Flexible Recommendations for Course Planning

Georgia Koutrika [1], Benjamin Bercovitz [2], Robert Ikeda [3], Filip Kaliszan [4], Henry Liou [5], Hector Garcia-Molina [6]

Computer Science Department, Stanford University
USA
{koutrika[1], berco[2], kaliszan[4], liouh[5]}@stanford.edu
{rmikeda[3], hector[6]}@cs.stanford.edu

***Abstract*— Most recommendation methods are 'hard-wired' into the system and they support only fixed recommendations. The purpose of this demo is to show the expressivity of flexible recommendation workflows, how flexible recommendations can be efficiently processed over relational data, and to show flexible recommendations in action through a real system used for course planning.**

I. Introduction

In our demo, we will show the flexible recommendation component of *CourseRank* [1] that removes important limitations of most recommendation systems. *CourseRank* is a social tool for course planning we have developed in Infolab at Stanford. The flexible recommendation framework (*FlexRecs*) and engine are described in detail in a companion ICDE'09 submission [7]. In this demo paper:

- We summarize the *FlexRecs* framework.
- We describe the architecture of the system for defining and processing flexible recommendation requests.
- We describe an application of flexible recommendations for course planning in the context of *CourseRank*.

II. Motivation and Outline

Since the appearance of the first recommendation systems [5], [10], a lot of work has been done from improving and evaluating recommendation methods [3], [6] to designing trustworthy systems [9]. Recommendation systems providing advice on movies, products, and other topics, have become very popular in systems, such as Google News [4] and Amazon [8]. However, most 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.

Limited world model: Traditionally, 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 for *books* from users *with similar ratings*.

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. Different recommendations may be expected by different users.

FlexRecs remove these limitations allowing *flexible recommendations to be easily defined, customized, and processed*. A given recommendation approach can be expressed *declaratively* as a high-level workflow. The workflow may contain *traditional relational operators* such as select, project and join, *plus new recommendation operators* that generate or combine recommendations. All the data (including recommendations) is *relational* in nature. The recommendation operators may call upon functions in a library that implement common tasks for recommendations, such as computing the Jaccard or Pearson similarity of two sets of objects.

A designer can easily express multiple workflows for both content-based and collaborative recommendations, as well as new types that the designer may think of. 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. The system executes a workflow by "compiling" it into a sequence of SQL calls, which are executed by a conventional DBMS. When possible, library functions are compiled into the SQL statements themselves; in other cases we can rely on external functions that are called by the SQL statements.

This functionality is essentially similar to advanced searches: a designer builds a set of parameterized SQL queries and the interface allows users to transparently execute them with parameter values and receive different results.

III. Flexible Recommendations (*FlexRecs*)

We review the main concepts of our *FlexRecs* work (formally described in [7]) with an example using three relations:

Courses(<u>CourseID</u>, DepID, Title, Description, Units, Url)
Students(<u>SuID</u>, Name, Class, GPA)
Comments(<u>SuID</u>, <u>CourseID</u>, <u>Year</u>, <u>Term</u>, Text, Rating, Date)

Assume we want to compute course recommendations for a student with id 444 based on the ratings of similar students. Two students are similar if their course ratings are similar. In order to "build" recommendations, we have a library of comparison functions (e.g., to compare course ratings, course topics, student names, etc), such as Pearson's, and Jaccard

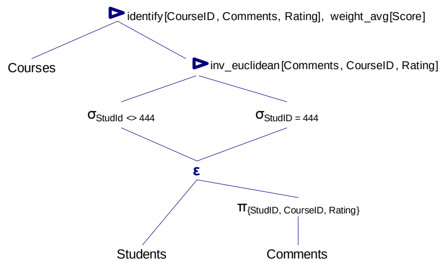

⊳ identify[CourseID, Comments, Rating], weight_avg[Score]
Courses
⊳ inv_euclidean[Comments, CourseID, Rating]
σ StudId <> 444
σ StudID = 444
ε
π {StudID, CourseID, Rating}
Students
Comments

Fig. 1. Recommendation expression tree.

index, and for aggregations, such as average and weighted average. For our example, we will compute similarity between two students taking the inverse Euclidean distance of their ratings, and ratings for courses taking the weighted average of the ratings by the similar students.

We can express the desired recommendations using the workflow represented by the expression tree shown in Figure 1. We observe that the tree is composed of more traditional select, project, join operators, and it also contains some new operators that we analyze below.

Although most structured data used by production systems (including *CourseRank*) is stored in relational databases, one problem with a "pure" relational data model is that information that "conceptually" refers to a single entity (e.g., course ratings of a student) may be found in different relations due to database structuring and normalization. In our example, in order to compare students based on their ratings, we would have to join together two relations, since student ratings are stored separately from the rest of information regarding students, making the whole process cumbersome.

Ideally, we would like to represent our application entities with a single relation. For instance, a tuple in a such relation could contain base information on a student (e.g., name), plus the courses a student has taken. For this purpose, we use an *extend operator* (ε) that generates a virtual 2-level nested relation. This operator allows "extending" each tuple from one relation with the set of joining tuples from a different relation. In our example workflow, students are extended with their course ratings, so that the set of ratings for each student can be "viewed" as another attribute of the student by subsequent operators in the workflow irrespective of the database schema. In a sense, *extended relations* can be thought of as "views" that group together information related to an individual entity and represent it as a single tuple that can be easily handled by other operators.

We observe that recommendations are based on comparisons (e.g., courses are rated against student ratings, students are compared to a particular student based on their ratings in order to find similar students, and so forth). For this purpose, we use the *recommend operator* ($\triangleright_{cf}$), which rates the tuples of a set by comparing them to the tuples of another set using a comparison function cf. Our example workflow has two recommend operators. The lower one finds similar students to the student with id 444 using the inverse Euclidean distance of their ratings. The upper one finds courses recommended by these students taking a weighted average of their ratings.

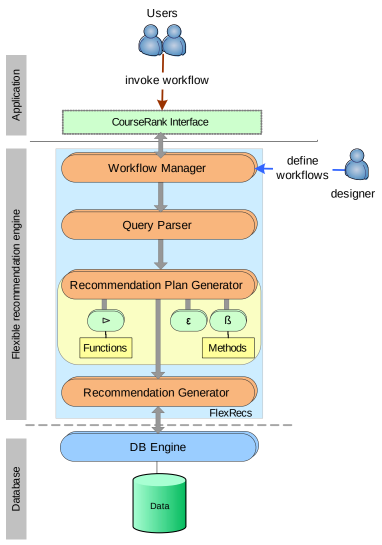


Fig. 2. FlexRecs System Architecture

In addition, we may want to combine recommendations generated through two different processing paths into one. For example, we may want to combine the course recommendations generated using our example workflow with courses that are required for graduating, and provide one recommendation. For this purpose, we use the *blend operator* (β_M). Different blending methods are part of the system library. The recommend and blend operators capture the essence of most recommendation workflows. They can be combined with more traditional select, project, join operators, and workflows can be executed leveraging the underlying DBMS.

IV. Flexible Recommendation Engine

We have built a flexible recommendation engine in Java on top of MySQL (shown in Figure 2) that compiles and executes *FlexRecs* workflows. Our strategy 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 *Workflow Manager* allows a designer to define different recommendation workflows and end-users to invoke any of the defined workflows with different inputs and receive customized recommendations. It 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.

The *Query Parser* takes as input a recommendation workflow and constructs an expression tree, like the one shown in Figure 1, keeping the order of the operators as defined in the query, since no real processing is performed at this level.

The *Recommendation Plan Generator* takes as input an expression tree and generates a recommendation execution

Query 1:

```
CREATE TEMPORARY TABLE temp
SELECT t1.SuID, 1/SQRT(SUM((t1.Rating − t2.Rating) ∗ (t1.Rating − t2.Rating))) as score
FROM Comments t1, Comments t2
WHERE t1.CourseID = t2.CourseID AND t2.SuID = 444 AND t1.SuID <> 444
GROUP BY t1.SuID
```

Query 2:

```
CREATE TEMPORARY TABLE
temp2
SELECT t1.*, score
FROM Comments t1, temp
WHERE t1.SuID = temp.SuID;
```

Query 3:

```
SELECT Courses.*, SUM(score*rating)/SUM(score) AS CScore
FROM temp2, Courses
WHERE temp2.CourseID=Courses.CourseID
GROUP BY CourseID
ORDER BY CScore
```

Fig. 3. An example recommendation plan

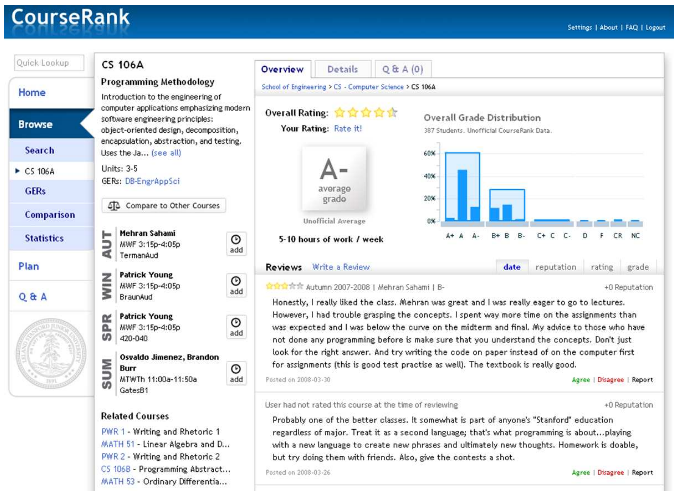
plan. The generated plan comprises a sequence of SQL statements and function calls and specifies how and in what order operators are executed and any results that are shared in a particular workflow and need to be materialized. This module leverages the underlying DB engine's query optimizer for implementing 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 themselves; in other cases we rely on external functions that are called by the SQL statements. In this way, the system can support new types of recommendations by extending its library.

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.

We sketch the recommendation plan generation process and the implementation of the new operators using our example workflow shown in Figure 1.

For each recommend operator in the workflow, the system builds one (or more) queries that implement the requested comparison function and group together any operators that are found in the subtree under this operator. (A similar strategy is used for blend operators.) We support an extensible library of comparison functions. Interestingly, several functions, such as Pearson, Cosine, Avg, can be implemented by computing and combining aggregations supported by the underlying database.

We will first see the queries for implementing the lower operator. The corresponding subtree contains select, project, and extend operators, which are all combined with the root recommend operator into Query 1 in Figure 3. This query has several parts (shown shaded), 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 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. The query does not join the relations Students and Comments specified in the expression tree, because 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 with two attributes for each student: the student id and a score. Following a similar philosophy, Queries 2 and 3



CS 106A

Programming Methodology

Introduction to the engineering of computer applications emphasizing modern software engineering principles: object-oriented design, decomposition, encapsulation, abstraction, and testing. Uses the Ja... (see all)

Units: 3-5
GERs: DB-EngrAppSci

AUT	Mehran Sahami MWF 3:15p-4:05p TermanAud
WIN	Patrick Young MWF 3:15p-4:05p BraunAud
SPR	Patrick Young MWF 3:15p-4:05p 420-040
SUM	Osvaldo Jimenez, Brandon Burr MTWTh 11:00a-11:50a GatesB1

Related Courses
PWR 1 - Writing and Rhetoric 1
MATH 51 - Linear Algebra and D...
PWR 2 - Writing and Rhetoric 2
CS 106B - Programming Abstract...
MATH 53 - Ordinary Differentia...

5-10 hours of work / week

Autumn 2007-2008 | Mehran Sahami | B- +0 Reputation

Honestly, I really liked the class. Mehran was great and I was really eager to go to lectures. However, I had trouble grasping the concepts. I spent way more time on the assignments than was expected and I was below the curve on the midterm and final. My advice to those who have not done any programming before is make sure that you understand the concepts. Don't just look for the right answer. And try writing the code on paper instead of on the computer first for assignments (this is good test practise as well). The textbook is really good.

Posted on 2008-03-30

User had not rated this course at the time of reviewing +0 Reputation

Probably one of the better classes. It somewhat is part of anyone's "Stanford" education regardless of major. Treat it as a second language; that's what programming is about...playing with a new language to create new phrases and ultimately new thoughts. Homework is doable, but try doing them with friends. Also, give the contests a shot.

Posted on 2008-03-26

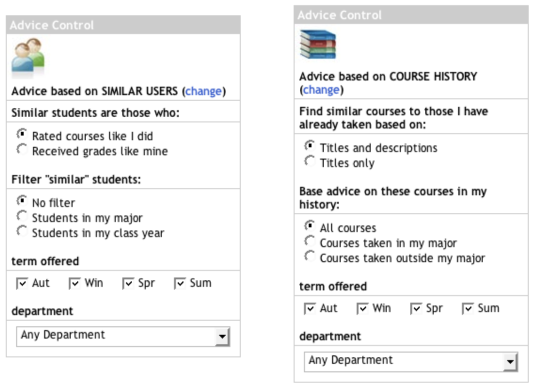
Fig. 4. Course page

correspond to the higher recommend operator and compute course recommendations based on the ratings provided by the similar users found by Query 1.

V. COURSERANK

Our work on *FlexRecs* was motivated by, and has been implemented as part of *CourseRank*. *CourseRank* has several features to help Stanford students 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. (Figure 4 shows a course page.) Students can get personalized recommendations, organize their classes into a quarterly schedule or devise a four year plan. *CourseRank* also functions as a feedback tool for faculty and administrators, ensuring that information is as accurate as possible. Faculty can also modify or add comments to their own courses, and can see how their class compares to other classes. A little over a year after its launch, it is already used by more than 6,200 Stanford students, out of a total of about 14,000 students. The vast majority of *CourseRank* users are undergraduates, and there are only about 7,000 undergraduates at Stanford. Thus, *CourseRank* is already used by a very large fraction of Stanford undergraduates.

Although the initial version of *CourseRank* has been very popular with students (see editorial in [2]), we received many requests, from students and administrators, for more flexible recommendations. For example, students want to specify the type of course they are interested in (e.g., a biology class, something that satisfies Stanford's writing requirement), to request recommendations from a peer group they select (e.g., students in their major with similar grades, or freshmen only), and so forth. With *FlexRecs*, it became possible to accommodate these requests and build a unified interface, which allows the user to search, tune recommendations and personalize the whole process. Users will only make choices from simple menus and sliders but they will actually be selecting what



Advice Control

Advice based on SIMILAR USERS (change)

Similar students are those who:

[✓ Rated courses like I did]
Received grades like mine

Filter "similar" students:

[✓ No filter]
Students in my major
Students in my class year

term offered

☑ Aut ☑ Win ☑ Spr ☑ Sum

department

Any Department

Advice Control

Advice based on COURSE HISTORY (change)

Find similar courses to those I have already taken based on:

[✓ Titles and descriptions]
Titles only

Base advice on these courses in my history:

[✓ All courses]
Courses taken in my major
Courses taken outside my major

term offered

☑ Aut ☑ Win ☑ Spr ☑ Sum

department

Any Department

(a) flexible collaborative recs (b) flexible content-based recs

Fig. 5. Advice Panel.

workflow to run, and with what parameters. The new version of *CourseRank* will be released September 2008.

VI. A Flexible User Interface

Designing interfaces for flexible recommendations is also challenging because flexible recommendations are a new concept, so it is necessary not to confuse users. A flexible recommendation interface should allow the user to select a recommendation workflow, to set the parameters, and then to see the recommendations generated. We have designed an *advice panel* that allows a single workflow to be active at a time, and given a particular workflow, the user can set the relevant parameters. We currently offer two recommendation workflows: a collaborative and a content-based workflow.

In the current version, to receive recommendations the user clicks a top-level (tab) link *Advice*. After clicking the link, the user is shown recommendations created with the collaborative recommendation workflow using some default parameter values. Along with the recommendations, the system presents the advice panel shown in Figure 5(a). In this panel, there is an icon of two people and explanatory text saying *Advice based on SIMILAR USERS*, indicating that the collaborative workflow is currently active. To switch between recommendation workflows, the user can simply click the *change* link. Now, the icon will display a stack of books (Figure 5(b)). The text under this icon will also change to read: *Advice based on COURSE HISTORY*.

The user can control recommendations through the advice panel, which has two sets of parameters: a set for "pure" recommendation operations and a set for filtering returned recommendations. *The first set* allows currently the user to specify for the collaborative workflow: (a) how similarity between two students should be calculated, i.e., using the users' course ratings or based on grades, and (b) how to filter the set of similar students. For example, the user may be interested in recommendations based on students in her major. In the content-based workflow, the advice panel allows the user to specify: (a) how similarity between courses should be calculated: using both the course titles and course descriptions or only course titles, and (b) the part of the user's course history that should be considered. For example, the user may want recommendations within her major and hence, specify that only the courses taken within the major should be examined. *The second set* of parameters allows the user to filter the returned recommendations by term offered and by department.

VII. Demonstration

Most recommendation methods are 'hard-wired' into the system and they support only fixed recommendations. The purpose of the demo is to show: (a) the expressivity and flexibility of the *FlexRecs* framework, (b) how flexible recommendations can be expressed and efficiently processed over relational data, (c) a real application of flexible recommendations for course planning, and (d) how flexible recommendations can be accessed using an advice panel.

Demo Scenario: We will explore flexible recommendations from two points of view, that of a user (a student) and that of an administrator that runs the site (*CourseRank*).

Using one student as an example, we will be able to navigate through *CourseRank* and explore its various course planning features and particularly examine course recommendations. The "student" will be able to search for courses. Changing the various options that the advice panel offers, the "student" will be able to control the recommendation process and experiment with different types of course recommendations. Furthermore, assuming different student identities inside the system will allow one to see how different students can tune and personalize their recommendations.

Assuming an administrator role, we will show how flexible recommendations offered by *CourseRank* are defined as high-level workflows and examine the recommendation plans that the underlying *FlexRecs* system generates for different user requests. We will also show how a new recommendation workflow can be defined in a declarative way combining different operators, comparison functions and blending methods from the ones that are provided by the system library.

References

[1] The CourseRank system: url: http://courserank.stanford.edu/.

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

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

[6] J. Herlocker, J. Konstan, L. Terveen, and J. Riedl. Evaluating collaborative filtering recommender systems. *TOIS*, 22:553, 2004.

[7] G. Koutrika, B. Bercovitz, R. Ikeda, F. Kaliszan, H. Liou, and H. Garcia-Molina. FlexRecs: Expressing and combining flexible recommendations. In *http://dbpubs.stanford.edu/pub/2008-19 (submitted to ICDE'09)*, 2008.

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

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

[10] 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.