Caching Strategies for In-Memory Neighborhood-Based Recommender Systems

Simon Dooms, Toon De Pessemier and Luc Martens
WiCa group, Dept. of Information Technology, iMinds-Ghent University
Gaston Crommenlaan 8 box 201, B-9050 Ghent, Belgium
{*simon.dooms, toon.depessemier, luc.martens*}*@intec.ugent.be*

Keywords: Recommender Systems, Cache, LRU, SMART, Speedup

Abstract: Neighborhood-based recommender systems rely greatly on calculated similarity values to match interesting items with users in online information systems. Because sometimes there are too many similarity values or available memory is limited it is not always possible to calculate and store all these values in advance. Sometimes only a subset can be stored and recalculations cannot be avoided. In this work we focus on caching systems that optimize this trade-off between memory requirements and computational redundancy in order to speed up the recommendation calculation process. We show that similarity values are not equally important and some are used considerably more than others during calculation. We devised a caching strategy (referred to as SMART-cache) that incorporates this usage frequency knowledge and compared it with a basic least recently used (LRU) caching mechanism. Results showed total execution time could be reduced by a factor of 5 using LRU for a cache storing only 0.2% of the total number of similarity values. The speedup of the SMART approach on the other hand was less affected by the order in which user-item pairs were calculated.

1 INTRODUCTION

Every day recommender systems in al kinds of domains are processing huge amounts of data to match users with items in a personalized way. Because user adoption increases, technical requirements such as performance in terms of execution time and memory usage of these systems are becoming increasingly important. Especially for online scenarios (e.g., movie recommendation website) execution time can be the differentiating factor between real-time incorporating user feedback and daily batch processing. In this work we try to reduce execution time and allow flexible memory usage by introducing caching principles.

A cache enables rapid access to popular or frequently accessed data (Qasim, 2011). Its use can significantly speed up data throughput and therefore also greatly impacts the total execution time of algorithms that are largely data dependent (e.g., recommender systems).

We will focus on collaborative filtering systems, more specifically nearest neighbor (k-NN) algorithms because they are most likely to benefit from cache enhancement. The general idea behind collaborative filtering algorithms is that community knowledge can be exploited to generate more accurate recommendations for individual users (Jannach et al., 2011). Nearest neighbor algorithms try to harvest this community information to identify similar users or items in the system. Similarity can be computed in many ways, but most often involves comparing the ratings either given by users or received by items and the calculation of some kind of similarity metric. The rating behavior of such similar neighbors can then be used to extrapolate ratings for new users or items. These types of algorithms will have to determine the pairwise similarity values for all users or items in the system, values which will then be reused many times during the recommendation calculations and seem therefore very suited for caching.

Research literature is rather limited when it comes to the subject of caches and recommender systems. To our knowledge, two main contributions exist that report on enhancing recommendation response times through the use of caches. The work of Qasim et al. has introduced the concept of partial-order based active cache (Qasim et al., 2009) in which they construct a caching system that allows to estimate nearest neighbor type queries from other queries in the cache. The usefulness of the cache is thereby extended beyond exact previously-cached entries. The caching structure is intended to prevent recommendation lists

to be calculated by estimating them using only previous answers in the cache.

Another cache-enhanced recommender system was the genSpace recommender system presented by Sheth et al. as a prefetching cache that prefetches all recommendations in order to prevent slow recommendation calculation on-demand (Seth and Kaiser, 2011). Their approach differs from ours since their cache rather prevents unnecessary recommendation recalculation while we attempt to speed up the calculation of the complete recommendation process (rating prediction for all users and items) in order to reduce overall calculation time without compromising recommendation accuracy.

Our contribution comprises the study of how similarity values are used during recommendation calculations (for user-based collaborative filtering) and the introduction of a new caching principle (i.e., SMART cache) that takes advantage of this knowledge.

In the next sections we will define the user-based collaborative filtering (UBCF) algorithm and show why and how this algorithm may benefit from caching internal values.

2 UBCF ALGORITHM

We specifically focus on in-memory recommendation algorithms, which we define as algorithms working completely in the random-access memory (RAM) of the computer without using external data resources (e.g., databases or files) for data storage or retrieval during computation. Because the RAM of a computing machine is usually limited in size, keeping track of all internal temporary values (e.g., similarity values) may be impossible and some kind of caching paradigm can be introduced. To be able to experiment with different caching modes with respect to in-memory recommendation algorithms, we implemented the well known user-based collaborative filtering algorithm (Jannach et al., 2011). This algorithm is widely accepted and commonly used in recommender systems in various domains (Jannach et al., 2011). Here we apply it to predict a rating for all user-item pairs, based on the ratings of similar users (i.e., neighbors). The high-level algorithm structure can be found in the following pseudocode fragments.

```
Algorithm Recommendation_calculation
    for user in users
        for item in items
            rec_value ← Calc_rec_value(user, item)
End ALGORITHM
```

```
Function Calc_rec_value(user, item)
    vote ← 0
    weights ← 0
    neighbors ← Neighbors_who_rated_item(user, item)
    for neighbor in neighbors
        simil ← neighbor_similarity
        rating ← neighbor_rating
        vote ← vote + (simil × rating)
        weights ← weights + simil
    Return vote / weights
End FUNCTION
```

```
Function Neighbors_who_rated_item(user, item)
    for neighbor in {users who rated item}
        neighbor_similarity ← Pearson(user, neighbor)
    Return the 20 most similar neighbors together with their original rating for item (neighbor_rating).
End FUNCTION
```

Our implementation employs a simple weighted average scheme to come up with the final recommendation value. Pearson correlation (Resnick et al., 1994) was used as similarity metric and the neighborhood size was restricted to the top 20 neighbors (as was found to be a reasonable amount (Herlocker et al., 2002)). A straightforward linear transformation was employed to rescale the Pearson correlation value from [-1,+1] to [0,1] to simplify the calculations.

3 USAGE OF SIMILARITIES

To calculate the predicted rating of a user for an item, neighboring users will have to be determined. That in turn requires the pair-wise similarity of that user with every other user in the system that rated the item. The most important line of code here is:

```
neighbor_similarity ← Pearson(user, neighbor)
```

Using a code profiler, we analyzed the runtime of our recommender system and found that 78% of the total execution time was spent calculating these similarity values. Therefore, in this work, we want to construct a caching system fit for the storage of exactly these values in order to reduce recalculation overload on the one hand and memory (RAM) requirements on the other hand.

We start off with the hypothesis that not all calculated similarity values are equally useful. So when the recommendation values for all user-item pairs are calculated, in the end some user similarity values will

have been used more than others.

To gain insight into the distribution of the usage of user-user similarity values, we set up an experiment using the MovieLens dataset[1]. This dataset has been a popular tool for many researchers of recommender systems because of its high density (Park et al., 2006) and straightforward domain (i.e., movies). In our experiments we used the 100K dataset, which comprises 100K ratings of 943 users on 1682 movies. More details about this dataset can be found in recommender system literature.

We calculated the recommendation value for each user-item pair in the dataset (without caching) and kept track of how frequent every user-user similarity was used. User similarity (with Pearson correlation) is a symmetric relationship and so the total number of calculated user-user couples can be defined as $\frac{943*942}{2}$ (self similarities are not taken into account). Fig. 1 shows every one of these couples on the x-axis and their corresponding usage frequency on the y-axis.

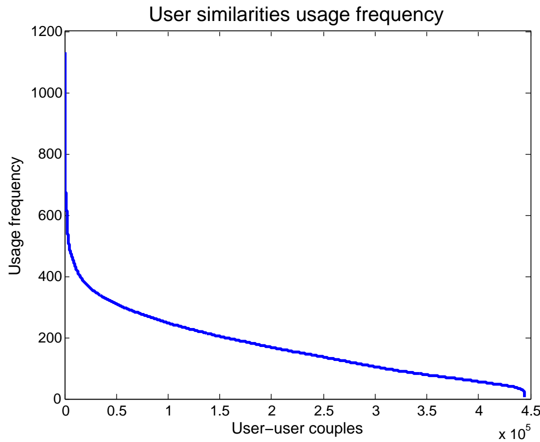

Figure 1: Every user-user couple in the system with its corresponding usage frequency. Couples are ordered by descending frequency.

Since Fig. 1 does not show a horizontal flat line but rather a long tail curve, we find our hypothesis confirmed: Some user similarities are more frequently used than others in the recommendation calculation process. The question now remains on how we can use this knowledge to our advantage. In the previous section the problem of limited RAM was already mentioned. Because of this limitation it may be impossible to simply compute all user-user similarities and keep them available in memory during the recommendation calculation. So if we can store only a limited amount of pre-calculated similarity values in memory, it may be a good idea to make sure the most interesting values are stored. If we define most interesting as 'most used' then we need a way of predicting how much a given user-user similarity value will be used throughout the recommendation process.

The usage frequency of user similarity values will be largely dependent on the number of ratings provided by each user. Users with a large number of ratings may show overlap (rated the same items) with more users and their similarity will consequentially be needed more often. To examine the correlation of the number of ratings of users with their similarity usage throughout the recommendation process, we set up an experiment with three simple usage prediction formulas (i.e., min, max, and sum). For each user-user similarity value we tried to predict its usage by applying each of the aggregation operators in Table 1 on the number of ratings of both users.

Aggregation operators
$minimum\ number\ of\ ratings(user_x,\ user_y)$
$maximum\ number\ of\ ratings(user_x,\ user_y)$
$sum\ number\ of\ ratings(user_x,\ user_y)$

Table 1: Aggregation operators to be applied to the number of ratings of a user-user pair in order to predict the frequency usage of the similarity value throughout the recommendation process.

Fig. 2 plots the prediction formulas applied to all user-user similarity pairs in the system together with the empirically measured actual usage value. As expected, we can see a correlation between the total number of ratings of a user-user pair and its usage frequency. The *maximum* operator seems to be the best estimation operator but has still has a far from perfect accuracy.

To improve this accuracy, we looked into deterministically computing the usage frequency instead of predicting it in a heuristic way. We considered the abstract user similarity pair $(u_x,\ u_y)$. This similarity may be used while calculating the recommendation pair $(u_x,\ i)$ with i an item that u_x has not rated. To determine the recommendation value of $(u_x,\ i)$, user similarities of u_x are needed for every user that has rated item i (see UBCF algorithm in Section 2). So the similarity pair $(u_x,\ u_y)$ will be calculated a number of times equal to the number of items that u_y has rated, but that are not rated by u_x. Since the user similarity has symmetric properties, the user similarity of $(u_x,\ u_y)$ and $(u_y,\ u_x)$ will be the same, and both should be taken into account. To conclude we can state that the number of times a user similarity pair $(u_x,\ u_y)$ will be used during the UBCF recommendation process, will be equal to the sum of the number of items rated by u_y but not by u_x and the number of items rated by u_x but not by u_y. We can reformulate this as the car-

[1] http://www.grouplens.org/node/73

dinality of the *inverse intersection* of the sets of rated items by u_x and u_y as shown in Fig. 3.

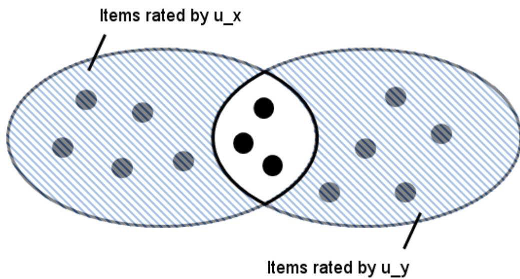

Figure 3: Venn diagrams indicating the items rated by users u_x and u_y. The cardinality of the inverse intersection corresponds to the usage frequency of the similarity value (u_x, u_y).

In the next sections we experiment with a caching system that incorporates this usage frequency knowledge and measure how it affects overall performance.

4 CACHING ALGORITHMS

To measure the impact of caching on the performance of the UBCF algorithm, we compare two different caching strategies. On the one hand we work with the existing LRU (Least Recently Used) caching principle and on the other hand we present our self-designed 'SMART' cache.

The LRU caching principle is a commonly used caching system where entries that have been used the least recently will be overwritten when the cache is full. This approach follows the temporal locality principle that recently requested data has a high probability to be requested again in the near future (Qasim, 2011). A disadvantage of LRU is that only the time of the data access is considered and not the frequency. Therefore we devised a 'SMART' cache which takes this data frequency information into account.

The SMART cache is a type of priority cache that incorporates information about how much an entry will be accessed throughout the program life cycle. Every cache entry is associated with a priority that reflects the number of times the entry will be accessed. When the cache is full, a new entry with a larger priority (i.e., predicted number of accesses) will overwrite an existing one with the lowest priority.

5 EXPERIMENTS AND RESULTS

The first thing to measure is the performance of the UBCF algorithm when using either the LRU or SMART caching strategy. As baseline we considered the UBCF algorithm implementation without a cache. In that case no user similarities are stored and they have to be recalculated every time they are used throughout the recommendation calculation. We compared the execution time of this baseline algorithm to the UBCF algorithm with an LRU cache or SMART cache for storing the user similarities during execution. Recommendation values (predicted ratings) were calculated for all user-item pairs in the MovieLens (100K) dataset. The experiment was repeated for cache sizes of 20%, 40%, 60%, 80% and 100%. This cache size is expressed as a percentage of the total number of user-user similarity values (which is $\frac{n*(n-1)}{2}$ for n the number of users in the system). A cache size of 100% therefore indicates that all user-user similarity values can be stored in memory and no values need to be recalculated. Fig. 4 shows the results in terms of speedup (i.e., execution time reduction compared to the baseline) for the different settings.

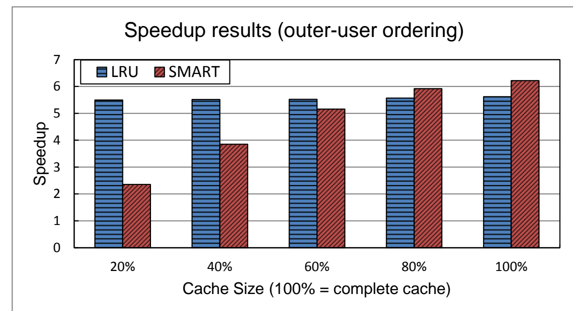

Figure 4: Speedup results for the LRU and SMART caching approaches towards the no-caching baseline for different cache sizes.

The simple LRU caching principle performed considerably better than the SMART system for the low cache sizes (20% and 40%). When the cache size grows, this difference decreases, and with cache sizes nearing 100%, the SMART approach overtakes LRU in terms of speedup towards the baseline. Interestingly, the performance of the LRU system remains stable between speedup values of 5 and 6 (in comparison to the SMART system) for the varying cache sizes. This seems to indicate that the LRU caching system has an optimal cache size (where performance gain saturates) below the 20% limit.

We repeated the experiment using only the LRU cache and tested with smaller cache stepsizes. Fig. 5 shows the results of an interesting range for cache sizes between 0.1% and 0.6%. Zoomed in on that range, the saturation effect of the speedup towards the baseline is clearly visible. While the SMART caching approach seemed to linearly improve with an increas-

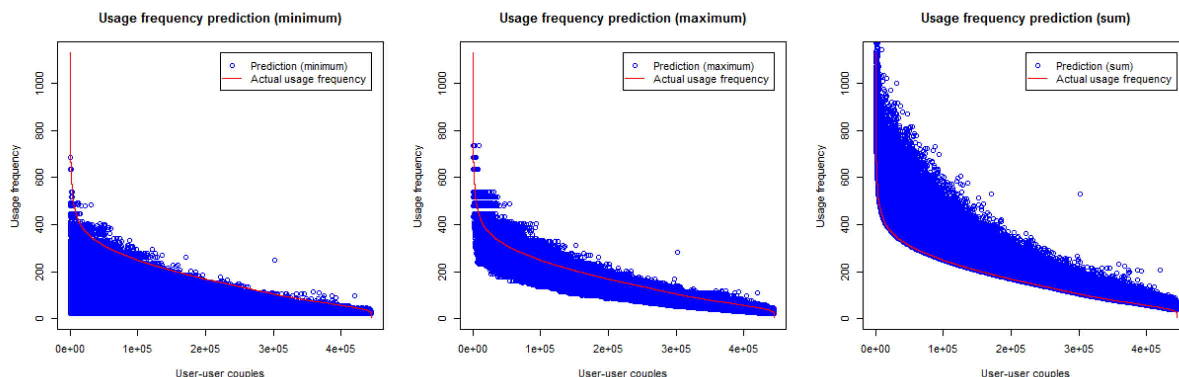

Figure 2: Prediction of the usage frequency of user-user similarity values by means of three aggregation operators together with the actual empirically measured usage frequency.

ing cache size, the LRU reaches a saturated maximum speedup value at a cache size between 0.2% and 0.3%. This range seems to correspond with the amount of user similarities associated with one user. For our dataset, there are 943 users and so 942 $(n-1)$ user similarities per user. To store 942 user similarities in the cache, a cache size of 0.21% would be needed. This 0.21% corresponds to the optimal cache size in the range 0.1% and 0.6%.

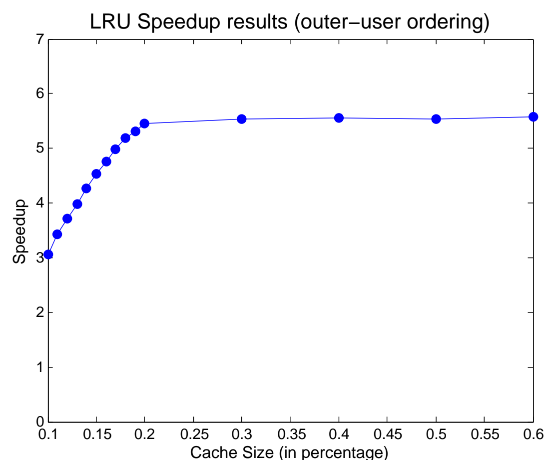

Figure 5: Speedup results for the LRU cache with smaller cache sizes to reveal the performance gain saturation point.

The performance of a cache depends partly on the order in which new entries are provided, therefore it is interesting to see how the 2 caching systems perform under altered job execution orderings. The order in which the user-user similarity values will be inserted in the cache depends greatly on how the UBCF algorithm loops over the different user-item pairs. In our reference implementation (see pseudocodefragment in Section 2) we calculate the recommendation value for every user for every item in that specific order. So the user-item pairs will be sequentially $(u_1,\ i_1)$, $(u_1,\ i_2)$, ... $(u_1,\ i_{n_i})$, $(u_2,\ i_1)$, $(u_2,\ i_2)$, ... $(u_2,\ i_{n_i})$, ... $(u_{n_u},\ i_{n_i})$ for n_u and n_i being respectively the total number of users and items. Since every user is handled sequentially, many similarities will be re-used in short intervals. This behavior is exactly what the LRU caching approach takes advantage of and would explain its success compared to the SMART cache.

To study the impact of the job order on the caching performance, the experiment was re-run with an altered job ordering. To change the order in which similarities would be inserted in the cache, we simply switched the first two lines of the pseudocodefragment. Because of the switch, user-item pairs are processed iterating over the items first. We refer to this approach as the ‘outer-item’ order (as opposed to the ‘outer-user’ order, which we started off with). The corresponding user-item pairs will now be sequentially $(u_1,\ i_1)$, $(u_2,\ i_1)$, ... $(u_{n_u},\ i_1)$, $(u_1,\ i_2)$, $(u_2,\ i_2)$, ... $(u_{u_n},\ i_2)$, ... (u_{n_u}, i_{n_i}). Fig. 6 shows the speedup results for this altered scenario.

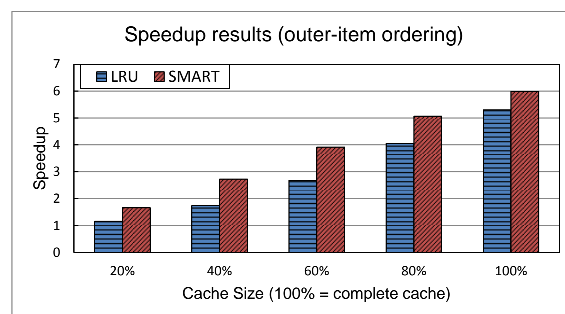

Figure 6: Speedup results for the LRU and SMART caching approaches (for different cache sizes) with a reversed user-item pair handling execution ordering.

As expected, the performance of the LRU cache is greatly reduced and for every cache size the SMART cache shows faster speedup values. Although the SMART cache is faster than the LRU method, it is

still somewhat slower than the speedup values of the SMART cache in the outer-user ordering situation. In general the speedup values are below those of the outer-user ordering, but the SMART cache seems less affected (than LRU) by the change in ordering. The LRU cache shows no indication of saturation as was the case for the outer-user situation.

We repeated the experiment with a third alternate job ordering situation where all user-item pairs were processed in random order. The aggregated results of multiple runs turned out to be almost identical to the outer-item results.

6 CONCLUSIONS

In this work we set out to find caching strategies that allow in-memory user-based collaborative filtering algorithms to store intermediate user-user similarity results. First, we showed that user similarities are not equally important, as some are used considerably more than others during the recommendation calculation process. We tried predicting this usage frequency upfront by applying aggregation operators on the number of ratings, but ultimately succeeded in accurately calculating this value by determining the cardinality of the 'inverse intersection' of the set of rated items.

We then presented two caching strategies: a basic LRU (least recently used) cache and a novel SMART caching approach which acted like a priority cache that incorporated the knowledge about usage frequency of user-user similarities. A number of experiments were run on the MovieLens dataset to compare the performance (execution time speedup) of each of these caches against a no-cache baseline and under varying cache sizes and job execution orderings.

Our results showed that the order in which user-item recommendation values are calculated can dramatically impact the LRU cache performance and therefore also the total execution time. Optimal results were obtained when calculating the recommendation values of each user for every item sequentially before moving on to the next user (outer-user strategy). The LRU-enhanced UBCF algorithm performed between 5 and 6 times better in that situation than the no-cache baseline and required a cache size of only 0.2% (vs. the SMART approach which required a cache size of 60% to obtain similar results). For a random job (and outer-item) execution ordering on the other hand, the SMART approach came out best in terms of stability and performance.

Although this work focussed mainly on in-memory algorithms, these results may as well be useful for other situations where caching strategies can be applied to user-based collaborative filtering algorithms (e.g., caching similarity values to reduce database access).

7 FUTURE WORK

As future work we would like to investigate the generalizability of the obtained results to other datasets and recommendation algorithms. We also intend to improve results in terms of speedup by further refining the job execution ordering and involving other cache algorithms like LFU and ARC.

ACKNOWLEDGEMENTS

The described research activities were funded by a PhD grant to Simon Dooms of the Agency for Innovation by Science and Technology (IWT Vlaanderen).

REFERENCES

Herlocker, J., Konstan, J. A., and Riedl, J. (2002). An empirical analysis of design choices in neighborhood-based collaborative filtering algorithms. *Inf. Retr.*, 5(4):287–310.

Jannach, D., Zanker, M., Felfernig, A., and Friedrich, G. (2011). *Recommender Systems An Introduction*. Cambridge University Press.

Park, S.-T., Pennock, D., Madani, O., Good, N., and DeCoste, D. (2006). Naïve filterbots for robust cold-start recommendations. In *Proc. ACM SIGKDD Conf. on Knowledge discovery and data mining (KDD 2006)*, pages 699–705, New York, NY, USA.

Peralta, V. (2007). Extraction and integration of movielens and imdb data. Technical report, Technical Report, Laboratoire PRiSM, Université de Versailles, France.

Qasim, U. (2011). *Active Caching For Recommender Systems*. PhD thesis, New Jersey Institute of Technology, New Jersey.

Qasim, U., Oria, V., fang Brook Wu, Y., Houle, M. E., and Özsu, M. T. (2009). A partial-order based active cache for recommender systems. In *Proc. ACM Conf. Recommender systems (RecSys 2009)*, pages 209–212.

Resnick, P., Iacovou, N., Suchak, M., Bergstrom, P., and Riedl, J. (1994). Grouplens: an open architecture for collaborative filtering of netnews. In *Proc. ACMF Conf. on Computer supported cooperative work (CSCW '94)*, pages 175–186, New York, NY, USA. ACM.

Seth, S. and Kaiser, G. (2011). Towards using cached data mining for large scale recommender systems. In *Proc. Conf. Data Engineering and Internet Technology (DEIT 2011)*.