldots,d_n)$ that coincide with $(d_{j1}, \ldots, d_{jl}) \in D_{j1} \times \ldots \times D_{jl}$ on dimensions $D_{j1}, \ldots, D_{jl}$.

Example 1. Consider the application for recommending movies to users and having the following dimensions:

- *Movie:* the set of all the movies that can be recommended; it is defined as Movie(MovieID, Name, Studio, Director, Year, Genre, MainActors).
- *User:* the people to whom movies are recommended; it is defined as User(UserID, Name, Address, Age, Occupation, etc.).
- *Theater:* the movie theaters showing the movies; it is defined as Theater(TheaterID, Name, Address,

[1] We would like to point out that SQL is not suitable for defining the model-theoretic semantics of this form-based recommendation interface, because the semantics in this case should be defined in terms of the *multidimensional* recommendation model rather than the relational model. This point will be discussed further in Section 3.

Capacity, Location, County, State).

- *Time:* the time when the movie can be or has been seen; it is defined as Time(TimeOfDay, DayOfWeek, Month, Year).
- *Companion:* represents a person or a group of persons with whom one can see the movie. Companion consists of a *single* attribute having values "alone," "friends," "girlfriend/boyfriend," "family," "co-workers," and "others."

We also use *two* rating measures in this example: PublicRating, specifying how much the general public liked the movie, and PersonalRating, specifying how much a particular person liked the movie in the settings specified by the Time, MovieTheater, and Companion dimensions. 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. For example, the type of a movie to recommend to college student Jane Doe can differ significantly depending on whether she is planning to see it on a Saturday night with her boyfriend in a movie theater as opposed to on a weekday with her parents at home. ■

The ratings $R(d_1, \ldots, d_n)$ of the recommendation space $S = D_1 \times D_2 \times \ldots \times D_n$ are either explicitly provided by the users or are implicitly inferred by the system [2] and are stored in a partially filled multidimensional cube. For example, R(Aviator, Jane, UA-NYC-34St, 2/19/2005, boyfriend) = 6 means that Jane gave rating 6 to "Aviator" that she saw with her boyfriend on February 19, 2005 in the movie theater UA-NYC-34St.

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 [3], 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 a Saturday night in a movie theater. If the Time dimension is partitioned into weekend and weekday components and since Saturday night falls on a weekend, the reduction-based approach uses *only* the ratings for the movies seen on weekends by customers with their boy/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 [3].

Finally, the multidimensional recommendation model allows for OLAP-based aggregation hierarchies [3]. For example, movies can be classified into genres, people into segments of customers, and time has its standard classification hierarchies (such as day of week, month, year, etc.). These hierarchies help aggregate ratings according to the methods described in [3]. The aggregation capabilities can be used in RQL when formulating complex recommendations.

The work described in [3] focuses on the multidimensional recommendation model and on the reduction-based approach to estimating unknown ratings. However, it does not specify how to *express* the wide variety of recommendations that are possible in multidimensional settings, and simply assumes that the recommendation algorithms are "hard-wired" into the system by their developers. In the next section we address this limitation by presenting query language RQL for expressing such recommendations.

3. Recommendation Query Language (RQL)

In this section, we describe RQL by first providing several examples of RQL queries demonstrating various features of the language and then presenting a more formal definition of the language. As pointed out in Section 1, RQL in its text-based form is designed primarily for the power users. Later in this section we will also discuss a UI for the naïve users that allows expressing recommendations using a form-based approach.

The first example presents a classic recommendation request supported by most recommender systems.

Query 1: Recommend best movies to users:

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

In Query 1, the RECOMMEND clause specifies that movies will be recommended to users, the FROM clause specifies that the recommendation is based on the 5-dimensional MovieRecommender cube of ratings, and the BASED ON clause specifies that personal ratings are used for the recommendation purposes. In particular, the movies 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. Some of the specifics of this process will be described further when discussing Query 7. Finally, the best movie is selected for the user based on the movie ordering.

The next example (Query 2) demonstrates the SHOW clause and how to use restrictions on recommendation criteria. In Query 2, the WHERE clause is used to select specific movies and users satisfying the selection criteria. Then only the selected movies are ordered for each selected user based on the value of PersonalRating measure and the top 5 movies are recommended for each user. We would like to point out that in Query 1 the SHOW TOP 1 clause was omitted by default when only the best movie was selected for each user.

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

```
RECOMMEND  Movie TO User
FROM  MovieRecommender
BASED ON  PersonalRating
WHERE  Movie.Genre = "Action" AND
    User.Age >= 18
SHOW  TOP 5
```

The next example shows how ratings are selected based on the criteria specified in the WITH clause.

Query 3: Recommend top 5 movies to the user to see over the weekend, but only if the 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
FROM  MovieRecommender
BASED ON  PersonalRating
WHERE  Time.WeekTime ="Weekend"
WITH  PersonalRating > 7
SHOW  TOP 5
```

Query 3 demonstrates that different selection clauses (WHERE and WITH) are used for the selections of attributes and ratings. There are two reasons for providing different types of restrictions for attributes of dimensions and rating measures with separate WHERE and WITH clauses. First, at the OLAP level, restrictions on attributes are implemented by slice and dice operation [5, 8], whereas restrictions on rating measures are implemented by extracting particular rating information from the cells of the cube. Second, it is useful to distinguish the two sets of restrictions because they are semantically very different: the first provides restrictions on the contextual information, while the other on the quality of target concept, e.g., how good the movie is. Also, this is not unlike the situation in temporal databases, where separate WHERE and WHEN clauses are used for regular and temporal dimensions [14].

The next example shows that more than one dimension can be used in recommendations.

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
FROM  MovieRecommender
BASED ON  PersonalRating
WHERE  User.Name = "Tom" AND
    Time.WeekTime ="Weekend" AND
    Companion.Type = "Girlfriend"
SHOW  TOP 3
```

Note that the RECOMMEND clause in Query 4 has two dimensions (Movie and Time) and the TO clause has also two dimensions (User and Companion).

Sometimes, a certain *group* of people may be interested in a certain *genre* of movies. These types of recommendations can be achieved using aggregation capabilities of RQL, as the next example shows.

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

```
RECOMMEND  Movie.Genre TO User.Profession
FROM  MovieRecommender
BASED ON  AVG (PersonalRating)
WITH  PersonalRating > 6
AGGR BY  Movie.Genre, User.Profession
```

This query aggregates rating scores for individual movies into aggregated rating scores for different genres of movies. The aggregated score is calculated by averaging individual scores using the AVG function in the BASED ON clause. Besides AVG, other aggregation functions may include SUM, MAX, MIN, etc. Similarly, in this example individual users are also aggregated by profession, and each profession becomes a new target for a recommendation. Any aggregation operation is specified with the AGGR BY clause.

The next example demonstrates that recommendations are not restricted to the User dimension. More generally, Query 6 show how 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
FROM  MovieRecommender
BASED ON AVG(PersonalRating)
WHERE  Movie.Title= "Beautiful Mind"
AGGR BY  User.Profession
SHOW  TOP 2
```

A rating score for a movie can either be *explicitly specified* by the user or can be *estimated* from the existing user-specified ratings using one of the rating estimation methods described in [3]. The next query (Query 7) demonstrates this feature of RQL as well as the ability to provide recommendations based on multiple ratings.

Query 7: 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
FROM  MovieRecommender
BASED ON PersonalRating, PublicRating
WHERE  User.Profession = "Student"
WITH  Public_Rating>8 AND
    RATED (PersonalRating) >8
SHOW TOP 5
```

Note that, since in a recommender system ratings can be provided by users or estimated by software, for each measure we have an option of specifying the EST rating flag to indicate which values for this particular measure have been specified by the users (e.g., users saw a movie and provided their ratings) and which have been estimated by some recommender system. In Query 7, only previously seen movies for each user are used in the recommendation.

More specifically, Query 7 demonstrates how RQL differentiates between explicitly specified and estimated ratings by supporting RATED, ESTIMATED, and ALL qualifiers. If the RATED qualifier is specified, then only the *actual* ratings (i.e., ratings for which EST rating flag = False) are used to compute recommendations. If the ESTIMATED qualifier is used, then only the *estimated* ratings (i.e., where EST rating flag = True) are used for this purpose. If omitted, one of these qualifiers is set as a default (e.g., ESTIMATED).[2] If the ALL qualifier is used then both actual and estimated ratings are used to compute recommendations (i.e., EST rating flag is ignored).

Query 7 also shows how two separate measures are used in RQL and how restrictions are imposed on them. Also, when multiple measures are used in the BASED ON clause, lexicographic ordering is used to order recommendation results.

The next example demonstrates the usage of the HAVING clause and multiple aggregations in RQL.

Query 8: Recommend movie genre to different professions and show those results with average ratings bigger than 6:

```
RECOMMEND  Movie.Genre TO User.Profession
FROM  MovieRecommender
BASED ON  AVG (PersonalRating)
AGGR BY  Movie.Genre, User.Profession
HAVING  AVG (PersonalRating) > 6
```

In Query 8, AGGR BY and HAVING clauses are analogous to GROUP BY and HAVING clauses in SQL. Also, in this example the aggregated average rating scores are restricted to those that are bigger than 6.

Examples 1–8 of RQL queries introduced various features of the language. A more formal top-level specification of RQL syntax is presented in Figure 1.

As Figure 1 shows, recommendations are restricted to a single cube of ratings *cube*, thus disallowing joins between cubes in RQL. This is the case because multicube recommendations seldom have meaningful and practically important applications, but lead to many complications and side-effects.

The *recommend_dim_attr_list* and *recipient_dim_attr_list* provide the *mutually exclusive* lists of either dimensions (e.g., Time, User, etc.) or attributes (e.g., User.Profession, Movie.Genre, etc.). Moreover, if RQL has an aggregated recommendation in the RECOMMEND-TO clause (i.e., it appears in *recommend_dim_attr_list* or in *recipient_dim_attr_list*), it must also be specified in the AGGR BY clause (i.e., appear in *aggregation_dim_attr_list*). [3] Finally, the *aggregation_dim_attr_list* has the same structure as the *recommend_dim_attr_list* and *recipient_dim_attr_list*.

```
RECOMMEND recommend_dim_attr_list
TO recipient_dim_attr_list
FROM cube
BASED ON measure_list
WHERE dimension_restrictions                //optional
WITH measure_restrictions                   //optional
AGGR BY aggregation_dim_attr_list           //optional
HAVING aggregation_restriction              //optional
SHOW measure_rank_restriction
                  //optional, default: SHOW TOP 1
```

Figure 1. Top-Level Specification of RQL Syntax.

The WHERE clause contains *dimension_restrictions* that constitute the standard restrictions of the "slice-and-dice" operator in OLAP systems. *Measure_list* consists of one or several measures used for sorting recommendations, as shown in Query 7. If more than one measure is used for this purpose, then the ordering is lexicographic. Also, aggregation functions can be applied to measures, as is shown in Queries 5 and 8. Finally, *measure_restrictions* constitute standard restrictions on different types of ratings.

The output of a recommendation is a matrix of the "TO" dimensions with the entries consisting of the lists of the records corresponding to the "RECOMMEND" dimensions. These lists of records and their measures are determined using the rating estimation methods, described in Section 2 (and in [3]), and the rating qualifiers, described in Query 7. Finally, these lists of records are sorted by the measures in the BASED ON clause and truncated according to the SHOW clause. For example, the output of Query 4 is a two-dimensional matrix of User and Companion dimensions with the restrictions that the User is Tom and the companion is his girlfriend, which results in a singleton matrix (Tom, girlfriend). The entry in this matrix is the list of movies and the best times to see these movies sorted by the personal ratings and restricted to the top 3 entries.

The operational semantics of RQL queries is defined as follows. First, the WHERE and WITH clauses are applied to restrict the rating cube only to the ratings and

[2] To simplify the presentation, we assume some default value for the ratings qualifier in the remainder of the paper and omit the use of the EST flag from the subsequent query processing examples.

[3] This situation is similar to SQL when aggregation is used in the SELECT and the GROUP BY clauses.

values specified in these clauses. Then, the aggregation (based on the AGGR BY clause) is performed to produce aggregated ratings. Then, only the aggregate ratings that satisfy the HAVING clause are selected. Finally, the recommendation matrix is generated according to the RECOMMEND and SHOW clauses, where entries are sorted based on the "BASED ON" clause.

Although we used the term RQL queries throughout the paper, recommendations are really not queries according to the standard meaning of the term since they return a very idiosyncratic output in the form of the recommendation matrix described above. To address this issue, we distinguish between the *core RQL query* containing FROM, BASED ON, WHERE, WITH, AGGR BY and HAVING clauses, and the recommendation *wrapper* containing RECOMMEND and SHOW clauses. The core RQL query operates on a multidimensional cube of ratings *and* always returns an object of the same type – a multidimensional 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. In processing RQL queries, first, the core query is evaluated, and then the wrapper (RECOMMEND and SHOW clauses) is applied to the output of the core RQL query.

Although RQL is related to OLAP query languages, it has certain distinctive characteristics pertaining to recommendations that make it different from other languages. In particular, as explained above, RQL queries are divided into the "core" and "wrapper" components, each component requiring separate evaluation methods. Also, ratings can be actual, i.e., specified by the user, or inferred, i.e., derived from the actual ratings using various estimation methods. Therefore, RQL provides mechanisms for distinguishing between different types of ratings, as Query 7 demonstrates. Finally, the language provides various other recommendation-specific properties, such as using a single cube of ratings, the WITH clause, and recommendation-specific types of aggregations, which make RQL uniquely suited for recommendation purposes.

As mentioned earlier, the text-based form of RQL will primarily be used by the power users and, therefore, it is important to have a graphical UI of RQL for the naïve users. An example of a form-based version of Query 3 is provided in Figure 2. As we can see from this example, all the syntax of RQL Query 3, including multiple dimensions, what to recommend for whom, which ratings to use, and what the selection criteria are, is specified in a graphical form understandable for a naïve user.

Note that, depending on the needs of a specific recommendation application, user interface forms could be designed to be simpler or more advanced, using the same general RQL for their implementation. Therefore, we would like to stress that the semantics of the UI should be based on the multidimensional recommendation model and on RQL in particular. This allows to have a tight coupling between the UI and RQL primitives (e.g., both describe which dimensions to recommend and to whom, which ratings to use and how, and so on) and, therefore, to use RQL as a guide for designing the form-based UI. Alternatively, one could try mapping the UI-based requests directly into SQL. However, this would create discontinuity between the front-end UI model and the back-end SQL-based recommendation model, thus making the semantics of the UI hard to understand and causing interpretation errors and problems. Since RQL is needed for the power-users anyway, it is much better to define the semantics of the UI in terms of RQL rather than SQL.

In the next section, we present a recommendation algebra that is needed for processing RQL queries. Since algebraic operators should return objects of the same type as their inputs, we will target the recommendation algebra only to the core RQL queries. To process a complete RQL query, we first construct an algebraic expression equivalent to the core RQL query, evaluate it, and then "feed" the results of the evaluation into the RQL wrapper to produce the final output.

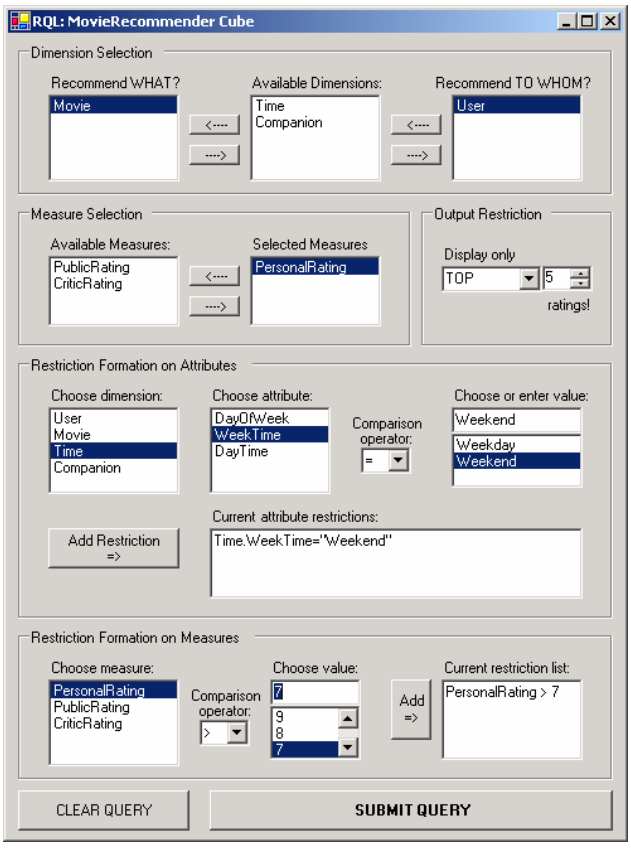

Figure 2. Form-based example of Query 3.

4. Recommendation Algebra (RA)

Since multidimensional recommendations are based on the OLAP paradigm, we use the OLAP algebras introduced in the database community [1, 6, 15, 9, 10] to define the recommendation algebra. However, since the RQL language is tailored specifically for the domain of recommendations, only a subset of the standard OLAP operators is needed to be able to process RQL queries. For example, we do not use a *join* operator in the recommendation algebra because RQL deals with only one recommendation cube for the reasons explained in Section 3. Similarly, we do not use PUSH and PULL operators of the OLAP algebras because they do not occur naturally in the recommendation contexts. In fact, they would "destroy" the cube of ratings by pushing other dimensions into measures and mixing them with the natural recommendation measures, i.e., ratings.

In the rest of this section, we describe the recommendation algebra RA. We will follow the definitions of the OLAP operators introduced in [15]. However, before introducing the OLAP operators, we formally define a *data cube* as a 5-tuple $<D, M, A, f, L>$, where these five components are defined as:

- $D = \{d_1, d_2, \ldots, d_n\}$ is a set of n dimensions, where each d_i is a dimension name representing some domain $dom(d_i)$.
- $M = \{m_1, m_2, \ldots, m_k\}$ represents a set of measures, where each m_i is a different type of a rating representing some rating domain $dom(m_i)$.
- $A = \{a_1, a_2, \ldots, a_t\}$ is a set of attributes where each a_i is an attribute name.
- f: $D \rightarrow 2^A$ is a mapping that identifies a set of attributes for each dimension. In other words, f plays the role of *schema* – it partitions the set of all attributes A among all the dimensions. Mapping f is such that $f(d_i) \cap f(d_j) = \varnothing$ and $\cup_i f(d_i) = A$.
- $L = \{l_1, l_2, \ldots, l_j\}$ denotes a set of cube cells, where each cube cell l_i is represented by its address and its content. That is, $l_i = <addr_i, cont_i>$, where $addr_i \in dom(d_1) \times dom(d_2) \times \ldots \times dom(d_n)$ and $cont_i \in dom(m_1) \times dom(m_2) \times \ldots \times dom(m_k)$.

Given this definition, we can introduce the following algebraic operators on the cube.

Restriction (RSTR). The restriction operator defines the "slice and dice" operation [5, 8] on the cube by putting restrictions on one dimension or multiple dimensions, or on the attributes of dimensions. Similarly, restrictions could be put on any of the measures m_i. The restrictions are represented by compound predicate

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

where $<op>$ represents a logical operator AND or OR and p_i is a domain restriction for a single attribute, e.g., "Age>21". Then, the restriction operator maps inputs $C_I=<D,M,A,f,L>$ into outputs $C_O=<D,M,A,f,L_o>$ using compound predicate P and is denoted as $RSTR_P(C_I)=C_O$. In this case, L_o is calculated simply by retaining those elements of L that satisfy predicate P, i.e., as $L_o = \{l \in L \mid P(l) \text{ is true}\}$.

Metric Projection (MRPJ). The metric projection operator restricts the output of a cube to include only a subset of the original set of measures. It is defined as $MRPJ_{mi}(C_I)=C_O$, where $C_I=<D,M,A,f,L>$, $C_O = <D,M_o,A,f,L_o>$, $M_o=M-\{m_i\}$, and L_o is obtained from L by taking each element $l_j = <addr_j, cont_j>$ and removing the i^{th} measure component from the $cont_j$ vector. In other words, this operator projects the measure $m_i \in M$ out of the cube.

Destroy Dimension (DTDM). The destroy dimension operator is denoted as $DTDM_{di}(C_I)=C_O$, where $C_I=<D,M,A,f,L>$, $C_O = <D_o,M,A_o,f,L_o>$. Here d_i is the dimension to be destroyed, and the outputs are calculated as follows: $D_o = D - d_i$, $A_o = A - f(d_i)$, and L_o is obtained from L by by taking each element $l_j = <addr_j, cont_j>$ and removing the i^{th} dimension component from the $addr_j$ vector. Note that, after destroying a dimension it is possible to have a number of updated l_j elements that have identical $addr_j$ components. Therefore, all such elements are aggregated into a single element $<addr_j, F_{aggr}.cont_j> \in L_o$ by aggregating each individual measure component of the $cont_j$ vector using some default measure aggregation function F_{aggr}.

Aggregation (AGGR). The aggregation operator performs aggregation on one or more dimensions (e.g., aggregating a movie dimension based on movie genres). This operator applies one of the aggregation functions F_{aggr}, such as SUM, AVG, MAX, or MIN, to the cube with one or more dimensions specified as grouping attributes. Formally, $AGGR_{di}(C_I)=C_O$, where $C_I=<D,M,A,f,L>$, $C_O = <D_o,M,A_o,f_o,L_o>$. Here d_i is the dimension being aggregated, which after aggregation becomes a modified dimension d_i^* together with its own aggregated attributes and mapping function $f_o(d_i^*)$. The outputs are calculated as follows: $D_o = D - d_i + d_i^*$, f_o is the new mapping function that incorporates the newly aggregated dimension, $A_o = A - f(d_i) + f_o(d_i^*)$, and L_o is obtained from L by by taking each element $l_j = <addr_j, cont_j>$, replacing the i^{th} dimension component of the $addr_j$ vector by its aggregate (e.g., replacing individual movies by more general movie genres). Note that, as with DTDM operator, after aggregating a dimension it is possible to have a number of updated l_j elements that have identical $addr_j$ components. Therefore, all such elements

are aggregated into a single element $<addr_j, F_{aggr}.cont_j> \in L_o$ by aggregating each individual $cont_j$ component using some measure aggregation function F_{aggr}.

The recommendation algebra RA is formed by the composition of these four operators. To illustrate how this algebra works, consider the following examples of some of the RQL queries expressed in Section 3.

The next example illustrates how a simple RQL query can be expressed in the recommendation algebra.

Example 2. This example presents the algebraic expression of Query 3 (i.e., recommend top 5 movies to the user to see over the weekend, but only if the personal ratings of the movies are higher than 7).

$$\text{DTDM}_{(\text{MovieTheater, Time, Companion})}(\text{MRPJ}_{(\text{PublicRating})}(\text{RSTR}_{(\text{PersonalRating} > 7)}(\text{RSTR}_{(\text{Time.WeekTime="Weekend"})}(\text{MovieRecommender}))))) \quad \blacksquare$$

As explained before, this algebraic expression specifies only the *core* part of the RQL query. The actual recommendation results are generated by the RQL *wrapper* from the core query. Therefore, this algebraic expression destroys all other dimensions at 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 measure and MovieTheater, Time, and Companion dimensions are removed from the MovieRecommender cube.

The next example shows how aggregation is done in the recommendation algebra.

Example 3. This example presents the algebraic expression of Query 5 (i.e., recommend movie genre to different professions using those movies with Personal ratings bigger than 6).

$$\text{DTDM}_{(\text{Theater, Time, Companion})}(\text{MRPJ}_{(\text{PublicRating})}(\text{AGGR}_{(\text{User.Profession})}(\text{AGGR}_{(\text{Movie.Genre})}(\text{RSTR}_{(\text{PersonalRating} > 6)}(\text{MovieRecommender}))))) \quad \blacksquare$$

Next, we present a more complex example of an RQL query that has aggregation and a HAVING clause selection.

Example 4. This example presents the algebraic expression of Query 8 (i.e., recommend movie genre to different professions and show those results with average ratings bigger than 6).

$$\text{DTDM}_{(\text{Theater, Time, Companion})}(\text{MRPJ}_{(\text{PublicRating})}(\text{RSTR}_{(\text{PersonalRating}>6)}(\text{AGGR}_{(\text{User.Profession})}(\text{AGGR}_{(\text{Movie.Genre})}(\text{MovieRecommender}))))) \quad \blacksquare$$

The resulting recommendation algebra is more expressive than RQL. As will be shown in Section 5, any core RQL query can be mapped into the algebra. However, for example, the following algebraic expression cannot be expressed in RQL:

$$\text{RSTR}_{(\text{PersonalRating}>7)}(\text{AGGR}_{(\text{Movie.Genre})}(\text{RSTR}_{(\text{PersonalRating} > 6)}(\text{AGGR}_{(\text{User.Profession})}(\text{MovieRecommender}))))$$

This is the case because this algebraic expression applies restriction (selection) in two cases: $\text{RSTR}_{(\text{PersonalRating} > 6)}$ for the partially aggregated case and subsequently $\text{RSTR}_{(\text{PersonalRating} > 7)}$ for the fully aggregated case. In contrast to this, RQL can do aggregation and the subsequent restriction (selection) only once. Therefore, restriction on partially aggregated ratings, as was done in the above algebraic expression, cannot be expressed in RQL.

One way to deal with this problem is to extend RQL in such a way that it would be able to express any algebraic query. However, we decided against this, because, in our experience, algebraic queries not expressible in RQL are of little practical importance in recommender systems. Therefore, the support for these queries would only add extra complexity to the language without providing any significant results.

5. Processing RQL Queries

As mentioned earlier, we chose to implement RQL queries by mapping them into SQL. This mapping is decomposed into the following three stages. First, we translate RQL queries into the recommendation algebra RA. Second, we translate RA expressions into the relational algebra. Third, we translate the relational algebra queries into SQL. We describe these steps in the rest of this section.

5.1. Translation from RQL to RA

The mapping from the RQL *core* query to the recommendation algebra is done by parsing the RQL query and generating corresponding algebraic operators. The MAP algorithm, presented in Figure 3, shows how to translate an arbitrary RQL query (as defined in Figure 1) with its specific parameters, such as *measure_restrictions*,

dimension_restrictions, etc., into an algebraic expression in RA.

Based on the input query *RQL_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. First, we check whether *RQL_Query* has any restrictions on dimensions (Line 2) or measures (Line 4), and if so, we then generate a restriction operator ***RSTR*** with corresponding parameters (Lines 3 and 5). Note, that we build the resulting algebraic expression *RA_op* by composing the current value of *RA_op* with each newly generated operator. 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.

MAP(*RQL_Query*) **{**

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

(2) if (∃ WHERE clause in *RQL_Query*) then
(3) *RA_op* := *RA_op* ⊕ $\boldsymbol{RSTR}_{(dimension\_restrictions)}$;

(4) if (∃ WITH clause in *RQL_Query*) then
(5) *RA_op* := *RA_op* ⊕ $\boldsymbol{RSTR}_{(measure\_restrictions)}$;

(6) if (∃ AGGR BY clause in *RQL_Query*) then
(7) *RA_op* := *RA_op* ⊕ $\boldsymbol{AGGR}_{(aggregation\_dim\_attr\_list)}$;

(8) if (∃ HAVING clause in *RQL_Query*)
(9) *RA_op* := *RA_op* ⊕ $\boldsymbol{RSTR}_{(aggregation\_restriction)}$;

(10) *RA_op* := *RA_op* ⊕ $\boldsymbol{MRPJ}_{(M-measure\_list)}$;

(11) *final_dim_list* := f^{-1} (*recommend_dim_attr_list*) +
(12) f^{-1} (*recipient_dim_attr_list*);
(13) *RA_op* := *RA_op* ⊕ $\boldsymbol{DTDM}_{(D-final\_dim\_list)}$;

(14) Return *RA_op*;
}

Figure 3. Mapping RQL queries into RA expressions.

Line 6 checks whether *RQL_Query* requests any aggregation, and Line 7 generates the corresponding algebraic operator. Similarly, Lines 8 and 9 handle the restrictions on aggregated dimensions, if any. Since the recommendation may be based only on some of the rating measures, Line 10 generates a ***MRPJ*** operator that projects the unused measures out. Next, we remove the dimensions that are not present in *recommend_dim_attr_list* and *recipient_dim_attr_list*, i.e., we only retain the dimensions that are present in either RECOMMEND or TO clauses of *RQL_Query*. For notational simplicity, we define f^{-1} as the function that returns a corresponding dimension for each attribute. (It is not a true inverse of function *f* as defined in Section 4, but it is very easy to construct f^{-1} if mapping *f* is known.) Then, Lines 11 and 12 compute the list of relevant dimensions, and Line 13 generates a *DTDM* operator that removes all other (irrelevant) dimensions. Finally, Line 14 returns the resulting algebraic expression.

Note that MAP produces three selection operators *RSTR* corresponding to the *compound* selection statements in the WHERE, WITH, and HAVING clauses respectively. The next step is to decompose them into *atomic* selections having only atomic restriction conditions using standard algebraic methods used in query optimization [11, 4]. Example in Section 5.4 will illustrate this point further.

5.2. Translation from RA to Relational Algebra

In this section we map recommendation algebraic operators introduced in Section 4 into relational algebra. First, we assume that the multidimensional cube of ratings is mapped into its snowflake schema relational representation using the standard mapping [5, 8] as follows. The fact table *Rating* contains tuples that represent the cells of the multidimensional cube, where each cell is represented by some combination of the identifiers (keys) for all dimensions as well as the combination of values for all the metrics. In other words, the fact table essentially represents set *L*, as defined in Section 4. For each dimension d_i, there is a set of tables $\{D_{i1}, \ldots, D_{im}\}$ forming the *snowflake hierarchy* [8] based on the foreign key relationships between "neighbouring" tables. Finally, table D_{i1} represents the "main" table of dimension d_i and can be joined with table *Rating* on the foreign key.

To illustrate these concepts, consider the MovieRecommender cube presented in Example 1 and its mapping into the following snowflake schema presented in Figure 4. The *User* dimension in the MovieRecommender cube is represented by *User* and *Profession* tables in the snowflake schema in Figure 4 that can be joined together on ProfessionID and to *Rating* table on UserID. Given this cube-to-snowflake schema mapping, RA algebraic operators are mapped into the relational algebra operators as follows.

Restriction (RSTR). If the restrictions are on measures, we do selection on the fact table:

Rating = $Selection_{Pm}$(Rating)

If the restrictions are on dimensions, we need to join the corresponding dimension table with fact table and then apply the selection (restriction) conditions on the fact table and dimension table respectively. Note that the restrictions can be applicable to the *n*-th level table D_{in} in the snowflake hierarchy for dimension d_i. These restrictions should be propagated from table D_{in} all the

way down the snowflake hierarchy to table D_{i1} and into the Rating table by joining all these tables. Let P_{dn} be the restriction condition applicable to table D_{in}. Then operator *RSTR* can be implemented as

$$D_{in} = Selection_{Pdn}(D_{in})$$
$$D_{i(j-1)} = (Projection_{Xn}(Join_{D_{i(j-1)}.ID=D_{in}.ID}(D_{i(j-1)}, D_{ij}))), j = 2,...,n$$
$$Rating = (Projection_{Xr}(Join_{D_{i1}.ID=Rating.D_iID}(D_{i1}, Rating)))$$

Note that the first expression updates table D_{in} as described above, and the second expression updates all the intermediate tables $D_{i(n-1)}$, …, D_{i1}. Finally, Rating table is updated by joining itself with D_{i1}, as shown in the third expression. The projection operator is needed to retain the original attributes *Xn* of the tables after they have been joined with other tables.

Rating (ID, MovieID, UserID, LocationID, TimeID, CompanionID, PublicRating, PersonalRating)

User (UserID, ProfessionID, UserName, Age, Gender)
Profession (ProfessionID, ProfessionName)

Movie (MovieID, GenreID, MovieName, Length, ReleaseYear)
Genre(GenreID, GenreName)

Theater (TheaterID, CityID, ThearterName, Capacity)
City(CityID, CityName, RegionID)
Region (RegionID, RegionName)

Time (TimeStampID, DayofTheWeekID)
DoW(DayofTheWeekID, Weekend, DayOfTheWeek)

Companion (CompanionID, CompanionType)

Figure 4. Snowflake schema for the rating cube from Example 1.

Metric Projection (MRPJ). To get a subset of measures stored in the fact table, we only need to keep remaining measures by applying the Projection operator. The projected-out measures are denoted as M_p

$$\text{Rating} = Projection_{(M-Mp)}(\text{Rating})$$

Destroy Dimension (DTDM). In order to destroy dimension d_i, we first need to drop all the tables {D_{i1}, …, D_{im}} forming the snowflake hierarchy for dimension d_i. Second, we need to delete corresponding dimension ID attribute from the Rating table. After projecting out the dimension, several tuples in the Rating table can have the same values for all the remaining foreign keys. All these similar tuples need to be merged and their individual ratings aggregated according to the F_{aggr} function specified for each measure.[4] We follow [4] and use their relational aggregation operator *FN* as a part of the relational algebra to express this aggregation operation. Formally:

$$Rating = Projection_{(D-D_i.ID)}(FN_{(M,F_{aggr},D-D_i)}(Rating))$$

Aggregation (AGGR). Aggregation needs to be specified at a certain level *n* of the snowflake hierarchy for dimension d_i. More specifically, it should be applied to table *Rating* and all the tables D_{i1}, …, D_{in} defined earlier in Section 5.2, by joining all of these tables and then grouping the ratings based on the attribute $D_{in}.ID$. Formally, this can be expressed as:

$$D_{i(j-1)} = Join_{D_{i(j-1)}.ID=D_{ij}.ID}(D_{i(j-1)}, D_{ij}), j = 2,...,n$$
$$Rating = FN_{(M,F_{aggr},D_{in}.ID)}(Join_{D_{i1}.ID=Rating.D_iID}(D_{i1}, Rating))$$
$$D_{i1} = D_{in}$$

After the aggregation, D_{in} becomes the basic table for the dimension d_i (new D_{i1}).

5.3. Translation from Relational Algebra to SQL

Once we map the RA algebraic expressions into the relational algebra, the translation from the relational algebra into SQL constitutes an old and a well-studied problem [11]. Therefore, we omit its coverage and only present a translation example in Section 5.4.

5.4. Example: Mapping RQL Queries into SQL

To demonstrate the mapping from RQL to SQL, consider the following RQL query recommending top 5 action movies to the female users living in New York:

```
RECOMMEND  Movie TO User
FROM       MovieRecommender
BASED ON   PersonalRating
WHERE      Movie.Genre = "Action" AND
               Theater.City = "New York"
               AND User.Gender = "Female"
SHOW TOP 5
```

First, algorithm MAP translates the core of this RQL query into the following RA expression:

```
DTDM (Theater, Time, Company) (
    MRPJ (PublicRating) (
        RSTR (Movie.Genre = "Action"AND
            Theater.City = "New York"AND User.Gender = "Female")
            (MovieRecommender)))
```

Then the compound selection operator *RSTR* can be decomposed into atomic selections using standard methods, resulting in the following RA expression:

[4] In the subsequent query processing examples we use the AVG function for this purpose.

```
DTDM (Theater, Time, Company) (
    MRPJ (PublicRating) (
        RSTR (Movie.Genre = "Action") (
            RSTR(Theater.City = "New York") (
                RSTR (User.Gender = "Female") (
                    MovieRecommender ) ) ) ) )
```

When translating this RA expression into the relational algebra, we assume that the MovieRecommender ratings cube from Example 1 is mapped into the snowflake schema presented in Figure 4. Then the above RA expression is converted into the following sequence of relational algebra operators[5]:

```
User1= Selection (User.Gender='Female') (User)

Theater1= Join (Theater.TheaterID=City.TheaterID)(
            Selection (City.CityName='New York') (City),
            Theater)

Movie1= Join (Genre.GenreID=Movie.GenreID)(
            Selection (Genre.GenreName='Action') (Genre),
            Movie)

Rating1=Join (Moive1.MovieID=Rating.MovieID)(
            Movie1,
            Join (Theater1.TheaterID=Rating.TheaterID)(
                Theater1,
                Join (User1.UserID=Rating.UserID)(
                    User1,
                    Rating ) ) )

Rating2=
    Projection (PersonalRating) (Rating1)

Rating3=
    FN(PersonalRating, AVG, {Theater, Time, Company}) (Rating2)

Rating4=
    Projection (Movie.Name, User.Name, User.ID) (Rating3)
```

Note that the operator *FN* aggregates ratings in the cube when Theater, Time, and Companion dimensions are projected out.

Finally, this relational algebra expression is mapped into SQL by simply simulating in SQL each of the six relational algebra operators from that expression in the appropriate order and substituting all of them into the following *compound* SQL statement:

```
SELECT R4.MovieID, R4.UserId, R4.PersonalRatings
FROM
    (SELECT *
     FROM
        (SELECT *
        FROM Users
        WHERE Users.Gender="female"
        ) as U1,
        (SELECT *
        FROM
            (SELECT *
            FROM Theater,
                (SELECT *
                FROM City
                WHERE City.CityName="New York"
                ) as A
            WHERE Theater.CityID = A.CityID
            ) as T1,
            (SELECT *
            FROM Rating,
                (SELECT *
                FROM Rating,
                    (SELECT *
                    FROM Movie,
                        (SELECT *
                        FROM Genre
                        WHERE Genre.GenreName
                                    ="Action"
                        ) as G
                    WHERE Movie.GenreId =
                                    G.GenreId
                    ) as M1
                WHERE Rating.MovieID =
                                    M1.MovieID
                ) as R1
            WHERE Rating.RatingId=R1.RatingId
            ) as R2
        WHERE T1.TheaterId=R2.TheaterId
        ) as R3
    WHERE U1.UserId =R3.UserId
    ) as R4
GROUP BY R4.MovieID, R4.UserID
```

As this example demonstrates, even simple RQL queries can produce highly complex SQL expressions as a result of the translation method presented in this section. Therefore, query optimization methods need to be developed to simplify such mapping. This query optimization can be performed in the following two places during the translation process:

- *Recommendation algebra:* replace the "naïve" algebraic expression generated by algorithm MAP in Section 5.1 with an equivalent optimized expression.
- *Relational algebra:* replace the "naïve" relational algebra expression generated in Section 5.2 with an

[5] Note that we should have updated Rating table after each of the three selections when we computed User1, Theater1 and Movie1 relations. However, this would have complicated the subsequent generation of the SQL statement very significantly, and we have decided to optimize the translation a little bit, for the purpose of improving readability.

equivalent optimized relational algebra expression using standard query optimization methods [11].

6. Conclusions

In this paper we introduced the language RQL for querying multidimensional recommender system. RQL empowers the end-users by letting them formulate recommendations of interest in a flexible and user-friendly manner. RQL queries are formulated on multidimensional cubes of ratings, support OLAP-based aggregation capabilities, and are expressed in a SQL-like language incorporating idiosyncrasies of recommender systems. Since recommendations are not classical queries, RQL consists of the core query that operates on multidimensional cubes of ratings and returns rating cubes as outputs, and the recommendation wrapper that takes these cubes and converts them into the recommendation outputs. Also, the recommendation cubes are only partially filled with actual ratings, and require the missing ratings to be estimated using various estimation methods. Therefore, RQL provides the flexibility by letting the end-user ask questions about actual, estimated, and overall ratings.

We also addressed the RQL query execution problem by mapping RQL queries into the recommendation algebra RA and then into relational algebra and SQL. However, this mapping can be inefficient in some cases, and, therefore, query optimization methods need to be developed at the RA and relational algebra levels. We plan to study this problem in the future.

Although our mapping procedure follows the ROLAP approach, RQL is a general query language and can be implemented using other methods, including the MOLAP approach, where RQL queries are evaluated directly on the multidimensional cubes of ratings. One query evaluation problem pertaining to recommender systems deals with the determination of which new ratings need to be evaluated in order to answer a particular RQL query. 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 in the recommendation cube described in Example 1. Since rating estimation 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 also plan to study this problem in the future.

References

[1] Agrawal R., Gupta A., Sarawagi S., Modeling multidimensional databases. *ICDE*, p. 232-243, 1997.

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

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

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

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

[6] Gyssens, M and L.V.S. Lakshmanan, A foundation for multi-dimensional databases. *VLDB*, 106--115, 1997.

[7] Hill, W, L. Stead, M. Rosenstein, and G. Furnas. Recommending and evaluating choices in a virtual community of use. *CHI*, 1995.

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

[9] C. Li, X. Sean Wang. A Data Model for Supporting On-Line Analytical Processing. *CIKM*, 1996.

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

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

[12] Resnick, P., N. Iakovou, M. Sushak, P. Bergstrom, and J. Riedl. GroupLens: An open architecture for collaborative filtering of netnews. In *Proc. of the Computer Supported Cooperative Work Conf.*, 1994.

[13] 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.

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

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

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