小 型 微 型 计 算 机 系 统
Journal of Chinese Computer Systems
DOI:10.20009/j.cnki.21-1106/TP.2025-0120
2026 年 4 月 第 4 期
Vol.47 No.4 2026

面向大规模推荐的基于哈希的 DRAM-SSD 嵌入表存储系统

敖旭扬,楼博涵,王永福,李 京

(中国科学技术大学 计算机科学与技术学院,合肥 230026)
E-mail:lj@ustc.edu.cn

摘 要:随着深度学习推荐模型规模的持续扩展,处理分类特征的嵌入表所需内存(DRAM)容量呈指数级增长,单机内存已难以满足其存储需求,传统的多机分布式方案在节点数量过多时会显著增加系统成本.最近的研究尝试使用固态硬盘(SSD)存储部分嵌入表,但由于大多依赖定制硬件而难以广泛应用.本文提出的 SSDHashEmbed 利用哈希分区技术优化缓存替换策略的并发性能,并基于哈希表实现了一个高效的两级嵌入表存储模块.通过普通商用 SSD 与少量 DRAM 的结合,该方法能够有效支持大规模嵌入表的存储,从而大幅降低内存需求.基于真实公共数据集的实验评估表明,SSDHashEmbed 可将训练和推理阶段的内存使用量分别降低 50% 和 80%,同时将训练速度提高到基线方案的 1.8~3.4 倍,推理端到端延迟降至基线方案的 70%.
关 键 词:嵌入表;固态硬盘;哈希;推荐系统;性能优化;缓存系统
中图分类号:TP399 **文献标识码**:A **文 章 编 号**:1000-1220(2026)04-0802-08

Hash-based DRAM-SSD Storage System for Large-scale Recommendation Embedding Tables

AO Xuyang,LOU Bohan,WANG Yongfu,LI Jing
(School of Computer Science and Technology,University of Science and Technology of China,Hefei 230026,China)

Abstract:As deep learning-based recommendation models continue to scale,the memory (DRAM) required for storing embedding tables of categorical features grows exponentially. Single-machine memory is increasingly insufficient to accommodate these storage demands,while traditional multi-node distributed solutions incur significant system costs as the number of nodes increases. Recent studies have explored leveraging solid-state drives (SSDs) to store parts of embedding tables,but their reliance on specialized hardware limits widespread adoption. This paper presents SSDHashEmbed,a novel hybrid storage framework that integrates hash-based partitioning to enhance the concurrency of cache replacement strategies and implements an efficient two-tier embedding table storage module using hash tables. By combining commercial off-the-shelf SSDs with a small amount of DRAM,it effectively supports the storage of large-scale embedding tables,significantly reducing memory consumption. Experimental evaluations on real-world public datasets demonstrate that SSDHashEmbed reduces memory usage by 50% during training and 80% during inference,while achieving 1.8× to 3.4× speedup in training and reducing end-to-end inference latency to 70% of the baseline solution.
Keywords:embedding table;solid-state drive;hashing;recommendation system;performance optimization;caching system

0 引 言

随着互联网技术的快速发展,网络视频[1]、文本内容[2]、电子商务商品[3]等信息资源呈现爆炸式增长.面对信息过载的挑战,用户难以高效获取符合个人兴趣和需求的内容.在此背景下,推荐系统应运而生,它通过分析用户的显式反馈和隐式行为数据,为用户提供个性化推荐服务,有效提升信息获取效率.

早期的推荐模型主要依赖于协同过滤算法[4,5],然而随着数据规模的急剧膨胀和应用场景的日益复杂,基于深度学习的推荐模型逐渐占据主导地位[6].相较于传统协同过滤方法,深度学习推荐系统能够构建更为精细的用户画像,尤其在长尾内容推荐方面展现出显著优势.

深度学习推荐模型的输入通常包含分类特征和数值特征.其中,单个分类特征可能涵盖数百万乃至数十亿个离散类别(如大型电商平台的用户 ID).为了充分利用深度神经网络(Deep Neural Networks,DNN)进行特征交互并生成推荐结果,通常需要将稀疏的分类数据转换为稠密的数值向量,也叫嵌入向量(Embedding Vector,EV),这一转换过程通过嵌入表(Embedding Table)实现,即从分类到对应的嵌入向量的映射表.在模型训练过程中,系统会对嵌入表中存储的嵌入向量进行初始化和持续更新,其中大多数嵌入向量会经历多次更新;在推理阶段则直接使用训练好的嵌入向量,无需进行更新.为确保服务级别协议(SLA)并实现实时响应,现有推荐系统通常将所有嵌入表完全存储在 DRAM 中.例如在阿里巴巴的 DeepRec[7]系统中,嵌入表通过哈希表结构实现索引映射,其中键为特征 ID,值为对应嵌入向量在内存中的存储地址.

基于深度学习的推荐模型主要包含两部分:

1)嵌入层:由多个嵌入表构成,主要功能是将离散型特征 ID 转换为对应的嵌入向量.每个嵌入表需要维护特征 ID

收稿日期:2025-03-14 收修改稿日期:2025-04-11 基金项目:中国科学院战略先导科技专项项目(A 类)(XDA19020102)资助. 作者简介:**敖旭扬**,男,2000 年生,硕士研究生,研究方向为高性能计算;**楼博涵**,男,2001 年生,硕士研究生,研究方向为高性能计算;**王永福**,男,1999 年生,博士研究生,研究方向为高性能计算;**李 京**(通信作者),男,1966 年生,博士,教授,博士生导师,CCF 高级会员,研究方向为分布式系统、区块链、高性能计算.

到嵌入向量的映射关系,其存储效率直接影响模型性能.

2)深度神经网络层:负责处理数值型特征和嵌入层生成的向量,通过多层网络结构学习特征组合规律,最终输出预测结果.

在深度推荐模型的训练或推理过程中,稀疏输入通常表现为一系列 ID 数字. 对于每个特征的 ID,系统首先从 DRAM 中对应的嵌入表查找对应的嵌入向量,然后将所有检索到的嵌入向量与稠密输入一起进行拼接或相加等操作,随后送入计算密集的 DNN 层进行处理,最终输出推荐结果或用于参数更新,图 1 展示了这一基本流程.

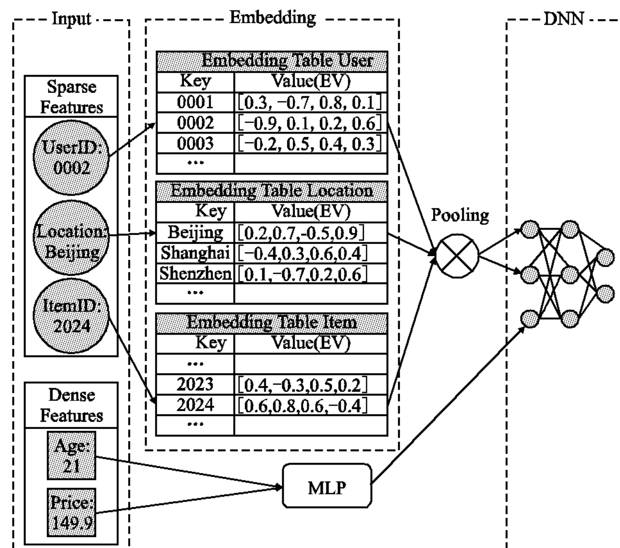

图 1　深度推荐模型训练前向或推理的基本流程

Fig. 1　Basic workflow of forward training or inference in deep recommendation models

然而,单个嵌入表可能包含数十亿个嵌入向量,从而占用数百 GB 内存空间. 此外,为提升推荐效果,研究者倾向于采用更多参数和更稀疏的特征[8],这进一步加剧了嵌入表的膨胀问题. 近年来,推荐模型的参数量已从 10 亿激增至 100 万亿[1]. 在某些系统中,嵌入表整体规模甚至可达 10TB,远超单机 DRAM 容量上限[9-11]. 传统解决方案采用模型并行的分布式架构,将嵌入表等模型参数存储在参数服务器(PS)[12,13]的 DRAM 中,由 Worker 节点负责计算任务. 这种 PS-Worker 架构可能包含数百台服务器[14],其中大量参数服务器的 CPU 资源处于闲置状态,不仅造成高昂的成本,还带来了显著的通信开销和系统复杂性.

因此探索降低嵌入表 DRAM 需求的方法具有重要价值,任何微小的改进在推荐系统的规模效应下都可能带来显著的成本节约. 研究者们注意到推荐场景中嵌入向量访问呈现显著的幂律分布特征. Bagpipe[11]在 Kaggle(Criteo)数据集上观察到 90% 的访问请求集中在嵌入表中 0.1% 的嵌入向量,本文在 Eleme 数据集[15]上也发现了类似规律. 基于这种访问特征,可以采用更经济高效的多级存储方案:使用小容量高速存储(如 GPU 高带宽内存 HBM 或显存、DRAM)保存高频访问的嵌入向量,同时利用大容量低速存储(如 GPU 场景中的 DRAM、PMem、SSD)保存低频访问的嵌入向量,从而优化嵌入表的整体存储效率. 通过动态调整嵌入向量在高速存储和低速存储中的位置,并采用合适的缓存替换策略,可以确保大多数数据访问发生在高速存储上. 这种多级存储方案能够在保证性能无明显下降的前提下,使用更少的高速存储介质,从而以更低的成本训练或推理更大规模的稀疏推荐模型,并提高单位硬件成本的数据吞吐量.

近年来 SSD 的读写性能得到了提升的同时单位容量的价格也逐渐降低,一种可行的降低嵌入表 DRAM 需求的方法是将访问频率较低的嵌入向量卸载到 SSD 存储. 然而,现有研究要么依赖于定制硬件,要么采用有损压缩技术[16,17],或仅限于推理场景[10]. 本文提出的 SSDHashEmbed 系统无需特殊硬件支持,可在现有设备上部署,且不会降低推荐模型的准确性,同时支持训练和推理场景. 本工作的主要贡献如下:

1)设计了 EV-SSD-Storage,通过精心设计的索引机制和读写策略,支持嵌入向量在 DRAM 和 SSD 之间的高效迁移,最大限度地降低 SSD 存储带来的性能损耗.

2)开发了 Block-Cache,通过哈希分区 LRU/LFU 缓存并结合无锁哈希表,提升缓存替换策略的并发性能,确保大多数嵌入向量访问发生在 DRAM 中,同时降低额外性能开销.

3)采用流水线并行技术,实现计算与嵌入向量查找的重叠执行,显著提升系统在训练场景下的数据吞吐效率.

1　相关工作

1.1　基于 NVM 存储的方法

为了提供比纯 DRAM 存储更具成本效益的替代方案,Eisenman 等人[18]提出了 Bandana 系统,该系统利用 NVM 存储推荐系统的嵌入向量. 它面临的主要挑战是 NVM 的有限读取带宽和读取放大问题[19]. 为解决这些问题,Bandana 对嵌入向量进行重新排序,并将相关嵌入向量物理存储在一起,以实现高效预取. 此外,Bandana 通过为每个嵌入表设置一个微型缓存来优化缓存大小,并采用最近最少使用(LRU)的缓存替换策略. 然而,该方法存在以下局限性:首先,它需要在训练开始前遍历所有嵌入向量以设置参数,且缓存大小是静态分配的,无法在运行时动态调整;其次,NVM 在性能和成本上均高于 SSD,其设计不能很好地用于普通商用 SSD. Bandana 仅适用于稀疏推荐模型的训练,在在线学习或推理场景中表现不佳. 此外,嵌入向量的访问具有不规则性和较差的局部性[20,21],导致查找到的"相关"嵌入向量未必真正相关,其实际效果高度依赖于特定的嵌入表访问模式,且可能需要频繁进行嵌入向量的分区和聚类操作,这会导致性能严重下降. 本文以高效 SSD 嵌入向量读写为目标,用一系列实现级优化提高了 DRAM-SSD 混合存储架构下的嵌入向量访问性能,以适应训练和推理场景.

1.2　使用多级流水线隐藏延迟

在业务需求的驱动下,百度的 Zhao 等人[22]提出了 AIBox 系统,该系统通过单个 CPU、多个 GPU 以及 SSD 实现了超大规模点击率(CTR)预测模型的高效训练. AIBox 采用 SSD-DRAM-HBM 三级存储架构来存储嵌入表,将超大规模 CTR 预测模型分为两部分,内存密集型训练任务保留在 CPU 上,同时利用内存有限的 GPU 加速计算密集型任务. 为了在

单个节点上存储 10TB 规模的模型参数，AIBox 引入了稀疏表，这是一个在 SSD 上存储嵌入表的键值系统，通过哈希索引和两层缓存管理实现. AIBox 通过三阶段流水线（网络、稀疏表、CPU + GPU）执行方式来消除异构设备之间的延迟差距. AIBox 在每个微型批次之间应用流水线并行处理，并在每个微型批次结束后同步参数，这种半同步训练方式虽然比集群异步训练对模型精度的影响更小，但仍会一定程度降低模型效果. 且其存储管理依赖粗粒度分块，未针对 SSD 的块大小特性进行细粒度优化，在流水线空缺时效率较低. Jain 等人[21]也在训练场景中通过预取下一个训练批次的参数来隐藏低速存储介质的延迟，但这需要合理设置缓存容量以确保命中率足够高，从而避免流水线被 SSD 的高访问延迟阻塞.

这些工作存在的主要问题是效果非常依赖缓存命中率，但忽略了缓存策略本身元操作对系统性能的损耗，且在在线学习或推理场景更加难以适应，同时半同步训练会降低模型效果. 本文实现了一个高并发性能和高命中率的缓存替换策略，使得 DRAM-SSD 混合存储系统能够高效读写嵌入向量，并确保参数及时同步.

1.3 优化 SSD 读写嵌入向量

针对 SSD 读写或者缓存容量设置等局部优化工作，Rec-SSD[20]、NDRec[23]和 SmartSSD[24]采用近数据处理解决方案，将嵌入向量的收集和聚合计算操作卸载到 SSD，充分利用 SSD 内部带宽并减少主机 CPU 和 SSD 内存之间的往返数据通信开销. FlashEmbedding[25]提出的 EV-SSD 通过减少读取放大和简化 I/O 堆栈来快速访问 SSD 中的嵌入向量，并使用内存缓存这些嵌入向量，同时利用流水线重叠计算来隐藏 I/O 延迟，从而优化嵌入向量的访问效率. FlashGNN[26]充分利用闪存芯片中的 I/O 并行性，并最大限度地提高获取的闪存块中的数据重用性，以实现高效的 GNN 训练. 然而这些方法由于需要定制的硬件而难以规模化部署，本文在普通商用 SSD 上设计和优化性能，不依赖特殊硬件，适用性广.

2 系统设计

本文提出的 SSDHashEmbed 是一种在内存容量受限环境下支持大规模推荐模型训练与推理的创新解决方案. 该方案不仅避免了精度损失，而且无需依赖特殊硬件支持. 这使现有计算设备能够支持更大规模的模型训练，有效防止因内存不足导致的程序崩溃，同时将性能损失控制在可接受范围内. 如图 2 所示，SSDHashEmbed 主要由两个核心组件构成：

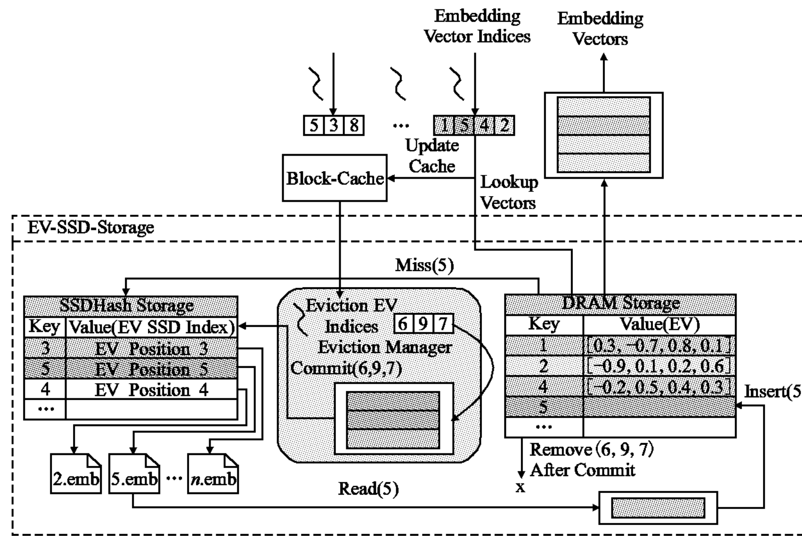

图 2 SSDHashEmbed 的总体设计

Fig. 2 Overall design of SSDHashEmbed

1）EV-SSD-Storage：提供与纯 DRAM 嵌入表访问接口一致的两级存储服务，包含从 SSD 高效读写嵌入向量的实现机制. 该组件利用少量 DRAM 作为高速缓存，并采用优化的缓存替换策略动态维护 DRAM 缓存中的高频嵌入向量集合.

2）Block-Cache：提供高并发性能的缓存替换策略实现，能够在多线程环境下快速更新访问特征信息，同时保持较低的缓存缺失率，从而有效减少读取嵌入向量的平均延迟.

2.1 EV-SSD-Storage

EV-SSD-Storage 由 DramStorage、SsdHashStorage 和 Eviction Manager 3 个核心模块组成，用于存储单个嵌入表. DramStorage 负责管理 DRAM 中缓存的高频嵌入向量集合，SsdHashStorage 则负责 SSD 中存储的全量嵌入向量. Eviction Manager 模块通过缓存替换策略在 DramStorage 容量超出限制时，将低频访问的嵌入向量迁移至 SsdHashStorage. 当查找的嵌入向量不在 DramStorage 中时，系统会从 SsdHashStorage 读取该嵌入向量并加载到 DramStorage，确保嵌入表中嵌入向量的实际使用和更新操作均在 DRAM 中进行.

DramStorage 作为 DRAM 中的 KV 查找表，其键为嵌入向量查找索引（通常为整数 ID），值为嵌入向量的内存地址. 在内存充足的情况下，所有嵌入向量均存储于此，为模型训练和推理提供最优的访问性能. 该模块采用定制的 EVAllocator 高效管理从操作系统申请的内存，其设计简洁高效，嵌入向量在其中连续存储，没有任何压缩或量化操作.

SsdHashStorage 作为 SSD 中的 KV 查找表，其键与 DramStorage 一致，但值为嵌入向量在 SSD 上的索引. 其中的所有嵌入向量实际存储在多个 emb 文件中，采用与 DRAM 相

同的格式紧密排列. 索引信息包括 emb 文件编号、文件内偏移量等辅助信息,它被压缩为 32 位无符号整数以优化内存使用,如图 3 所示. 其中,Invalid 和 Flushed 标志位各占 1bit,剩余 30bit 分配给文件编号和文件内偏移. 由于单个嵌入表中嵌入向量长度固定,文件内偏移以整个嵌入向量为单位计算. 这种设计支持最大 10 亿规模的单个嵌入表(2^{30}),并可轻松扩展至 64 位无符号数以支持更大规模的嵌入表存储.

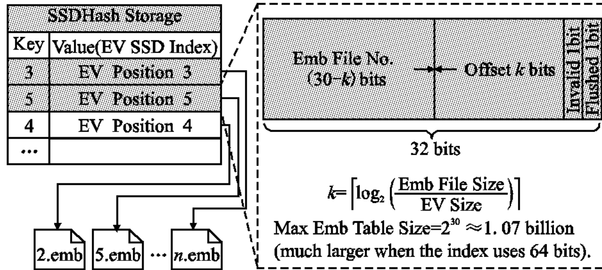

图 3　紧凑的嵌入向量位置索引设计

Fig. 3　Compact indexing design for embedding vectors

考虑到 SSD 的块级读写特性(通常大于 4KB)与推荐模型中典型嵌入向量大小(128B ~ 2KB[25])的差异,本文采用"连续写随机读"策略. 淘汰的嵌入向量先写入 Buffer,待 Buffer 写满后批量追加写入多个嵌入向量. 由于嵌入向量访问局部性较差,读取时采用随机读方式,通过内存中的 SSD 索引实现单次 IO 读取,如图 4 所示.

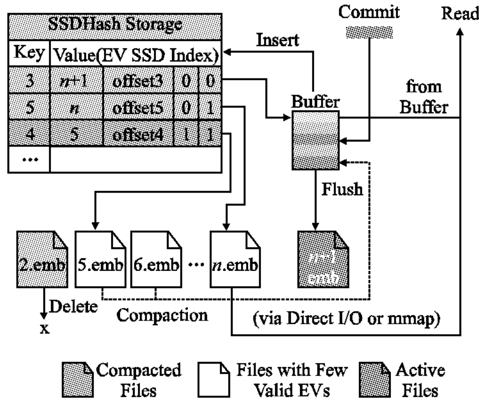

图 4　采用合并优化的嵌入向量连续写随机读

Fig. 4　Compaction-optimized sequential write-random read access patterns for embedding vectors

追加写会导致同一特征对应的嵌入向量可能被多次写入 SSD 不同位置,造成存储空间浪费,为此 SsdHashStorage 在生成新存储文件后触发压实流程. 该操作在后台遍历 SSD 嵌入向量索引表,识别有效嵌入向量比例低于阈值的 emb 文件,将其有效嵌入向量迁移至最新 emb 文件,并将这类低效文件加入待删除集合,在下一压实周期进行延迟删除,算法 1 展示了嵌入向量淘汰和压实细节.

算法 1. SsdHashStorage 的提交和压实

```
初始化嵌入向量写缓冲 Buffer
初始化文件编号 Version = 0
初始化缓冲指针 Offset = 0
初始化待删除文件结合 DFilesSet = { }
While True do
  从 Block-Cache 中获取待提交嵌入向量 ids
  For id in ids do
    If Buffer 未满 then
      从 DramStorage 中读取 id 对应嵌入向量写入 Buffer + Offset
      将 V 和 O 打包成索引 posi 存入 SsdHash 表,两标志位为 0
      Offset = Offset + 嵌入向量长度
    Else
      将 Buffer 一次刷写到 Version. emb 文件中
      把 Buffer 中的嵌入向量 Flushed 标志位改为 1
      Version = Version + 1
      Offset = 0
      // Compaction 子过程
      先删除 DFilesSet 中的文件
      无锁遍历 SsdHash 统计每个 emb 文件的有效嵌入向量数
      筛选出有效嵌入向量少于阈值的 emb 文件集合 InvalidSet
      For inemb in InvalidSet do
        将 inemb 中的有效嵌入向量写入 Buffer 并更新 Offset
        把 inemb 加入 DFilesSet
      End For
    End If
  End For
End While
```

在读取嵌入向量时,系统支持多种 IO 模式. mmap 模式通过将 Page Cache 映射到用户空间虚存地址,提供优异的文件读写性能,但可能导致内存占用持续增加和 Page Fault 引发的性能下降[27]. mmap 结合 madvise 的方法可部分缓解这一问题. 直接 IO 模式则绕过 Page Cache,实现磁盘到用户内存的直接数据传输,避免双重拷贝,在内存容量受限场景下表现最佳,具体评估将在第 3.3 节详细阐述.

Eviction Manager 模块负责执行嵌入向量淘汰操作,通过后台线程监控 DramStorage 中缓存的嵌入向量数量. 当超过缓存容量限制时,利用内部缓存替换策略识别待淘汰嵌入向量索引并执行淘汰操作. 这种设计确保仅有一个线程执行 SSD 写入操作,既避免了复杂的多线程写文件问题,又防止了淘汰操作阻塞嵌入向量查找.

2.2　Block-Cache

每个特征嵌入表对应的 EV-SSD-Storage 中的 Eviction Manager 都关联一个 Cache 模块,它通过分析嵌入向量访问历史识别重用模式,从而在有限缓存容量内优化嵌入向量存储位置,确保大多数查找请求可在 DRAM 中完成,尽可能避免 SSD 访问. Block-Cache 作为嵌入向量索引的缓存,独立于 DramStorage 运行,仅记录 DRAM 中嵌入向量的索引信息. 当 DRAM 中嵌入向量超过容量限制时,Block-Cache 识别待淘汰索引,由 Eviction Manager 执行实际淘汰操作.

当缓存达到设定容量时,由缓存替换策略决定淘汰条目. LRU 和 LFU 能提供较高命中率,其传统实现(LRU 采用双向链表 + 哈希表,LFU 采用多频次双向链表 + 哈希表)在单线程下表现优异,但在多线程训练场景中,由于需要全局锁保证线程安全,导致性能显著下降. Block-Cache 创新性地采用非链表实现方式,如图 5 所示,将整个 Cache 分解为多个小 Cache 块(每块包含 8 ~ 64 个连续存储的缓存条目),每个 Cache 块配备独立细粒度锁. 通过哈希方法将每个键稳定映

射到特定Cache块,在块内独立执行更新和淘汰操作,仅需对当前Cache块上锁,大幅缓解锁竞争,提升并发性能.同时,得益于Cache块的连续存储特性和CPU的缓存行机制,遍历适当大小的Cache块执行缓存替换策略效率极高,因此Block-Cache具有很高的并发性能.

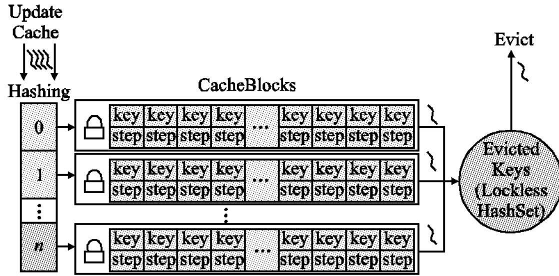

图5 Block-Cache使用哈希分区提升并发性能

Fig.5 Block-cache accelerates concurrency through hash partitioning

为实现EV-SSD-Storage中单后台线程批量淘汰嵌入向量的能力,Block-Cache需要支持一次性获取多个待淘汰条目.分块设计使得传统链表式cache的尾部淘汰策略不再适用.为解决这一问题,本文采用开源的LocklessHashset在小Cache块更新时及时保存淘汰条目.该结构无需显式加锁,具有优异的并发性能,在批量淘汰时仅需遍历LocklessHashset,同样无需加锁操作,如算法2所示.

算法2. Block-Cache(LFU)更新和批量淘汰

```
初始化Cache块Blocks[]
初始化Cache块数量BlockNum
初始化LocklessHashset
// 访问了一系列嵌入向量ids后更新Block-Cache
For id in ids do
  bidx = hash_function(id) % BlockNum
  初始化最低频次条目min_item
  给Blocks[bidx]加锁
  For item in Blocks[bidx] do
    If item 空闲 then
      item = (id,1)
    Else
      比较并记录最低频次
    End If
    把min_item.id加入LocklessHashset
    min_item = (id,1)
  End For
  解锁Blocks[bidx]
End For
// 批量淘汰时遍历LocklessHashset并从中删除
```

3 实验

本文在阿里巴巴开源的基于TensorFlow[28]的推荐引擎DeepRec上,使用2500行C++代码实现了SSDHashEmbed,其中流水线并行采用了多会话并行的朴素实现.该实现提供了与DRAM嵌入表完全一致的接口,将DRAM和SSD两级存储的复杂逻辑封装在内部,包括多种缓存替换策略、嵌入向量访问的不同IO模式以及嵌入向量淘汰机制等核心功能.

实验在本地搭建的单机环境中进行,硬件配置如表1所示,软件环境基于Ubuntu 22.04.5 LTS操作系统,使用Docker版本27.3.1进行环境隔离和内存容量限制.

表1 实验平台硬件配置

Table 1 Hardware specifications of the computing testbed

硬件类型	详细信息
处理器	Intel(R) Core(TM) i7-10710U
内存	32 GiB Kingston DDR4 2667 MHz
固态硬盘	Samsung SSD 970 EVO Plus 250GB

本文选取了DLRM[29]、WDL[30]、MMoE[31]和DIEN[32]4种模型用于测试SSDHashEmbed性能,数据集分别为Eleme、Tao Ads、TinyTaobao和Amazon Books,他们已经在DeepRec系统经过适配,其中Eleme数据集有8天的数据,分为多个格式相同的压缩包,共有1TiB左右.完整数据集对于本文实验平台来说过于庞大,但是预先分析后发现第一个压缩包的数据也符合实验要求的幂律分布,因此本文选取第一个压缩包用于实验,这保证了实验的可行性和效率,同时不影响实验效果,其他数据集均使用DeepRec提供的原始完整版本.每种模型所使用的数据集、嵌入表个数、嵌入层大小(所有嵌入表中的嵌入向量大小之和)和充足内存下的训练过程内存峰值如表2所示.

表2 测试模型与数据集

Table 2 Test models and datasets

模型	数据集	嵌入表个数	嵌入层大小	内存峰值
DLRM	Eleme	4	16.43 GiB	22.1 GiB
WDL	Tao Ads	26	10.82 GiB	13.4 GiB
MMoE	Tiny Taobao	16	0.49 GiB	2.2 GiB
DIEN	Amazon Books	1	0.60 GiB	3.1 GiB

实验中使用SSDHashEmbed等不同方式存储各个模型中总计大小占90%以上的嵌入表,总缓存容量设置为可用物理内存减去除嵌入层外的内存开销,并以每个嵌入表的大小为权重分配,剩余较小的嵌入表均存储在DRAM中,保持各种模型嵌入层部分实际可用内存与理想内存比值相同.训练和推理过程中,Batch Size均设置为512.

本文首先对比了Block-Cache与基于链表的Cache在性能和命中率方面的优劣,之后主要将SSDHashEmbed与以下两种场景进行对比:1)理想情况下的充足DRAM配置;2)有限DRAM+操作系统SWAP的Baseline方案.通过Docker限制模型训练和推理进程的可用内存量,以模拟不同内存容量约束场景.此外,本文还在DRLM模型上进一步详细对比了SsdHashStorage与LevelDB作为存储后端的性能差异,评估了不同IO方式和缓存替换策略的性能表现.

3.1 Block-Cache性能评估

本文基于Block-Cache设计,使用C++实现了Block-LRU和Block-LFU,并通过参数控制每个Cache块的条目数量.同时,本文实现了基于链表的LRU和LFU作为对比基准.这些缓存替换策略均经过优化,支持多线程访问.

本文使用具有倾斜分布的数据测试了不同缓存替换策略

在50% 缓存容量下的更新(包括插入和淘汰)性能和命中率.测试数据包含118万个唯一值.采用以下方式测试不同Cache性能:将一批键平均分配给多个线程,每个线程循环访问并更新对应的缓存条目.当缓存大小超过容量限制时,执行批量淘汰(最多10000个条目).实验共进行20个Batch, Batch Size为1024×128,测试得到更新耗时和命中率.

3.1.1　更新性能

图6展示了不同线程数下,各种缓存替换策略实现执行更新操作的耗时.实验结果表明,基于链表的LRU在单线程更新场景下表现优于块大小为16的Block-LRU.然而,随着线程数增加,基于链表的LRU性能显著下降,而Block-Cache

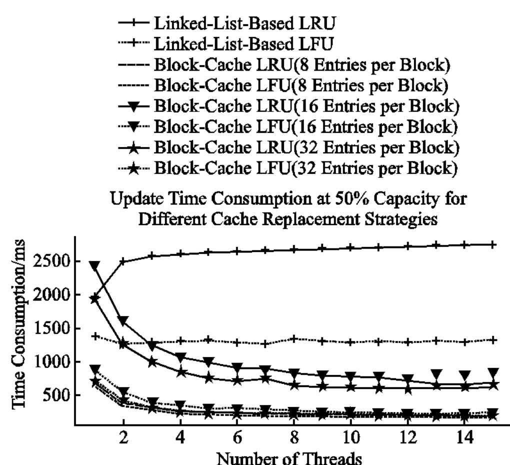

图6　不同线程数下各种缓存替换策略实现的更新耗时

Fig. 6　Update time of various cache replacement strategies under different thread counts

的性能则快速提升.基于链表的LFU性能相对稳定,但远不及Block-Cache.考虑到推荐模型训练和推理通常采用多线程方式,Block-Cache在此类场景下展现出显著优势.如表3所示,在15线程更新场景下,Block-Cache相比基于链表的LRU实现了高达16.11倍的加速.此外,Cache块大小对Block-Cache性能也有影响,增大Cache块会导致性能下降.

表3　在15线程更新场景下各种Cache的更新耗时和命中率

Table 3　Update time and hit rate of caches under 15 threads

Cache	更新时间(ms)	命中率(%)
LRU	2736.010(16.11x)	37.36
LFU	1314.707(7.74x)	38.83
B8LRU	189.575(1.12x)	35.65
B8LFU	169.879(1.00x)	36.67
B16LRU	678.453(3.99x)	36.47
B16LFU	217.959(1.28x)	37.56
B32LRU	614.179(3.62x)	36.86
B32LFU	187.762(1.11x)	37.36

3.1.2　命中率

实验结果显示,在Cache测试数据中,LFU的命中率总是高于LRU,在训练数据上也是同样的结果.此外,由于Block-Cache在每个小Cache块中独立执行缓存替换策略,导致其命中率略低于基于链表的Cache.表3最右边一列还展示了线程数为15的情况下,各种缓存替换策略实现执行更新操作的命中率,实际上线程数对所有缓存替换策略实现的命中率几乎没有影响.

总之线程数量对Block-Cache和基于链表的Cache的命中率影响细微.对于两种Cache策略,LFU的命中率均高于LRU.对于Block-Cache而言,Cache块越大,命中率越高.因此,Block-Cache的设计细节需要在更新性能和命中率之间进行权衡.基于实验结果,本文最终选择B32LFU(块大小为32的Block-Cache-LFU)进行系统整体性能测试.

3.2　整体性能评估

本文在充足内存、操作系统SWAP机制(Baseline)和SSDHashEmbed 3种配置下训练各种模型,其中后两种配置将其嵌入层可用内存分别限制到所需充足内存的67%(L)和53%(M).图7显示了4种模型和数据集的训练总耗时对比, SSDHashEmbed相较于Baseline在L级别的内存容量限制下可带来1.3~2.0倍的性能提升,在M级别的内存容量限制下可带来1.8~3.4倍的性能提升,其中DLRM和WDL两个模型中嵌入表相关操作占比较多,加速效果更为明显.

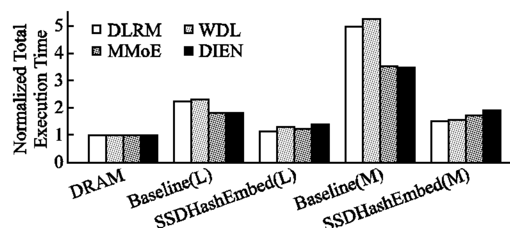

图7　完整训练过程的耗时

Fig. 7　Training execution time

本文在DLRM模型上与第1节中提到的AIBox、RecSSD和FlashEmbedding进行对比,RecSSD与FlashEmbedding使用了硬件级加速,因此本文计算缓存带来的加速比来统一比较项,详细结果如表4所示.其他工作只关注了缓存替换策略命中率而忽略了其实现性能,相比之下Block-Cache的高命中率和并发性能设计带来了最大的缓存加速比.表4最右边一列表示限制内存容量后的模型整体训练耗时与内存充足时耗时的比值,该值越小表示性能损失越小,其中1.14的比例表明SSDHashEmbed能够有效降低内存需求,即节省内存使用量造成的性能损耗较小,优于AIBox.

表4　嵌入表缓存加速比

Table 4　Speedup ratio of embedding table caching

加速比	Lookup	端到端	训练耗时
AIBox	1.77x	1.38x	1.25x
RecSSD	1.45x	1.25x	-
FlashEmbedding	2.75x	1.20x	-
SSDHashEmbed	**3.76x**	**1.41x**	**1.14x**

本文还对比了使用采用LevelDB作为存储后端的两级嵌入表的性能来验证EV-SSD-Storage设计的有效性,该实验在DLRM模型上实施,训练过程的内存限制不变,将推理内存限制到嵌入层所需的27%(L)、20%(M)和13%(S).每步耗时与充足DRAM下耗时的比值如表5所示,其中“K”表示系统因内存不足而崩溃,表头中B、H和L分别表示Baseline、SSDHashEmbed和LevelDB.

结果表明在内存容量受限场景下,SSDHashEmbed 训练的端到端每步延迟显著低于 Baseline 方案,远远好于 LevelDB,且随着内存限制的加剧,其优势愈加明显. 这是因为 SWAP 机制的固定粒度设计及其对各类内存数据的统一管理,导致无法充分利用嵌入表中嵌入向量的分布特性和结构特征. 前面训练整体耗时的加速效果比这里每步加速更明显是因为 SSDHashEmbed 使用更少的 DRAM,从而减轻了模型训练过程中其他部分的内存压力. 采用 LevelDB 作为存储后端的方案在内存匮乏时性能急剧下降,这主要源于 LevelDB 作为通用键值数据库所引入的持久化、缓冲、排序等额外开销.

表 5　DLRM 训练和推理的每步延迟

Table 5　Per-step latency of DLRM training and inference

配置	B(L)	H(L)	L(L)	B(M)	H(M)	L(M)	B(S)	H(S)
训练	1.43	1.22	4.32	2.36	1.67	10.95	-	-
推理	6.46	4.73	K	K	6.61	K	K	11.32

在 27% 的内存限制下,SSDHashEmbed 的推理步延迟只有 Baseline 方案的 70%. 虽然随着内存限制的加剧,其延迟也有所增加,但仍能提供可靠的推理服务,避免了系统崩溃. 相比之下,Baseline 和 LevelDB 方案在这种内存限制下因内存资源耗尽而无法完成推理任务.

3.3　IO 模式敏感性

如前文所述,在发生缓存缺失需要从 SSD 读取嵌入向量时,系统可采用不同的 IO 模式,包括 DirectIO(drio)、mmap 以及 mmap + madvice(madv). 表 6 展示了不同 IO 模式下训练和推理过程中产生的 SSD 读写流量与训练耗时、推理端到端延迟,数值均是和内存充足情况下的比值.

表 6　不同 IO 模式下的 I/O 流量和耗时

Table 6　I/O traffic volume and time cost across I/O modes

指标\模式	drio (L)	mmap (L)	madv (L)	drio (M)	mmap (M)	madv (M)
训练读量	2.91	9.22	8.81	6.77	12.81	12.43
推理读量	6.15	31.55	31.82	6.74	33.26	33.91
训练写量	2.35	4.27	4.43	4.78	5.94	6.01
推理写量	25.55	43.09	39.63	29.14	46.64	44.16
训练耗时	1.14	2.66	2.84	1.52	3.4	3.96
推理延迟	4.73	15.89	17.37	6.61	16.36	19.06

训练场景下 DirectIO 的读取和写入流量都远远少于其他两种方式,mmap + madvice 方式的读取流量相比于 mmap 只有微弱的优化. 低效的读取方式也造成了训练过程中写入流量的上升,进一步影响了训练性能. SSDHashEmbed 选择以 DirectIO 的方式读取 SSD 中的嵌入向量来获得内存容量受限情况下的最佳性能. 推理场景下结果与训练场景一致,DirectIO 展现出明显优势. 在最终的耗时或延迟表现上,DirectIO 相比于其他两种方式都有 2 倍以上的巨大优势.

3.4　缓存替换策略敏感性

尽管本文采用的 Block-LFU 相较于基于链表的 LFU 导致了略高的缓存缺失率,理论上可能增加 SSD 访问次数从而影响性能,但实际上缓存更新性能的提升带来了更大的正向影响. 表 7 展示了采用 Cache 块大小为 32 的 Block-LFU (B32LFU)与基于链表的普通 LFU 在训练和推理步延迟方面的对比.

表 7　不同缓存替换策略实现的训练和推理步延迟

Table 7　Training and inference step latency of different cachee

Cache	B32LFU(L)	LFU(L)	B32LFU(L)	LFU(M)
训练延迟	1.22	1.63	4.32	6.93
推理延迟	4.73	5.57	-	-

在训练和推理过程中,每次嵌入向量查找都会触发一次缓存更新. 采用大粒度锁的链表 LFU 在多线程负载下严重阻碍了训练任务的流水线执行,其负面影响超过了高命中率带来的收益. 实际上在 16GiB 内存配置下,B32LFU 的命中率为 89.77%,而 LFU 的命中率为 89.92%,两者差异微乎其微.

4　结　论

本文提出了 SSDHashEmbed,这是一种高效的混合嵌入表存储系统,通过将 SSD 作为 DRAM 的有效扩展,在内存容量受限的系统中支持更大规模的推荐模型训练和推理. 本文设计了 EV-SSD-Storage,用于在 DRAM 和 SSD 之间高效管理和迁移嵌入向量,以及 Block-Cache,用于实现支持高并发更新和高命中率的缓存替换策略. 这些设计均基于哈希方法,并面向高性能实现. 通过进一步采用流水线并行技术重叠计算与嵌入向量查找操作,在真实公共数据集上的评估结果表明,SSDHashEmbed 能够在内存容量受限的条件下显著减少系统训练和推理的耗时,降低服务延迟.

本文还有几个方面可以进一步研究,首先,GPU 凭借其强大的算力在 AI 领域越来越重要,如果能结合算力更强的 GPU,实现 HBM-DRAM-SSD 的三级混合嵌入表存储,则能够进一步提高单机能力. 其次,可以研究自动分配和动态调整多个嵌入表的 DRAM 缓存容量的方法,使得系统整体有较高的命中率,避免配置不合理导致个别嵌入表中的嵌入向量查找操作阻塞整体的训练或推理. 最后,在实现级别方面,训练场景下可以使用预取机制和更精细的流水线来进一步优化,而推理场景下能够对嵌入表做一些预分析和处理从而提高命中率、降低推荐服务延迟.

References:

[1] Lian X, Yuan B, Zhu X, et al. Persia: an open, hybrid system scaling deep learning-based recommenders up to 100 trillion parameters [C]//Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, 2022: 3288-3298.

[2] Gupta U, Wu C J, Wang X, et al. The architectural implications of facebook's DNN-based personalized recommendation [C]//IEEE International Symposium on High Performance Computer Architecture (HPCA), 2020: 488-501.

[3] Zhou G, Zhu X, Song C, et al. Deep interest network for click-through rate prediction [C]//Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, 2018: 1059-1068.

[4] Koren Y, Bell R. Advances in collaborative filtering [M]. Boston, MA: Springer US, 2011: 145-186.

[5] Shen S, Hu B, Chen W, et al. Personalized click model through col-

laborative filtering[C]//Proceedings of the fifth ACM International Conference on Web Search and Data Mining,2012:323-332.

[6] Zhang Y,Chen L,Yang S,et al. PICASSO:unleashing the potential of GPU-centric training for wide-and-deep recommender systems [C]//IEEE 38th International Conference on Data Engineering (ICDE),2022:3453-3466.

[7] DeepRec-AI DeepRec is a high-performance recommendation deep learning framework based on TensorFlow [EB/OL]. https://github. com/DeepRec-AI/DeepRec,2025-03-10.

[8] Wu C J,Brooks D,Chen K,et al. Machine learning at facebook:understanding inference at the edge[C]//IEEE International Symposium on High Performance Computer Architecture(HPCA),2019:331-344.

[9] Xie M, Lu Y, Lin J, et al. Fleche: an efficient GPU embedding cache for personalized recommendations[C]//Proceedings of the 17th European Conference on Computer Systems,2022:402-416.

[10] Sun X, Wan H, Li Q, et al. RM-SSD in-storage computing for large-scale recommendation inference [C]//IEEE International Symposium on High-Performance Computer Architecture (HPCA),2022:1056-1070.

[11] Agarwal S,Yan C,Zhang Z,et al. Bagpipe:accelerating deep recommendation model training[C]//Proceedings of the 29th Symposium on Operating Systems Principles,2023:348-363.

[12] Dean J, Corrado G, Monga R, et al. Large scale distributed deep networks[C]//Advances in Neural Information Processing Systems,2012:1223-1231.

[13] Chen C,Wang Y,Yang J,et al. OpenEmbedding:a distributed parameter server for deep learning recommendation models using persistent memory[C]//IEEE 39th International Conference on Data Engineering(ICDE),2023:2976-2987.

[14] Lin Z,Feng L,Ardestani E K,et al. Building a performance model for deep learning recommendation model training on GPUs[C]//IEEE 29th International Conference on High Performance Computing,Data,and Analytics(HiPC),2022:48-58.

[15] Recommendation Data From Ele. me[EB/OL]. https://tianchi.aliyun. com/dataset/dataDetail? dataId = 131047,2025-03-10.

[16] Kurniawan D H,Wang R,Zulkifli K S,et al. EVStore:storage and caching capabilities for scaling embedding tables in deep recommendation systems[C]//Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems,2023:281-294.

[17] Yang J A,Huang J,Park J,et al. Mixed-precision embedding using a cache[J]. arXiv e-prints, 2020-10-01, doi: 10. 48550/arXiv. 2010. 11305.

[18] Eisenman A,Naumov M,Gardner D,et al. Bandana:using non-volatile memory for storing deep learning models[C]//Proceedings of Machine Learning and Systems,2019:40-52.

[19] Chen C Y,Yen J N,Lai Y R,et al. RecTS:a temporal-aware memory system optimization for training deep learning recommendation models[C]//Proceedings of the 17th ACM International Systems and Storage Conference,2024:104-117.

[20] Wilkening M,Gupta U,Hsia S,et al. RecSSD:near data processing for solid state drive based recommendation inference[C]//Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems,2021:717-729.

[21] Jain R,Cheng S,Kalagi V,et al. Optimizing CPU performance for recommendation systems at-scale[C]//Proceedings of the 50th Annual International Symposium on Computer Architecture,2023:1-15.

[22] Zhao W,Zhang J,Xie D,et al. AIBox:CTR prediction model training on a single node[C]//Proceedings of the 28th ACM International Conference on Information and Knowledge Management,2019:319-328.

[23] Li S,Wang Y,Hanson E,et al. NDRec:a near-data processing system for training large-scale recommendation models[J]. IEEE Transactions on Computers,2024,73(5):1248-1261.

[24] Soltaniyeh M,Lagrange Moutinho Dos Reis V,Bryson M,et al. Near-storage processing for solid state drive based recommendation inference with SmartSSDs? [C]//Proceedings of the ACM/SPEC on International Conference on Performance Engineering,2022:177-186.

[25] Wan H, Sun X, Cui Y, et al. FlashEmbedding: storing embedding tables in SSD for large-scale recommender systems[C]//Proceedings of the 12th ACM SIGOPS Asia-Pacific Workshop on Systems,2021:9-16.

[26] Niu F,Yue J,Shen J,et al. FlashGNN:an in-SSD accelerator for GNN training[C]//IEEE International Symposium on High-Performance Computer Architecture(HPCA),2024:361-378.

[27] Choi J,Kim J,Han H. Efficient memory mapped file I/O for in-memory file systems[C]//Proceedings of the 9th USENIX Conference on Hot Topics in Storage and File Systems,2017:27-32.

[28] Martín Abadi, Ashish Agarwal, Paul Barham, et al. TensorFlow: large-scale machine learning on heterogeneous systems[Z/OL]. http://www. tensorflow. org,2015-11-09.

[29] Naumov M,Mudigere D,Shi H J M,et al. Deep learning recommendation model for personalization and recommendation systems [J]. arXiv e-prints,2019,doi:10. 48550/arXiv. 1906. 00091.

[30] Cheng H T,Koc L,Harmsen J,et al. Wide & deep learning for recommender systems[C]//Proceedings of the 1st Workshop on Deep Learning for Recommender Systems,2016:7-10.

[31] Ma J,Zhao Z,Yi X,et al. Modeling task relationships in multi-task learning with multi-gate mixture-of-experts[C]//Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining,2018:1930-1939.

[32] Zhou G,Mou N,Fan Y,et al. Deep interest evolution network for click-through rate prediction[J]. arXiv e-prints, 2018, doi: 10. 48550/arXiv. 1809. 03672.