Efficient Incremental Cooccurrence Analysis for Item-Based Collaborative Filtering

Sebastian Schelter
New York University
sebastian.schelter@nyu.edu

Ufuk Celebi
Freie Universität Berlin
u.celebi@fu-berlin.de

Ted Dunning
MapR Technologies
tdunning@apache.org

ABSTRACT

Recommender systems are ubiquitous in the modern internet, where they help users find items they might like. A widely deployed recommendation approach is item-based collaborative filtering. This approach relies on analyzing large item cooccurrence matrices that denote how many users interacted with a pair of items. The potentially quadratic number of items to compare poses a scalability bottleneck in analyzing such item cooccurrences. Additionally, this problem intensifies in real world use cases with incrementally growing datasets, especially when the recommendation model is regularly recomputed from scratch. We highlight the connection between the growing cost of item-based recommendation and densification processes in common interaction datasets. Based on our findings, we propose an efficient incremental algorithm for item-based collaborative filtering based on cooccurrence analysis. This approach restricts the number of interactions to consider from 'power users' and 'ubiquitous items' to guarantee a provably constant amount of work per user-item interaction to process. We discuss efficient implementations of our algorithm on a single machine as well as on a distributed stream processing engine, and present an extensive experimental evaluation. Our results confirm the asymptotic benefits of the incremental approach. Furthermore, we find that our implementation is an order of magnitude faster than existing open source recommender libraries on many datasets, and at the same time scales to high dimensional datasets which these existing recommenders fail to process.

ACM Reference Format:
Sebastian Schelter, Ufuk Celebi, and Ted Dunning. 2018. Efficient Incremental Cooccurrence Analysis for Item-Based Collaborative Filtering. In *SSDBM '19: International Conference on Scientific and Statistical Database Management, July 23–25, 2019, Santa Cruz, CA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

Today's internet users face an ever increasing amount of data, which makes it constantly harder and more time consuming to pick out the interesting pieces of information from all the noise. This situation has triggered the development of recommender systems: intelligent filters that learn about the users' preferences and figure out the most relevant information for them. With rapidly growing data sizes, the processing efficiency and scalability of machine learning-based systems and their underlying computations becomes a major concern as well as their ability to continously update the underlying models [26, 31].

Item-based collaborative filtering, a classic approach to recommender systems [27, 28], is based on the idea of inspecting item cooccurrences ("people who interact with X also interact with Y"). It has the advantage of being intuitive to understand, due to the direct inspiration by recommendation in everyday life, where we tend to check out things that seem similar to what we already like. Because of these properties, item-based recommendation methods are widely deployed in industry [8, 9, 16], and form a strong baseline for recent approaches such as recurrent neural networks [17]. An operational aspect that makes item-based recommendations particularly useful is that the resulting models are easy to deploy with relational databases [29] or conventional text search engines [12].

In common with most other recommendation systems, end-to-end deployments of item-based recommenders typically regularly execute an offline learning phase to compute a model that is afterwards served for realtime recommendation. While this approach to update a recommender system is conceptually simple, it has a set of major drawbacks: (*i*) it is costly to recompute models from scratch and (*ii*) with growing input data, the growing job times make it more difficult to adhere to service-level agreements for downstream systems; (*iii*) furthermore, rapidly emerging trends and changes in the data are not picked up immediately. Unfortunately, tackling these drawbacks is not trivial, as the runtime of a naive approach to cooccurrence analysis grows superlinearly with the number of users in the data and quickly encounters memory pressure.

We therefore propose to replace the offline recomputation of the model with an efficient online algorithm with a provably constant amount of work per interaction to process. Our approach updates the parts of the model which are affected by the new interactions, instead of recomputing the model as a whole. We first introduce cooccurrence analysis as the basis of item-based recommendation (Section 2), outline scalability issues in item-based recommenders and discuss how these issues increase over time. We connect these scalability issues to well-known densification processes [20] in interaction datasets which lead to a growth of the average number of items with which a user interacts over time (Section 3.2). We adapt so-called 'interaction cuts' from previous work [32], which alleviate this scalability bottleneck using an approach based on reservoir sampling. The intuition behind these cuts is that they restrict the influence of interactions from 'power users' and very popular items (e.g., 'Lord of the Rings' in a movie recommendation scenario), which are not indicative of the taste of the majority of users. We derive an incremental version of the

SSDBM '19, July 23–25, 2019, Santa Cruz, CA

ACM ISBN 978-1-4503-9999-9/18/06…$15.00
https://doi.org/10.1145/1122445.1122456

cooccurrence-based recommendation algorithm and prove that this algorithm only requires a constant amount of work to incorporate a new user-item interaction into the recommendation model (Section 3.3). Next, we detail how to implement the algorithm efficiently on a single machine, and furthermore describe a distributed implementation in the stream processing engine Apache Flink [6] in Section 4. We review related work in Section 5, and present an extensive experimental evaluation of our proposed algorithm in Section 6, in which we investigate the benefits of an incremental approach, compare our implementation to existing libraries and analyze its scalability.

The contributions of this paper are the following:

- We propose an efficient online algorithm for cooccurrence-based recommendation, which guarantees a constant amount of work per interaction to process (Section 3).
- We highlight the connection between the cost of item-based recommendation and densification patterns in common interaction datasets (Section 3.2).
- We discuss efficient implementations of our algorithm on a single machine and on a distributed stream processing engine (Section 4).
- We present an experimental evaluation which confirms that the incremental approach is substantially faster than batch recomputation. We find that our implementations outperform existing libraries by an order of magnitude or more on many datasets, and scale to high dimensional datasets which existing libraries fail to process (Section 6).

2 BACKGROUND

We introduce cooccurrence analysis as a general form of item-based collaborative filtering for recommendations, and describe a common end-to-end deployment of such a recommender system. As already stated, item-based collaborative filtering [28, 32], compares user interactions to find related items in the sense of: 'people who like this item also like these other items'. In a movie recommendation case for example, the system records which movies often cooccur in the viewing histories of users. These pairs of cooccurring movies are then ranked and form the basis for recommendations later on. While academic research often restricts itself to predicting so-called *explicit feedback* for items (e.g., 'five-star ratings' for movies [5]), cooccurrence analysis focuses on the more general use case of *implicit feedback* which comprises of count data that can be easily gathered by recording user actions (such as purchases in online shops or the number of plays of a video on a movie platform). Note that this kind of data covers a much wider range of use cases than rating data, which requires explicit user actions and is only available in a limited set of domains (e.g., for movies or songs).

Deployment of item-based recommenders. A common way to implement and deploy an item-based collaborative filtering system is depicted in Figure 1. An offline training phase is conducted regularly. It computes a 'similarity matrix' of cooccurring item pairs and ranks these with a similarity measure or statistical tests. From these cooccuring items, a set of so-called *indicators* is retained per item, which comprise of its most strongly associated items. These indicators for each item form the recommendation model, and these

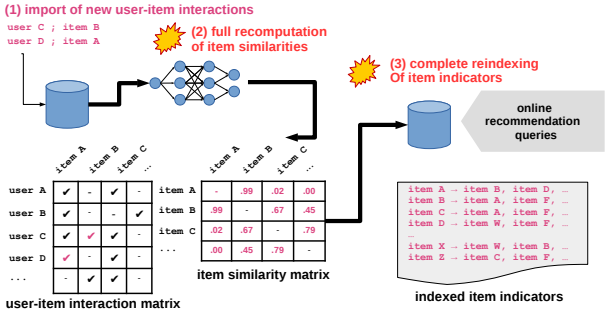

Figure 1: Traditional deployment of an item-based recommender system: New user interactions require a full, costly recomputation of the item similarities and the indexed item indicators. Often this recomputation is executed in regular intervals (e.g., nightly) with massively parallel systems such as Spark or Hadoop.

indicators are subsequently used to derive recommendations in realtime. Typically the system serving the recommendation model treats a list of items the user has interacted with recently as a query for a search engine or database, where items are indexed by their indicators. The ranked results comprise the items to recommend to the user. Such 'recommendation queries' can additionally filter the recommendations by additional indexed content (such as the category of products for example) [12, 28]. Note that efficiently serving recommendation models is a research area of its own, which is out of the focus of this work.

Computation of indicators from the cooccurrence matrix. We briefly detail how to compute the cooccurrence matrix and how to select the indicators from the item cooccurrences using statistical tests. Cooccurrence analysis works as follows: We represent the observed interactions between users U and items I by a binary matrix $\mathbf{A} \in \{0,1\}^{|U|\times|I|}$, whose rows correspond to users, and whose columns correspond to items. A cell a_{ij} of this matrix is set to one if user i interacted with item j and zero otherwise. In this setting, the computation of the cooccurrence matrix $\mathbf{C}$ corresponds to the matrix product $\mathbf{C} = \mathbf{A}^\mathrm{T}\mathbf{A}$. An entry $c_{j_1 j_2}$ of $\mathbf{C}$ denotes the number of users that interacted with both item j_1 and item j_2. This entry is obtained by computing the dot product between the respective columns $\mathbf{A}_{\bullet j_1}$ and $\mathbf{A}_{\bullet j_2}$ of $\mathbf{A}$ which corresponds to counting the overlapping user interactions of items j_1 and j_2.

For rating data, the item pairs are usually scored with a similarity function [28]. Instead, we focus on the strategy of identifying highly anomalous cooccurring item pairs with the loglikelihood-ratio (LLR) based G-test [11, 23]. In real world scenarios, where we deal with noisy, highly skewed count data, this approach is often preferred over similarity measures that were originally designed for rating data (according to our personal experiences with recommender systems deployed in industry). We use the statistic to retain a fixed number of anomalously cooccurring items per item, which we store as *item indicators*.

We compute the LLR-score from the entries of a contingency table summarizing the relationship between a particular pair of items j_1 and j_2. The table can be calculated from the cooccurrence

matrix and consists of four different counts: k_{11} denotes how often we encountered both items j_1 and j_2 in a user history; k_{21} denotes how often we encountered item j_1 in combination with other items than j_2 in a user history (and vice versa for k_{12}), while k_{22} denotes how many item pairs we saw which neither include j_1 nor j_2:

	item j_1	**no item j_1**
item j_2	j_1 and j_2 ($\boldsymbol{k_{11}}$)	j_2 without j_1 ($\boldsymbol{k_{12}}$)
no item j_2	j_1 without j_2 ($\boldsymbol{k_{21}}$)	neither j_1 nor j_2 ($\boldsymbol{k_{22}}$)

Given these counts from the contigency table, we calculate the score $\texttt{llr}(j_1, j_2)$ of the item pair as follows [11]:

$$2N \cdot [\, \texttt{ent}(k_{11}, k_{12}, k_{21}, k_{22}) - \texttt{ent}(k_{11} + k_{12}, k_{21} + k_{22}) - \texttt{ent}(k_{11} + k_{21}, k_{21} + k_{22})]$$

where N denotes the total number of cooccurrences and ent computes Shannon's entropy. Note that the LLR-score is equal to the mutual information between the row and column vector of the contingency table scaled by $2\,N$.

3 ALGORITHM

We first introduce notation and provide an overview of the proposed algorithm. Afterwards, we outline general scalability issues when applying cooccurrence analysis to large datasets and develop our online algorithm. Figure 2 illustrates our proposed approach.

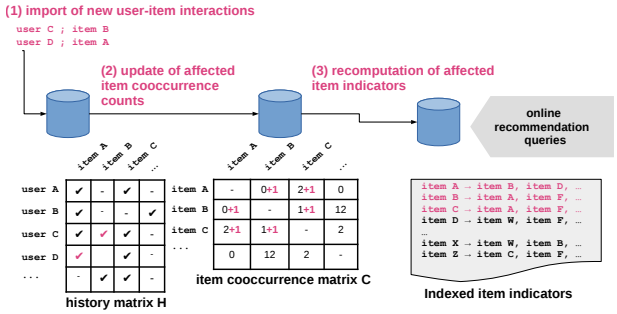

Figure 2: Proposed approach: Upon arrival of new user interactions, we conduct online point updates in the history matrix and cooccurrence matrix (with constant amount of work per interaction), and recompute only the affected item indicators.

Overview and notation. As already mentioned in the previous section, we assume that all interactions between users U and items I that occur over time are captured in a binary matrix $\mathbf{A} \in \{0,1\}^{|U|\times|I|}$. Instead of working directly with this matrix to compute a recommendation model from time to time, our algorithm maintains two other matrices to incrementally compute the item indicators which will be used for recommendation in the end. We process each new interaction (i, j) as follows:

(1) We incrementally maintain a history matrix $\mathbf{H} \in \{0,1\}^{|U|\times|I|}$, which contains a subset of $\mathbf{A}$ such that it has a bounded number of interactions per item and user.
(2) We incrementally maintain the cooccurrence matrix $\mathbf{C} \in \mathbb{N}^{|I|\times|I|}$ with $\mathbf{C} = \mathbf{H}^{\mathrm{T}}\mathbf{H}$ which denotes how often pairs of items occur together in user histories in $\mathbf{H}$.
(3) For each update, we recompute the affected item indicators with loglikelihood-ratio based scores.

3.1 Scalability Issues

Before we get to the online variant of our recommendation algorithm, we need to understand general scalability issues of cooccurrence analysis for large datasets. For that, we examine the cost of computing the cooccurrence matrix. In the worst case, this cost would be quadratic in the number of items $|I|$, as we would have to compare all pairs of items for all users. In reality however, the cost is much less than that, because almost all users interact with only a small fraction of the items. A simple way to compute the cooccurrence matrix is to pick a user, count all pairs of items in the history of her items, and repeat this step for all users [28]. For any user i, this involves only the non-zero entries of $\mathbf{A}_{i\bullet}$ (i.e., the items that she interacted with): We see that the cost of this algorithm decomposes

```
for user i :
    for history item j1 ∈ A_i• :
        for history item j2 ∈ A_i• :
            c_j1j2 ← c_j1j2 + 1        // count item pair (j1, j2)
```

by users: Let l_i denote the total number of items $\sum_j a_{ij}$ that user i interacted with. Then we have to look at all resulting item pairs. This requires a quadratic amount of operations per user i, resulting in a total effort of $\gamma \propto \sum_i l_i^2$. This illustrates that the cost of the computation γ is dominated by the densest rows of $\mathbf{A}$, resulting from the actions of the users with the highest number of items in their item history.

3.2 Power Law Patterns in Interaction Data

We continue to investigate how users with a high number of interactions impact the runtime of cooccurrence analysis, and focus on how this effect evolves over time. Fortunately and unfortunately, interaction datasets share a common property with many datasets representing the outcome of human behavior: the distribution of the number of interactions with contained entities is highly skewed [4], and this characteristic even increases over time in many datasets. In network analysis, it has been found that many networks follow a densification power law pattern, e.g., that the average vertex degree in those networks grows proportionally to a power law over time [20].

Densification in interaction datasets. We investigate whether this is also the case for interaction data. We therefore adapt the techniques from [20] and analyse the temporal evolution of user interaction histories in three time-stamped datasets[1] from different domains: *StackOverflow*, which comprises of 1,301,942 favorites of 545,196 users for 96,680 posts on a question-answering platform, *DBLP*, which contains 18,986,618 co-authorship relations between

[1]The networks as well as their corresponding statistics are publicly available via the Koblenz Network collection at http://konect.uni-koblenz.de/networks/{dblp_coauthor, stackexchange-stackoverflow, lastfm_band}.

1,314,050 researchers, and *lastFM*, a dataset containing 19,150,868 plays of 174,077 bands by 992 users. As the densification process has been studied for networks, we model these interaction datasets as bipartite networks [19], where users and items form the vertex sets and interactions connect these vertices as edges. Here the average size of the user interaction history corresponds to the average out-degree of user vertices. We plot the number of users versus the number of interactions over time on a doubly logarithmic scale in Figure 3. The apparent straight line hints at the existence of a growth power law for the average length of the user history. We fit a line to the data to confirm this, and find R^2 values over .99 in all cases.

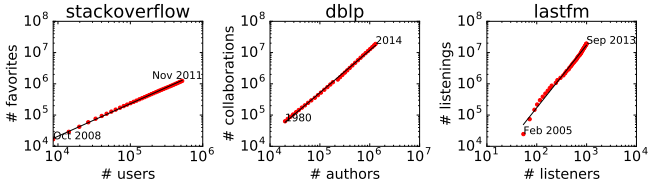

Figure 3: Temporal growth of the average length of the user interaction history in several interaction datasets. We plot the number of interactions versus the number of users in log-log scale. The corresponding datasets obey the densification power law for networks (slopes: $\alpha = 1.03, 1.36, 2.0$, note that slope 2 occurs due to the fact that lastFM counts multiple interactions of a user with the same song).

Growth of the average user history length. This analysis confirms that the average user history length grows with a power law over time, which poses a huge challenge for cooccurrence-based recommendation algorithms, because their computational cost is quadratically proportional to the contained user history sizes (as we explained in the previous section). In the case of a power law densification over time, the skew of the history size distribution increases, which leads to a superlinear growth of the computation cost for interaction datasets with a growing number of users. Temporal patterns like densification are often not examined by the research community as academics usually only have access to static snapshots of real world interaction datasets.

3.3 Incremental Cooccurrence Analysis

We detail the proposed incremental version of the cooccurrence analysis algorithm. It turns out that we can omit certain elements of $\mathbf{A}$ in the history matrix $\mathbf{H}$. This has the effect of making the overall computation much cheaper, but also makes it possible to compute the effect of each new interaction with strictly bounded effort. We restrict the size of any individual's history and the frequency of any particular item.

'Interaction cuts'. The insight that the full cooccurrence cost γ is quadratically proportional to the user history lengths $\gamma \propto \sum_i l_i^2 \in \omega(|U|)$ implies that naive cooccurrence computations will not scale to real world datasets. Any new interaction for user i has the potential to produce as many as l_i cooccurrences. Since the number of interactions for any given user increases without bounds, this means that a strictly real-time update of the model using naive cooccurrence is not possible. We placed bounds on the per user interaction cost of the cooccurrence computation in previous work [32] on efficient batch recommendation. The idea is to define a threshold k_{max} (called 'user interaction cut') for the number of item interactions to remember per user. If a user's interaction history exceeds this bound, a random subset of size k_{max} is used for the computation of the cooccurrence matrix. Note that this cut typically only applies to a small fraction of the overall users (sometimes referred to as 'power-users'). In real world scenarios, such users often correspond to automated bots or accounts shared by many people (which do not provide valuable input data to the recommender system anyways), and the interaction cut has been shown to have negligible effects on the prediction quality on hold-out sets [17, 32]. Similar techniques are used in natural language processing, e.g., approaches such as *word2vec* [25] typically incorporate such a limit automatically by considering only cooccurrence within a fixed distance or within a single sentence. In practice, variants of the interaction cut might be applied in order to for example retain more recent items or highly preferred items rather than a random sample.

The bound enforced by the user interaction cut makes the cooccurrence computation scalable no matter the form or evolution of A, as there are at most k_{max}^2 item pairs to count per user and thus $\gamma \propto \sum_i \min(k_{max}, l_i)^2 \leq k_{\max}^2 |U| \in O(|U|)$. We also introduce a second threshold f_{max} (called 'item interaction cut') for the number of user interactions to consider per item. This threshold only applies to 'ubiquitous items' and has a similar effect as the frequent item downsampling employed in *word2vec* or as stopword removal in information retrieval, where common words are removed as they typically provide only little discriminative information. The item interaction cut makes the maximum cooccurrence count relatively small ($\leq f_{max}$) and with the user interaction cut, the total number of non-zero pairs in the cooccurrence matrix is relatively small ($\leq k_{\max}^2 |U|$) as well. Note that we will experimentally evaluate the effects of the proposed interaction cuts of the prediction quality in the further course of this section.

By putting a bound on the number of non-zeros per row and column of the history matrix, we also bound the costs of the update $\Delta(\mathbf{H}^\mathrm{T}\mathbf{H})$ resulting from a new interaction (i, j) to the cooccurrence matrix $\mathbf{C}$. Due to the addition of an interaction of user i with item j, a matrix $\mathbf{E} \in \{0,1\}^{|U|\times|I|}$ which has $e_{ij} = 1$ and zero for the remaining entries will be added to $\mathbf{H}$, resulting in the following change:

$$\begin{aligned}\Delta(\mathbf{H}^\mathrm{T}\mathbf{H}) &= (\mathbf{H}+\mathbf{E})^\mathrm{T}(\mathbf{H}+\mathbf{E}) - \mathbf{H}^\mathrm{T}\mathbf{H} \\ &= \mathbf{E}^\mathrm{T}\mathbf{H} + \mathbf{H}^\mathrm{T}\mathbf{E} + \mathbf{E}^\mathrm{T}\mathbf{E} \\ &= \begin{bmatrix} \mathbf{0} \\ \mathbf{H_{i\bullet}} \\ \mathbf{0} \end{bmatrix} + \begin{bmatrix} \mathbf{0} & \mathbf{H_{i\bullet}^\mathrm{T}} & \mathbf{0} \end{bmatrix} + \delta_{(jj)}\end{aligned}$$

Here, $\delta_{(jj)} \in \{0,1\}^{|I|\times|I|}$ denotes a matrix with all zeros except for $\delta_{jj} = 1$. As the number of non-zeros in $\mathbf{H}_{i\bullet}$ is at most $k_{\max}$, the resulting updates involves at most twice the user interaction cut $k_{\max} + 1$ items. This bound on the number of updates trivially bounds the update time. With the item interaction cut at k_{max} we have $\left\|\Delta(\mathbf{H}^\mathrm{T}\mathbf{H})\right\|_0 \leq 2\,k_{max} + 1 \in O(1)$.

Incremental maintenance of the interaction cuts. In order to incrementally maintain the desired sparsity of the history matrix **H**, we apply reservoir sampling as depicted in Algorithm 1. The algorithm takes a new interaction (i, j) between a user i and an item j as input, and updates the history matrix **H** such that the maximum number of non-zero entries per row is k_{max} (the user interaction cut) and the maximum number of non-zero entries per column is f_{max} (the item interaction cut). It maintains a $|U|$-dimensional vector **l** which denotes the total number of interactions observed for each user, independent of whether we accept the interaction into **H**. For a new interaction (i, j), we proceed as follows. First,

Algorithm 1: Updating the history matrix for a new interaction (i, j) between user i and item j; we enforce the user interaction cut k_{max} for the number of items per user and the item interaction cut f_{max} for the number of users per item in the history matrix **H**.

```
   /* increment total interaction count for user i, regardless of whether
      we accept this interaction in H or not */
1  l_i ← l_i + 1
   /* Will be true for the majority of items */
2  if item j has less than f_max interactions in H :
     /* Will be true for the majority of users */
3    if user i has less than k_max interactions in H :
4      add new interaction (i, j) to history matrix
5    else:
       /* user history in H changes with decreasing probability
          depending on the total number of interactions l_i we have
          already seen for user i */
6      with probability k_max / l_i:
7        choose random item r from history of user i in H
       /* item r gets pushed out, its interaction count is decreased, and it
          will be reconsidered in line 2 the next time its seen (as its count
          is guaranteed to be less than f_max) */
8        replace item r with item j
```

we increment the total observed interaction count l_i for the user i in line 1. Next, we compare the current number of interactions of the item j in **H** with the desired frequency cut f_{max} in line 1. If the threshold is exceeded, we ignore the interaction. The point of checking the item interaction cut first is that extremely popular items (e.g., 'Lord of the Rings' in a movie dataset) quickly go into a state where they are ignored and no more work will be required to keep their counts up to date. They may only be reconsidered later however if they get pushed out of user histories in the subsequent steps. If the interaction passes the check, we compare the current number of interactions of user i in **H** to the interaction cut k_{max} in line 3. If the user has fewer interactions in **H** than the cut, we accept the new interaction and append it to the user's history in line 4. This will happen for almost all users. In the rare case where a user already has k_{max} interactions in her history, we randomly replace one of the existing items with the new item j with a probability of k_{max} / l_i as shown in lines 6 to 8 (note that this probability decreases inversely proportional with the total number of interactions l_i which we have already seen for user i). When changing the user history, we choose a random item r from the user's history and replace it with item j. As the interaction count of item r in **H** decreases in this case, item r will pass the condition in line 2 the next time it is found in an interaction, and it has a chance to re-enter the matrix (as its interaction count in **H** is guaranteed to be less than f_{max}).

Effects of interaction cuts. We empirically validate the benefits of interaction cuts to confirm the findings from [17, 32]. We apply different interaction cuts k_{max} and f_{max} to the three time-stamped interaction datasets *StackOverflow*, *DBLP* and *lastFM* introduced earlier in this section. For each dataset, we provide several measurements. Figure 4 shows (i) the overlap in the top-10 indicated items per item for different cuts compared to the original dataset, (ii) the runtime required to compute the indicators for different cuts, compared to the runtime on the original dataset, (iii) the effect on prediction quality. For the latter, we hold out 10 items from the interaction history of each user with 20 or more interactions. We make recommendations for these users by finding the top 10 items not already in the user's history. Recommended items were ranked by the overlap of the user's history with the top-100 indicators for each item. We compare the *precision@10* of the top 10 predicted items to the held out items for the original data and different cuts.

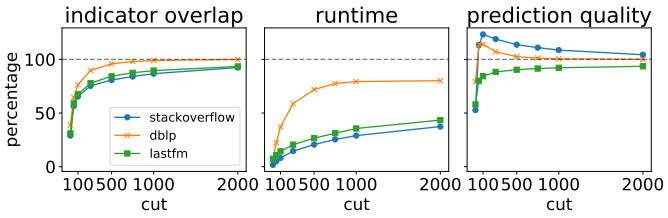

Figure 4: Effect of different interaction cuts on indicators, runtime and prediction quality for three datasets from different domains. The dotted line represents the results on the original data with no cuts applied, the y-axis denotes the percentage of indicator overlap, runtime and prediction quality compared to execution with no cuts applied.

The dashed gray line in Figure 4 indicates the results for the original data without application of the cuts. We see that the cuts quickly provide a high degree of overlap of item indicators (except for very small cuts). For cuts between 500 and 1,000 interactions per item and user, we find an overlap of 80% to 99% percent, depending on the dataset. At the same time, the cuts have a tremendous impact on the runtime, as they filter out the 'power users' and 'ubiquitous items' that have a disproportionate effect on the runtime. For the lastFM and stackoverflow datasets, we always run in less than half of the runtime when interaction cuts were used. For DBLP, the runtime was only reduced by 20% compared to the runtime on the original data in the end, which we attribute to the fact that there are only very few users with a very high number of co-authors.

The most interesting result is shown in the rightmost plot, where we see the effect on prediction quality: While the quality slightly decreases for the lastFM data with the cuts, applying the cuts increases the average prediction quality for the other datasets. For the stackoverflow data the increase can be as high as 15% better

precision. We attribute this effect largely to the removal of ubiquitous items since the user interaction cut would only affect a small minority of all users making it hard to change the average precision. The downsampling of highly frequent terms in *word2vec* [25] is motivated by similar improvements in quality. With larger cut sizes, we converge to the prediction quality on the unsampled data on average. This confirms our previous finding [32] that a cut size between 500 and 1,000 appears to be a reasonable default choice across many datasets. Cut thresholds of this size do not change item indicators very much, decrease runtime significantly, and have little or no negative effect on the average prediction quality.

Efficient indicator updates. After having updated the cooccurrence matrix $\mathbf{C}$, we recompute the item indicators which are affected by the changed cooccurrences. Algorithm 2 shows the steps required to recompute indicators $\mathbf{S}_j$ for item j. Our goal is to update this set to hold the list of n items with the largest LLR scores. For each row, we keep a min-heap of *(item, LLR-score)* tuples, where the ordering is based on the LLR scores and then on item id. In order to conduct this update, we need read access to the cooccurrence matrix $\mathbf{C}$, the total number of cooccurrences N, and the vector of row sums $\mathbf{r}$ of $\mathbf{C}$. For every item q that cooccurs with item j, we compute

Algorithm 2: Recomputing the top-n indicators S_j for a particular item j from the cooccurrence matrix $\mathbf{C}$ and its row sum $\mathbf{r}$.

```
1  function rescore(j, n):
2    S_j ← min-heap with capacity n
3    for c_jq ∈ non-zero entries of C_j• :
       /* compute LLR score from contingency table */
4      k_11 ← c_jq
5      k_12 ← r_j − k_11
6      k_21 ← r_q − k_11
7      k_22 ← N − k_12 − k_21 + k_11
8      s_q ← llr(k_11, k_12, k_21, k_22)
       /* maintain top-n item-score tuples via a min-heap */
9      if heap S_j has less than n entries :
10       add item item-score tuple (q, s_q) to heap
11     else:
12       if s_q is larger than the heap root :
13         update heap root with (q, s_q)
14   return S_j
```

the counts $k_{11}, k_{12}, k_{21}, k_{22}$ of the contingency table between j and q (as described in Section 2) as follows in lines $4-8$. The number of cooccurrences k_{11} of j and q corresponds to the entry c_{jq}. The number of cooccurrences k_{12} of j with items other than q is $r_j - k_{11}$. The number of cooccurrences k_{21} of q with items other than j is $r_q - k_{11}$, and the number of observed cooccurrences containing neither j nor q is $N - k_{12} - k_{21} + k_{11}$. Using the contingency table, we compute the LLR score s_q for the cooccurrences between j and q. If the heap for $\mathbf{S}_j$ contains less than n entries, we push the tuple (q, s_q) onto the heap (lines 9 and 10). Otherwise, we compare the current score s_q to the current smallest score from the tuple sitting on top in the heap (by invoking *peek*) and update the heap if s_q is larger than the smallest score in the heap (lines 12 and 13). Since every update operates exclusively on the indicators for an individual item j and only requires read access to $\mathbf{C}$, recomputing all affected item indicators is embarassingly parallel.

End-to-end algorithm. We combine history matrix maintenance, cooccurrence updates and indicator recomputation into an end-to-end approach to incremental cooccurrence analysis in Algorithm 3. Note that we omit the trivial updates of the overall number of cooccurrences N and the row sums of the cooccurrence matrix for readibility. Lines 2 to 23 contain the interwoven update steps for the history matrix and cooccurrence matrix, either directly incorporating a new interaction (lines 5 to 10) or replacing a previously selected one (lines 11 to 23). In the end, we trigger the parallel indicator recomputation for all affected items (lines 24 and 25). In a real-world deployment, the recomputed indicators would then lead to a corresponding update in the system serving the recommendation model (e.g., update of a search index for the items whose indicators changed).

It is useful from an algorithmic and cost analysis perspective to think about updating the model and the intermediate data structures for each new user-item interaction in isolation. In real systems however, online recommendation algorithms often need to be turned into a mini-batch variant which processes a small batch of newly observed interactions at a time (e.g., all interactions from the last minute) instead of only a single interaction. The main benefit from this is a reduction of coordination costs between different systems because of less frequent interactions, and an increased control over update intervals. Our end-to-end version in Algorithm 3 can easily be run in mini-batch mode: we first sequentially execute the history matrix updates of $\mathbf{H}$ and the corresponding updates of the cooccurrence matrix $\mathbf{C}$ for all interactions contained in the batch. Next, we collect the items for which we need to recompute indicators (the set R) for all interactions processed, and execute the indicator recomputation for all affected items $q \in R$ in parallel.

4 IMPLEMENTATION

While it is clear that interaction cuts will improve the runtime for computing and maintaining indicator sets for items, there are still some interesting optimizations available in the implementation itself. The available optimizations are, not surprisingly, very different depending on whether the implementation targets a single machine scale-up architecture or a streaming scale-out architecture.

4.1 Single Machine

In our single-machine implementation[2], we apply Algorithm 1 in a single pass over the mini-batch and thereby maintain the history matrix $\mathbf{H}$. As we bound the number of interactions per user because of the user interaction cut k_{max}, the memory requirements for the underyling array data structure to represent $\mathbf{H}$ is at most $|U|k_{max}$. For any accepted interaction, we update the appropriate entries in $\mathbf{H}$ and the corresponding cooccurrence counts in $\mathbf{C}$. We represent the sparse cooccurrence matrix $\mathbf{C}$ as an array of hashmaps indexed by row to allow for fast random access to the cooccurrence counts. As the maximum cooccurrence count between two items

[2] https://github.com/sscdotopen/puppies/blob/master/src/lib.rs

Algorithm 3: End-to-end incremental cooccurrence analysis algorithm for updating the history matrix **H**, the coccurrence matrix **C** and for recomputing affected item indicators from a new interaction (i, j).

```
1  initialize set of items to rescore R
   /* increment total interaction count for user i */
2  l_i ← l_i + 1
3  if item j has less than f_max interactions in H :
4    if user i has less than k_max interactions in H :
5      for item p ∈ user history H_i• :
         /* record cooccurrence with item j */
6        c_jp ← c_jp + 1
7        c_pj ← c_pj + 1
8        add item p to set of items to rescore R
       /* add item j to history of user i */
9      h_ij ← 1
10     add item j to R, the set of items to rescore
11   else:
       /* accept (i, j) with probability k_max / l_i */
12     n ← unif( 0, l_i )
13     if n ≤ k_max :
14       r ← index of n-th non-zero column in H_i•
         /* remove item r and add item j to history of user i */
15       h_ir ← 0
16       h_ij ← 1
         /* update affected coocurrence counts */
17       for item q ∈ user history H_i• :
18         c_iq ← c_iq + 1
19         c_qi ← c_qi + 1
20         c_qr ← c_rq − 1
21         c_rq ← c_qr − 1
22         add item q to set of items to rescore
23       add items j and r to set of items to rescore
24 for item q ∈ items to rescore R do in parallel :
25   S_q ← rescore(q, n);
```

is f_{max} with f_{max} typically in the hundreds, 16-bit precision is sufficient for the representation of the underlying matrix entries. As discussed in Section 3.3, the total number of non-zero entries in the matrix **C** is at most $|U|k^2_{max}$. After updating the cooccurrence matrix, we recompute the set of indicators **S** affected by the items in the current mini-batch. As already mentioned, this computation is embarrasingly parallel. We execute the operations in a multi-threaded fashion with read-only access to the cooccurrence matrix. As depicted in Algorithm 2, we maintain a priority queue of maximum size n for each item, holding the LLR scores and identifiers of the current top-n highest associated items. The total memory requirement for this data structure amounts to at most $|I|n$. Our algorithm is written in *Rust 1.20* using `fnv::FnvHashMap` to represent the rows of the cooccurrence matrix and `scoped::ScopedPool` for concurrency.

Optimized loglikelihood-ratio computations. Initial profiling showed that the runtime is dominated by the calculation of logarithms for the loglikelihood-ratio score in the `rescore` function. We introduce two optimizations to reduce the number of logarithm computations. First, we manually apply common subexpression elimination in our code. Next, we observe that five out of the nine remaining logarithm computations have their argument in the range $[0, (f_{max}\, k_{max} - 1)]$. This stems from the fact that an item can at most occur in f_{max} user histories, each of which can contains only a maximum of $k_{max} - 1$ other items. We precompute all the logarithms in this range. In micro-benchmarks, we find that these optimizations approximately halve the runtime of the loglikelihood-ratio tests.

4.2 Stateful Stream Processor

Our distributed implementation[3] is based on the stream processing engine *Apache Flink* [6]. Figure 5 gives an overview of the corresponding dataflow, which has three distinct phases: (*i*) *distributed history matrix updates*, (*ii*) *cooccurrence updates* and (*iii*) *indicator recomputation*. A distributed implementation cannot be a straightforward translation of the proposed algorithm, due to multiple challenges with respect to how to access and update the application state in a distributed setting. Due to partitioned state access, it becomes difficult to provide consistency when maintaining the history matrix. Furthermore, we lose ordering guarantees once we partition the stream across multiple independent subtasks, and therefore require windowing constructs for updating the history matrix and the cooccurrence matrix. Finally, we need upstream feedback to maintain the interaction cuts in the history matrix.

Distributed application of the 'interaction cut'. We partition the item and user interaction update operators across two different keyed state instances. Note that it is not possible to update both of these state instances atomically. Interactions flow from the `ItemInteractionUpdater` (which checks the item interaction cut and keeps track of the item interaction counts) downstream to the `UserInteractionUpdater`, which maintains the user interaction cut and keeps track of the corresponding user interaction counts. An interaction that passes the item interaction cut at the `ItemInteractionUpdater` might be subsequently rejected by the `UserInteractionUpdater`. Therefore, we require an upstream feedback mechanism and may encounter short temporal inconsistencies of the counts for an item in the brief timespan before the feedback is incorporated. Our implementation prevents overcounting and

[3] `https://github.com/uce/flink-cooccurrence`

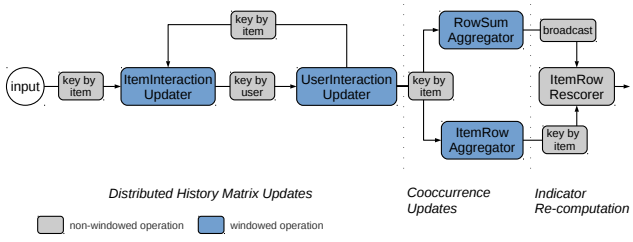

Figure 5: Our algorithms as dataflow in Apache Flink.

enforces the cuts, but can result in temporary undercounting in two situations: (*i*) When an item count is optimistically incremented and reaches the item interaction cut but does not pass the user interaction cut, the next interaction for that item will be ignored until the upstream feedback arrives. (*ii*) When the item count for an item has reached the item interaction cut and a user interaction results in an interaction with that item to be replaced, new interactions for that item will be ignored until the upstream feedback for the replacement arrives. Note that these situation only occurs for items which have already received many interactions, making it unlikely that the ignored interaction will strongly affect the LLR score. We conclude that this implementation is a reasonable approximation of the original algorithm since we still enforce the cuts and the common cases are handled well. We experimentally validate this in Section 6.3. We implement both update operators with tumbling event time windows to ensure order between events. We group all interactions by their corresponding time window before conducting the updates. Thereby we guarantee an order by the mini-batch granularity (e.g., the window for the i-th minute will be processed before the window for the $i + 1$-th minute). Such windowing is necessary as progress of event time is only tracked locally per stream partition (and we re-partition the data between the update operators). Furthermore, we need ordered output from the update operators for the downstream operations. We implement a custom operator for the upstream feedback. On a logical level, the dataflow simply has a feedback edge from the `UserInteractionUpdater` to the `ItemInteractionUpdater`. Physically, Flink's internal graph abstraction that specifies the dataflow does not allow us to create loops. Instead we implement the item update operator with two inputs: its regular input and a feedback source, and the subsequent update operator only gets a single output downstream and a reference to an in-memory blocking queue for the feedback in the user code. We accomplish this via a co-location scheduling constraint that guarantees that the i-th subtask of the `UserInteractionUpdater` and the i-th subtask of the feedback source are executed on the same worker instance, analogous to constructs that Flink leverages for its iterative dataflow abstraction [14].

Cooccurrence matrix updates. Next, we implement the cooccurrence matrix update. The `ItemRowAggregator` aggregates all item cooccurrences for a single item and the `ItemRowSumAggregator` aggregates the row sums. The `ItemRowRescorer` (which will consume their outputs afterwards) needs access to a single row holding the cooccurrences for a single item and all row sums. Both of these must be consistent with each other for a specific time window in order to guarantee consistent results during rescoring. We achieve this by again applying a tumbling event time window which groups the interactions by their event time. When a window fires, we emit aggregated row sum updates and item cooccurrence updates downstream to the rescoring operator. The emitted values correspond to the deltas for items that occurred in the current window. The `ItemRowRescorer` is responsible for recomputing the indicators for all items with changed cooccurrences. While this an embarrassingly parallel operation, a challenge arises as we require global access to the row sums of all items (denoted as $\mathbf{r}$ in Algorithm 2). For that, we implement a streaming broadcast join which broadcasts the output of the `ItemRowSumAggregator` to all parallel instances holding the rows of the cooccurrence matrix. We use Flink's time progress tracking via watermarks to decide when to evaluate the join. We buffer all received events from both sides, and when the watermark advances, we join all buffered elements that have a timestamp lower or equal to the watermark. We consume the buffered events and apply the deltas to the global row sums and each cooccurrence vector. This gives us the global view, which we keep in sync by following the watermark progress. After having applied all the delta updates for a certain timestamp, we recompute all item indicators up to that point in time.

5 RELATED WORK

Recommender systems are an active field of research [27]. In contrast to our work, many papers focus on 'explicit feedback' data (e.g., star ratings for movies) instead of the prevalent implicit feedback data (e.g., counts of views of product pages) and ignore the issue of skew in the data (which increases over time), as the majority of researchers only have access to static datasets.

Item-based collaborative filtering. Sarwar et. al [28] introduced the classical item-based collaborative filtering approach which computes a matrix of similarities based on cooccurrences between items in user interactions. In previous work, we proposed a parallel formulation of this approach for map-reduce [32], which introduced the interaction cuts leveraged in this work. Implementations of neighborhood-based methods can be found in many popular recommender libraries, e.g., in *Apache Mahout* [32], *Lenskit* [13] and *MyMediaLite* [15], and industry deployments [8, 9, 16]. There have been some proposals of incremental item-based recommenders already. Liu et al. [22] introduce time-based exponential decay to downweight past interactions for example. However their work only focuses on rating prediction with Pearson correlation as similarity measure and does not address the issue of skew over time. While they conduct no extensive performance evaluations, they report up to 20x speed increase in comparison to a non-incremental approach which is consistent with our experimental findings. *StreamRec* [7] is a real time recommender system based on the incremental computation of cosine similarities, which again only focuses on rating data. It is not clear that any of these implementations have hard bounds on the cost of a single update and thus it is not clear that any of them are suitable for hard real-time use. Another line of research are in-database recommender systems. *RecStore* [21] is a DBMS storage engine module for the online model maintenance of neighborhood-based recommenders. It features different materialization strategies for intermediate results. Unfortunately, the paper does not discuss skew in the data and the proposed solution is only evaluated on a small dataset. *RecDB* [29, 30] is a PostgreSQL-based system providing recommendation inside the database engine, which introduces new query operators inside the database kernel. Again the work focuses on rating data only and is evaluated only on small datasets with less than one million interactions.

Latent factor models. Due to the Netflix prize competition [5], so-called *latent factor models*, based on matrix factorization [18] have become widely popular for recommendation. These approaches are primarily designed for rating data and minimize the empirical regularized squared error of the model predictions to the observed ratings by projecting users and items onto a joint latent space.

However, the recommender systems community has acknowledged a set of drawbacks stemming from a sole focus on rating prediction [3, 24], and the winning algorithms from the Netflix prize have never been put into production [2]. Compared to latent factor models, neighborhood-based methods are complementary, simpler in their structure, and put their focus on local relationships between items rather than global patterns (distances in the joint latent space). Neighborhood-based methods shine performance-wise in the prediction phase: they just require search in a sparse indicator index, which is much cheaper than maximum inner product search in a dense matrix for latent factor models. Due to their high sparsity, the indicators produced by cooccurrence analysis can be efficiently indexed with a search engine or database, which allows queries to additionally filter recommendations based on item predicates. Another advantage of neighborhood methods is that they naturally handle new users (which are represented by the set of their item interactions), while latent factor models either require a retraining step or some way to project these new users into the latent space. Finally, cooccurrence-based recommendations are easy to explain as they are able to provide instant justifications for their recommendations by presenting the list of cooccurring items used for the recommendation. There has also been research on developing online versions of latent factor models [1]. Diaz-Aviles et al. [10] treat online collaborative filtering as online ranking problem, and propose a stream ranking matrix factorization approach based on selective sample of the stream, which they evaluate on a large dataset of tweets. *Factorbird* [33] is a prototype of a scalable matrix factorization approach built on the parameter server architecture, which has been leveraged to compute factorizations of large interaction networks at Twitter.

Deep neural networks. While deep neural networks exhibit massive potential for improving recommender systems, it has recently been shown that classical item-based methods constitute a strong baseline for deep networks in tasks such as session-based recommendation [17]. Additionally, the authors report that the item-based method applied in this work is an order of magnitude faster to train than the recurrent neural network which it was compared against.

6 EXPERIMENTAL EVALUATION

In the experimental evaluation, we focus on the efficiency and scalability of our proposed algorithm, as the prediction quality of item-based recommenders is already established [17, 28, 32] and we investigated the effect of our cuts on prediction quality in Section 3.3. We experimentally validate that an incremental approach is preferable over repeated recomputations, we compare our solution to various existing libraries, validate that the history matrix maintenance works as intended, and finally investigate scalability properties of our single machine and distributed implementations.

The datasets we use for performance evaluation are shown in Table 1 : *Movielens1M*[4], a popular dataset of movie ratings, *DBLP*[5] coauthorships and *Twitter* which contains user mentions of hashtags, that we extracted from the public twitter stream of January 2017[6]. We also evaluate on the *Netflix* [5] movie ratings dataset and a large publicly available dataset of song ratings from *Yahoo Music*[7]. If not explicitly mentioned otherwise, we process the data in chronological order if timestamps are available and set the number n of indicators to compute per item to 10, and the user interaction cut k_{max} and item interaction cut f_{max} to 500 based on our findings in Section 3.3. If not indicated otherwise, we experiment using our single machine implementation in Rust 1.20 on a machine with an Intel i7-7700HQ CPU @ 2.8GHz and 16GB of RAM, running Ubuntu Linux 16.04.

dataset	#users	#items	#interactions
Movielens1M	6040	3796	1,000,209
DBLP	1,314,051	1,314,051	18,986,618
Twitter	8,094,909	3,070,055	27,344,275
Netflix	480,189	17,770	100,480,507
Yahoo Music	1,823,179	136,736	699,640,226

Table 1: Datasets used during evaluation.

6.1 Benefits of Incrementalized Computation

Initially, we investigate the benefits of updating the recommendation model in an incremental fashion. We aim to showcase the improvements which an incremental approach provides. We compare this to the common use case of completely recomputing the indicators in regular intervals (e.g., once per day), as outlined in Section 2. We leverage the *MovieLens1M*, *DBLP* and *Twitter* datasets, which we partition into inputs of 10, 000, 250, 000 and 500, 000 interactions. We prepare the inputs to match the batch and incremental execution modes, e.g., for the batch variant the n-th input comprises of the union of all inputs up to n, whereas for the incremental variant the n-th input only contains the interactions of input n. In the batch variant, we have to read in the complete dataset up to the current batch before being able to recompute the indicators. In the incremental variant, we only make a single pass over the data and update the existing indicators. Both variants apply interaction cuts.

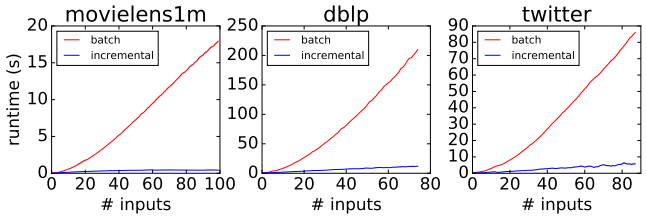

Figure 6: Comparison of repeated batch recomputation of the item indicators (whose costs grows linearly with the datasize) to an incremental update of the indicators with a bounded update cost.

Figure 6 illustrates the resulting runtimes of both the batch and incremental variant. We see that the incremental execution has an analogous effect on the runtime for all datasets: the amount of work that is required per input is virtually constant across all

[4] https://grouplens.org/datasets/movielens/1m/
[5] http://konect.uni-koblenz.de/networks/dblp_coauthor
[6] https://archive.org/details/archiveteam-twitter-stream-2017-01
[7] http://webscope.sandbox.yahoo.com

	Our approach		MyMediaLite		RecDB	
dataset	train	predict	train	predict	train	predict
Movielens1M	**18**	0.5	87	**0.2**	30	1 per user
DBLP	**234**	**6.6**	-	-	-	-
Twitter	**114**	**18**	-	-	-	-

Table 2: Batch train and predict times (in seconds) of our approach compared to MyMediaLite and RecDB on different datasets. Note that – indicates that the recommender ran out of memory or was not able to finish training within one hour.

inputs, scaling with the size of the input and not with the size of the growing dataset. Additionally, the incremental variant exhibits low variance in the daily runtime. The runtime of the batch variant on the other hand is linearly growing with the increasing size of the overall dataset. Even in this small experiment, the runtime for the last input is more than an order of magnitude less than in the incremental case for all three datasets. This experiment confirms that our incremental variant changes the asymptotic cost of repeatedly computing the item indicators from scaling with the growing dataset size to scaling with the number of newly added records instead. The slight growth of the incremental runtime occurs due to the accumulation of more potential cooccurrences, their number is eventually bounded however as detailed in Section 3.3.

6.2 Comparison Against Existing Item-Based Recommenders

In the next set of experiments, we compare our proposed approach with existing recommendation libraries. It turned out to be surprisingly difficult to find a publicly available implementation of an incremental item-based collaborative filtering approach. Both *StreamRec* [7] and *RecStore* [21] are not available as open source, and other established libaries such as *Apache Mahout* [32], *Lenskit* [13], *MyMediaLite* [15] and *RecDB*[8] [30] do not support incremental training for item-based recommenders out-of-the-box. We show (i) that existing libraries have scalability issues, and cannot even handle medium-sized datasets; (ii) that it is not sufficient alone to use an incremental approach, but that it is very important to carefully choose the applied data structures and update strategies in order to tackle said scalability issues.

We first present an experiment for the non-incremental case, comparing our approach to the recommendation libraries *RecDB* and *MyMediaLite*, which do not support incremental training. We aim to show that standard implementations of item-based collaborative filtering have scalability issues (even in the non-incremental case), and that the interaction cuts and careful consideration of the applied data structures are already necessary for handling medium-sized datasets in cooccurrence-based computations. We conduct an end-to-end comparison for the *Movielens1M*, *DBLP* and *Twitter* datasets, where we compute the item indicators and sequentially retrieve 10 recommended items for every user in the dataset afterwards. The end-to-end comparison is necessary as RecDB does not allow us to conduct the training alone and access the resulting indicators. As these libraries do not support loglikelihood-ratio-based tests, we run them with cosine-based item similarities (which are computationally cheaper than our approach because no logarithms need to be computed). We summarize the experimental outcome in Table 2. Our Rust-based implementation conducts the training for *Movielens1M* in 18s, and produces recommendations for all 6040 users in less than 500ms. We compute the recommendations by identifying the items that most often occur in the indicators of the user's history items. In the case of the *DBLP* dataset, the training takes less than four minutes (234s) and we compute the recommendations for $1,314,051$ users in 6.6s. Finally, we train the model for the *Twitter* data in less than two minutes (114s) and take 18s to compute recommendations for all $3,070,055$ users.

Comparison to RecDB. For our evaluation of RecDB, we implement the experiment by a adapting a python script shipped with RecDB. We import the interaction data via a `COPY` statement, create a recommender for the resulting table, and ask for recommendations for each user afterwards with the `RECOMMEND` clause provided by RecDB. For the *Movielens1M* data, the import and recommender setup takes less than 30s, and each query for 10 recommendations for a particular user returns in about 1.1s. For both the *DBLP* and *Twitter* dataset, we abort the experiment after one hour as the `CREATE RECOMMENDER` statement did not return in that time.

Comparison to MyMediaLite. We repeat the end-to-end comparison experiment for MyMediaLite, where we leverage its 'ItemKNN' algorithm to compute 10 recommendations for each user in the data using item-based collaborative filtering with cosine similarity. MyMediaLite trains on *Movielens1M* in 87s and produces recommendations for all users in 200ms. On the *DBLP* and *Twitter* data, it unfortunately crashes due to a lack of memory, even though it has 16GB of RAM available.

The fact that both RecDB and MyMediaLite fail to process the *DBLP* and *Twitter* datasets (which both have a size of less than 200MB when stored as gzip-compressed CSV files) highlights the importance to carefully choose appropriate data structures for cooccurrence-based computations, especially for extremely sparse but high dimensional data that is commonly encountered in real world use cases. Our Rust implementation carefully allocates memory when necessary for the intermediate data structures, and is therefore able to compute the indicators for each of these datasets in less than four minutes.

Comparison to Mahout Taste. In order to have a comparison against an external library for the incremental case, we decided to compare our single machine implementation to a customly designed, incrementalized implementation of the standard item-based approach [28] from the *Taste* recommendation library[9], which is part of *Apache Mahout*. (Note that Taste, in contrast to most other algorithms in Mahout, is a purely Java-based implementation that does not rely on map-reduce). As incremental updates are not naturally supported in Taste, we mimic this behavior by implementing an updatable version of Taste's `DataModel` class, which holds the

[8]RecDB supports incremental training in principle, however this is currently not implemented in the open source version as the authors confirmed via mail).

[9]https://github.com/apache/mahout/blob/master/mr/src/main/java/org/apache/mahout/cf/taste

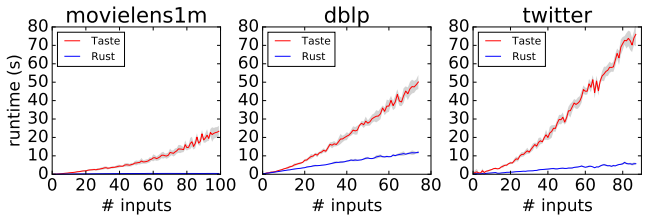

Figure 7: Comparison of incremental indicator computation in Mahout Taste and our Rust implementation. Taste exhibits a much steeper runtime growth pattern although both approaches apply the interaction cuts.

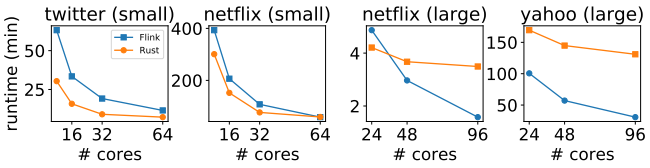

Figure 8: *Left*: For a small batch size of 10K, the single machine strictly outperforms Flink. *Right:* Flink scales well with large batch sizes of 1M and 10M, and outperforms the single machine as it efficiently parallelizes updates of both the history matrix and the cooccurrence matrix.

interaction data in hashmaps, and recompute the indicators using multiple threads with Taste's `MostSimilarEstimator` from the `GenericItemBasedRecommender` class, employing an implementation of loglikelihood-ratio tests via the `LoglikelihoodSimilarity` class. We evaluate both approaches on the *Movielens1M*, *DBLP* and *Twitter* datasets, which we again partition into inputs of 10,000, 250,000 and 500,000 interactions. Figure 7 illustrates the resulting processing times per input over time. While both approaches manage to handle the incremental computations, we see vastly different growth patterns in the runtime, although both Mahout and our implementation apply the interaction cuts. The main difference however is that our algorithm carefully updates not only the history matrix but also the cooccurrence matrix, which is not supported by Taste (e.g, Taste does not provide an abstraction for that). Instead, the `MostSimilarEstimator` from Taste conducts a search in the history matrix each time we recompute the item indicators. This subtle difference has a tremendous effect on the resulting runtimes, e.g., for the last batch in Movielens1M, our implementation requires 0.4 seconds while Taste needs 23 seconds, and this is true for the other datasets as well (12s versus 50s for DBLP, and 6s versus 76s in case of the Twitter data).

6.3 Enforcing Interaction Cuts

In this experiment, we empirically validate that our implementations of the algorithm for updating the history matrix (Algorithm 1 in Section 3.3) correctly enforce the desired user interaction cut and item interaction cut. We generate a synthetic dataset from two independent two-parameter Poisson–Dirichlet distributions with $\alpha = 6000$ and $d = 0.3$. These distributions are helpful for generating power law distributed data, and have the property that the number of unique elements will increase roughly according to αT^d where T is the number of samples taken. Also, the fraction of elements that have only been seen once will be roughly equal to d. We take 100 million samples. Note that we choose the parameters so that after these 100 million samples, the number of unique elements $(1-d)\ \alpha\ T^d$ is approximately 1 million. We apply our single machine implementation of Algorithm 1 to this data with both interaction cuts set to 100, 200, 300 and 500 respectively. After that, we compute the cumulative distribution function of item interaction counts and user interaction counts for the resulting history data. We find that the approach works as intended as the probability of seeing a user or item with more interactions than dictated by the cut drops to zero directly at the corresponding threshold. We repeat this experiment for our distributed implementation in Flink and analogously observe that the desired bounds hold in the resulting data.

6.4 Scalability

In this section, we investigate how our single machine and distributed implementations handle growing machine sizes (scale-up of the single machine variant) or cluster sizes (scale-out of the distributed variant). In all experiments, we fix the input size and increase the number of cores and main memory. We apply our implementations to the Twitter dataset with 27M interactions for 3M items, the Netflix dataset which has more than 100M interactions for 17K items and to the Yahoo Music dataset which contains approximately 700M interactions for 136K items. Our aim in these experiments is not to benchmark the potential systems overhead of a distributed system, but to explore use cases (in relation to the applied mini-batch size) where it is beneficial to move to a distributed solution, which provides fault tolerance and elastic scaling of the underlying hardware out-of-the-box. All experiments are executed on virtual machines running on the Google Cloud Platform. Data is consumed directly from Google Cloud Storage for the distributed Flink experiments, and from the locally attached disk in the single machine experiments. The cloud instances run debian-9, and we use Rust 1.20 for the single machine implementation and Apache Flink 1.3.2 with the HotSpot JVM 1.8.0_144 for the distributed cases.

Scaling with small mini-batches. In our first experiment, we execute the incremental indicator computation with both Rust and Flink on the Twitter and Netflix datasets. We leverage a `n1-highcpu-64` instance with 64 cores for the single machine implementation and a cluster of `n1-standard-8` instances with 8 cores and 30GB of main memory per machine for Flink. We apply a very small mini-batch size: we partition the datasets into inputs of 10,000 interactions and have both variants consume these to incrementally update the resulting indicators and cooccurrences. We repeat this computation for setups of 8, 16, 32 and 64 cores and plot the resulting runtimes in the left part of Figure 8. We see a strong speedup from 8 to 16 cores, which quickly flattens when further increase the number of cores which we attribute to decreasing cache locality. In this setup the single machine implementation always outperforms the Flink implementation which has additional network communication overheads.

Scaling with large mini-batches. We repeat the scaling experiment for the Netflix and Yahoo Music data, and increase the batch size by several orders of magnitude, e.g., we partition these datasets into inputs of 1 million and 10 million interactions and run the incremental indicator computation with 24, 48 and 96 cores. For the single machine implementation, we leverage a `n1-highcpu-96` instance while we run the Flink experiments on cluster of `custom-24-22272` instances with 24 cores and 22GB of memory each. The resulting runtimes are shown in the rightmost plots of Figure 8. The runtimes for the single machine implementation look similar to the previous experiment, as we see a quickly flattening speedup for larger numbers of cores. However, the Flink implementation shows a different behavior in this setup: It achieves better speedups than the single machine implementation and manages to outperform the single machine implementation by a factor of more than two for 96 cores, despite of the overheads incurred by a distributed system. We attribute this difference to the fact that the Flink implementation also parallelizes the updates of the history matrix (which is executed sequentially in the Rust code). This has a much higher impact on the runtime for large batch sizes. We conclude from these findings that our single machine implementation should be preferred for use cases with small batch sizes which requires frequent updates; if we encounter memory pressure or can live with larger batch sizes, a distributed implementation becomes the preferred choice.

7 CONCLUSION

We have described and characterized an efficient, scalable algorithm for incremental item-based collaborative filtering with cooccurrence analysis. We outlined generic scalability issues in item-based recommenders and related these issues to well-known densification processes in interaction datasets. Our approach deals with these scalability issues by introducing interaction cuts and carefully choosing data structures for intermedate results to guarantee a provably constant amount of work per interaction to process. We described how to replace the common repeated offline recomputation of the model with an efficient online algorithm that updates only the parts of the recommendation model which are affected by new interactions. We have demonstrated the substantial benefits of an efficient incremental algorithm, with both scale-up and scale-out reference implementations. Finally, we conducted an experimental evaluation of our approach. Our results confirmed that the incremental approach is dramatically faster than repeated batch recomputation. We found that our implementation commonly outperforms existing libraries by an order of magnitude or more on many datasets, and scales well to medium-sized and large datasets which some existing libraries fail to process. In future work, we would like to investigate the incorporation of multiple interaction types and optimizations based on thresholds for the LLR scores. Another interesting direction is to focus on the serving part (particularly efficient updates) of recommendation systems.

We would like to thank Frank McSherry for advice on optimizing the Rust implementation of our algorithm. This work was supported by the Moore-Sloan Data Science Environment at New York University.

REFERENCES

[1] Jacob Abernethy, Kevin Canini, John Langford, and Alex Simma. 2007. Online collaborative filtering. *University of California at Berkeley, Tech. Rep.*
[2] Xavier Amatriain. 2012. Building Industrial-scale Real-world Recommender Systems. *RecSys*, 7–8.
[3] Xavier Amatriain. 2013. Mining large streams of user data for personalized recommendations. *SIGKDD* 14, 2, 37–48.
[4] Albert-László Barabási and Réka Albert. 1999. Emergence of scaling in random networks. *Science* 286, 5439, 509–512.
[5] Robert M. Bell and Yehuda Koren. 2007. Lessons from the Netflix prize challenge. *SIGKDD* 9, 75–79.
[6] Paris Carbone, Stephan Ewen, Gyula Fóra, Seif Haridi, Stefan Richter, and Kostas Tzoumas. 2017. State management in Apache Flink®: consistent stateful distributed stream processing. *PVLDB* 10, 12, 1718–1729.
[7] Badrish Chandramouli, Justin J Levandoski, Ahmed Eldawy, and Mohamed F Mokbel. 2011. StreamRec: a real-time recommender system. *SIGMOD*, 1243–1246.
[8] Abhinandan S Das, Mayur Datar, Ashutosh Garg, and Shyam Rajaram. 2007. Google news personalization: scalable online collaborative filtering. *WWW*, 271–280.
[9] James Davidson, Benjamin Liebald, Junning Liu, Palash Nandy, Taylor Van Vleet, Ullas Gargi, Sujoy Gupta, Yu He, Mike Lambert, Blake Livingston, and Dasarathi Sampath. 2010. The YouTube video recommendation system. *RecSys*, 293–296.
[10] Ernesto Diaz-Aviles, Lucas Drumond, Lars Schmidt-Thieme, and Wolfgang Nejdl. 2012. Real-time top-n recommendation in social streams. *RecSys*, 59–66.
[11] Ted Dunning. 1993. Accurate methods for the statistics of surprise and coincidence. *Computational Linguistics* 19, 1, 61–74.
[12] Ted Dunning and Ellen Friedman. 2014. *Practical Machine Learning: Innovations in Recommendation*. O'Reilly Media, Inc.
[13] Michael D Ekstrand, Michael Ludwig, Jack Kolb, and John T Riedl. 2011. LensKit: a modular recommender framework. *RecSys*, 349–350.
[14] Stephan Ewen, Kostas Tzoumas, Moritz Kaufmann, and Volker Markl. 2012. Spinning fast iterative data flows. *PVLDB* 5, 11, 1268–1279.
[15] Zeno Gantner, Steffen Rendle, Christoph Freudenthaler, and Lars Schmidt-Thieme. 2011. MyMediaLite: A Free Recommender System Library. *RecSys*.
[16] Yanxiang Huang, Bin Cui, Wenyu Zhang, Jie Jiang, and Ying Xu. 2015. TencentRec: Real-time Stream Recommendation in Practice. *SIGMOD*, 227–238.
[17] Dietmar Jannach and Malte Ludewig. 2017. When recurrent neural networks meet the neighborhood for session-based recommendation. *RecSys*, 306–310.
[18] Yehuda Koren, Robert Bell, and Chris Volinsky. 2009. Matrix factorization techniques for recommender systems. *Computer* 42, 8.
[19] Jérôme Kunegis, Ernesto De Luca, and Sahin Albayrak. 2010. The Link Prediction Problem in Bipartite Networks. *Computational Intelligence for Knowledge-Based Systems Design*, 380–389.
[20] Jure Leskovec, Jon Kleinberg, and Christos Faloutsos. 2007. Graph Evolution: Densification and Shrinking Diameters. *TKDD* 1, 1.
[21] Justin J Levandoski, Mohamed Sarwat, Mohamed F Mokbel, and Michael D Ekstrand. 2012. RecStore: an extensible and adaptive framework for online recommender queries inside the database engine. *EDBT*, 86–96.
[22] Nathan N Liu, Min Zhao, Evan Xiang, and Qiang Yang. 2010. Online evolutionary collaborative filtering. *RecSys*, 95–102.
[23] John H McDonald. 2009. *Handbook of biological statistics*. Vol. 2.
[24] Sean M McNee, John Riedl, and Joseph A Konstan. 2006. Being accurate is not enough: how accuracy metrics have hurt recommender systems. *CHI*, 1097–1101.
[25] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. 2013. Distributed representations of words and phrases and their compositionality. *NeurIPS*, 3111–3119.
[26] Neoklis Polyzotis, Sudip Roy, Steven Euijong Whang, and Martin Zinkevich. 2018. Data Lifecycle Challenges in Production Machine Learning: A Survey. *ACM SIGMOD Record* 47, 2, 17–28.
[27] Francesco Ricci, Lior Rokach, Bracha Shapira, and Paul B. Kantor. 2011. *Recommender Systems Handbook*.
[28] Badrul Sarwar, George Karypis, Joseph Konstan, and John Riedl. 2001. Item-based collaborative filtering recommendation algorithms. *WWW*, 285–295.
[29] Mohamed Sarwat, James Avery, and Mohamed F Mokbel. 2013. RecDB in action: recommendation made easy in relational databases. *PVLDB* 6, 12, 1242–1245.
[30] Mohamed Sarwat, Raha Moraffah, Mohamed F Mokbel, and James L Avery. 2017. Database system support for personalized recommendation applications. *ICDE*, 1320–1331.
[31] Sebastian Schelter, Felix Biessmann, Tim Januschowski, David Salinas, Stephan Seufert, and Gyuri Szarvas. 2018. On challenges in machine learning model management. *Data Engineering*, 5.
[32] Sebastian Schelter, Christoph Boden, and Volker Markl. 2012. Scalable similarity-based neighborhood methods with mapreduce. *RecSys*, 163–170.
[33] Sebastian Schelter, Venu Satuluri, and Reza Zadeh. 2014. Factorbird-a parameter server approach to distributed matrix factorization. *Distributed Machine Learning and Matrix Computations workshop at NeurIPS*.