

QSRP: Efficient Reverse k -Ranks Query Processing on High-dimensional Embeddings

Zheng Bian^{1,2}, Xiao Yan², Jiahao Zhang³, Man Lung Yiu¹, Bo Tang^{2§}

¹Department of Computing, The Hong Kong Polytechnic University

²Department of Computer Science and Engineering, Southern University of Science and Technology

³Huawei Technologies Co., Ltd.

{cszbian, csmlyiu}@comp.polyu.edu.hk, {yanx, tangb3}@sustech.edu.cn, cupermanrose@gmail.com

Abstract—Embedding models represent users and products as high-dimensional embedding vectors and are widely used for recommendation. In this paper, we study the reverse k -ranks query, which finds the users that are the most interested in a product and has many applications including product promotion, targeted advertising, and market analysis. As reverse k -ranks solutions for low dimensionality (e.g., trees) fail for the high-dimensional embeddings generated by embedding models, we propose the QSRP framework. QSRP precomputes the score table between all user and product embeddings to facilitate pruning and refinement at query time. As the score table is usually large, QSRP samples some of its columns as the index to fit in memory. To tackle the problem that naive uniform sampling results in poor pruning effect, we propose *query-aware sampling*, which conducts sampling by explicitly maximizing the pruning effect for a set of sample queries. Moreover, we introduce *regression-based pruning*, which fits cheap linear functions to predict the bounds used for pruning. We also design techniques to build the index with limited memory, reduce index building time, and handle updates. We evaluate QSRP under various configurations and compare with state-of-the-art baselines. The results show that QSRP achieves shorter query time than the baselines in all cases, and the speedup is usually over 100x.

Index Terms—Reverse k -Ranks, Ranking Query, High Dimensional Data, Product-oriented Recommendation, Embedding Model

I. INTRODUCTION

To conduct recommendations, embedding models learn to embed users and products as high-dimensional vectors from data (e.g., user-product interactions, knowledge graphs, and text descriptions) [1], [2], [3]. Representative embedding models include matrix factorization [1], [4], [5], neural network-based models [6], [7], [8], and graph neural networks [9], [10], [11]. They are widely adopted in industry and extensively explored in academia due to their strong learning ability and good recommendation performance. The high-dimensional vectors produced by these models can be stored in vector databases [12], [13] and used for various downstream applications, e.g., product recommendation [14], video recommendation [8], and music recommendation [15].

Mathematically, every user u in a user set \mathcal{U} (and every product p in a product set \mathcal{P}) is represented by an embedding vector. The preference of a user for a product is indicated by

the inner product between their embeddings, i.e., $f(u, p) = u^\top p$ (also called the score) [16]. Each user ranks the products in the product set according to their scores in descending order. Given the embedding vectors of users and products and the score function, many works study queries for downstream applications, and we classify these queries into the following two categories.

(I) User-oriented queries. A representative of this category is *maximum inner product search (MIPS)* [17], which takes a user and an integer k and returns the k products that have the largest scores for the user. MIPS recommends products to a user and is a key subroutine in recommendation systems [12]. Various techniques [18], [19], [20] have been proposed to process the MIPS query efficiently for high-dimensional embeddings, e.g., vector quantization, and proximity graph.

(II) Product-oriented queries. *Reverse k -ranks* is a well-known query in this category [21], [22], [23]. Given a product p and an integer k , reverse k -ranks returns the k users that give the smallest ranks to the query product. Intuitively, reverse k -ranks recommends users to products by identifying the k users that are the most interested in a product. It has many applications such as product promotion, targeted advertising, and market analysis. For example, in product promotion, a product producer has a budget to buy online advertisements [24], and reverse k -ranks query can identify the k most promising users. In Section II, we compare reverse k -ranks with other product-oriented queries and show that it yields better accuracy.

In this work, we focus on processing reverse k -ranks query on the high-dimensional embedding vectors generated by embedding models. The reverse k -ranks query has been studied [21], [22], [25] but existing works assume that each dimension of the embeddings has explicit meanings (e.g., price of a hotel and age of customer), and thus the embedding dimensionality is low (e.g., <10). Their solutions use either computational geometry techniques (e.g., skyline [26], onion layer [27], and UTK [28]) or space-partitioning data structures (e.g., R-tree [29], M-tree [30], and τ -LevelIndex [31]). These techniques fail in high dimensionality (e.g., >50) due to the curse of dimensionality [32]. For example, the pruning effect of R-tree diminishes with dimensionality and almost vanishes when the dimensionality exceeds 30 [22], and the skyline

[§]Dr. Bo Tang is the corresponding author.

| Product set | | | | User set | | Score | Sorted list | |
|-----------------------------|-----------|----------------------------|-----------|-------------------------|-----------|-------------------------------|-------------|--|
| Inception(\mathbf{p}_1) | (0.6,0.9) | Léon(\mathbf{p}_5) | (0.3,0.6) | Bob(\mathbf{u}_1) | (1.5,0.9) | $f(\mathbf{u}_1, \mathbf{q})$ | 4.59 | $\langle \mathbf{p}_4, 5.13 \rangle, \langle \mathbf{p}_3, 4.77 \rangle, \langle \mathbf{q}, 4.59 \rangle, \langle \mathbf{p}_6, 4.41 \rangle, \langle \mathbf{p}_2, 2.73 \rangle, \langle \mathbf{p}_1, 1.71 \rangle, \langle \mathbf{p}_7, 1.35 \rangle, \langle \mathbf{p}_5, 0.99 \rangle$ |
| Devotion(\mathbf{p}_2) | (0.2,2.7) | Iron Man(\mathbf{p}_6) | (2.4,0.9) | Mary(\mathbf{u}_2) | (2.7,1.2) | $f(\mathbf{u}_2, \mathbf{q})$ | 8.01 | $\langle \mathbf{p}_4, 8.10 \rangle, \langle \mathbf{q}, 8.01 \rangle, \langle \mathbf{p}_3, 7.83 \rangle, \langle \mathbf{p}_6, 7.56 \rangle, \langle \mathbf{p}_2, 3.78 \rangle, \langle \mathbf{p}_1, 2.70 \rangle, \langle \mathbf{p}_7, 2.43 \rangle, \langle \mathbf{p}_5, 1.53 \rangle$ |
| Troll(\mathbf{p}_3) | (2.1,1.8) | Titanic(\mathbf{p}_7) | (0.9,0.0) | Alan(\mathbf{u}_3) | (0.3,2.7) | $f(\mathbf{u}_3, \mathbf{q})$ | 2.43 | $\langle \mathbf{p}_4, 7.83 \rangle, \langle \mathbf{p}_2, 7.35 \rangle, \langle \mathbf{p}_3, 5.49 \rangle, \langle \mathbf{p}_6, 3.15 \rangle, \langle \mathbf{p}_1, 2.61 \rangle, \langle \mathbf{q}, 2.43 \rangle, \langle \mathbf{p}_5, 1.71 \rangle, \langle \mathbf{p}_7, 0.27 \rangle$ |
| Smile(\mathbf{p}_4) | (1.8,2.7) | Fight Club(\mathbf{q}) | (2.7,0.6) | Tom(\mathbf{u}_4) | (1.2,0.0) | $f(\mathbf{u}_4, \mathbf{q})$ | 3.24 | $\langle \mathbf{q}, 3.24 \rangle, \langle \mathbf{p}_6, 2.88 \rangle, \langle \mathbf{p}_3, 2.52 \rangle, \langle \mathbf{p}_4, 2.16 \rangle, \langle \mathbf{p}_7, 1.08 \rangle, \langle \mathbf{p}_1, 0.72 \rangle, \langle \mathbf{p}_5, 0.36 \rangle, \langle \mathbf{p}_2, 0.24 \rangle$ |
| | | | | Jerry(\mathbf{u}_5) | (0.9,1.2) | $f(\mathbf{u}_5, \mathbf{q})$ | 3.15 | $\langle \mathbf{p}_4, 4.86 \rangle, \langle \mathbf{p}_3, 4.05 \rangle, \langle \mathbf{p}_2, 3.42 \rangle, \langle \mathbf{p}_6, 3.24 \rangle, \langle \mathbf{q}, 3.15 \rangle, \langle \mathbf{p}_1, 1.62 \rangle, \langle \mathbf{p}_5, 0.99 \rangle, \langle \mathbf{p}_7, 0.81 \rangle$ |

(a) User and product embeddings

(b) User ranks for products

Fig. 1: An example of reverse k -ranks query, we use 2 dimension embeddings for illustration.

of an embedding set contains almost all the embeddings in high dimensionality [33]. In addition, reverse k -ranks is more complex than MIPS because it considers the ranks of a product for all users while MIPS considers a single user. We highlight the two main research challenges of efficient reverse k -ranks query processing on high-dimensional embeddings as follows.

- **C1: High computation cost.** To obtain the k -users having the smallest ranks for the query product, a brute-force solution computes the scores for every user-product pair (called score table) to determine the ranks, which is expensive because there are many user-product pairs (e.g., trillions) and each pair involves two high-dimensional embeddings.
- **C2: Large memory footprints.** To reduce the computation cost, a straightforward solution is to precompute the user-product score table and store it as the index. However, the score table can take up several TBs, which is beyond the memory capacity of most servers.

Our solution QSRP. To overcome the above challenges, we propose the QSRP framework to process high-dimensional reverse k -ranks query. In particular, QSRP devises a *sampling-based index* (Section III) to facilitate pruning at query time. Since the score table can be very large for real datasets (e.g., several TBs), QSRP samples some columns (i.e., the scores at certain ranks for all users) of the score as the *sampling-based index* to meet the memory budget. With the sampling-based index, QSRP determines a rough rank for each user, which allows pruning users that cannot enter the result set. For the remaining users, QSRP computes scores to refine their rough ranks of the query product to derive the final query result.

The ranks to sample for the sampling-based index are crucial for pruning effectiveness and query efficiency. In QSRP, we propose query-aware sampling (Section IV), which chooses the ranks to sample according to a set of training queries that resemble the real queries. In particular, we formulate the computation cost of the training queries as a function of the sampled ranks and design a dynamic programming procedure to solve the optimal sampled ranks that minimize computation cost. To further reduce the costs of score computation and sampling-based index search, we design the regression-based pruning technique (Section V) in QSRP, which is inspired by learned index techniques [34]. Specifically, we compute cheap lower and upper bounds of the user-product score and fit a

simple linear function to map the score bounds to the rank bounds of the query product for each user. These rank bounds allow us to prune users from score computation, and we utilize linear programming to minimize the maximum prediction error of the linear functions for a good pruning effect. Besides the two key designs, QSRP also incorporates techniques to build index for large datasets with limited memory, speed up index construction, and handle dataset updates (Section VI).

We compare our QSRP with two state-of-the-art reverse k -ranks solutions in low dimensionality (i.e., MPA [21] and Grid [22]) and an adaption of high-dimensional MIPS (i.e., reverse RMIPS [35]) (Section VII). We use four large-scale real datasets that contain millions of users and hundreds of thousands of products and experiment with various configurations of the result size k , index size, and embedding dimensionality. The results show that QSRP achieves a shorter query processing time than the baselines in all cases, and the speedup is usually two orders of magnitude and can be up to three orders of magnitude. Micro benchmarks suggest that our optimizations (e.g., query-aware sampling and regression-based pruning) are effective in reducing query time, and QSRP can build index for large datasets with reasonable time (e.g., half an hour).

II. PROBLEM DEFINITION

For a user set \mathcal{U} that contains m users and a product set \mathcal{P} that contains n products, embedding models represent each user $\mathbf{u} \in \mathcal{U}$ and product $\mathbf{p} \in \mathcal{P}$ as a d -dimensional vector. The dimensionality d is usually large for high recommendation accuracy, e.g., hundreds or even thousands [36]. The score of a user for a product is typically calculated as the inner product between their embedding vectors,

$$f(\mathbf{u}, \mathbf{p}) = \mathbf{u}^\top \mathbf{p},$$

with a larger score indicating a higher preference. For a product \mathbf{q} that may or may not belong to the product set \mathcal{P} (e.g., an existing product or a new product processed incrementally by the embedding model [37]), we use $R(\mathbf{u}, \mathbf{q}, \mathcal{P})$ to denote its rank for user \mathbf{u} , which is defined as follows.

Definition 1 (Rank). Given a user \mathbf{u} and a product set \mathcal{P} , the rank of a product \mathbf{q} is $R(\mathbf{u}, \mathbf{q}, \mathcal{P}) = 1 + \sum_{\mathbf{p} \in \mathcal{P}} \mathbb{I}[f(\mathbf{u}, \mathbf{p}) > f(\mathbf{u}, \mathbf{q})]$, where $\mathbb{I}[\cdot]$ is the indicator function.

We add 1 in Definition 1 such that $R(\mathbf{u}, \mathbf{q}, \mathcal{P})$ starts from 1. If $R(\mathbf{u}, \mathbf{q}, \mathcal{P})$ is small, there are only a few products \mathbf{p} with $f(\mathbf{u}, \mathbf{p}) > f(\mathbf{u}, \mathbf{q})$ for user \mathbf{u} , which suggests that product \mathbf{q}

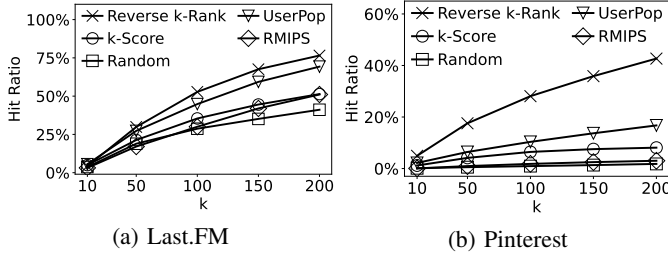


Fig. 2: Number of recommended users (i.e., k) vs. hit ratio, a higher hit ratio means better recommendation accuracy.

is attractive to \mathbf{u} . We also denote $R(\mathbf{u}, \mathbf{q}, \mathcal{P})$ as $R_{\mathbf{u}}$ or $R(\mathbf{u}, \mathbf{q})$ when the meaning is clear from the context.

Definition 2 (Reverse k -ranks query, $R\text{-Ranks}(\mathbf{q}, k)$). Given a positive integer k and a query product $\mathbf{q} \in \mathbb{R}^d$, return a user set \mathcal{U}_R with $\mathcal{U}_R \subseteq \mathcal{U}$ and $|\mathcal{U}_R| = k$ such that for any user pair $\mathbf{u} \in \mathcal{U}_R$ and $\mathbf{u}' \in \mathcal{U} - \mathcal{U}_R$, $R_{\mathbf{u}} \leq R_{\mathbf{u}'}$ holds.

$R\text{-Ranks}(\mathbf{q}, k)$ returns the k users \mathbf{u} that have the smallest ranks $R_{\mathbf{u}}$ for the query product \mathbf{q} in the user set \mathcal{U} . There can be some users $\mathbf{u}' \in \mathcal{U} - \mathcal{U}_R$ outside the result set with $R_{\mathbf{u}'} = R_{\mathbf{u}}$ for a user \mathbf{u} in the result set \mathcal{U}_R . We break ties arbitrarily among these users with the same ranks and allow to include any of them in the result set.

Example 1. Figure 1 shows an example of $R\text{-Ranks}(\mathbf{q}, k)$, where each product is a movie and each user is a movie viewer. The query product is *Fight Club* with $\mathbf{q} = (2.7, 0.6)$ and $k = 2$. In Figure 1(b), the products are sorted in descending order of their scores for each user, and the rank $R_{\mathbf{u}}$ of query product \mathbf{q} is its position in the sorted list (marked with the dash). Thus, we have $R_{\mathbf{u}_1} = 3$, $R_{\mathbf{u}_2} = 2$, $R_{\mathbf{u}_3} = 6$, $R_{\mathbf{u}_4} = 1$, and $R_{\mathbf{u}_5} = 5$. The result of $R\text{-Ranks}(\mathbf{q}, k = 2)$ is $\{\text{Mary}(\mathbf{u}_2), \text{Tom}(\mathbf{u}_4)\}$ as \mathbf{u}_2 and \mathbf{u}_4 have the smallest ranks for \mathbf{q} among all users.

Alternative queries. To show the effectiveness of reverse k -ranks for product-oriented recommendation, we compare its recommendation accuracy with 4 alternative queries on two datasets (see [38] for source) in Figure 2. In particular, we use matrix factorization to obtain the embeddings and measure recommendation accuracy by hit ratio (HR) [39], which is widely used in recommender systems [39], [40], [41] and defined as

$$HR@k = \frac{1}{|\mathcal{Q}|} \sum_{\mathbf{q} \in \mathcal{Q}} \mathbb{I}[|\mathcal{T}_{\mathbf{q}} \cap \mathcal{R}_{\mathbf{q}}|], \quad (1)$$

where \mathcal{Q} is the set of query products; $\mathcal{T}_{\mathbf{q}}$ and $\mathcal{R}_{\mathbf{q}}$ are the actual interacted user set and algorithm recommended user set for product \mathbf{q} , respectively; k is the size of $\mathcal{R}_{\mathbf{q}}$. Larger hit ratio means that $\mathcal{T}_{\mathbf{q}}$ and $\mathcal{R}_{\mathbf{q}}$ overlap for more products. Among the alternative queries, *Random* randomly samples k users and serves as a naive baseline; *UserPop* [40] selects the k users that have the largest number of product interactions; *k-Score* [21] chooses the k users that have the largest scores for the query product. *Reverse MIPS* [42] chooses the users whose top- k' products include the target product, and to align with the other queries, we tune k' such that the average number of recommended users for a product is k .

Figure 2 shows that reverse k -ranks yields higher rec-

| Rank | 1 | 2 | 3 | 4 | 5 | 6 | 7 | $f(\mathbf{u}, \mathbf{q})$ | Rank | 1 | 4 | 7 |
|----------------|------|------|------|------|------|------|------|-----------------------------|----------------|------|------|------|
| \mathbf{u}_1 | 5.13 | 4.77 | 4.41 | 2.73 | 1.71 | 1.35 | 0.99 | 4.59 | \mathbf{u}_1 | 5.13 | 2.73 | 0.99 |
| \mathbf{u}_2 | 8.10 | 7.83 | 7.56 | 3.78 | 2.70 | 2.43 | 1.53 | 8.01 | \mathbf{u}_2 | 8.10 | 3.78 | 1.53 |
| \mathbf{u}_3 | 7.83 | 7.35 | 5.49 | 3.15 | 2.61 | 1.71 | 0.27 | 2.43 | \mathbf{u}_3 | 7.83 | 3.15 | 0.27 |
| \mathbf{u}_4 | 2.88 | 2.52 | 2.16 | 1.08 | 0.72 | 0.36 | 0.24 | 3.24 | \mathbf{u}_4 | 2.88 | 1.08 | 0.24 |
| \mathbf{u}_5 | 4.86 | 4.05 | 3.42 | 3.24 | 1.62 | 0.99 | 0.81 | 3.15 | \mathbf{u}_5 | 4.86 | 3.24 | 0.81 |

(a) Score table

(b) Sample index

Fig. 3: An example of US with $\tau = 3$.

ommendation accuracy than these alternative queries. For UserPop, users that interact with many products may not like the particular query product. For k -Score, some users tend to give large scores to all products and thus may not like the query product even if their scores are large. Reverse MIPS suffers from the uncontrollable result size problem, i.e., it recommends a popular product to many users and an ordinary product to only a few or no users. In comparison, reverse k -ranks effectively identifies the most interested users for a product by using their preference ranks for all products and allows explicit control on the result size.

III. SAMPLING-BASED INDEX

As space partitioning indexes (e.g., R-tree and M-tree) are not effective in high dimensionality [22], [32], we precompute the table that contains the scores $f(\mathbf{u}, \mathbf{p})$ of all user-product pairs to facilitate pruning at query time. In this section, we introduce the sampling-based index, and a baseline sampling solution called *uniform sample* (US).

A. Index Structure

We use \mathcal{T} to denote the score table, which contains the scores of all user-product pairs, i.e., $\mathcal{T} = \{f(\mathbf{u}, \mathbf{p}) \mid \forall \mathbf{p} \in \mathcal{P}, \forall \mathbf{u} \in \mathcal{U}\}$. $\mathcal{T}_{\mathbf{u}}$ is a row of \mathcal{T} , which contains the scores of a user $\mathcal{T}_{\mathbf{u}} = \{f(\mathbf{u}, \mathbf{p}) \mid \forall \mathbf{p} \in \mathcal{P}\}$ and is sorted in descending order. As shown in Figure 3(b), the sampling-based index samples a subset of ranks (i.e., columns) of the score table \mathcal{T} as index. We denote the set of sampled ranks as $\mathcal{S} = \{s_1, s_2, \dots, s_{\tau}\}$, where $1 \leq s_i \leq n$ for $s_i \in \mathcal{S}$ and τ is the number of sampled ranks. The ranks in \mathcal{S} are arranged in ascending order with $s_i < s_{i+1}$. The index can be expressed as $\mathcal{T}(\mathcal{S}) = \{f(\mathbf{u}, \mathbf{p}) \mid R(\mathbf{u}, \mathbf{p}, \mathcal{P}) \in \mathcal{S}, \forall \mathbf{u} \in \mathcal{U}\}$. We use $\mathcal{T}_{\mathbf{u}}(\mathcal{S})$ to denote the row of user \mathbf{u} in the index. For US, the sampled ranks \mathcal{S} span $[1, n]$ with uniform spacing. For Figure 3(b), $\mathcal{S} = \{1, 4, 7\}$, which means that US keeps the first, fourth, and seventh columns of the score table \mathcal{T} . We determine τ by the memory budget θ_{mem} of the sampling-based index as

$$\tau = \lfloor \theta_{mem} / (m \theta_{scr}) \rfloor,$$

where m is the number of users, θ_{scr} is the size of a score, and $\lfloor \cdot \rfloor$ is the floor function.

B. Query Processing

The sampling-based index allows to derive bounds for the rank of the query product for each user, which can be used to prune users that do not belong to the result set.

Theorem 1 (Rank bound). *For a product \mathbf{q} and rank set \mathcal{S} , define $\text{pos}_{\mathbf{u}}$ as the position of $f(\mathbf{u}, \mathbf{q})$ in the index $\mathcal{T}_{\mathbf{u}}(\mathcal{S})$ of*

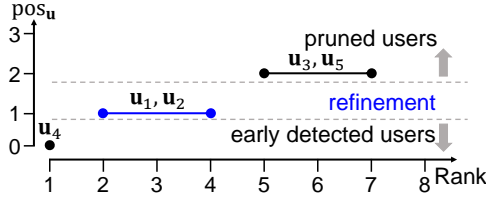


Fig. 4: Rank bound pruning example.

Algorithm 1 Query processing for US (product \mathbf{q} , k , \mathcal{U} , \mathcal{P})

- 1: **for** each user $\mathbf{u} \in \mathcal{U}$ **do**
- 2: $\text{pos}_{\mathbf{u}} \leftarrow \text{BinarySearch}(\mathcal{T}_{\mathbf{u}}(\mathcal{S}), f(\mathbf{u}, \mathbf{q}))$
- 3: $k\text{pos} \leftarrow k^{\text{th}}$ minimum of $\{\text{pos}_{\mathbf{u}} \mid \forall \mathbf{u} \in \mathcal{U}\}$
- 4: Candidate set $\mathcal{U}_C \leftarrow \{\mathbf{u} \in \mathcal{U} \mid \text{pos}_{\mathbf{u}} = k\text{pos}\}$
- 5: Result set $\mathcal{U}_R \leftarrow \{\mathbf{u} \in \mathcal{U} \mid \text{pos}_{\mathbf{u}} < k\text{pos}\}$
- 6: Compute $R_{\mathbf{u}}$ for $\forall \mathbf{u} \in \mathcal{U}_C$ by Definition 1
- 7: $\mathcal{U}_C \leftarrow \{k - |\mathcal{U}_R| \text{ users with the smallest } R_{\mathbf{u}} \text{ in } \mathcal{U}_C\}$
- 8: **return** $\mathcal{U}_R \cup \mathcal{U}_C$

user \mathbf{u} if $\mathcal{T}_{\mathbf{u}}[\text{pos}_{\mathbf{u}}] > f(\mathbf{u}, \mathbf{q}) \geq \mathcal{T}_{\mathbf{u}}[\text{pos}_{\mathbf{u}} + 1]$, where $\mathcal{T}_{\mathbf{u}}[i]$ is the i^{th} entry of $\mathcal{T}_{\mathbf{u}}(\mathcal{S})$. By extending \mathcal{S} with $s_0 = 0$ and $s_{\tau+1} = n+1$, the rank of \mathbf{q} for \mathbf{u} satisfies $s_{\text{pos}_{\mathbf{u}}} + 1 \leq R_{\mathbf{u}} \leq s_{\text{pos}_{\mathbf{u}}+1}$. That is, the lower and upper bounds for $R_{\mathbf{u}}$ are

$$R_{\mathbf{u}}^{\downarrow} = s_{\text{pos}_{\mathbf{u}}} + 1, \quad R_{\mathbf{u}}^{\uparrow} = s_{\text{pos}_{\mathbf{u}}+1}. \quad (2)$$

The proof of Theorem 1 is straightforward and thus omitted.

Example 2. Consider the query product \mathbf{q} and user \mathbf{u}_1 in Figure 3, we have $f(\mathbf{u}_1, \mathbf{q}) = 4.59$, and $\text{pos}_{\mathbf{u}_1} = 1$ (shown with blue bracket). By Theorem 1, we have $R_{\mathbf{u}_1}^{\downarrow} = s_1 + 1 = 2$, and $R_{\mathbf{u}_1}^{\uparrow} = s_2 = 4$. The actual rank of \mathbf{q} for \mathbf{u}_1 is $R_{\mathbf{u}_1} = 3$.

Theorem 2 (Rank bound pruning). *For a product \mathbf{q} , denote the k^{th} minimum among the positions of all users (i.e., $\{\text{pos}_{\mathbf{u}} \mid \forall \mathbf{u} \in \mathcal{U}\}$) as $k\text{pos}$, it suffices to determine the exact ranks for users whose $\text{pos}_{\mathbf{u}}$ is $k\text{pos}$, i.e., $\{\mathbf{u} \mid \text{pos}_{\mathbf{u}} = k\text{pos}\}$.*

Proof. Let \mathbf{u}' be the user(s) whose $\text{pos}_{\mathbf{u}'}$ is the k^{th} minimum among all users. If a user \mathbf{u} has $\text{pos}_{\mathbf{u}} < \text{pos}_{\mathbf{u}'}$, we have $R_{\mathbf{u}}^{\uparrow} < R_{\mathbf{u}'}^{\downarrow}$, and there are at most $k - 1$ such users, thus \mathbf{u} is in the result set. In contrast, if $\text{pos}_{\mathbf{u}} > \text{pos}_{\mathbf{u}'}$, we have $R_{\mathbf{u}}^{\downarrow} > R_{\mathbf{u}'}^{\uparrow}$, and thus \mathbf{u} is not in the result set. Therefore, we do not need to determine the exact ranks for users in $\{\mathbf{u} \mid \text{pos}_{\mathbf{u}} \neq k\text{pos}\}$. \square

Example 3. Figure 4 illustrates rank bound pruning for the R-Ranks(\mathbf{q} , $k=2$) query in Figure 3. The blue part represents the k^{th} minimum position ($k\text{pos}=1$). By Theorem 2, we determine that \mathbf{u}_4 is in the result set, and \mathbf{u}_3 and \mathbf{u}_5 are not in the result set. Thus, we only need to compute the exact rank of the query product for \mathbf{u}_1 and \mathbf{u}_2 , which fall in the blue part.

Algorithm 1 summarizes the query processing procedure of the sampling-based index. Function $\text{BinarySearch}(\mathbf{A}, v)$ returns the position of the rightmost element in the sorted array \mathbf{A} that is larger than the key v . For each user \mathbf{u} , we first compute the score $f(\mathbf{u}, \mathbf{q})$ and $\text{pos}_{\mathbf{u}}$ (lines 1-2). Then, we conduct pruning and initialize the candidate set \mathcal{U}_C and the result set \mathcal{U}_R by Theorem 2 (lines 3-5). This step (lines 1-5) is referred to *pruning*. For each user in the candidate

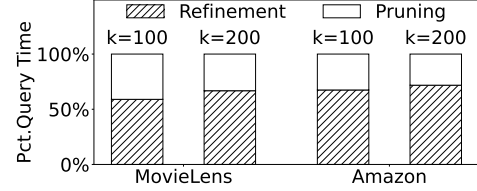
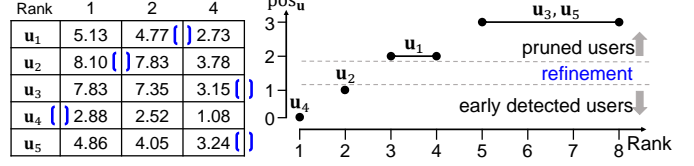


Fig. 5: Query time decomposition for US.



(a) An optimal sample (b) Pruning situation in (a)

Fig. 6: An example of the optimal rank set.

set \mathcal{U}_C , we compute their scores for all products to obtain the exact rank $R_{\mathbf{u}}$ (line 6), which is referred to *refinement*. With the exact ranks, we select the users with the smallest ranks from \mathcal{U}_C and merge them with \mathcal{U}_R .

Performance profiling. Figure 5 decomposes the query time of US into pruning part and refinement part. The result shows that both refinement and pruning take up a significant portion of the query time. As such, we design *query-aware sampling* in Section IV and *regression-based pruning* in Section V to reduce refinement and pruning costs, respectively.

IV. QUERY-AWARE SAMPLING

In this part, we first show that the refinement cost can be reduced by carefully choosing the rank sample set \mathcal{S} with a toy example. Then, we express the refinement cost as a function of \mathcal{S} and propose a dynamic programming procedure to find the optimal \mathcal{S} that minimizes the refinement cost.

A. Motivation

Consider the query R-Ranks(\mathbf{q} , $k=2$) in Figure 1(a) and the memory index in Figure 3(b) that uses $\mathcal{S} = \{1, 4, 7\}$. We can prune \mathbf{u}_3 , \mathbf{u}_4 and \mathbf{u}_5 from refinement but still need to determine the exact ranks of \mathbf{u}_1 and \mathbf{u}_2 for the query product. Define the *refinement cost* as the number of users whose exact ranks need to be computed, $\mathcal{S} = \{1, 4, 7\}$ yields a refinement cost of 2. If the ranks of all users for the query product are known beforehand, we can construct a rank sample set \mathcal{S} to minimize the refinement cost. In fact, for the example query, any rank sample set that contains 2 incurs zero refinement cost because it allows determining that $R_{\mathbf{u}_2}, R_{\mathbf{u}_4} \leq 2$ and $R_{\mathbf{u}_1}, R_{\mathbf{u}_3}, R_{\mathbf{u}_5} \geq 3$, which gives $\{\mathbf{u}_2, \mathbf{u}_4\}$ as the result set without refinement. We show an example of such optimal rank sample sets, i.e., $\mathcal{S} = \{1, 2, 4\}$ in Figure 6.

In practice, we cannot have the queries and hence the ranks of the users for the queries beforehand. However, we can sample a set of historical query products $\mathcal{Q} = \{\mathbf{q}_1, \mathbf{q}_2, \dots, \mathbf{q}_w\}$ (called training queries hereafter) and result set sizes (denoted as k_{idc} to distinguish from the result size k_{qry} of online queries) and assume that the historical queries resemble the

online queries. If we can select a rank set \mathcal{S} to minimize the refinement cost of the training queries, the refinement cost of the online queries will also be small. The problem of rank set selection is challenging for two reasons. (i) Different queries may require to include different ranks in the rank set \mathcal{S} , and thus we need to define a cost function that considers all queries. (ii) The search space is large. With n products and τ ranks to sample, there are $\binom{n}{\tau}$ possible rank sets, and thus we need an efficient search strategy.

B. Cost Function and Dynamic Programming

Similar to Theorem 1, we extend the rank set $\mathcal{S} = \{s_1, s_2, \dots, s_\tau\}$ with two virtual elements, i.e., $s_0 = 0$ and $s_{\tau+1} = n+1$. For a set of training queries \mathcal{Q} , we define the refinement cost $C^\mathcal{Q}(\mathcal{S})$ as the total number of users whose exact ranks (denoted as $R(\mathbf{u}, \mathbf{q})$) needs refinement for the queries in \mathcal{Q} . $C^\mathcal{Q}(\mathcal{S})$ is a function of the rank set \mathcal{S} and can be expressed as

$$C^\mathcal{Q}(\mathcal{S}) = \sum_{\mathbf{q} \in \mathcal{Q}} \sum_{\mathbf{u} \in \mathcal{U}} \sum_{i=1}^{\tau+1} \mathbb{I}[s_{i-1} < R(\mathbf{u}', \mathbf{q}), R(\mathbf{u}, \mathbf{q}) \leq s_i]. \quad (3)$$

Eq. (3) utilizes Theorem 2, which states that for a query \mathbf{q} , we only need to refine the users with $\text{pos}_{\mathbf{u}} = k\text{pos}$. In particular, $R(\mathbf{u}, \mathbf{q})$ is the rank of user \mathbf{u} for product \mathbf{q} , and $R(\mathbf{u}', \mathbf{q})$ is the $k_{\text{idx}}^{\text{th}}$ smallest rank among all users for \mathbf{q} . In the indicator function $\mathbb{I}[\cdot]$, $s_{i-1} < R(\mathbf{u}', \mathbf{q}) \leq s_i$ indicates that $k\text{pos}$ falls in score segment $(s_{i-1}, s_i]$ while $s_{i-1} < R(\mathbf{u}, \mathbf{q}) \leq s_i$ indicates that $\text{pos}_{\mathbf{u}}$ also falls in the score segment. If the two conditions hold, it suggests that $\text{pos}_{\mathbf{u}} = k\text{pos}$ and means that user \mathbf{u} needs to be refined for product \mathbf{q} . Thus, Eq. (3) uses the indicator function $\mathbb{I}[\cdot]$ to determine if a user needs to be refined for a query, and sums over all queries, users, and score segments. Note that the rank of a user $R(\mathbf{u}, \mathbf{q})$ can fall in only one score segment for each query, and thus Eq. (3) will not count one user multiple times for a query. With the cost function $C^\mathcal{Q}(\mathcal{S})$, we define the rank set selection problem.

Definition 3 (Rank set selection problem). Given the user set \mathcal{U} , product set \mathcal{P} , a set of training queries \mathcal{Q} , and the size of the rank set τ , find the rank set that minimizes the refinement cost $C^\mathcal{Q}(\mathcal{S})$. That is,

$$\begin{aligned} & \arg \min_{\mathcal{S}} C^\mathcal{Q}(\mathcal{S}) \\ & \text{s.t. } \mathcal{S} \subseteq \{1, 2, \dots, n\} \text{ and } |\mathcal{S}| \leq \tau. \end{aligned} \quad (4)$$

At first glance, the rank set selection problem in Definition 3 requires enumerating all possible rank sets. In the following, we massage the cost function $C^\mathcal{Q}(\mathcal{S})$ in Eq. (3) into a form that is amendable for efficient optimization.

Lemma 1. Define function $\Upsilon(s_{i-1}, s_i)$ as

$$\Upsilon(s_{i-1}, s_i) = \sum_{\mathbf{q} \in \mathcal{Q}} \sum_{\mathbf{u} \in \mathcal{U}} \mathbb{I}[s_{i-1} < R(\mathbf{u}', \mathbf{q}), R(\mathbf{u}, \mathbf{q}) \leq s_i], \quad (5)$$

then the cost function $C^\mathcal{Q}(\mathcal{S})$ in Eq. (3) can be expressed as

$$C^\mathcal{Q}(\mathcal{S}) = \sum_{i=1}^{\tau+1} \Upsilon(s_{i-1}, s_i). \quad (6)$$

Proof. The proof is straightforward by moving the summation over the score segments to the outermost for Eq. (3). \square

Lemma 1 shows that $C^\mathcal{Q}(\mathcal{S})$ can be expressed as a summation over the refinement cost of each score segment $(s_{i-1}, s_i]$. This suggests that if we modify some score segments (e.g., by inserting a rank in between $(s_{i-1}, s_i]$), the refinement costs of the unaffected segments will not change.

Lemma 2. Denote $\mathcal{S}(i, j)$ as a rank set that satisfies (i) the largest rank in $\mathcal{S}(i, j)$ is i and (ii) the rank set $\mathcal{S}(i, j)$ contains exactly j ranks, and we also extend $\mathcal{S}(i, j)$ with $s_0 = 0$ and $s_{j+1} = n+1$. Define function $\Gamma(t, i)$ as

$$\Gamma(t, i) = \Upsilon(t, i) + \Upsilon(i, n+1) - \Upsilon(t, n+1). \quad (7)$$

For rank set $\mathcal{S}(t, j)$ with $t < i \leq n$ and $2 \leq j \leq \tau$, we have

$$C^\mathcal{Q}(\mathcal{S}(t, j-1) \cup \{i\}) = \Gamma(t, i) + C^\mathcal{Q}(\mathcal{S}(t, j-1)). \quad (8)$$

Proof.

$$\begin{aligned} \Gamma(t, i) + C^\mathcal{Q}(\mathcal{S}(t, j-1)) &= \Gamma(t, i) + \sum_{i=1}^j \Upsilon(s_{i-1}, s_i) \\ &= \Gamma(t, i) + \Upsilon(t, n+1) + \sum_{i=1}^{j-1} \Upsilon(s_{i-1}, s_i) \\ &= \Upsilon(t, i) + \Upsilon(i, n+1) + \sum_{i=1}^{j-1} \Upsilon(s_{i-1}, s_i) \\ &= C^\mathcal{Q}(\mathcal{S}(t, j-1) \cup \{i\}). \end{aligned} \quad \square$$

We can construct rank set $\mathcal{S}(i, j)$ by inserting rank i at the end of rank set $\mathcal{S}(t, j-1)$. Lemma 2 shows that $C^\mathcal{Q}(\mathcal{S}(i, j))$ can be expressed by adding $C^\mathcal{Q}(\mathcal{S}(t, j-1))$ and $\Gamma(t, i)$, and $\Gamma(t, i)$ is not affected by $C^\mathcal{Q}(\mathcal{S}(t, j-1))$.

Theorem 3. Given parameter i and j , denote $\mathcal{S}^*(i, j)$ as the rank set that has the minimum refinement cost for the query set \mathcal{Q} among all possible $\mathcal{S}(i, j)$. For $1 \leq i \leq n$ and $1 \leq j \leq \tau$, we have

$$C^\mathcal{Q}(\mathcal{S}^*(i, j)) = \begin{cases} \Upsilon(0, i) + \Upsilon(i, n+1) & j=1 \\ \min_{1 \leq t < i} \{C^\mathcal{Q}(\mathcal{S}^*(t, j-1)) + \Gamma(t, i)\} & \text{otherwise} \end{cases} \quad (9)$$

Proof. When $j = 1$, the rank set can only be $\mathcal{S}^*(i, 1) = \{i\}$, which divides the ranks into two segments $(0, i]$ and $(i, n+1]$. This gives the upper half of Eq. (9).

When $j \neq 1$, by Lemma 2, we have $C^\mathcal{Q}(\mathcal{S}(i, j)) = C^\mathcal{Q}(\mathcal{S}(t, j-1)) + \Gamma(t, i)$. For a fixed t and i , $\Gamma(t, i)$ is fixed, and thus we should choose $\mathcal{S}^*(t, j-1)$ among all possible $\mathcal{S}(t, j-1)$ to minimize $C^\mathcal{Q}(\mathcal{S}(t, j-1))$; then we should minimize $C^\mathcal{Q}(\mathcal{S}^*(t, j-1)) + \Gamma(t, i)$ over all possible choices of t , which gives the lower half of Eq. (9). \square

With $\mathcal{S}^*(i, \tau)$ for all possible i , the solution to the rank set selection problem in Definition 3 can be expressed as

$$\mathcal{S}^* = \arg \min_{\mathcal{S}^*(i, \tau), i \in [\tau, n]} C^\mathcal{Q}(\mathcal{S}^*(i, \tau)). \quad (10)$$

We constrain $i \in [\tau, n]$ as the rank set \mathcal{S} contains τ ranks and thus the maximum rank $i \geq \tau$. If $i < \tau$ for some optimal rank set \mathcal{S}^* (which also means that $|\mathcal{S}^*| < \tau$), we can add ranks to \mathcal{S}^* without increasing $C^\mathcal{Q}(\mathcal{S}^*)$.¹

C. Overall Procedure

Algorithm 2 shows the procedure of finding the optimal rank set using Eq. (9). In particular, we use two arrays to record the states: $\text{cost}(i, j)$ for the refinement cost of the optimal rank set $\mathcal{S}^*(i, j)$ and $\text{rank}(i, j)$ for the second largest rank in

¹It is straightforward to show that if $\mathcal{S}' \subset \mathcal{S}$, then $C^\mathcal{Q}(\mathcal{S}') \geq C^\mathcal{Q}(\mathcal{S})$.

Algorithm 2 Query-aware sampling (rank set size τ)

```

1:  $cost \leftarrow$  2D array with size  $(n+1) \times \tau$ 
2:  $rank \leftarrow$  2D array with size  $(n+1) \times \tau$ 
3: Precompute and cache  $\Upsilon(i, j), \forall 1 \leq i < j \leq n$ 
4: for  $i \in [1, n]$  do
5:    $cost(i, 1) \leftarrow \Upsilon(0, i) + \Upsilon(i, n+1), rank(i, 1) \leftarrow i$ 
6: for  $j$  from 2 to  $\tau$  do
7:   for  $i \in [1, n]$  do
8:      $rank(i, j) \leftarrow \arg \min_{t \in [1, i]} \{cost(t, j-1) + \Gamma(t, i)\}$ 
9:      $cost(i, j) \leftarrow cost(rank(i, j), j-1) + \Gamma(rank(i, j), i)$ 
10: Result rank set  $\mathcal{S} \leftarrow \emptyset, i^* \leftarrow \arg \min_{i \in [\tau, n]} cost(i, \tau)$ 
11: for  $j$  from  $\tau$  to 1 do
12:    $\mathcal{S} \leftarrow \mathcal{S} \cup rank(i^*, j), i^* \leftarrow rank(i^*, j)$ 
13: return  $\mathcal{S}$ 

```

$\mathcal{S}^*(i, j)$. We precompute and store $\Upsilon(i, j)$ for all i, j at line 3, which is used to compute $\Gamma(\cdot, \cdot)$ (see Eq. (7)). Lines 4-5 initialize $cost(i, j)$ for $j=1$, and lines 6-9 compute $cost(i, j)$ and $rank(i, j)$ from $j=2$ to $j=\tau$. The optimal refinement cost is given by the minimum $cost(i, \tau)$ over all i , and the corresponding rank set is obtained by recursively traversing the $rank$ array (lines 11-12).

Algorithm 2 consists of two main parts: (i) compute $\Upsilon(\cdot, \cdot)$ (line 3) and (ii) nested loop (lines 6-9). Computing a single $\Upsilon(\cdot, \cdot)$ takes $O(wm \log n)$ time, and thus enumerating all possible i and j takes $O(wmn^2 \log n)$ time. The nested loop takes $O(\tau n^2)$ time because both i and t range in $[1, n]$. Both parts involve n^2 but the number of products n is usually large.

We introduce two optimizations to speedup Algorithm 2. First, we precompute $R(\mathbf{u}, \mathbf{q})$ for all user-query pairs and look up the required entries when computing $\Upsilon(\cdot, \cdot)$. In particular, we store a 2D array with dimension $w \times (n+1)$ named CT with $CT_{\mathbf{q}}[i] = \sum_{\mathbf{u} \in \mathcal{U}} \mathbb{I}[R(\mathbf{u}, \mathbf{q}) \in [1, i]]$. To compute $\Upsilon(s_1, s_2)$, we find all the queries with $\mathcal{Q}' = \{\mathbf{q} \mid R(\mathbf{u}', \mathbf{q}) \in [s_1, s_2]\}$, and thus $\Upsilon(s_1, s_2) = \sum_{\mathbf{q} \in \mathcal{Q}'} (CT_{\mathbf{q}}[s_2] - CT_{\mathbf{q}}[s_1 - 1])$. By summing over the queries, the computation of $\Upsilon(\cdot, \cdot)$ reduces the $O(wmn^2 \log n)$ time of part (i) to $O(wn^2)$ time.

Second, we use a simple heuristic to reduce the number of candidate ranks for the nested loop. In particular, Eq. (3) shows that user refinement is invoked when $R(\mathbf{u}, \mathbf{q})$ falls in the same score segment $(s_{i-1}, s_i]$ as $R(\mathbf{u}', \mathbf{q})$, which is the $k_{\text{idx}}^{\text{th}}$ smallest rank for \mathbf{q} . Intuitively, the optimal rank set should be around $R(\mathbf{u}', \mathbf{q})$ because this shortens the length of the score segments surrounding $R(\mathbf{u}', \mathbf{q})$ and thus reduces the number of users whose $R(\mathbf{u}, \mathbf{q})$ falls in the same score segment as $R(\mathbf{u}', \mathbf{q})$. Thus, we change the rank candidate set from $[1, n]$ in line 7 to $\Omega = \bigcup_{\mathbf{q} \in \mathcal{Q}} R(\mathbf{u}', \mathbf{q}) \cup \{R(\mathbf{u}', \mathbf{q}) + 1\}$. This optimization reduces the number of $\Upsilon(\cdot, \cdot)$ to compute from $O(n^2)$ to $O(w^2)$ and the number of possible values for i and t from n to w . With the two optimizations, the overall time complexity becomes $O(w^3)$, which is much smaller because $w \ll m$ and $w \ll n$ usually hold.

Parameter selection. Query-aware sampling requires to specify the historical query workloads, which involves (i) the query

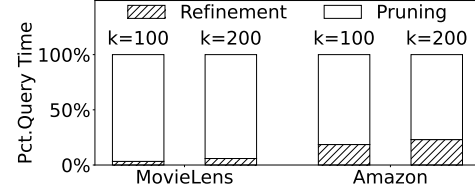


Fig. 7: Query time decomposition for QS.

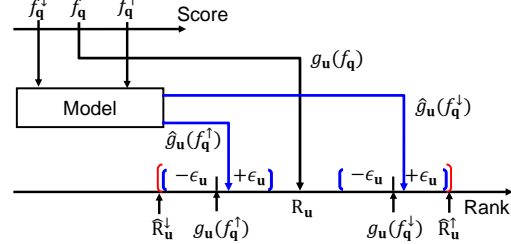


Fig. 8: The schematic diagram of regression-based pruning.

vector \mathbf{q} , (ii) the number of users k_{idx} required by each query for index building, and (iii) the number of training queries w . We randomly sample the product embeddings as \mathbf{q} . Setting k_{idx} is subtle because the number of users required by the actual queries (denote as k_{qry}) may be different from k_{idx} . We suggest configuring k_{idx} as the maximum possible k_{qry} and explain the reasons in Section VII-C. For the number of training queries w , we empirically observe that query time first decreases with w but stabilizes after w reaches several thousand (see Figure 15(a)) and thus suggest using $w=5,000$.

Performance profiling. Query-aware sampling (QS) achieves significantly shorter query time than uniform sample (US) by reducing the refinement cost. For instance, with $k=200$ on the Amazon dataset, US takes 272ms while QS takes only 117ms for an average query. This is because US computes 55 million scores for each query while QS computes only 0.4 million scores. Figure 7 decomposes the query time of QS. The results show that the pruning part now dominates the query time. Thus, we introduce regression-based pruning to reduce the pruning time in the next section.

V. REGRESSION-BASED PRUNING

Recall that the pruning part of query processing computes the score $f(\mathbf{u}, \mathbf{q})$ and searches the sampling-based index for each user (i.e., lines 1-2 in Algorithm 1). We optimize the pruning part because the score computation and index searching need to be invoked many times. Figure 8 shows how *regression-based pruning* (RP) accelerates the pruning part. In particular, RP first computes cheap lower and upper bounds for the score $f(\mathbf{u}, \mathbf{q})$, denoted as $f^{\downarrow}(\mathbf{u}, \mathbf{q})$ and $f^{\uparrow}(\mathbf{u}, \mathbf{q})$ (or f_q^{\downarrow} and f_q^{\uparrow} for conciseness). Inspired by the learned index [34], RP learns a function $\hat{g}_{\mathbf{u}} : \mathbb{R} \rightarrow \mathbb{R}$ that predicts the rank of a score for each user \mathbf{u} . As the learned function $\hat{g}_{\mathbf{u}}$ incurs errors, we relax the predicted ranks by the maximum prediction error $\epsilon_{\mathbf{u}}$. This yields rank bounds for each user (marked by the red bracket in Figure 8) and thus allows fast pruning.

Score bounds. We use the Cauchy–Schwarz inequality to compute bounds for f_q . In particular, we have

$$\begin{cases} f_q \leq f_q^\uparrow = f(\mathbf{u}^h, \mathbf{q}^h) + \|\mathbf{u}^{\mathcal{C}_h}\| \|\mathbf{q}^{\mathcal{C}_h}\| \\ f_q \geq f_q^\downarrow = f(\mathbf{u}^h, \mathbf{q}^h) - \|\mathbf{u}^{\mathcal{C}_h}\| \|\mathbf{q}^{\mathcal{C}_h}\| \end{cases}, \quad (11)$$

where \mathbf{u}^h is the first h dimensions of vector \mathbf{u} and $\|\mathbf{u}^{\mathcal{C}_h}\|$ is the l_2 -norm of the last $d-h$ dimensions of \mathbf{u} . Note that $\|\mathbf{q}^{\mathcal{C}_h}\|$ is computed only once for each query, and $\|\mathbf{u}^{\mathcal{C}_h}\|$ is precomputed during index building. We adopt the SVD technique in [43] to project the embeddings such that the norm of the last dimensions is small. Compared with computing the exact score with d dimensions, Eq.(11) uses only h dimensions.

A. Regression-based Rank Bound

Based on the score bounds f_q^\uparrow and f_q^\downarrow , we obtain rank bounds for each user \mathbf{u} without conducting the binary search on the sampling-based index. In particular, we regard the index of user \mathbf{u} (i.e., $\mathcal{T}_u(\mathcal{S})$) as a function $g_u: \mathbb{R} \rightarrow \{1, 2, \dots, \tau+1\}$ that maps score (i.e., f_q) to position (i.e., pos_u). Regression-based pruning fits a function \hat{g}_u to approximate g_u . We define the approximation error ϵ_u as

$$\epsilon_u = \max_{\mathcal{T}_u[1] \leq \text{scr} \leq \mathcal{T}_u[\tau]} |g_u(\text{scr}) - \hat{g}_u(\text{scr})|, \quad (12)$$

where $\mathcal{T}_u[1]$ and $\mathcal{T}_u[\tau]$ are the largest and smallest scores in the index of user \mathbf{u} , respectively. With the approximation error ϵ_u , we can obtain the bounds for pos_u as

$$\text{pos}_u^\downarrow = \lfloor \hat{g}_u(f_q^\uparrow) - \epsilon_u \rfloor, \quad \text{pos}_u^\uparrow = \lceil \hat{g}_u(f_q^\downarrow) + \epsilon_u \rceil. \quad (13)$$

The bounds for pos_u may overflow, e.g., $\text{pos}_u^\downarrow < 1$ or $\text{pos}_u^\uparrow > \tau+1$, we set pos_u^\downarrow as 1 and pos_u^\uparrow as $\tau+1$ in these cases. With the bounds for pos_u , rank bounds (i.e., \hat{R}_u^\downarrow and \hat{R}_u^\uparrow) for user \mathbf{u} are derived using Theorem 1 as

$$\hat{R}_u^\downarrow = s_{\text{pos}_u^\downarrow} + 1, \quad \hat{R}_u^\uparrow = s_{\text{pos}_u^\uparrow + 1}, \quad (14)$$

where s_i is the i^{th} rank in the rank set \mathcal{S} . We use Eq.(14) for user pruning following the logic of Algorithm 1. In particular, we first compute the rank lower bound and upper bound for each user as \hat{R}_u^\downarrow and \hat{R}_u^\uparrow , and then find the k^{th} smallest lower and upper bounds as R' and R'' . Users with $\hat{R}_u^\uparrow < R'$ are in the result set and users with $\hat{R}_u^\downarrow \geq R''$ cannot be in the result set (i.e., pruned), and the remaining users are in the candidate set and checked with the sampling-based index.

B. Model Training

We fit $\hat{g}_u(\cdot)$ as a linear function, i.e.,

$$\hat{g}_u(f_q) = af_q + b, \quad (15)$$

where a and b are the slope and intercept of the linear function, f_q is the score, and $\hat{g}_u(f_q)$ is the predicted pos_u . To obtain a and b , we need to define a loss function for model training. As ϵ_u quantifies how well $\hat{g}_u(f_q)$ approximates $g_u(f_q)$, we use ϵ_u as the loss, which is expressed as follows.

Theorem 4. *Given the rank set \mathcal{S} , the maximum prediction error ϵ_u in the interval of $[\mathcal{T}_u[\tau], \mathcal{T}_u[1]]$ can be computed as*

$$\epsilon_u = \max_{i \in \{1, 2, \dots, \tau-1\}} \{ |i+1 - \hat{g}_u(\mathcal{T}_u[i+1])|, |i+1 - \hat{g}_u(\mathcal{T}_u[i])| \},$$

where $\mathcal{T}_u[i]$ is the i^{th} score in the sampling-based index $\mathcal{T}_u(\mathcal{S})$.

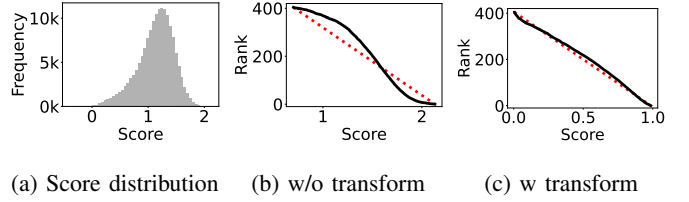


Fig. 9: Relation between score and rank for a user from the Yelp dataset without and with distribution transformation. The red dotted line plots the learned linear function.

Proof. By the definition of pos_u , $g_u(f_q)$ can be expressed as

$$g_u(f_q) = \begin{cases} 1 & \mathcal{T}_u[1] \leq f_q \\ i+1 & \mathcal{T}_u[i+1] \leq f_q < \mathcal{T}_u[i], \forall i \in \{1, 2, \dots, \tau-1\} \\ \tau+1 & \mathcal{T}_u[\tau] \geq f_q \end{cases}$$

That is, $g_u(f_q)$ is constant for every score interval $[\mathcal{T}_u[i+1], \mathcal{T}_u[i]]$. As $\hat{g}_u(f_q)$ is a linear function and thus monotonic, function $\hat{g}_u(\cdot) - g_u(\cdot)$ is also monotonic in each score interval. Therefore, the maximum error of an interval only appears at the boundary points, i.e., $f_q = \mathcal{T}_u[i]$ or $f_q = \mathcal{T}_u[i+1]$. Thus we have $\epsilon_u \geq \max\{|i+1 - \hat{g}_u(\mathcal{T}_u[i+1])|, |i+1 - \hat{g}_u(\mathcal{T}_u[i])|\}$ for each interval. Traversing all the score intervals gives ϵ_u . \square

We minimize ϵ_u for tight rank bounds, i.e.,

$$\underset{a, b}{\text{argmin}} \epsilon_u.$$

Given the expression of ϵ_u , the optimization problem can be transformed into the following linear programming problem,

Minimize ϵ_u

$$\text{s.t.} \quad \begin{cases} a\mathcal{T}_u[i+1] + b + \epsilon_u \geq i+1 \\ a\mathcal{T}_u[i+1] + b - \epsilon_u \geq i+1 \\ a\mathcal{T}_u[i] + b + \epsilon_u \geq i+1 \\ a\mathcal{T}_u[i] + b - \epsilon_u \geq i+1 \end{cases} \quad \forall i \in \{1, 2, \dots, \tau-1\}. \quad (16)$$

There are 3 variables and $4\tau-4$ constraints in Eq. (16), and thus Eq. (16) can be solved with $O(\tau)$ complexity using the Seidel's linear programming algorithm [44].

Distribution transformation. We find that the approximation error ϵ_u is large for some datasets. This is because the score-rank relation does not fit well with a linear function. For instance, for the user in Figure 9(a), the score distribution is bell-shaped and resembles a normal distribution. Figure 9(b) shows that the rank changes slowly when the score is large but very quickly for moderate scores, which is difficult to fit with a linear function that changes at a constant rate.

To tackle this problem, we apply a distribution transformation before fitting the linear function for each user. Denote the cumulative distribution function (CDF) of the scores for a user as $L_u(\cdot)$, the linear function to fit becomes

$$\hat{g}_u(f_q) = aL_u(f_q) + b. \quad (17)$$

This is because $L_u(f_q)$ is the percentage of scores that are smaller than f_q , and the rank of f_q among the n products is approximately $n - nL_u(f_q)$, which is linear function w.r.t. $L_u(f_q)$. We observe that the score distribution resembles the normal distribution for some datasets. In this case, to obtain the CDF, we estimate the normal distribution parameters (i.e., mean and variance) for each user using the maximum likelihood method. Figure 9(c) shows the relation between

Algorithm 3 QSRP index building (\mathcal{U} , \mathcal{P} , query set \mathcal{Q})

```

1: for user  $\mathbf{u} \in \mathcal{U}$  do  $\triangleright$  Comp. the rank of sample queries
2:   Compute  $\mathcal{T}_{\mathbf{u}} \leftarrow \{f(\mathbf{u}, \mathbf{p}) \mid \mathbf{p} \in \mathcal{P}\}$ 
3:   for sample query  $\mathbf{q} \in \mathcal{Q}$  do
4:      $R(\mathbf{u}, \mathbf{q}) \leftarrow \text{BinarySearch}(\mathcal{T}_{\mathbf{u}}, f(\mathbf{u}, \mathbf{q}))$ 
5:  $\mathcal{S} \leftarrow$  Find the optimal rank sample set  $\triangleright$  Algorithm 2
6: for user  $\mathbf{u} \in \mathcal{U}$  do
7:   Compute  $\mathcal{T}_{\mathbf{u}} \leftarrow \{f(\mathbf{u}, \mathbf{p}) \mid \mathbf{p} \in \mathcal{P}\}$ 
8:   Store  $\mathcal{T}_{\mathbf{u}}(\mathcal{S})$  as the sampling-based index of  $\mathbf{u}$ 
9:   Fit function  $\hat{g}_{\mathbf{u}}(\cdot)$  using  $\mathcal{T}_{\mathbf{u}}(\mathcal{S})$   $\triangleright$  Regression index

```

$L_{\mathbf{u}}(f_{\mathbf{q}})$ and rank after the transformation, which is similar to a linear function and easier to fit than Figure 9(b).

Prediction objective. Instead of predicting the position of a score in the sampling-based index (i.e., learn $g_{\mathbf{u}} : \mathbb{R} \rightarrow \{1, 2, \dots, \tau+1\}$), one may predict the rank of the score among all products (i.e., learn $g'_{\mathbf{u}} : \mathbb{R} \rightarrow \{1, 2, \dots, n+1\}$). We experimented with this choice and observed that it yields longer query time. This is because the maximum error becomes larger with more data points to fit for each user, and thus the pruning effect of regression deteriorates.

VI. DISCUSSIONS

Index construction. Algorithm 3 summarizes the index construction procedure of QSRP, which consists of 4 main steps, i.e., compute the rank of the sample queries for each user to facilitate query-aware sampling (lines 1-4, which will be used to compute $\Upsilon(i, j)$), query-aware sampling to determine the optimal rank set \mathcal{S} (line 5), sample the scores according to \mathcal{S} for each user (line 8), and build the regression index (line 9). The score table is usually large (e.g., several TBs) and cannot be stored in memory. Fortunately, our index construction does not require to materialize the entire score table. As shown in lines 1 and 6 of Algorithm 3, we compute the scores at user granularity and discard the scores after use. This requires computing the scores twice but we observe that it is faster than storing the scores to disk and then loading them for use. We adopt GPU to compute the scores efficiently and parallelize among different users as they are independent.

Dataset updates. QSRP allows to delete and insert both users and products. Deleting a user is conducted by removing its regression function and the corresponding row of the sampling-based index; to insert a user, we compute its scores with all products with $O(nd)$ time and store the scores for solving the rank of the user for the query product. To insert or delete a product, we compute its scores for all users and store these scores to compute rank offset. In particular, for each user, we first compute how many inserted and deleted products have larger scores than $f(\mathbf{u}, \mathbf{q})$ to derive a rank offset and then calibrate the rank bounds obtained by the regression index and sampling-based index with the rank offset. We recommend rebuilding the index after accumulating some updates because updates require to store scores and degrade query efficiency.

Query distribution shift. The online queries can become different from the queries used for query-aware sampling, which

TABLE I: Statistics of the experiment datasets

| Name | # User | # Product | Score table size |
|-------------|-----------|-----------|------------------|
| Yahoo!Music | 1,823,179 | 135,736 | 0.9TB |
| Yelp | 2,189,457 | 159,585 | 1.2TB |
| MovieLens | 2,197,225 | 272,038 | 2.2TB |
| Amazon | 2,511,610 | 409,243 | 3.7TB |

may degrade query performance. In particular, a historical query takes the form (\mathbf{q}, k) , where \mathbf{q} is the product embedding and k is the number of required users. For the change of \mathbf{q} , we randomly sample 5,000 query embeddings and use the same k for index building, and we observe that the average query time differs by less than 5% over 10 trials for both Yelp and Amazon. Changing k has more significant influence and the effect depends on the relation between the k of online queries (i.e., k_{qry}) and the k of historical queries (i.e., k_{idx}); as we will show in Figure 15(b), query time becomes much longer when $k_{\text{qry}} > k_{\text{idx}}$, but the degradation is small when $k_{\text{qry}} \leq k_{\text{idx}}$. Thus, considering query distribution shift, we only need to rebuild the index when $k_{\text{qry}} > k_{\text{idx}}$, and note that (i) the case $k_{\text{qry}} > k_{\text{idx}}$ does not happen frequently because k_{idx} is set as the maximum possible k_{qry} , and (ii) our index building cost is reasonable (e.g., around 25 minutes of the Amazon dataset with a score table of 3.7TB).

Efficient refinement. To refine $R(\mathbf{u}, \mathbf{q})$ of user \mathbf{u} , Algorithm 1 computes the user's exact scores for all products. To improve efficiency, we filter the candidates by Cauchy-Schwarz inequality, where only products with $\{\mathbf{p} \mid f_{\mathbf{q}} \leq \|\mathbf{u}\| \|\mathbf{p}\|\}$ needs further refinement. Then we compute the cheap score bounds in Eq.(11) of $f(\mathbf{u}, \mathbf{p})$ for each product \mathbf{p} , and the exact score is only computed when $f(\mathbf{u}, \mathbf{q})$ falls in between the lower and upper bounds.

Other score functions. QSRP generalizes to score functions other than inner product (e.g., Euclidean distance [45] and hyperbolic distance [46]). The sampling-based index, query-aware sampling, and rank regression techniques directly apply, and the only modification is to change the score bounds in Eq.(11) to suit the target score function. For instance, if the score function is Euclidean distance, i.e., $f_{\mathbf{q}}^{\text{Euc}} = \|\mathbf{u} - \mathbf{q}\|$, the score bounds can be expressed as

$$\begin{cases} f_{\mathbf{q}}^{\text{Euc}} \leq f_{\mathbf{q}}^{\text{Euc}, \uparrow} = \|\mathbf{u}^h - \mathbf{q}^h\| + \|\mathbf{u}^{\text{G}_h}\| + \|\mathbf{q}^{\text{G}_h}\| \\ f_{\mathbf{q}}^{\text{Euc}} \geq f_{\mathbf{q}}^{\text{Euc}, \downarrow} = \|\mathbf{u}^h - \mathbf{q}^h\| - \|\mathbf{u}^{\text{G}_h}\| - \|\mathbf{q}^{\text{G}_h}\| \end{cases} \quad (18)$$

Parallelization. QSRP is easy to parallelize in retrieval because both rank bound computation and rank refinement are independent for different users.

VII. EXPERIMENTAL EVALUATION

We introduce experiment settings in Section VII-A, compare QSRP with state-of-the-art baselines in Section VII-B, and evaluate the designs of QSRP in Section VII-C.

A. Experiment Settings

Datasets. We use the 4 datasets (see [38] for source) in Table I, which are widely used in recommendation [10], [40]. In particular, *Yahoo!Music* [47], *Yelp* [48], *MovieLens* [49], and *Amazon* [50] record user-product interactions for music,

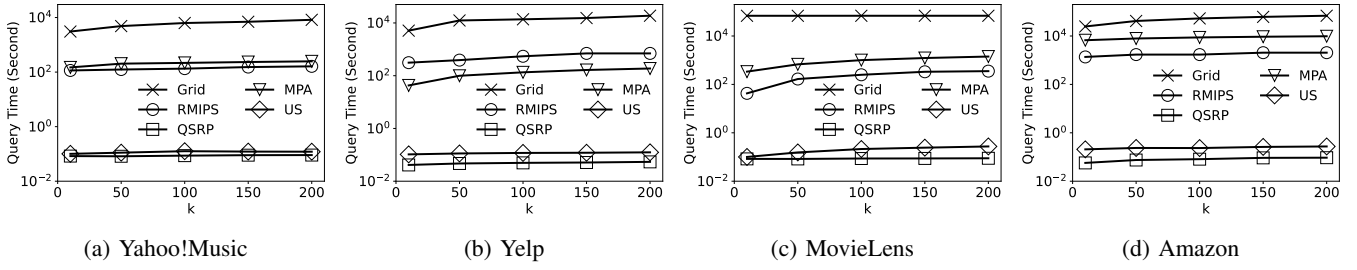


Fig. 10: Average query processing time under different k .

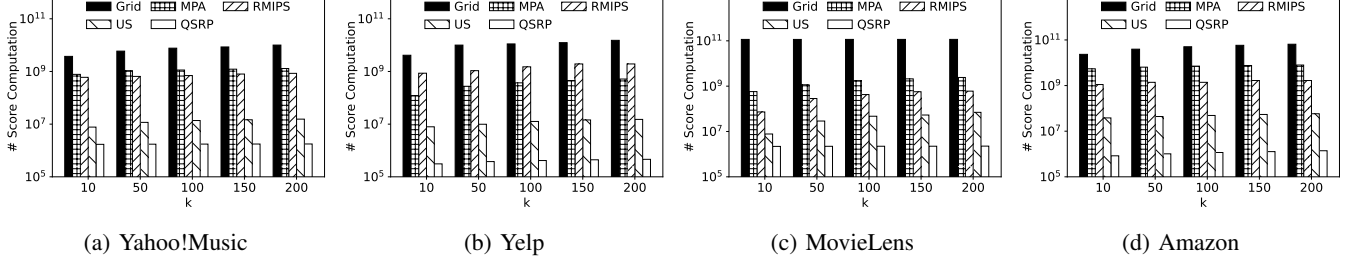


Fig. 11: The average number of score computations conducted by each query.

restaurant, movie, and e-commerce recommendation, respectively. Following [43], we use LIBPMF [51] to generate the user and product embedding vectors via matrix factorization. The dimensionality of the embeddings is set as 150 by default.

Baselines. We compare QSRP² with 4 baselines, i.e., *uniform sample* (US), *marked pruning approach* (MPA) [21], *grid index* (Grid) [22] and *reverse MIPS* (RMIPS) [35]. US is our initial solution in Section III, which uniformly samples the score table as the index. MPA builds an R-tree on the product set to compute the rank of the query product for each user. Grid improves linear scan with two techniques, (i) the concept of dominance to filter products from score computation, and (ii) a uniform quantization for each dimension of the embedding vectors to speedup inner product computation. RMIPS adapts the state-of-the-art reverse MIPS solution Simpfer++ [35]. As Simpfer++ cannot control the number of returned users, we start with $k' = 1$ and double k' each time until the result set contains at least k users. For a fair comparison, we ensure that all methods use the same index size. Among the baselines, MPA and Grid are state-of-the-art reverse k -ranks solutions for low-dimensional embeddings while RMIPS targets high-dimensional embeddings.

Evaluation methodology. We use *query processing time* as the main performance metric. In particular, for each dataset, we randomly sample 10,000 products and choose k from $\{10, 50, 100, 150, 200\}$ to generate the queries and measure the average query processing time for each k . As RMIPS, MPA, and Grid can be very slow, we terminate an experiment after 12 hours and calculate query processing time according to the finished queries. By default, the memory index size is set as 64GB for all methods. QSRP uses 5,000 training queries and $k_{idx} = 200$ in query-aware sampling. The products

of the training queries are randomly sampled from the product embeddings. The training queries are excluded from the performance evaluation for a fair comparison.

QSRP uses the GPU to compute the score table and all threads of the CPU for the other index building part. As micro performance metrics, we report the average *number of refined users* and *computation cost* for each query. The refined users are those that cannot be pruned by index checking in US and QSRP, and thus require score computation for rank refinement. Computation cost is measured by the number of exact score computations. For methods that use score bounds, computation cost is the cost of exact score plus the score bound, which is measured by profiling the costs of the two operations.

Platform. The experiment machine is equipped with a Intel(R) Xeon(R) Gold 5318Y@2.10GHz CPU with 96 threads, a NVIDIA A10 GPU, and 512GB main memory with Linux version 4.15.0. All experiment codes are written in C++17 and compiled with the -O3 optimization flag.

B. Main Results

Figure 10 and Table II compare the average query time of QSRP with US, RMIPS, MPA, and Grid for different k . We report the average computation costs in Figure 11 to help understand the query time performance. Two observations can be made from these results.

First, both US and QSRP are significantly faster than MPA, RMIPS, and Grid. In particular, the speedup of QSRP over MPA, RMIPS, and Grid is more than 1,000 on the Amazon dataset and more than 100 on the other datasets. Figure 11 shows that this is because MPA, RMIPS, and Grid conduct many more score computations than US and QSRP. The poor performance of Grid is attributed to two reasons. First, its dominance-based pruning does not work for the learned embeddings as they contain negative elements. Second, its quantization-based score computation has limited speedup

²QSRP is open source at <https://github.com/DBGROUP-SUSTech/reverse-k-ranks>.

TABLE II: Query time (in seconds) for US and QSRP.

| k | Yahoo!Music | | Yelp | | MovieLens | | Amazon | |
|-----|-------------|------|------|------|-----------|------|--------|------|
| | US | QSRP | US | QSRP | US | QSRP | US | QSRP |
| 10 | 0.10 | 0.08 | 0.10 | 0.04 | 0.10 | 0.08 | 0.20 | 0.05 |
| 50 | 0.11 | 0.08 | 0.11 | 0.04 | 0.15 | 0.08 | 0.23 | 0.07 |
| 100 | 0.12 | 0.08 | 0.11 | 0.04 | 0.21 | 0.08 | 0.23 | 0.08 |
| 150 | 0.12 | 0.09 | 0.11 | 0.05 | 0.24 | 0.08 | 0.26 | 0.09 |
| 200 | 0.12 | 0.09 | 0.12 | 0.05 | 0.27 | 0.08 | 0.27 | 0.09 |

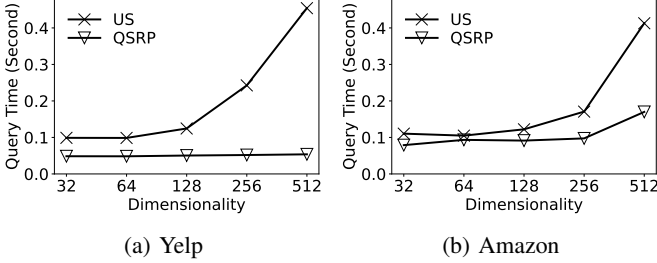


Fig. 12: The influence of dimensionality on query time.

over exact score computation for high-dimensional vectors (about 1.05 according to our measurement). The R-tree in MPA suffers from the curse of dimensionality [32] and degrades to linear scan. The poor performance of Grid and MPA suggests that reverse k -ranks solutions in low dimensionality do not work for high dimensionality. Detailed profiling finds that RMIPS usually requires a large k' to output k users for a query. For instance, on Amazon and with $k=100$, the average k' is 3,806. This phenomenon can be explained by the fact that most products are not popular and thus need a large k' to appear in the top- k' MIPS set of the users. As reported in [35], the query time of Simpfer++ increases quickly with k' , which explains the poor performance of RMIPS.

Second, Table II shows that QSRP consistently outperforms US for all datasets and values of k , and the speedup is usually about 1.5x and could be 4x. This suggests that our two key optimizations, i.e., query-aware sampling and regression-based pruning, are effective, which can also be observed from Figure 11. In particular, query-aware sampling prunes users from rank refinement, and regression-based pruning avoids exact score computation in pruning. The speedup of QSRP over US on Yahoo!Music is smaller than the other datasets because Yahoo!Music has the smallest score table, and thus US already has a good pruning effect with the default 64GB memory index, which is evidenced by the score computation cost in Figure 11(a). Compared with the score computation reduction in Figure 11, the query time speedup of QSRP over US is smaller because operations other than score computation also take up a considerable portion of query time, e.g., binary search and linear function computation.

As MPA, RMIPS, and Grid perform much worse than US and QSRP, we exclude them in the subsequent experiments. Moreover, we mainly report the results on Yelp and Amazon because the observations are similar on the other datasets.

Dimensionality. Figure 12 shows how embedding dimensionality affect the query time of US and QSRP. We observe that the query time of US increases quickly with dimensionality while the query time of QSRP is more stable. This is because

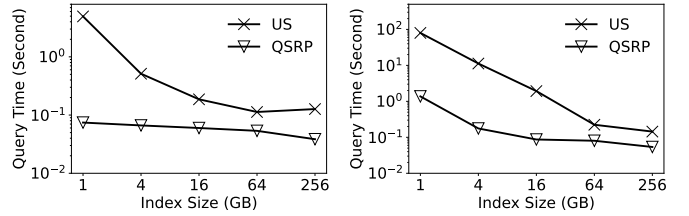


Fig. 13: The influence of memory index size on query time.

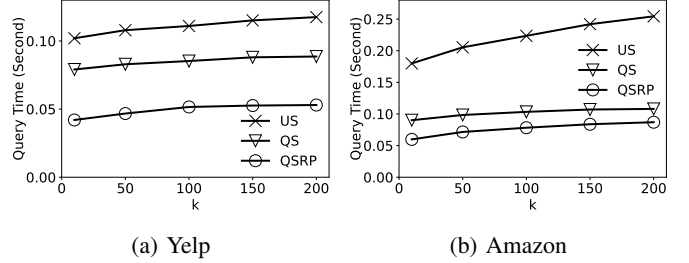


Fig. 14: Query time comparison with different techniques.

pruning becomes more difficult in higher dimensionality, and QSRP has better pruning effect than US due to our query-aware sampling and regression-based pruning optimizations.

Index size. Figure 13 reports the query time with different index sizes. The results show that both QSRP and US achieve shorter query time with a larger index but the query time of US observes a more significant reduction. This is because QSRP adopts query-aware sampling to build the memory index, and its pruning power is already good with a small index size. For instance, to match the query time of QSRP with a 4GB memory index, US needs a memory index of 256GB and 64GB for Yelp and Amazon, respectively. The query time of US increases slightly in the Yelp dataset when the index size increases from 64GB to 256GB. This is because a large index size reduces the refinement cost but incurs additional searching costs in the sampling-based index. The refinement cost of US is very small at 64GB, and searching the sampling-based index (i.e., binary search) becomes the bottleneck.

C. Micro Results

This part conducts experiments to evaluate the designs of QSRP. We use $k=100$ and 64GB memory index size by default. Only Amazon and Yelp are reported for conciseness.

Figure 14 conducts an ablation study for our optimizations, and QS only applies the query-aware sampling. The results show that QS outperforms US. This is because query-aware sampling reduces the number of refined users for each query and thus the computation cost. For example, query-aware sampling needs to refine 28 users on average with $k=100$ in the Amazon dataset, while that number in US is 113. Adding regression-based pruning, QSRP further improves QS because regression-based pruning avoids expensive exact score computation in the pruning stage. The gain of regression-based pruning is larger than query-aware sampling on the Yelp dataset while the opposite is true on the Amazon dataset. This suggests that both optimizations are effective and crucial.

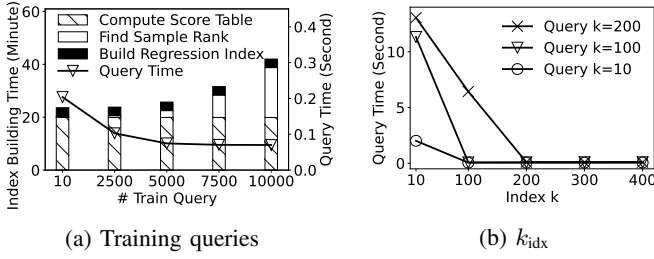


Fig. 15: Query time and index building time with the training parameter k_{idx} and training queries on Amazon.

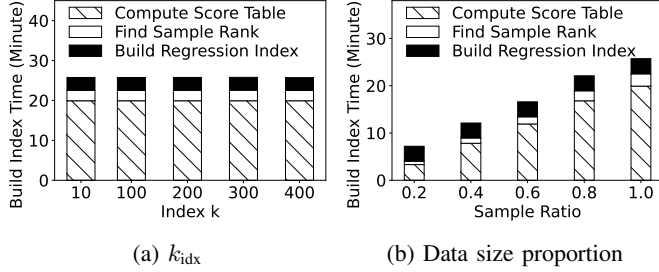


Fig. 16: Index building time w.r.t. the number of required users (i.e., k_{idx}) and dataset size for the Amazon dataset.

Figure 15(a) shows the influence of the number of training queries on query time and index building time. The result shows that time for query-aware sampling increases slowly with the number of training queries, which suggests that the cubic scaling analyzed in Section IV is pessimistic. Figure 15(a) also shows that the query time reduces when increasing from 10 training queries to 2,500 but stabilizes afterward, which suggests that using 2,500-5,000 training queries is sufficient. This is favorable as query-aware sampling does not need many queries to achieve good performance.

Recall that query-aware sampling needs to assign k_{idx} to each training query and we use $k_{idx} = 200$ by default. Figure 15(b) reports how the value of k_{idx} (used in index construction) and k_{qry} (used in actual queries) affects the query time. The results show that the query time reaches the minimum when $k_{qry} = k_{idx}$, becomes stable when $k_{idx} > k_{qry}$ but blows up when $k_{idx} < k_{qry}$. This is because query-aware sampling builds the memory index to minimize the refinement cost in finding the top- k_{idx} users; when $k_{idx} > k_{qry}$, the index is optimized to prune refinement for the top- k_{idx} users, for which the candidates contain the top- k_{qry} users. In contrast, when $k_{idx} < k_{qry}$, the index cannot learn to prune refinement for candidates that rank in $[k_{idx}, k_{qry}]$, resulting in high refinement cost. Thus, we recommend setting k_{idx} as the maximum k_{qry} that may be used by the actual queries.

In Figure 16, we explore the influence of k_{idx} and dataset size on index building time. Figure 16(a) shows that the index construction time stays constant w.r.t. k_{idx} , and our analysis in Section IV-C also suggests that the time complexity of index construction does not depend on k_{idx} . Using μ to denote the sample ratio, for a dataset with m users and n products, we sample $\sqrt{\mu}m$ users and $\sqrt{\mu}n$ products such that the proportion of sampled user-product scores is μ of the original dataset. Figure 16(b) shows that the index building time scales almost

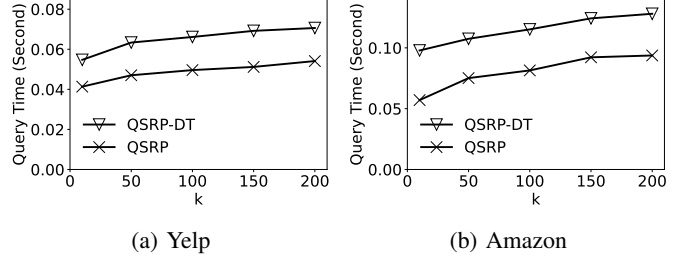


Fig. 17: The influence of distribution transformation.

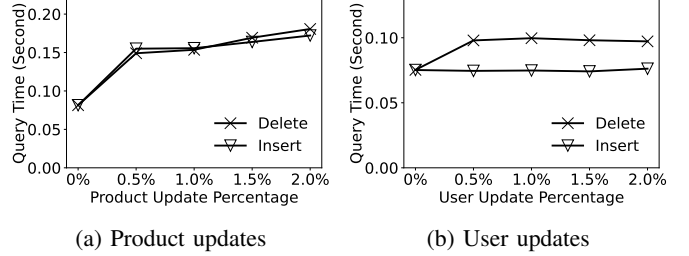


Fig. 18: Query time with updates on Amazon

linearly with the sample ratio. This is because the index-building time is dominated by score computation, whose cost is determined by the number of user-product scores. For the complete Amazon dataset with 3.7TB, QSRP can build an index with around 25 minutes, which shows reasonable scalability w.r.t. dataset size.

In regression-based pruning, we introduce a distribution transformation (i.e., DT) such that the relation between score and rank can be easily fitted with a linear function. Figure 17 compares QSRP with (called QSRP) and without DT (called QSRP-DT). The results show that DT reduces query time on both Amazon and Yelp. This is because the score distribution of both datasets is a bell-shaped score distribution and thus the score-rank relation is difficult to fit without DT.

Figure 18 reports how query time changes with user and product update. The results show that query time increases quickly with product updates but slowly for user updates. This is because product updates introduce rank offset to our rank bounds as discussed in Section VI, and this can cause the users that initially fail in different score segments to overlap in their score bounds, which increases the number of users to refine. Consider an example where the rank bounds of a user are $(1, 5]$ and the rank offset introduced by product updates to this user is $+2$; and the rank bounds of another user are $(5, 10]$ and the rank offset is -1 . The bounds of the two users do not overlap initially, and thus at most one of them may be refined; applying the offsets, their bounds become $(3, 7]$ and $(4, 9]$, which overlap and may require to refine both users to determine their ranking relation.

Figure 18(b) shows that query time increases with user deletes but does not increase with user inserts. This is counter-intuitive because deleting users should reduce query workload while inserting users does the opposite. We attribute this phenomenon to our query-aware sampling. In particular, query-aware sampling learns to prune for the top- k_{idx} users. When deleting users, a user that ranks $k' > k_{idx}$ initially may become the top- k_{idx} but query-aware sampling does not learn to prune

for such users. In contrast, when inserting users, the users need to have a smaller rank $\tilde{k} < k_{\text{idx}}$ to enter the result set. As shown in Figure 15(b), query-aware sampling has good pruning effect when $\tilde{k} < k_{\text{idx}}$. Thus, to accommodate user updates, we recommend to use a large k_{idx} for index building.

VIII. RELATED WORK

Queries for low-dimensional embeddings. These queries consider scenarios where (i) each attribute (i.e., feature or dimensionality of an embedding) indicates a specific feature of the product, e.g., the price of a restaurant; (ii) all attributes are non-negative; (iii) the number of attributes is small (e.g., dimensionality $d < 10$). Given a user preference vector, the *top-k query* returns the k products having the maximum scores (e.g., inner product) for the user [52], [53], [54]. To find potential users for a product, the *reverse top-k query* returns all users whose top- k choices contain the query product [55], [56], [57]. The *reverse k-ranks query* is proposed in [21], which builds an R-tree to index the products and group the users via d -dimensional histograms to facilitate pruning. Grid index [22] proposes a linear scan-based solution, which uses dominance, a popular technique in computation geometry, for product filtering, and computes a cheap score bound by uniformly quantizing each dimension in an embedding vector. RADAR [25] proposes an approximate solution for reverse k -ranks, which guesses the product rank by the ranking situation in each dimension. The *ranked reverse nearest neighbor query* is proposed in [58], which resembles reverse k -ranks but changes the score function from inner product to Euclidean distance. The solution uses the perpendicular bisector and minimum bounding box to partition the space for pruning.

Techniques in these works cannot be applied to the high-dimensional embeddings generated by embedding models for two reasons. First, they usually rely on space-partitioning indexes (e.g., R-tree, X-tree and SR-tree [59]) or dominance-based filtering (e.g., skyline [26], onion layer [27] and UTK [28]), which suffer from the curse of dimensionality [32]. For example, in high dimensionality (e.g., $d > 30$), the filtering power of R-tree vanishes [22], and the skyline contains almost the entire product set [33]. Index construction for dominance-based filtering also becomes prohibitively expensive, e.g., it takes $O(n^{\lfloor d/2 \rfloor})$ time to compute a convex hull for n vectors in d dimensionality [60]. Second, they usually require all elements in the user and product vectors to be non-negative (e.g., for dominance-based filtering to work) while the vectors generated by embedding models contain negative elements.

Queries for high-dimensional embeddings. The *maximum inner product search (MIPS)* query returns the top- k products having the maximum inner product for a given user. Exact solutions for MIPS utilize norm-based pruning and angle-based pruning to filter unpromising products [61], [62], [63]. For instance, FEXIPRO [43] adopts singular value decomposition (SVD) to obtain a tight norm bound and leverage fast integer computation for effective pruning. GPU-IPR [64] process MIPS in GPU and use norm pruning to save the computation

cost. Approximate solutions for MIPS trade accuracy for efficiency and representative methods can be classified into locality-sensitive hashing [17], [20], proximity graph [19], [65], and vector quantization [18], [66]. The *reverse MIPS query* resembles the reverse top- k query and is proposed by Simpfer [42]. Both Simpfer and Simpfer++ [35] store the top- k scores of each user to facilitate pruning. SAH [67] proposes an approximate solution for reverse MIPS and solves reverse MIPS via MIPS, i.e., by checking if the MIPS result set of a user contains the query product. It designs a locality-sensitive hashing scheme for MIPS and proposes a score bound to prune the users that are checked for MIPS.

To our knowledge, we are the first to study the reverse k -ranks query for high-dimensional embeddings. Reverse k -ranks is more challenging than MIPS and reverse MIPS because it considers the ranks of all users (MIPS considers a single user), and the query product may not rank top for the resultant users (reverse MIPS considers top-rank products for each user). As such, our key designs are fundamentally different from the solutions for MIPS and reverse MIPS. For instance, storing only the top- k scores for each user as in Simpfer and Simpfer++ is insufficient for reverse k -ranks as the query product may not rank top, and thus we design query-aware sampling to determine the scores to keep.

IX. CONCLUSIONS

We study the efficient processing of reverse k -ranks query on the high-dimensional embedding vectors generated by embedding models, which are becoming the common practice for the recommendation. As the techniques for low dimensionality (e.g., tree) fail in high dimensionality, we rely on precomputation to speedup query processing. In particular, we propose a sampling-based index architecture to facilitate pruning, which precomputes the score table for all user-product pairs and samples some columns to fit in memory. We also design techniques including query-aware sampling and regression-based pruning to improve the effectiveness of pruning and reduce the cost of computation. Experimental results show that QSRP outperforms state-of-the-art baselines by orders of magnitude and our designs are effective.

ACKNOWLEDGEMENT

This work is supported by grant GRF 152043/23E from Hong Kong RGC, the Guangdong Basic and Applied Basic Research Foundation (Grant No.2021A151110067), Shenzhen Fundamental Research Program (Grant No. 2022081511 2848002), the Guangdong Provincial Key Laboratory (Grant No. 2020B121201001) and a research gift from Huawei Gauss department. Dr. Bo Tang is also affiliated with the Research Institute of Trustworthy Autonomous Systems, Southern University of Science and Technology, Shenzhen, China.

REFERENCES

- [1] Y. Koren, R. M. Bell, and C. Volinsky, "Matrix factorization techniques for recommender systems," *Computer*, vol. 42, no. 8, pp. 30–37, 2009.
- [2] H. Wang, F. Zhang, J. Wang, M. Zhao, W. Li, X. Xie, and M. Guo, "Ripplet: Propagating user preferences on the knowledge graph for recommender systems," in *CIKM*, 2018, pp. 417–426.

- [3] T. Chen, L. Hong, Y. Shi, and Y. Sun, "Joint text embedding for personalized content-based recommendation," *CoRR*, vol. abs/1706.01084, 2017.
- [4] X. He, H. Zhang, M. Kan, and T. Chua, "Fast matrix factorization for online recommendation with implicit feedback," in *SIGIR*, 2016, pp. 549–558.
- [5] Y. Hu, Y. Koren, and C. Volinsky, "Collaborative filtering for implicit feedback datasets," in *ICDM*, 2008, pp. 263–272.
- [6] H. Xue, X. Dai, J. Zhang, S. Huang, and J. Chen, "Deep matrix factorization models for recommender systems," in *IJCAI*, 2017, pp. 3203–3209.
- [7] C. Chen, P. Zhao, L. Li, J. Zhou, X. Li, and M. Qiu, "Locally connected deep learning framework for industrial-scale recommender systems," in *WWW*, 2017, pp. 769–770.
- [8] P. Covington, J. Adams, and E. Sargin, "Deep neural networks for youtube recommendations," in *RecSys*, 2016, pp. 191–198.
- [9] R. Ying, R. He, K. Chen, P. Eksombatchai, W. L. Hamilton, and J. Leskovec, "Graph convolutional neural networks for web-scale recommender systems," in *SIGKDD*, 2018, pp. 974–983.
- [10] X. He, D. Deng, X. Wang, Y. Li, Y. Zhang, and M. Wang, "Lightgcn: Simplifying and powering graph convolution network for recommendation," in *SIGIR*, 2020, pp. 639–648.
- [11] H. Tang, S. Wu, G. Xu, and Q. Li, "Dynamic graph evolution learning for recommendation," in *SIGIR*, 2023, p. 1589–1598.
- [12] R. Guo, X. Luan, L. Xiang, X. Yan, X. Yi, J. Luo, Q. Cheng, W. Xu, J. Luo, F. Liu *et al.*, "Manu: a cloud native vector database management system," *PVLDB*, vol. 15, no. 12, pp. 3548–3561, 2022.
- [13] J. Wang, X. Yi, R. Guo, H. Jin, P. Xu, S. Li, X. Wang, X. Guo, C. Li, X. Xu, K. Yu, Y. Yuan, Y. Zou, J. Long, Y. Cai, Z. Li, Z. Zhang, Y. Mo, J. Gu, R. Jiang, Y. Wei, and C. Xie, "Milvus: A purpose-built vector data management system," in *SIGMOD*, 2021, pp. 2614–2627.
- [14] J. Wang, P. Huang, H. Zhao, Z. Zhang, B. Zhao, and D. L. Lee, "Billion-scale commodity embedding for e-commerce recommendation in alibaba," in *SIGKDD*, 2018, pp. 839–848.
- [15] A. van den Oord, S. Dieleman, and B. Schrauwen, "Deep content-based music recommendation," in *NeurIPS*, 2013, pp. 2643–2651.
- [16] S. Rendle, W. Krichene, L. Zhang, and J. Anderson, "Neural collaborative filtering vs. matrix factorization revisited," in *RecSys*, 2020, p. 240–248.
- [17] A. Shrivastava and P. Li, "Asymmetric LSH (ALSH) for sublinear time maximum inner product search (MIPS)," in *NeurIPS*, 2014, pp. 2321–2329.
- [18] X. Dai, X. Yan, K. K. W. Ng, J. Liu, and J. Cheng, "Norm-explicit quantization: Improving vector quantization for maximum inner product search," in *AAAI*, 2020, pp. 51–58.
- [19] J. Liu, X. Yan, X. Dai, Z. Li, J. Cheng, and M. Yang, "Understanding and improving proximity graph based maximum inner product search," in *AAAI*, 2020, pp. 139–146.
- [20] X. Yan, J. Li, X. Dai, H. Chen, and J. Cheng, "Norm-ranging LSH for maximum inner product search," in *NeurIPS*, 2018, pp. 2956–2965.
- [21] Z. Zhang, C. Jin, and Q. Kang, "Reverse k-ranks query," *PVLDB*, vol. 7, no. 10, pp. 785–796, 2014.
- [22] Y. Dong, H. Chen, J. X. Yu, K. Furuse, and H. Kitagawa, "Grid-index algorithm for reverse rank queries," in *EDBT*, 2017, pp. 306–317.
- [23] Y. Qian, H. Li, N. Mamoulis, Y. Liu, and D. W. Cheung, "Reverse k-ranks queries on large graphs," in *EDBT*, 2017, pp. 37–48.
- [24] Ç. Aslay, W. Lu, F. Bonchi, A. Goyal, and L. V. S. Lakshmanan, "Viral marketing meets social advertising: Ad allocation with minimum regret," *PVLDB*, vol. 8, no. 7, pp. 822–833, 2015.
- [25] S. Dutta, "RADAR: fast approximate reverse rank queries," in *Intelligent Systems and Applications*, vol. 1252, 2020, pp. 748–757.
- [26] S. Börzsönyi, D. Kossmann, and K. Stocker, "The skyline operator," in *ICDE*, 2001, pp. 421–430.
- [27] Y. Chang, L. D. Bergman, V. Castelli, C. Li, M. Lo, and J. R. Smith, "The onion technique: Indexing for linear optimization queries," in *SIGMOD*, 2000, pp. 391–402.
- [28] K. Mouratidis and B. Tang, "Exact processing of uncertain top-k queries in multi-criteria settings," *PVLDB*, vol. 11, no. 8, pp. 866–879, 2018.
- [29] N. Beckmann, H. Kriegel, R. Schneider, and B. Seeger, "The r*-tree: An efficient and robust access method for points and rectangles," in *SIGMOD*, 1990, pp. 322–331.
- [30] P. Ciaccia, M. Patella, and P. Zezula, "M-tree: An efficient access method for similarity search in metric spaces," in *VLDB*, 1997, pp. 426–435.
- [31] J. Zhang, B. Tang, M. L. Yiu, X. Yan, and K. Li, "T-levelindex: Towards efficient query processing in continuous preference space," in *SIGMOD*, 2022, pp. 2149–2162.
- [32] R. Weber, H. Schek, and S. Blott, "A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces," in *VLDB*, 1998, pp. 194–205.
- [33] Z. Zhang, X. Guo, H. Lu, A. K. H. Tung, and N. Wang, "Discovering strong skyline points in high dimensional spaces," in *CIKM*, 2005, pp. 247–248.
- [34] T. Kraska, A. Beutel, E. H. Chi, J. Dean, and N. Polyzotis, "The case for learned index structures," in *SIGMOD*, 2018, pp. 489–504.
- [35] D. Amagata and T. Hara, "Reverse maximum inner product search: Formulation, algorithms, and analysis," *ACM Trans. Web*, vol. 17, no. 4, 2023.
- [36] Z. Zhou, S. Tan, Z. Xu, and P. Li, "Möbius transformation for fast inner product search on graph," in *NeurIPS*, 2019, pp. 8216–8227.
- [37] H. Cheng, L. Koc, J. Harmsen, T. Shaked, T. Chandra, H. Aradhye, G. Anderson, G. Corrado, W. Chai, M. Ispir, R. Anil, Z. Haque, L. Hong, V. Jain, X. Liu, and H. Shah, "Wide & deep learning for recommender systems," in *RecSys*, 2016, pp. 7–10.
- [38] "Dataset source," <https://github.com/RUCAIBox/RecSysDatasets/>.
- [39] F. Ricci, L. Rokach, and B. Shapira, "Introduction to recommender systems handbook," in *Recommender systems handbook*. Springer, 2010, pp. 814–816.
- [40] X. He, L. Liao, H. Zhang, L. Nie, X. Hu, and T. Chua, "Neural collaborative filtering," in *WWW*, 2017, pp. 173–182.
- [41] J. Zheng, J. Mai, and Y. Wen, "Explainable session-based recommendation with meta-path guided instances and self-attention mechanism," in *SIGIR*, 2022, pp. 2555–2559.
- [42] D. Amagata and T. Hara, "Reverse maximum inner product search: How to efficiently find users who would like to buy my item?" in *RecSys*, 2021, pp. 273–281.
- [43] H. Li, T. N. Chan, M. L. Yiu, and N. Mamoulis, "FEXIPRO: fast and exact inner product retrieval in recommender systems," in *SIGMOD*, 2017, pp. 835–850.
- [44] R. Seidel, "Small-dimensional linear programming and convex hulls made easy," *Discret. Comput. Geom.*, vol. 6, pp. 423–434, 1991.
- [45] C. Hsieh, L. Yang, Y. Cui, T. Lin, S. J. Belongie, and D. Estrin, "Collaborative metric learning," in *WWW*, 2017, pp. 193–201.
- [46] T. D. Q. Vinh, Y. Tay, S. Zhang, G. Cong, and X. Li, "Hyperbolic recommender systems," *CoRR*, vol. abs/1809.01703, 2018.
- [47] G. Dror, N. Koenigstein, Y. Koren, and M. Weimer, "The yahoo! music dataset and kdd-cup '11," in *Proceedings of KDD Cup 2011 competition*, vol. 18, 2012, pp. 8–18.
- [48] N. Asghar, "Yelp dataset challenge: Review rating prediction," *CoRR*, 2016.
- [49] F. M. Harper and J. A. Konstan, "The movielens datasets: History and context," *ACM Trans. Interact. Intell. Syst.*, vol. 5, no. 4, pp. 19:1–19:19, 2016.
- [50] J. Ni, J. Li, and J. J. McAuley, "Justifying recommendations using distantly-labeled reviews and fine-grained aspects," in *EMNLP-IJCNLP*, 2019, pp. 188–197.
- [51] H. Yu, C. Hsieh, S. Si, and I. S. Dhillon, "Scalable coordinate descent approaches to parallel matrix factorization for recommender systems," in *ICDM*, 2012, pp. 765–774.
- [52] I. F. Ilyas, G. Beskales, and M. A. Soliman, "A survey of top-k query processing techniques in relational database systems," *ACM Computing Surveys (CSUR)*, vol. 40, no. 4, pp. 11:1–11:58, 2008.
- [53] G. Das, D. Gunopulos, N. Koudas, and N. Sarkas, "Ad-hoc top-k query answering for data streams," in *PVLDB*, 2007, pp. 183–194.
- [54] K. Mouratidis, K. Li, and B. Tang, "Marrying top-k with skyline queries: Relaxing the preference input while producing output of controllable size," in *SIGMOD*, 2021, pp. 1317–1330.
- [55] A. Yu, P. K. Agarwal, and J. Yang, "Processing a large number of continuous preference top-k queries," in *SIGMOD*, 2012, pp. 397–408.
- [56] A. Vlachou, C. Doulkeridis, Y. Kotidis, and K. Nørnvåg, "Reverse top-k queries," in *ICDE*, 2010, pp. 365–376.
- [57] A. Vlachou, C. Doulkeridis, K. Nørnvåg, and Y. Kotidis, "Branch-and-bound algorithm for reverse top-k queries," in *SIGMOD*, 2013, pp. 481–492.
- [58] K. C. K. Lee, B. Zheng, and W. Lee, "Ranked reverse nearest neighbor search," *TKDE*, vol. 20, no. 7, pp. 894–910, 2008.
- [59] C. Böhm, S. Berchtold, and D. A. Keim, "Searching in high-dimensional spaces: Index structures for improving the performance of multimedia databases," *ACM Computing Surveys (CSUR)*, vol. 33, no. 3, pp. 322–373, 2001.

- [60] A. Yu, P. K. Agarwal, and J. Yang, "Top-k preferences in high dimensions," *TKDE*, vol. 28, no. 2, pp. 311–325, 2016.
- [61] P. Ram and A. G. Gray, "Maximum inner-product search using cone trees," in *SIGKDD*, 2012, pp. 931–939.
- [62] C. Teflioudi, R. Gemulla, and O. Mykytiuk, "LEMP: fast retrieval of large entries in a matrix product," in *SIGMOD*, 2015, pp. 107–122.
- [63] C. Teflioudi and R. Gemulla, "Exact and approximate maximum inner product search with LEMP," *ACM Trans. Database Syst.*, vol. 42, no. 1, pp. 5:1–5:49, 2017.
- [64] L. Xiang, B. Tang, and C. Yang, "Accelerating exact inner product retrieval by CPU-GPU systems," in *SIGIR*, 2019, pp. 1277–1280.
- [65] S. Morozov and A. Babenko, "Non-metric similarity graphs for maximum inner product search," in *NeurIPS*, 2018, pp. 4726–4735.
- [66] R. Guo, S. Kumar, K. Choromanski, and D. Simcha, "Quantization based fast inner product search," in *AISTATS*, vol. 51, 2016, pp. 482–490.
- [67] Q. Huang, Y. Wang, and A. K. Tung, "Sah: Shifting-aware asymmetric hashing for reverse k maximum inner product search," in *AAAI*, 2023, pp. 4312–4321.