Contributions to a Java Recommender System

Yun Wang

Major Professor: Jon Herlocker

Committee Member: Tim Budd

Ron Metoyer

ABSTRACT

Identifying the most relevant items in an e-commerce site is becoming more and more difficult nowadays because of the heavy overload of information. A Java Recommender System that uses Collaborative Filtering techniques has been developed to reduce such information overload and even personalize the information to the individual's preference. The conventional recommendation provided in the earlier systems is not capable to recommend items on a specific category that the user is interested in. Recent development of a new capability in the Java Recommender System has fixed this problem. This new capability concentrates on the individual's interests, and provides recommendations based on categories. A new CORBA API has been developed to facilitate distributed environments with different programming languages, different platforms. Finally, the correctness testing has been applied to ensure the stability of the whole system.

Acknowledgements

I would like to express my sincere appreciation to my adviser, Dr. Jon Herlocker, for his guidance and encouragement. I would like to thank Dr. Tim Budd and Dr. Ron Metoyer for serving as my committee member. I would also like to acknowledge all my team members for their helpings, especially to Danial Lowd and Olivia Godd. I'm grateful to my husband for his supporting. At last, I would like to thank my family, my friends and all the people that helped me during my study.

Table of Contents

List of Figures

List of Tables

Contributions to a Java Recommender System

1 Introduction

The internet is playing an increasingly important role in peoples' lives. We read news, search for information, and even buy products via the web. However, there are so many pieces of information out in the web, what is relevant to us? Information overload has become a serious problem encountered by internet users.

To overcome this problem, several filtering techniques emerge. The Google search engine (www.google.com) succeeded in PageRank -- a content-based filtering technology, which selects documents based on the text and the links in them. However, it concentrates on the documents rather than the user's preference and the classification of the information. Collaborative filtering systems can help solve this problem.

We see Collaborative Filtering in action every day in our lives. When we want to know whether a book is worth reading, we ask for our friends' opinions whether they like the book. We ask our friends to recommend new restaurants to us when we feel adventurous and want to try something different. Collaborative Filtering is rapidly becoming an important tool and achieving widespread success on E-Commerce sites, such as www.amazon.com. It helps to target advertising, bring like-minded people together, push information selectively within a business, and direct customer service queries. Various collaborative filtering systems benefit customers by filtering out the irrelevant information on the internet and enabling them to find products they like. Conversely, they help businesses by generating more sales.

Most research about Collaborative Filtering and Recommender Systems has attempted to improve the quality of the recommendations for the users, and a variety of successful algorithms have been developed. The quality of recommendation, of course, is very

important to a recommender system. But in practice, the scalability and compatibility are also critical. We explain these three factors as follows:

Quality: System provides predictions and recommendations that accurately reflect the users' tastes.
Scalability: As the number of users grows, the quality of predictions should improve and the speed should not deteriorate.
Compatibility: The system can be easily deployed with existing content.

Our Java Recommender System addresses the above challenges. Its goal is to provide a robust and scalable recommender engine that provides high quality recommendations in a variety of environments. This includes compatibility with a multiple programming languages and applicability in both commercial and research domains. The system can be applied to any content domain and also can be a research tool that easily integrates new algorithms.

In this report, we present some improvements made on this system, which includes adding a new capability (recommendation-by-category) into our recommender system, building a new API (Application Program Interface) and testing the correctness of the whole system.

The report is structured into six sections. After a brief introduction, we give an overview of the recent Collaborative Filtering techniques and our Java Recommender System. Secondly, we will describe the new capability, recommendation-by-category, which gives the recommendation on the category that interests the user. Some experimental results will also be presented. Then, we propose two designs about CORBA API and compare its performance with Java RMI. In section 5, we present the discussion of the correctness testing of the whole system and its results. Finally, we will end with some conclusions.

2 Overview of Collaborative Filtering and Java Recommender System

The technique of Collaborative Filtering has been widely used on the web, and its goal is to provide the user opinions based on the other people's preferences. Our Java Recommender System is a tool that uses this technique for personalization in applications. We will explain this technique and overview the system in this section.

2.1 Overview of Collaborative Filtering

Collaborative Filtering, also called recommender systems, can produce personal recommendations by computing the similarity between a person's preferences and the preferences of other people. It aims to reduce the information overload, and to help the users focus attention on what they really want. The main idea is to automate the process of "word-of-mouth" by which people recommend products or services to one another. If someone needs to choose between various options with which he or she does not have any experience, he/she will often rely on the opinions of others who do have such experience. However, when there are thousands or millions of options, like on the web, it becomes practically impossible for an individual to locate reliable experts that can give advice about each of the options. By shifting from an individual to a collective method of recommendation, the problem becomes more manageable.

There are many Collaborative Filtering systems that use various techniques. One of the basic mechanisms behind them is the following:

- A large group of people's preferences are recorded;
- Using a similarity metric, a subgroup of people is selected, whose preferences are similar to the preferences of the person who seeks advice;
- A weighted average of the preferences for that subgroup is calculated;
- A resulting preference function that generates predictions or recommendations, is used to recommend options on which the advice-seeker has expressed no personal opinion as yet.

2.1.1 Collaborative Filtering Algorithms

The algorithms in collaborative filtering predict the utility of the items to an active user according to user votes/ratings of the other users. Algorithms can be classified into two categories: Memory-based and Model-based. Memory-based algorithms operate over the user database (part of database with sampling or entire database without sampling) to make predictions, while Model-based algorithms use the database to estimate or learn a model, then produce predictions based on that particular model. [3]

Memory-Based Algorithms

The characteristic of an algorithm in this class is that it keeps ratings in memory, and uses those ratings to generate each prediction.

Most nearest neighbor correlation based algorithms belong to this category. For instance, like the user-user algorithm, item-item algorithm, horting algorithm and etc. They use all the available data to compute similarity between two users or two items. The advantage of the algorithms in this category is that the computation reflects the changing of the data (ratings changed or added by the user) immediately, but can take fairly large amounts of computations in generating a single prediction, as the number of ratings in memory grows.

Model-Based Algorithms

Model-based algorithms compute a structured model from the entire data set. Once the model is built, they store the model instead of the entire data and calculate the expected value of an explicit value (rating) based on that model.

The famous model-based algorithms include Cluster model, Bayesian network, SVD and etc. It can take a long time to build the model, so the model is usually computed offline.

The model must be re-computed periodically as new ratings are added into the ratings database. Therefore, model-based algorithms do not reflect the newly added data on the predictions immediately.

2.2 Java Recommender System Overview

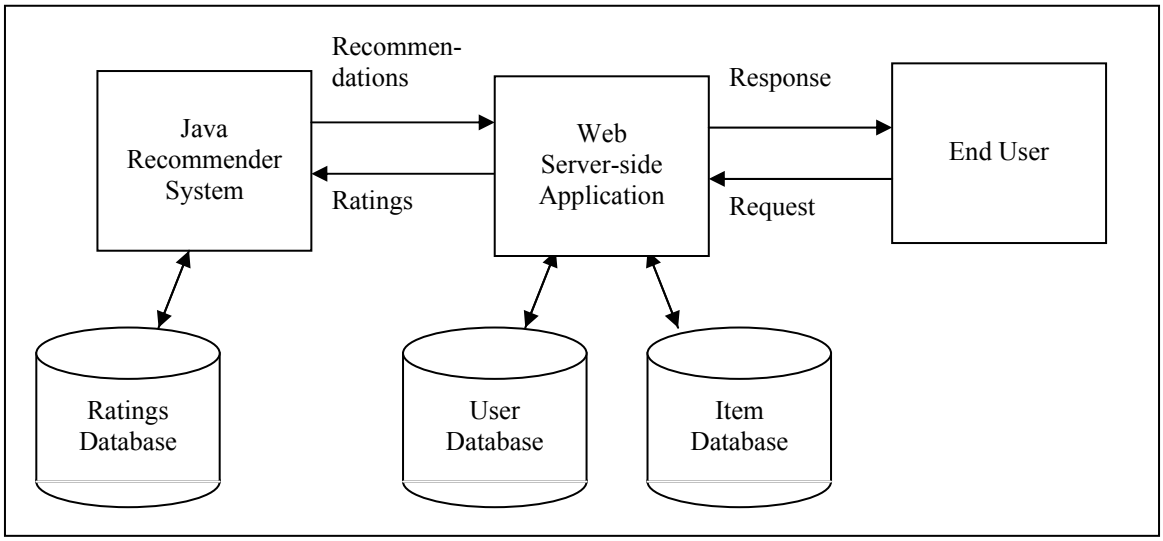


Figure 1: An example Usage of Java Recommender System

The Java Recommender System is a tool for personalizing user interests and reducing information overload on the Internet. An example usage is illustrated by Figure 1. The web server-side application is using the CF Recommender System to predict what items of specific content should be displayed to which users. After the web server-side application locates the user's record in the user database and receives the request from that user, it sends this request along with the I.D. of that user to the Recommender System. After computation, the Recommender System responds to the web server with a ranked list of recommending items. The server-side application then accesses the content associated with the items and displays them to the user [12].

2.2.1 Functionality of the Java Recommender System

As depicted in Figure 1, the Java Recommender System manages all the ratings, and it is also responsible for all the facilities included in Collaborative Filtering. It provides the following functions to the web-server application via the interface:

- It allows the user to access the ratings database, which includes *retrieve*, *add* and *remove*.
- The user can get *prediction* of a specific item, or a list of *recommendations*. These are the core methods of Collaborative Filtering. The server of the Java Recommender System manages all the computations.
- The system gives out the unique user IDs and item IDs. Some peripheral functions, like testing the server, shutting the server down, are also provided.

2.2.2 Architecture of the Java Recommender System

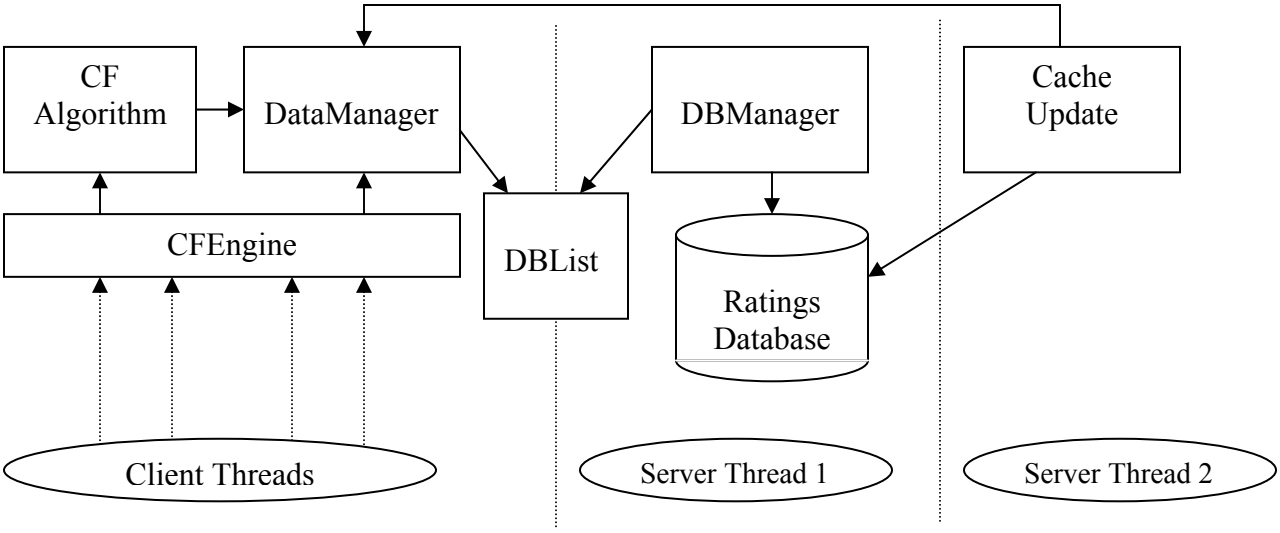


Figure 2: Architecture of Java Recommendation System

The JRS (Java Recommender System) is Client-Server structured, where the client and the server talk via a single remote interface – *CFEngine*. Clients are typically threads from the web-server side applications. There are two main modules in the JRS: *CFAlgorithm* manages all the core computation; and *DataManager* is in charge of all the data. The JRS Architecture is depicted in Figure 2.

The server side main thread handles the requests from the client. There are three server-side threads in addition to the main thread: one is for database updates; the second is for updating cache; and the third is for re-sampling that is periodically activated and not presented in Figure2. The last two threads are needed because the JRS supports computation on a smaller sampled data set.

The server is designed to deal with the requests from multi clients. Thus, concurrency issues exist and introduce difficulties in the testing and debugging, which will be discussed in the later section.

3 A New Capability -- Recommendations-By-Category

The Java Recommender System gives *prediction*s and *recommendations*, which are the conventional functions in Collaborative Filtering. What if the users do not want recommendations on all kinds of items? For example with movies, a user just wants to get recommendations on a subset, like comedy, romance, etc. She/he is just simply not interested in other classes. But the conventional *recommendations* capability only recommends movies across all items, mixing the items from all the categories. In this kind of situation, recommendation-by-category becomes an important feature that has to be provided in the system.

In this section, we first explain the requirements for the implementation. Secondly, we will introduce 3 algorithms where this new feature has been implemented. Finally, the implementation and the experimental results will be stated.

3.1 Requirements for Implementation

First of all, recommendation-by-category, as a capability of the collaborative filtering, needs to provide high quality recommendations to the users. The result of this capability must be accurate. At least, the recommendations generated by this *recommendation-by-category* capability are not worse than those generated by the conventional *recommendations*.

Speed is the second consideration. Our Java Recommender system promises to be robust and scalable. We don't want the *recommendation-by-category* capability to respond much slower than the conventional *recommendations* capability does and degrade the system. A good response time of this new capability is critical to the system.

Consistency is the final requirement. The rankings in the *recommendation-by-category* have to be consistent with the ones in the *recommendations*. The users will not be happy

to see the rankings differ between the conventional *recommendations* and the *recommendation-by-category*.

3.2 Algorithms

First of all, we will talk about the algorithms used for this new capability. We have implemented the *recommendation-by-category* in three algorithms. In order to explain how this new capability be implemented in the algorithms, we first have to explain the basic algorithms in generating *recommendations*. Then, describe the algorithms in generating *recommendation-by-category*. Finally, we explain why it can't be done by the existing system.

3.2.1 Basic Algorithms for *Recommendations*

In order to generate recommendations for an active user, we compute the predictions for every item that hasn't been rated by the user. Then, we will sort the predicted ratings, considering the item with the highest prediction as ranking top 1. Finally, the top part of sorted list (from top 1 ranking to top *N*) will be stored in the server side cache, where *N* is a predetermined threshold. That is to say, the user can only get *recommendations* in the range of $1^{st} \sim N^{th}$, and *N* is a configurable parameter.

During the generation of the recommendations, the important step is computing the predictions. In this section, we will introduce two memory-based algorithms in generating predictions:

- The Simple Pearson algorithm that is the most popular algorithm.
- The Item-Item algorithm that explores the relationships between items instead of users.

3.2.1.1 Simple Pearson Algorithm

To generate a prediction, the algorithm first computes the Pearson correlation coefficient of two users -- active user *a*, and one of the other users *u* -- on all the items rated by both. This process has to be repeated on all neighbors. The formula is given as follows:

$$w_{a,u} = \frac{\sum_{i=1}^{m} (r_{a,i} - \overline{r_a}) * (r_{u,i} - \overline{r_u})}{\sigma_a * \sigma_u} \qquad (1)$$

where $r_{u,i}$ stands for the rating of neighbor *u* on item *i*, $\overline{r_a}$ and $\overline{r_u}$ are the mean rating of active user *a* and neighbor *u*. $\sigma_a * \sigma_u$ is the product of standard deviations of the active user *a* and neighbor *u*, and *m* is the number of items rated by both *a* and *u*. $w_{a,u}$ is in the range of -1 ~ 1, which can reflect distance, correlation, or similarity between user *u* and the active user *a*. -1 means neighbor *u* is in perfect disagreement with *a*, and 1 stands for perfect agreement. 0 means there is no linear relationship between *u*'s ratings and *a*'s ratings. The best neighbors are the highest positive weights with the active user *a*. We treat negative correlations as 0, as negative correlations have not been shown to improve accuracy.

A final prediction is computed by taking a weighted average of all the ratings that the neighbors rated on the given item *i* according to the following scheme:

$$p_{a,i} = \overline{r_a} + \frac{\sum_{u=1}^{n} (r_{u,i} - \overline{r_u}) * w_{a,u}}{\sum_{u=1}^{n} w_{a,u}} \qquad (2)$$

where *n* is the number of users with nonzero weights. Therefore, the neighbors who have the highest weights will most strongly influence the predicted rating. [2]

In practice, there are several important parameters affecting the predicted ratings. First of all, the number of common items – *m* in formula (1) is critical while computing the weights. Setting a minimum threshold can improve accuracy. Secondly, who should be

considered as neighbors? Setting a minimum threshold value of the weights can also affect accuracy. Finally, while computing the prediction, the threshold for the minimum number of users – *n*, also plays a significant role.

3.2.1.2 Item-Item Algorithm

Conventional nearest neighbor collaborative filtering algorithms have focused on the user behavior and computed predictions based on the relationship between users. However, the Item-Item algorithm looks into the set of items that the active user has rated and computes how close they are to the target item *i* then selects the ratings of the *k* most similar items to use in computing a prediction. [5]

Several ways have been proposed to compute the similarity between items: cosine-based, correlation-based and adjusted-cosine similarity [9]. Adjusted-cosine will be used in our implementation, which is given by

$$s_{i,j} = \frac{\sum_{u \in U} (r_{u,i} - \overline{r_u}) * (r_{u,j} - \overline{r_u})}{\sqrt{\sum_{u \in U} (r_{u,i} - \overline{r_u})^2} \sqrt{\sum_{u \in U} (r_{u,j} - \overline{r_u})^2}} \quad (3)$$

In this case, *U* is the set of users that rated both *i* and *j*. The mean of the user *u*'s ratings is shown by $\overline{r_u}$. Once we obtain the set of the most similar items, we will use the formula (4) to get the predictions. This is the weighted average over all *n* similar items rated by user *u*.

$$p_{u,i} = \overline{r_i} + \frac{\sum_{j=1}^{n} (r_{u,j} - \overline{r_j}) * s_{i,j}}{\sum_{j=1}^{n} s_{i,j}} \quad (4)$$

The same as the Simple Pearson algorithm, this algorithm also has some important parameters affecting the predictions: the minimum size of *U* in computing similarity – in

formula (3); the threshold of the minimum allowed value of similarity; and the minimum size of *n* – the number of similar items in computing prediction – in the formula (4).

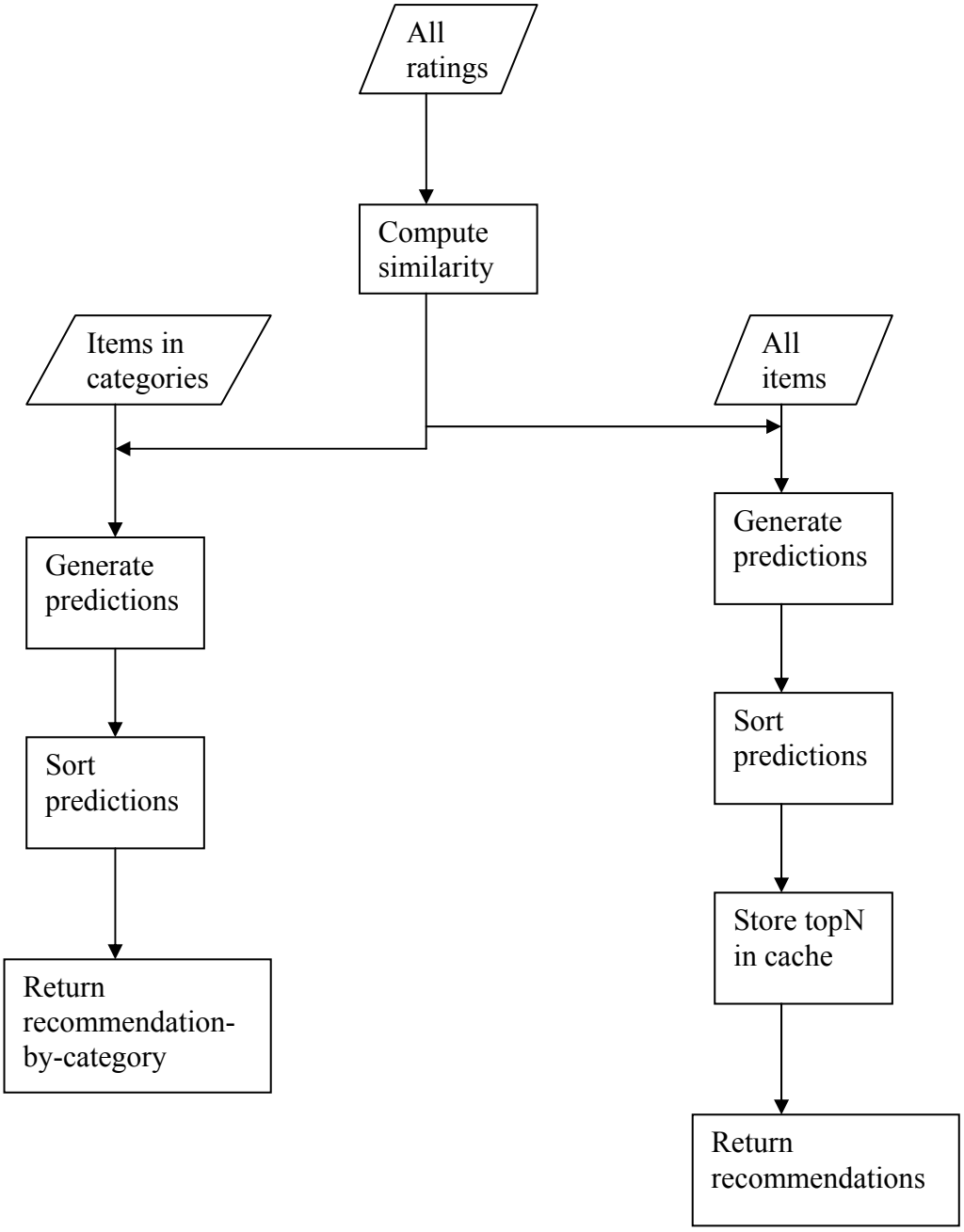


Figure 3: Algorithms for *Recommendation-by-category* and *Recommendations*

3.2.2 Algorithms for Recommendation-By-Category

We use the basic ideas stated above to develop the algorithms for our new capability — *recommendation-by-category*. In order to distinguish between *recommendation-by-category* and conventional *recommendations*, we call the latter one the *recommendation-all-categories* here. The *recommendation-by-category* allows an active user *a* to require the recommendations to be limited to a category *Y*.

The first step is to get a list of items that belong to the category *Y*. Secondly, we generate the predictions for all the items in that list. The final step is to sort the items according to their predicted rating and return the sorted result. The algorithm is described by Figure 3.

The *recommendation-by-category* uses the same method as the *recommendation-all-categories* to generate the correlations (similarity weights). However, the *recommendation-by-category* generates predictions only for the items in a specific category. Another difference between these two capabilities is that the information about the category is necessary in computing the *recommendation-by-category*.

3.2.3 Reason to develop New Capability

The *recommendation-by-category* could be done by first performing the *recommendation-all-categories*, then filtering by category. But just filtering out the *recommendation-all-categories* by category is not feasible, either from the results stored in cache or from the direct computing results. We only store part of the results (top1 - top *N*) generated by the *recommendation-all-categories* in the cache, where *N* is a predetermined number, and it is possible that there are no recommendations for a specific category within the cache. Items in the categories that are popular and interested by the user get high predictions and could occupy top 1 - top *N* rankings, and the items in the unpopular categories in the lower rankings, which are not kept in the cache.

Furthermore, generating the *recommendation-all-categories*, then filtering out the ones that do not belong to a specific category is time-consuming. Compared to eliminating the items during the computation of recommendations, it requires much more computation time. Therefore, post-filtering the results of the *recommendation-all-categories* by category will not work well in the current system.

3.3 Implementation

First, we needed to add functionality for storing category information in the database and in memory. We modified the server to manage computation of the *recommendation-by-category* and provide the results to the client. During the implementation, we added one table in the database, developed many methods in the existing recommender system modules, and added some structures in the memory cache.

3.3.1 Addition to the database and the existing system

In order to store the information of the categories, we added one table for the server database. The server side table has only two columns: one for item ID and the other for category ID, which describes the relations between items and categories. The server uses this table to support all the computation.

The server's architecture has been stated previously as in Figure 2. There are many methods added into the existing server, which can be classified into the following three categories:

- *Providing the results to the client*: In the *CFEngine* interface and its implementation class, a method that provides the results of the *recommendation-by-category* was added.
- *Managing the data*: We added some methods in module *DataManager*. They manage all the data about the categories, which retrieve the categories'

information from the database and pass the computation results to the *CFEngine* interface.

- *Managing the computation*: A method, *getRecommendationsByType()*, has been added into *CFAlgorithm* interface, and it was further implemented in three algorithms: Simple Pearson, Item-Item, and Horting. This method uses the algorithms stated in the previous section to compute the recommendation-by-category.

3.3.2 Structure added in the cache

Our current system caches the ratings in the main memory to avoid disk I/Os during the computation. The *recommendation-by-category* uses this caching structure in its computation, which gives good performance in calculating the similarity weights, selecting the neighbors, and computing predicted ratings. But we currently do not cache the items in each category.

If we don't cache the categories, the items of a specific category have to be queried from the database every time, which takes huge amounts of time. Caching the categories into the memory at the time the server is booting up is a good way to reduce the cost.

Caching the categories will not require much memory, if the number of categories is reasonable and no more than the number of items. The data retrieved from the database are item IDs for each category, which are integers (32 bits). If we have a total of 10,000 items, and every item belongs to 4 categories on average, we only need less than 2M bytes. Compared to the cache of ratings, it is much smaller. All the items in one category form a vector, and the total *n* categories become an array of vectors. The operations defined on this cache now are to get all the items of one category and to add new items into the category. Vectors are easier to maintain for these two operations than arrays since vector lengths can be fluctuated. If new operations requiring searching are added, such as getting one specific item from a category, vectors can be easily converted to arrays.

3.4 Implementation vs. Requirements – Experimental Results

Some experimental results are provided to see if the implementations meet the following requirements: consistency, speed and accuracy. First, we will talk about the consistency and the performance. Then, the experimental results about accuracy will be stated.

3.4.1 Consistency and Speed

The *recommendation-by-category* is guaranteed to be consistent with the *recommendation-all-categories*, since they are using the similarity weights to compute the prediction. No empirical results are necessary.

Time (ms)	Simple Pearson	Item-Item	Horting
Recommendation-by-category	2996	4269	3190
Recommendation-all-categories	620	101	94

Table 1: The computation time before categories caching applied (Recommendation-by-category vs. Recommendation-all-categories)

As to the speed, we compared the response time of the *recommendation-by-category* with the *recommendation-all-categories*. Table 1 shows the response time before the categories cache was added to the current cache structure, which caches the information of all the categories. In the Simple Pearson algorithm, almost 78% of its response time in generating *recommendation-by-category* is spent in getting the categories. In the Item-Item and the Horting algorithm, it almost spends 97% of total response time in getting the categories. It is not comparable with the response time of the *recommendation-all-categories*.

After caching the categories, there is a jump in the *recommendation-by-category*'s response time. It responds almost as fast as the *recommendation-all-categories*. The

results are shown in Table 2. The caching enables us compute the *recommendation-by-category* at almost no extra cost. The only cost in the *recommendation-by-category* is to select the items in a specific category instead of looping through all the items.

Time (ms)	Simple Pearson	Item-Item	Horting
Recommendation-by-category	582	103	104
Recommendation-all-categories	588	105	95

Table 2: The computation time after categories caching applied (Recommendation-by-category vs. Recommendation-all-categories)

3.4.2 Accuracy

The experiments focus on the Simple Pearson algorithm and the Item-Item algorithm. As a matter of fact, when a user asks for recommendations, he/she normally asks for "good" product. Almost no one is interested in "bad" products. Thus, our experiment focuses on whether the *recommendation-by-category* can produce the recommendations with high quality.

3.4.2.1 Data set

The dataset we used is from the GroupLens Research Group, and is a part used in the MovieLens recommender system that debuted in 1997. The site has over 35,000 users who have contributed their opinions on more than 3,000 different movies.

This data set is randomly selected from the site's database. It has 1,682 items, 943 users, 100,000 ratings and 19 categories. About 59 people rated each item on average. Of all the ratings, 55,375 were "good". In our experiments, we focused on 18 categories, with the smallest category containing 50 items and the biggest containing 725 items, ignoring the "unknown" category with only 2 items. We divided the database into 4 sets randomly, then one set to be used as the test set (25%) and left three sets to be used as the training set (75%).

3.4.2.2 Evaluation Metric and Experimental Method

The metrics of Precision and Recall have been borrowed from the field of information retrieval. The relation of precision and recall is depicted in Figure 4 [19]. For these metrics, we need to identify items as "good" or "bad" for each user. We assume items rated 4 or greater on a scale of 1-5 are good.

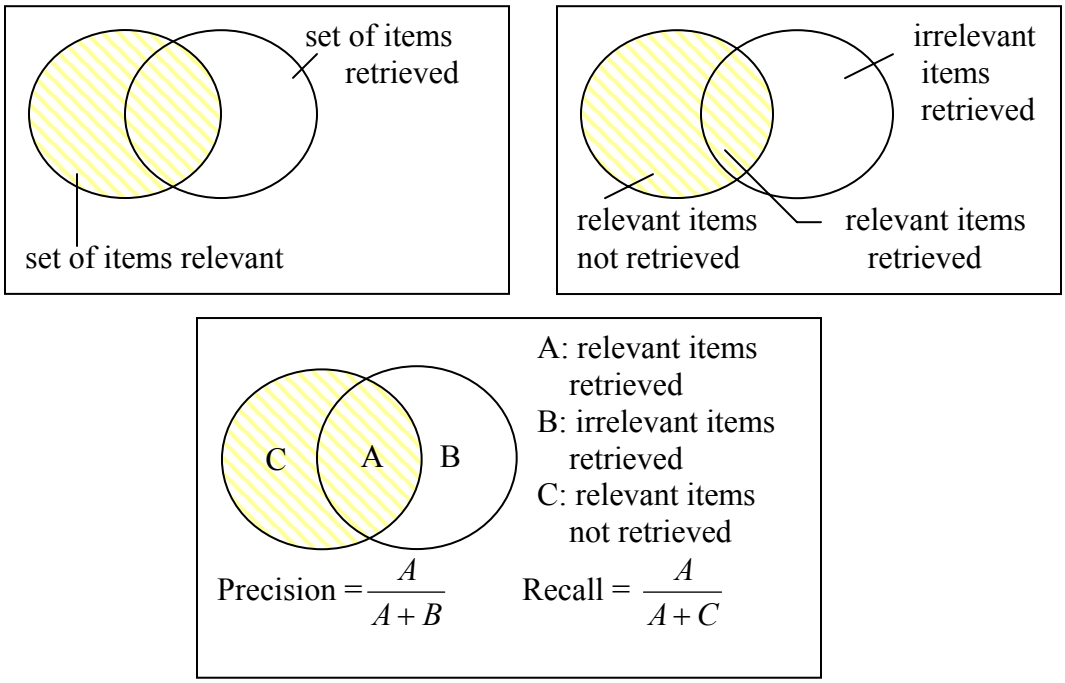


Figure 4: Precision and Recall

In the experiments, we have computed the *recommendation-by-category* for each category based on the training set. From every list of generated recommendations of a specific category, we pick out the ones appearing in the test set and known to be "good" (rating ≥ 4). The formulas are given by the following:

$$precision = \frac{|recommendation\_list \cap good\_in\_test\_set|}{|recommendation\_list|},$$

$$recall = \frac{|\, recommendation\_list \cap good\_in\_test\_set \,|}{|\, good\_in\_test\_set \,|}$$

where $|\, good\_in\_test\_set \,|$ is the number of "good" items of a specific category known to be "good" (appearing in the test set). Here, "good" items means the items with existing ratings more than or equal to 4.

Precision is the fraction of the recommended items that is "good". Recall is the fraction of the "good" items that have been recommended. In our experiment, we interpolate the precision at predetermined recall levels of 0, 5%, 10%, 20%, 30%, …, 100%. We compare the results of recommendations for each category to the baseline of the *recommendation-all-categories*.

During comparison, we will concentrate on the precisions of the recall levels under 50%. Higher recalls means more items in the recommendation list. There is almost no user that will be interested in the recommended items ranking the 100th or even the 50th, for example in movies, books, music, etc. We believe that the user almost always focuses on the first 1~20 recommendations.

Furthermore, we used some constraints in the experiments on both algorithms besides the base case (a minimum of 2 items have to be co-rated by user *a* and user *u* to generate weight), showed as follows. One of the reasons is that both algorithms showed very low precision under the base case. The other reason is that we want to see if the *recommendation-by-category* behaves like the *recommendation-all-categories* as the conditions in the algorithm changes.

- *Constraint (a)*: It requires sufficient number of neighbors to predict an item. We don't generate the prediction for the required item *i* for a particular user *a,* unless there are more than 5 neighbors of user *a* that rated item *i* in the Simple Pearson. In the Item-Item, there are at least 5 neighbor items that have been rated by *a*.

- *Constraint (b)*: We only pick the good neighbors. Good neighbors have to be similar to the active user in some degree. User *u* is considered as a neighbor of user *a* only if their similarity weight is greater than 0.1 in the Simple Pearson. In the Item-Item, item *j* is considered as a neighbor of item *i* unless their similarity is greater than 0.02.
- *Constraint (c)*: We have to be confident with the neighbors, and enough common items between the active user and his/her neighbors give confidence. In order to be considered similar, user *a* and user *u* should have a minimum of 7 items co-rated in order in the Simple Pearson; in the Item-Item, item *i* and item *j* must have at least 30 users co-rated.

3.4.2.3 Experimental Results

	All Categories	**Category 1**	**Category 5**	**Category 7**	**Category 8**	**Category 10**
Property	All categories	Action	Comedy	Documentary	Drama	Film-noir
Total ratings	100,000	25,589	29,832	758	39,895	1,733
Total "good" ratings	55,375	13,534	14,946	477	24,605	1,244
Total items	1,682	251	505	50	725	24
Average ratings per item	59.45303	101.9482	59.0732673	15.16	55.02759	72.20833
Percentage of "good" ratings	0.55375	0.528899	0.50100563	0.6292876	0.616744	0.71783

Table 3: Representative categories

We pick 5 representative categories from the experimental results and the *recommendation-all-categories* here. As shown in Table 3, there is one big category with 725 items and one small category with 24 items. Among these 5 categories, there is also a very sparse category with an average of 15 ratings per item and a more dense category with sufficient data which has over 100 ratings per item. We ran both user-user and item-item algorithms for the base case.

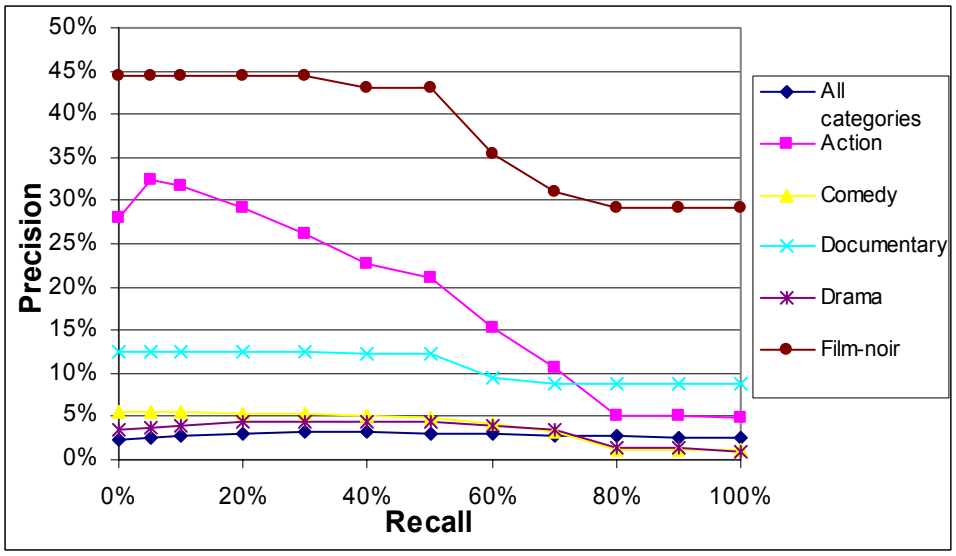


Figure 5: Precision vs. Recall (the Simple Pearson with Base case)

In the Simple Pearson algorithm, every category has better precision than the *recommendation-all-categories* in the recall levels less than 50%, although many of the differences may not be statistically significant. As shown in Figure 5, category *Action* and *Film-noir* perform much better than any other category, and the precision at the 20% recall are 29.15% and 44.44%. The cause may primarily be the sufficient available ratings they have, with 101 ratings/item and 72 ratings/item respectively. Category *Comedy* and *Drama* are the fairly large subsets of the whole item set and have similar average ratings per item (59 ratings/item, 55 ratings/item) compared to the *recommendation-all-categories* (59 ratings/item). Both categories have similar precision like the *recommendation-all-categories*, with 5.39% and 4.40% at the 20% recall vs. the *recommendation-all-categories*' 3.02%. Conversely, category *Documentary* and *Film-noir*'s precision, 12.48% and 44.44% at the 20% recall, are much better than the *recommendation-all-categories*, which are the small categories with very small amounts of items, 50 and 24 items respectively.

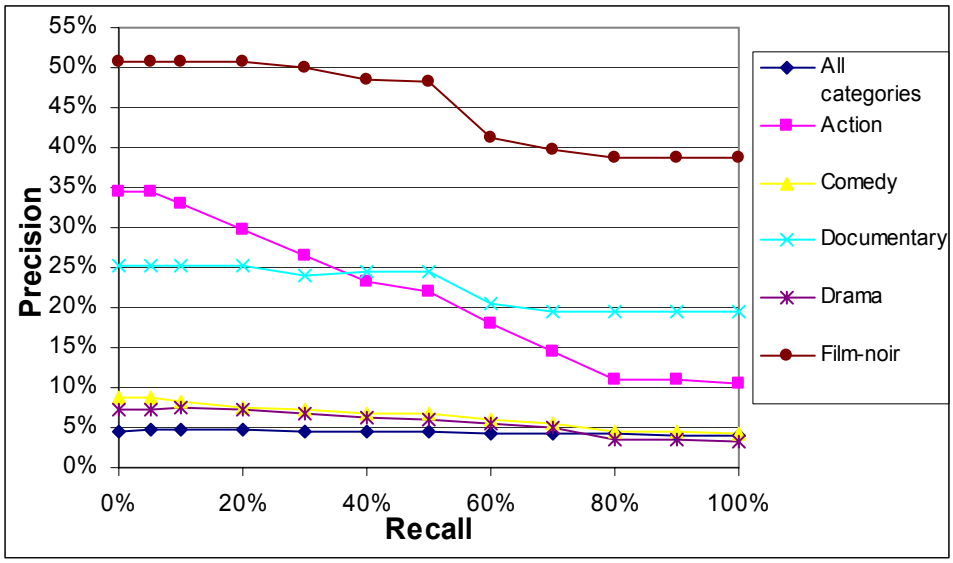
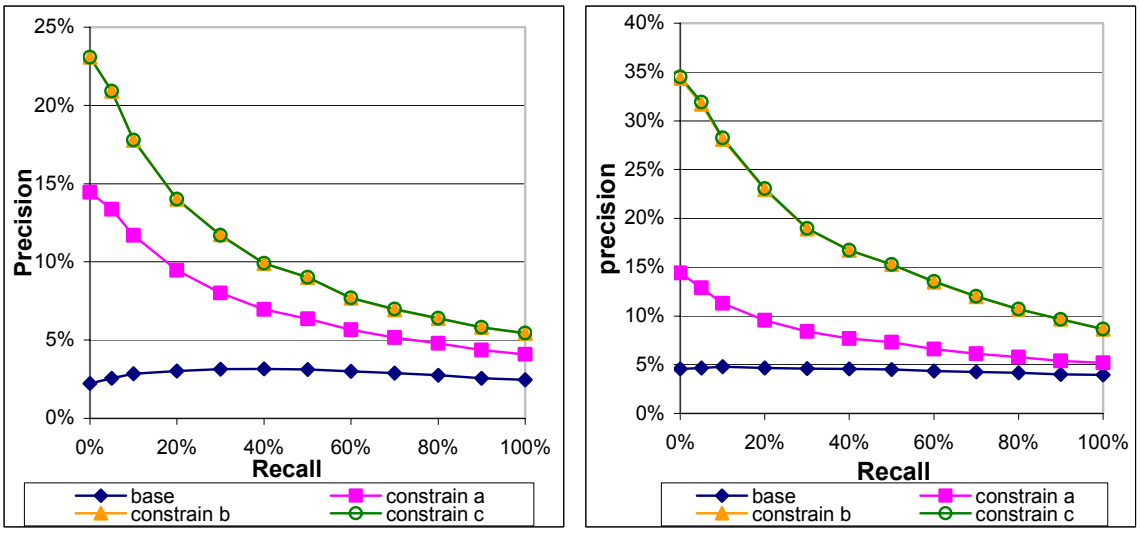


Figure 6: Precision vs. Recall (the Item-Item with Base case)

By using the same data set, we present the results of these 5 categories and the *recommendation-all-categories* in the Item-Item algorithm in Figure 6. The same as in the Simple Pearson algorithm, all the categories also have better precision than the *recommendation-all-categories*. Category *Action*, *Documentary* and *Film-noir* outperform the other two categories (*Comedy* & *Drama*) and the *recommendation-all-categories*, with 29.82%, 25.14% and 50.72% at 20% recall respectively. The reason may be identical to the Simple Pearson case.

Previously we identified several constraints that can improve the accuracy of predictions. The interesting thing is that those constraints have significant effects on the quality of *recommendation-by-category* as well as the *recommendation-all-categories*. We ran two algorithms, the Simple Pearson and the Item-Item, based on four cases: base case, *Constraint (a)*, *Constraint (b)* and *Constraint (c)*. The results show that the tighter constraints on the neighbors result in better precision. We see consistent improvement due to constraints in all categories.



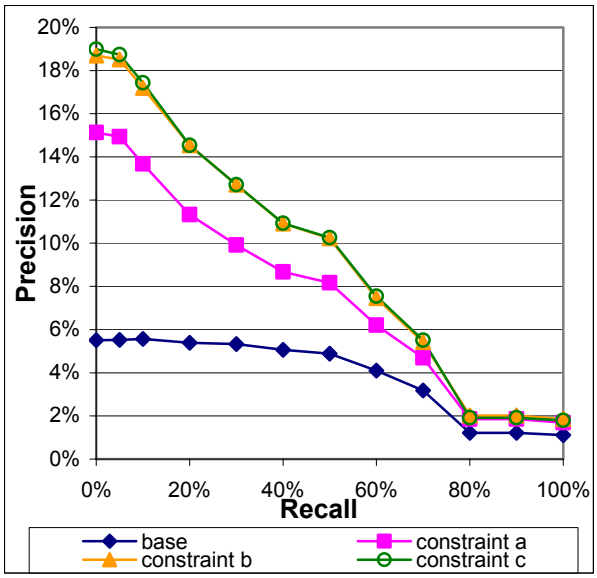
(1) All categories in the Simple Pearson **(2) All categories in the Item-Item**

Figure 7: Four cases of the recommendation-all-categories in the Simple Pearson and the Item-Item algorithms

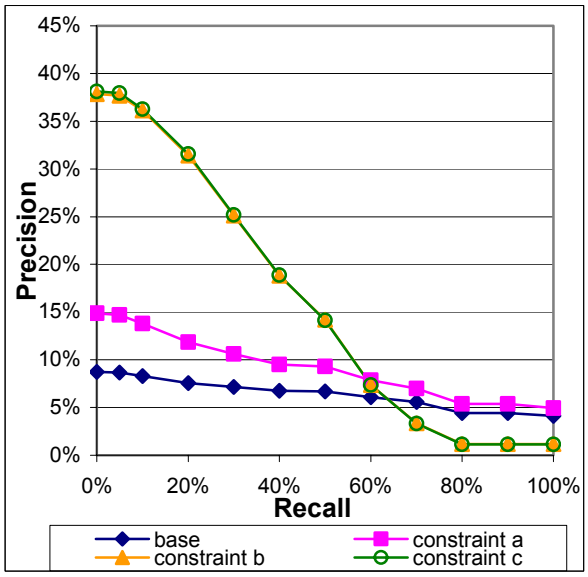
In both the Simple Pearson and the Item-Item algorithm, the precision of the *recommendation-all-categories* in the base case are incredibly low, approx. 3.02% and 4.67% at the 20% recall. *Constraint (a)* increases the precision by more than 200% to 9.47% in the Simple Pearson and by 100% to 9.55% in the Item-Item algorithm. *Constraint (b)* gives the precision another jump to 14.01% and 22.97% respectively. Compared to *Constraint (a)* and *Constraint (b)*, the improvement of the precision made by *Constraint (c)* is not significant. The results are shown in Figure 7.

The same with the *recommendation-all-categories*, the precision of almost all the categories improve as the constraint gets tighter. Worthy to be noticed is that from *Constraint (b)* to *Constraint (c)*, the precision of categories *Documentary* and *Film-noir* that are small categories show almost no improvement. More interesting thing is that category *Film-noir* even starts to decline in the Simple Pearson (from 60.41% in *Constraint (b)* to 60.22% in *Constraint(c)*), which is depicted in Figure 9. Category *Comedy* and *Drama* that are big categories show very little improvement from constraint (b) to (c), shown in Figure 8 (category *Comedy* only). From *Constraint (a)* to *(b)*, or from *Constraint (b)* to *(c)* are one step tighter eliminating neighbors. Tighter constraint,

of course, gives a pool of high quality neighbors, but it also shrinks the pool size of potential neighbors. When the constraint becomes too tight, the pool size becomes too small for the active user to get enough neighbors.

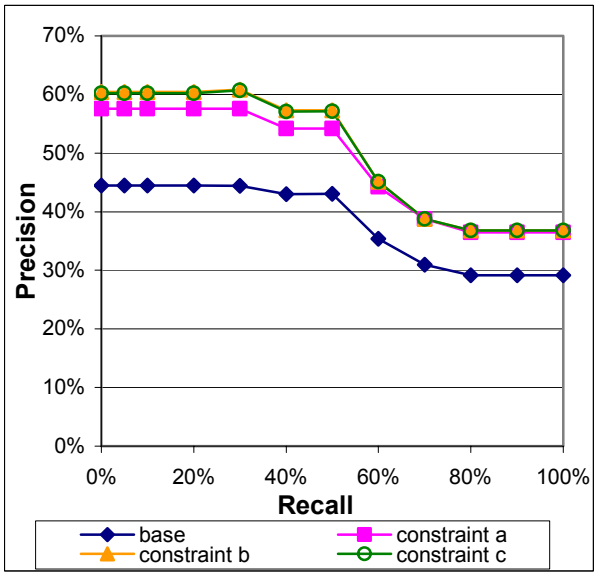


(a) Category Comedy in Simple Pearson

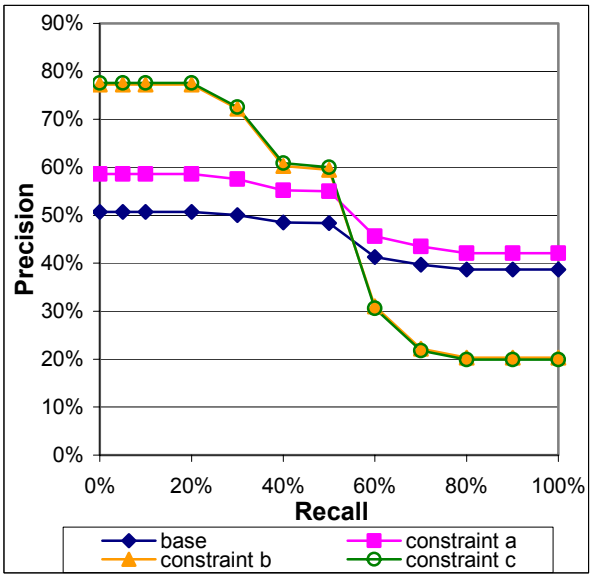


(b) Category Comedy in Item-Item

Figure 8: Four cases of Category Comedy in the Simple Pearson and the Item-Item



(a) Category Film-noir in Simple Pearson



(b) Category Film-noir in Item-Item

Figure 9: Four cases of Category Film-noir in Simple Pearson and Item-Item

3.4.3 Conclusion of the Experiments

The experimental results show that the implementation of this new capability meets the three requirements: consistency, speed and quality.

In the experiment, neither the Simple Pearson nor the Item-Item algorithm shows very good results in the *recommendation-all-categories*, especially in the base case. This precision is lower than the precision values reported by other collaborative filtering researchers. The reason may be that we treat items unrated in the test set as "bad" instead of not counting them by other researchers.

The *recommendation-by-category* shows better precision than the *recommendation-all-categories*, especially for the small categories and the categories that have dense ratings. The less dense categories tend to perform like the *recommendation-all-categories*, but they still perform a little better than the *recommendation-all-categories*.

Both the *recommendation-by-category* and the *recommendation-all-categories* are sensitive to the neighbors' size and quality, which give us a plenty of space to give constraints. But it is also hard to balance: higher threshold in eliminating neighbors produces good quality recommendations, but left more users unpredictable; lower threshold degrades the quality, but a lot more people can get recommendations.

3.5 Discussions and Future Works

The implementation of this new capability is successful, but it also leaves us plenty of questions about the algorithms we have used. Here, we will discuss a different way generating the similarity weight for the *recommendation-by-category* and ask some questions for future works.

3.5.1 Discussions

The similarity weight used in the *recommendation-by-category* is in some sense biased. For example, user *A* and *B* who have similar opinions in category *X*, may differ in every other category. But user *A* and *C* who have opposite opinions in category *X*, agree with each other in all the other categories. Under this situation, weight $w_{A,B}$ and $w_{A,C}$ generated by function (1) stated in section 3.2.1 cannot present the closeness of user *A* and *B*, and the closeness of *A* and *C* in category *X*. To generate recommendations for category *X* to user *A*, it might be better to re-compute the similarity between *A* and *B*, and the similarity between *A* and *C* through all the items in that category. Theoretically, the *recommendation-by-category* will be much more precise in this manner.

However, the above method has some practical difficulties. If we compute different similarity weights for each category, we will encounter two problems. The first is the sparsity problem. Narrowing down to a subset (category) of items makes the rating matrix become much more sparse, which means we will have less ratings per user. If an active user has less ratings, he/she will have less overlap with the other users. Less overlapping means less neighbors in generating predictions, and we will have less confidence about the neighbors. For example, we have a category that has a total of 50 items and 758 associated ratings with an average rating of 15 per item. In such a category, we are not able to get any good quality predictions or even predictions at all if we only compute similarities based on ratings for the same category.

The other problem with computing separate similarity weights for each category is consistency. The *recommendation-all-categories* and *recommendation-by-category* will give the same item different scores and rankings. The users will not be happy once they see the rankings of the same set of items differ depending on how they request a recommendation.

3.5.2 Future Works

In the future, we may focus on how to balance the constraints to get an optimized size and quality of the neighbors in the future. Should *recommendation-by-category* use a different method to compute similarity that is category owned? We could penalize other categories when generating predictions? Or perhaps even penalize the weight from categories that the user doesn't like in computing the conventional *recommendations*. All these questions could be considered.

4 CORBA API of Java Recommender System

The Java Recommender System initially uses Java RMI as its remote interface, which prevents the server from talking to the clients in programming language other than Java. In this section, we will explain the CORBA API that can be used to support a wide variety of languages and platforms.

Consider CORBA as the middleware because it integrates machines from many vendors, ranging from mainframes through desktops to hand-holds and embedded systems, and allows the computer network to work together by using the standard protocol -- IIOP (Internet Inter-ORB Protocol). It also has various language mappings including Java and C++.

We initially expected the CORBA implementation to be easy. But in the implementation, we faced many problems. In order to understand what we did and the problems we encountered, we give a brief overview of CORBA. Then, we explain the designs and problems we had. And finally, we present the empirical result of comparing CORBA and Java RMI.

4.1 CORBA Overview

CORBA (The Common Object Request Broker Architecture) is an open distributed object computing infrastructure being standardized by the OMG (Object Management Group). As a distributed system middleware, CORBA can potentially operate across different hardware platforms and use different operating systems and programming languages.

Like RPC (Remote Procedure Call), CORBA is used in distributed computing environment. But the main difference is that CORBA is Object Oriented. The CORBA object resides in the server and is invoked by the client that must obtain the CORBA

object reference. The CORBA object is location transparent, which means we can do the operations in the same syntax no matter where the object is. CORBA has a lot of language mappings, including C, C++, Java, COBOL, Ada, Lisp and Smalltalk. It also supports bridges connecting other distributed computing environment, like Microsoft's DCOM.

The services that an object provides are given by its interface. The interface to a CORBA object is defined by OMG's IDL (Interface Definition Language), which is adapted to different programming languages. IDL allows us to define interfaces for CORBA objects in a way that is platform, implementation independent, and language neutral. IDL *compiler* is necessary in developing CORBA applications. The compiler maps the IDL definition to a specific programming language, and generates stub code for client side invocation and skeleton code for server defining CORBA object implementation.

An object's IDL defines the interface of the object and consequently how to invoke it. An IOR (Interoperable Object Reference) is a globally unique identifier for a CORBA object. If an object's IOR is known, it can be invoked without further knowledge of where it is, what machine it runs on, how it is implemented and so on. This is one of the chief strengths of CORBA, since it hides a lot of nasty details that programmers would otherwise have to worry about.

4.2 API Design

There were two CORBA-like approaches we could take: RMI-IIOP, and pure CORBA. They both have advantages, as well as drawbacks. RMI-IIOP can free us from developing the IDL, which is the necessary step for CORBA. But RMI-IIOP is newer yet, so it isn't well supported.

4.2.1 RMI-IIOP Design

To use Java RMI over IIOP as the server side protocol and CORBA in the client side was our first thought. This translates CORBA requests from a client to Java RMI requests to a server. This would allow CORBA clients to connect to our server with minimal changes to the server. And the compiler *rmic* generating IDL file from Java programs is freely available.

RMI-IIOP Overview

According to the Java documents, Java RMI over IIOP (RMI-IIOP), in a sense, is a marriage between RMI and CORBA. It allows CORBA client to talk to Java RMI servers. Like CORBA, RMI-IIOP is based on open standards defined by OMG and uses IIOP as its communication protocol.

RMI-IIOP preserves existing investment in RMI binaries, so it can communicate with RMI without any source-code changes or recompilation. As to CORBA, since the IDL file can be generated by *rmic* compiler, the CORBA client will easily talk to RMI-IIOP server. The semantics of CORBA objects defined in IDL are a superset of those available to RMI-IIOP objects.

Problem Encountered

Developing the server was quite simple. Migrating to RMI-IIOP required only changing a few lines of code. However, in client side development, we encountered far more difficulties than what we expected. Instead of one IDL file, there are a number of recursively connected IDL files full of value types (pass-by-value objects) generated by *rmic*, the RMI compiler. An essential property of value types is that their implementations are always local, and the receiving side of a parameter passed by value receives a description of the state of the object, then instantiates a new instance with the state but having a separate identity from that of the sending side.

The reason is that all the *Java serializables* are mapped to the CORBA value types in the Java language to IDL specification. Therefore, every *Java serializable* to be passed between client and server must be re-implemented in the language of the client.
This means we have to develop a lot of C++ codes which are compatible to a large amount of Java basic classes like java.lang.Throwable, java.lang.Exception, etc. To our knowledge, only Websphere implements these in C++ and includes them in their release. This cause the coding become so complicated -- almost need to develop a library, and it will take too much time.

4.2.2 Pure CORBA Design

The pure CORBA was our second choice. In this design, server and client will both use CORBA to communicate. Unlike in RMI_IIOP, we can manually construct our IDL file to avoid value types.

```
module server {
      struct ItemPredictionCorba {
        long itemID;
        float prediction;
        long userID;
    };

     struct ItemRatingCorba {
        long itemID;
        float rating;
        long userID;
    };
    …  …
};
```

Figure 10: structs in IDL file

IDL file

Rather than defining value types, we can use simple **structs** to pass rating and prediction data between client and server, as shown in Figure 10. Compared to the value types, it simplifies coding on both server and client side, which free us from keeping two copies

of implementation of the same object and registering the factories in both (server and client) side.

Exception Handling

Another important thing is how to deal with the exceptions caught in the server side. In the server side, there are two types of exceptions: CF user exceptions (application-specific exceptions) and Java exceptions. For each CF user exceptions, related CORBA user exceptions have been created. In CORBA implementation class of the server side, all the CF exceptions are caught and thrown as related CORBA user exceptions.

Since Java exceptions are all *Java serializables* and the value type is the only way to implement these objects on the client side, we would encounter the same problem as in RMI-IIOP design. To avoid using value types, we implemented a wrapper class on the server side that catches all Java exceptions and throws a single CORBA user exception. Thus, besides all the CF user exceptions, a CORBA user exception – *org::recommender::server::corba::CFEx* – is defined in IDL to catch all the Java exceptions.

4.2.3 Discussion

Our two API designs are two different perspectives. RMI-IIOP design is not desirable due to lack implementation of Java basic *serializable* C++ implementations. As RMI-IIOP gets mature and libraries become available, RMI-IIOP will become more desirable. By using Pure CORBA on the server side, we can manipulate server objects and exceptions in an easier way: using **struct** wrapping the server side objects – *ItemPrediction* and *ItemRating*; throwing CORBA user exceptions on the client side. On the other hand, downside of using pure CORBA is that we need two separate server implementations.

4.3 Comparison of CORBA and Java RMI

The basic functionality of RMI and CORBA is similar since they both hide the communication details of remote method invocations. However, RMI and CORBA were built with different goals. RMI is designed for Java and support all the details of Java language. On the other hand, it only offers basic functionality of ORB (Object Request Broker). CORBA is a middleware that is not bound to a particular programming language and offers a rich set of object services, common facilities and domain facilities besides ORB. It can be easily seen that CORBA is more complex.

Some performance comparisons between RMI and CORBA have been made by Juric, Zivkovic and Rozman [6]. Here we will focus on the unique objects of our Java Recommender System. In our CF System, Java RMI and CORBA only have tiny differences on the server core, where some objects passing rating and prediction in the CORBA implementation are the **struct**s whose values passed by the rating and prediction class of the RMI version. This apparently introduced a little redundancy. Those redundancies can be ignored when a large amount of computation is performed.

In order to compare these two models, a CORBA Java Client has been developed since we want to measure the remote protocol overhead only. In our experiments, we will only compare the performance of the core method of our Recommender System, which is the recommendation. We then simulated single and multiple client scenarios.

4.3.1 Single Client

In this single client experiment, we compare the execution time of fixed number of recommendations in two cases: the server and the client on the same machine, and on different machines.

Figure 11 shows that RMI is consistently faster than CORBA. We believe this is because of the much greater complexity of the CORBA architecture. But RMI introduced more

overhead than CORBA when the server and the client are running on different machines. Compared the different machines to the same machine case, the average performance degradation of RMI is 4.29% vs. CORBA's 2.09%. This could indicate that CORBA handles network communication better than RMI. But even with this advantage, CORBA is still slower than the RMI on the network.

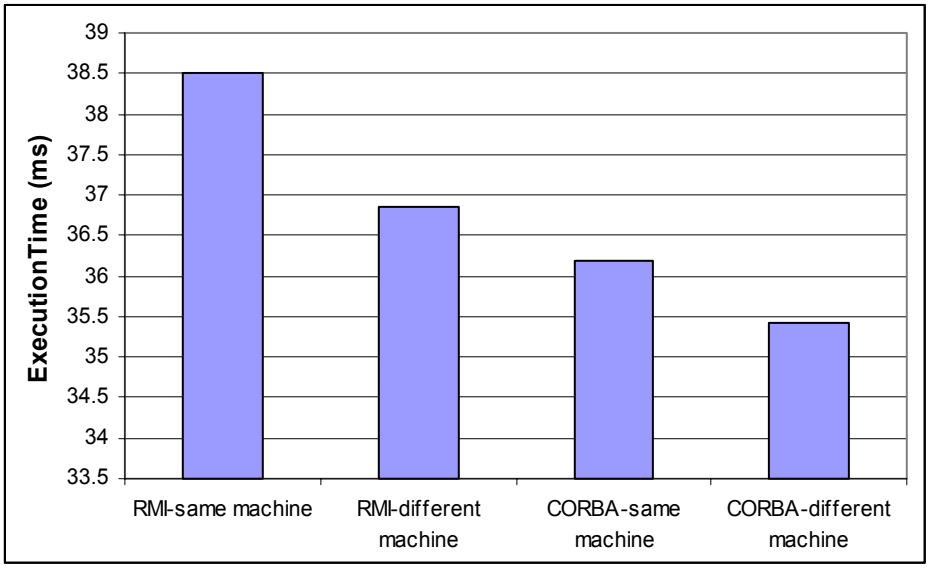


Figure 11: Single Client Scenario: Average Time for Top 200 recommendations (number of recommendations = 200)

4.3.2 Multiple Clients

Each server object will have to handle multiple clients. To test that, we create multiple threads which simulate multiple clients, each performing the same task – get recommendations. The experiment was performed on the computer which has two processors and is heavily loaded by many processes.

Figure 12 shows the execution time increases in both RMI and CORBA with the number of the clients increasing. But the performance degradation of RMI is slightly larger than CORBA, but not significant. The reason might be the less optimized RMI code. RMI is almost 500ms faster in the 1 client scenario. While the number of clients increases,

RMI's performance degradation is slightly larger than CORBA. After the point of 30 clients, CORBA's speed started to exceed RMI's about 200 ~ 300 ms, however, the difference is not so obvious.

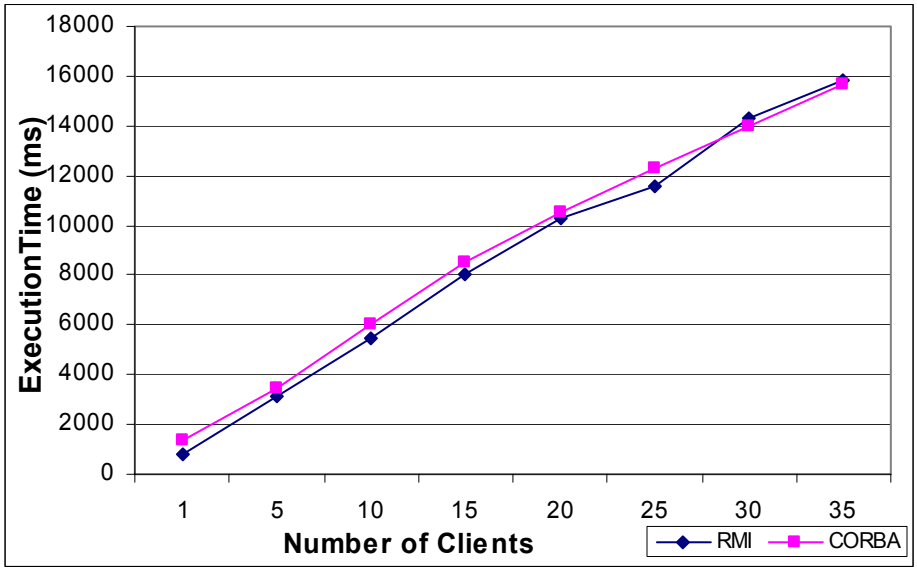


Figure 12: Multi Clients scenario: RMI vs. CORBA (number of recommendations per thread = 10)

4.3.3 Discussion

It's not clear for our application that either CORBA or RMI performs significantly better than the other. To our Java Recommender System that is set for middle size database, Java RMI is enough for pure Java environment. CORBA can be seen as a helping API which provides cross-language function to bridge different platforms.

5 Correctness Testing for Java Recommender System

The goal of the correctness testing is to ensure that our Java Recommender System working correctly and free of bugs. After finishing the implementations of the initial Java Recommender System, we ran into some problems. First of all, the system didn't respond correctly to some extreme inputs. Secondly, the synchronization part in our system didn't work properly. The server crashed within 20 minutes while 10 client threads were running.

The above problems made testing not only necessary but also urgent. However, which kind of test should be done? There are many modules and plenty of threads in the server side, as depicted in Figure 2 of section 2. As to the modules, unit test was the first that have come to our thought, but was not practical to finish testing all the units and threads and bug-fixing in two and half months' time limit.

Thus we focused on end-to-end testing of the whole system. It can be divided into two parts: CF Interface Component test and Multi Thread test.

5.1 CF Interface Component Test

The component test is the most basic part, which ensures main server interface methods work properly. A test plan was made for each main interface method.

Test Plan & Implementation

The plan is combined with 14 individual test modules, and each module is for one *CFEngine* interface method. For each individual module, various test cases have been involved. Generally, all the test cases are developed from the following basics.

- *Illegal Parameter*: It is important to deal with all predictable illegal parameters in every method. The server should be robust to such errors from the clients.

- *Single User*: Our second case is to simulate one user. A *CFEngine* interface method is executed once or several times and the result is checked.
- *Multiple Users*: This test case is to demonstrate several users doing the same operation on different items.
- *New Users (Items)*: All the new users and items not appearing in the database and the cache have to be handled correctly. That the user or item doesn't appear in the database does not mean that it does not exist.

During the implementation, all the server interface methods have been classified into three groups: core components, DB components and peripheral components. There are group specific testing criteria in developing each testing module. For example with the group of core components, the return values of *getRecommendations()* and *getPredictedRating()* can be compared to see if they have the same predicted rating.

Results of Testing

When the server is initially designed, there were no coding standard. Thus, each team member has programmed in his or her own way. This meant the majority of the methods have some problems. For example, almost no method has dealt with the illegal parameters correctly and those illegal parameters caused various Java exceptions to be thrown. We also found that an inconsistency exists between the predicted rating of *getRecommendations()* and *getPredictedRating()*.

In the process of writing tests, we were forced to clearly define the specifications for each method. The specifications for each object and function has been developed and applied, after all the problems have been found. The specifications, as a basic of our system, not only help fixing the existing problems, but will also apply to the future development.

5.2 Multiple Thread Test

The cache makes the computation in the server side much faster, but the multiple server threads accessing the cache concurrently does introduce a lot of concurrency issues. Thus, the right synchronization scheme is critical to make the server work correctly, and a careful plan and strategy is very important.

Test Plan & Implementation

To test all the server side threads, we decided to run many client threads, which simulate different users doing various operations. To ensure a correct synchronization scheme, the interaction of different types of threads also need be tested. Based on the above thoughts, we have designed the testing cases that can be used in different threads' interaction if the server is appropriately configured.

1. Several threads of *getRecommendations*() simulating multiple readers. They read the information from the cache only.
2. One or more threads of *getRecommendations*() interacting with one thread of *setRating*(). It simulates that one client is writing to the cache, while the other ones are reading.
3. Several threads of *setRating*(). This test case can be used to assure that only one writer is allowed at a time if both threads are ran by same user.
4. One or more threads of *setRating*() and one or more threads of *removeRating*(). It performs the same task as in test case 3 and provides another perspective.
5. Several threads each of *getRecommendations*(), *setRating*(), *removeRating*(). Further tests on the synchronization of the cache.
6. Multiple threads: More than 10 threads randomly select the operations of *CFEngine* interface.

Result of Testing

The majority of the bugs and deadlocks were due to the server cache structure and carelessness in the synchronization design. For example, the entries of the big hash maps (*UserInfo* and *ItemInfo*) were synchronized hash map. It not only made a lot of

operations in these two objects unnecessarily synchronized, but also introduced a lot of synchronized blocks into the server's main module – *DataManager*. This made it very hard for us to trace the deadlocks. Since the cache behaves exactly like the reader-writer problem, a much neater scheme -- reader-writer lock -- has been developed instead. We have changed these two objects to non-synchronized hash maps and created a reader-writer lock for each object. This new scheme drastically reduced the number of synchronized blocks in the *DataManager* and makes testing much easier.

The result of this multiple-thread testing is significant. In the beginning of this testing, the system could not run for more than 20 minutes without crashing. During the period of whole test, more than 30 synchronization bugs and deadlocks have been fixed, and some basic synchronization schemes have been changed. The system is much more stable and has run more than 2 days with 20 threads.

5.3 Discussion

The correctness testing ensures the stability of the Java Recommender System. If we had made object and function specifications available before we started to create the software in the first place, we would have saved much time in debugging and overall development. We could have made the testing cases, which are based on the specifications, available before programming.

Programming is not the only part in building software. A careful and detailed picture or model is critical, which is the macro view of the software. It not only gives the direction to programming, it also can save time in testing. On the other hand, testing is also an important part in software engineer. It can give a micro view and help to consider every details of the system.

6 Conclusion

In this project, we developed a new capability, a new remote API and created correctness testing code for the Java Recommender System. The new capability, *recommendation-by-category*, makes the system more functional, which gives the user an option to limit recommendations to a category they are interested in. The experimental results show that this new capability performs well compared to the conventional *recommendation* capability. The CORBA API makes the system much more compatible and suitable for distributed multiple platforms and various programming languages. Finally, the correctness testing makes the whole system much more stable and ready to run. During the testing, not only the specifications for the whole system have been considered, a much simpler and neater way that utilizes the reader-writer object locks for synchronization has also been provided.

The Java Recommender System implements Collaborative Filtering techniques to alleviate the effects of information overload. It achieves scalability by limiting the amount of data in cache. Caching technique also makes the whole system robust and fast. It suits distributed multiple platform applications via Java RMI and CORBA IDL. Furthermore, it is also a great tool for research. For research purposes, it provides a lot of collaborative filtering algorithms and an experimental class to test the algorithm without little effort. It also can integrate new algorithms easily through the *CFAlgorithm* interface.

Future works may focus on building the client in various programming languages and refine cache structures to enhance the performance and reduce the memory requirements. It is possible to improve the quality of recommendations in some algorithms by constraining the important parameters to reach an optimal point.

7 Reference

[1] Upendra Shardanand and Patti Maes. Social Information Filtering: Algorithms for Automating "Word of Mouth". In Proceedings of ACM CHI '95 Conference on Human Factors in Computing Systems, Vol 1, 210 -217, 1995.

[2] Jonathan L. Herlocker, Joseph A. Konstan, Al Borchers, and John Riedl. An algorithmic framework for performing Collaborative Filtering. *Communication of the ACM.* 1999.

[3] Jack Breese, David Heckerman, and Carl Kadie. Empirical Analysis of Predictive Algorithms for Collaborative Filtering. In *Proceedings of the Fourteenth Conference on Uncertainty in Artificial Intelligence*, Madison, WI, July, 1998.

[4] Konstan, J., Miller, B., Maltz, D., Herlocker, J., Gordon, L., and Riedl, J.. GroupLens: Applying Collaborative Filtering to Usenet News. *Communications of the ACM* 40,3 (1997), 77-87.

[5] Badrul M. Sarwar and George Karypis and Joseph A. Konstan and John Reidl. Item-based collaborative filtering recommendation algorithms. *World Wide Web*, 2001, 285-295.

[6] Charu C. Aggarwal and Joel L. Wolf and Kun-Lung Wu and Philip S. Yu. Horting Hatches an Egg: A New Graph-Theoretic Approach to Collaborative Filtering. *Knowledge Discovery and Data Mining*, 1999, 201-212.

[7] Badrul Sarwar, George Karypis, Joseph Konstan, John Riedl. Analysis of Recommendation Algorithms for E-Commerce. *Proceedings of the 2nd ACM conference on Electronic commerce*, October, 2002.

[8] Chumki Basu and Haym Hirsh and William W. Cohen. Recommendation as Classification: Using Social and Content-Based Information in Recommendation. *In proceedings AAAI*, 1998, 714-720.

[9] Finton Bolton. Pure CORBA. *Sams Publishing*, 2002.

[10] M.B. Juric, A. Zivkovic, I. Rozman. Comparison of CORBA and Java RMI Based on Performance Analysis. *MHSS'98*, 1998

[11 Michi Henning, Steve Vinoski. Advanced CORBA programming with C++. *Addison-Wesley*, c1999.

[12] Jonathan L. Herlocker. CFEngine Manual.

[13] Jon Siegel. Quick CORBA 3. *New York : Wiley*, c2001

[14] Sun Microsystems. RMI-IIOP tutorial, http://java.sun.com/j2se/1.3/docs/guide/rmi-iiop/index.html

[15] Sun Microsystems. Java Remote Method Invocation, http://java.sun.com/products/jdk/rmi/

[16] Sun Microsystems. Java IDL, http://java.sun.com/j2se/1.3/docs/guide/idl/index.html

[17] Object Management Group. Common Object Request Broker Architecture (CORBA) v 2.4. http://www.omg.org/docs/formal/00-10-01.pdf

[18] Abraham Silberschatz, Peter B. Galvin. Operating System Concepts, Fifth Edition. *John Wiley & Sons, Inc.* 1999.

[19] Ricardo Baeza-Yates, Berthier Ribeiro-Neto. Modern Information Retrieval. *ACM Press, Addison-Wesley*, 1999.