FlexRecs: Expressing and Combining Flexible Recommendations

Georgia Koutrika, Benjamin Bercovitz, Hector Garcia-Molina
Computer Science Department, Stanford University, Stanford, California, USA
{koutrika, berco, hector}@cs.stanford.edu

ABSTRACT

Recommendation systems have become very popular but most recommendation methods are 'hard-wired' into the system making experimentation with and implementation of new recommendation paradigms cumbersome. In this paper, we propose *FlexRecs*, a framework that decouples the definition of a recommendation process from its execution and supports flexible recommendations over structured data. In FlexRecs, a recommendation approach can be defined declaratively as a high-level parameterized workflow comprising traditional relational operators and new operators that generate or combine recommendations. We describe a prototype flexible recommendation engine that realizes the proposed framework and we present example workflows and experimental results that show its potential for capturing multiple, existing or novel, recommendations easily and having a flexible recommendation system that combines extensibility with reasonable performance.

Categories and Subject Descriptors

H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval—*Search Process*

General Terms

Algorithms, Languages, Performance

Keywords

flexible recommendations, recommendation operators, recommendation queries

1. INTRODUCTION

Recommendation systems provide advice on movies, products, travel, leisure activities, and many other topics, and have become very popular in systems, such as Google News [10], Amazon [17] and MovieLens [19]. Since the appearance of the first recommendation systems [12, 22, 26], many recommendation approaches have been proposed both by the industry and academia. However, most recommendation systems have a number of limitations:

- *Hard Wired*: The recommendation algorithm is typically embedded in the system code, not expressed declaratively. From the designer viewpoint, this fact makes it hard to modify the algorithm, or to experiment with different approaches.
- *No Flexibility:* The recommendations provided are fixed. End users are given few choices. For example, a user may be unable to request recommendations for movies that could be jointly seen *by her and her friend*, or that her recommendations be based on what people *in her age group* are watching. Users may expect diverse recommendations in different contexts.
- *Limited world model*: Recommendation approaches deal with two types of entities, users and items (e.g., movies), represented as sets of ratings or features. Providing recommendations using richer data representations is not straightforward. For example, a user may want recommendations for *courses* from users *with similar grades and similar ratings*.

In this paper, we propose *FlexRecs*, a framework that allows flexible recommendations to be easily defined, customized, and processed over structured data. FlexRecs (*a*) *decouples* the definition of a recommendation process from execution, (*b*) *declaratively defines* a recommendation process as a high-level workflow and (*c*) *enables generating* any recommendations with *the same engine*.

A given recommendation approach can be expressed as a high-level workflow, which may contain *traditional relational operators* such as select, project and join, *plus new recommendation operators* that generate or combine recommendations. A workflow can handle data (including recommendations) in *relational* form.

A designer can easily create *multiple, customizable workflows* for content-based, collaborative, as well as novel recommendation paradigms. The end user can select from them, depending on her information needs. This selection is done through a GUI, which also allows the user to enter *parameters* for workflows in order to get more accurate and personalized recommendations. For instance, the user may specify that her recommendations be based on what people in her age group are watching. This choice gets translated into a select condition, which is passed to the appropriate workflow. This functionality is essentially similar to advanced searches: a designer builds a set of parameterized SQL queries. End users can execute these queries choosing parameter values to receive different results through an advanced search interface.

We describe a prototype flexible recommendation engine that realizes the proposed framework. The system allows executing a workflow over a conventional DBMS by "compiling" it into a sequence of SQL calls. The recommendation operators may call upon functions in a library that implement common tasks for generating recommendations, such as computing the Jaccard or Pearson similarity of two sets of objects, or perform more fancy statistical predictions. When possible, library functions are compiled into the

SIGMOD'09, June 29–July 2, 2009, Providence, Rhode Island, USA.

Departments(DepID, DepCode, Name)
Courses(CourseID, DepID, Title, Description, Units, Url)
CourseSched(CourseID, Year, Term, InstrID, Location, TimeSlot, Days)
Instructors(InstrID, Name, Url)
Students(SuID, Name, Class, GPA, Status)
StudentStudies(SuID, StudyPrgID)
StudyPrograms(StudyPrgID, ProgramName, Classification, DepID)
StudentHistory(SuID, CourseID, Year, Term, Grade, Rating)
Comments(SuID, CourseID, Year, Term, Text, Rating, Date)

Figure 1: Extract from the course database

SQL statements; in other cases we can rely on external functions that are called by the SQL statements.

1.1 Motivation

Our work on FlexRecs was motivated by, and has been implemented as part of CourseRank, a social tool we have developed in our lab. CourseRank helps students in our university to make informed choices about classes and take advantage of the available learning options. It displays official university information and statistics, such as bulletin course descriptions, grade distributions, and results of official course evaluations. Students can anonymously rank courses they have taken, add comments, and rank the accuracy of each others' comments. They can also get recommendations, organize their classes into a quarterly schedule or devise a four year plan and track their progress. CourseRank also functions as a feedback tool for faculty and administrators, ensuring that information is as accurate as possible. To support this functionality, we maintain a database that stores rich information about the courses offered, the instructors, the students, the textbooks, and so forth. Figure 1 provides a small snapshot of the database schema. The system is already used by more than 9,000 students inside the university, out of a total of about 14,000 students. The vast majority of users are undergraduates, and there are only about 7,000 undergraduates in our university.

• *The need for flexibility and expressivity*. Although the initial version of CourseRank has been very popular [2], we received many requests, from students and administrators, for more flexible recommendations: As most commercial recommendation systems, our initial version offered no choices, just a list of recommended courses for the student to consider, as shown in Figure 2. The figure displays a general list of collaborative recommendations for a particular student containing a Robotics course and a Spanish language course. If the student is interested in Spanish courses, she may prefer to see more Spanish courses. If she is interested in French courses, she may not want to see any of our recommendations[1]. Students want to specify the type of course they are interested in (e.g., a biology class, something that satisfies the university's writing requirement). They also want to request recommendations based on a peer group they select (e.g., students in their major, or freshmen only) and based on different criteria. For example, a student may want recommendations for CS courses from CS students with similar grades (i.e., with similar performance) and for dance classes from students with similar ratings (i.e., with similar tastes). In some cases, students want to see the recommendations the system would give their best friend, not themselves.

• *The need for experimentation and higher productivity*. As recommendations comprise an integral part of the system, we want to experiment with different recommendation approaches: Would approach X be more effective than approach Y in our environment? What are the right weights for blending two recommendations (e.g., one based on what students like, and another based on what courses are more critical for completing a major)? What is the best way to predict, not good courses, but likely grades a student will obtain in future courses? Implementing many different recommendation approaches and their variants from scratch, possibly as different system modules, can be time-consuming and counter-productive and does not lend to high code reusability. It also leads to a recommendation system that is not easily expandable and manageable and it makes experimenting with different recommendation possibilities cumbersome and slow.

[1]Similar stories are known from other domains [1, 5]. For instance, an Amazon customer that once bought a book for his 9-year-old will be considered a kids' books fan or be mistakenly considered similar to other kids' books buyers and is bound to see irrelevant recommendations in his/her future transactions with the system.

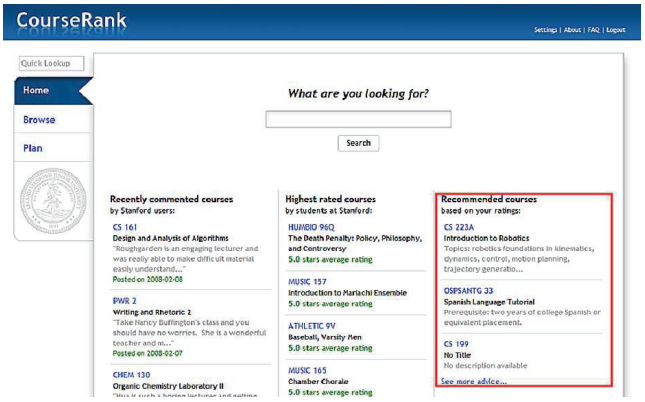


Figure 2: Fixed recommendations in CourseRank

Faced with the above challenges, we need a way to declaratively describe recommendations. These descriptions would also need to handle parameters, so that end-users could at run time personalize their recommendations, e.g., by choosing a selection condition, or by weighting recommendations that are blended. The resulting FlexRecs framework has indeed made it possible to (*a*) easily capture multiple recommendation paradigms and allow users dynamically personalize them to fit their needs, (*b*) experiment with novel recommendation strategies, (*c*) design a flexible and extensible system and increase developer productivity: instead of adding new modules, a developer needs to define a new recommendation workflow or expand the library of reusable functions if a new comparison or prediction model is needed. The new version of CourseRank employs the flexible recommendation engine to support different features and to let end-users tailor their recommendations. We have also designed a user interface for flexible recommendations in CourseRank [16].

1.2 Contributions

In summary, the contributions of our work are as follows:

- We propose *decoupling the definition* of a recommendation process from its execution and present *FlexRecs*, a framework for defining flexible recommendations over structured data. In FlexRecs, a recommendation approach can be defined declaratively as a high-level parameterized workflow comprising traditional relational operators and new recommendation operators.
- We introduce an *extend* operator that generates a virtual nested relation. For example, a tuple in an extended student relation may contain an attribute that gives all the courses taken by that student. Although there is nothing new in the concept of a nested relation, this particular flavor of nested relation is what makes the whole framework work: extended relations simplify the definition of our other operators.

- We define *recommend* and *blend* operators that capture the essence of most recommendation workflows. The operators can be composed and combined with more traditional select, project, and join operators. The main challenge in designing these operators is in capturing the appropriate functionality and generality. Without a lot of care, one can easily come up with operators that do not compose, are not useful for common recommendations or are unnecessarily complex.
- We provide several examples that illustrate how common recommendation strategies (found in literature or implemented in CourseRank) can be expressed as parameterized workflows with our operators, as well as new recommendation types.
- We describe a prototype *flexible recommendation engine* that realizes the proposed framework and executes multiple workflows transparently. Our strategy reduces the amount of data saved in temporary relations during execution. For instance, extended relations are never materialized. Furthermore, our new operators can be compiled into standard SQL for execution. Hence, recommendation workflows are converted into sequences of standard SQL calls, which are then executed by a conventional database system taking advantage of the underlying query optimization. The recommendation operators may call upon functions in a library. When possible, library functions are also compiled into the SQL statements.
- We present experimental results that show the potential of our approach for capturing multiple, existing or novel, recommendations easily and having a flexible recommendation system that combines extensibility with reasonable performance. For our evaluation, we use real data from our production system.

2. RELATED WORK

Since the appearance of the first papers on collaborative filtering [12, 22, 26], a lot of work has been done from improving and evaluating recommendation methods [3, 13, 27] to designing trustworthy systems [18, 20]. In their core, these systems provide (*a*) *content-based recommendations* for items similar to those the user preferred in the past [7, 21] or (*b*) *collaborative recommendations* for items that people with similar tastes liked [8, 22] or (*c*) *hybrid recommendations* by combining recommendation methods [6, 24]. Although a popular and highly researched area, recommendation systems present important limitations for users as well as researchers and developers [5].

The recommendation algorithm as well as its inputs are hard wired in the system code. Designing, implementing and experimenting with new methods can be time-consuming and counterproductive [15]. Furthermore, hard-wired algorithms generate only a predefined and fixed set of recommendations, which cannot capture the real-time user information needs. For example, in content-based methods, the user is limited to see items that are similar to those she liked in the past. On the other hand, collaborative filtering methods provide serendipitous recommendations but they have problems, such as the inability to recommend new items [6]. There have been many approaches that try to solve some of these problems, such as filtering out items if they are too similar to those seen before [7] or using genetic algorithms [27]. Several recommendation systems use a hybrid approach to avoid certain limitations of content-based and collaborative systems [6, 24]. However, often different recommendations may be required under different circumstances by different users. Current approaches do not support flexible, customizable, recommendations.

Furthermore, they deal with applications that have two types of entities, users and items (e.g., movies, web pages) and they recommend top N items to a user. Users and items are represented by rudimentary profiles (e.g., as sets of ratings or keywords) and recommendations are based on profile comparisons. For example, the content-based component of the Fab system [6] recommends web pages to users and represents web page content with the 100 most important words. Similarly, the Syskill & Webert system represents documents with the 128 most informative words [21]. While recommendations are based on a limited understanding of the world, many real applications use much richer data residing in databases. Different types of entities may co-exist in a single database (e.g., books, authors, publishers, and so forth) with several attributes. Current approaches are not very expressive, e.g., they cannot provide recommendations for different entities taking into account different subsets of their attributes.

The inherent limitations of recommendation systems have been acknowledged [5, 15] and some extensions have been recently proposed, such as incorporating multi-criteria ratings into recommendations [3]. The language RQL has been the first proposal that allows users to formulate recommendations in a flexible manner [4]. However, RQL queries are not very expressive because they are formulated on a pre-specified multidimensional cube of ratings.

In contrast to existing works on recommendations, in this paper, we do not propose or extend a particular recommendation method. We propose a general and open framework for expressing and processing flexible recommendations over relational data that covers existing as well as allows capturing novel recommendation paradigms. As part of our effort, we have also implemented a flexible recommendation interface in our system that allows students to request and customize their recommendations [16].

3. RECOMMENDATION FRAMEWORK

3.1 Data Model

Large amounts of data reside in structured form, and particularly in relational form, such as e-store, university, corporate, and scientific data. Our own university database is relational too. Therefore, we focus on databases that follow the relational model, which is enriched with some additional features.

A database comprises a set of relations. A relation R (denoted R_i when more than one relation is implied) has a set $\mathbf{A}$ of attributes. We will use $R.A_j$ to refer to an attribute in $\mathbf{A}$ or simply A_j when R is understood. A tuple r in R is a vector of single, numerical or categorical, values. We will use the notation $r[\mathrm{A}_j]$ to refer to the value of A_j in tuple r. An attribute that can be instantiated to a single value v is called *base attribute*. A relation with only base attributes is called *base relation*.

In addition, we introduce the concept of an extended relation.

DEFINITION 1. *An* extended relation S *has base attributes and extended attributes. An* extended attribute $E_i(A_1, ... A_e)$ *is instantiated to a relation with attributes* $A_1, ... A_e$.

Given a tuple s in the relation S, $s[E_i]$ is a relation with attributes $A_1, ... A_e$. If t is a tuple in this nested relation, then $t[A_j]$ gives the value of attribute A_j in this relation. We will use $E_i.A_j$ to refer to an attribute that is part of an extended attribute E_i or simply A_j when E_i is understood, and we call A_j an *embedded attribute*.

Consequently, under our semantics, the notion of an attribute value has a broader meaning capturing scalar values as well as relations. We consider that only base relations can be part of a tuple allowing one level of nesting. While our model and language could be generalized to arbitrary nesting, we have not seen a need for this generality in any of the practical recommendation scenarios we

Students

SuID	Name	Class	Status
1	Paul Little	2009	H
2	John Doe	2010	N

Ext_Students

SuID	Name	Comments (CourseID, Rating, Date)
1	Paul Little	C1, 5, 2 Feb 2008 C2, 6, 3 Dec 2007
2	John Doe	C1, 5, 15 Mar 2007 C2, 6, 12 Dec 2007 C5, 6.6, 22 Jun 2007 C7, 7, 22 Jun 2007

Figure 3: A base and an extended relation

have studied and implemented and we do not want to unnecessarily burden our design.

Example: We consider the course database with the base relations shown in Figure 1. Figure 3 shows an instance of the Students base relation. We can define the extended relation Ext_Students that contains all the base attributes of a student plus an extended attribute that represents the ratings made by each student as a single "unit of information" per student:

Ext_Students(SuID, Name, Comments(CourseID, Rating, Date))

Figure 3 shows an instance for this relation. To refer to the extended attribute, we use Ext_Students.Comments, and to the rating attribute within Ext_Students, we use Comments.Rating or simply Rating.

In a similar way, we can define other extended relations, such as the extended relation Ext_Courses that extends the base course relation with an attribute that shows all the instructors per year and term that a course was taught:

Ext_Courses(CourseID, DepID, Title, Instructors(Name, Year, Term))

In Section 3.3, we will show how these example extended relations can be defined using the *extend* operator.

Extended relations can be thought of as "views" that collect and group together information related to an individual entity and represent it as a single tuple that can be easily handled by the recommendation operators irrespective of the database structure (as we will see in Sections 3.4 – 3.6).

Although the issue of whether extended relations are materialized or not is orthogonal to their definition, in practice, extended relations may not be stored in the database. In the following subsections, we describe the operators required to handle extended relations and formulate recommendations. We start with the standard (core) relational algebra operators with set semantics and we extend them enabling interaction with extended relations as well.

3.2 Base Operators

We consider the following operators that can operate on base and extended relations.

- *Select*, σ_c, selects tuples from relation R, for which the condition c holds, i.e.,

  $$\sigma_c(R) := \{r | r \in R \wedge r \text{ satisfies } c\}$$

  R can be a base or extended relation and c is a condition that refers to *base attributes* of R. $\sigma_c(R)$ will be a base or extended relation depending on the type of R.

- *Project*, $\pi_{\mathbf{A}}$, creates a new relation by projecting R into a smaller set of its attributes, i.e.,

  $$\pi_{\mathbf{A}}(R) := \{r[\mathbf{A}] | r \in R\}$$

  $\mathbf{A}$ is a list of *base, embedded or extended* attributes from R. If $\mathbf{A}$ contains only base attributes, then the resulting relation is always a base one.

- *Join*, $\bowtie_c$, creates a new relation by combining tuples in R_i and R_j that meet some condition c, which refers only to *base attributes* of the two relations, i.e.,

  $R_i \bowtie_c R_j := \{(r_i, r_j) | r_i \in R_i \wedge r_j \in R_j \wedge r_i, r_j \text{ satisfy } c\}$

  A common type of join is natural join, which connects two relations by equating attributes of the same name, and projecting out one copy of each pair of equated attributes. This is denoted $R_i \bowtie R_j$ (the condition c is implied.)

The base operators defined above can handle extended relations and be combined with the new recommendation operators to be defined next but we have kept their original semantics. One could go further and define operators with extended semantics. For example, one could define that the select operator operates also on extended attributes. There is a rich literature on nested relational algebras that discuss extended, recursive operations such as nesting, unnesting and projections and joins on embedded relations [9, 11, 25]. We believe that such generality is not necessary for practical recommendations. Our framework can be easily extended to handle more generalized relational operators.

3.3 The Extend Operator

With many (normalized) database schemas, information that conceptually refers to a single entity (e.g., a student's course ratings) is often found in several relations. In our system, we want to represent such application entities with a single (extended) relation. For instance, a tuple in an extended relation could contain base information on a student (e.g., name, GPA), plus the set of courses a student has taken. For this purpose, we introduce a new operator that creates such extended attributes in the tuples of a relation.

DEFINITION 2. Extend, *ε, creates an extended relation by embedding a base relation R_j into another relation R_i, i.e.,*

$$R_i \; \varepsilon \; R_j := \{(r_i, v) | r_i \in R_i \; \wedge \; v := \pi_{\mathbf{A}}(r_i \bowtie R_j)\}$$

where $\mathbf{A}$ is the set of attributes of R_j.

In words, $R_i \; \varepsilon \; R_j$ is an extended relation that contains all the attributes and tuples from R_i plus an extended attribute whose (extended) value for each tuple r_i from R_i is a set of tuples from R_j that join to r_i on the common attributes. If there are no joining tuples, then the extended attribute will be an empty relation for r_i. The default name of this extended attribute is the name of R_j.

In the above definition, only R_j is required to be a base relation because we allow one-level nesting. R_i can be a base or extended relation, since any extended relation can have more than one extended attribute. The extend operator can be defined for multiple-level nesting as well.

Example: Using the extend operator we can generate the extended relation Ext_Students described earlier as follows:

PComments := $\pi_{\{\text{SuID,CourseID,Rating,Date}\}}$(Comments)

Ext_Students := $\pi_{\{\text{SuID,Name}\}}$(Students) ε PComments

In order to generate the extended relation Ext_Courses, we need to combine information found in more than two relations, as follows:

Schedules := $\pi_{\{\text{CourseID,Year,Term,InstrID}\}}$(CourseSched)

Instruct_Sched := $\pi_{\{\text{InstrID,Name}\}}$(Instructors) $\bowtie$ Schedules

Ext_Courses := $\pi_{\{\text{CourseID,DepID,Title}\}}$(Courses) ε Instruct_Sched

Note that joining over attributes with common names simplifies the presentation and the definition of the extend operator. It does

not impose any real constraints on its expressivity or on the relations and attributes that can be used with this operator because relations and attributes can be renamed accordingly. We can use a rename operator $\rho[R'(B_1, ...B_n)](R)$ that produces a relation identical to R but with name R' and attributes, in order, named B_1, ... B_n. For example, we can rename PComments to Comments in order to have the relation Ext_Students as shown in Figure 3:

$\text{Ext\_Students} := \pi_{\{\text{SuID,Name}\}}(\text{Students})\ \varepsilon\ \rho[\text{Comments(PComments)}]$

3.4 The Recommend Operator

Recommendations are based on comparisons (e.g., courses are rated by comparing their topics to a student's interests, students are compared to a particular student based on their ratings in order to find similar students, and so forth). In order to "build" recommendations, we will have a library of comparison functions that implement common recommendation tasks of the form defined below.

DEFINITION 3. *A* comparison function cf *is a parameterized function that maps a tuple* t *to a single scalar value* u *by comparing it to another tuple* s*.*

$$u = cf[P](t, s)$$

where t *and* s *are the two inputs to be compared and* P *denotes the parameters used in the function.*

For example, we may wish to apply a comparison function to some part of two tuples, i.e., to an attribute A of them; then, we would specify $cf[A](t, s)$. It could also be to a set of attributes for multi-criteria recommendations [4]. We do not make any particular assumptions for the tuples t and s, e.g., that they should have the same number of attributes or that corresponding attributes should have the same type, allowing in this way a wide spectrum of functions to be defined, as the forthcoming examples will illuminate. Also, we do not assume any properties for cf. It could be any function that computes a relationship between its two inputs based on either probabilistic approaches (e.g., [14]) or more traditional item-to-item correlations, such as Pearson's correlation, Kendall's tau, Euclidean distance, and Jaccard index, as well as user-defined, domain-specific metrics. It could also be a prediction model learnt offline (e.g., [8]). We illustrate several possibilities below.

• *Comparisons of string values*. For example, we could consider a function that measures the Jaccard similarity between two string values, each one belonging to one of the input tuples of the function and treated, for the purposes of comparison, as a bag of words, i.e.:

$$\text{Jaccard}[A](t, s) = \frac{|t[A] \bigcap s[A]|}{|t[A] \bigcup s[A]|}$$

Using this function, we could, for example, compare course descriptions using $\text{Jaccard}[\text{Description}](t, s)$, where t and s represent courses, or compare instructors on the basis of their names with $\text{Jaccard}[\text{Name}](t, s)$, where t and s are two tuples in Instructors.

• *Comparisons of numerical values*. For example, we could use a function that measures the distance of two numbers, i.e.:

$$\text{Distance}[A](t, s) = t[A] - s[A]$$

Such a function could be used to compare students based on their GPAs, i.e., $\text{Distance}[\text{GPA}](t, s)$, where t and s belong to Students, or to compare courses based on their units, i.e., $\text{Distance}[\text{Units}](t, s)$, where t and s represent courses.

• *Using conditional probabilities*. An alternate way of computing the similarity between two tuples is to use a measure that is based on the conditional probability of observing one of the tuples given that the other tuple has already been observed. For example, a simple comparison function that computes conditional probability is:

$$\text{Probability}[A, B, R](t, s) = \frac{|\sigma_{R.A=t[A]}(R) \bowtie_B \sigma_{R.A=s[A]}(R)|}{|\sigma_{R.A=s[A]}(R)|}$$

This function computes the number of pairs of tuples from a relation R with the same value on an attribute B and with $t[A]$ as the value of the attribute A in one of the pair's tuples and $s[A]$ as the value in the other pair tuple divided by the number of tuples in R with $s[A]$ as the value of the attribute A. For example, $\text{Probability}[\text{CourseID, SuID, StudentHistory}](t, s)$ finds the similarity between two courses based on the number of students (identified by their SuID) that took both courses t and s (identified by their course id CourseID) divided by the total number of students that took s.

• *Comparisons of extended values*. We could also define functions for comparing two extended values, each one belonging to one of the input tuples, using some metric, such as the Euclidean distance, Pearson coefficient, etc. For example, a comparison function using the Euclidean distance is:

$$\text{Euclidean}[E, A_1, A_2](t, s) = \sqrt{\sum_{\substack{i \in t[E] \\ j \in s[E] \\ i[A_1]=j[A_1]}} (i[A_2] - j[A_2])^2}$$

This function compares two tuples t and s on their extended attribute E by summing up the squared differences of the values of attribute A_2 in all tuples i and j from the nested relations $t[E]$ and $s[E]$, respectively, that join on A_1. Then, we could, for instance, use $\text{Euclidean}[\text{Comments, CourseID, Rating}](t, s)$ to compare students based on their common ratings, where students t and s are taken from Ext_Students.

• *Comparisons of single values to extended values*. We could define a comparison function that compares two values of different types, e.g., a single value to an extended value, i.e., to a relation E. For example, we could define a function that tries to locate a single attribute value within E, and if it finds it, it returns the value of an attribute B in the tuple where the value was found, i.e.:

$$\text{Identify}[A, E, B](t, s) = v[B] \text{ if } \exists v \in s[E]\ s.t.\ t[A] = v[A]$$

For instance, if t is a course and s is a student from Ext_Students, $\text{Identify}[\text{CourseID, Comments, Rating}](t, s)$ returns the rating of student s for course t based on the course id. In Section 3.6, we will see examples that illustrate how these functions can be used for generating recommendations.

Comparison functions compare one tuple to another tuple. It is frequently desirable to compare one tuple to a set of tuples, e.g., a student to a group of students. For this purpose, we define an aggregation comparison function.

DEFINITION 4. *An* aggregation comparison function a *is a parameterized function that maps a tuple* t *to a single scalar value* u *by comparing it to a set of tuples* S*:*

$$u = a[cf, P](t, S) = a[P](\{cf(t, s_i) | s_i \in S\})$$

Essentially, a *takes as parameter the comparison function to use for the tuple-to-tuple comparisons and generates a list of comparison function values* $cf(t, s_1), ...cf(t, s_n)$*,* $\forall s_i \in S$*, which are then combined through* a *into a single value.*

An aggregation comparison function combines all partial values into a final one using a function, such as the maximum, the average, and so forth, and essentially signifies the relationship of tuple t to the set of tuples S. P denotes other parameters to be used in the function. For example, we may wish to compute the weighted average of the partial comparison values using as weights the values of attribute A of the tuples in S:

FriendCourses	CourseID	Title	Score
t1	C2	Advanced Programming	10
t2	C5	AI Techniques	7
t3	C10	Graphics	9

ReqCourses	CourseID	Title	Score
ta	C2	Advanced Programming	10
tb	C10	Graphics	7
tc	C22	Compilers	8

(a) different recommendations

FriendCourses ß ReqCourses	CourseID	Title	Score	Bscore
t1, ta	C2	Advanced Programming	10	2
t2	C5	AI Techniques	7	1
t3	C10	Graphics	9	2
tb	C10	Graphics	7	2
tc	C22	Compilers	8	1

(b) occurrence-based blending

FriendCourses ß ReqCourses	CourseID	Title	Score	Bscore
t1	C2	Advanced Programming	10	0.7
t2	C5	AI Techniques	7	0.49
t3	C10	Graphics	9	0.63
ta	C2	Advanced Programming	10	1
tb	C10	Graphics	7	0.7
tc	C22	Compilers	8	0.8

(c) normalized blending

FriendCourses ß ReqCourses	CourseID	Title	Score	Bscore
ta, t1	C2	Advanced Programming	10	10
t2	C5	AI Techniques	7	2.88
t3	C10	Graphics	9	7.82
tb	C10	Graphics	7	7.82
tc	C22	Compilers	8	4.7

(d) weighted average blending

Figure 4: Blending examples

$$\text{W\_Avg}[cf, A](t, S) = \frac{\sum_{s_i \in S} s_i[A] * cf(t, s_i)}{\sum_{s_i \in S} s_i[A]}$$

As mentioned earlier, (aggregation) comparison functions, like the ones we illustrate here, will belong to a library of functions to be used with the *recommend* operator.

DEFINITION 5. Recommend, $\triangleright_{cf,a}$, *outputs the tuples of a relation* R_i *augmented with an attribute with default name* score, *whose value for each tuple is computed by comparing this tuple with the tuples of a relation* R_j, *as follows:*

$$R_i \triangleright_{cf,a} R_j := \{(r_i, v) | r_i \in R_i \ \wedge \ v := a[cf](r_i, R_j)\}$$

In other words, the *score* value of each tuple r_i in R_i is produced by comparing r_i to each tuple in R_j using function cf and then aggregating using function a.

The new attribute can be renamed, if so desired. Furthermore, we may specify a condition on which tuples from R_j will be considered for each tuple in R_i. For example, we may want to compare a course only with courses offered in previous terms. Another possibility is a select operator that may follow in order to filter the output of a recommend operator, e.g., selecting only tuples with score above a threshold. In these cases, we will use $\triangleright_{cf,a,c}$ as a shorthand. In what follows, for simplicity, we omit cf, a, and c from the operator notation whenever they are understood or not required (e.g., when there is no filtering), and we simply write $R_i \triangleright R_j$.

3.5 The Blend Operator

Often, we may want to combine recommendations generated through two different processing paths into one. For example, we may have a set of courses that can be recommended based on the student history and courses suggested by friends or courses that are required for graduating, and we want to provide one recommendation. For this purpose, we introduce the *blend* operator.

DEFINITION 6. Blend, β_M, *outputs the tuples from its two input relations,* R_i *and* R_j, *augmented with an attribute with default name* bscore, *as follows:*

$$R_i \ \beta_M \ R_j := \{(r, v) | r \in R_i \cup R_j \ \wedge \ v := M[R_i, R_j](r)\}$$

Tuples in the input relations should be union-compatible. In a sense, the blend operator is an advanced union, which does not only combine the tuples from its input relations but also augments each tuple with an attribute whose value is computed by taking into account the tuple in perspective with the tuples coming from both relations using a blending method M. The name of the new attribute can be different from the default by renaming.

Example: Figure 4(a) shows two sets of courses, which have been generated in different ways: the set ReqCourses is recommended based on degree requirements and the set FriendCourses is recommended by a student's friends. A course in either set is described by the course code, title and recommendation score. The attributes may have been renamed before the relations are presented in the blend operator using a rename operator.

• A blending method $\text{Occurrences}[A, R_i, R_j]$ counts the occurrences of each tuple, identified by the attribute A, in both relations. Figure 4(b) shows the result of mixing the sets of courses using $\text{Occurrences}[\text{CourseID}, \text{ReqCourses}, \text{FriendCourses}]$. For example, the Advanced Programming course is found in both sets in tuples t_1 and t_a. The Occurrences blending method computes the same value for both tuples, i.e., $\text{Occurrences}(t_1) = \text{Occurrences}(t_a) = 2$. This may not occur with other methods, like the next one.

• A $\text{Norm\_Blend}[B, w_i, w_j, R_i, R_j]$ method computes a *bscore* for each tuple by normalizing the value of its attribute B and weighting it using a real number in $[0, 1]$:

$$\text{Norm\_Blend}[B, w_i, w_j, R_i, R_j](t) = \begin{cases} \frac{t[B]}{\max B} * w_i & \text{if } t \in R_i, \\ \frac{t[B]}{\max B} * w_j & \text{if } t \in R_j. \end{cases}$$

For the two course sets, we could use $\text{Norm\_Blend}[\text{Score}, 0.7, 1, \text{FriendCourses}, \text{ReqCourses}]$, which gives more weight to the courses in the second set because satisfying the requirements is more important. In the combined set of courses shown in Figure 4(c), the Advanced Programming course now appears with different bscores: a $\frac{10}{10} * 0.7 = 0.7$ in t_1 and a $\frac{10}{10} * 1 = 1.0$ in t_a. On the other hand, the Graphics course is highly recommended by the student's friends but it has a lower score according to the requirements: a 9 in t_3 in contrast to a 7 in t_b. However, due to giving different weight to the two sets, the *bscore* for t_3 is $\frac{9}{10} * 0.7 = 0.63$ and for t_b is $\frac{7}{10} * 1.0 = 0.7$.

• A $\text{Wavg\_Blend}[A, B, w_i, w_j, R_i, R_j]$ method computes a *bscore* for each tuple, identified by an attribute A in the two input relations,

as the weighted average of the tuple values in the attribute B.

$$\text{Wavg\_Blend}[A, B, w_i, w_j, R_i, R_j](t) = \frac{w_i * t[B] + w_j * t[B]}{w_i + w_j}$$

For example, using Wavg_Blend[CourseID, Score, 0.7, 1, FriendCourses, ReqCourses], we can take into account that the Advanced Programming course is recommended both based on the requirements and by friends and give it a high score $\frac{0.7*10+1.0*10}{0.7+1.0} = 10$, as Figure 4(d) illustrates.

As in the case of comparison functions, blending methods will be part of an external library of possible blending methods, so that the application designer does not have to code them.

3.6 Recommendation Workflows

The recommend and blend operators capture the essence of most recommendation approaches and they can be composed and combined with the more traditional select, project, join operators to describe recommendations.

DEFINITION 7. *A* recommendation workflow (query) *is a directed acyclic graph of interconnected operators that describes how recommendations are generated.*

Below, we present several examples of workflows that illuminate the new recommendation capabilities and the expressivity of the framework. These workflows describe recommendations that are hypothetically requested by a student, Alice. We will show how common recommendation strategies (found in literature) can be expressed as parameterized workflows with our operators, as well as illustrate some of the new recommendation possibilities.

Clearly, we do not expect that Alice or any user will describe desired recommendations at this level rather users will use appropriate user interfaces. For the sake of presentation, in each example, we first explain the kind of comparison expected and how this is captured in the recommend operator, and then, in the workflows, we use only the symbol of the recommend operator without any parameters since they will be understood in the context of each example. Also, we do not show some common sense operations, such as excluding from the set of recommended courses all courses already taken by Alice, or selecting the courses with the highest recommendation scores.

Example 1 (*Related courses*): Alice is browsing the page of a course in the system with title "Programming: Part One" (and id $C22$). We would like to show other related courses that are also offered in 2008. Assume that the comparison function used for string values is Jaccard. Then, similar courses could be found using the recommend operator $\triangleright_{\text{Jaccard[Title]}}$ as follows:

ThisCourse := $\sigma_{\text{CourseID}=C22}$(Courses)
CandidateCourses := $\sigma_{\text{Year}=2008}$(Courses)
RelatedCourses := CandidateCourses $\triangleright$ ThisCourse

RelatedCourses contains courses with a *score* attribute that reports the strength of the recommendation based on how similar a course is to the one currently browsed by Alice. For example, it could contain a course named "Programming: Part Two" with a (Jaccard) score 0.5, and a course named "Advanced Programming Methodology" with a score 0.2. This example demonstrates the use of a single recommend operator for comparing a set of tuples to a single tuple. The next example shows how the recommend operator can be used for comparing a set of tuples to another set of tuples. Examples with multiple recommend and blend operators follow.

Example 2 (*Content-based recommendations*): Alice (with id 1234) has taken some courses on related topics (e.g., on literature, writing, etc) and she is interested in courses offered this year that will help her improve her knowledge on these topics. Therefore, she asks for suggestions on courses based on her course history. Course similarity will be determined based on the course descriptions. Each candidate course obtains the minimum Jaccard similarity score when compared to the list of Alice's courses, i.e., we will use the recommend operator $\triangleright_{\text{Jaccard[Description]},min}$:

AliceCourses := $\sigma_{\text{SuID}=1234}$(StudentHistory) $\bowtie$ Courses
CandidateCourses := $\sigma_{\text{Year}=2008}$(Courses)
RecCourses := CandidateCourses $\triangleright$ AliceCourses

Example 2 is a workflow for content-based recommendations.

Example 3 (*Nearest-neighbor collaborative filtering*): Alice's best friend is Joanne (with id 444) and Alice is curious to see what courses would be recommended to her friend by her colleagues with similar tastes, i.e., with similar ratings as Joanne (say the system allows friends to see each other's recommendations). We could use the inverse Euclidean (inv_Euclidean) function, so that students with more similar ratings will have higher weight. These students could be found using $\triangleright_{\text{inv\_Euclidean[Comments,CourseID,Rating]}}$:

Ext_Students := Students ε $\pi_{\{\text{SuID,CourseID,Rating}\}}$(Comments)
ThisStudent := $\sigma_{\text{SuID}=444}$(Ext_Students)
Other_Students := $\sigma_{\text{SuID}<>444}$(Ext_Students)
RStds := Other_Students $\triangleright$ ThisStudent

Then, courses could be rated by taking the weighted average of the ratings provided by these students. The recommend operator applied uses the Identify comparison function to extract the user ratings given to each course and aggregates them into one recommendation score per course using the W_Avg[Score] aggregation comparison function, where the scores of the students computed by the previous recommend operator are used to weigh their ratings.

Coll_Courses := Courses$\triangleright_{\text{Identify[CourseID,Comments,Rating]},\text{W_Avg[Score]}}$RStds

This workflow captures nearest-neighbor collaborative filtering. We could use other functions in place of the inverse Euclidean, such as the Pearson, and also replace the W_Avg with e.g., the weighted average of rating deviations from the neighbor's mean rating. Then, our workflow would represent the GroupLens collaborative filtering approach [22]. Note that it is easy to parameterize our workflow (unlike other approaches), and hence it is possible to generate recommendation for a person (Alice) assuming that they are intended for a different student (e.g., Joanne).

At this point, we observe that the recommend operator treats student ratings as a (virtual) attribute of students (namely, Comments), in the same way that it would do with base attributes, such as the description of courses in the previous example. However, in our relational database, student ratings are stored separately from the rest of information regarding students, so we would have to list as inputs of the recommend operator multiple relations. The recommend operator would also have to know how to join these multiple inputs and group ratings that refer to the same student making it cumbersome to define the operator. In order to enable our recommend operator to operate on attributes of entities transparently, i.e., irrespective of whether these are base attributes, such the GPA, or "extended" attributes, such as a set of ratings, we need the extend operator. In our example workflow, students are extended with an attribute Comments that groups their course ratings, so that the set of ratings for each student can be "viewed" as another attribute of the student in the workflow.

We can also easily define novel types of recommendations, as the following examples illustrate.

Example 4 (*Many recommend and blend operators*): Alice wants recommendations from students that are similar to her friend as before but she also wants them to be as good as she is based on

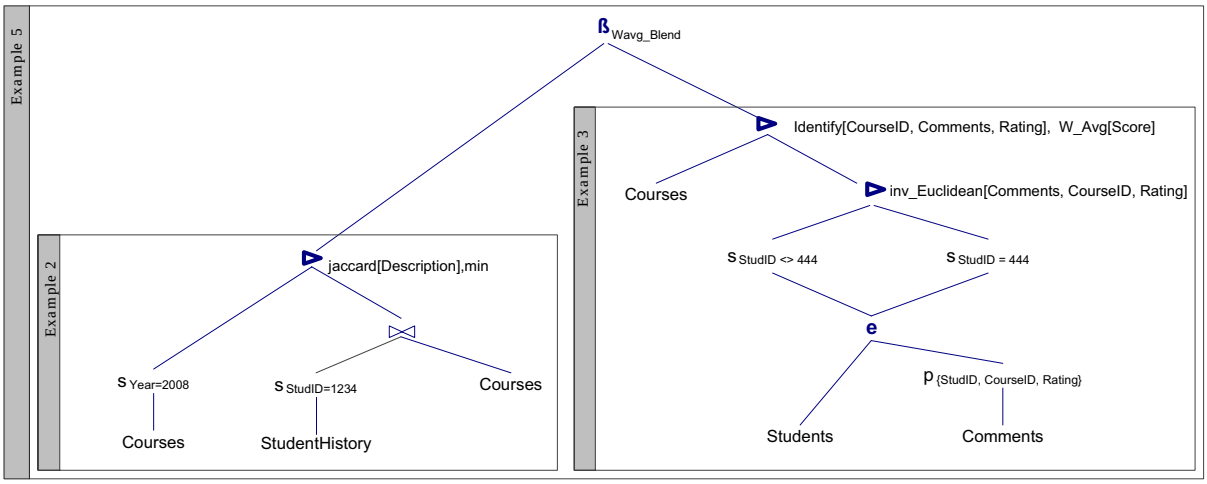


Figure 5: Recommendation workflows

GPA. So, continuing the previous example from the point that RStds have been found, we also want students that have similar GPA with Alice. As a measure of student similarity, we can take the inverse distance of their GPAs (inv_Distance), as follows:

AliceStudent := $\sigma_{\text{SuID}=1234}$(Students)

RStds2 := OtherStudents $\triangleright_{\text{inv\_Distance[GPA]}}$ AliceStudent

Then, we can compute an overall score for students taking into account how similar they are to both Joanne and Alice using the blending method Wavg_Blend[SuID, Score, 1, 1, RStds, RStds2]:

QStds := RStds $\beta_{\text{Wavg\_Blend}}$ RStds2

Finally, course recommendations can be obtained in the same way as when computing Coll_Courses before, with the only difference that bscores will play the role of weights in the computation of the weighted average of student ratings:

StricterRecs := Courses$\triangleright_{\text{Identify[CourseID,Comments,Rating],W_Avg[Bscore]}}$ QStds

Consequently, this example query involves a total of three recommend operators plus a blend operator.

Example 5 (*Blending recommendations*): Assume we want to blend course recommendations based on Alice's course history (RecCourses) with those from similar students (Coll_Courses) giving more weight to the latter in order to increase diversity in the final recommendations. We could, for example, use a blending method Wavg_Blend[CourseID, Score, 0.7, 1, RecCourses, Coll_Courses]:

MixedCourses := RecCourses $\beta_{\text{Wavg\_Blend}}$ Coll_Courses

Using our framework, one can define workflows that go beyond course recommendations, as the upcoming examples show.

Example 6 (*Classification*): Honors students achieve high grades in their course work and are often rewarded for their achievements. The university wants to keep track of any student with an exceptional performance. We would like to notify the program administrators of students that may qualify as honor students. To see if Alice qualifies, we compare her to the honors students (having status 'H' in our data) based on the courses taken and grades earned. We could use again the inverse Euclidean (inv_Euclidean) to compute student-to-student similarity and take Alice's average similarity to all honors students as a score for her. Hence, we will use the recommend operator $\triangleright_{\text{inv\_Euclidean[StudentHistory,CourseID,Grade],Avg}}$:

HStudents := $\sigma_{\text{status}=\text{"H"}}$ (Students) ε $\pi_{\{\text{SuID,CourseID,Grade}\}}$ (StudentHistory)
AliceStudent := $\sigma_{\text{SuID}=1234}$(Students) ε $\pi_{\{\text{SuID,CourseID,Grade}\}}$ (StudentHistory)
IsHStudent := AliceStudent $\triangleright$ HStudents

Note how we can parameterize the workflow and let the selection conditions being determined at run time so that the workflow compares any student to any group of students.

Example 7 (*Recommending a major*): We can easily devise a new workflow that recommends a major to Alice based on her course history and so far performance (in fact, we are currently integrating recommendations for majors into our system.) We first need to find students that have already selected a major and compare their course selections and performance to Alice's. We will use $\triangleright_{\text{inv\_Euclidean[StudentHistory,CourseID,Grade]}}$.

MajStudents := $\pi_{\{\text{SuID}\}}$($\sigma_{\text{Classification}=\text{"major"}}$ (StudentStudies $\bowtie$ StudyPrograms))
Ext_MajStudents := MajStudents ε $\pi_{\{\text{SuID,CourseID,Grade}\}}$ (StudentHistory)
AliceStudent := $\sigma_{\text{SuID}=1234}$(Students) ε $\pi_{\{\text{SuID,CourseID,Grade}\}}$ (StudentHistory)
RStds := Ext_MajStudents $\triangleright$ AliceStudent

Then, we will rate majors following a voting scheme. We will sum up the scores of the students that have followed each major.

RecMajors := StudyPrograms $\triangleright_{\text{Identify[StudyPrgID,StudyPrgID,Score],sum}}$ RStds

Finally, let us see recommendations from a different domain.

Example 8 (*Item-to-item movie recommendations*): Alice decided to visit her favorite online movie site, FlexMDB, and download a movie. The movie site keeps the following database:

Movies(Mid, Title, Year), Viewer(Vid, Name, Age), Saw(Vid, Mid, Rating)

Say FlexMDB has adopted FlexRecs and offers various customizable recommendations. Among them, it offers recommendations adopting the Amazon item-to-item collaborative filtering approach described in [17]. At a high-level, item-to-item collaborative filtering consists of two distinct components: the first component builds a model that captures the relations between the different items, and the second component applies this pre-computed model to derive the recommendations for an active user [23]. The offline part builds a similar-items table by finding items that customers tend to purchase together, as follows:

MoviePairs := $\pi_{\{\text{Mid,Mid2}\}}$(Saw $\bowtie_{\text{Vid}}$ ρ[Saw(Vid, Mid2, Rating2)](Saw))
Ext_Movie1 := MoviePairs ε Saw
Ext_Movie2 := MoviePairs ε ρ[Saw(Vid, Mid2, Rating)](Saw)

MoviePairs keeps the pairs of movies seen together. The extended relation Ext_Movie1(Mid, Saw(Vid, Rating), Mid2)) keeps all the ratings given by viewers for the first movie of a pair and Ext_Movie2 keeps the ratings for the second movie of a pair. To compute similar movies, we compare Ext_Movie1 to Ext_Movie2 using the operator $\triangleright_{\text{inv\_Euclidean[Saw,Vid,Rating],\{Mid,Mid2\}}}$, which ensures that only paired

movies are compared, i.e., tuples from the two relations are compared if they have the same {Mid, Mid2}.

$\text{Model} := \pi_{\{\text{Mid},\text{Mid2},\text{Score}\}}(\text{Ext\_Movie1} \triangleright \text{Ext\_Movie2})$

Given the relation Model, the algorithm finds movies similar to those rated by the user, aggregates those items (we will sum up the similarity scores of the movies to the user's movies) in order to recommend the most correlated items.

$\text{AliceMovies} := \sigma_{\text{Vid}=3232}\text{Saw} \bowtie_{\text{Mid}} \text{Model}$
$\text{RecMovies} := \text{AliceMovies} \triangleright_{\text{Score},\text{Sum},\text{Mid2}} \text{AliceMovie}$

AliceMovies has all the movies that Alice has rated (identified by Mid) and movies similar to them (identified by Mid2). Recommended movies, i.e., RecMovies, are selected from the similar movies.

Figure 5 shows possible expression trees representing some of the example workflows above.

With FlexRecs, it is easy to define novel recommendations, since the new operators can be composed and combined with more traditional operators to form recommendation workflows. To execute recommendation workflows, algebraic laws will help an optimizer to enumerate or transform plans in search of efficient ones for generating the desired recommendations.

Under the general operator semantics we assume, we cannot make any assumptions for the general properties of all the operators. For example, we cannot make any assumption w.r.t. the associativity or commutativity of any blend operator because its properties depend on the semantics assumed in any given implementation and, in particular, on the blending method used. For example, the bscores of tuples may depend on the order in which the input sets of tuples are presented to the operator (e.g., the operator may give more weight to the tuples of its first input relation.) Of course, in any particular realization of the framework, one may give more specific or restricted semantics to some operator and be able to define additional rules. For example, a blend operator based on an Occurrences method will be always commutative. On the other hand, the extend and recommend operators are neither commutative nor associative based on their definition. There are laws that can be derived directly from the definition of the operators. For example, selection can be pushed through an extend to its first argument: $\sigma_c(R_i \,\varepsilon\, R_j) \equiv \sigma_c(R_i)\,\varepsilon\, R_j$.

4. SYSTEM ARCHITECTURE

In this section, we present a prototype system that compiles and executes FlexRecs workflows on top of a database. Implementation on top of an existing database system has a number of advantages, such as implementation ease, portability and faster deployment and testing of the framework in our course planning tool. It also offers a proof of the framework's feasibility. Extending the database query engine with the processing capabilities required for supporting flexible recommendations is the next step in order to try more sophisticated optimizations. A high level representation of the architecture is shown in Figure 6.

Our strategy for processing workflows reduces the amount of data saved in temporary relations during execution. For instance, extended relations are never materialized. On the other hand, results generated by any recommend or blend operators, which are later re-used in the same workflow are materialized for efficiency. Our workflows are converted into sequences of SQL queries, which are then executed by the database system. Thus, we can take advantage of the underlying query optimization. The *FlexRecs* engine has the following parts.

The *Workflow Manager* allows a designer to define different recommendation workflows and end-users to invoke any of the defined workflows (through an appropriate user interface) with different inputs and receive customized recommendations. The Workflow Manager hides the details of how flexible recommendations are generated, such as the details of the algorithms and structures used to implement the functionality of the operators that comprise a recommendation workflow and the execution order of operators, which can be different from the workflow definition in order to optimize the process.

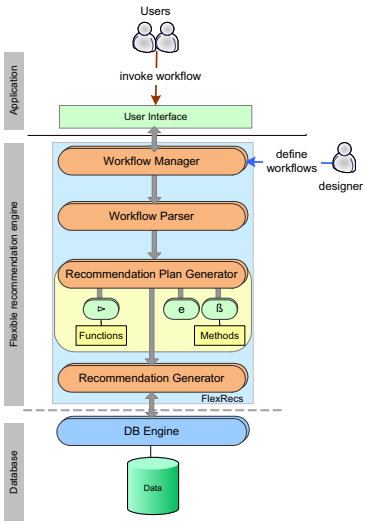


Figure 6: System Architecture

The *Workflow Parser* takes as input a workflow and constructs an expression tree, like these shown in Figure 5, keeping the order of the operators as defined in the workflow, since no real processing is performed at this level.

The *Recommendation Plan Generator* takes as input an expression tree and generates a recommendation execution plan. The generated plan comprises a sequence of SQL statements and function calls and specifies how and in what order operators are executed and any results that are shared in a particular workflow and need to be materialized. This module leverages the underlying DB engine's query optimizer for implementing the new operators, and in particular, its ability to efficiently compute joins and aggregations. In addition, it allows calling functions that implement various comparison functions and blending methods. When possible, these functions are compiled into the SQL statements; in other cases external functions can be called by the SQL statements. In this way, the system can support new types of recommendations by extending its library. In the next section, we describe in more detail the recommendation plan generation process.

The *Recommendation Generator* executes a plan and returns the recommendations. It sends the SQL queries to the DB engine, assembles any intermediate results and invokes the functions used in the plan for comparing and blending tuples.

4.1 Recommendation Plan Generation

The system builds a recommendation plan by traversing an expression tree from its root and going down to its leaves. To illustrate the recommendation plan generation process and the implementation of the new operators, we walk through the following examples.

Example 3 (cont'd): We return to Example 3 (Sec. 3.6) and we consider its expression tree (Figure 5), which has two recommend operators. We will first see the set of queries that can be generated for implementing the lower operator.

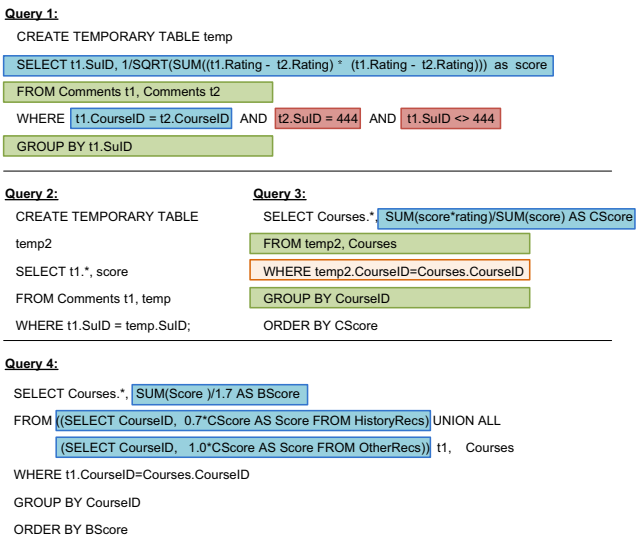


Figure 7: An example recommendation plan

The corresponding subtree contains select, project, and extend operators, which are all combined with the root recommend operator into one SQL query (Query 1 in Figure 7). This query has several parts (shown shaded in Query 1), each one mapping to one of the operators: (a) the select operators have been included as conditions in the WHERE clause, (b) the recommend operator, which compares students using the inverse Euclidean comparison function on their course ratings, is implemented by combining the aggregation functions that are supported by the underlying database, (c) the extend operator is implemented by a GROUP BY clause. Observe how, in this example, the query does not join the relations Students and Comments specified in the expression tree, because the system can recognize that all the attributes required by the extend and recommend operators can be provided by the latter relation. This query creates a temporary in-memory table that contains two attributes for each student: the student id and a score.

The higher recommend operator computes course recommendations based on the ratings provided by the similar users found in the previous step, i.e., it makes use of the materialized results of the lower recommend operator. Queries 2 and 3 correspond to this operator and use results generated by lower operators. Since the result of the previous recommend operator has been generated by aggregating the student ratings, we need again to associate students with their ratings in order to compute course recommendations combining student scores and ratings. Therefore, Query 2 combines for each student the score and the ratings information into one relation. Then, Query 3 contains a hidden extend operator in its GROUP BY clause. The computations required by the recommend operator, which now uses a comparison function (Identify) and an aggregation comparison function (W_Avg), are again realized by leveraging the database's existing aggregation capabilities. The course recommendations may be ordered by their score for presenting them to the user.

Instead of having Query 2 generating an in-memory table and then Query 3 using this table, we could have generated a single, equivalent, query. Splitting the computations in simpler queries and exploiting in-memory tables allows more efficient recommendation processing over the DB engine that we use.

The next example shows how the blend operator can be implemented on top of a database. For the sake of brevity, we only detail the parts of the recommendation plan that refer to this operator.

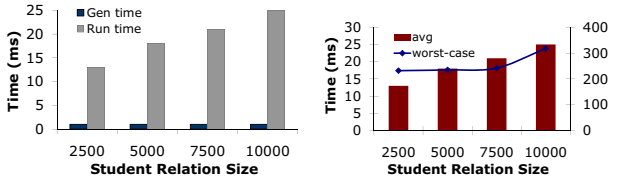
Example 5 (cont'd): We consider the expression tree corresponding to Example 5 in Figure 5. Recall that this tree corresponds to a workflow that combines recommendations based on a student's history with course recommendations from other students. The recommendation plan for Example 3 is part of the plan for this example, with the only difference that the results of Query 3 in Figure 7 are materialized as an in-memory table (for the reasons explained earlier), and they are not ordered in this phase, since they are not the final results yet. Let us assume that this materialized table is named OtherRecs and that a second materialized table, HistoryRecs, holds the recommendations based on the student history (corresponding to Example 2 in Figure 5). Query 4 in Figure 7 shows how the blending method Wavg_Blend can be implemented with the help of SQL. Scores in OtherRecs are given a greater weight than scores in HistoryRecs. Note that all queries are built on the fly based on the workflow. Hence, the weights are not hard-coded but they comprise inputs of the blending method specified in the workflow. This query generates the final recommendations returned by the system, which may be additionally ordered for presentation purposes.

As we have seen in the examples above, for each recommend operator, the system builds a set of queries that implement the requested comparison function and group together any operators that are found in the subtree under this recommend operator. We support an extensible library of (aggregation) comparison functions. Interestingly, a large number of functions, such as Pearson, Jaccard, Cosine, W_Avg, Avg can be implemented by computing and combining aggregations supported by the underlying database (as e.g., in Query 1 in Figure 7). In such cases, the part of the recommendation plan that corresponds to the recommend operator comprises mainly aggregate queries. The standard query operators, such as selections and projections, are mapped to appropriate SQL clauses, which are inserted into these queries.

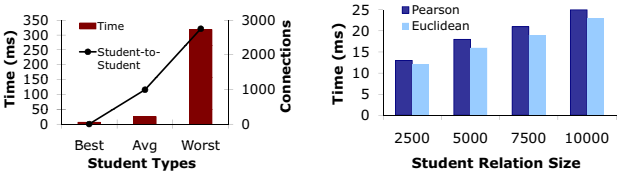
Whenever there is an extend operator, no extended relation is actually materialized in memory because that would require fetching tuples (e.g., students) in memory, augmenting them with a new attribute, and executing a (possibly large) number of queries in order to populate this attribute with joining tuples (e.g., courses) from another relation. To save unnecessary I/O operations and tuple processing, the joins implied by an extend operator are executed only when tuples are actually requested by some upper in the expression tree operator (typically a recommend operator) and the information related to a single entity is grouped together using SQL. For instance, Query 1, which performs the necessary aggregations required by the respective recommend operator, also realizes the extend operator that is in the subtree of the recommend operator as a GROUP BY clause.

In addition, as we have seen, if there are more than one recommend operator or if a recommend operator is followed by a blend in an expression tree, then a separate set of queries is built for each of these operators. The results generated by the intermediate recommend operators are materialized in order to avoid building complicated, possibly nested, queries and reuse results of earlier computations. The output of an intermediate recommend operator is an in-memory relation with two attributes: a tuple id and a score.

Blend operators are treated in a similar way as recommend operators. A set of queries is built that implement the requested blending method and also combine any standard operators, such selections and projections, that are found in the subtree under a blend operator. We support an extensible set of blending methods. Again, we can leverage the expressivity of SQL joins and aggregations to implement several methods, as illustrated in Query 4. Finally, results of blend operators that are internal nodes in an expression tree are materialized as in the case of the recommend operators.



(a) Collab. Filtering phases (b) Collab. Filtering run times



(c) Student connections (d) Comparison functions

Figure 8: Performance results: collaborative filtering

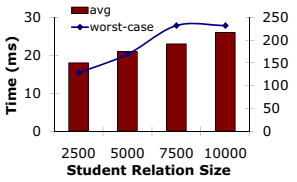
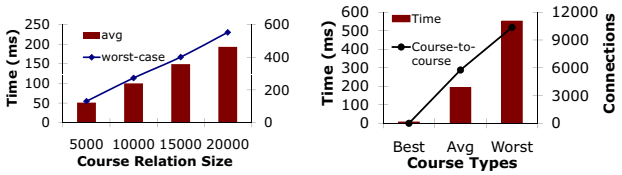


Figure 9: Performance Results: majors workflow



(a) Related courses run times (b) Course connections

Figure 10: Performance Results: related courses workflow

5. EXPERIMENTS

In this section, we examine the feasibility and performance of flexible recommendations. For this purpose, we study different workflows with different characteristics, i.e., different comparison functions, different number of operators, different input sizes, and different outputs. Our FlexRecs recommendation system is written in Java on top of MySQL. In our evaluation, we used real data from our production system. Times are in msecs.

Collaborative Filtering. This workflow (Example 3, Sec. 3.6) generates course recommendations for a student based on the ratings of similar students. It is interesting because it is a very common approach in practice and it uses two back-to-back recommend operators. Figure 8(a) shows execution times for different-size sets from the student relation. The sets have been generated so that: the 2.5K set is contained in the 5K set, which is contained in the 7.5K set, and so forth. The workflow is run on every user in the respective set and we take average times. Similarity between students is computed using Pearson. *Gen Time* is the time for building a recommendation plan, i.e., building the set of SQL queries. *Run Time* is the time required to execute a plan and collect the recommendations. Query execution times dominate because the queries depend heavily on aggregations that are required for computing similarity scores. In what follows, our evaluation focuses on run times only.

Figure 8(b) shows the average execution times (on the left Y-axis) and the worst-case execution times (on the right Y-axis) for different sizes of the student relation. Increasing the number of students increases the number of comparisons (more students will be compared to the individual for who recommendations are intended) and ultimately increases the number of similar students found by the first recommend operator, who in turn may contribute more candidate courses as input to the second operator. On average, the workflow scales very well as the number of students in the relation increases. We observed similar trends when varying the size of the course relation. For the sake of space, we do not report them here.

Depending on the courses one has rated, one may become connected with other students in multiple ways. The number of connections affects performance and explains the worst-case times observed in Figure 8(b). Figure 8(c) shows the effect of the number of connections to performance. We consider three cases in the whole student relation (10K). The best-case user has no connections to other students. The most-connected student connects to 2753 students through co-rated courses. On average, a student connects to 992 students in the database. This figure in combination with Figure 8(b) shows that it is not the input sizes that primarily affect scalability but the density of ratings. However, as in most cases, our site has relatively sparse data. Only around $1/5$ of the students have more than 900 connections.

Figure 8(d) shows how (average) times are shaped by the comparison function used. We compare the workflow execution times using the inverse Euclidean and Pearson for computing the similarity between students. We pick these two functions from our library, because they are quite representative in terms of computational complexity and practical importance. In this way, we can get a sense of how "*flexible*" the system is when objects are compared in different ways. As in Figure 8(a), we ran the workflow for different subsets of the student relation and on every user in the respective subset and we take average times. The figure shows that workflows using different functions can be executed in reasonable times for different sizes of the student relation. We have also observed with the other workflows that using either of the two functions does not have a significant impact on performance. Overall, the results of the collaborative filtering workflows are good news regarding the flexibility of the system w.r.t. different types of object comparisons and its scalability w.r.t. the data processed.

Major recommendation. This workflow (Example 7, Sec. 3.6) recommends majors to a student based on the majors of similar students. In the evaluation, similarity between students is estimated using the Pearson correlation. We ran the workflow for different subsets of the student relation with 2.5K, 5K and 10K students and computed majors for all students in each subset. Figure 9 shows the average (left Y-axis) and the worst-case (right Y-axis) execution times of the workflow for the different sizes of the student relation. We observe that for the average student in our dataset, the workflow is very efficient (it takes less than 25msec in all cases). The worst-case times are still very good. Worst-cases are students that are well-connected. It is interesting to note that the major recommendation workflow, although very close to the collaborative filtering approach for courses, is more efficient: there are few majors but possibly many courses to recommend.

Related courses. This workflow (Example 1, Sec. 3.6) finds related courses for a given course. In the experiment, we used the Jaccard similarity function on the course title and description and a sample of 500 randomly selected courses. For each one of them, the workflow returned all related courses from the course relation. Figure 10(a) shows the average execution times (on the left Y-axis)

and the worst-case execution times (on the right Y-axis) for the workflow for different subsets of the course relation with 5K, 10K, 15K and 20K courses. We sliced the course relation so that each larger set is a superset of the smaller sets.

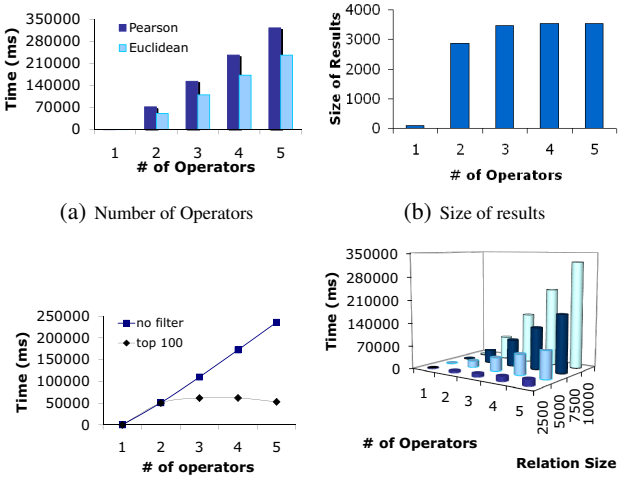
If we compare Figure 10(a) and Figure 8(b), we see that worst-times for the related-course workflow are higher than the collaborative filtering worst execution times. Comparing Figures 10(b) and 8(c), we observe that there are more course-to-course connections (based on common words in course title and description) than student-to-student (based on co-rated courses). The most-connected course (in the sample of 500 courses) has over 10000 connections to other courses while the most-connected student connects to 2753 students. On average, a course is connected to 5756 courses and the least connected course connects to just 7 courses. In the whole database, over half of the courses have over 5000 connections. The above observations lead to an important conclusion: despite the density of course-to-course, the system performance is still very reasonable even in the worst-cases.

Friends-of-friends. Recommend operators can be combined in sequences or trees of interconnected operators. We study the effect of a growing sequence of recommend operators to evaluate the system performance for recommendation workflows that are more complex than content-based and collaborative filtering ones.

We start with a workflow that contains one recommend operator that computes the set of similar students to an individual. We call these students "friends" of the student. Then, we consider a workflow of two recommend operators: the second one takes as input the "friends" found by the first recommend operator and computes "friends" of them. In the same way, we take workflows of 3, 4 and 5 recommend operators in a row, the output of one being the input to the other, for computing "friends" of "friends". As the individual who is input to the first operator, we use the most connected student, i.e., a student that through his ratings is connected to the greatest number of students. Hence, we will study the worst-case execution times of friends-of-friends workflows.

Figure 11(a) shows execution times using different functions (but the same within any single workflow) for different workflow sizes. All comparisons are performed against the whole student relation (10K size). We explain these results with the help of Figure 11(b), which shows the size of the workflow output for different workflow sizes. Essentially, this figure shows the output of each of the operators. Execution times for up to 3 operators reflect the impact of two factors: the increasing number of operators and the increasing number of tuples processed. The first operator performs few comparisons (since comparing to an individual student) and finds a small number of friends (100). The second recommend finds friends for these 100 people, so the additional time that it needs is reasonable. But it finds many friends (over 2800), which are fed into the third recommend and that explains the gradual increase in execution times observed for three operators in a row. The execution time of the third operator dominates the first two. However, Figure 11(b) shows that the size of the output of the third and of subsequent operators reaches an upper bound. This was expected since the size of the output could be at most equal to the size of the students relation, and in our case it is smaller, since not all students are transitively "friends" with each other. Hence, Figure 11(b) reveals that when having more than 3 operators, we only experience in performance the additive effect of having more operators.

Of course, we are using a worst-case scenario solely to stress test our system. In practice, we may have filters that reduce the size of the output of a recommend operator, i.e., by selecting the top k most similar students (Figure 11(c)), or its inputs, e.g. by selecting only a subset of students to be compared (Figure 11(d)). To illustrate,



(a) Number of Operators (b) Size of results

(c) Filtering Operator Output (d) Reducing Input Size

Figure 11: Performance Results: friends-of-friends workflow

Figure 11(c) shows what happens when we select only the top 100 similar students at the output of each recommend operator. Figure 11(d) shows that if we scale down the size of the input relations, we can scale system performance up to workflows of many operators. One known method to achieve that is to pre-cluster users so that a user is compared only to users within the same cluster.

In summary, our experiments show that with FlexRecs it is easy to create multiple workflows and execute them transparently over the same flexible recommendation system that combines extensibility with reasonable performance.

6. CONCLUSIONS AND FUTURE WORK

In this paper, we have argued that we need to decouple the definition of a recommendation process from its execution in order to support flexible recommendations over structured data. We have presented FlexRecs, a framework for declaratively defining recommendations combining traditional relational operators and special recommendation operators. Using FlexRecs, designers can easily capture multiple recommendation paradigms and experiment with novel recommendation strategies and users can dynamically personalize recommendation to fit their needs. We have provided several example workflows that capture recommendation approaches found in literature as well as novel paradigms from our live course planning site. As an application of declarativeness principle, we have presented an extensible prototype system that realizes the proposed framework over a conventional RDBMS and we have discussed experimental results that show its potential and flexibility.

A nice feature of our framework is that it makes possible to study the optimization of multiple recommendation workflows. Such optimization is analogous to what a query engine does in a traditional database system, except that now we have to handle new operators. We are currently working on scaling them over very large inputs. The framework opens the door to researchers to try different implementations and optimizations for the introduced operators and for recommendations. For example, the implementation of the recommend operator depends on comparison functions and there are many candidates, some more effective and other more efficient. The system could automatically balance complexity and effectiveness and identify the best option. One could define more restricted semantics for the operators in order to make them more efficient, for example for building execution plans that re-order the operators

defined in a workflow for better performance. One may go further and extend the database query engine with the processing capabilities required for supporting recommendations.

In addition, FlexRecs define recommendations over relational data. Large amounts of data are relational. Our university data are also relational. It would be interesting to define flexible recommendations for XML or ontologies. For example, XML is a more flexible data model for handling nested relations but there are challenges in defining and implementing the recommendation operators. Finally, our FlexRecs system serves as an extensible research and development platform for novel recommendation workflows. We are experimenting with different recommendation types, such as recommending majors or blending methods. Part of this effort is designing appropriate user interfaces for enabling users express flexible recommendations.

7. REFERENCES

[1] Problems with recommendations: url: http://www.amazon.com/problemswith-kindle-store-recommendations/forum/fxbvkst06pwp9b/tx1togp7gq4xa4t/1?_encoding=utf8&asin=b000fi73ma.

[2] The Stanford Daily: url: http://stanforddaily.com/article/2007/12/5/-editorialcourserankalongoverduesuccess.

[3] G. Adomavicius and Y. Kwon. New recommendation techniques for multi-criteria rating systems. *IEEE Intelligent Systems*, 22(3), 2007.

[4] G. Adomavicius and A. Tuzhilin. Multidimensional recommender systems: A data warehousing approach. In *WELCOM*, 2001.

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

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

[7] D. Billsus and M. Pazzani. User modeling for adaptive news access. *User Modeling and User-Adapted Interaction*, 10(2-3):147–180, 2000.

[8] J. Breese, D. Heckerman, and C. Kadie. Empirical analysis of predictive algorithms for collaborative filtering. In *14th UAI Conf.*, 1998.

[9] L. Colby. A recursive algebra for nested relations. *Inf. Syst.*, 15(5):567–582, 1990.

[10] A. Das, M. Datar, A. Garg, and S. Rajaram. Google news personalization: scalable online collaborative filtering. In *WWW*, pages 271–280, 2007.

[11] V. Deshpande and P. Larson. Transforming from flat algebra to nested algebra. *System Sciences*, 2(2-5):298–307, 1990.

[12] D. Goldberg, D. Nichols, B. Oki, and D. Terry. Using collaborative filtering to weave an information tapestry. *C. of ACM*, 35(12):61–70, 1992.

[13] J. Herlocker, J. Konstan, L. Terveen, and J. Riedl. Evaluating collaborative filtering recommender systems. *TOIS*, 22:5–53, 2004.

[14] G. Karypis. Evaluation of item-based top-n recommendation algorithms. In *CIKM*, 2001.

[15] G. Koutrika, R. Ikeda, B. Bercovitz, and H. Garcia-Molina. Flexible recommendations over rich data. In *RecSys*, pages 203–210, 2008.

[16] G. Koutrika, B. Bercovitz, R. Ikeda, F. Kaliszan, H. Liou, and H. Garcia-Molina. Flexible Recommendations for Course Planning. In *ICDE*, 2009.

[17] G. Linden, B. Smith, and J. York. Amazon.com recommendations: Item-to-item collaborative filtering. *IEEE Internet Computing*, Jan/Feb 2003.

[18] M. Mahony, N. Hurley, N. Kushmerick, and G. Silvestre. Collaborative recommendation: A robustness analysis. *ACM TOIT*, 4(4):344–377, 2004.

[19] B. Miller, I. Albert, S. Lam, J. Konstan, and J. Riedl. Movielens unplugged: Experiences with an occasionally connected recommender system. In *Int'l Conf. Intelligent User Interfaces*, 2003.

[20] B. Mobasher, R. Burke, R. Bhaumik, and C. Williams. Towards trustworthy recommender systems: An analysis of attack models and algorithm robustness. *ACM Trans. on Internet Technology*, 7(2), 2007.

[21] M. Pazzani and D. Billsus. Learning and revising user profiles: The identification of interesting web sites. *Machine Learning*, 27:313–331, 1997.

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

[23] B. Sarwar, G. Karypis, J. Konstan, and J. Riedl. Item-based collaborative filtering recommendation algorithms. In 10^{th} *WWW Conf.*, 2001.

[24] A. Schein, A. Popescul, L. Ungar, and D. Pennock. Methods and metrics for cold-start recommendations. In *ACM SIGIR Conf.*, 2002.

[25] H. J. Schek and M. H. Scholl. The relational model with relation-valued attributes. *Inf. Syst.*, 11(2):137–147, 1986.

[26] U. Shardanand and P. Maes. Social information filtering: Algorithms for automating 'word of mouth'. In *Conf. on Human Factors in Comp. Sys.*, 1995.

[27] B. Sheth and P. Maes. Evolving agents for personalized information filtering. In *IEEE Conf. Artificial Intelligence for Applications*, 1993.