MaxEmbed: Maximizing SSD bandwidth utilization for huge embedding models serving

Ruwen Fan Minhui Xie* Haodi Jiang Youyou Lu*
{frw23,xmh19,jhd20}@mails.tsinghua.edu.cn,luyouyou@tsinghua.edu.cn
Department of Computer Science and Technology, Tsinghua University

Abstract

Deep learning recommendation models (DLRMs) have gained widespread application across search, advertising, and e-commerce. Still, DLRMs present notable challenges as they depend heavily on large embedding tables to represent sparse features in recommendation systems. This raises concerns about both memory capacity and cost. Solid-state drives (SSDs) offer a cost-effective solution with a significantly larger capacity, but they introduce read amplification issues because of the mismatch between embedding size and SSD read granularity. Prior SSD embedding storage systems aim to tackle these challenges by employing hypergraph partitioning to co-locate co-appearing embeddings onto the same SSD page, alleviating read amplification. However, this approach has a drawback as it divides embeddings into completely disjoint clusters, limiting potential combinations between embeddings.

In response to this limitation, we introduce MaxEmbed. Capitalizing on the extensive storage capacity of SSDs, MaxEmbed effectively mines relationships between storage combinations of embeddings with replication, thereby enhancing the effective bandwidth of SSDs. Additionally, MaxEmbed incorporates a corresponding online service module for embedding query request handling, leveraging two key optimizations to reduce the overhead brought by replication. Our evaluations demonstrate that MaxEmbed boosts SSD embedding serving throughput by up to 18.7% under various settings.

CCS Concepts: • **Information systems** → *Information storage systems.*

Keywords: Embedding models, SSD, Data placement, Data replication

*Minhui Xie and Youyou Lu are the corresponding authors.

ASPLOS '24, April 27-May 1, 2024, La Jolla, CA, USA

ACM ISBN 979-8-4007-0391-1/24/04.
https://doi.org/10.1145/3622781.3674172

ACM Reference Format:
Ruwen Fan, Minhui Xie, Haodi Jiang and Youyou Lu. 2024. MaxEmbed: Maximizing SSD bandwidth utilization for huge embedding models serving. In *29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 4 (ASPLOS '24), April 27-May 1, 2024, La Jolla, CA, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3622781.3674172

1 Introduction

Deep learning recommendation models (DLRMs) [29] find extensive applications in various domains, including search, advertising, and e-commerce [16, 42, 45]. DLRMs depend heavily on large *embedding* parameters to capture unique object characteristics. However, the rapid expansion in model size — growing tenfold annually — contrasts starkly with the slower growth of memory capacity [14, 15, 26, 39]. In this context, solid-state drives (SSDs) offer a cost-effective solution with superior storage capacity compared to traditional memory, making them increasingly vital for storing DLRM embeddings [21, 35, 37, 38, 41].

However, utilizing SSDs for storing model parameters introduces a notable challenge. SSDs have a fixed page granularity, typically 4KB, while embedding parameters are much smaller, ranging from 64 to 512 bytes [28, 30]. When reading an embedding, the whole 4KB page needs to be read from SSD, which causes read amplification in SSDs. It restrains the effective bandwidth and thus degrades the whole system throughput.

To mitigate read amplification, a prior study tries to co-locate those embeddings that commonly appear together on one SSD page [11], thus serving more embeddings per SSD page read. It employs hypergraph partitioning to identify commonly co-occurring embeddings to guide the embedding placement. Although it alleviates the phenomenon of reading amplification, the effective bandwidth of SSD is still very limited, only about 8.58% in our evaluation.

With deep analysis, we find that the number of naturally co-appearing embeddings often surpasses an SSD page's storage capacity. This intrinsically prohibits the aggregation of these embeddings. Some of those embeddings are destined to be scattered into different SSD pages if there is only one copy of each embedding, which will continue to constrain SSD bandwidth utilization. This indicates that simply rearranging the embedding placement is not enough.

To overcome these limitations, we introduce MaxEmbed, an SSD-based solution for embedding storage and retrieval. MaxEmbed aims to exploit more co-appear patterns by selectively replicating embeddings. It leverages the vast space of SSD for additional replicas of embedding to store more possible combinations between embeddings. MaxEmbed contains *offline* and *online* two phases. These two phases focus on how to make a replication and handle embedding query requests in a replica scenario, respectively. The offline phase of MaxEmbed processes the embedding placement. We propose a replica strategy based on hypergraph partition. Leveraging the connectivity and co-appearance information, it wisely replicates and places the embeddings to exploit the benefits of extra storage space. The online phase of MaxEmbed focuses on selecting proper replicas to read for online serving. With the introduction of replicas, selecting the minimum number of pages to read is NP-hard, and greedy approximation incurs non-negligible overheads. MaxEmbed addresses this by combining a pipeline strategy to offset software overhead with SSD read latency and employing an index limit to reduce this overhead further.

We evaluate MaxEmbed on five real-world datasets with two types of SSDs, and the results demonstrate that using only 10% additional embedding storage space, MaxEmbed can improve the SSD effective bandwidth by 2% - 10.0% compared with the baseline embedding placement and achieve 1.7% – 8.8% end-to-end throughput improvement than prior embedding placement strategy. When the replication ratio comes to 80%, the effective bandwidth improves by 7% - 19.0%, and end-to-end throughput improves by 8.9% – 18.7%.

We summarize the contribution of this paper as follows:

- We identify the inherent defect of the existing embedding placement strategy and the opportunities of using replication for embedding storage and retrieval to enhance the bandwidth utilization of SSDs.
- We propose MaxEmbed, an SSD-based solution for embedding storage and retrieval, to overcome the limitations when using hypergraph partition in embedding placement and to reduce the overhead when processing online query serving with replication.
- We evaluate MaxEmbed using five real-world datasets and demonstrate the effective bandwidth of SSD when using MaxEmbed to perform embedding replication, placement, and online serving.

2 Background

2.1 Deep learning recommendation models

Deep learning recommendation models (DLRMs) [29] find widespread application in various scenarios, including searching [16], e-commerce [44, 45], and advertising [31, 42]. Different from traditional deep learning scenarios like CV or NLP, the input features in these scenarios are extremely high-dimensional sparse IDs, e.g., user IDs, item IDs, and user-item cross IDs [13]. Due to the dimension disaster, traditional DNNs can not directly learn from high-dimensional inputs. Thus, DLRMs leverage an *embedding layer* to map high-dimensional sparse IDs to low-dimensional dense vectors (called *embedding vectors* or *embeddings* in short). Specifically, for each ID category, the model maintains a separate hash table (called *embedding table*), where its keys are IDs and its values are embedding vectors. Owing to the large capacity of IDs in a feature category (billions-level [26, 39]) and the high number of feature categories (around several hundred [3, 18]), the embedding layer contributes to up to 99.9% of the parameters of the whole model [12, 26].

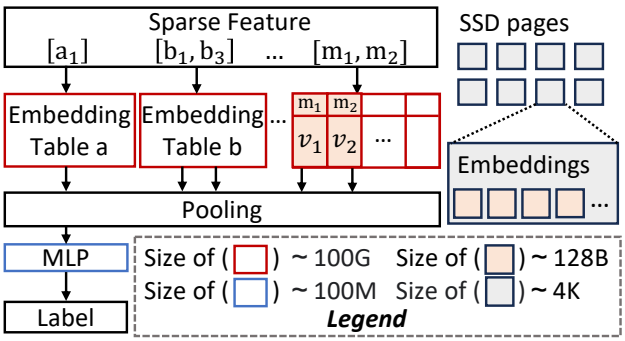

Figure 1. Basic structure of Deep Learning Recommendation Model (DLRM) with SSD storage. The embedding table occupies the vast majority of DLRM storage space. Storing it on SSD will bring the read amplification problem.

Figure 1 shows the structure of a DLRM example. It first uses the corresponding embedding tables to cast sparse input features (a_i, b_i, m_i in the Figure) to embedding vectors. Then, these vectors are fed into DNN [9, 25] to determine the probability of a user clicking on or viewing a product. In this system, the size of the embedding table accounts for most of the entire model parameters.

DLRMs pose great pressure on the underlying storage system. 1) Currently, the parameter capacity of modern DLRMs has surged to hundreds of trillions, reflecting an astounding annual growth rate of 10 times over the past five years [14, 15, 26, 39]. This exponential expansion has resulted in significant increases in storage costs. 2) Looking ahead, this trend is expected to persist and continue its upward trajectory. As common wisdom in the ML community is that more parameters tend to achieve better model performance, ML engineers never stop exploring and exploiting more complex models in pursuit of corresponding revenue in model accuracy, e.g., with more complicated feature acquisition [27], or special feature engineering techniques like crossing [40]. These endeavors will lead to more parameters with more significant access pressure.

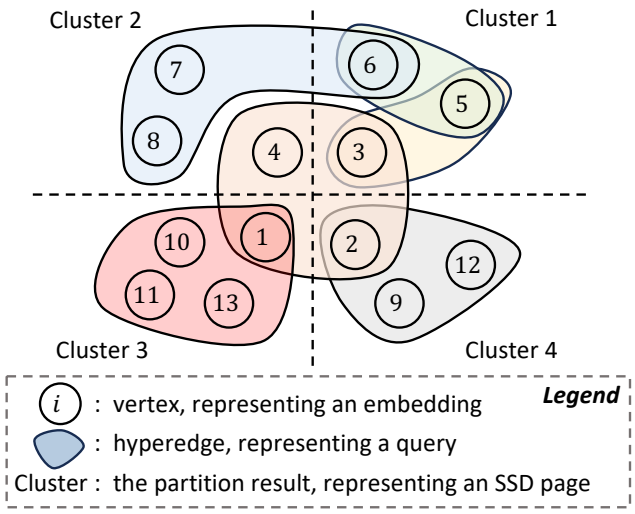

Figure 2. Existing system: hypergraph partition. Each circle represents a vertex, and each colored region represents a hyperedge. Each vertex will only be partitioned into one cluster.

2.2 Using SSD to store embedding vectors

Historically, such large-scale DLRMs are stored in DRAM [29]. However, the growing trend of parameters makes this practice increasingly cost-inefficient, for the high cost of DRAM medium and the sluggish growth rate in DRAM capacity. Solid State Drives (SSDs) offer more cost-effective and capacious storage. With the help of new hardware medium (like Optane [10]) and high-performance software protocols (like Non-Volatile Memory Express, NVMe), off-the-shelf SSDs can deliver over 7 GB/s bandwidth and 5 μs I/O latency, which shows great potential in storing large-scale DLRMs.

Despite these benefits, an inherent challenge emerges when employing SSDs: the mismatch between the inner granularity of SSDs and the access granularity of embedding parameters, which leads to severe read amplification. Specifically, SSD devices have a minimum read and write granularity, as they internally store data at a page granularity (a typical page size is 4KB). In contrast, typical embedding parameters are relatively small, ranging from 64 to 512 bytes [28, 30]. This mismatch exhibits a pronounced read amplification phenomenon, limits the effective read bandwidth of SSDs, and undermines the capacity advantages SSDs offer.

To handle this challenge, Bandana [11], a recent work proposed by Meta, suggests that the placement of embedding vectors in SSD should be carefully considered to reduce read amplification. Specifically, it first analyzes the past access patterns with the hypergraph partitioning technique and finds embedding vectors that may be accessed simultaneously. Note that hypergraph is an extension of the traditional graph, where an edge (called *hyperedge*) can connect more than two vertices. As Figure 2 shows, to build the hypergraph,

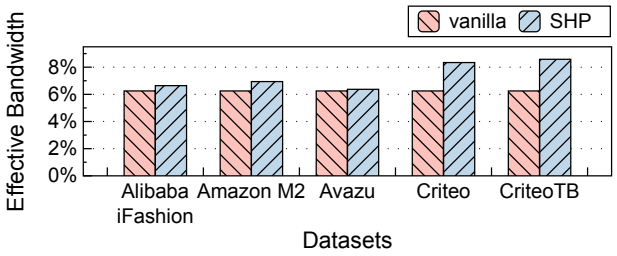

Figure 3. SSD effective bandwidth under vanilla and SHP-partitioned embedding placement. Compared with vanilla embedding placement, using SHP to guide the embedding placement does improve the effective bandwidth. But the effective bandwidth is still low.

Bandana maps all embeddings in the recommendation system to vertices of a hypergraph, and treats the relationship in an embedding read request as a hyperedge (which connects all vertices corresponding to embeddings in this request).

To minimize the number of times the embedding query reads SSD, that is, to reduce the number of connected clusters on the hyperedge in the above hypergraph, which is consistent with the goal of the hypergraph partitioning connectivity optimization problem. Then, Bandana uses the Social Hash Partition (SHP) [20] algorithm for hypergraph partitioning. Hypergraph partitioning can divide the vertices into a specified number of clusters so that the number of vertices in them is balanced. Bandana believes the vertices in each cluster obtained by hypergraph partition often co-appear. Guided by these co-appearing patterns, Bandana strives to co-locate the related embedding vectors within the same SSD page. This strategic arrangement allows a single read operation to fetch more effective embeddings, enhancing the overall effective read bandwidth.

3 Motivation

Our observation. Figure 3 shows the effective SSD read bandwidth of vanilla placement (sequentially arranged one by one) and the state-of-the-art placement solution (SHP, hypergraph partition algorithm used by Bandana) across four datasets; refer to §8.1 for detailed experimental settings. From the results, we observe that although the effective bandwidth is greatly improved in Bandana (improved by $1.1\times - 2.2\times$), it is still significantly below the bandwidth cap of SSD (e.g., utilizing only 8.58% in the Criteo dataset). Hence, the severity of read amplification still persists.

With deep analysis, we find that the inherent defect of hypergraph partition algorithms causes this extreme underutilization of bandwidth: the optimization goal of partition algorithms is to maximize the opportunity of co-appearance *within one SSD page*. Thus, any vertex in the hypergraph will be partitioned into *only one cluster*, as shown in Figure 2,

which only captures a limited number of co-appearance relationships (e.g., in Figure 2, vertex 1 is partitioned into cluster 3, causing the combination relationship between vertices $\langle 1, 2, 3, 4\rangle$ to be destroyed). However, in real-world workloads, an embedding can easily co-appear with more embeddings than an SSD page can hold (e.g., in the CriteoTB dataset, the top 5% of the hottest embeddings are likely to co-appear with more than 40 embeddings, but the number of embeddings that can be stored within an SSD page, typically 8 to 32).

This mismatch means an embedding vector is adjacent to many vertices in a hypergraph but stored nearby with only a small part of them. Consequently, when embeddings that are not co-located are queried simultaneously, it results in multiple SSD read operations. This severely restricts the effective read bandwidth of SSDs. It indicates the insufficiency of merely rearranging the placement of embeddings. To overcome this limitation, we need to find more potential combinations between embeddings.

Key idea. Based on the analysis above, we aim to *exploit more co-appear patterns by selectively replicating embeddings.* Specifically, by carefully trading off the potential benefits and space overhead (under a predetermined space amplification ratio r), we selectively replicate some hotspot embeddings to multiple pages of SSD to capture more co-appearing patterns and finally boost the effective bandwidth of SSDs. For example, in Figure 2, by replicating embedding 6 to both cluster 1 and 2, we can simultaneously alleviate reading amplification when queried with $\langle 6, 7, 8\rangle$ and $\langle 5, 6\rangle$.

Making replication will also lead to additional storage space. Given the ample storage capacity of SSDs, which can extend to several terabytes, there is enough room to allocate additional space for storing more combinations of these replicas.

Through replication, we hope to capture possible combinations that are not included in the hypergraph partition results. Therefore, more embeddings that are easily queried together can be stored on the same SSD page, which is expected to reduce SSD read operations, mitigate the read amplification phenomenon, and finally enhance the effective bandwidth of SSD read operations. Although the concept is clear, making such a replication is challenging.

Challenge #1: embedding placement with replication. The main design challenge is the embedding placement, which we define as max-bandwidth embedding placement with replication problem (Rep-MBEP). In Rep-MBEP, N embeddings (vertices) and E queries (hyperedges) are given. We need to find an embedding placement on SSD where each page (cluster) contains at most d embeddings. In addition, some embeddings could be placed on multiple pages while following the constraint that the proportion of replicas must not exceed r (replication ratio). The objective is to minimize the number of SSD pages each query needs to find all its

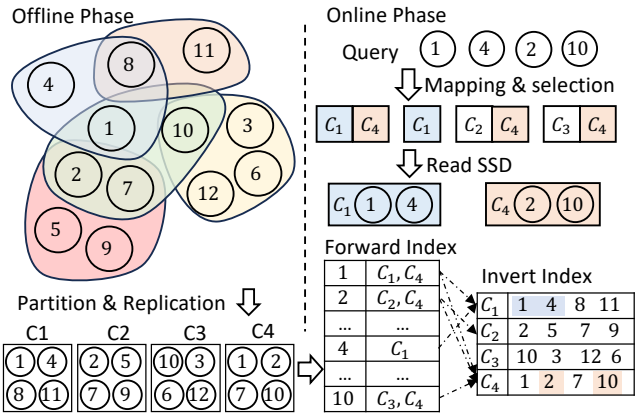

Figure 4. MaxEmbed overview.

embeddings. Primarily, Rep-MBEP can be reducible to max-bandwidth embedding placement problem (MBEP) when $r = 0$, while MBEP itself, which equals to the hypergraph partition problem, is an NP-hard problem [24] and existing placement algorithms [7, 20, 32] all use some heuristics to solve it. Thus, Rep-MBEP is also NP-hard, and developing an embedding placement in the Rep-MBEP scenario will be even more difficult.

Challenge #2: page selection when online serving. Since the replication results in potentially many SSD pages containing the same embedding, a subsequent design challenge emerges when online serving is the page selection algorithm to identify a minimal SSD page set that covers all embeddings in a given request. This is difficult since an online query may contain tens of embeddings, multiplying its enumeration complexity. This predicament is equivalent to the set cover optimization problem, also a well-known NP-hard challenge. Opting for an aggressive approach to select replicas may introduce considerable software overhead, substantially increasing latency. This latency spike could potentially negate the advantage of reduced page read count by embedding placement.

4 MaxEmbed Overview

Based on the above analysis, we present MaxEmbed, an SSD embedding storage solution aiming to improve the effective bandwidth of SSD through replicas and alleviate the problem of read replica selection. MaxEmbed comprises two primary components, as illustrated in Figure 4.

Offline phase: partitioning and replication. In the offline phase, MaxEmbed primarily mitigates the problem of Rep-MBEP. The input for this offline component includes a sequence of logs containing historical logs, replica ratios, and cluster sizes. MaxEmbed utilizes these logs to construct a hypergraph, performs partitioning and replication, and

subsequently generates a set of mappings for each embedding key to its corresponding storage page locations on the SSD.

Online phase: query request processing. The online phase of MaxEmbed is primarily dedicated to serving incoming queries for embedding lookup. In this phase, MaxEmbed leverages the partitioned mapping information from the offline phase to serve online query requests. From the partition information, MaxEmbed will build a *Forward Index* (map of embedding to SSD storage position) and an *Invert Index* (map of each SSD page to embeddings it contains). MaxEmbed divides the incoming parameter query requests based on a replica selection strategy according to the Forward Index and the Invert Index. Additionally, MaxEmbed adopts two optimization methods to reduce the query process overhead.

5 Offline Phase: Partitioning and Replication

The objective of Rep-MBEP is to selectively replicate a subset of the entire embedding vectors at a predetermined replication ratio r and strategically make embedding placement to maximize the effective bandwidth of the SSD. As analyzed in §3, the reason for the insufficient effective bandwidth of SSD is that only a small amount of valid embeddings can be obtained in a single read operation caused by limited embedding combinations. Therefore, we aim to introduce vertex replicas into the hypergraph to explore a wider range of embedding parameter combinations.

In order to perform replication operations, two problems must be addressed: 1) which vertices should be replicated, and 2) how should the replicas be placed? Building upon the existing hypergraph partitioning algorithm, we have two options: 1) replication prior to partitioning, leveraging the hypergraph partition algorithm to decide the placement of replicas. 2) First partition, then replication, determining which embeddings need to be replicated based on the results of the hypergraph partition. We have developed three replication strategies. Each strategy has been analyzed, allowing us to understand their advantages and disadvantages.

5.1 Strawman 1: Replication prior to partition (RPP)

As hot embeddings tend to co-appear with more other embeddings, making replicas of these vertices increases the probability that they will co-occur with other neighbor vertices. As shown in Figure 5 (a), a direct way to make replication is to generate replicas of the first r proportion of vertices according to their popularity and connect these replicas with the hyperedges. All these replicas are treated as regular vertices and put all these vertices into the partition algorithm. This leaves the embedding placement problem to the hypergraph partition algorithm.

Poor performance improvements. However, as evaluated in §8.4, this method was not effective. This can be attributed to two main factors. First, selecting replicas depends entirely on hotspot information. As a result, vertices chosen based on hotspots may not have necessary adjacent relations, and the replicas could offer limited advantages (e.g., as in Figure 5 (a), cold vertex like vertex 4 would not be replicated, but replicating vertex and co-locating it with vertices $\langle 1, 2, 3\rangle$ will be better.) Unfortunately, predicting potential combinations between vertices is difficult before hypergraph partitioning. Second, it is hard to prevent duplicated combinations from being created using this method, causing storage space wasted.

5.2 Strawman 2: Finer-partition and fill with replication (FPR)

As a vertex can be placed into only one cluster during the hypergraph partition, the possible combinations of the adjacent vertices will be limited. A way to alleviate this is to scan each cluster to find the most common co-appeared vertices in all clusters and add these vertices as replicas into the cluster. However, the number of vertices that can be placed in each cluster is limited, so it can be considered to divide the cluster into smaller ones and then create replicas through the above method. When a replication ratio r is set, $(1+r)N/d$ clusters, each containing d embeddings, will be generated. As shown in Figure 5 (b), the hypergraph is directly partitioned into $(1+r)N/d$ clusters to get smaller clusters. The number of vertex in each cluster is less than d. We count the vertices with the most common occurrences of vertices in each cluster to identify adjacent vertices to make replicas to populate the cluster.

Instability of this method. We also evaluate this method in §8.4. Results show that the performance of this method is not stable, and sometimes, it is even worse than without replicas in certain datasets. The main drawback of this method is that the hypergraph is partitioned into finer clusters, which may destroy the original embedding combination (e.g., as shown in Figure 5 (b), vertices $\langle 1, 2, 3, 4\rangle$ are separated in this example).

5.3 Solution: Connectivity-priority replication

From §5.1, it's evident that creating replicas before partitioning ignores the relationships between vertices and the information of the hypergraph, resulting in an unsatisfactory result. Thus, it's more effective to replicate after partitioning. Additionally, as §5.2 highlights, finer partitioning destroys vertices association, thus leading to a worse partition. It's crucial to preserve the original hypergraph partitioning result before initiating replication.

Vertices that need to be replicated are those in a hyperedge with high connectivity. Here, if the vertices in a hyperedge are distributed in λ clusters, the connectivity of the hyperedge is λ. We treat each query as a hyperedge, so the connectivity of a hyperedge equals the read operation count

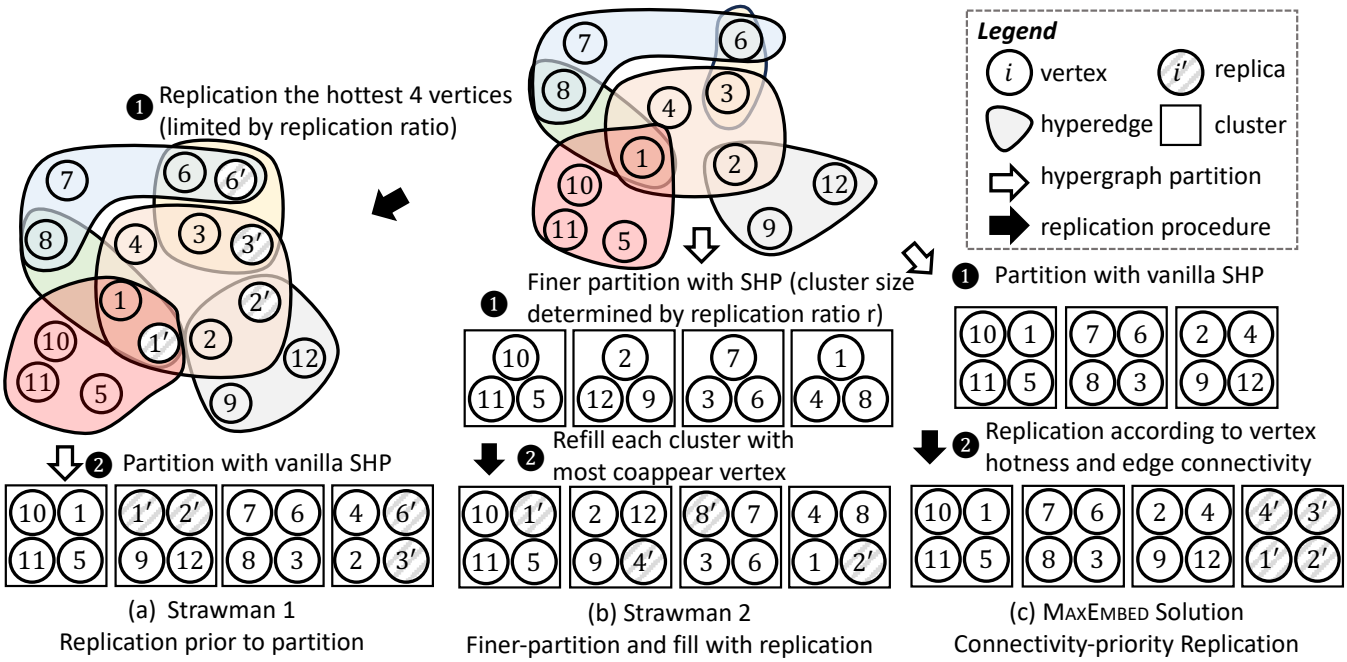

Figure 5. Three replication strategies. (a) RPP method: making replication based on hotness of vertices before partition, leveraging hypergraph partition to decide the placement. (b) FPR method: partition the origin hypergraph into finer clusters and fill each cluster with replication. (c) MaxEmbed solution: partition the hypergraph as there's no replication, then make replication based on the connectivity of the hyperedge and partition result.

of a query. Based on this, we calculate a score for each vertex by the following equation:

$$score(v_i) = \sum_{j \in related\_edge(v_i)} (\lambda(j) - 1)$$

where v_i represent a vertex, $related\_edge(v_i)$ is the set of hyperedges connect to vertex v_i, $\lambda(j)$ represents the connectivity of hyperedge j.

This equation considers both the contribution of vertices to the connectivity in hyperedges and the hotness of the embedding vertices. Vertices with higher scores indicate more connected clusters linked by their adjacent hyperedges and appear more frequently.

Identifying which vertices to replicate is a critical yet insufficient step. High-scoring vertices might not have strong relationships with each other, and their combination could fail to bring significant benefits. To address this challenge, we incorporate the hypergraph partition results and the co-appearance information of the chosen vertices into the replication process. For vertices with higher scores, we examine the frequency of other vertices appearing alongside them in a hyperedge. When forming a replica cluster, we select a high-scored vertex as a base, assess the co-occurrence frequency of other vertices with the base, and aggregate them. This approach mitigates the issues of the weak correlation of vertices in selecting vertex and making replication, thereby improving the overall effectiveness of the replication process.

Our replication algorithm consists of the following steps:

1. Partition the hypergraph using vanilla SHP algorithm.
2. Calculate the score for each vertex (using equation(1)).
3. Select top rN/d scored vertices (with d representing the embedding count a cluster can accommodate).
4. For each selected vertex, find the most frequent $(d-1)$ neighbors by traversing the hyperedges connecting to it (excluding vertex that has been assigned to the same cluster in the first step) and create a replica cluster for these vertices.

Adding replicas after partition retains the result of the original hypergraph partition, which ensures that the replication operation will not damage the existing combination relationship. In selecting vertices for replication, we consider a vertex's contributions to hyperedge connectivity and its significance (hotness) and extend replication to vertices adjacent to the selected ones. This approach prevents issues related to weak correlations between selected vertices. By replicating these selected vertices and their adjacent counterparts, we can effectively diminish edge connectivity, contributing to a more streamlined and optimized system.

6 Online Phase: Query Request Processing

After partition and replication, we obtained the placement of the embeddings. Since one embedding may be stored in multiple SSD pages, MaxEmbed faces the page selection problem, i.e., choosing which pages to read to minimize the whole read operation count. This is known as the set cover optimization problem, a classic NP-hard problem [6].

Most current approximation algorithms for the set cover problem rely on greedy approaches. MaxEmbed leverages a well-known and near-optimal greedy algorithm [19, 34]. Specifically, in each step, it selects the embedding page that can serve the most requests until all the requests are covered. To achieve this, it iterates all the embedding pages, intersecting them with the current request set. Therefore, it has a high complexity (in worst case $O(|S| \cdot |Q|)$ set operations, where Q means the query set and S contains all the possible sets that can be selected), adding severe computing overhead. In our evaluation, the procedure of greedy selection accounts for over 56% of the end-to-end latency, even more than reading SSD itself.

However, we find that this procedure can be optimized because of the following two key observations in our scenario. 1) Replicated, hot embeddings are likely to co-locate with non-replicated, cold embeddings, thus hitchhiking on their only candidate page. 2) The page selection process can be pipelined with SSD reads, hiding the computing overhead.

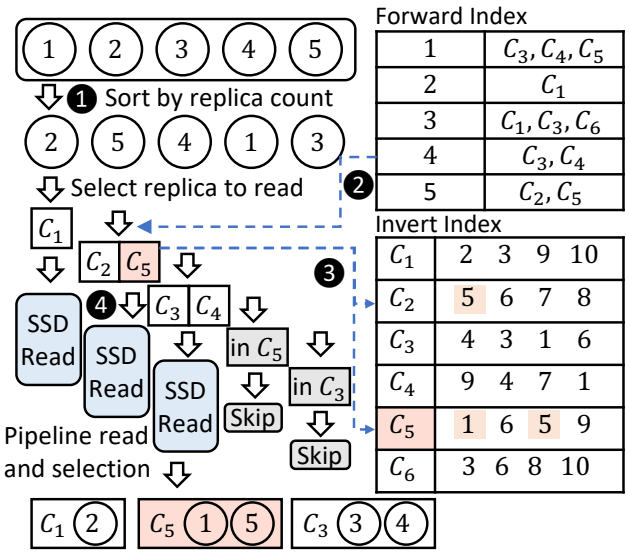

Figure 6. Replica selection and SSD read pipeline. Take the selection of embedding 5 (E5) as an example: ❶ All queried embeddings are sorted by their replica count from low to high. ❷ From the Forward index, E5 exists in page C_2 and C_5. ❸ Match the Invert Index of C_2 and C_5 and find C_5 covers the most embeddings in the rest of the query (E1, E5). ❹ Issue SSD request of C_5.

6.1 One pass selection algorithm

With the first observation, we modify the algorithm above, as depicted in Figure 6.

Two indexes should be initialized for $O(1)$ embedding-to-pages and page-to-embeddings mapping. The Forward Index stores the SSD pages each embedding is in, and the Invert Index stores the embedding keys each SSD page contains.

To iterate the embeddings with fewer replicas first, ❶ we sort the embedding keys in ascending order based on their replica counts. After that, each key (which has not been removed) is iterated. ❷ Select the SSD pages that contain the target key using the Forward Index. Then, ❸ find the page that covers the most queried embeddings with the Invert Index. Finally, we remove the covered keys from the query and ❹ issue an SSD page read operation.

As embeddings with fewer replicas are first iterated, the computing overhead will be much lower, and embeddings with more replicas are likely to be read together with preceding embeddings, eliminating high access overheads. In addition, it reduces the complexity to $O(|S| + |Q|)$ set operations for each query due to the fact that each SSD page will be enumerated $O(1)$ times. (See §7.2 for more discussions.)

Index shrinking. If highly-replicated embeddings are unfortunately not covered by the read of previous low-replicated embeddings, they may cause high computing overhead in ❸, since this often results in the searching of hundreds of clusters on the Invert Index. To address this, we further propose the index shrinking optimization to limit the number of clusters for all embeddings in the Embedding Index. As shown in Figure 7, in the index, we only store and enumerate the first k (noted as *index limit*, $k = 3$ in this case) clusters instead of all clusters. With index shrinking, we not only reduce the size of the Embedding Index but also limit the index searching overhead in ❸ to k reads in the worst case. Note that index shrinking has negligible impact on the SSD effective bandwidth in practice since, at this point, most of the co-appear patterns in this query have already been read.

6.2 Pipelined replica selection and SSD access

Despite the optimization above, the overhead is still considerably high. However, as Figure 6 indicates, an SSD read is independent of the subsequent page selections. Additionally, we observe that replica selection and SSD read two procedures have comparable order of magnitude of latency (§8.4). These insights enable us to leverage the pipeline technique to further hide the software overhead. Specifically, after selecting the SSD page to read in each iteration, we issue an asynchronous SSD read request and proceed with the following page selections. At the end of our algorithm, we poll for all SSD reads to complete.

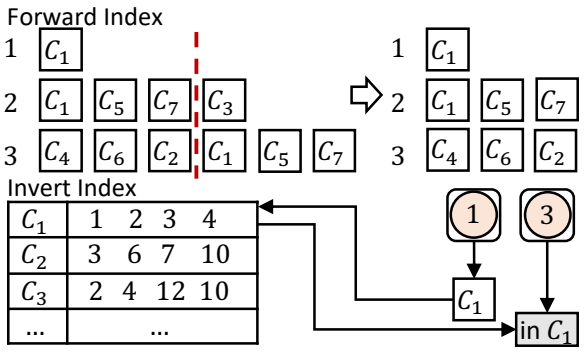

Figure 7. Index shrinking. Although the index of vertex 3 has been shrunk, and there is no C_1 in the index of vertex 3, due to the existence of the invert index when querying vertices $\langle 1, 3\rangle$, these two embeddings can still be obtained in one read operation.

7 Overhead of MaxEmbed

7.1 Memory overhead

MaxEmbed has to maintain two DRAM-resident indexes, which introduce additional storage overhead: 1) Embedding index that maps each embedding vector to its positions in SSD. This requires a space of $O((1+r)N)$, where r is the replication ratio. 2) Inverted index that maps each SSD page to the embedding keys it contains. The mapping size equals the SSD page count, a linear relationship with the total number of embeddings. It is $O((1+r)N)$. Since the replication ratio is usually a tiny constant, it can be considered the same magnitude as $O(1)$. The total space overhead is $O(N)$.

7.2 Time overhead

Offline partition cost. Let N be the number of vertices, E be the number of edges, r be the replication ratio, and d be the embedding count an SSD page holds (determined by the SSD page size and the embedding dimension). The time cost of the offline process contains two parts: 1) partitioning a hypergraph using SHP, which incurs a time complexity of $O(E \log B)$ [20], with $B = N/d$ representing the total number of partitions. As d can be considered as a constant, the complexity can be simplified as $O(E \log N)$. 2) The replication procedure requires $O(rNE)$ time cost, as it iterates $rN/d = O(rN)$ rounds, with each round enumerating $O(E)$ edges in the worst case.

The original SHP implementation was designed on Hadoop. Both this algorithm and our extension to this algorithm can be seamlessly transformed into a map-reduce program, allowing for straightforward parallelization, which will significantly reduce the time cost. Since hypergraph partitioning is offline, the time overhead is acceptable.

Table 1 shows the partition time (with a replication ratio of 10%) in Criteo and CriteoTB datasets.

Table 1. Partition time of real-world datasets.

Dataset	16 in a part	32 in a part	64 in a part
Criteo	5 min	4.9 min	4.8 min
CriteoTB	3 hour	2.8 hour	2.7 hour

Online query process cost. Suppose a request contains q embedding keys. According to §6, the time overhead of online processing consists of two parts: 1) sort the request in replica count ascendant order. The complexity of the sort is $O(q \log q)$. 2) Match all SSD pages corresponding to embedding keys and select a page that covers the maximum number of elements in the current request. With the aforementioned optimization, the complexity of this process becomes $O(kq)$, where k represents the *index limit* of the Embedding Index. Here, the *index limit* controls the upper bound of candidate SSD pages to kq, which gives a more precise complexity than §6.1. This process can be pipelined with SSD read operations to cover the overhead further. The total time complexity is $O(q \log q)$. We have examined the influence of k on latency, which is detailed in §8.4.

7.3 Total Cost of Ownership (TCO) of MaxEmbed

The main cost of MaxEmbed comes from the additional space of SSD. We use the price of AWS[1], Intel P5800X (Optane-based) and Samsung PM 1735 (NAND-based) to estimate the TCO of MaxEmbed. Taking the largest publicly available dataset (CriteoTB) as an example, its embedding table size is about 225 GB. At a replication ratio of 0.8×, the size is about 400 GB. A c6g.16xlarge instance costs $1,588 per month, an 800GB P5800X drive costs around $1,000 ($1.25/GB), and a 1.6TB PM1735 drive costs approximately $500 ($0.3125/GB). The TCO estimation is shown in Table 2. The assumption is PM 1735 gives the same perf as Intel Optane P5800X as it's not measured.

Table 2. TCO estimation of MaxEmbed.

Item	Baseline (SHP)	MaxEmbed (with r=80%)
Total Cost (with P5800X)	$1,869.25	$2,088.00
Total Cost (with PM1735)	$1,658.31	$1,713.00
Performance	1×	1.16×
Performance/Cost (P5800X)	1×	1.04×
Performance/Cost (PM1735)	1×	1.12×

8 Evaluation

8.1 Experiment setup

By default, we conducted all our experiments on a machine with Intel(R) Xeon(R) GOLD 6530 CPU processor, 128G Memory, and Intel Optane P5800X SSD. All evaluations were conducted on Ubuntu 22.04 LTS. All SSD requests are issued through the user space driver, SPDK[2]. We use CacheLib [5], a high-performance concurrent cache proposed by Meta, as DRAM cache to evaluate all our baseline and MaxEmbed. CacheLib introduces very low memory overhead, making it ideal for caching gigabytes of objects such as large embedding tables, and previous DLRM-related research by Meta [4] also use CacheLib as DRAM cache. We use CacheLib's LRU cache and the default insertion and eviction policy (updateOnRead, but not updateOnWrite), which is a configuration suitable for read-intensive workloads. Except for special instructions, the cache ratio is set to 10% in our evaluation.

Real-world datasets. To evaluate the effectiveness of MaxEmbed, we use public datasets for the recommendation systems: Amazon M2 [33], Criteo [17], Avazu [36], Alibaba-iFashion [8], and CriteoTB [22] datasets, as detailed in Table 3. Among these, Avazu, Criteo, and CriteoTB are datasets used predominantly in advertising scenarios, while Alibaba-iFashion and Amazon M2 are more pertinent to shopping scenarios.

Table 3. Real-world Datasets information.

Dataset	# of Items	# of Queries	Query Len.	Log Size
Amazon M2	1.39M	3.6M	5.24	0.26 GB
Alibaba-iFashion	4.46M	999K	53.63	1.4 GB
Avazu	9.45M	40.4M	21	5.3 GB
Criteo	35M	45.8M	26	9.5 GB
CriteoTB	882M	4.37B	26	1.1 TB

We set the default embedding vectors dimension to be 64 (256 bytes) and evaluate other embedding dimensions (from 32 to 128) in sensitive analysis §8.5.

8.2 Overall performance

Effective bandwidth improvement. Figure 8 shows SSD effective bandwidth improvement of MaxEmbed across various replication ratios (0.1, 0.2, 0.4, 0.8) and datasets. The term "SHP" represents the baseline metric. Each bar labeled "ME(r=x%)" demonstrates the effective bandwidth increment when MaxEmbed is applied for embedding placement with an x% replication rate.

It's noteworthy that the performance gains are particularly pronounced in the context of shopping datasets, where the co-appearance phenomenon is more prominent. This enhancement, however, is less significant in advertising datasets.

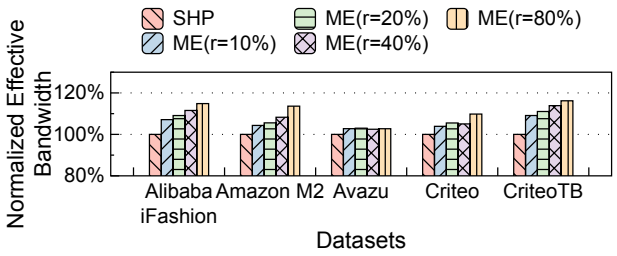

Figure 8. Effective bandwidth under different replication ratios.

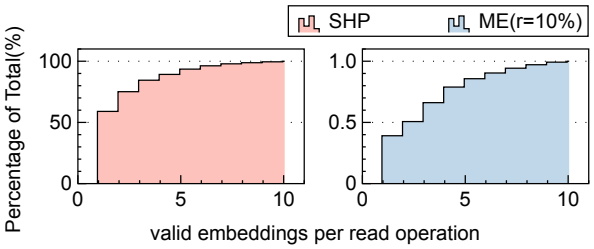

Figure 9. Cumulative Distribution Function (CDF) of valid embeddings per read operation.

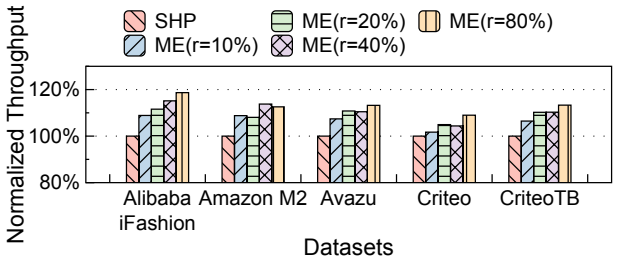

Figure 10. Throughput of end-to-end evaluation.

This variance can also be attributed to dataset size. Larger datasets provide more sample data for hypergraph partitioning and replica selection, enabling more precise embedding placements.

Explore the valid embedding count in a read. We leverage the Criteo dataset to examine how the replicas impact effective read bandwidth. We use MaxEmbed to perform partitioning and replication operations with a ratio of 10% and count the average number of valid embedding counts in a single read operation (without cache). As illustrated in Figure 9, the situation where only one valid embedding is obtained in a single read operation is significantly reduced, and other situations where more valid embeddings are obtained increase. The average valid embedding obtained by each read operation increased from 3.59 to 4.79. This shift results in a marked improvement in the efficiency of read operations.

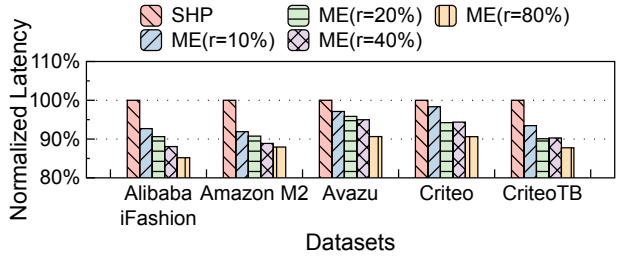

Figure 11. Latency of end-to-end evaluation.

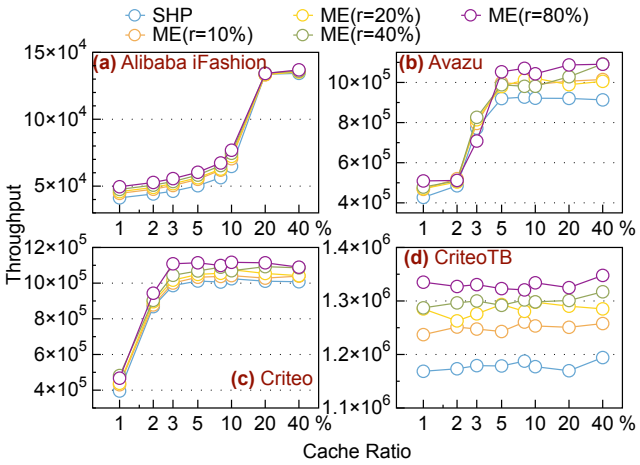

Figure 12. Throughput under different cache ratios.

End-to-end performance. Figure 10 shows the end-to-end throughput results. The changes in end-to-end throughput are slightly different from the changes in effective bandwidth, mainly due to the different average request lengths for different datasets. Putting multiple batches of queries simultaneously may cause duplication, and the effective bandwidth and throughput after deduplication are not necessarily proportional. The throughput improves 1.7%-8.88% across the datasets in a 10% replication ratio and reaches 8.9% - 18.7% when the replication ratio increases to 80% compared to the baseline.

Figure 11 shows the end-to-end latency results. After adding replicas, we can read fewer SSD pages with one query, which reduces the end-to-end performance's read latency. Since the number of times a single embedding query needs to read the SSD is reduced, the latency is also significantly reduced. With a replication ratio of 10%, the end-to-end latency is reduced by 2%-7.4%. When the replication ratio comes to 80%, the latency is reduced by 10%-14.8%.

8.3 Performance under different cache ratios

Figure 12 shows the end-to-end throughput under different cache ratios. We set the cache size to $1-40\%$ of the whole embedding table size. We can observe that: 1) as the cache size increases, the throughput of different configurations gradually increases and eventually remains stable. 2) In all the evaluated datasets, MaxEmbed can bring up to 1.2× throughput improvement under different cache ratios. This is due to the fact that even though the cache absorbs access to hot embeddings, MaxEmbed can still utilize the combination relationship of cold embeddings to reduce read amplification of SSDs. 3) Since the combination relationships of different datasets contain different numbers of hotspots, the performance improvement that MaxEmbed can achieve varies with different caching ratios. For example, in CriteoTB, the combination relationships are colder in comparison with other datasets, so MaxEmbed is less sensitive to the cache size, and thus increasing r can improve performance significantly.

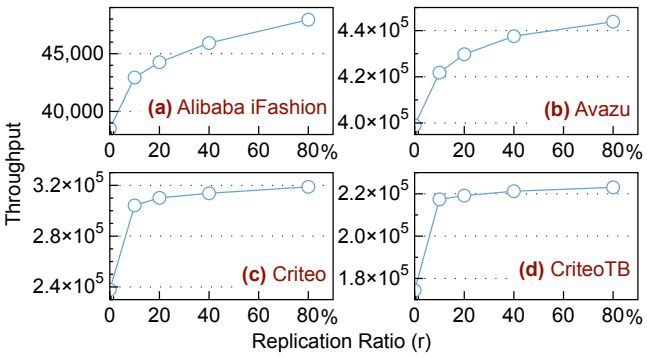

Figure 13. Throughput of end-to-end evaluation without cache.

Considering there are scenarios where using DRAM as an embedding cache is impractical, such as near-data processing [37, 41], we also benchmark MaxEmbed without cache. As shown in Figure 13, the performance of MaxEmbed is more pronounced in this case, with just a small r (e.g., 0.2), MaxEmbed can improve the throughput by $1.08-1.31\times$. We also evaluate a pure DRAM system. Its throughput is 9-26 × of MaxEmbed (omitted in the figure).

8.4 Technique analysis

Comparison of different replication strategies. Figure 14 shows the bandwidth improvement of three algorithms mentioned in §5 on Alibaba-iFashion, Amazon M2, and Avazu datasets (the performance of the three replication strategies on the Criteo and CriteoTB datasets are similar to that on Avazu). RPP can maintain a relatively stable increase in effective bandwidth, but the overall improvement is slight. The main reason is that RPP performs replicas before the graph is divided, and there may not necessarily be a combination relationship between the selected replicas, resulting in wasted space but no benefit.

The FPR does not perform stably on different datasets. It can be found that FPR has achieved good results only on the Amazon M2 dataset. However, it receives poor results

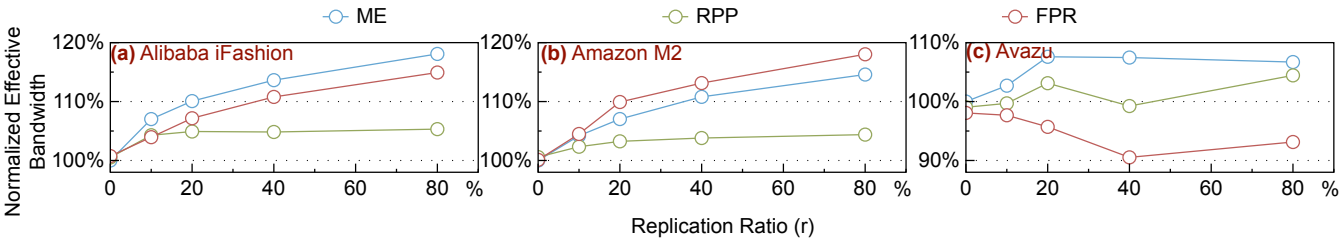

Figure 14. Comparison of different replica strategies.

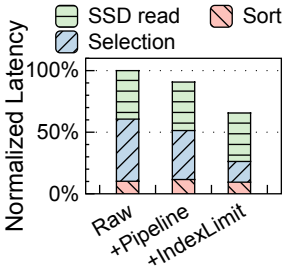

Figure 15. Time breakdown of an online query.

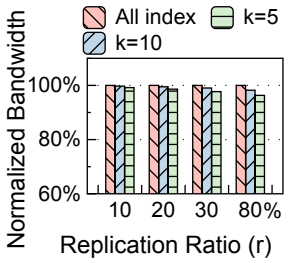

Figure 16. Impact of index shrinking.

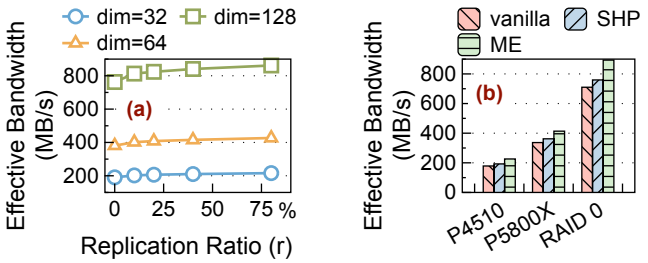

Figure 17. Sensitivity analysis. (a) different embedding dimensions, (b) different SSD types.

on other datasets and may even cause a decrease in effective bandwidth. To understand the reason, we note that there is an apparent difference in average request length between Amazon M2 and other datasets. As mentioned in §5.2, finer partitions may destroy the combination of embeddings. However, in the Amazon M2 dataset, the average request length is shorter, causing smaller hyperedges. In this way, even if the hypergraph is divided into thinner parts, it will not affect the graph partition. Therefore, FPR performs well on this particular dataset, but its performance on other datasets is relatively poor.

MaxEmbed keeps a stable and relatively high improvement of the effective bandwidth on all datasets. This is mainly due to MaxEmbed taking into account the contribution of a vertex to the number of hyperedge-connected blocks, its popularity, and its neighboring vertices when making replication.

Time breakdown of an online query.

As mentioned in §6, online query processing with embedding replication incurs significant overhead. We adopted the pipeline and index shrinking to reduce the overhead. We break down the MaxEmbed online procedure by running with 8 threads to demonstrate the effectiveness of these optimizations. The partitioned Alibaba-iFashion dataset with a 40% replication ratio is used. Figure 15 shows the latency breakdown of the online processing procedure. After using the pipeline strategy, the overhead of request processing is reduced by 10.23%. With setting the *index limit* (k) to 5 in addition to the pipeline, the overhead is reduced by 34.4% compared to the one without any optimization. With these optimizations, the selection procedure occupies less than 25% CPU of the entire SSD operation procedure.

Impact of index shrinking. Figure 16 depicts the influence of varying the *index limit* (denoted as k) on effective bandwidth, utilizing the Alibaba-iFashion dataset. As discussed in §6.1, we strategically reduce the index of an embedding, limiting it to only k entries. This reduction serves to alleviate memory overhead and replication selection overhead. Although the index is shrunk, the impact on effective bandwidth remains remarkably marginal. By constraining storing 10 indexes for each key, MaxEmbed achieves an effective bandwidth exceeding 98% of storing and retrieving all indexes in an 80% replication ratio. By constraining storing just 5 indexes for each key, MaxEmbed still achieves an effective bandwidth exceeding 96% of the one without optimization in an 80% replication ratio.

8.5 Sensitivity analysis

Impact of embedding vector dimensions. The dimension of the embedding vector determines how many embeddings can be placed on a single SSD page, and different numbers of embeddings on an SSD page will significantly affect the possible combination of embeddings and the effective bandwidth. Figure 17 (a) explores MaxEmbed's sensitivity of embedding sizes with the Alibaba-iFashion dataset. In all embedding vector sizes, the effective bandwidth increases as the replication ratio increases. The larger the size of the embedding vector, the worse the SHP effect without replication. This is because when the embedding vector size increases, there are fewer embedding vectors a single SSD page can accommodate and

fewer combinations a single embedding can produce with other embeddings. And the relative effect will be better after making replication.

Sensitivity to SSD types. Figure 17 (b) illustrates the influence of SSD types on MaxEmbed using the AlibabaiFashion dataset. The effective bandwidth improvement demonstrates consistent results across Intel Optane P5800X SSD, Intel P4510 SSD, and RAID 0 consisting of 2 P5800X SSDs. The major difference in the results owes to the total bandwidth of these SSDs. Therefore, the type of SSDs has minimal impact on MaxEmbed.

9 Related Work

Minimize request size with hypergraph partition. Bandana [11] suggests using past access patterns of the embedding vectors to aggregate adjacent embedding vectors with a hypergraph partition algorithm. Bandana uses SHP [20] to identify and cluster frequently co-appearing embedding vectors. However, the partition algorithm restricts each embedding to only appearing in one cluster, causing limited effective bandwidth improvement.

Based on SHP, we add a replication strategy to the hypergraph partition algorithm to effectively create more embedding combinations to further increase the possibility of embedding co-appearance in an SSD page. Correspondingly, we also designed an online service module for embedding query serving with replication.

Reduce reduction overhead via sub-query memoization. Merci [23] introduces a mechanism for memoizing correlated embeddings to mitigate the overhead associated with reduction operations and memory access bottlenecks. Merci also uses hypergraph partition tools to aggregate related embeddings and uses additional memory space to store the results of embedding reduction to alleviate the memory bandwidth-bound problem. GRACE [43] also leverages additional memory to store embedding reduction results. GRACE proposes a sys-aware clustering algorithm to find embedding item co-occurrences to get a better clustering result.

Different from the SSD used by MaxEmbed, memory has no reading and writing amplification problem. The mechanism of Merci and GRACE is designed to reduce the expenses of aggregation operations for embeddings.

Customized SSD for recommendation system embedding storage. RecSSD [41] proposes a near-data processing solution for neural recommendation inference. RecSSD reduces the total traffic on PCI-e and fully utilizes the SSD internal bandwidth by offloading computations for key embedding table operations to SSD hardware. FlashEmbedding [38] proposes a hardware/software co-design embedding storage system, which incorporates an embedding sematic-aware SSD. By using a light weight I/O stack and supporting fine-grained access, FlashEmbedding effectively reduces read amplification.

These studies require modification of the SSD internal mechanism, while MaxEmbed is designed to reduce read amplification brought by read granularity mismatch and is suitable for general SSDs.

10 Conclusion

In this paper, we propose MaxEmbed, a replicated hypergraph partition-based SSD embedding storage and retrieval system. Using SSD to store small-sized embedding parameters will encounter severe read amplification problems. Existing systems use hypergraph partition to find potential combination relationships between embeddings to alleviate the problem but ignore the shortcomings of hypergraph partitioning in this problem. MaxEmbed uses replication to construct more possible combinations between embeddings to co-locate more embeddings together to improve the bandwidth of SSD embedding. Meanwhile, for the request preprocessing problem caused by the replication, MaxEmbed adopts a heuristic selection strategy, effectively avoiding software overhead. The evaluation shows that MaxEmbed boosts the SSD effective bandwidth up to 1.19×.

Acknowledgments

We sincerely thank our shepherd Vikram Sharma Mailthody and anonymous reviewers for their valuable feedback, which greatly improved the paper. This work is supported by the National Key R&D Program of China (Grant No. 2021YFB0300500), the National Natural Science Foundation of China (Grant No. 62332011), and the Huawei-Tsinghua University joint program on embedding retrieval and management.

A Artifact Appendix

A.1 Abstract

This appendix describes the workflow of MaxEmbed, including code, evaluation scripts, and datasets used for evaluations. The source code of MaxEmbed is available at: https://github.com/Ksitta/MaxEmbed.

A.2 Description

A.2.1 How to access

- Download source code from https://github.com/Ksitta/MaxEmbed.

A.2.2 Hardware dependencies

- Intel CPU (for Intel TBB)
- NVMe SSD (for SPDK)
- x86-64 Architechture

A.2.3 Software dependencies

- System: Ubuntu 22.04.4 LTS
- CMake
- SPDK

- vcpkg: We use vcpkg to install the following dependencies, vcpkg can be downloaded from https://github.com/microsoft/vcpkg
  - CacheLib
  - Folly
  - TBB
  - atomic-queue
  - argparse

A.2.4 Data sets

- Amazon M2 [33]
- Criteo [17]
- Avazu [36]
- Alibaba-iFashion [8]
- CriteoTB [22]

A.3 Installation

Firstly, install CMake, SPDK, and vcpkg. Follow these steps:

- Run cmake –preset=default in the source directory of MaxEmbed.
- Run make in the generated build folder.

A.4 Server

We provide a server with processed datasets and partitioned results for AE reviewers. We will provide the method of connecting to the server through HotCRP later.

A.5 Evaluation workflow

A.5.1 Major Claims. The paper presents the following major claims in the evaluation section:

1. Effective Bandwidth Improvement: MaxEmbed enhances the effective bandwidth for reading embeddings from SSDs.
2. End-to-End Throughput Improvement: MaxEmbed increases the end-to-end throughput for embedding serving.
3. End-to-End Latency Reduction: MaxEmbed significantly reduces the end-to-end latency for embedding serving.

A.5.2 Experiments. The scripts under ae_scripts folder are provided to reproduce the results for Figure 8,9, 10, 11, 13 and 14. Run run_all.sh <log_path> script, all figures will be generated at <log_path>/figures. If no parameters are specified, the script help will be displayed.

We also provide log files that can be used to draw figures directly, please follow the script instructions and use it directly.

To run several of the experiments, use run_all.sh <log_path> x, where x is a number from 1 to 4, corresponding to the following four experiments:

- **Experiment 1 (about 2.5 hours):**
  Run Command bash run_all.sh <log_path> 1
  MaxEmbed under different cache ratio. Run MaxEmbed online procedure under different cache ratios and replication ratios to check the effectiveness of replicas.
  Figure 8, 10, 11 will be generated at <log_path>/figures.
- **Experiment 2 (about 0.75 hours):**
  Run Command bash run_all.sh <log_path> 2
  MaxEmbed with no DRAM cache. Run MaxEmbed online procedure to check the throughput improvement under different replication ratios with no DRAM cache to show the effectiveness of MaxEmbed when applying to a cacheless embedding serving scenario.
  Then, Figure 13 will be generated at <log_path>/figures.
- **Experiment 3 (about 0.2 hours):**
  Run Command bash run_all.sh <log_path> 3
  MaxEmbed with different replication algorithm. Run MaxEmbed online procedure to check the difference between the selected replication method and the other two methods.
  Figure 14 will be generated at <log_path>/figures.
- **Experiment 4 (about 0.1 hours):**
  Run Command bash run_all.sh <log_path> 4
  Explore the valid embedding count in a read. Run MaxEmbed online procedure to compare two different embedding placements (no replication vs. 10% replication) and explore how many valid embeddings can be obtained from a single read operation.
  Figure 9 will be generated at <log_path>/figures.

References

[1] 2023. Storage Performance Development Kit. https://spdk.io/
[2] 2024. Amazon EC2 Pricing. https://aws.amazon.com/ec2/pricing/?nc1=h_ls
[3] Ehsan K. Ardestani, Changkyu Kim, Seung Jae Lee, Luoshang Pan, Jens Axboe, Valmiki Rampersad, Banit Agrawal, Fuxun Yu, Ansha Yu, Trung Le, Hector Yuen, Dheevatsa Mudigere, Shishir Juluri, Akshat Nanda, Manoj Wodekar, Krishnakumar Nair, Maxim Naumov, Chris Petersen, Mikhail Smelyanskiy, and Vijay Rao. 2022. Supporting Massive DLRM Inference through Software Defined Memory. In *42nd IEEE International Conference on Distributed Computing Systems, ICDCS 2022, Bologna, Italy, July 10-13, 2022*. IEEE, 302–312. https://doi.org/10.1109/ICDCS54860.2022.00037
[4] Ehsan K. Ardestani, Changkyu Kim, Seung Jae Lee, Luoshang Pan, Jens Axboe, Valmiki Rampersad, Banit Agrawal, Fuxun Yu, Ansha Yu, Trung Le, Hector Yuen, Dheevatsa Mudigere, Shishir Juluri, Akshat Nanda, Manoj Wodekar, Krishnakumar Nair, Maxim Naumov, Chris Petersen, Mikhail Smelyanskiy, and Vijay Rao. 2022. Supporting Massive DLRM Inference through Software Defined Memory. In *42nd IEEE International Conference on Distributed Computing Systems, ICDCS 2022, Bologna, Italy, July 10-13, 2022*. IEEE, 302–312. https://doi.org/10.1109/ICDCS54860.2022.00037
[5] Benjamin Berg, Daniel S. Berger, Sara McAllister, Isaac Grosof, Sathya Gunasekar, Jimmy Lu, Michael Uhlar, Jim Carrig, Nathan Beckmann, Mor Harchol-Balter, and Gregory R. Ganger. 2020. The CacheLib Caching Engine: Design and Experiences at Scale. In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020*. USENIX Association, 753–768. https://www.usenix.org/conference/osdi20/presentation/berg

[6] Korte Bernhard and Jens Vygen. 2008. Combinatorial optimization: Theory and algorithms. *Springer, Third Edition, 2005.* (2008).

[7] Ümit V Çatalyürek and Cevdet Aykanat. 2011. Patoh (partitioning tool for hypergraphs). In *Encyclopedia of parallel computing*. Springer, 1479–1487.

[8] Wen Chen, Pipei Huang, Jiaming Xu, Xin Guo, Cheng Guo, Fei Sun, Chao Li, Andreas Pfadler, Huan Zhao, and Binqiang Zhao. 2019. POG: Personalized Outfit Generation for Fashion Recommendation at Alibaba iFashion. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, KDD 2019, Anchorage, AK, USA, August 4-8, 2019*, Ankur Teredesai, Vipin Kumar, Ying Li, Rómer Rosales, Evimaria Terzi, and George Karypis (Eds.). ACM, 2662–2670. https://doi.org/10.1145/3292500.3330652

[9] Heng-Tze Cheng, Levent Koc, Jeremiah Harmsen, Tal Shaked, Tushar Chandra, Hrishi Aradhye, Glen Anderson, Greg Corrado, Wei Chai, Mustafa Ispir, Rohan Anil, Zakaria Haque, Lichan Hong, Vihan Jain, Xiaobing Liu, and Hemal Shah. 2016. Wide & Deep Learning for Recommender Systems. In *Proceedings of the 1st Workshop on Deep Learning for Recommender Systems, DLRS@RecSys 2016, Boston, MA, USA, September 15, 2016*, Alexandros Karatzoglou, Balázs Hidasi, Domonkos Tikk, Oren Sar Shalom, Haggai Roitman, Bracha Shapira, and Lior Rokach (Eds.). ACM, 7–10. https://doi.org/10.1145/2988450.2988454

[10] Intel Corporation. 2023. Intel® Optane™ SSD P5800X series specifications. https://www.intel.com/content/www/us/en/products/docs/memory-storage/solid-state-drives/data-center-ssds/optane-ssd-p5800x-p5801x-brief.html

[11] Assaf Eisenman, Maxim Naumov, Darryl Gardner, Misha Smelyanskiy, Sergey Pupyrev, Kim M. Hazelwood, Asaf Cidon, and Sachin Katti. 2019. Bandana: Using Non-Volatile Memory for Storing Deep Learning Models. In *Proceedings of Machine Learning and Systems 2019, MLSys 2019, Stanford, CA, USA, March 31 - April 2, 2019*, Ameet Talwalkar, Virginia Smith, and Matei Zaharia (Eds.). mlsys.org. https://proceedings.mlsys.org/book/277.pdf

[12] Udit Gupta, Carole-Jean Wu, Xiaodong Wang, Maxim Naumov, Brandon Reagen, David Brooks, Bradford Cottel, Kim M. Hazelwood, Mark Hempstead, Bill Jia, Hsien-Hsin S. Lee, Andrey Malevich, Dheevatsa Mudigere, Mikhail Smelyanskiy, Liang Xiong, and Xuan Zhang. 2020. The Architectural Implications of Facebook's DNN-Based Personalized Recommendation. In *IEEE International Symposium on High Performance Computer Architecture, HPCA 2020, San Diego, CA, USA, February 22-26, 2020*. IEEE, 488–501. https://doi.org/10.1109/HPCA47549.2020.00047

[13] Xiangnan He, Lizi Liao, Hanwang Zhang, Liqiang Nie, Xia Hu, and Tat-Seng Chua. 2017. Neural Collaborative Filtering. *CoRR* abs/1708.05031 (2017). arXiv:1708.05031 http://arxiv.org/abs/1708.05031

[14] Joel Hestness, Newsha Ardalani, and Gregory F. Diamos. 2019. Beyond human-level accuracy: computational challenges in deep learning. In *Proceedings of the 24th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2019, Washington, DC, USA, February 16-20, 2019*, Jeffrey K. Hollingsworth and Idit Keidar (Eds.). ACM, 1–14. https://doi.org/10.1145/3293883.3295710

[15] Joel Hestness, Sharan Narang, Newsha Ardalani, Gregory F. Diamos, Heewoo Jun, Hassan Kianinejad, Md. Mostofa Ali Patwary, Yang Yang, and Yanqi Zhou. 2017. Deep Learning Scaling is Predictable, Empirically. *CoRR* abs/1712.00409 (2017). arXiv:1712.00409 http://arxiv.org/abs/1712.00409

[16] Po-Sen Huang, Xiaodong He, Jianfeng Gao, Li Deng, Alex Acero, and Larry P. Heck. 2013. Learning deep structured semantic models for web search using clickthrough data. In *22nd ACM International Conference on Information and Knowledge Management, CIKM'13, San Francisco, CA, USA, October 27 - November 1, 2013*, Qi He, Arun Iyengar, Wolfgang Nejdl, Jian Pei, and Rajeev Rastogi (Eds.). ACM, 2333–2338. https://doi.org/10.1145/2505515.2505665

[17] Olivier Chapelle Jean-Baptiste Tien, joycenv. 2014. Display Advertising Challenge. https://kaggle.com/competitions/criteo-display-ad-challenge

[18] Wenqi Jiang, Zhenhao He, Shuai Zhang, Thomas B. Preußer, Kai Zeng, Liang Feng, Jiansong Zhang, Tongxuan Liu, Yong Li, Jingren Zhou, Ce Zhang, and Gustavo Alonso. 2021. MicroRec: Efficient Recommendation Inference by Hardware and Data Structure Solutions. In *Proceedings of Machine Learning and Systems 2021, MLSys 2021, virtual, April 5-9, 2021*, Alex Smola, Alex Dimakis, and Ion Stoica (Eds.). mlsys.org. https://proceedings.mlsys.org/paper/2021/hash/ec8956637a99787bd197eacd77acce5e-Abstract.html

[19] David S. Johnson. 1974. Approximation Algorithms for Combinatorial Problems. *J. Comput. Syst. Sci.* 9, 3 (1974), 256–278. https://doi.org/10.1016/S0022-0000(74)80044-9

[20] Igor Kabiljo, Brian Karrer, Mayank Pundir, Sergey Pupyrev, Alon Shalita, Yaroslav Akhremtsev, and Alessandro Presta. 2017. Social Hash Partitioner: A Scalable Distributed Hypergraph Partitioner. *Proc. VLDB Endow.* 10, 11 (2017), 1418–1429. https://doi.org/10.14778/3137628.3137650

[21] Daniar Heri Kurniawan, Ruipu Wang, Kahfi S. Zulkifli, Fandi A. Wiranata, John Bent, Ymir Vigfusson, and Haryadi S. Gunawi. 2023. EVStore: Storage and Caching Capabilities for Scaling Embedding Tables in Deep Recommendation Systems. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, ASPLOS 2023, Vancouver, BC, Canada, March 25-29, 2023*, Tor M. Aamodt, Natalie D. Enright Jerger, and Michael M. Swift (Eds.). ACM, 281–294. https://doi.org/10.1145/3575693.3575718

[22] Criteo AI Lab. 2023. Download Criteo 1TB Click Logs dataset. https://ailab.criteo.com/download-criteo-1tb-click-logs-dataset/

[23] Yejin Lee, Seong Hoon Seo, Hyunji Choi, Hyoung Uk Sul, Soosung Kim, Jae W. Lee, and Tae Jun Ham. 2021. MERCI: efficient embedding reduction on commodity hardware via sub-query memoization. In *ASPLOS '21: 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Virtual Event, USA, April 19-23, 2021*, Tim Sherwood, Emery D. Berger, and Christos Kozyrakis (Eds.). ACM, 302–313. https://doi.org/10.1145/3445814.3446717

[24] Thomas Lengauer. 2012. *Combinatorial algorithms for integrated circuit layout.* Springer Science & Business Media.

[25] Jianxun Lian, Xiaohuan Zhou, Fuzheng Zhang, Zhongxia Chen, Xing Xie, and Guangzhong Sun. 2018. xDeepFM: Combining Explicit and Implicit Feature Interactions for Recommender Systems. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, KDD 2018, London, UK, August 19-23, 2018*, Yike Guo and Faisal Farooq (Eds.). ACM, 1754–1763. https://doi.org/10.1145/3219819.3220023

[26] Xiangru Lian, Binhang Yuan, Xuefeng Zhu, Yulong Wang, Yongjun He, Honghuan Wu, Lei Sun, Haodong Lyu, Chengjun Liu, Xing Dong, Yiqiao Liao, Mingnan Luo, Congfei Zhang, Jingru Xie, Haonan Li, Lei Chen, Renjie Huang, Jianying Lin, Chengchun Shu, Xuezhong Qiu, Zhishan Liu, Dongying Kong, Lei Yuan, Hai Yu, Sen Yang, Ce Zhang, and Ji Liu. 2022. Persia: An Open, Hybrid System Scaling Deep Learning-based Recommenders up to 100 Trillion Parameters. In *KDD '22: The 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, Washington, DC, USA, August 14 - 18, 2022*, Aidong Zhang and Huzefa Rangwala (Eds.). ACM, 3288–3298. https://doi.org/10.1145/3534678.3539070

[27] Weilin Lin, Xiangyu Zhao, Yejing Wang, Tong Xu, and Xian Wu. 2022. AdaFS: Adaptive Feature Selection in Deep Recommender System. In *KDD '22: The 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, Washington, DC, USA, August 14 - 18, 2022*, Aidong Zhang and Huzefa Rangwala (Eds.). ACM, 3309–3317. https://doi.org/10.1145/3534678.3539204

[28] Haochen Liu, Xiangyu Zhao, Chong Wang, Xiaobing Liu, and Jiliang Tang. 2020. Automated Embedding Size Search in Deep Recommender Systems. In *Proceedings of the 43rd International ACM SIGIR conference on research and development in Information Retrieval, SIGIR 2020, Virtual Event, China, July 25-30, 2020*, Jimmy X. Huang, Yi Chang, Xueqi Cheng, Jaap Kamps, Vanessa Murdock, Ji-Rong Wen, and Yiqun Liu (Eds.). ACM, 2307–2316. https://doi.org/10.1145/3397271.3401436

[29] Maxim Naumov, Dheevatsa Mudigere, Hao-Jun Michael Shi, Jianyu Huang, Narayanan Sundaraman, Jongsoo Park, Xiaodong Wang, Udit Gupta, Carole-Jean Wu, Alisson G. Azzolini, Dmytro Dzhulgakov, Andrey Mallevich, Ilia Cherniavskii, Yinghai Lu, Raghuraman Krishnamoorthi, Ansha Yu, Volodymyr Kondratenko, Stephanie Pereira, Xianjie Chen, Wenlin Chen, Vijay Rao, Bill Jia, Liang Xiong, and Misha Smelyanskiy. 2019. Deep Learning Recommendation Model for Personalization and Recommendation Systems. *CoRR* abs/1906.00091 (2019). arXiv:1906.00091 http://arxiv.org/abs/1906.00091

[30] Liang Qu, Yonghong Ye, Ningzhi Tang, Lixin Zhang, Yuhui Shi, and Hongzhi Yin. 2022. Single-shot Embedding Dimension Search in Recommender System. In *SIGIR '22: The 45th International ACM SIGIR Conference on Research and Development in Information Retrieval, Madrid, Spain, July 11 - 15, 2022*, Enrique Amigó, Pablo Castells, Julio Gonzalo, Ben Carterette, J. Shane Culpepper, and Gabriella Kazai (Eds.). ACM, 513–522. https://doi.org/10.1145/3477495.3532060

[31] David Rohde, Stephen Bonner, Travis Dunlop, Flavian Vasile, and Alexandros Karatzoglou. 2018. RecoGym: A Reinforcement Learning Environment for the problem of Product Recommendation in Online Advertising. *CoRR* abs/1808.00720 (2018). arXiv:1808.00720 http://arxiv.org/abs/1808.00720

[32] Sebastian Schlag, Tobias Heuer, Lars Gottesbüren, Yaroslav Akhremtsev, Christian Schulz, and Peter Sanders. 2022. High-Quality Hypergraph Partitioning. *ACM J. Exp. Algorithmics* 27 (2022), 1.9:1–1.9:39. https://doi.org/10.1145/3529090

[33] Amazon Search. 2023. Amazon KDD Cup '23: Multilingual Recommendation Challenge. https://www.aicrowd.com/challenges/amazon-kdd-cup-23-multilingual-recommendation-challenge

[34] Peter Slavík. 1997. A Tight Analysis of the Greedy Algorithm for Set Cover. *J. Algorithms* 25, 2 (1997), 237–254. https://doi.org/10.1006/JAGM.1997.0887

[35] Mohammadreza Soltaniyeh, Veronica Lagrange Moutinho dos Reis, Matthew Bryson, Xuebin Yao, Richard P. Martin, and Santosh Nagarakatte. 2022. Near-Storage Processing for Solid State Drive Based Recommendation Inference with SmartSSDs®. In *ICPE '22: ACM/SPEC International Conference on Performance Engineering, Bejing, China, April 9 - 13, 2022*, Dan Feng, Steffen Becker, Nikolas Herbst, and Philipp Leitner (Eds.). ACM, 177–186. https://doi.org/10.1145/3489525.3511672

[36] Will Cukierski Steve Wang. 2014. Click-Through Rate Prediction. https://kaggle.com/competitions/avazu-ctr-prediction

[37] Xuan Sun, Hu Wan, Qiao Li, Chia-Lin Yang, Tei-Wei Kuo, and Chun Jason Xue. 2022. RM-SSD: In-Storage Computing for Large-Scale Recommendation Inference. In *IEEE International Symposium on High-Performance Computer Architecture, HPCA 2022, Seoul, South Korea, April 2-6, 2022*. IEEE, 1056–1070. https://doi.org/10.1109/HPCA53966.2022.00081

[38] Hu Wan, Xuan Sun, Yufei Cui, Chia-Lin Yang, Tei-Wei Kuo, and Chun Jason Xue. 2021. FlashEmbedding: storing embedding tables in SSD for large-scale recommender systems. In *APSys '21: 12th ACM SIGOPS Asia-Pacific Workshop on Systems, Hong Kong, China, August 24-25, 2021*, Haryadi S. Gunawi and Xiaosong Ma (Eds.). ACM, 9–16. https://doi.org/10.1145/3476886.3477511

[39] Jizhe Wang, Pipei Huang, Huan Zhao, Zhibo Zhang, Binqiang Zhao, and Dik Lun Lee. 2018. Billion-scale Commodity Embedding for E-commerce Recommendation in Alibaba. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, KDD 2018, London, UK, August 19-23, 2018*, Yike Guo and Faisal Farooq (Eds.). ACM, 839–848. https://doi.org/10.1145/3219819.3219869

[40] Ruoxi Wang, Bin Fu, Gang Fu, and Mingliang Wang. 2017. Deep & Cross Network for Ad Click Predictions. In *Proceedings of the ADKDD'17, Halifax, NS, Canada, August 13 - 17, 2017*. ACM, 12:1–12:7. https://doi.org/10.1145/3124749.3124754

[41] Mark Wilkening, Udit Gupta, Samuel Hsia, Caroline Trippel, Carole-Jean Wu, David Brooks, and Gu-Yeon Wei. 2021. RecSSD: near data processing for solid state drive based recommendation inference. In *ASPLOS '21: 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Virtual Event, USA, April 19-23, 2021*, Tim Sherwood, Emery D. Berger, and Christos Kozyrakis (Eds.). ACM, 717–729. https://doi.org/10.1145/3445814.3446763

[42] Zhibo Xiao, Luwei Yang, Wen Jiang, Yi Wei, Yi Hu, and Hao Wang. 2020. Deep Multi-Interest Network for Click-through Rate Prediction. In *CIKM '20: The 29th ACM International Conference on Information and Knowledge Management, Virtual Event, Ireland, October 19-23, 2020*, Mathieu d'Aquin, Stefan Dietze, Claudia Hauff, Edward Curry, and Philippe Cudré-Mauroux (Eds.). ACM, 2265–2268. https://doi.org/10.1145/3340531.3412092

[43] Haojie Ye, Sanketh Vedula, Yuhan Chen, Yichen Yang, Alex M. Bronstein, Ronald G. Dreslinski, Trevor N. Mudge, and Nishil Talati. 2023. GRACE: A Scalable Graph-Based Approach to Accelerating Recommendation Model Inference. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3, ASPLOS 2023, Vancouver, BC, Canada, March 25-29, 2023*, Tor M. Aamodt, Natalie D. Enright Jerger, and Michael M. Swift (Eds.). ACM, 282–301. https://doi.org/10.1145/3582016.3582029

[44] Xiangyu Zhao, Changsheng Gu, Haoshenglun Zhang, Xiwang Yang, Xiaobing Liu, Jiliang Tang, and Hui Liu. 2021. DEAR: Deep Reinforcement Learning for Online Advertising Impression in Recommender Systems. In *Thirty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2021, Thirty-Third Conference on Innovative Applications of Artificial Intelligence, IAAI 2021, The Eleventh Symposium on Educational Advances in Artificial Intelligence, EAAI 2021, Virtual Event, February 2-9, 2021*. AAAI Press, 750–758. https://doi.org/10.1609/AAAI.V35I1.16156

[45] Lichun Zhou. 2020. Product advertising recommendation in e-commerce based on deep learning and distributed expression. *Electron. Commer. Res.* 20, 2 (2020), 321–342. https://doi.org/10.1007/S10660-020-09411-6