Efficient Similarity Computation for Collaborative Filtering in Dynamic Environments

Olivier Jeunen
University of Antwerp
Belgium

Koen Verstrepen
Froomle
Belgium

Bart Goethals
University of Antwerp, Belgium
Froomle, Belgium
Monash University, Australia

ABSTRACT

The problem of computing all pairwise similarities in a large collection of vectors is a well-known and common data mining task. As the number and dimensionality of these vectors keeps increasing, however, currently existing approaches are often unable to meet the strict efficiency requirements imposed by the environments they need to perform in. Real-time neighbourhood-based collaborative filtering (CF) is one example of such an environment in which performance is critical.

In this work, we present a novel algorithm for efficient and exact similarity computation between sparse, high-dimensional vectors. Our approach exploits the sparsity that is inherent to implicit feedback data-streams, entailing significant gains compared to other methods. Furthermore, as our model learns incrementally, it is naturally suited for dynamic real-time CF environments. We propose a MapReduce-inspired parallellisation procedure along with our method, and show how even more speed-up can be achieved. Additionally, in many real-world systems, many items are actually not *recommendable* at any given time, due to recency, stock, seasonality, or enforced business rules. We exploit this fact to further improve the computational efficiency of our approach. Experimental evaluation on both real-world and publicly available datasets shows that our approach scales up to millions of processed user-item interactions per second, and well advances the state-of-the-art.

CCS CONCEPTS

• **Information systems** → **Collaborative filtering**; **Recommender systems**; • **Theory of computation** → *Online algorithms*; • **Computing methodologies** → *Learning from implicit feedback*;

KEYWORDS

Nearest-neighbours; incremental algorithms; distributed algorithms

ACM Reference format:
Olivier Jeunen, Koen Verstrepen, and Bart Goethals. 2019. Efficient Similarity Computation for Collaborative Filtering in Dynamic Environments. In *Proceedings of Thirteenth ACM Conference on Recommender Systems, Copenhagen, Denmark, September 16–20, 2019 (RecSys '19),* 9 pages.
https://doi.org/10.1145/3298689.3347017

RecSys '19, September 16–20, 2019, Copenhagen, Denmark

ACM ISBN 978-1-4503-6243-6/19/09…$15.00
https://doi.org/10.1145/3298689.3347017

1 INTRODUCTION

Many important recommender system use-cases are highly dynamic in nature: news, movie, music or retail recommenders all want to incorporate new behaviour into their models as quickly as possible. With new user-item interactions arriving at high rates, the need for dynamic models that can efficiently handle incremental updates in approximately real time becomes more and more apparent [9]. In the context of highly dynamic environments where items have limited lifetimes, this issue becomes even more pressing. News websites typically only want to recommend recent articles, and interactions with newly written articles need to be incorporated into the model as quickly as possible. Auction websites frequently deal with items that are only available for a few days and face the same concerns. Many more examples exist. Traditional Collaborative Filtering (CF) approaches fall short in this setting, as frequent model updates often become too time consuming. Typically, the entire CF model will be retrained at certain fixed points in time, after which the updated model is then deployed. For highly dynamic use-cases, the time between subsequent model updates should ideally be kept minimal, in order to allow information from new incoming user-item interactions to be incorporated into the recommendation process as soon as possible. However, as more and more data arrives, the iterative recomputation of the entire model becomes more and more costly as well, putting a hard upper limit on the frequency with which model updates can be performed. We see a fundamental divide here, and such a trade-off is unacceptable for many present-day applications. A clear need arises for CF models that can instantaneously process new transactions and incorporate them into the model in an incremental manner, while avoiding the periodical re-processing of old data.

In this paper, we present a novel exact algorithm to tackle the problem of efficient similarity computation for high-dimensional and fast changing sparse implicit feedback data streams. Such algorithms are at the basis of nearest-neighbour-based CF techniques, which have recently been shown to attain competitive results with more advanced state-of-the-art approaches, such as recurrent neural networks [7]. On top of this, they provide naturally explainable recommendations [22]. As a consequence, they remain a popular approach to recommendation. Currently existing alternative methods for efficient similarity computation often make use of approximations, sacrificing accuracy for efficiency [6, 12, 21]. Our algorithm, on the other hand, computes all *exact* item-item similarities. The algorithm learns incrementally, making it naturally suitable for real-time CF environments. We exploit the data's sparsity to avoid unnecessary iterative computations and propose the use of an inverted index to quickly identify affected pairs of items

when updates arrive. Our approach is presented in a MapReduce-inspired formulation, demonstrating its scalability.

As the number of users and items in present-day real world systems quickly scales up to hundreds of thousands and millions, it often becomes undesirable or unnecessary to keep updated recommendation scores for all catalogued items in the database. Again, in the case of a news website recommendation engine, scores for old articles will be irrelevant as only *recent* items are allowed to be recommended. Or, in the case of a retail environment: recommending items that are currently out of stock is to be avoided. Media recommenders that deal with expiring licenses encounter the same issues. As such, for many different use-cases, the set of *recommendable* items at a given time is a much smaller subset of the full item collection. This imbalance is exploited by our algorithm, as we compute and maintain recommendation scores only for those items that are recommendable. We show that incorporating this natural aspect into our algorithm has dramatic effects on system throughput.

To summarise, the main contributions of this paper are:

(1) We introduce a novel algorithm, called "Dynamic Index", for efficiently computing all pairwise similarities in a collection of sparse high-dimensional vectors, which are typical for recommender systems.
(2) Our approach learns incrementally, making it suitable for real-time environments.
(3) We further exploit non-recommendable items to improve the computational efficiency of our method.
(4) By presenting our algorithm in a MapReduce-inspired formulation, it is easily parallellised and scalable.
(5) Experimental results on real-world data demonstrate the efficiency and performance of our methodology.[1]

2 RELATED WORK

Nearest-neighbour or similarity join processing is not a new problem, and has been thoroughly investigated in the last 15 to 20 years. Most recent trends for speeding up computation tend to either focus on approximate solutions [12], distributed algorithms [31, 32] or incremental approaches [25, 30]. The first notable work in the latter area is the kNNJoin$^+$ algorithm [30], which uses the *iDistance* similarity measure [28, 29] and a *Sphere-tree* index to efficiently reduce the high-dimensional search to a single dimension. However, when updating two points i and j, the distance between these two points still needs to be re-evaluated in the high-dimensional space before the index can be updated to enable efficient nearest neighbour search. Moreover, this work was aimed at a dimensionality ranging from 20 to 50 and only 100 000 data points, whereas we focus on much larger but very sparse datasets consisting of millions of dimensions, as is typical for recommender systems.

Yang et al. propose a method called *HDR-tree* for incrementally updating nearest neighbour joins in the context of recommender systems [25], exploiting the distance-preserving properties of Principal Component Analysis (PCA). Their algorithm focuses on content-based filtering with a strict window size of recent items that they consider for recommendations, whereas our algorithm focuses on collaborative filtering with a much more flexible set of recommendable items that can change over time. Furthermore, they require a fixed set of users, which is too restrictive for the more typical setting we consider. In the context of CF algorithms for streaming scenarios, multiple online learning approaches for matrix factorization, learning-to-rank and neural network models have been presented as well [5, 17, 23, 24]. Several incremental or online learning algorithms specifically for nearest-neighbour-based CF models have also been published in recent years. Liu et al. propose an incremental learning algorithm that includes temporal information in their novel similarity measure to tackle concept drift in users' preferences over time [10]. The work of Luo et al. focuses on reducing model storage complexity and increasing rating prediction accuracy by incrementally learning biases on top of similarities [11]. TencentRec is a framework implementing several well-known recommendation algorithms in a streaming environment to provide real-time recommendations [6]. Their variant prunes probable dissimilar items, leading to an approximate solution instead of an exact one. Another neighborhood-based approach is proposed by Subbian et al., where a probabilistic data structure is used to approximate item-item similarities and provide recommendations in a real-time manner [21]. Sreepada and Patra present a novel similarity measure that is incrementally learned more easily than other common similarity measures, called *item tendency* [20].

However, most of the above-mentioned methods [10, 11, 20, 21] rely on explicit-feedback data, which is vastly different than the implicit-feedback data use-case we tackle with this work in terms of similarity measure computation as well as general aspects of the dataset. Moreover, several of these methods [6, 21] use approximations to speed up computation time, at the cost of similarity- (and as a consequence recommendation-) accuracy. In this work, we focus on the task of *exact* nearest-neighbour and similarity computations from implicit-feedback data, without the use of any approximations or need of explicit rating data. In addition, with our approach, non-relevant items or users are not considered at computation time, which allows us to work directly on the high-dimensional space, as we can take maximal advantage of the highly sparse nature of the data. Finally, as our algorithm only needs a simple inverted index to efficiently identify affected pairs of items when updates arrive, we can formulate it in accordance with the MapReduce paradigm, ensuring scalability through parallel processing [2].

3 BACKGROUND

3.1 Preliminaries

Let U be a set of m users and I a set of n items. Our work focuses on transactional data with implicit feedback. More specifically: we work with a set of user-item pairs $(u, i) \in U \times I$ denoting that user u has *consumed* item i, be it in the form of a product purchase, a movie streaming, a click on a news article or otherwise. We call such preference expressions *pageviews*, and represent them as a tuple (u, i, t_c), where t_c denotes the consumption time. The set of all pageviews up to, but not including time t is denoted by $\mathcal{P}_t$. We can represent these pageviews in the form of a sparse user-item matrix $\mathbf{P}_t \in \{0, 1\}^{m \times n}$ for m unique users and n unique items. We omit the timestamp t when it is clear from context. Rows in this matrix are users represented by the items they have consumed, and vice versa for columns: $\mathbf{P}_{u,i} = 1$ if and only if user u has consumed

[1] Code available at: https://github.com/olivierjeunen/dynamicindex

Algorithm 1 Naive Baseline

Input: A set of pageviews $|\mathcal{P}_t|$.
Output: An inverted index from items to users $\mathcal{K}$, a matrix of item similarities $\mathbf{S}$.

1: $\mathcal{K} \leftarrow \emptyset$, $\mathbf{S} \leftarrow \mathbf{I}$
2: **for** $(u, i, t_c) \in \mathcal{P}_t$ **do**
3: $\quad \mathcal{K}[i] = \mathcal{K}[i] \cup \{u\}$
4: **for** $i \in \mathcal{K}$ **do**
5: $\quad$ **for** $j \in \mathcal{K}$ **do**
6: $\quad\quad$ **if** $i < j$ **then**
7: $\quad\quad\quad \mathbf{S}_{i,j} \leftarrow \frac{|\mathcal{K}[i] \cap \mathcal{K}[j]|}{\sqrt{|\mathcal{K}[i]|} \cdot \sqrt{|\mathcal{K}[j]|}}$
8: **return** $\mathcal{K}, \mathbf{S}$

item i and $\mathbf{P}_{u,i} = 0$ otherwise. When we represent an item i by the i-th column-vector of the matrix $\mathbf{P}_t$, we denote it as $\vec{i}$. The set of users that have consumed a specific item $i \in I$ is denoted as U_i. Vice versa, the set of items that a certain user $u \in U$ has consumed is denoted as I_u.

Between items $i, j \in I$, similarity is expressed as the well-known cosine similarity: $\cos(\vec{i}, \vec{j})$. The goal at hand is to efficiently and incrementally compute and store the similarity $\cos(\vec{i}, \vec{j})$ for every such item-pair. In a worst-case scenario, this would incur a memory overhead of $\frac{n \cdot (n-1)}{2}$ item similarities that need to be stored. In many real world datasets, the user-item matrix $\mathbf{P}$ is extremely sparse. For many implicit-feedback datasets, this can lead to sparsity in the item co-occurence matrix $\mathbf{M}$. We denote the sparsity of any matrix by the function $\sigma(\cdot)$. Partially to alleviate spatial complexity issues, and partially to exploit this inherent sparseness and avoid unnecessary iterative computations on zero-values, we propose the use of sparse data-structures throughout the algorithms presented in the following sections. Finally, familiarity with item-based nearest neighbour collaborative filtering approaches is assumed [19].

3.2 Baseline Approaches

The naive approach to computing cosine similarities between pairs of items occurring in a given set of pageviews $\mathcal{P}_t$ is laid out in Algorithm 1. First, an inverted index from every item i to the set of users that have seen that item, U_i, is constructed. Subsequently, the algorithm iterates over said sets of users for every item-pair $i, j \in I$ and computes the sparse dot-product $\vec{i} \cdot \vec{j}$, which is equivalent to the intersection of their user-sets $|U_i \cap U_j|$. Because of the symmetric nature of our similarity measure, only half of the iterations lead to actual computations (line 6). Note that only the *size* of the intersection needs to be computed, and not the set intersection itself. Efficient algorithms with linear time complexity exist for this operation over sorted inverted indices. However, the naive baseline approach explicitly computes all $\frac{n \cdot (n-1)}{2}$ sparse vector dot-products, even when a significant amount of them are irrelevant. For many (sparse) real world datasets, this is extremely inefficient.

An improved baseline, specifically tuned to the setting of sparse data is presented in Algorithm 2. On top of the original item-to-user inverted index, we now construct a user-to-item inverted index as well. As a result, we can deconstruct the sparse vector dot-product, and iteratively count which item-pairs $i, j \in I$ also appear in I_u for every user $u \in U_i$. As $|U_i| \ll |U|$ in sparse datasets, this entails a significant efficiency advantage. Note that this baseline is less

Algorithm 2 Sparse Baseline

Input: A set of pageviews $|\mathcal{P}_t|$.
Output: An inverted index from items to users $\mathcal{K}$, an inverted index from users to items $\mathcal{L}$, a matrix of item similarities $\mathbf{S}$.

1: $\mathcal{K} \leftarrow \emptyset$, $\mathcal{L} \leftarrow \emptyset$, $\mathbf{S} \leftarrow \mathbf{I}$
2: **for** $(u, i, t_c) \in \mathcal{P}_t$ **do**
3: $\quad \mathcal{K}[i] = \mathcal{K}[i] \cup \{u\}$
4: $\quad \mathcal{L}[u] = \mathcal{L}[u] \cup \{i\}$
5: **for** $i \in \mathcal{K}$ **do**
6: $\quad$ **for** $u \in \mathcal{K}[i]$ **do**
7: $\quad\quad$ **for** $j \in \mathcal{L}[u]$ **do**
8: $\quad\quad\quad$ **if** $i < j$ **then**
9: $\quad\quad\quad\quad \mathbf{S}_{i,j}$ += 1
10: **for** $i, j \in \mathbf{S}$ **do**
11: $\quad$ **if** $\mathbf{S}_{i,j} > 0$ **then**
12: $\quad\quad \mathbf{S}_{i,j}$ /= $\sqrt{|\mathcal{K}[i]|} \cdot \sqrt{|\mathcal{K}[j]|}$
13: **return** $\mathcal{K}, \mathcal{L}, \mathbf{S}$

memory efficient than the naive baseline, as it needs a second inverted index to efficiently exploit the sparse nature of the data. In both baseline algorithms, the square roots of the item-norms $\sqrt{|U_i|}$ can be pre-computed for improved efficiency.

4 METHODOLOGY

4.1 Recommendable Items

Traditionally, recommender systems are seen as functions that predict some relevance score specific to a user-item pair: $f_{\mathbf{P}} : U \times I \rightarrow [0, 1]$. Here, the recommender system represented by the function f is dependent on the user-item matrix $\mathbf{P}$, hence the subscript. In real-world present-day systems, the number of users and items can quickly scale up to hundreds of thousands and even millions. It is clear that the model represented by the function $f_{\mathbf{P}}$ becomes much more complex to compute and will take up much more memory to store in the case of ever-growing user- and item-sets and the matrix $\mathbf{P}$. We identify two possible methods to alleviate this issue: either reduce the size of the training matrix $\mathbf{P}$, or reduce the complexity of $f_{\mathbf{P}}$ by putting restrictions on the set of items to compute recommendation scores for. Although the first option opens up interesting directions for future research in how datasets can be optimally summarised with minimal loss of information, we focus on the latter. We define our model as follows: $f_{\mathbf{P}} : U \times R \rightarrow [0, 1]$, where $R \subseteq I$ denotes the set of *recommendable* items. This set can be highly dynamic, and depends on any number of factors such as recency, seasonality, stock and much more. Throughout the rest of this manuscript, R_t will represent the set of recommendable items at time t. When omitted, all items are considered recommendable ($R_t = I$).

4.2 Incremental Similarity Computation

Papagelis et al. present an incremental user-based CF method, focused on explicit feedback [15]. This work has later been adapted by Yang et al. to allow incremental updates of item-based CF methods relying on explicit feedback [27]. Inspired by their work, our work focuses on incremental updates with implicit feedback. We split cosine similarity into three key components and incrementally update these components instead of recomputing the entire similarity

after every update. Equation 1 shows the formula for computing the cosine similarity between two item vectors, where $\vec{i}_k$ represents whether user k has consumed item i.

$$\cos(\vec{i},\vec{j}) = \frac{\vec{i}\cdot\vec{j}}{\left\|\vec{i}\right\|_2\left\|\vec{j}\right\|_2} = \frac{\sum_{k=1}^{m}\vec{i}_k\vec{j}_k}{\sqrt{\sum_{k=1}^{m}\vec{i}_k^2}\sqrt{\sum_{k=1}^{m}\vec{j}_k^2}} \tag{1}$$

In the case of implicit feedback (0's and 1's) from transactional data and the use of sparse data-structures, this formulation can be rewritten as shown in Equation 2. Here, items i and j are no longer explicitly represented by vectors in a user-dimensional space, but rather as sets of users that have consumed these items. These sets can be easily computed from the aforementioned historical transaction data, as they are effectively a sparse column-wise representation of the binary preference matrix $\mathbf{P}$.

$$\cos(\vec{i},\vec{j}) = \frac{|U_i \cap U_j|}{\sqrt{|U_i|}\cdot\sqrt{|U_j|}} \tag{2}$$

Thus, item similarities can be directly computed when the set-intersection between their respective user sets and their set sizes are known. We exploit this formulation to reduce the problem of incrementally updating item similarities to continuously updating $|U_i|$, $|U_j|$ and $|U_i \cap U_j|$ for every pair of items $i, j \in I$. We denote the vector containing all item-vectors' l_1−norms and the matrix containing all item-pair intersections at time t as follows:

$$\mathbf{N}_t \in \mathbb{N}^n : \mathbf{N}_{i,t} = |U_{i,t}|, \text{and}$$
$$\mathbf{M}_t \in \mathbb{N}^{n\times n} : \mathbf{M}_{i,j,t} = |U_{i,t} \cap U_{j,t}|.$$

The final formula for computing the similarity between two items i, j at time t then becomes the following: $\cos(\vec{i}_t,\vec{j}_t) = \frac{\mathbf{M}_{i,j,t}}{\sqrt{\mathbf{N}_{i,t}}\cdot\sqrt{\mathbf{N}_{j,t}}}$.

Since $\mathbf{M}$ is a symmetrical matrix, we can further improve performance by using appropriate data structures.

4.3 The Dynamic Index Algorithm

Suppose we have a set of recommendable items R_t at time t. Define $U_t \subseteq U$ as the set of all users u that have *ever* seen an item that is recommendable *at time t*:

$$U_t = \{u | \exists (u,i,s) \in \mathcal{P}_t \wedge i \in R_t\}.$$

Define $\mathcal{A}_t \subseteq \mathcal{P}_t$ as the set of all pageviews by users in that set:

$$\mathcal{A}_t = \{(u,i,s) \in \mathcal{P}_t | u \in U_t\}.$$

$\mathcal{A}_t$ now holds all pageviews that are relevant to the intersections $|U_i \cap U_j|$ where either i or j is a recommendable item. Naturally, when $R_t = I$, $\mathcal{A}_t = \mathcal{P}_t$. Using Algorithm 3, we can compute the co-occurrence matrix $\mathbf{M}$, and thus all pair-wise similarities, efficiently. The algorithm dynamically builds two inverted indices for every user: one for all items recommendable at time t and one for all other items. The idea of dynamically indexing the data rather than doing this in a preprocessing step, is adopted from the work of Sarawagi and Kirpal [18]. This approach enables us to exploit the sparsity that is inherent to the data as we quickly identify those pairs of items that are of interest, i.e. (i, j) where (1) either i or $j \in R_t$, and (2) $|U_i \cap U_j| > 0$, while avoiding unnecessary computations on all other pairs of items. Note that this proposed algorithm is more space-efficient than the Sparse Baseline shown in Algorithm 2: $\mathcal{P}_t$ is indexed only once instead of twice.

Algorithm 3 Dynamic Index

Input: A set of pageviews $\mathcal{P}_t$, a set of recommendable items R_t.
Output: A matrix of item intersections $\mathbf{M}$, a vector of items' l_1−norms $\mathbf{N}$, an inverted index of users to rec. items $\mathcal{L}_r$, an inverted index of users to non-rec. items $\mathcal{L}_n$.

1: $\mathbf{M} \leftarrow \mathbf{0}, \mathbf{N} \leftarrow \mathbf{0}$
2: $\forall u \in U : \mathcal{L}_r[u] \leftarrow \emptyset, \mathcal{L}_n[u] \leftarrow \emptyset$
3: **for** $(u, i, s) \in \mathcal{P}_t$ **do**
4: **for** $j \in \mathcal{L}_r[u]$ **do**
5: $\mathbf{M}_{i,j}$ += 1
6: **if** $i \in R_t$ **then**
7: **for** $j \in \mathcal{L}_n[u]$ **do**
8: $\mathbf{M}_{i,j}$ += 1
9: $\mathcal{L}_r[u] = \mathcal{L}_r[u] \cup \{i\}$
10: $\mathbf{N}_i$ += 1
11: **else**
12: $\mathcal{L}_n[u] = \mathcal{L}_n[u] \cup \{i\}$
13: **return** $\mathbf{M}, \mathbf{N}, \mathcal{L}_r, \mathcal{L}_n$

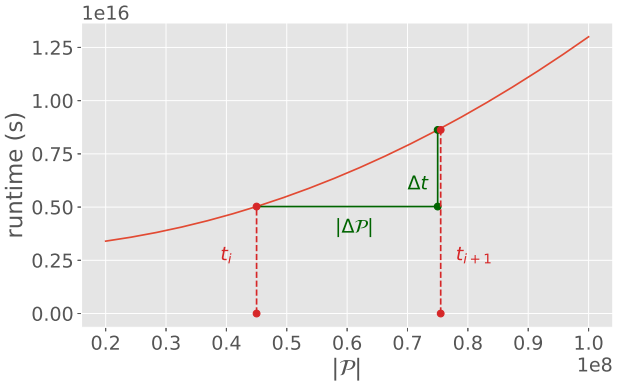

Figure 1: A visualisation of incremental computation, in comparison with the classical iterative variant. As more data becomes available, iterative models need to be retrained from scratch, with computation time t_{i+1}. In contrast, online or incremental methods, can update the existing model after $|\Delta\mathcal{P}|$ new user-item interactions occur, requiring only Δt time.

As the inverted indices are dynamically built, the core algorithm consists of a single *for*-loop over the set of pageviews. Consequently, when a set of new user-item interactions $\Delta\mathcal{P}$ arrives, the model can be updated by executing lines 3-12 from Algorithm 3 on top of the already initialised model computed on the data $\mathcal{P}_t$. As $|\mathcal{P}|$ grows, this benefit becomes increasingly important. Figure 1 provides some visual intuition into this phenomenon.

As we have hinted at before, $\mathbf{M}$ is a symmetrical matrix. We avoid explicitly incrementing $\mathbf{M}_{j,i}$ when incrementing $\mathbf{M}_{i,j}$ since they will be represented as one number in an efficient implementation. Additionally, the dynamically constructed inverted indices $\mathcal{L}_r$ and $\mathcal{L}_n$ do not need to store the sets of items in an ordered manner, improving further on runtime efficiency.

From an existing model $\mathcal{M} = \{\mathbf{M}, \mathbf{N}, \mathcal{L}_r, \mathcal{L}_n\}$, we can compute all recommendable neighbours j of i with their respective

Algorithm 4 Merging two models (*reduce*)

Input: $\mathbf{M}$, $\mathbf{M}'$, $\mathbf{N}$, $\mathbf{N}'$, $\mathcal{L}_r$, $\mathcal{L}'_r$, $\mathcal{L}_n$, $\mathcal{L}'_n$.
Output: $\mathbf{M}$, $\mathbf{N}$, $\mathcal{L}_r$, $\mathcal{L}_n$.

```
M += M'
N += N'
for u ∈ L'_r do
    for i ∈ L'_r[u] do
        for j ∈ L_r[u] do
            M_{i,j} += 1
        for j ∈ L_n[u] do
            M_{i,j} += 1
for u ∈ L'_n do
    for i ∈ L'_n[u] do
        for j ∈ L_r[u] do
            M_{i,j} += 1
∀u ∈ U : L_r[u] = L_r[u] ∪ L'_r[u]
∀u ∈ U : L_n[u] = L_n[u] ∪ L'_n[u]
return M, N, L_r, L_n
```

cosine similarities as follows: $\cos(\vec{i},\vec{j}) = \frac{|U_i \cap U_j|}{\sqrt{|U_i|}\cdot\sqrt{|U_j|}} = \frac{\mathbf{M}_{i,j}}{\sqrt{|U_i|}\cdot\sqrt{\mathbf{N}_j}}$. It should be noted that $|U_i|$ cannot simply be extracted from $\mathbf{N}$, since we have only computed these item norms from $\mathcal{A}_t \subseteq \mathcal{P}_t$. By definition, this vector of item norms will be up to date for recommendable items, but it might not be for non-recommendable items. However, retrieving $|U_i|$ from $\mathcal{P}_t$ is only needed when the actual cosine similarity is important and not just the internal ranking among neighbours. Since all similarities are divided by the constant factor $\sqrt{|U_i|}$, it is trivial to see that the internal ordering will not be impacted by this.

4.4 Parallellisation Procedure

From Algorithm 3, we can see a clear independence between the contribution of different users to the similarity of an item-pair. As $\vec{i}\cdot\vec{j}$ equals the number of users that have consumed both i and j, it is easy to see that a pageview (u, i, s) only has to be correlated with other items j seen by user u. This insight, albeit trivial, allows the computation of Algorithm 3 to be easily and efficiently parallellised following the MapReduce paradigm [2]: if the sets of users processed by every map-process are mutually disjoint, the reduce-process effectively consists of a summation of the different matrices $\mathbf{M}$ and vectors $\mathbf{N}$.

Let $\mathcal{M} = \{\mathbf{M}, \mathbf{N}, \mathcal{L}_r, \mathcal{L}_n\}$ be a model, as obtained through Algorithm 3. Figure 2 visualises the MapReduce-inspired parallellisation procedure we adopt in this work. With n available cores, Algorithm 3 generates n different models in parallel, as shown in the top row of Figure 2. As this step is embarrasingly parallel, this is the so-called Map-procedure. We then go on to recursively merge models in parallel, until we obtain one final model. This is visualised in the subsequent rows of Figure 2, and correlates with the Reduce-procedure. Algorithm 4 presents the process to correctly merge two models $\mathcal{M}$ and $\mathcal{M}'$. After i iterations of parallel reduce-processes have been completed, $\frac{n}{2(i-1)}$ models remain. Ergo, $\log_2(n)$ iterations of parallel reduce steps are required to obtain a single final model. From Algorithm 4, it is clear to see that most of the complexity comes from correlating items that a given user has seen in model $\mathcal{M}'$ with items the same user has seen in $\mathcal{M}$. When parallellising the initial similarity computation, we therefore ensure that the data used for all map-processes and models $\{\mathcal{M}_0, \ldots, \mathcal{M}_n\}$ consists of entirely disjoint sets of users: $|U \cap U'| = \emptyset$. However, for incremental model updates, this is less straightforward: as the new model $\mathcal{M}'$ is trained on newly incoming interactions, we have no way of ensuring that the intersection between U and U' is kept minimal. As a consequence, the computational complexity of the reduce-step grows significantly, and with it the overhead of the parallellisation procedure.

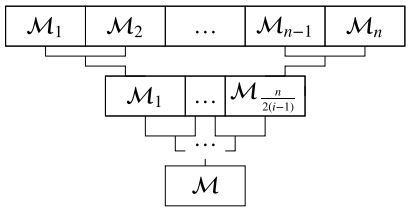

Figure 2: A visualisation of the MapReduce-inspired parallellisation procedure adopted in this work. Assuming n indepedent map-processes, n models $\{\mathcal{M}_1, \ldots, \mathcal{M}_n\}$ are obtained through Algorithm 3, and subsequently recursively merged through Algorithm 4. After i iterations of the reduce step, $\frac{n}{2(i-1)}$ models remain. Consequently, $\log_2(n)$ reduce iterations are required.

4.5 Incremental Model Updates with Dynamic Recommendability

At time $t+1$, the model needs to be updated for a new set of recommendable items R_{t+1}. As the set of recommendable items changes, the set of users with interactions that are relevant to these items needs to be re-evaluated as well. We compute U_{t+1} analogous to the previous iteration: $U_{t+1} = \{u | (u, i, t_c) \in \mathcal{P}_{t+1} : i \in R_{t+1}\}$. $\mathcal{A}_{t+1}$ is initialised as the empty set $\emptyset$. Three different possibilities for every user u in $U_t \cup U_{t+1}$ emerge:

Case $u \in U_t \backslash U_{t+1}$: The user u was relevant in the previous iteration, but no longer is. Since their inverted indices $\mathcal{L}_r[u]$ and $\mathcal{L}_n[u]$ will not be needed during this iteration, we remove them out of memory.

Case $u \in U_t \cap U_{t+1}$: The user u was relevant and still is. As all u's interactions up until time t were already incorporated in the model, we only need to take into account new interactions between time t and $t+1$: $\mathcal{A}_{t+1} = \mathcal{A}_{t+1} \cup \{(u, i, t_c) \in \mathcal{P}_{t+1} \backslash \mathcal{P}_t | u \in U_{t+1} \cap U_t\}$.

Case $u \in U_{t+1} \backslash U_t$: The user u was not relevant during the previous iteration, but has become now. As the model has no record of any interactions by this user, we need to take into account their full history: $\mathcal{A}_{t+1} = \mathcal{A}_{t+1} \cup \{(u, i, t_c) \in \mathcal{P}_{t+1} | u \in U_{t+1} \backslash U_t\}$.

At this point, an updated set of pageviews $\mathcal{A}_{t+1}$ to be incorporated into the model has been computed analogous to Algorithm 3. However, some precautions still need to be taken with relation to the recommendability of items over time. For every item i in $R_t \cup R_{t+1}$, three analogous cases to the ones outlined above occur:

Case $i \in R_t \backslash R_{t+1}$: The item i was recommendable in the previous iteration, but no longer is. We drop all entries in the matrix $\mathbf{M}_{i,j}$ where $j \notin R_{t+1}$. This is important to ensure consistency when the item i would later become recommendable again, otherwise increments might not start at 0. Additionally, we move item i from $\mathcal{L}_r[u]$ to $\mathcal{L}_n[u]$.

Case $i \in R_t \cap R_{t+1}$: The item i was recommendable and still is, nothing needs to be done here.

Case $i \in R_{t+1} \backslash R_t$: The item i was not recommendable during the previous iteration, but has become recommendable now. Since item i might have already been included in the index, we should compute possible intersections $\mathbf{M}_{i,j}$ that were not included in the matrix before. This is true for all users u who have seen item i before time t: $\{u | (u, l, t_c) \in \mathcal{P}_t : l = i\}$. For every non-recommendable item $j \in \mathcal{L}_n[u]$ seen by those users, we increment $\mathbf{M}_{i,j}$. Afterwards, item i has to be deleted from $\mathcal{L}_n[u]$ and inserted into $\mathcal{L}_r[u]$.

If recommendability of items is a monotonically decreasing function over time, one does not have to worry about these issues: $\{(u, l, t_c) \in \mathcal{P}_t : l = i\}$ will be the empty set for items $i \in R_{t+1} \backslash R_t$, since items that *become* recommendable are per definition new in this context. In, for example, a news recommendation setting this makes perfect sense: older articles should not be considered for recommendation. In a retail environment, however, this is not the case: recommendability will often depend on seasonality and current stock.

5 EXPERIMENTAL RESULTS

Table 1 shows the characteristics of the datasets we used to experimentally validate the efficiency of our proposed approach. *Movielens* is the latest well-known Movielens dataset [4], *Netflix* refers to the full dataset that was used for the famous Netflix-Prize [1]. For both movie datasets, we converted explicit ratings to binary implicit feedback, entirely disregarding the actual ratings. *Outbrain* is a dataset containing logs from users and articles they read, published in a recent Kaggle competition [14]. We use a deduplicated version of the first 200 million logged user-item events in our experiments: in the case of recurring user-item pairs, we keep only the earliest entry. *News* is a proprietary real-world dataset consisting of roughly 96 million user-item pairs originating from article reads on the website of a large Belgian newspaper. Our algorithm, as well as the baseline methods, are implemented in C++ and compiled with all the available optimisation flags. Experiments ran on a single Intel Xeon processor. We aim to answer three research questions, respectively covered in the following sections:

(1) Is the proposed Dynamic Index algorithm more efficient than the state-of-the-art in computing similarity between pairs of high-dimensional sparse vectors?
(2) Is the proposed MapReduce-inspired parallellisation procedure effective in reducing the necessary computation time?
(3) What is the impact of restrictions on the set of recommendable items on the efficiency of the algorithm?

5.1 Efficiency of Dynamic Index

To validate the efficiency of our proposed algorithm, we report computation time for the sparse baseline and Dynamic Index, as shown in Figure 3. Both algorithms run on a single computational core. The naive baseline presented in Algorithm 1 is not included in these results, as it is orders of magnitude slower than the Sparse Baseline or Dynamic Index on every dataset we consider. We do not consider other algorithms in our comparison, as other proposed exact approaches in the literature were demonstrated only on dense datasets, covering a few hundred dimensions at most [25, 28–30].

Table 1: Experimental dataset characteristics. Datasets denoted by an asterisk (*) are binarised from explicit-feedback, to mimick the implicit-feedback setting.

	Movielens*	Netflix*	News	Outbrain
$\lvert\mathcal{P}\rvert$	20e6	100e6	96e6	200e6
$\lvert U\rvert$	138e3	480e3	5e6	113e6
$\lvert I\rvert$	27e3	18e3	297e3	1e6
$\overline{\lvert I_u\rvert}$	144.41	209.25	18.29	1.76
$\overline{\lvert U_i\rvert}$	747.84	5654.50	242.51	184.50
$\sigma(\mathbf{P})$	99.46%	98.82%	99.99%	99.99%
$\sigma(\mathbf{M})$	59.90%	0.22%	99.83%	99.98%
$\overline{\mathbf{S}_{i,j} : \mathbf{S}_{i,j} > 0}$	0.050	0.037	0.027	0.012

Our method, aimed towards sparse datasets, can efficiently handle millions of dimensions. Additionally, to the best knowledge of the authors, no competing exact methods or implementations are available at the time of writing.

All datasets were chronologically sorted, and we gradually retrained every algorithm with more data, in order to provide a realistic view of the benefits of online or incremental computation. We can see that for all datasets but *Movielens*, our proposed algorithm significantly outperforms the sparse baseline. *Movielens* has the highest average non-zero similarity between any item-pair, which might make it more suited to Algorithm 2. However, as our method learns incrementally, the potential efficiency gains are much more tangible than merely shown by the area between the two lines in the plot. The improvement of Dynamic Index is most apparent for the largest dataset: our algorithm provides a speedup factor larger than four when all available user-item interactions are considered. Looking at the average number of pageviews processed by Algorithm 3 per second at every point in the plots in Figure 3, we observe throughputs ranging from $14\,500 \frac{|\mathcal{P}|}{\text{s}}$ for the *Netflix* dataset, to more than $834\,000 \frac{|\mathcal{P}|}{\text{s}}$ for *Outbrain*. These numbers effectively represent an upper bound on the number of new incoming pageviews per second the single-core streaming model could process in real-time, assuming a constant-rate influx. From Table 1 and the nature of Algorithm 3, we can deduce interesting observations about the efficiency of our approach. First, as the throughput is highest for those datasets with large $|I|$, it seems this is not an important factor. This may seem counter-intuitive at first, as more unique items will lead to more similarities that have to be computed. However, the sparsity of the co-occurence matrix $\sigma(\mathbf{M})$ is more significant than its absolute dimensions, as we effectively leverage this by avoiding computations on zero-values. The second decisive factor is $\overline{|I_u|}$. As most of the complexity of the algorithm lies in iterating over inverted indices containing user histories, it should come as no surprise that shorter lists imply faster iterations.

5.2 Efficiency of Parallellisation Procedure

To validate the efficiency of our proposed parallellisation procedure, we report runtime results for the same experimental setting as laid out in Section 5.1, for a varying number of available cores. Results from this experiment are visualised in Figure 4. We see a clear benefit from parallellising the computation over multiple cores, over all datasets. For the *Netflix* and *News* datasets, using 8 cores

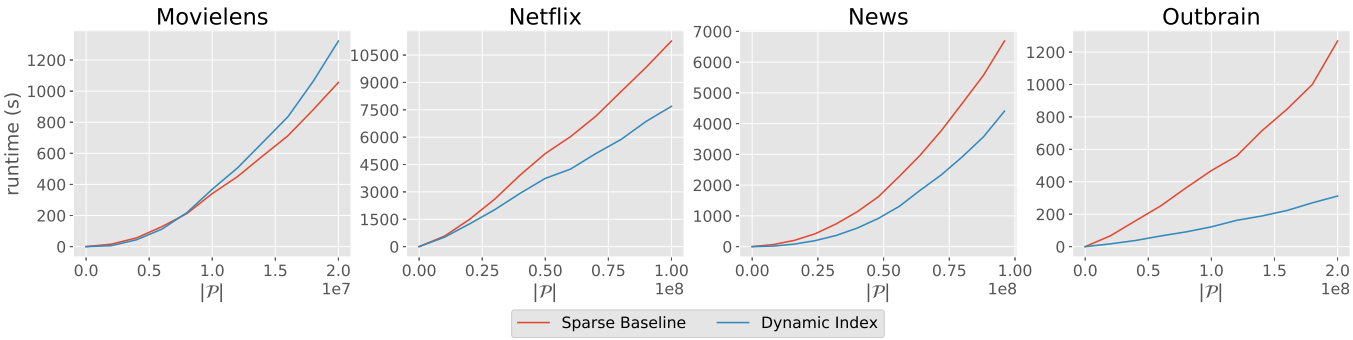

Figure 3: Computation time for the sparse baseline (Algorithm 2) and our proposed Dynamic Index algorithm (Algorithm 3) on the 4 different datasets laid out in Table 1. Both algorithms run on a single core, and all available items in the dataset are considered recommendable ($R_t = I$). We chronologically sorted the available user-item interactions and gradually increased the size of the training data passed to the algorithm (over the x-axis), in order to provide a realistic view of the benefit of online or incremental computation.

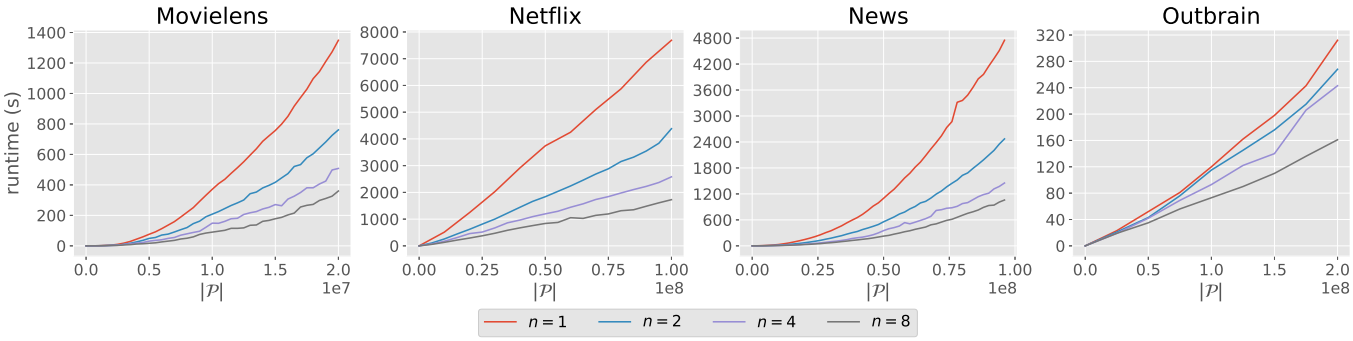

Figure 4: Computation time for our proposed algorithm on the 4 different datasets laid out in Table 1, parallellised over a varying number of computational cores ($n \in \{1, 2, 4, 8\}$). We chronologically sorted the available user-item interactions and gradually increased the size of the training data passed to the algorithm (over the x-axis). However, the model is iteratively retrained as more data becomes available. All items are considered recommendable ($R_t = I$).

provides a speedup larger than factor 4 compared to the single-core variant. The *Outbrain* dataset, which gains the least from the parallellisation scheme, was also the dataset on which the highest throughputs for the single-core algorithm were reported.

As mentioned in Section 4.3, the reduce-step for merging two models in Algorithm 4 is especially efficient when both models were generated from logged interactions by mutually disjoint sets of users. When this condition can not be guaranteed, it becomes significantly more complex. Therefore, when the batch-size $|\Delta\mathcal{P}|$ is small, the single-core variant proves to be more efficient at incremental updates than the parallellised version. However, for sufficiently large $|\Delta\mathcal{P}|$, the bulk of computation time needed to incrementally update the existing model will come from dynamically indexing the new data using Algorithm 3 to generate the new model $\mathcal{M}_{t+1}$, contrary to merging the old model $\mathcal{M}_t$ with $\mathcal{M}_{t+1}$ using Algorithm 4. In these cases, the multi-core variant proves itself to be advantageous. Moreover, in cases where the influx of new data is limited, periodically retraining the model in parallel or performing the incremental updates batch-wise might be more cost-efficient than performing incremental updates in a streaming fashion. Simplistically: if the entire model can be retrained in 20 minutes and an hour of new data can be processed in 1 minute, these options are respectively 3 and 60 times more cost-efficient than a 24/7 streaming solution.

5.3 Efficiency of Restricted Recommendability

Up until now, we have assumed no restrictions on the set of recommendable items. However, as we have argued before, we believe that this will often not hold in real-world applications. Whether based on recency, seasonality, available stock, business rules or any other reason, the set of items that actually should *not* be recommended can grow to be of significant size.

To demonstrate the effect that a varying set of recommendable items R_t can have on the Dynamic Index algorithm, we focus on the news recommendation application. We define δ as the recommendability threshold in this recency-focused setting: when a new item arrives, it remains recommendable for δ hours. After this period has passed, the item is no longer considered newsworthy and should no longer be recommended. Figure 5 shows runtime (top plot) and the number of recommendable items (bottom plot) when the model is retrained iteratively on more data, using the Dynamic Index algorithm with varying thresholds δ. Note that both y-axes are logarithmically scaled. We focus on the case where ample data is available, and show results for the last week in the *News* dataset, where the entire model is iteratively retrained on a growing set of user-item interactions.

Clear performance gains are observed when comparing the results from the restricted-recommendability variants to the unrestricted algorithm ($\delta = \infty$). First, absolute runtimes are decreased massively when focusing on a smaller, yet more relevant, set of items. For $\delta = 48$h, the algorithm computes the exact similarity for all relevant item-pairs in $< 10\%$ of the time needed for $\delta = \infty$. The number of recommendable items, however, still exceeds 17 000, leaving plenty of room for personalisation. With $\delta = 24$h, runtime reduces to $< 5\%$, with more than 8 000 recommendable items. At $\delta = 6h$, these numbers turn to 1.6% of the original runtime, retaining an average of 2 100 recommendable items. The sinusoid pattern that emerges in the bottom plot for low values of δ is an artefact originating from the data: as fewer news articles are published at night, the number of recent items drops and rises periodically.

Second, looking at the slope of the runtime of the unrestricted variant compared to that of all restricted variants, we observe that the latter variants all suffer far less from ever-growing dataset sizes in terms of reduced efficiency. Last, the model size, number of recommendable items, and runtime are highly correlated with δ.

A reasonable question to ask might be how the restricted recommendability impacts the accuracy of the generated recommendations. We did not further explore this due to the following reasons: (1) When recommendability depends on recency, seasonality or available stock, these are often hard-imposed business rules. As a result, restricting recommendability is often not a choice in real–world settings. Our approach deals with this in a flexible way, and effectively exploits the imbalance for improved efficiency. (2) In offline experiments on logged feedback data, a multitude of biases is consistently present [3, 8, 26]. As users are often presented with only *recent* articles on news websites, offline experiments will heavily favour recency-based approaches. On the other hand, presenting users with irrelevant and old news in an online experiment is also inappropriate for obvious reasons.

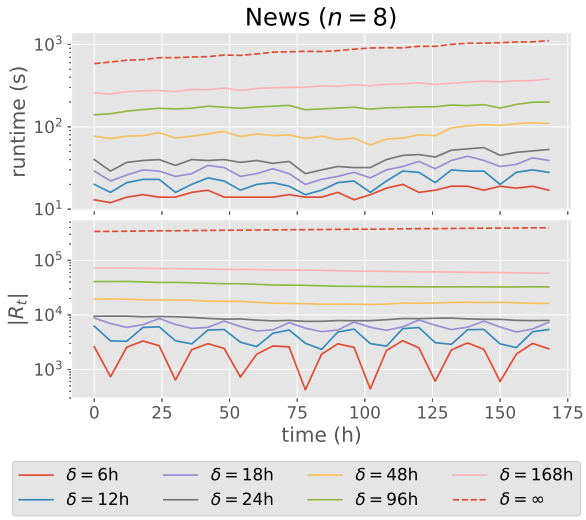

Figure 5: Computation time (top) and number of recommendable items (bottom) for varying recommendability thresholds in the *News* dataset ($n = 8$). δ denotes how long a new item remains recommendable after its first appearance, mimicking the real-world application of news recommendation where recency is critical. Note that both y-axes are logarithmically scaled.

6 CONCLUSION

In this paper, we have motivated and discussed the need for highly dynamic collaborative filtering algorithms that are incrementally updateable in near real-time, to keep up with the highly dynamic environments these algorithms need to perform in. As a step towards this goal, we proposed a novel parallel approach to incrementally compute similarity among high-dimensional vectors, specifically tuned to the inherent sparsity of real-world datasets in a nearest-neighbour collaborative filtering recommender system setting. Our algorithm uses simple inverted indices to quickly identify relevant pairs of items when updates arrive, and as a consequence avoids further unnecessary computations. Moreover, we have formulated our method in accordance with the MapReduce paradigm, making it readily parallellisable and distributable. We have shown that our approach easily scales up to millions of pageviews, and is able to process industrial-sized datasets in a matter of minutes on non-specialised hardware. Attainable processing throughputs vary with configuration and data, but can easily range from tens of thousands to millions of pageviews per second. Our approach is highly scalable and flexible in terms of new users and items arriving over time. We introduced the concept of item recommendability and how it can be exploited to avoid wasting unnecessary computation time for the right use-cases. In our experiments, we effectively increased system throughput by a factor of up to 60 when considering a smaller, yet more relevant set of recommendable items.

As future work, we intend to further experimentally validate the efficiency of incremental updates to our model with non-monotonic recommendability constraints. In an attempt to further improve upon the scalability of CF systems, summarisation algorithms to compress a transactional dataset with minimal information loss, specifically in the context of recommender systems, would be an interesting direction for future research. Furthermore, we intend to look into other similarity functions to determine whether they can be decomposed and incrementally computed as well. As Jaccard Index, Pointwise Mutual Information and Pearson's correlation coefficient all depend on the co-occurrence matrix $\mathbf{M}$, we believe this to be an attainable extension of our work. Throughout this manuscript, we have focused on item-to-item nearest-neighbour collaborative filtering as the main application of our work. When changing the terminology from "users" and "items" to "terms" and "documents", we believe that our approach is applicable to more general information retrieval use-cases as well. Nevertheless, in these settings, extensions for non-binary data (by including a term-value pair in the inverted indices instead of just the term) would be appropriate. Naturally, when these inverted indices keep growing in size, compression techniques might be convenient to improve on space efficiency. However, most state-of-the-art compression techniques do not support incremental updates, and random access would be imperative [13, 16].

ACKNOWLEDGMENTS

We are very grateful to the anonymous reviewers, and especially to Harald Steck for useful comments on earlier versions of this work.

REFERENCES

[1] J. Bennett, S. Lanning, et al. 2007. The Netflix prize. In *Proc. of the KDD cup and workshop*, Vol. 2007. 35.

[2] J. Dean and S. Ghemawat. 2008. MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM* 51, 1 (Jan. 2008), 107–113.

[3] A. Gruson, P. Chandar, C. Charbuillet, J. McInerney, S. Hansen, D. Tardieu, and B. Carterette. 2019. Offline Evaluation to Make Decisions About Playlist Recommendation Algorithms. In *Proc. of the 12th ACM International Conference on Web Search and Data Mining (WSDM '19)*. ACM, New York, NY, USA, 420–428.

[4] F. M. Harper and J. A. Konstan. 2015. The MovieLens Datasets: History and Context. *ACM Transactions on Interactive Intelligent Systems* 5, 4, Article 19 (2015), 19 pages.

[5] X. He, H. Zhang, M. Kan, and T. Chua. 2016. Fast Matrix Factorization for Online Recommendation with Implicit Feedback. In *Proc. of the 39th International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR '16)*. ACM, 549–558.

[6] Y. Huang, B. Cui, W. Zhang, J. Jiang, and Y. Xu. 2015. TencentRec: Real-time Stream Recommendation in Practice. In *Proc. of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD '15)*. ACM, 227–238.

[7] D. Jannach and M. Ludewig. 2017. When Recurrent Neural Networks Meet the Neighborhood for Session-Based Recommendation. In *Proc. of the 11th ACM Conference on Recommender Systems (RecSys '17)*. ACM, 306–310.

[8] O. Jeunen, K. Verstrepen, and B. Goethals. 2018. Fair Offline Evaluation Methodologies for Implicit-feedback Recommender Systems with MNAR Data. In *Proc. of the REVEAL 18 Workshop on Offline Evaluation for Recommender Systems (RecSys '18)*.

[9] M. Karimi, D. Jannach, and M. Jugovac. 2018. News recommender systems - Survey and roads ahead. *Information Processing & Management* 54, 6 (2018), 1203 – 1227.

[10] N. Liu, M. Zhao, E. Xiang, and Q. Yang. 2010. Online Evolutionary Collaborative Filtering. In *Proc. of the 4th ACM Conference on Recommender Systems (RecSys '10)*. ACM, 95–102.

[11] X. Luo, Y. Xia, Q. Zhu, and Y. Li. 2013. Boosting the K-Nearest-Neighborhood based incremental collaborative filtering. *Knowledge-Based Systems* 53 (2013), 90 – 99.

[12] M. Muja and D. G. Lowe. 2014. Scalable Nearest Neighbor Algorithms for High Dimensional Data. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 36, 11 (2014), 2227–2240.

[13] G. Ottaviano and R. Venturini. 2014. Partitioned Elias-Fano Indexes. In *Proc. of the 37th International ACM SIGIR Conference on Research & Development in Information Retrieval (SIGIR '14)*. ACM, 273–282.

[14] Outbrain. 2017. Kaggle Click Prediction Dataset. https://www.kaggle.com/c/outbrain-click-prediction/data. (2017).

[15] M. Papagelis, I. Rousidis, D. Plexousakis, and E. Theoharopoulos. 2005. Incremental Collaborative Filtering for Highly-Scalable Recommendation Algorithms. In *Foundations of Intelligent Systems*, M. Hacid, N. Murray, Z. Raś, and S. Tsumoto (Eds.). Springer, 553–561.

[16] G. E. Pibiri, M. Petri, and A. Moffat. 2019. Fast Dictionary-Based Compression for Inverted Indexes. In *Proc. of the 12th ACM International Conference on Web Search and Data Mining (WSDM '19)*. ACM, 6–14.

[17] S. Rendle and L. Schmidt-Thieme. 2008. Online-updating Regularized Kernel Matrix Factorization Models for Large-scale Recommender Systems. In *Proc. of the 1st ACM Conference on Recommender Systems (RecSys '08)*. ACM, 251–258.

[18] S. Sarawagi and A. Kirpal. 2004. Efficient Set Joins on Similarity Predicates. In *Proc. of the 2004 ACM SIGMOD International Conference on Management of Data (SIGMOD '04)*. ACM, New York, NY, USA, 743–754.

[19] B. Sarwar, G. Karypis, J. Konstan, and J. Riedl. 2001. Item-based Collaborative Filtering Recommendation Algorithms. In *Proc. of the 10th International Conference on World Wide Web (WWW '01)*. ACM, 285–295.

[20] R. S. Sreepada and B. K. Patra. 2018. An Incremental Approach for Collaborative Filtering in Streaming Scenarios. In *Advances in Information Retrieval*, G. Pasi, B. Piwowarski, L. Azzopardi, and A. Hanbury (Eds.). Springer International Publishing, 632–637.

[21] K. Subbian, C. Aggarwal, and K. Hegde. 2016. Recommendations For Streaming Data. In *Proc. of the 25th ACM International on Conference on Information and Knowledge Management (CIKM '16)*. ACM, 2185–2190.

[22] K. Verstrepen and B. Goethals. 2014. Unifying Nearest Neighbors Collaborative Filtering. In *Proc. of the 8th ACM Conference on Recommender Systems (RecSys '14)*. ACM, 177–184.

[23] Q. Wang, H. Yin, Z. Hu, D. Lian, H. Wang, and Z. Huang. 2018. Neural Memory Streaming Recommender Networks with Adversarial Training. In *Proc. of the 24th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '18)*. ACM, 2467–2475.

[24] W. Wang, H. Yin, Z. Huang, Q. Wang, X. Du, and Q. Nguyen. 2018. Streaming Ranking Based Recommender Systems. In *Proc. of the 41st International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR '18)*. ACM, 525–534.

[25] C. Yang, X. Yu, and Y. Liu. 2014. Continuous KNN Join Processing for Real-Time Recommendation. In *Proc. of the 14th IEEE International Conference on Data Mining (ICDM '14)*. 640–649.

[26] L. Yang, Y. Cui, Yuan X., C. Wang, S. Belongie, and D. Estrin. 2018. Unbiased Offline Recommender Evaluation for Missing-not-at-random Implicit Feedback. In *Proc. of the 12th ACM Conference on Recommender Systems (RecSys '18)*. ACM, New York, NY, USA, 279–287.

[27] X. Yang, Z. Zhang, and K. Wang. 2012. Scalable Collaborative Filtering Using Incremental Update and Local Link Prediction. In *Proc. of the 21st ACM International Conference on Information and Knowledge Management (CIKM '12)*. ACM, 2371–2374.

[28] C. Yu, B. Cui, S. Wang, and J. Su. 2007. Efficient Index-based KNN Join Processing for High-dimensional Data. *Inf. Softw. Technol.* 49, 4 (April 2007), 332–344.

[29] C. Yu, B. Ooi, K. Tan, and H. Jagadish. 2001. Indexing the distance: An efficient method to knn processing. In *Proc. of the 27th International Conference on Very Large Databases (VLDB '01)*. 421–430.

[30] C. Yu, R. Zhang, Y. Huang, and H. Xiong. 2009. High-dimensional kNN joins with incremental updates. *GeoInformatica* 14, 1 (2009), 55–82.

[31] R. Zadeh and A. Goel. 2013. Dimension independent similarity computation. *The Journal of Machine Learning Research* 14, 1 (2013), 1605–1626.

[32] C. Zhang, F. Li, and J. Jestes. 2012. Efficient Parallel kNN Joins for Large Data in MapReduce. In *Proc. of the 15th International Conference on Extending Database Technology (EDBT '12)*. ACM, 38–49.