Reverse Maximum Inner Product Search: Formulation, Algorithms, and Analysis

DAICHI AMAGATA and TAKAHIRO HARA, Osaka University, Japan

26

The maximum inner product search (MIPS), which finds the item with the highest inner product with a given query user, is an essential problem in the recommendation field. Usually e-commerce companies face situations where they want to promote and sell new or discounted items. In these situations, we have to consider the following questions: Who is interested in the items, and how do we find them? This article answers this question by addressing a new problem called reverse maximum inner product search (reverse MIPS). Given a query vector and two sets of vectors (user vectors and item vectors), the problem of reverse MIPS finds a set of user vectors whose inner product with the query vector is the maximum among the query and item vectors. Although the importance of this problem is clear, its straightforward implementation incurs a computationally expensive cost.

We therefore propose Simpfer, a simple, fast, and exact algorithm for reverse MIPS. In an offline phase, Simpfer builds a simple index that maintains a lower bound of the maximum inner product. By exploiting this index, Simpfer judges whether the query vector can have the maximum inner product or not, for a given user vector, in a constant time. Our index enables filtering user vectors, which cannot have the maximum inner product with the query vector, in a batch. We theoretically demonstrate that Simpfer outperforms baselines employing state-of-the-art MIPS techniques. In addition, we answer two new research questions. Can approximation algorithms further improve reverse MIPS processing? Is there an exact algorithm that is faster than Simpfer? For the former, we show that approximation with quality guarantee provides a little speed-up. For the latter, we propose Simpfer++, a theoretically and practically faster algorithm than Simpfer. Our extensive experiments on real datasets show that Simpfer is at least two orders of magnitude faster than the baselines, and Simpfer++ further improves the online processing time.

CCS Concepts: • **Information systems** → **Recommender systems**; **Proximity search;**

Additional Key Words and Phrases: Reverse maximum inner product search, high dimensional data, algorithm

ACM Reference format:
Daichi Amagata and Takahiro Hara. 2023. Reverse Maximum Inner Product Search: Formulation, Algorithms, and Analysis. *ACM Trans. Web* 17, 4, Article 26 (July 2023), 23 pages.
https://doi.org/10.1145/3587215

1 INTRODUCTION

The **maximum inner product search (MIPS)** problem, or k**-MIPS** problem, is an essential tool in the recommendation field. Given a query (user) vector, this problem finds the k item vectors

This research is partially supported by JST PRESTO Grant Number JPMJPR1931, JSPS Grant-in-Aid for Scientific Research (A) Grant Number 18H04095, and JST CREST Grant Number JPMJCR21F2.
Authors' address: D. Amagata and T. Hara, Graduate School of Information Science and Technology, Osaka University, 1-5 Yamadaoka, Suita, Osaka, 565-0871 Japan; emails: {amagata.daichi, hara}@ist.osaka-u.ac.jp.

1559-1131/2023/07-ART26
https://doi.org/10.1145/3587215

Table 1. Example of $\mathbf{U}$ and $\mathbf{P}$

$\mathbf{U}$		$\mathbf{P}$	
$\mathbf{u}_1$	$\langle 3.1, 0.1\rangle$	$\mathbf{p}_1$	$\langle 2.8, 0.6\rangle$
$\mathbf{u}_2$	$\langle 2.5, 2.0\rangle$	$\mathbf{p}_2$	$\langle 2.5, 1.8\rangle$
$\mathbf{u}_3$	$\langle 1.5, 2.2\rangle$	$\mathbf{p}_3$	$\langle 3.2, 1.0\rangle$
$\mathbf{u}_4$	$\langle 1.8, 3.2\rangle$	$\mathbf{p}_4$	$\langle 1.4, 2.6\rangle$
		$\mathbf{p}_5$	$\langle 0.5, 3.4\rangle$

Table 2. The Result of MIPS ($\mathbf{p}^*$) on $\mathbf{P}$ for Each $\mathbf{u} \in \mathbf{U}$

$\mathbf{p}^*$ (MIPS result)	
$\mathbf{u}_1$	$\mathbf{p}_3$
$\mathbf{u}_2$	$\mathbf{p}_2$
$\mathbf{u}_3$	$\mathbf{p}_5$
$\mathbf{u}_4$	$\mathbf{p}_5$

with the highest inner product with the query vector among a set of item vectors. The search result, i.e., k item vectors, can be used as recommendation for the user, and the user and item vectors are obtained via Matrix Factorization, which is well employed in recommender systems [4, 7, 12, 15, 19, 21, 30, 40]. Although some learned similarities via MLP (i.e., neural networks) have also been devised, e.g., in References [46, 49], a study [34] has actually demonstrated that inner product-based (i.e., Matrix Factorization-based) recommendations show better performances than learned similarities. We hence focus on the inner product between d-dimensional vectors that is obtained via Matrix Factorization.[1]

1.1 Motivation

The k-MIPS problem is effective for the case where a user wants to know items that s/he prefers (i.e., user-driven cases) [16, 17, 28], but e-commerce companies usually face situations where they want to advertise an item [18], which may be a new or discounted one, to users, which corresponds to item-driven cases. Trivially, an effective advertisement is to recommend such an item to users who would be interested in this item.

In the context of the k-MIPS problem, if this item is included in the top-k item set for a user, then we should make an advertisement of the item to this user. That is, *we should find a set of such users*. This article addresses this new problem, called the *reverse k-MIPS problem*. For ease of presentation, this section assumes that $k = 1$ (the general case is defined in Section 2). Given a query vector $\mathbf{q}$ (the vector of a target item) and two sets of d-dimensional vectors $\mathbf{U}$ (set of user vectors) and $\mathbf{P}$ (set of item vectors), the reverse MIPS problem finds all user vectors $\mathbf{u} \in \mathbf{U}$ such that $\mathbf{q} = \arg\max_{\mathbf{p} \in \mathbf{P} \cup \{\mathbf{q}\}} \mathbf{p} \cdot \mathbf{u}$.

Example 1. Table 1 illustrates $\mathbf{U}$ and $\mathbf{P}$, while Table 2 shows the MIPS result, i.e., $\mathbf{p}^* = \arg\max_{\mathbf{p} \in \mathbf{P}} \mathbf{u} \cdot \mathbf{p}$, of each vector in $\mathbf{U}$ in Table 1. Let $\mathbf{q} = \mathbf{p}_5$, and the result of reverse MIPS is $\{\mathbf{u}_3, \mathbf{u}_4\}$, because $\mathbf{p}_5$ is the top-1 item for $\mathbf{u}_3$ and $\mathbf{u}_4$. When $\mathbf{q} = \mathbf{p}_1$, we have no result, because $\mathbf{p}_1$ is not the top-1 item $\forall \mathbf{u} \in \mathbf{U}$. Similarly, when $\mathbf{q} = \mathbf{p}_2$, the result is $\{\mathbf{u}_2\}$.

From this example, we see that if an e-commerce service wants to promote the item corresponding to $\mathbf{p}_5$, then this service can obtain the users who would prefer this item through the reverse MIPS and send them a notification about this item.

The reverse k-MIPS problem is an effective tool not only for item-driven recommendations but also market analysis. Assume that we are given a vector of a new item, $\mathbf{q}$. It is necessary to design an effective sales strategy to gain a profit. Understanding the features of users that may prefer the item is important for the strategy. Solving the reverse k-MIPS of the query vector $\mathbf{q}$ supports this understanding.

[1] Actually, as long as users and items are represented by vectors and the relationships between users and items are evaluated by inner products of them, our problem and techniques are available.

1.2 Challenge

The above practical situations clarify the importance of reverse MIPS. Because e-commerce services have large number of users and items, $|\mathbf{U}|$ and $|\mathbf{P}|$ are large. In addition, a query vector is not pre-known and is specified on-demand fashion. The reverse k-MIPS is therefore conducted online and is a computationally intensive task. Now the question is how to efficiently obtain the reverse MIPS result for a given query.

A straightforward approach is to run a state-of-the-art exact MIPS algorithm *for every vector in* $\mathbf{U}$ and check whether $\mathbf{q} = \arg\max_{\mathbf{p} \in \mathbf{P} \cup \{\mathbf{q}\}} \mathbf{u} \cdot \mathbf{p}$. This approach obtains the exact result, but it incurs unnecessary computation. The poor performance of this approach is derived from the following observations. First, we do not need the MIPS result of $\mathbf{u}$ when $\mathbf{q}$ does not have the maximum inner product with $\mathbf{u}$. Second, this approach certainly accesses all user vectors in $\mathbf{U}$, although many of them do not contribute to the reverse MIPS result. However, it is not trivial to skip evaluations of some user vectors without losing correctness. Last, its theoretical cost is the same as the brute-force case, i.e., $O(nmd)$ time, where $n = |\mathbf{U}|$ and $m = |\mathbf{P}|$, which is not appropriate for online computations. These concerns pose challenges for solving the reverse MIPS problem efficiently.

1.3 Contribution

To address the above issues, we propose *Simpfer*, a *simp*le, *f*ast, and *e*xact algorithm for *r*everse MIPS. The general idea of Simpfer is to efficiently solve the decision version of the MIPS problem. Because the reverse MIPS of a query $\mathbf{q}$ requires a yes/no decision for each vector $\mathbf{u} \in \mathbf{U}$, it is sufficient to know whether $\mathbf{q}$ can have the maximum inner product for $\mathbf{u}$. Simpfer achieves this in $O(1)$ time in many cases by exploiting its index built in an offline phase. This index furthermore supports a constant time filtering that prunes vectors in a batch if their answers are no. We theoretically demonstrate that the time complexity of Simpfer is lower than $O(nmd)$. To summarize, we make the following contributions:

- We address the problem of reverse k-MIPS. To our knowledge, this is the first work to study this problem.
- We propose Simpfer as an exact solution to the reverse MIPS problem. Simpfer solves the decision version of the MIPS problem at both the group level and the vector level efficiently. Simpfer is surprisingly simple, but our analysis demonstrates that Simpfer theoretically outperforms a solution that employs a state-of-the-art exact MIPS algorithm.

These contents appear in our conference version [2].

1.3.1 Comparison to Our Preliminary Version. In addition to the above contributions, we address new research questions that are worth investigating for designing faster algorithms than Simpfer.

(1) Is approximation useful for efficiency acceleration?
(2) Is there a faster exact algorithm than Simpfer?

First, we address the important question: Can approximation algorithms further improve reverse MIPS processing? To answer this question, we first incorporate the concept **c-Approximate MIPS (c-AMIPS)** [20, 29, 35, 43] into the reverse MIPS problem. Consider a user vector $\mathbf{u}$ and its exact MIPS result $\mathbf{p}^*$. If $\mathbf{u} \cdot \mathbf{q} \geq c \times \mathbf{u} \cdot \mathbf{p}^*$ for a given query vector $\mathbf{q}$ and $c \in (0, 1)$, then $\mathbf{q}$ can be an answer of c-AMIPS. Then, we show that the approach of Simpfer can be extended so that we always guarantee to have user vectors $\mathbf{u}$ such that $\mathbf{u} \cdot \mathbf{q} \geq c \times \mathbf{u} \cdot \mathbf{p}^*$ for arbitrary queries and c. We theoretically show that this approximation algorithm provides a speed-up, but it is a slight improvement against Simpfer.

Then, we consider the second new question: Is there an exact algorithm that is faster than Simpfer? The answer is yes if we take a more pre-processing time than that of Simpfer. In a nutshell, if we run a maximum inner product join between $\mathbf{U}$ and $\mathbf{P}$ offline, then *we can avoid accessing* $\mathbf{P}$ online, suggesting the existence of an $O(nd)$ time algorithm. By taking this observation, we propose Simpfer++, an extension of Simpfer.

Below, we summarize our additional contributions of this article (Sections 5, 6, and 7 are new contents compared with Reference [2]).

- We answer the question: Can we improve reverse MIPS processing time by approximation? We incorporate the concept c-approximate MIPS into the reverse MIPS problem and show that the framework of Simper is surprisingly easy to have a quality guarantee. However, this approximation, denoted by c-Simpfar (*simp*le, *f*ast, and *a*pproximate *r*everse), yields a little speed-up theoretically and practically.
- We next answer the question: Is there an exact algorithm that is faster than Simpfer? We propose Simpfer++, which needs at most $O(nd)$ time while guaranteeing the correctness by taking more pre-processing time than Simpfer. The advantage of Simpfer++ is that it does need to access $\mathbf{P}$ in the online processing.
- We conduct extensive experiments on four real datasets, MovieLens, Netflix, Amazon, and Yahoo!. The results show that Simpfer is at least two orders of magnitude faster than baselines. In addition, Simpfer++ further improves the reverse MIPS and is at most 70 times faster than Simpfer.
- Simpfer and Simpfer++ are easy to deploy: *If recommender systems have user and item vector sets that are designed in the inner product space, then they are ready to use Simpfer(++) via our open source implementation.*[2] This is because Simpfer and Simpfer++ are unsupervised and have only a single parameter (the maximum value of k) that is easy to tune and has no effect on the running time of online processing.

1.4 Organization

The rest of this article is organized as follows. We formally define our problem in Section 2. We review related work in Section 3. Simpfer, its approximation, and Simpfer++ are presented in Sections 4, 5, and 6, respectively. Our experimental results are reported in Section 7. Last, we conclude this article in Section 8.

2 PROBLEM DEFINITION

Let $\mathbf{P}$ be a set of d-dimensional dense real-valued item vectors, and we assume that d is high [26, 35]. Given a query vector, the MIPS problem finds

$$\mathbf{p}^* = \arg\max_{\mathbf{p} \in \mathbf{P}} \mathbf{p} \cdot \mathbf{q}.$$

The general version of the MIPS problem, i.e., the k-MIPS problem, is defined as follows:

Definition 1 (k-MIPS problem). Given a set of vectors $\mathbf{P}$, a query vector $\mathbf{q}$, and k, the k-MIPS problem, denoted by $f_{MIPS}(\mathbf{P}, \mathbf{q}, k)$, is defined as

$$f_{MIPS}(\mathbf{P}, \mathbf{q}, k) = \mathbf{S} \subseteq \mathbf{P},$$

such that $|\mathbf{S}| = k$ and, for each $\mathbf{p} \in \mathbf{S}$ and $\mathbf{p}' \in \mathbf{P} \backslash \mathbf{S}$, we have $\mathbf{p} \cdot \mathbf{q} \geq \mathbf{p}' \cdot \mathbf{q}$. That is, the k-MIPS problem returns the set $\mathbf{S}$ of k vectors in $\mathbf{P}$ that have the highest inner products with $\mathbf{q}$.

[2] https://github.com/amgt-d1/Simpfer.

For a user (i.e., query), the k-MIPS problem can retrieve k *items* (e.g., vectors in $\mathbf{P}$) that the user would prefer. Different from this, the reverse k-MIPS problem can retrieve *a set of users* who would prefer a given item. That is, in the reverse k-MIPS problem, a query can be an item, and *this problem finds users attracted by the query item.* Therefore, the reverse k-MIPS is effective for advertisement and market analysis, as described in Section 1. We formally define this problem.[3]

Definition 2 (Reverse k-MIPS problem). Given a query (item) vector $\mathbf{q}$, k, and two sets of vectors $\mathbf{U}$ (set of user vectors) and $\mathbf{P}$ (set of item vectors), the reverse k-MIPS problem, denoted by $f_{RMIPS}(\mathbf{U}, \mathbf{P}, \mathbf{q}, k)$, is defined as

$$f_{RMIPS}(\mathbf{U}, \mathbf{P}, \mathbf{q}, k) = \{\mathbf{u} \mid \mathbf{u} \in \mathbf{U}, \mathbf{q} \in f_{MIPS}(\mathbf{P} \cup \{\mathbf{q}\}, \mathbf{u}, k)\}.$$

That is, the reverse k-MIPS problem finds all vectors $\mathbf{u} \in \mathbf{U}$ such that $\mathbf{q}$ is included in the k-MIPS result of $\mathbf{u}$ among $\mathbf{P} \cup \{\mathbf{q}\}$.

Note that $\mathbf{q}$ can be $\mathbf{q} \in \mathbf{P}$, as described in Example 1. We use n and m to denote $|\mathbf{U}|$ and $|\mathbf{P}|$, respectively.

Our only assumption is that there is a maximum k that can be specified, denoted by k_{max}. This is practical, because k should be small, e.g., $k = 5$ [21] or $k = 10$ [3], to make applications effective. (We explain how to deal with the case of $k > k_{max}$ in Section 4.1.) The objective of this article is to develop an algorithm that can quickly solve the reverse k-MIPS problem.

3 RELATED WORK

3.1 Exact k-MIPS Algorithm

The reverse k-MIPS problem can be solved exactly by conducting an exact k-MIPS algorithm for each user vector in $\mathbf{U}$. The first line of solution to the k-MIPS problem is a tree-index approach [8, 22, 33]. For example, Reference [33] proposed a tree-based algorithm that processes k-MIPS not only for a single user vector but also for some user vectors in a batch. Unfortunately, the performances of the tree-index algorithms degrade for large d because of the curse of dimensionality.

LEMP [38, 39] avoids this issue and significantly outperforms the tree-based algorithms. LEMP uses several search algorithms according to the norm of each vector. In addition, LEMP devises an early stop scheme of inner product computation. During the computation of $\mathbf{u} \cdot \mathbf{q}$, LEMP computes an upper bound of $\mathbf{u} \cdot \mathbf{q}$. If this bound is lower than an intermediate kth maximum inner product, then $\mathbf{q}$ cannot be in the final result, and thus the inner product computation can be stopped. LEMP is actually designed for the top-k inner product join (i.e., all pairs k-MIPS) problem: For each $\mathbf{u} \in \mathbf{U}$, it finds the k-MIPS result of $\mathbf{u}$. Therefore, LEMP can solve the reverse k-MIPS problem, but it is not efficient as demonstrated in Section 7.

FEXIPRO [23] further improves the early stop of inner product computation of LEMP. Specifically, FEXIPRO exploits singular value decomposition, integer approximation, and a transformation to positive values. These techniques aim at obtaining a tighter upper bound of $\mathbf{u} \cdot \mathbf{q}$ as early as possible. Reference [23] reports that state-of-the-art tree-index algorithm [33] is completely outperformed by FEXIPRO. Maximus [1] takes hardware optimization into account. However, it is limited to specific CPUs, so we do not consider Maximus. Note that LEMP and FEXIPRO are heuristic algorithms, and $O(nmd)$ time is required for the reverse k-MIPS problem.

[3]Actually, the reverse top-k query (and its variant), a similar concept to the reverse k-MIPS problem, has been proposed in References [41, 42, 48]. It is important to note that these works do not suit recent recommender systems. First, they assume that d is low (d is around 5), which is not probable in Matrix Factorization. Second, they consider the Euclidean space, whereas inner product is a non-metric space. Because the reverse top-k query processing algorithms are optimized for these assumptions, they cannot be employed in Matrix Factorization-based recommender systems and cannot solve (or be extended for) the reverse k-MIPS problem.

3.2 Approximation k-MIPS Algorithm

To solve the k-MIPS problem in sub-linear time by sacrificing correctness, many works proposed approximation k-MIPS algorithms. There are several approaches to the approximation k-MIPS problem: sampling-based [6, 26, 45], LSH-based [20, 29, 32, 35, 36, 43], graph-based [25, 27, 37, 50], and quantization approaches [9, 13, 47]. They have both strong and weak points. For example, LSH-based algorithms enjoy a probabilistic accuracy guarantee. However, they are empirically slower than graph-based algorithms that have no theoretical performance guarantee. The literature [3] shows that the MIPS problem can be transformed into the Euclidean nearest-neighbor search problem, but it still cannot provide the correct answer. Besides, existing works that address the (reverse) nearest-neighbor search problem assume low-dimensional data [44] or consider approximation algorithms [24].

When applications allow approximate results, these approximation k-MIPS algorithms can be utilized. However, approximate answers may lose effectiveness of the reverse k-MIPS problem. If applications cannot contain users, who are the answer of the k-MIPS problem, then these users may lose chances of knowing the target item, which would reduce profits. In addition, users, who have much less inner products with a given query vector than that with the exact k-MIPS answer, would not be interested in the item. Including them as a reverse k-MIPS result may lose future profits, because such users may stop receiving advertisements from the recommender systems if they get those of non-interesting items. Therefore, a reasonable approximation is to allow a solution to additionally include only users vectors, which have similar inner products with a given query vector to that with the exact k-MIPS answers. In Section 5, we formulate such an approximate reverse k-MIPS problem and show that the approach of Simpfer is easy to give a quality guarantee. Note that, for approximation, we do not consider existing algorithms that cannot provide any theoretical error guarantee for approximate k-MIPS results [6, 9, 13, 25–27, 31, 37, 45, 47, 50]. This is because they cannot provide any error guarantee for our problem (see Definition 7). Remark 2 further clarifies this point.

4 SIMPFER

To efficiently solve the reverse MIPS problem, we propose Simpfer. Its general idea is to efficiently solve the decision version of the k-MIPS problem for each $\mathbf{u} \in \mathbf{U}$.

Definition 3 (k-MIPS DECISION PROBLEM). Given a query $\mathbf{q}$, k, a user vector $\mathbf{u}$, and $\mathbf{P}$, this problem, denoted by $f_{MIPS\text{-}Dec}(\mathbf{P} \cup \{\mathbf{q}\}, \mathbf{u}, k)$, is defined as

$$f_{MIPS\text{-}Dec}(\mathbf{P} \cup \{\mathbf{q}\}, \mathbf{u}, k) = \begin{cases} 1 & (\mathbf{q} \in f_{MIPS}(\mathbf{P} \cup \{\mathbf{q}\}, \mathbf{u}, k)) \\ 0 & (\text{otherwise}) \end{cases}.$$

That is, this problem returns yes, i.e., 1, (no, i.e., 0) if $\mathbf{q}$ is (not) included in the k-MIPS result of $\mathbf{u}$.

From Definitions 2 and 3, we see that if $f_{MIPS\text{-}Dec}(\mathbf{P} \cup \{\mathbf{q}\}, \mathbf{u}, k)$ returns 1, then $\mathbf{u} \in f_{RMIPS}(\mathbf{U}, \mathbf{P}, \mathbf{q}, k)$, i.e., $\mathbf{u}$ is included in the reverse k-MIPS result. Therefore, by solving the k-MIPS decision problem for each $\mathbf{u} \in \mathbf{U}$, we can solve the reverse k-MIPS result. The main merit of the k-MIPS "decision" problem is that *this problem does not require the complete k-MIPS result.* We can terminate the k-MIPS of $\mathbf{u}$ whenever it is guaranteed that $\mathbf{q}$ is (not) included in the k-MIPS result.

To achieve this early termination efficiently, it is necessary to obtain a lower bound and an upper bound of the kth highest inner product of $\mathbf{u}$. Let ϕ and η respectively be a lower bound and an upper bound of the kth highest inner product of $\mathbf{u}$ on $\mathbf{P}$. If $\phi > \mathbf{u} \cdot \mathbf{q}$, then it is guaranteed that $\mathbf{q}$ does not have the k highest inner product with $\mathbf{u}$. Similarly, if $\eta \leq \mathbf{u} \cdot \mathbf{q}$, then it is guaranteed that $\mathbf{q}$ has the k highest inner product with $\mathbf{u}$. This observation implies that we need to efficiently obtain

ϕ and η. Simpfer does pre-processing to enable it in an offline phase. Besides, since $n = |\mathbf{U}|$ is often large, accessing all user vectors online is time-consuming. This requires a filtering technique that enables the pruning of user vectors that are not included in the reverse k-MIPS result *in a batch*. During the pre-processing, Simpfer arranges $\mathbf{U}$ so that batch filtering is enabled. Simpfer exploits the data structures built in the pre-processing phase to quickly solve the k-MIPS decision problem.

4.1 Pre-processing

The objective of this pre-processing phase is to build data structures that support efficient computation of a lower bound and an upper bound of the kth highest inner product for each $\mathbf{u}_i \in \mathbf{U}$, for arbitrary queries. We utilize Cauchy–Schwarz inequality for upper bounding. Hence we need the Euclidean norm $\|\mathbf{u}_i\|$ for each $\mathbf{u}_i \in \mathbf{U}$. To obtain a lower bound of the kth highest inner product, we need to access at least k item vectors in $\mathbf{P}$. The norm computation and lower-bound computation are independent of queries (as long as $k \leq k_{max}$), so they can be pre-computed. In this phase, Simpfer builds the following array for each $\mathbf{u}_i \in \mathbf{U}$.

Definition 4 (Lower-bound array). The lower-bound array L_i of a user vector $\mathbf{u}_i \in \mathbf{U}$ is an array whose jth element, L_i^j, maintains a lower bound of the jth inner product of $\mathbf{u}_i$ on $\mathbf{P}$, and $|L_i| = k_{max}$.

Furthermore, to enable batch filtering, Simpfer builds a *block*, which is defined below.

Definition 5 (Block). A block $\mathbf{B}$ is a subset of $\mathbf{U}$. The set of vectors belonging to $\mathbf{B}$ is represented by $\mathbf{U}(\mathbf{B})$. Besides, we use $L(\mathbf{B})$ to represent the lower-bound array of this block, and

$$L^j(\mathbf{B}) = \min_{\mathbf{u}_i \in \mathbf{U}(\mathbf{B})} L_i^j. \tag{1}$$

The block size $|\mathbf{U}(\mathbf{B})|$ can be arbitrarily determined, and we set $|\mathbf{U}(\mathbf{B})| = O(\log n)$ to avoid system parameter setting.

4.1.1 *Algorithm Description.*

Algorithm 1 describes the pre-processing algorithm of Simpfer.

(1) Norm computation: First, for each $\mathbf{u} \in \mathbf{U}$ and $\mathbf{p} \in \mathbf{P}$, its norm is computed. Then, $\mathbf{U}$ and $\mathbf{P}$ are sorted in descending order of norm.
(2) Lower-bound array building: Let $\mathbf{P}'$ be the set of the $O(k_{max})$ vectors with the highest norm in $\mathbf{P}$. For each $\mathbf{u}_i \in \mathbf{U}$, L_i is built by using $\mathbf{P}'$. That is, $L_i^j = \mathbf{u}_i \cdot \mathbf{p}$, where $\mathbf{p} \in \mathbf{P}'$ yields the jth highest inner product for $\mathbf{u}$. The behind idea of using the first $O(k_{max})$ vectors with the highest norm in $\mathbf{P}$ is that vectors with large norms tend to provide large inner products [25]. This means that we can obtain a tight lower bound at a lightweight cost.
(3) Block building: After that, blocks are built, so that user vectors in a block keep the order and each block is disjoint. Given a new block $\mathbf{B}$, we insert user vectors $\mathbf{u}_i \in \mathbf{U}$ into $\mathbf{U}(\mathbf{B})$ in sequence while updating $L^j(\mathbf{B})$, until we have $|\mathbf{U}(\mathbf{B})| = O(\log n)$. When $|\mathbf{U}(\mathbf{B})| = O(\log n)$, we insert $\mathbf{B}$ into a set of blocks $\mathcal{B}$, and make a new block.[4]

Example 2. Figure 1 illustrates an example of block building. For ease of presentation, we use b as a block size and $n = 3b$. For example, $\mathbf{U}(\mathbf{B}_1) = \{\mathbf{u}_1, \ldots, \mathbf{u}_b\}$, and $\|\mathbf{u}_1\| \geq \cdots \geq \|\mathbf{u}_b\|$.

Generally, this pre-processing is done only once. An exception is the case where a query with $k > k_{max}$ is specified. In this case, Simpfer re-builds the data structures then processes the query. This is actually much faster than the baselines, as shown in Section 7.8.

[4]We employ norm-based blocking (grouping) to enable filtering based on Cauchy–Schwarz inequality. Because inner product is not metric (i.e., it does not satisfy triangle inequality), filtering based on triangle inequality (e.g., by using a clustering method) is impossible.

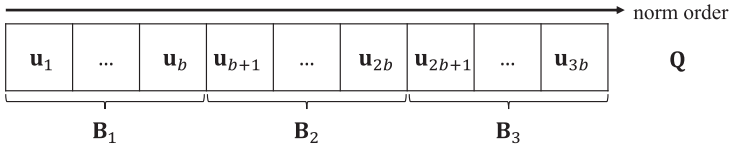


Fig. 1. Example of block building.

ALGORITHM 1: Pre-Processing of Simpfer

Input: $\mathbf{U}$, $\mathbf{P}$, and k_{max}
for *each* $\mathbf{u}_i \in \mathbf{U}$ **do**
  Compute $\|\mathbf{u}_i\|$
for *each* $\mathbf{p}_j \in \mathbf{P}$ **do**
  Compute $\|\mathbf{p}_j\|$
Sort $\mathbf{U}$ and $\mathbf{P}$ in descending order of norm size
$\mathbf{P}' \leftarrow$ the first $O(k_{max})$ vectors in $\mathbf{P}$
for *each* $\mathbf{u}_i \in \mathbf{U}$ **do**
  $\mathbf{R} \leftarrow k_{max}$ vectors $\mathbf{p} \in \mathbf{P}'$ that maximize $\mathbf{u}_i \cdot \mathbf{p}$
  for $j = 1$ *to* k_{max} **do**
    $L_i^j \leftarrow \mathbf{u}_i \cdot \mathbf{p}$, where $\mathbf{p}$ provides the jth highest inner product with $\mathbf{u}_i$ in $\mathbf{R}$
$\mathcal{B} \leftarrow \emptyset$,
$\mathbf{B} \leftarrow$ a new block
for *each* $\mathbf{u}_i \in \mathbf{U}$ **do**
  $\mathbf{U}(\mathbf{B}) \leftarrow \mathbf{U}(\mathbf{B}) \cup \{\mathbf{u}_i\}$
  for $j = 1$ *to* k_{max} **do**
    $L^j(\mathbf{B}) \leftarrow \min\{L^j(\mathbf{B}), L_i^j\}$
  if $|\mathbf{U}(\mathbf{B})| = O(\log n)$ **then**
    $\mathcal{B} \leftarrow \mathcal{B} \cup \{\mathbf{B}\}$
    $\mathbf{B} \leftarrow$ a new block

4.1.2 Analysis. We here prove that the time complexity of this pre-processing is reasonable. Without loss of generality, we assume $n \geq m$, because this is a usual case for many real datasets, as the ones we use in Section 7.

Theorem 1. *Algorithm 1 requires* $O(n(d + \log n))$ *time.*

Proof. The norm computation requires $O((n + m)d) = O(nd)$ time, and sorting requires $O(n \log n)$ time. The building of lower-bound arrays needs $O(n \times k_{max})$ time, since $O(|\mathbf{P}'|) = O(k_{max})$. Because $k_{max} = O(1)$, $O(n \times k_{max}) = O(n)$. The block building also requires $O(n \times k_{max}) = O(n)$ time. In total, this pre-processing requires $O(n(d + \log n))$ time. □

The space complexity of Simpfer is also reasonable.

Theorem 2. *The space complexity of the index is* $O(n)$.

Proof. The space of the lower-bound arrays of user vectors is $O(\sum_n |L_i|) = O(n)$, since $O(|L_i|) = O(1)$. Blocks are disjoint, and the space of the lower-bound array of a block is also $O(1)$. We hence have $O(\frac{n}{\log n})$ lower-bound arrays of blocks. Now this theorem is clear. □

4.2 Upper and Lower Bounding for the k-MIPS Decision Problem

Before we present the details of Simpfer, we introduce our techniques that can quickly answer the k-MIPS decision problem for a given query $\mathbf{q}$. Recall that $\mathbf{U}$ and $\mathbf{P}$ are sorted in descending order of norm. Without loss of generality, we assume that $\|\mathbf{u}_i\| \geq \|\mathbf{u}_{i+1}\|$ for each $i \in [1, n-1]$ and $\|\mathbf{p}_j\| \geq \|\mathbf{p}_{j+1}\|$ for each $j \in [1, m-1]$, for ease of presentation. That is, for example, we assume that the vectors in $\mathbf{P}$ are ordered as $\mathbf{p}_1, \mathbf{p}_2, \ldots, \mathbf{p}_n$.

Given a query $\mathbf{q}$ and a user vector $\mathbf{u}_i \in \mathbf{U}$, we have $\mathbf{u}_i \cdot \mathbf{q}$. Although our data structures are simple, they provide effective and "light-weight" filters. Specifically, we can quickly answer the k-MIPS decision problem on $\mathbf{q}$ through the following observations.[5]

Lemma 1. *If $\mathbf{u}_i \cdot \mathbf{q} < L_i^k$, then it is guaranteed that $\mathbf{q}$ is not included in the k-MIPS result of $\mathbf{u}_i$.*

Proof. Let $\mathbf{p}$ be the vector in $\mathbf{P}$ such that $\mathbf{u}_i \cdot \mathbf{p}$ is the kth highest inner product in $\mathbf{P}$. The fact that $L_i^k \leq \mathbf{u}_i \cdot \mathbf{p}$ immediately derives this lemma. □

It is important to see that the above lemma provides "no" as the answer to the k-MIPS decision problem on $\mathbf{q}$ in $O(1)$ time (after computing $\mathbf{u}_i \cdot \mathbf{q}$). The next lemma deals with the "yes" case in $O(1)$ time.

Lemma 2. *If $\mathbf{u}_i \cdot \mathbf{q} \geq \|\mathbf{u}_i\|\|\mathbf{p}_k\|$, then it is guaranteed that $\mathbf{q}$ is included in the k-MIPS result of $\mathbf{u}_i$.*

Proof. From Cauchy–Schwarz inequality, we have $\mathbf{u}_i \cdot \mathbf{p}_j \leq \|\mathbf{u}_i\|\|\mathbf{p}_j\|$. Since $\|\mathbf{p}_k\|$ is the kth highest norm in $\mathbf{P}$, $\mathbf{u}_i \cdot \mathbf{p} \leq \|\mathbf{u}_i\|\|\mathbf{p}_k\|$, where $\mathbf{p}$ is defined in the proof of Lemma 1. That is, $\|\mathbf{u}_i\|\|\mathbf{p}_k\|$ is an upper bound of $\mathbf{u}_i \cdot \mathbf{p}$. Now it is clear that $\mathbf{q}$ has $\mathbf{u}_i \cdot \mathbf{q} \geq \mathbf{u}_i \cdot \mathbf{p}$ if $\mathbf{u}_i \cdot \mathbf{q} \geq \|\mathbf{u}_i\|\|\mathbf{p}_k\|$. □

We next introduce a technique that yields "no" as the answer for *all user vectors* in a block $\mathbf{B}$ in $O(1)$ time.

Lemma 3. *Given a block $\mathbf{B}$, let $\mathbf{u}_i$ be the first vector in $\mathbf{U}(\mathbf{B})$. If $\|\mathbf{u}_i\|\|\mathbf{q}\| < L^k(\mathbf{B})$, then, for all $\mathbf{u}_j \in \mathbf{U}(\mathbf{B})$, it is guaranteed that $\mathbf{q}$ is not included in the k-MIPS result of $\mathbf{u}_j$.*

Proof. From Cauchy–Schwarz inequality, $\|\mathbf{u}_i\|\|\mathbf{q}\|$ is an upper bound of $\mathbf{u}_j \cdot \mathbf{q}$ for all $\mathbf{u}_j \in \mathbf{U}(\mathbf{B})$, since $\mathbf{U}(\mathbf{B}) = \{\mathbf{u}_i, \mathbf{u}_{i+1}, \ldots\}$. We have $L^k(\mathbf{B}) \leq L_j^k$ for all $\mathbf{u}_j \in \mathbf{U}(\mathbf{B})$, from Equation (1). Therefore, if $\|\mathbf{u}_i\|\|\mathbf{q}\| < L^k(\mathbf{B})$, then $\mathbf{u}_j \cdot \mathbf{q}$ cannot be the k highest inner product. □

If a user vector $\mathbf{u}_i$ cannot obtain a yes/no answer from Lemmas 1–3, then Simpfer uses a linear scan of $\mathbf{P}$ to obtain the answer. Let τ be a threshold, i.e., an intermediate kth highest inner product for $\mathbf{u}$ during the linear scan. By using the following corollaries, Simpfer can obtain the correct answer and early terminate the linear scan.

Corollary 1. *Assume that $\mathbf{q}$ is included in an intermediate result of the k-MIPS of $\mathbf{u}_i$ and we now evaluate $\mathbf{p}_j \in \mathbf{P}$. If $\mathbf{u}_i \cdot \mathbf{q} \geq \|\mathbf{u}_i\|\|\mathbf{p}_j\|$, then it is guaranteed that $\mathbf{q}$ is included in the final result of the k-MIPS of $\mathbf{u}_i$.*

Proof. Trivially, we have $j \geq k$. Besides, $\|\mathbf{u}_i\|\|\mathbf{p}_j\| \geq \mathbf{u}_i \cdot \mathbf{p}_l$ for all $k \leq l \leq m$, because $\mathbf{P}$ is sorted. This corollary is hence true. □

From this corollary, it it trivial to see that

Corollary 2. *When we have $\tau > \mathbf{u}_i \cdot \mathbf{q}$, it is guaranteed that $\mathbf{q}$ is not included in the final result of the k-MIPS of $\mathbf{u}_i$.*

Algorithm 2 summarizes the linear scan that incorporates Corollaries 1 and 2.

[5] Existing algorithms for top-k retrieval, e.g., References [10, 11], use similar (but different) bounding techniques. They use a bound (e.g., obtained by a block) to *early stop* linear scans. However, our bounding is designed to *avoid* linear scans and to filer multiple user vectors in a batch.

ALGORITHM 2: LINEAR-SCAN(u)

```
Input: u ∈ U, P, q, and k
1  I ← {u · q}
2  τ ← 0
3  for each p_i ∈ P do
4      if u · q ≥ ||u|| ||p_i|| then
5          return 1 (yes)
6      γ ← u · p_i
7      if γ > τ then
8          I ← I ∪ {γ}
9          if |I| > k then
10             Delete the (k + 1)-th inner product from I
11             τ ← the kth inner product in I
12         if τ > u · q then
13             return 0 (no)
```

ALGORITHM 3: SIMPFER

```
Input: U, P, q, k, and B
1  U_r ← ∅                                   ▷ U_r is a set of the reverse k-MIPS result
2  Compute ||q||
3  for each B ∈ B do
4      u ← the first user vector in U(B)
5      if ||u|| ||q|| ≥ L^k(B) then
6          for each u_i ∈ U(B) do
7              γ ← u_i · q
8              if γ ≥ L_i^k then
9                  if ||u_i|| ||p_k|| > γ then
10                     f ← LINEAR-SCAN(u_i)
11                     if f = 1 then
12                         U_r ← U_r ∪ {u_i}
13                 else
14                     U_r ← U_r ∪ {u_i}
15 return U_r
```

4.3 The Algorithm

Now we are ready to present Simpfer. Algorithm 3 details it. To start with, Simpfer computes $\|\mathbf{q}\|$. Given a block $\mathbf{B} \in \mathcal{B}$, Simpfer tests Lemma 3 (line 5). If the user vectors in $\mathbf{U}(\mathbf{B})$ may have yes as an answer, then for each $\mathbf{u}_i \in \mathbf{U}(\mathbf{B})$ Simpfer does the following. (Otherwise, all user vectors in $\mathbf{U}(\mathbf{B})$ are ignored.) First, it computes $\mathbf{u}_i \cdot \mathbf{q}$ and then tests Lemma 1 (line 8). If $\mathbf{u}_i$ cannot have the answer from this lemma, then Simpfer tests Lemma 2. Simpfer inserts $\mathbf{u}_i$ into the result set $\mathbf{U}_r$ if $\mathbf{u}_i \cdot \mathbf{q} \geq \|\mathbf{u}_i\| \|\mathbf{p}_k\|$. Otherwise, Simpfer conducts LINEAR-SCAN($\mathbf{u}_i$) (Algorithm 2). If LINEAR-SCAN($\mathbf{u}_i$) returns 1 (yes), then $\mathbf{u}_i$ is inserted into $\mathbf{U}_r$. The above operations are repeated for each $\mathbf{B} \in \mathcal{B}$. Finally, Simpfer returns the result set $\mathbf{U}_r$.

The correctness of Simpfer is obvious, because it conducts Linear-scan$(\cdot)$ for all vectors that cannot have yes/no answers from Lemmas 1–3. Besides, Simpfer accesses blocks sequentially, so it is easy to parallelize by using multicore (although this article focuses on a single-threaded case). Interested readers may refer to Reference [2], and note that distributed computing environments [14] are out of the scope of this work.

4.4 Complexity Analysis

We theoretically demonstrate the efficiency of Simpfer. Specifically, we have the following.

Theorem 3. *Let α be the pruning ratio ($0 \leq \alpha \leq 1$) of blocks in $\mathcal{B}$. Furthermore, let m' be the average number of item vectors accessed in* Linear-scan$(\cdot)$. *The time complexity of Simpfer is $O((1-\alpha)nm'd)$.*

Proof. Simpfer accesses all blocks in $\mathcal{B}$, and $|\mathcal{B}| = O(\frac{n}{\log n})$. Assume that a block $\mathbf{B} \in \mathcal{B}$ is not pruned by Lemma 3. Simpfer accesses all user vectors in $\mathbf{U}(\mathbf{B})$, so the total number of such user vectors is $(1-\alpha) \times O(\frac{n}{\log n}) \times O(\log n) = O((1-\alpha)n)$. For these vectors, Simpfer computes inner products with $\mathbf{q}$, and an inner product computation needs $O(d)$ time. The evaluation cost of Lemmas 1 and 2 for these user vectors is thus $O((1-\alpha)nd)$. The worst cost of Linear-scan$(\cdot)$ for vectors that cannot obtain the answer from these lemmas is $O((1-\alpha)nm'd)$. Now the time complexity of Simpfer is

$$O\left(\frac{n}{\log n} + (1-\alpha)nd + (1-\alpha)nm'd\right) = O\left(\frac{n}{\log n} + (1-\alpha)nm'd\right) = O((1-\alpha)nm'd). \tag{2}$$

Consequently, this theorem holds. □

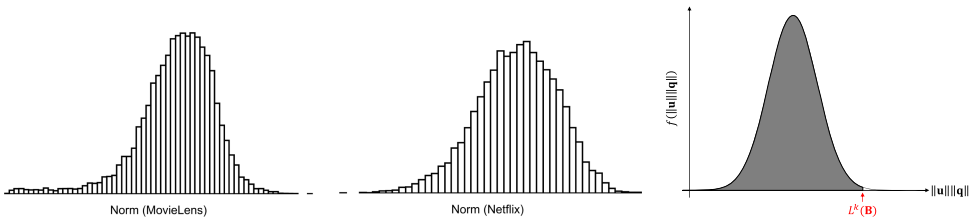
Remark 1. There are two main observations in Theorem 3. First, because we practically have $m' < m$ and $\alpha > 0$, Simpfer outperforms a k-MIPS-based solution that incurs $O(nmd)$ time. (Our experimental results show that $m' = O(k)$ in practice.) The second observation is obtained from Equation (2), which implies the effectiveness of blocks. If Simpfer does not build blocks, then we have to evaluate Lemma 1 *for all* $\mathbf{u} \in \mathbf{U}$. Equation (2) suggests that the blocks theoretically avoids this.

It is important to notice that α in Theorem 3 is directly related to the probability that $\|\mathbf{u}\|\|\mathbf{q}\| < L^k(\mathbf{B})$ for a given $\mathbf{q}$ and a block $\mathbf{B}$. We therefore analyze this probability.[6] To enable this analysis, we put two assumptions: (i) The norm of each item vector in $\mathbf{P}$ follows a normal distribution. This assumption is reasonable, because real datasets tend to have this case, see Figure 2(a) and (b). (ii) A query vector is a randomly sampled one from $\mathbf{P}$. Under these assumptions, the probability is computable (and high).

Recall that, given a block $\mathbf{B}$, we use its first vector $\mathbf{u}$ and lower-bound array $L^k(\mathbf{B})$ to apply Lemma 3. Notice that, for $\mathbf{B}$, $\|\mathbf{u}\|$ is fixed, so $\|\mathbf{u}\|\|\mathbf{q}\|$ also follows a normal distribution $\mathcal{N}(\mu, \sigma^2)$, where μ and σ represent mean and variance, respectively. Since the norm of each vector is computed offline, μ and σ are also obtained easily. The probability density function, denoted by $f(\|\mathbf{u}\|\|\mathbf{q}\|)$, is as follows:

$$f(\|\mathbf{u}\|\|\mathbf{q}\|) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(\|\mathbf{u}\|\|\mathbf{q}\| - \mu)^2}{2\sigma^2}\right).$$

[6] If this probability is high, then α becomes high. For each block $\mathbf{B} \in \mathcal{B}$, its first vector $\mathbf{u}$ and lower-bound array $L^k(\mathbf{B})$ are independent of those of the other blocks. We thus can only provide a probability that $\|\mathbf{u}\|\|\mathbf{q}\| < L^k(\mathbf{B})$, i.e., it is impossible to calculate how many blocks are pruned for a random query vector.



(a) Norm distribution in MovieLens (b) Norm distribution in Netflix (c) Distribution of $\|\mathbf{u}\|\|\mathbf{q}\|$ for a given block $\mathbf{B}$. The gray area represents the probability that $\|\mathbf{u}\|\|\mathbf{q}\| < L^k(\mathbf{B})$, i.e., $\int_{-\infty}^{L^k(\mathbf{B})} f(x)dx$, where $x = \|\mathbf{u}\|\|\mathbf{q}\|$.

Fig. 2. Analysis of probability that $\|\mathbf{u}\|\|\mathbf{q}\| < L^k(\mathbf{B})$.

Then, the probability that we have $\|\mathbf{u}\|\|\mathbf{q}\| < L^k(\mathbf{B})$, $\Pr[\|\mathbf{u}\|\|\mathbf{q}\| < L^k(\mathbf{B})]$ is

$$\Pr[\|\mathbf{u}\|\|\mathbf{q}\| < L^k(\mathbf{B})] = \int_{-\infty}^{L^k(\mathbf{B})} f(x)dx,$$

where $x = \|\mathbf{u}\|\|\mathbf{q}\|$. This probability is high if $L^k(\mathbf{B})$ is high. We have high $L^k(\mathbf{B})$ when k is small, and small k is the standard setting, as mentioned in Section 2. Figure 2(c) illustrates an intuitive example suggesting that $\Pr[\|\mathbf{u}\|\|\mathbf{q}\| < L^k(\mathbf{B})] \approx 1$ for a high $L^k(\mathbf{B})$. In particular, under the second assumption, the expected $\|\mathbf{u}\|\|\mathbf{q}\|$ is μ, and we usually have $\mu < L^k(\mathbf{B})$ for small k. That is, for small k and a random $\mathbf{q} \in \mathbf{P}$, we have a high α in expectation. If we further assume that only $O(k)$ item vectors have $\|\mathbf{u}\|\|\mathbf{q}\| \geq L^k(\mathbf{B})$, then $\Pr[\|\mathbf{u}\|\|\mathbf{q}\| < L^k(\mathbf{B})] = 1 - O(k/m)$, which also leads to a high α.

5 APPROXIMATE REVERSE MIPS

This section answers the following question: Can approximation algorithms with quality guarantee improve reverse k-MIPS processing? To start with, we introduce the concept c-AMIPS [20, 29, 35, 43].

Definition 6 (c-Approximate MIPS problem). Given a set of vectors $\mathbf{P}$, a query vector $\mathbf{q}$, and an approximation factor $c \in (0, 1)$, the c-Approximate MIPS problem returns a vector $\mathbf{p} \in \mathbf{P}$ satisfying that $\mathbf{p} \cdot \mathbf{q} \geq c \times \mathbf{p}^* \cdot \mathbf{q}$, where $\mathbf{p}^*$ is the exact result for $\mathbf{q}$.

Extending this definition for c-k-Approximate MIPS is straightforward. Let $\mathbf{p}_i^*$ be the ith exact answer of the k-MIPS for a given $\mathbf{q}$. Then, c-k-Approximate MIPS returns a vector $\mathbf{p} \in \mathbf{P}$, which satisfies that $\mathbf{p} \cdot \mathbf{q} \geq c \times \mathbf{p}_i^* \cdot \mathbf{q}$, as the ith result. This approximation accepts vectors, which have similar inner products with a given query vector to those of the exact results, as answers.

5.1 Designing an Approximation Version of the Reverse k-MIPS Problem

Now consider the reverse k-MIPS problem. For a user vector $\mathbf{u}$ and a query vector $\mathbf{q}$, even if $\mathbf{q}$ does not exist in the k-MIPS answer for $\mathbf{u}$ but satisfies $\mathbf{u} \cdot \mathbf{q} \geq c \times \mathbf{u} \cdot \mathbf{p}_k^*$, $\mathbf{u}$ may be interested in $\mathbf{q}$. Hence, $\mathbf{u}$ can be one of the results of *approximate* reverse k-MIPS. By taking this idea, we formulate an approximation version of the reverse k-MIPS problem.

Definition 7 (Reverse c-k-Approximate MIPS problem). Given a query (item) vector $\mathbf{q}$, k, c, and two sets of vectors $\mathbf{U}$ (set of user vectors) and $\mathbf{P}$ (set of item vectors), the reverse c-k-Approximate MIPS problem finds all vectors $\mathbf{u} \in \mathbf{U}$ such that $\mathbf{q}$ can be included in the c-k-Approximate MIPS result of $\mathbf{u}$ among $\mathbf{P} \cup \{\mathbf{q}\}$ (i.e., $\mathbf{u}$ must satisfy $\mathbf{u} \cdot \mathbf{q} \geq c \times \mathbf{u} \cdot \mathbf{p}_k^*$ to be included in the result set).

Compared with the reverse k-MIPS problem, the above problem allows user vectors $\mathbf{u}$ to be included in the result iff $\mathbf{u} \cdot \mathbf{q} \geq c \times \mathbf{u} \cdot \mathbf{p}_k^*$. Therefore, *it does not miss any user vectors that are in the exact reverse k-MIPS result.*

5.2 The Algorithm

Our next challenge is how to solve this problem while guaranteeing the result quality. We can actually exploit our approaches in Simpfer to return all user vectors satisfying the condition in Definition 7. Surprisingly, we need a few modifications: We extend only Lemma 2 and Corollary 1. Recall that item vectors in $\mathbf{P}$ are sorted in descending order of norm.

Corollary 3. *If $\mathbf{u}_i \cdot \mathbf{q} \geq c\|\mathbf{u}_i\|\|\mathbf{p}_k\|$, then it is guaranteed that $\mathbf{q}$ can be included in the c-k-Approximate MIPS result of $\mathbf{u}_i$.*

Proof. From Lemma 2 and Definition 7. □

Corollary 4. *Assume that $\mathbf{q}$ is included in an intermediate result of the k-MIPS of $\mathbf{u}_i$ and we now evaluate $\mathbf{p}_j \in \mathbf{P}$. If $\mathbf{u}_i \cdot \mathbf{q} \geq c\|\mathbf{u}_i\|\|\mathbf{p}_j\|$, then it is guaranteed that $\mathbf{q}$ can be included in the final result of the c-k-Approximate MIPS of $\mathbf{u}_i$.*

Proof. This corollary is derived by combining the proof of Corollary 1 and Definition 7. □

By using Corollaries 3 and 4, we propose c-Simpfar k-MIPS. This is a variant of Simpfer. The difference is that c-Simpfar replaces

- line 9 of Algorithm 3 with "$c\|\mathbf{u}_i\|\|\mathbf{p}_k\| > \gamma$" and
- line 4 of Algorithm 2 with "$\mathbf{u} \cdot \mathbf{q} \geq c\|\mathbf{u}\|\|\mathbf{p}_i\|$".

(Therefore, when $c = 1$, c-Simpfar is equivalent to Simpfer.)

5.3 Analysis

We introduce that c-Simpfar has a quality guarantee, i.e., does not miss any user vectors that satisfy the condition in Definition 7.

Theorem 4. *For arbitrary c and $\mathbf{q}$, c-Simpfar returns all user vectors that satisfy the condition in Definition 7.*

Proof. From Corollaries 3 and 4. □

Next, we have

Theorem 5. *The time complexity of c-Simpfar is $O((1-\alpha)nm''d)$, where $m' \geq m''$.*

Proof. Corollaries 3 and 4 are applied when a given block cannot be pruned, so c-Simpfar has the same α with Simpfer. However, it is trivial to see that c-Simpfar can reduce the number of sequential scans compared with Simpfer, from Corollary 3. Also, thanks to Corollary 4, c-Simpfar can reduce the number of item vectors accessed. Therefore, m'', the average number of item vectors accessed in Linear-scan$(\cdot)$, has $m' \geq m''$. □

Remark 2. At a glance, iterating an existing c-k-AMIPS algorithm for each user vector in $\mathbf{U}$ can solve the problem defined in Definition 7. However, this iteration-based approach has two critical drawbacks. First, this approach cannot guarantee the correctness. As mentioned in Section 3, LSH-based algorithms have only *probabilistic* accuracy guarantees, so they may lose user vectors that are in the exact result. However, c-Simpfar certainly guarantees the result quality. The other approaches for approximate MIPS (e.g., References [31, 37, 47]) cannot be reasonably employed in our problem, because they have no theoretical accuracy guarantee "w.r.t. c-k-AMIPS." Second,

this iteration-based approach still needs $O(nmd)$ or $O(nmd \log n)$ time theoretically, because (i) proximity graphs and quantization techniques have no performance guarantee and (ii) the state-of-the-art LSH algorithms for c-k-AMIPS [20, 32] need superlinear time or space in the worst case. Therefore, this approach provides no clear advantages over c-Simpfar.

From the above results, we say yes as the answer to the question about the possibility of approximation algorithms for improving reverse k-MIPS processing, since we have $m' \geq m''$. However, as our experimental results show, m' of Simpfer is already small in practice, so reducing this value does not yield a significant speed-up. This means that its practical impact is unfortunately small, which is also confirmed in Section 7.3. To summarize, the approximation based on Definition 7 does not yield a significant efficiency improvement in practice, and it is an open question to design different settings/definitions allowing approximate results for further speed-up.

6 SIMPFER++

In this section, we answer the question: Is there an exact algorithm that is faster than Simpfer? Our answer is positive, and we propose a variant of Simpfer, namely Simpfer++. In a nutshell, this new algorithm has the following two main differences to Simpfer.

- *Maintaining the tightest lower bound.* To maximize the impact of Lemmas 1 and 3, it is trivial to see that the lower-bound arrays should have tighter values. Notice that, for each user vector $\mathbf{u}_i \in \mathbf{U}$, its jth ($1 \leq j \leq k_{max}$) MIPS result on $\mathbf{P}$ is obtained offline, as it does not depend on $\mathbf{q}$. Simpfer++ does this in its pre-processing step. That is, it sets

$$L_i^j = \mathbf{u}_i \cdot \mathbf{p}_j^*, \tag{3}$$

  where $\mathbf{p}_j^* \in \mathbf{P}$ is the jth MIPS result for $\mathbf{u}_i$ on $\mathbf{P}$. From this equation, it is trivial to see that L_i^k is the tightest lower bound of the top kth inner product for $\mathbf{u}_i$. Thanks to this, given a query vector $\mathbf{q}$, to see whether $\mathbf{u}$ can be included in the reverse k-MIPS, all we have to do online is to compare L_i^k with $\mathbf{u}_i \cdot \mathbf{q}$. This means that Simpfer++ does not need LINEAR-SCAN$(\cdot)$ anymore, removing the factor of m' held by the time complexity of Simpfer.
- *Using matrix multiplication.* Consider $\mathbf{U}$ as a matrix:

$$\mathbf{U} = \begin{pmatrix} \mathbf{u}_1 \\ \mathbf{u}_2 \\ \vdots \\ \mathbf{u}_n \end{pmatrix}.$$

  From the first difference, what we need to do is to compute $\mathbf{U}\mathbf{q}^t$, where $\mathbf{q}^t$ is the transposition of $\mathbf{q}$. That is, when Simpfer++ computes inner products of user vectors and $\mathbf{q}$, it employs matrix multiplication. This manner can exploit hardware optimization, so this computation is faster than simply computing $\mathbf{u} \cdot \mathbf{q}$ in an iteration manner.
  As with Simpfer, Simpfer++ uses the block-based user filtering, i.e., the vectors in blocks are maintained in the matrix manner. This means that the cost of Simpfer++ is usually less than the cost of computing $\mathbf{U}\mathbf{q}^t$, as analyzed in Section 4.4.

The above differences enable filtering both user and all item vectors and (ii) fast inner product computation if necessary, which guarantees the superiority of Simpfer++ against Simpfer practically and theoretically (see Theorem 6). In the subsequent sub-sections, we present operational differences to those of Simpfer,

6.1 Pre-processing

There are two main differences from the pre-processing of Simpfer.

- Instead of using $O(k_{max})$ vectors of $\mathbf{P}$, we use Equation (3) to compute the lower-bound array of each $\mathbf{u}_i \in \mathbf{U}$. This corresponds to **$k$-Maximum Inner Product Join (k-MIPJ)**, which retrieves the top-k (item) vectors for each user vector in $\mathbf{Q}$. We use LEMP [39], a state-of-the-art k-MIPJ algorithm, to compute the lower-bound arrays.
- We maintain the user vectors in each block $\mathbf{B}_i$ by using a matrix $\mathbf{M}_i$, where its each row corresponds a user vector. That is,

$$\mathbf{M}_i = \begin{pmatrix} \mathbf{u}_{(i-1)b+1} \\ \mathbf{u}_{(i-1)b+2} \\ \vdots \\ \mathbf{u}_{ib} \end{pmatrix}$$

  (we use the same assumption w.r.t. vector order in Section 4.1). This is because we use fast matrix operations in the online processing, as stated earlier. The detail is described in Section 6.2.

The other operations are the same as those in Algorithm 1.

Remark 3. Clearly, the space complexity of Simpfer++ is the same as Simpfer, so it is $O(n)$. However, the time complexity of the pre-processing of Simpfer++ is $O(nmd)$. This needs a more computational cost than $O(n(d + \log n))$ time, i.e., the pre-processing time of Simpfer. However, LEMP uses several filtering techniques to reduce its practical time as noted in Section 3, and LEMP can be parallelized by using multi-threading [38]. Recall that this pre-processing is done only once, so the pre-processing of Simpfer++ is still reasonable, as our experimental results show (see Table 5).

6.2 Online Processing

6.2.1 Algorithm Description. Algorithm 4 describes the online processing of Simpfer++. Compared with Simpfer, Simpfer++ has two differences:

- Given a block $\mathbf{B}_i$, after applying Lemma 3 (the block-based filtering), we compute the inner products of $\mathbf{q}$ and every user vectors in $\mathbf{B}_i$ via a matrix multiplication. Specifically, we have

$$\mathbf{M}_i\mathbf{q}^t = \begin{pmatrix} \mathbf{u}_{(i-1)b+1} \cdot \mathbf{q} \\ \mathbf{u}_{(i-1)b+2} \cdot \mathbf{q} \\ \vdots \\ \mathbf{u}_{ib} \cdot \mathbf{q} \end{pmatrix}.$$

  We then evaluate whether each row of $\mathbf{M}_i\mathbf{q}^t$ is larger than the corresponding lower bound. Iff yes, it is added into the result set.
- Simpfer++ does not use Lemma 2 and Linear-scan$(\cdot)$, since they are not necessary.

6.2.2 Analysis. The main result of Simpfer++ is that *its online time always beats those of Simpfer and any solutions that iteratively run an exact or approximate k-MIPS algorithm.* This is demonstrated by:

Theorem 6. *The time complexity of Simpfer++ is $O((1 - \alpha')nd)$, where α' is the pruning ratio of blocks in $\mathcal{B}$, and we have $\alpha' \geq \alpha$.*

Proof. From Algorithm 4, we see that (i) Simpfer++ computes the inner product of $\mathbf{u}$ and $\mathbf{q}$ only when its block is not filtered and (ii) Simpfer++ does not access $\mathbf{P}$ online. Then, it is straightforward

ALGORITHM 4: SIMPFER++

Input: $\mathbf{U}$, $\mathbf{P}$, $\mathbf{q}$, k, and $\mathcal{B}$

$\mathbf{U}_r \leftarrow \emptyset$
Compute $\|\mathbf{q}\|$
for *each* $\mathbf{B} \in \mathcal{B}$ **do**
  $\mathbf{u} \leftarrow$ the first user vector in $\mathbf{U}(\mathbf{B})$
  if $\|\mathbf{u}\|\|\mathbf{q}\| > L^k(\mathbf{B})$ **then**
    $IP \leftarrow \mathbf{M}_i\mathbf{q}^t$ ▷ IP is a set of the inner products with $\mathbf{q}$ and the user vectors in $\mathbf{M}_i$
    for *each* $i = 1$ *to* $|\mathbf{M}_i|$ **do**
      $\mathbf{u}_i \leftarrow$ the jth vector in $\mathbf{B}$
      if $IP_j \geq L_i^k$, *where* $IP_j = \mathbf{u}_i \cdot \mathbf{q}$ **then**
        $\mathbf{U}_r \leftarrow \mathbf{U}_r \cup \{\mathbf{u}_i\}$
return $\mathbf{U}_r$

to see that its time complexity is $O((1-\alpha')nd)$. In addition, as Simpfer++ is guaranteed to use lower bounds that are tighter than or the same as Simpfer, it is obvious that $\alpha' \geq \alpha$. □

7 EXPERIMENT

This section reports our experimental results. All experiments were conducted on a Ubuntu 18.04 LTS machine with a 12-core 3.0-GHz Intel Xeon E5-2687w v4 processor and 512 GB RAM.

7.1 Setting

7.1.1 Datasets. We used four popular real datasets: MovieLens,[7] Netflix, Amazon,[8] and Yahoo!.[9] These are usually used as benchmark data in MIPS works (under recommendation scenarios) [9, 20, 27, 37, 50].

The user and item vectors of these datasets were obtained by the Matrix Factorization in Reference [5]. These are 50-dimensional vectors (the dimensionality setting is the same as in References [23, 39][10]). The other statistics is shown in Table 3. We randomly chose 1,000 vectors as query vectors from $\mathbf{P}$.

7.1.2 Evaluated Algorithms. We evaluated the following algorithms.

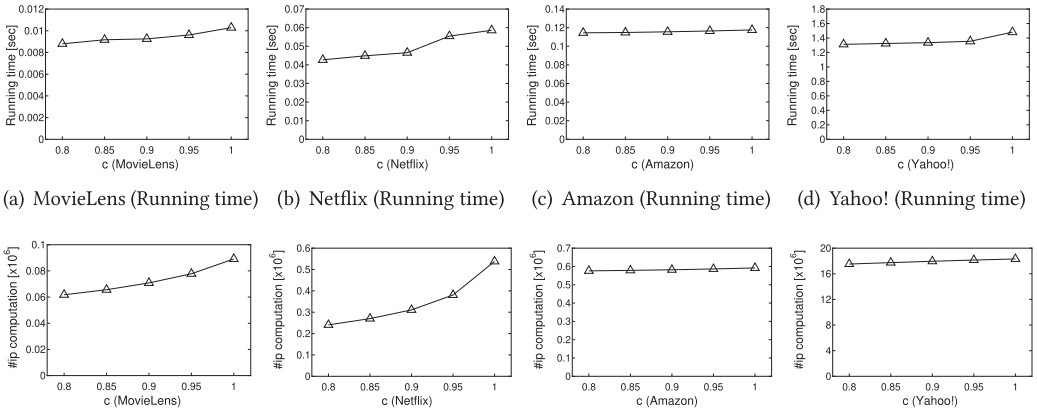
- LEMP [39]: the state-of-the-art exact *all-k*-MIPS algorithm. LEMP originally does k-MIPS for all user vectors in $\mathbf{U}$.
- FEXIPRO [23]: the state-of-the-art exact k-MIPS algorithm. We simply ran FEXIPRO for each $\mathbf{u} \in \mathbf{U}$.
- Simpfer: the exact algorithm proposed in Section 4.
- c-Simpfar: the approximation algorithm proposed in Section 5.
- Simpfer++: the exact algorithm proposed in Section 6. We used Eigen[11] for matrix operations.

We set $k_{max} = 25$. These algorithms were implemented in C++ and compiled by g++ 7.5.0 with -O3 flag.

[7] https://grouplens.org/datasets/movielens/.
[8] https://jmcauley.ucsd.edu/data/amazon/.
[9] https://webscope.sandbox.yahoo.com/.
[10] As our theoretical analysis shows, the time of Simpfer is trivially proportional to d, thus its empirical impact is omitted.
[11] https://eigen.tuxfamily.org/dox/.

Table 3. Dataset Statistics

	MovieLens	Netflix	Amazon	Yahoo!
$\|\mathbf{U}\|$	138,493	480,189	1,948,882	2,088,620
$\|\mathbf{P}\|$	26,744	17,770	98,211	200,941



(a) MovieLens (Running time) (b) Netflix (Running time) (c) Amazon (Running time) (d) Yahoo! (Running time)

(e) MovieLens (#ip computations) (f) Netflix (#ip computations) (g) Amazon (#ip computations) (h) Yahoo! (#ip computations)

Fig. 3. Impact of c of c-Simpfar: Running time (top) and #ip computations (bottom).

Note that References [23, 39] have demonstrated that the other exact MIPS algorithms are outperformed by LEMP and FEXIPRO, so we did not use them as competitors. All experiments except for Section 7.3 focused on the exact answer, thus we did not use approximate MIPS algorithms as competitors. (In Section 7.5, we demonstrate that solutions that iteratively run an approximate MIPS algorithm cannot beat Simpfer and Simpfer++.) In addition, LEMP and FEXIPRO also have a pre-processing (offline) phase. We did not include the offline time as the running time.

7.2 Effectiveness of Blocks

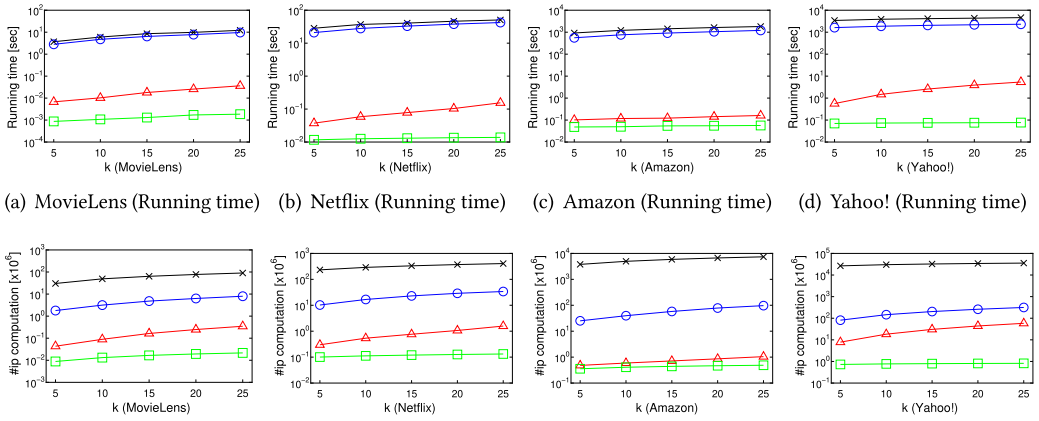
We first clarify the effectiveness of blocks employed in Simpfer. To show this, we compare Simpfer with Simpfer without blocks (which does not evaluate line 5 of Algorithm 3). We set $k = 10$.

On MovieLens, Netflix, Amazon, and Yahoo!, Simpfer (Simpfer without blocks) takes 10.3 (22.0), 58.6 (100.8), 117.6 (446.2), and 1481.2 (1586.2) [msec], respectively. This result demonstrates that, although the speed-up ratio is affected by data distributions, blocks surely yield speed-up.

7.3 Impact of c

We here report the performance of c-Simpfar. Figure 3 depicts the result with varying c (we set $k = 10$). Note that $c = 1$ corresponds to the exact case (i.e., Simpfer). Recall Theorem 4: c-Simpfar does not miss any user vectors that satisfy the condition in Definition 7. We hence focus on its running time.

From Figure 3(a)–(d), we see that c-Simpfar is faster than Simpfer, which is consist with the theoretical result in Section 5, but it does not provide a significant speed-up compared with Simpfer. Lemma 2 and Corollary 1 already function well, so m' in Theorem 3 is sufficiently small (this is shown in Table 4). Hence, their relaxed versions, Corollaries 3 and 4, bring a little gains, implying that $m' \geq m''$ but $m' \approx m''$. This is confirmed by the results in Figure 3(e)–(h) showing that the



(a) MovieLens (Running time) (b) Netflix (Running time) (c) Amazon (Running time) (d) Yahoo! (Running time)

(e) MovieLens (#ip computations) (f) Netflix (#ip computations) (g) Amazon (#ip computations) (h) Yahoo! (#ip computations)

Fig. 4. Impact of k: Running time (top) and #ip computations (bottom). "×" shows LEMP, "○" shows FEXIPRO, "△" shows Simpfer, and "□" shows Simpfer++.

number of inner product (ip) computations do not reduce so much even when $c = 0.8$. Therefore, we do not consider c-Simpfar in the subsequent experiments.

7.4 Efficiency of Matrix Multiplication

Next, we show the efficiency of matrix multiplication employed by Simpfer++. As a competitor, we used a variant of Simpfer++ that simply computes inner products, denoted by Simpfer++ without matrix. On MovieLens, Netflix, Amazon, and Yahoo!, Simpfer++ (Simpfer++ without matrix) takes 1.1 (2.4), 12.6 (22.6), 49.6 (91.0), and 73.1 (148.8) (ms), respectively. We see that Simpfer++ obtains about 2x speed-up against its variant. This result clarifies the importance of employing matrix-based implementation to reduce practical time.

7.5 Impact of k

We investigate how k affects the computational performance of each algorithm. Figure 4 depicts the experimental results.

We first observe that, as k increases, the running time of each algorithm increases, as shown in Figure 4(a)–(d). This is reasonable, because the cost of (decision version of) k-MIPS increases and k affects α' of Simpfer++. As a proof, Figure 4(e)–(h) show that the number of inner product computations increases as k increases. The running time of Simpfer is (sub-)linear to k (the plots are log-scale). This suggests that $m' = O(k)$. However, the running time of Simpfer++ is less affected by k, compared with the other algorithms. The reason is that Simpfer++ does not run any k-MIPS. Therefore, Simpfer++ always outperforms the other algorithms. Simpfer++ is in particular useful on Yahoo!, and it has 70× speed-up against Simpfer. We found that, compared with the cases of the other datasets, Simpfer needs more LINEAR-SCAN$(\cdot)$ on Yahoo!, thereby Simpfer++, which does not run LINEAR-SCAN$(\cdot)$, functions well.

Second, Simpfer significantly outperforms LEMP and FEXIPRO. This result is derived from our idea of quickly solving the k-MIPS decision problem. The techniques introduced in Section 4.2 can deal with both yes and no answer cases efficiently. Therefore, our approach functions quite well in practice. An interesting observation is the performance differences between FEXIPRO and Simpfer. Let us compare them with regard to running time. Simpfer *is at least two* orders of magnitude

Table 4. Average #ip Computations per User Vector in Simpfer

k	MovieLens	Netflix	Amazon	Yahoo!
5	0.31	0.62	0.23	4.02
10	0.64	1.12	0.28	9.40
15	1.18	1.59	0.34	15.64
20	1.79	2.22	0.41	22.49
25	2.51	3.29	0.50	29.87

faster than FEXIPRO. However, with regard to the number of inner product computations, that of Simpfer is *one* order of magnitude lower than that of FEXIPRO. This result suggests that the filtering cost of Simpfer is light, whereas that of FEXIPRO is heavy. Recall that Lemmas 1–3 need only $O(1)$ time, and Corollaries 1–2 need $O(k)$ time in practice. However, for each user vector in $\mathbf{U}$, FEXIPRO incurs $\Omega(k)$ time, and its filtering cost is $O(d')$, where $d' < d$. For high-dimensional vectors, the difference between $O(1)$ and $O(d')$ is large. From this point of view, we can see the efficiency of Simpfer.

Last, we focus on the average number of inner product computations *per user* in Simpfer, i.e., m' in Theorem 3. This is shown in Table 4. Although we have a few exceptions, we have $m' < k$ (and the exceptions have trivial differences). Notice that the average number of inner product computations *per user* in Simpfer++ is at most 1. This results confirms that any solutions that iteratively run a k-MIPS algorithm cannot beat Simpfer and Simpfer++. *This also holds for any solutions that use an approximate k-MIPS algorithm*, because even the state-of-the-art approximation algorithm [13] requires $\Omega(kd)$ time[12] for an approximate k-MIPS.

7.6 Impact of Cardinality of U

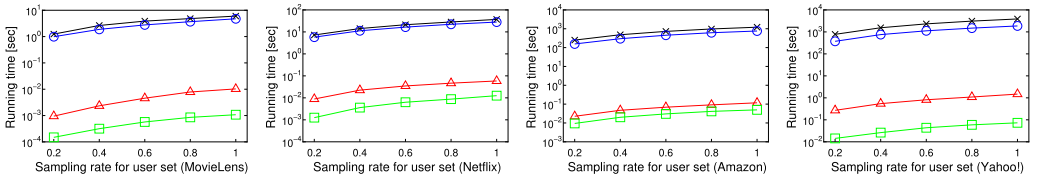
We next study the scalability to $n = |\mathbf{U}|$. To this end, we randomly sampled $s \times n$ user vectors in $\mathbf{U}$, and this sampling rate s has $s \in [0.2, 1.0]$. We set $k = 10$. Figure 5 shows the experimental result.

In a nutshell, we have a similar result to that in Figure 4. As n increases, the running times of Simpfer and Simpfer++ linearly increase. This result is consistent with Theorems 3 and 6. Notice that the tendency of the running times of Simpfer and Simpfer++ follow that of the number of inner product computations. This phenomenon is also supported by Theorems 3 and 6, because the main bottlenecks of Simpfer and Simpfer++ are Linear-scan$(\cdot)$ and matrix multiplications, respectively.

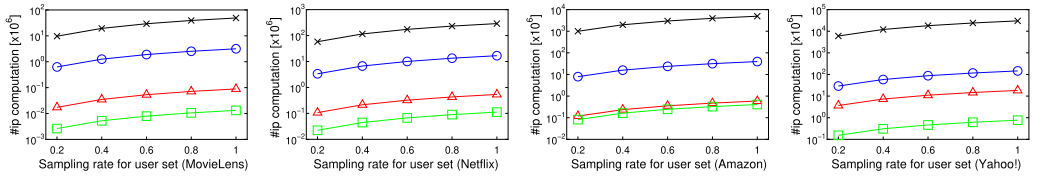
7.7 Impact of Cardinality of P

The scalability to $m = |\mathbf{P}|$ is also investigated. We randomly sampled $s \times m$ user vectors in $\mathbf{P}$, as with the previous section. Figure 6 shows the experimental result, where $k = 10$. Interestingly, we see that the result is different from that in Figure 5. The running times of Simpfer and Simpfer++ are almost stable for different m. In this experiment, n and k were fixed, and recall that $m' = O(k)$. From this observation, the stable performance of Simpfer is theoretically obtained. Recall again that Simpfer++ does not run any k-MIPS, so m has an impact only on L^k of each user vector. The experimental result suggests that this impact is negligible. This scalability of Simpfer and Simpfer++ is an advantage over the other algorithms, since the running times of the baselines increase as m increases.

[12]The state of the art [13] suggests accessing at least $\beta \times k$ vectors, where β is sufficiently large (e.g., 10) to obtain an accurate result. Therefore, it needs much more item vector accesses for a user vector than Simpfer.

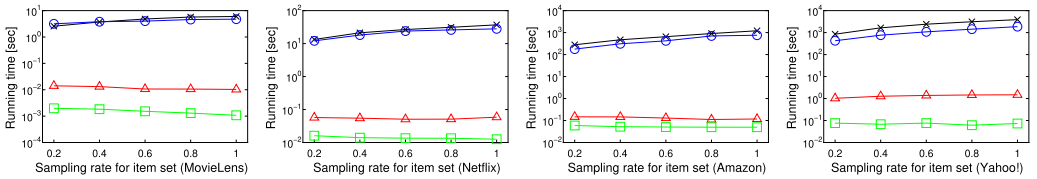


(a) MovieLens (Running time) (b) Netflix (Running time) (c) Amazon (Running time) (d) Yahoo! (Running time)

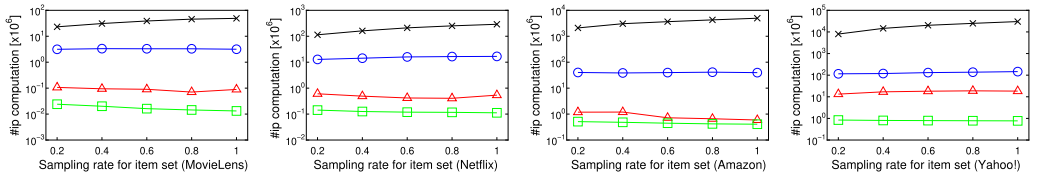


(e) MovieLens (#ip computations) (f) Netflix (#ip computations) (g) Amazon (#ip computations) (h) Yahoo! (#ip computations)

Fig. 5. Impact of |U|: Running time (top) and #ip computations (bottom). "×" shows LEMP, "∘" shows FEXIPRO, "△" shows Simpfer, and "□" shows Simpfer++.



(a) MovieLens (Running time) (b) Netflix (Running time) (c) Amazon (Running time) (d) Yahoo! (Running time)



(e) MovieLens (#ip computations) (f) Netflix (#ip computations) (g) Amazon (#ip computations) (h) Yahoo! (#ip computations)

Fig. 6. Impact of |P|: Running time (top) and #ip computations (bottom). "×" shows LEMP, "∘" shows FEXIPRO, "△" shows Simpfer, and "□" shows Simpfer++.

7.8 Pre-processing Time

Last, we report the pre-processing times of Simpfer and Simpfer++. We used 12 threads for their pre-processing (we used OpenMP for multi-threading). Table 5 shows the results. As Theorem 1 demonstrates, the pre-processing times of Simpfer and Simpfer++ increase as n increases.

We see that the pre-processing time of Simpfer is reasonable and much faster than the online (running) time of the baselines. For example, when $k = 25$, the running time of FEXIPRO on Amazon with 12 threads was 114 [sec]. When $k = 25$ (i.e., $k = k_{max}$), the total time of pre-processing and online processing of Simpfer with 12 threads is $11.18 + 0.02 = 11.20$ (s). Therefore, even if $k > k_{max}$ is specified, re-building blocks then processing the query by Simpfer is still faster.

For small datasets (i.e., MovieLens and Netflix), the pre-processing time of Simpfer++ is competitive with that of Simpfer. However, for large datasets (i.e., Amazon and Yahoo!), the pre-processing

Table 5. Pre-processing Time of Simpfer and Simpfer++ [sec]

	MovieLens	Netflix	Amazon	Yahoo!
Simpfer	0.78	2.55	11.18	10.52
Simpfer++	1.00	3.56	77.79	191.68

time of Simpfer++ is clearly longer than that of Simpfer. However, it still completes the pre-processing within about a few minutes. If user and item vectors are not updated so frequently and applications do not change k_{max} frequently, then Simpfer++ can be the first choice.

8 CONCLUSION

This article introduced a new problem, reverse maximum inner product search (reverse MIPS). The reverse MIPS problem supports many applications, such as recommendation, advertisement, and market analysis. Because even state-of-the-art algorithms for MIPS cannot solve the reverse MIPS problem efficiently, we proposed Simpfer as an exact and efficient solution. Simpfer exploits several techniques to efficiently answer the decision version of the MIPS problem. Our theoretical analysis has demonstrated that Simpfer is always better than a solution that employs a state-of-the-art algorithm of MIPS. We also proposed an approximation version of the reverse MIPS problem. As its solution, we proposed c-Simpfar that always guarantees the result quality. In addition, we extended Simpfer to demonstrate that there exists an exact algorithm that is faster than Simpfer. Our last algorithm, Simpfer++, requires at most $O(nd)$ time, i.e., it needs to access only user vectors, different from Simpfer and c-Simpfar. Our experimental results on four real datasets show that (i) Simpfer is at least two orders of magnitude faster than the MIPS-based solutions, (ii) c-Simpfar can improve the reverse MIPS processing but its impact is not significant, and (iii) Simpfer++ yields the fastest online processing time.

REFERENCES

[1] Firas Abuzaid, Geet Sethi, Peter Bailis, and Matei Zaharia. 2019. To index or not to index: Optimizing exact maximum inner product search. In *ICDE*. 1250–1261.

[2] Daichi Amagata and Takahiro Hara. 2021. Reverse maximum inner product search: How to efficiently find users who would like to buy my item? In *RecSys*. 273–281.

[3] Yoram Bachrach, Yehuda Finkelstein, Ran Gilad-Bachrach, Liran Katzir, Noam Koenigstein, Nir Nice, and Ulrich Paquet. 2014. Speeding up the xbox recommender system using a euclidean transformation for inner-product spaces. In *RecSys*. 257–264.

[4] Chong Chen, Min Zhang, Yongfeng Zhang, Weizhi Ma, Yiqun Liu, and Shaoping Ma. 2020. Efficient heterogeneous collaborative filtering without negative sampling for recommendation. In *AAAI*. 19–26.

[5] Wei-Sheng Chin, Bo-Wen Yuan, Meng-Yuan Yang, Yong Zhuang, Yu-Chin Juan, and Chih-Jen Lin. 2016. LIBMF: A library for parallel matrix factorization in shared-memory systems. *J. Mach. Learn. Res.* 17, 1 (2016), 2971–2975.

[6] Edith Cohen and David D. Lewis. 1999. Approximating matrix multiplication for pattern recognition tasks. *J. Algor.* 30, 2 (1999), 211–252.

[7] Paolo Cremonesi, Yehuda Koren, and Roberto Turrin. 2010. Performance of recommender algorithms on top-n recommendation tasks. In *RecSys*. 39–46.

[8] Ryan R. Curtin, Parikshit Ram, and Alexander G. Gray. 2013. Fast exact max-kernel search. In *SDM*. 1–9.

[9] Xinyan Dai, Xiao Yan, Kelvin K. W. Ng, Jiu Liu, and James Cheng. 2020. Norm-explicit quantization: Improving vector quantization for maximum inner product search. In *AAAI*. 51–58.

[10] Shuai Ding and Torsten Suel. 2011. Faster top-k document retrieval using block-max indexes. In *SIGIR*. 993–1002.

[11] Marcus Fontoura, Vanja Josifovski, Jinhui Liu, Srihari Venkatesan, Xiangfei Zhu, and Jason Zien. 2011. Evaluation strategies for top-k queries over memory-resident inverted indexes. *Proc. VLDB* 4, 12 (2011), 1213–1224.

[12] Marco Fraccaro, Ulrich Paquet, and Ole Winther. 2016. Indexable probabilistic matrix factorization for maximum inner product search. In *AAAI*. 1554–1560.

[13] Ruiqi Guo, Philip Sun, Erik Lindgren, Quan Geng, David Simcha, Felix Chern, and Sanjiv Kumar. 2020. Accelerating large-scale inference with anisotropic vector quantization. In *ICML*. 3887–3896.

[14] Rob Hall and Josh Attenberg. 2015. Fast and accurate maximum inner product recommendations on map-reduce. In *World Wide Web*. 1263–1268.
[15] Xiangnan He, Lizi Liao, Hanwang Zhang, Liqiang Nie, Xia Hu, and Tat-Seng Chua. 2017. Neural collaborative filtering. In *World Wide Web*. 173–182.
[16] Kohei Hirata, Daichi Amagata, Sumio Fujita, and Takahiro Hara. 2022. Solving diversity-aware maximum inner product search efficiently and effectively. In *RecSys*. 198–207.
[17] Kohei Hirata, Daichi Amagata, Sumio Fujita, and Takahiro Hara. 2023. Categorical diversity-aware inner product search. *IEEE Access* 11 (2023), 2586–2596.
[18] Kohei Hirata, Daichi Amagata, and Takahiro Hara. 2022. Cardinality estimation in inner product space. *IEEE Open J. Comput. Soc.* 3 (2022), 208–216.
[19] Jun Hu and Ping Li. 2018. Collaborative filtering via additive ordinal regression. In *WSDM*. 243–251.
[20] Qiang Huang, Guihong Ma, Jianlin Feng, Qiong Fang, and Anthony K. H. Tung. 2018. Accurate and fast asymmetric locality-sensitive hashing scheme for maximum inner product search. In *KDD*. 1561–1570.
[21] Jyun-Yu Jiang, Patrick H. Chen, Cho-Jui Hsieh, and Wei Wang. 2020. Clustering and constructing user coresets to accelerate large-scale top-k recommender systems. In *The Web Conference*. 2177–2187.
[22] Noam Koenigstein, Parikshit Ram, and Yuval Shavitt. 2012. Efficient retrieval of recommendations in a matrix factorization framework. In *CIKM*. 535–544.
[23] Hui Li, Tsz Nam Chan, Man Lung Yiu, and Nikos Mamoulis. 2017. FEXIPRO: Fast and exact inner product retrieval in recommender systems. In *SIGMOD*. 835–850.
[24] Wen Li, Ying Zhang, Yifang Sun, Wei Wang, Mingjie Li, Wenjie Zhang, and Xuemin Lin. 2020. Approximate nearest neighbor search on high dimensional data-experiments, analyses, and improvement. *IEEE Trans. Knowl. Data Eng.* 32, 8 (2020), 1475–1488.
[25] Jie Liu, Xiao Yan, Xinyan Dai, Zhirong Li, James Cheng, and Ming-Chang Yang. 2020. Understanding and improving proximity graph based maximum inner product search. In *AAAI*. 139–146.
[26] Rui Liu, Tianyi Wu, and Barzan Mozafari. 2019. A bandit approach to maximum inner product search. In *AAAI*. 4376–4383.
[27] Stanislav Morozov and Artem Babenko. 2018. Non-metric similarity graphs for maximum inner product search. In *NeurIPS*. 4721–4730.
[28] Hayato Nakama, Daichi Amagata, and Takahiro Hara. 2021. Approximate top-k inner product join with a proximity graph. In *IEEE Big Data*. 4468–4471.
[29] Behnam Neyshabur and Nathan Srebro. 2015. On symmetric and asymmetric lshs for inner product search. In *ICML*. 1926–1934.
[30] Ulrich Paquet and Noam Koenigstein. 2013. One-class collaborative filtering with random graphs. In *World Wide Web*. 999–1008.
[31] Ninh Pham. 2021. Simple yet efficient algorithms for maximum inner product search via extreme order statistics. In *KDD*. 1339–1347.
[32] Ninh Pham and Tao Liu. 2022. Falconn++: A locality-sensitive filtering approach for approximate nearest neighbor search. In *NeurIPS*.
[33] Parikshit Ram and Alexander G. Gray. 2012. Maximum inner-product search using cone trees. In *KDD*. 931–939.
[34] Steffen Rendle, Walid Krichene, Li Zhang, and John Anderson. 2020. Neural collaborative filtering vs. matrix factorization revisited. In *RecSys*. 240–248.
[35] Anshumali Shrivastava and Ping Li. 2014. Asymmetric LSH (ALSH) for sublinear time maximum inner product search (MIPS). In *NIPS*. 2321–2329.
[36] Anshumali Shrivastava and Ping Li. 2015. Asymmetric minwise hashing for indexing binary inner products and set containment. In *World Wide Web*. 981–991.
[37] Shulong Tan, Zhaozhuo Xu, Weijie Zhao, Hongliang Fei, Zhixin Zhou, and Ping Li. 2021. Norm adjusted proximity graph for fast inner product retrieval. In *KDD*. 1552–1560.
[38] Christina Teflioudi and Rainer Gemulla. 2016. Exact and approximate maximum inner product search with lemp. *ACM Trans. Database Syst.* 42, 1 (2016), 1–49.
[39] Christina Teflioudi, Rainer Gemulla, and Olga Mykytiuk. 2015. Lemp: Fast retrieval of large entries in a matrix product. In *SIGMOD*. 107–122.
[40] Aaron Van den Oord, Sander Dieleman, and Benjamin Schrauwen. 2013. Deep content-based music recommendation. In *NIPS*. 2643–2651.
[41] Akrivi Vlachou, Christos Doulkeridis, Yannis Kotidis, and Kjetil Nørvåg. 2010. Reverse top-k queries. In *ICDE*. 365–376.
[42] Akrivi Vlachou, Christos Doulkeridis, Yannis Kotidis, and Kjetil Norvag. 2011. Monochromatic and bichromatic reverse top-k queries. *IEEE Trans. Knowl. Data Eng.* 23, 8 (2011), 1215–1229.
[43] Xiao Yan, Jinfeng Li, Xinyan Dai, Hongzhi Chen, and James Cheng. 2018. Norm-ranging lsh for maximum inner product search. In *NeurIPS*. 2952–2961.

[44] Shiyu Yang, Muhammad Aamir Cheema, Xuemin Lin, and Wei Wang. 2015. Reverse k nearest neighbors query processing: Experiments and analysis. *Proc. VLDB* 8, 5 (2015), 605–616.
[45] Hsiang-Fu Yu, Cho-Jui Hsieh, Qi Lei, and Inderjit S Dhillon. 2017. A greedy approach for budgeted maximum inner product search. In *NIPS*. 5453–5462.
[46] Hamed Zamani and W Bruce Croft. 2020. Learning a joint search and recommendation model from user-item interactions. In *WSDM*. 717–725.
[47] Han Zhang, Hongwei Shen, Yiming Qiu, Yunjiang Jiang, Songlin Wang, Sulong Xu, Yun Xiao, Bo Long, and Wen-Yun Yang. 2021. Joint learning of deep retrieval model and product quantization based embedding index. In *SIGIR*. 1718–1722.
[48] Zhao Zhang, Cheqing Jin, and Qiangqiang Kang. 2014. Reverse k-ranks query. *Proc. VLDB* 7, 10 (2014), 785–796.
[49] Xing Zhao, Ziwei Zhu, Yin Zhang, and James Caverlee. 2020. Improving the estimation of tail ratings in recommender system with multi-latent representations. In *WSDM*. 762–770.
[50] Zhixin Zhou, Shulong Tan, Zhaozhuo Xu, and Ping Li. 2019. Möbius transformation for fast inner product search on graph. In *NeurIPS*. 8216–8227.

Received 22 October 2021; revised 25 February 2023; accepted 27 February 2023