Speeding up Collaborative Filtering with Parametrized Preprocessing

Victor Anthony Arrascue Ayala Anas Alzoghbi Martin Przyjaciel-Zablocki
Alexander Schätzle Georg Lausen

Department of Computer Science
University of Freiburg
Georges-Köhler-Allee 051, 79110 Freiburg, Germany
arrascue|alzoghba|zablocki|schaetzle|lausen@informatik.uni-freiburg.de

ABSTRACT

Collaborative filtering (CF) aims at producing recommendations for a user based on other users of similar taste, their *k-neighbors*. Since the computation of the neighborhood dominates the complexity of CF for a large number of users and ratings, this is done off-line in most commercial systems. As more and more systems allow users to continuously rate resources, neighborhoods are rapidly outdated and lose accuracy. Hence, neighborhoods have to be updated more often but traditional approaches do not meet the speed requirements. Our major contribution in this paper is to present a technique to split the computation of the neighborhood into an off-line and on-line task. This enables the system to speed up the on-line computation time up to 97% in relation to the time required by the state-of-the-art approach, as our experiments on the MovieLens dataset demonstrate.

Categories and Subject Descriptors

H.3.3 [**Information Search and Retrieval**]: [Information filtering]; H.3.4 [**Systems and Software**]: [Performance evaluation (efficiency and effectiveness)]

Keywords

Rec. Systems, RecSPARQL, Collaborative Filtering

1. INTRODUCTION

Collaborative filtering (CF) is a fundamental approach to deliver recommendations [1]. The first step and pre-requisite for *user-based CF* is to find for a user u the *top-k* most similar users, a peer referred to as the *neighborhood* of u. The goal is to predict u's degree of preference for an item i based on the ratings given by their neighbors. One of the main problems is that when the number of users and ratings in the system is significantly large, computing the neighborhoods for all users is costly. The *item-based* variant, which focuses on finding similar rated items, is more accurate when the number of users is much greater than the number of items, but it also suffers from this problem of high-dimensionality. For this reason, many systems compute the neighborhoods completely off-line, whereas the recommendations are computed on-line. However, off-line computed neighborhoods do not meet the requirements of the big data era for two reasons [8].

First, more and more systems are able to massively capture users' behavior and preferences. In this scenario, the time span in which a user-based neighborhood remains valid is too short because new users, potential neighbor candidates, and their preferences continuously join the system. For instance in domains like music users continuously listen to and rate songs and neighborhoods based on song ratings rapidly lose their validity. To keep neighborhoods up-to-date it is possible to recompute them after some interval of time, but the time required for computing the neighborhoods prevents this approach from being of much practical use in many cases.

Secondly, given that richer information is available, this can be used as a means to increase the quality of recommendations. Although a neighborhood based on user ratings is built out of neighbors which rate similarly, one might want to consider different kinds of data, preferences or behavior patterns. However, it is not possible to store off-line neighborhoods for each possible criteria adopted. Flexibility has been the focus of our research in previous work [3], in which we propose an extension of the SPARQL query language[1] to obtain recommendations from RDF graphs. There, the query writer can arbitrarily choose which features to consider for computing the neighborhood, and queries with different features can lead to a dynamic computation of the neighborhoods. In this scenario, an efficient evaluation of queries has posed a big challenge.

Being aware that no solution can tackle all of the above mentioned problems, we proposed a strategy to arbitrarily split the computation of k-neighbors into an off-line and on-line task which can temporarily satisfy the on-line time requirements, until a significant number of new users and ratings join the system. We adopt the Information Retrieval (IR) approach for computing CF [4, 7] as the baseline of

KDD'15, August 10 – 13, 2015, Sydney, Australia.

[1] http://www.w3.org/TR/rdf-sparql-query/

our work. Using inverted indexes to evaluate queries efficiently [12], it is possible to alleviate the scalability problem and to obtain a performance similar to that observed in ranking tasks [5].

Our paper is structured as follows: We formalize the problem in Section 2. In section 3 we present our approach. In section 4 we present the results of our experiments. Finally, we present the related work in section 5 and our conclusions in section 6.

2. PROBLEM STATEMENT

Let $U = \{u_1, u_2, ..., u_n\}$ be a set of users, $I = \{i_1, i_2, ..., i_m\}$ a set of items. The *recommender system* (RS) collects explicit ratings $r(u, i)$ from users $u \in U$ given to items $i \in I$. The value of a rating is an element of the set $S = \{s_1, s_2, ...s_l\}$ which is the rating scale used in the RS. Without loss of generality we assume S is a finite set. For example, for the five stars rating scale, in which half stars are not allowed, $S = \{1, 2, 3, 4, 5\}$. A *user profile* is represented as a vector of ratings:

$$\vec{v}_u =< r(u, i_1), r(u, i_2), ..., r(u, i_m) >$$

Let $V = \{\vec{v}_{u_1}, \vec{v}_{u_2}, ..., \vec{v}_{u_n}\}$ be the set of profiles of all users from U. The cosine similarity of two user profiles, $cos_{sim}(\vec{v}_{u_x}, \vec{v}_{u_y})$, is defined as follows:

$$cos_{sim}(\vec{v}_{u_x}, \vec{v}_{u_y}) = \frac{\sum_{j=1}^{m} v_{u_x}[j] \cdot v_{u_y}[j]}{\|\vec{v}_{u_x}\| \cdot \|\vec{v}_{u_y}\|}, \qquad (1)$$

where $v_{u_x}[j]$ is the value of the vector at the j^{th}-dimension and $\|\vec{v}_{u_x}\|$ is the L2-norm of vector $\vec{v}_{u_x}$ and hence:

$$\|\vec{v}_{u_x}\| = \sqrt{\sum_{j=1}^{m} (v_{u_x}[j])^2}$$

Let *AllNB* be the problem of finding top k-neighbors for all users from U. The naive approach consists in computing $cos_{sim}(v_{u_x}, v_{u_y})$ by iterating over all pairs of users and ranking the results for each user. However, this leads to a quadratic complexity in the size of $\mid U \mid$ and for a large number of users this approach is not feasible. Moreover, given that ratings are very sparse, a model based on vectors of m dimensions, the number of items, is not used in practice.

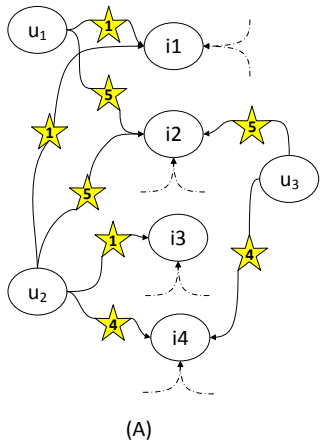
Let L_i be a *traditional inverted list* of an item $i \in I$. L_i's elements are pairs of users and ratings assigned to item i:

$$L_i : \{(u_x, r(u_x, i)), (u_y, r(u_y, i)), ..., (u_w, r(u_w, i))\}$$

To facilitate the retrieval of the lists, these are indexed using the identifier of an item as the key. Figure 1 shows an example (A) along with its traditional inverted lists (B).

2.1 K-neighbors based on inverted lists

Modeling the problem using traditional inverted lists allows one to compute the k-neighbors problem and to reach the state-of-the-art in terms of efficiency and scalability [5]. The algorithm *K-neighbors*, which is based on [7], uses this kind of inverted lists to compute the top-k neighbors for a single user, the *active user*.



Traditional indexing

L_{i1}: {(u_1,1),(u_2,1)}
L_{i2}: {(u_1,5),(u_2,5),(u_3,5)}
L_{i3}: {(u_2,1)}
L_{i4}: {(u_2,4),(u_3,4)}

(B)

Scaled indexing

$L_{i1,1}$: {(u_1,1),(u_2,1)}
$L_{i2,1}$: {(u_1,5),(u_2,5),(u_3,5)}
$L_{i3,1}$: {(u_2,1)}
$L_{i4,1}$: {(u_2,4),(u_3,4)}
$L_{i1,2}$: {(u_1,2),(u_2,2)}
$L_{i2,2}$: {(u_1,10),(u_2,10),(u_3,10)}
$L_{i3,2}$: {(u_2,2)}
$L_{i4,2}$: {(u_2,8),(u_3,8)}
...
$L_{i1,5}$: {(u_1,5),(u_2,5)}
$L_{i2,5}$: {(u_1,25),(u_2,25),(u_3,25)}
$L_{i3,5}$: {(u_2,5)}
$L_{i4,5}$: {(u_2,20),(u_3,20)}

(C)

Figure 1: (A) Ratings example; (B) Traditional indexing; (C) Scaled indexing.

K-neighbors: cos. similarity based on trad. inverted lists

INPUT: $\vec{v}_{u_x}$ the profile of active user u_x with ID x;
n the number of users; k, the number of neighbors;
$L2NORMS$ array of size n, L2-norms of all $\vec{v}_u$ profiles;
OUTPUT: $NEAR$ an array of size k, the k-nearest neighbors.
1: Initialize and allocate COS and $NEAR$
2: $NUM = Numerator1(\vec{v}_{u_x}, n)$
#Normalization of NUM
3: for each j index of NUM do
4: $COS[j] = NUM[j]$ / $(L2NORMS[x] * L2NORMS[j])$
5: end for
6: Sort $COS[j]$ values
7: Insert the highest k values $(u_y, COS[y]))$ to $NEAR$
8: return $NEAR$

The size of the neighborhood k is given as input. For the computation, the L2-norm of each user vector is precomputed and stored in the static array *L2NORMS*. *K-neighbors* uses the algorithm *Numerator1* to compute the NUM array, which contains in each cell the dot product between the active user and each other user in the dataset, i.e. the numerators of formula (1). The NUM array is then normalized to obtain the cosine similarities, which are stored in the array COS. Finally, COS is sorted and the first k cells, which contain those users with the highest similarity scores are then used to build the $NEAR$ array.

Numerator 1: computes the numerator of the cosine similarity between u_x and all other users

INPUT: $\vec{v}_{u_x}$ the profile of active user u_x with ID x;
n the number of users.
OUTPUT: NUM array of size n that stores the dot product of the cos_{sim} (numerator) between $\vec{v}_{u_x}$ and each other user vector.
1: Initialize and allocate NUM
2: for all $r(u_x, i) \in \vec{v}_{u_x}$ do
3: retrieve inv. list $L_i = \{(u_z, r(u_z, i)), ..., (u_w, r(u_w, i)))\}$
4: for each user u_y and rating $r(u_y, i) \in L_i$ do
5: $NUM[y] = NUM[y] + r(u_x, i) * r(u_y, i)$
6: end for
7: end for
8: return NUM

The complexity of *K-neighbors* clearly depends on two factors: (1) the number of ratings of the active user, which determines the number of retrieved lists and (2) the overall number of elements of all lists. At the same time, the number of elements in their lists depends on the popularity of the rated items. As we will demonstrate in detail in the

next section, it is possible to obtain the desired on-line performance by specifying an off-line task which compacts some of the ratings in each user profile, and together with these, the corresponding retrieved lists (line 3 of *Numerator1*).

3. SCALED INDEXING

A good off-line approach reduces the space in which the on-line algorithm performs so that this can run faster. Our off-line approach first reduces the number of ratings in each user profile using a special kind of rating as the criteria, i.e. the more ratings of this kind a user has, the more information is compacted off-line. The result is a single artificial rating which replaces them all. Secondly, it compacts the corresponding set of lists. In this way, the on-line algorithm has to run only on the reduced profiles. This is explained in detail in the next section.

For the second step, it is necessary to retrieve all the lists corresponding to the compacted ratings and to make a single list out of them. The problem with traditional inverted list is that these are built for each item without taking the rating into account. Therefore, we propose a new indexing technique. Let $L_{i,s}$ be a scaled inverted list, where $i \in I$ and $s \in S$. The elements of the list are:

$$L_{i,s} : \{(u_x, r(u_x, i) * s), ..., (u_w, r(u_w, i) * s))\} = \\ L_{i,s} : \{(u_x, d(u_x, i)), ..., (u_w, d(u_w, i))\},$$

which corresponds to the previously defined list L_i in which each element of the list is scaled by s. The figure 1(C) shows the scaled inverted lists for the example in (A).

This means that instead of having m traditional inverted lists, the system maintains $m \times \mid S \mid$ scaled inverted lists. This approach has however several advantages. The first advantage is that it is possible to modify *Nominator1* into a new version which doesn't require multiplications (line 5):

Numerator 2: modification of Numerator 1 to support scaled indexes

```
...
2: for all r(u_x, i) ∈ v⃗_{u_x} do
3:   retrieve L_{i,r(u_x,i)} = {(u_z, d(u_z, i)), ., (u_w, d(u_w, i))}
4:   for each user u_y and rating d(u_y, i) ∈ L_{i,r(u_x,i)} do
5:     NOM[j] = NOM[j] + d(u_y, i)
6:   end for
7: end for
...
```

The second advantage is that it is possible to define a merge operation for the off-line task to compact a set of lists. Let $R = \{L_x, ...L_z\}$ be a set of scaled inverted lists. The merge operator returns a so-called *materialized inverted list* M, whose elements are pairs $(u_x, d_{sum}(u_x))$, such that u_x is in at least in one list of R and $d_{sum}(u_x) = \sum_i d(u_x, i)$, i.e. the sum of all scores found for user u_x in all lists in R.

For instance, suppose that in figure 1(A) most of users have rated item i_1 with one star and i_2 with five stars. Then lists $L_{i_1,1}$ and $L_{i_2,5}$ are merged producing the materialized list $M = \{(u_1, 26), (u_2, 26), (u_3, 25)\}$. Merging scaled inverted lists can be done without knowing anything about users' profiles, whereas the elements of traditional lists have to be scaled by the user rating before these can be merged.

3.1 Top ratings and off-line computation

Scaled indexing provides an additional degree of freedom, which didn't exist in traditional inverted lists, i.e. it enables the defining of a merge operation independent of user profiles. Thereby, it is possible to implement the off-line approach which is based in the concept of *top ratings*, the criteria used to compact ratings in users' profiles. This is defined as follows. Let (i, s) be a pair composed of $i \in I$ and $s \in S$ and α the desired number of top ratings. Moreover, let $users : (i, s) \rightarrow \mathbb{N}$ be a function which counts the number of users in the dataset who rated i with s. For all $i \in I$ and all $s \in S$ it is possible to apply the function $users$ and sort the results by the returned values. The first α pairs (i, s) are the top ratings. In practice, top ratings can be easily obtained by executing a query against the dataset.

The off-line phase iterates over users. To solve the sparsity problem of representing user profiles with vectors, our recommender system stores for each user a set of ratings to easily retrieve their set of inverted lists. For each user u their set of ratings is then intersected with the set of top ratings. If the size of the intersection is at least two, all inverted lists $L_{i,r(u,i)}$ in the intersection are merged to create a new materialized list. For instance, suppose that in figure 1(A) $\alpha = 3$ and the top ratings are $(i_1, 1)$, $(i_2, 5)$ and $(i_4, 4)$. The initial set of ratings of each user is the following:

$$\text{For } u_1 : \{r(u_1, i_1) = 1, r(u_1, i_2) = 5)\}$$
$$\text{For } u_2 : \{r(u_2, i_1) = 1, r(u_2, i_2) = 5, r(u_2, i_3) = 1, r(u_2, i_4) = 4\}$$
$$\text{For } u_3 : \{r(u_3, i_2) = 5, r(u_3, i_4) = 4\}$$

Now consider user u_2. Three of his ratings $r(u_2, i_1) = 1$, $r(u_2, i_2) = 5$ and $r(u_2, i_4) = 4$ are top ratings. Therefore, a new inverted list M' is materialized by merging lists $L_{i_1,1}$, $L_{i_2,5}$ and $L_{i_4,4}$:

$$M' : \{(u_1, 26), (u_2, 42), (u_3, 41)\}$$

M' has to be retrieved from the set of ratings of u_2 and therefore his initial set of ratings is replaced by:

$$\text{For } u_2 : \{r(u_2, i_3) = 1, r(u_2, \{i_1, i_2, i_4\}) = 6.48\}$$

The artificial rating $r(u_2, \{i_1, i_2, i_4\})$ makes it possible to retrieve the correct materialized list using a new identifier[2] and the value of the rating. The value at which the artificial rating is set to is:

$$r(u_2, \{i_1, i_2, i_4\}) = \sqrt{r(u_2, i_1)^2 + r(u_2, i_2)^2 + r(u_2, i_4)^2} \approx 6.48$$

The new rating is not only important for the retrieval of the new list, but also keeps the L2-norm of user u_2 unchanged:

$$\text{Old profile of } u_2 : \sqrt{1^2 + 5^2 + 1^2 + 4^2} = 6.55...$$
$$\text{New profile of } u_2 : \sqrt{1^2 + 6.48^2} = 6.55...$$

This guarantees that the reduced profiles do not alter the results of the cosine similarity metric. Suppose that another user u_y is processed and that the intersection of top ratings is the same as for u_2. It is not necessary to recompute M'. Instead, the set of ratings of u_y is replaced by the artificial rating which retrieves that materialized list. This is the reason that considering top ratings is beneficial; these ratings and their combinations are likely to appear for many users.

[2]The identifier is made out of the old ratings. Imagine for instance that the top ratings are $(i_1, 1)$,$(i_1, 5)$,$(i_2, 5)$,$(i_2, 4)$,$(i_4, 4)$ and $(i_4, 1)$: the combination of $(i_1, 1)$, $(i_2, 5)$ and $(i_4, 4)$ produces the same value as $(i_1, 5)$, $(i_2, 4)$ and $(i_4, 1)$.

3.2 Trade-off

Merging the scaled inverted lists into materialized lists allows us to arbitrarily shorten the on-line computation time of the *AllNB* problem at the cost of memory consumption. We show this by considering the extreme cases with respect to parameter α. Let IL be the set of traditional inverted lists, SL and ML be respectively the scaled inverted lists and the materialized lists in our approach. When $\alpha = 0$, the number of scaled inverted lists $|SL| = |IL| \times |S| = m \times |S|$, whereas $|ML| = 0$, i.e. no materialized lists are generated. The time to solve the *AllNB* problem is ca. the same as for the traditional indexing approach, but more memory is required. However, the extra required memory is an affordable amount in most RS, where the rating scale is chosen to be simple, discrete and limited to just a few values and therefore $\mid S \mid$ is small. On the other hand, as α increases, more materialized lists are generated by merging more scaled inverted lists off-line, leaving less operations to the on-line phase and resulting in a fewer cost. When α is set equal to the number of all existing ratings in the system the off-line phase will merge all the scaled inverted lists for each user into one materialized list that contains the results of the dot products between the active user vector and all other users' vectors. This leads to $|ML| = n$ materialized lists, but even in this case the amount of memory required is bounded. Moreover, the larger the value to which α is set, the more ratings are removed from users' profiles: if $\mid I_u \mid$ is the size of intersected top ratings for user u, $\mid I_u \mid -1$ ratings are removed from his profile. Different values of α have different impacts in performance as the analysis of the experiments shows. Although a larger number of top ratings always results in a shorter computation time for the on-line task, the speed up is not necessarily uniform.

3.3 New users and new ratings

The pre-processing of top ratings helps in reducing the computation time for the *AllNB* problem. However, new users and new ratings progressively slow down the computation time, because the number of elements in the scaled inverted lists increases again. Although a significant number of users and ratings are required to have a big change in performance, a new off-line processing might be necessary at some point in time to speed up the on-line request.

The strategy chosen to deal with this problem is to automatically run the off-line procedure when the set of top ratings changes in the system for the same parameter α, i.e when at least one top rating is replaced by a non-top rating. To achieve this, artificial ratings have to be split again into single ratings to search for an intersection with the new set of top ratings. If the intersection remains the same, then the set of ratings of that user previous to that split remains unaltered. Otherwise, a new materialized list is created, whereas old materialized lists are removed if these are no longer required by any user. Updates in existing ratings are not allowed in our system, but this is also not likely to happen in real scenarios.

4. EXPERIMENTS

The experiments were carried out on the following MovieLens datasets:

- ML100k: the smallest data set. It consists of 100,000 ratings from 943 users on 1,682 movies.
- ML1M: it contains 1,000,209 ratings from ca. 6,040 users and 3,900 movies.

	ML100k	**ML1M**
trad. indexing (ms)	1,482	108,311
scaled indexing (ms)	1,230	116,604

Table 1: Time required (ms) for computing *AllNB* on-line with $\alpha = 0$

The rating scale is based on 1-5 stars. Originally, these datasets do not contain users who rated fewer than 20 movies. If a user rates a movie more than once, only the last rating is used. The neighborhood size is set to 20, but this has only a minimal impact on the computation time, since this only determines the size of the queue in which the results are stored but all lists have to be merged anyway.
We run the experiments on a machine with Ubuntu 12.04.5 LTS, a processor Intel(R) Xeon(R) CPU X5667@3.07GHz with 8 cores and 32 GB of RAM. The initial and maximum memory allocation pool for the Java Virtual Machine (JVM) are set respectively to 4 GB and 30 GB.

The recommender system is implemented on top of the Sesame[3] triple store, but it could have been easily integrated on top of any data layer as it simply queries the data interface to obtain the ratings. In our case this is achieved by means of SPARQL queries. The RDF repository stores the data in-memory. We opt for an in-memory storage in order to speed up the query evaluation and therefore the time required for building the inverted lists, even if this reduces the amount of memory available for the neighborhoods' computation. We implemented both approaches for building the inverted lists, i.e. the traditional indexing and scaled indexing approach. Both kinds of lists are also maintained in-memory and their indexes used for the retrieval are stored in a hash map. In the case of the traditional inverted lists the key is the ID of an item, whereas for scaled indexing a hash function returns a key based on both the item ID and the rating. For each user their set of ratings is also stored in-memory. In this way, each corresponding list can be rapidly retrieved as described in the *K-neighbors* algorithm.

On each experiment the neighborhoods for all users (*AllNB*) are computed. For scaled indexing in addition to the input data, the number of top ratings to be pre-processed, α, has to be specified. We incrementally increase α on each new experiment. When α reaches ca 10% of the ratings, we proceed with a larger dataset. We repeat one experiment several times to obtain reliable execution times. For ML100k the number of repetitions for each experiment is 10, for ML1M it is 5. We then compute the avg. computation time for the off-line and on-line tasks. We also keep track of the number of built inverted lists, the number of removed ratings from the users' profiles and the memory usage. These metrics are reported for chosen values of α in table 2.

The results obtained for $\alpha = 0$ are shown in table 1. Although the scaled indexing approach does not require multiplications, this is advantageous only for the smallest dataset. Since the hash map which stores the scaled inverted lists is five times larger than that of the traditional indexing, the slightly worse performance is due to the more expensive retrieval of lists: probing a list which is based not only on

[3]Version: 2.7.14, http://rdf4j.org/

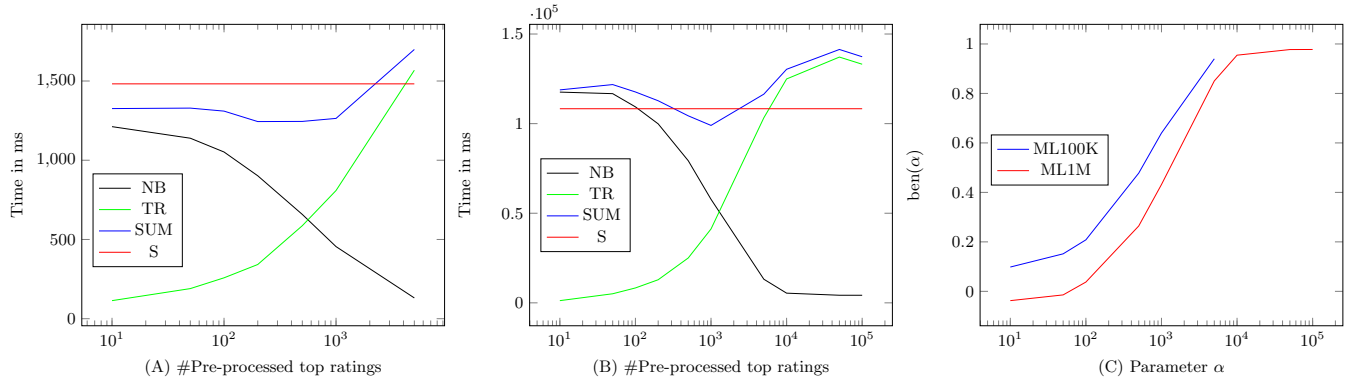


Figure 2: Computation times on (A) ML100k and (B) ML1M of AllNB on-line (NB), off-line top ratings pre-processing (TR), NB + TR (SUM), state-of-the-art (S). (C) shows the benefit of the parameter α.

the ID of an item but also on the scale factor s requires a hash function based on more operations. Moreover, saving a significant number of multiplications for the *AllNB* problem does not necessarily lead to a large impact in performance. For instance, a modern CPU can execute ca. 4 double floating point operations per cycle. The removed multiplications should be in the order of millions just to obtain one second of improvement.

Figures 2(A) and (B) depict the computation time required for computing the neighborhoods on-line (NB) in the first two datasets. Since the off-line procedure is only required once, the time needed for computing the neighborhoods on-line can be drastically reduced. For instance for ML1M, the computation with the traditional approach required ca. 108,000 msecs, but with scaled indexing and $\alpha = 10,000$ (ca 1% of the overall ratings), it was possible to reduce this to ca. 5,400 msecs, which corresponds to 5% of the original time. Interestingly, when α was increased from this point on, the benefit of pre-processing additional top ratings does not pay off well. For instance, for $\alpha = 100,000$ (ca 10% of the overall ratings), the time is reduced to ca. 4,200 msecs, i.e 3.8% of the original time.

Figures 2(A) and (B) also illustrate the cost of the off-line procedure to process top ratings (TR) in terms of computation time. The sum of the off-line and on-line computation time for the neighborhoods (SUM) is worse than time required by the traditional approach. This is due to the work of sorting the additional materialized lists by user ID, which is a pre-requisite for merging lists efficiently. A larger hash map has also to be resized more times.

Figure 2(C) shows the benefit of each parameter α. This is calculated as follows:

$$ben(\alpha) = 1/2 * \left(\frac{rm}{rat} + \frac{t_{ti} - t_{si}}{t_{ti}} \right)$$

where rm is the overall number of removed ratings in the dataset, rat is the overall number of ratings in the system. On the second member of the sum we have t_{ti} and t_{si}, which are respectively the on-line computation time required by the traditional indexing and scaled indexing approach.

The figure suggests that for both datasets, ML100k and ML1M the benefit rises up when $\alpha > 10^2$. As α increases, the benefit grows for ML100k constantly, whereas for ML1M the benefit start to decrease when $\alpha = 10^4$. The overall impact in performance for *AllNB* depends on the ratings distribution, but even with a uniform distribution the results have a large impact in performance.

α	0	10	100	1000	10000
ML100k					
NB (ms)	1230	1212	1052	455	–
# scal. lists	8410	8410	8410	8410	–
# mat. lists	0	295	920	943	–
# rem. rat.	0	1410	12607	58657	–
Mem. (MB)	5.72	8.19	14.77	15.49	–
ML1M					
NB (ms)	116604	117590	109351	57785	5408
# scal. lists	18530	18530	18530	18530	18530
# mat. lists	0	762	5824	6034	6040
# rem. rat.	0	10546	84761	397219	959488
Mem. (MB)	57.23	99.47	412.02	448.63	457.09

Table 2: Metrics for chosen parameters α

5. RELATED WORK

A traditional way of classifying CF algorithms is by dividing them into memory-based and model-based algorithms [1]. In this work we adopt a memory-based approach, because models lose accuracy rapidly and are not able to satisfy the quality requirement when new users and ratings continuously join the system. Approaches based on Information Retrieval are well researched and have been able to satisfy the performance and scalability needs for search in the web. Therefore much effort has been undertaken to bridge these two paradigms [5, 13]. In [7] Cöster et al. propose the use of a disk-based inverted file structure. The authors first describe different algorithms in a suitable form for information retrieval (IR) and then combine this with two IR termination heuristics, *Quit* and *Continue*. These algorithms are not suitable for our proposed indexing strategy, mainly due to the metrics, e.g. Pearson Correlation, which use the average of users' ratings, for which a re-engineering of our approach might be possible. In [4] Bayardo et al. propose a new indexing strategy for solving the all-pairs similarity search problem, which is a generalization of the k-neighbors problem. In this approach indexes are built dynamically and they use a threshold t to reduce the amount of information indexed and to compute bounds for the dot product.

Although the work alleviates the scalability problem, the all-pairs algorithm has to be fully computed on-line. Inverted index structures can be used to implement clustering strategies too. In [2] Altingovde et al. design a cluster-skipping inverted index to store clusters. Individual search is thus enabled in single clusters without too much degradation in the efficiency. This allows them to reduce the neighborhood formation time by up to 60%. However, it is not possible to have a parametrized reduction. Our approach is similar to incremental approaches such as the strategy proposed by [10] Papagelis et al., designed to deliver results faster while keeping the same accuracy in the recommendations. Additionally, more and more effort has been expended to implement these strategies in new processing paradigms, such as stream processing systems [6, 8], map-reduce [14, 9], or even in specific-purpose hardware [11]. There are many other approaches based on approximations, but in most of the cases these have a negative impact on the quality of the recommendations.

6. CONCLUSION AND FUTURE WORK

The efficiency of the neighborhoods computation is one of the weakest point of CF-based recommendations. We propose, therefore, a strategy to split it into an off-line and on-line task. The goal of the off-line task is to reduce the sizes of the users' profiles by compacting a set of ratings considered to be top ratings. This triggers the merging of sets of lists which materializes partial results. The off-line process is parametrized by the number of top ratings α. Results obtained in our experiments show that it is possible to reduce the on-line computation time up to 97% with respect to the traditional approach. We therefore conjecture that top ratings are a good strategy to set the workload of a partial computation of k-neighbors. The off-line computation is performed at the cost of memory usage. However, to overcome memory deficiencies in larger datasets, we might in the future store part of the inverted lists, e.g. the materialized lists, in disk, avoiding the storing of all lists in-memory as in our experimental setting. Our approach can be easily extended to item-based CF for which we need to assess if a symmetric definition of top ratings, *top raters*, would be equally useful in reducing the item profiles. As another future direction of our research, we would like to extend our approach to other CF-IR compatible algorithms and metrics and to investigate the impact on the quality of recommendations. For instance, in *Inverse User Frequency*, items with a high number of ratings are penalized with lower weights. A similar approach could be easily integrated into our method, because each materialized list is already a combination of popular items whereas the weight could be chosen to depend on the number of merged lists a materialized list is made of. Moreover, the use of pruning heuristics such as Quit or Continue or even compression techniques are also applicable in our approach and could help to further speed-up the computation time.

7. REFERENCES

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

[2] I. S. Altingövde, Ö. N. Subakan, and Ö. Ulusoy. Cluster searching strategies for collaborative recommendation systems. *Inf. Process. Manage.*, 49(3):688–697, 2013.

[3] V. A. A. Ayala, M. Przyjaciel-Zablocki, T. Hornung, A. Schätzle, and G. Lausen. Extending sparql for recommendations. In *Proceedings of Semantic Web Information Management on Semantic Web Information Management*, SWIM'14, pages 1:1–1:8, New York, NY, USA, 2014. ACM.

[4] R. J. Bayardo, Y. Ma, and R. Srikant. Scaling up all pairs similarity search. In *Proceedings of the 16th International Conference on World Wide Web, WWW 2007, Banff, Alberta, Canada, May 8-12, 2007*, pages 131–140, 2007.

[5] A. Bellogín, J. Wang, and P. Castells. Bridging memory-based collaborative filtering and text retrieval. *Inf. Retr.*, 16(6):697–724, 2013.

[6] B. Chandramouli, J. J. Levandoski, A. Eldawy, and M. F. Mokbel. Streamrec: a real-time recommender system. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2011, Athens, Greece, June 12-16, 2011*, pages 1243–1246, 2011.

[7] R. Cöster and M. Svensson. Inverted file search algorithms for collaborative filtering. In *SIGIR 2002: Proceedings of the 25th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, August 11-15, 2002, Tampere, Finland*, pages 246–252, 2002.

[8] Y. Huang, B. Cui, W. Zhang, J. Jiang, and Y. Xu. Tencentrec: Real-time stream recommendation in practice. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, pages 227–238, 2015.

[9] W. Lu, Y. Shen, S. Chen, and B. C. Ooi. Efficient processing of k nearest neighbor joins using mapreduce. *CoRR*, abs/1207.0141, 2012.

[10] M. Papagelis, I. Rousidis, D. Plexousakis, and E. Theoharopoulos. Incremental collaborative filtering for highly-scalable recommendation algorithms. In *Foundations of Intelligent Systems, 15th International Symposium, ISMIS 2005, Saratoga Springs, NY, USA, May 25-28, 2005, Proceedings*, pages 553–561, 2005.

[11] H. Shu, R. Yu, W. Jiang, and W. Yang. Efficient implementation of k-nearest neighbor classifier using vote count circuit. *IEEE Trans. on Circuits and Systems*, 61-II(6):448–452, 2014.

[12] T. Strohman, H. R. Turtle, and W. B. Croft. Optimization strategies for complex queries. In *SIGIR 2005: Proceedings of the 28th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, Salvador, Brazil, August 15-19, 2005*, pages 219–225, 2005.

[13] J. Wang, A. P. de Vries, and M. J. T. Reinders. Unified relevance models for rating prediction in collaborative filtering. *ACM Trans. Inf. Syst.*, 26(3), 2008.

[14] T. Warashina, K. Aoyama, H. Sawada, and T. Hattori. Efficient k-nearest neighbor graph construction using mapreduce for large-scale data sets. *IEICE Transactions*, 97-D(12):3142–3154, 2014.