计算机研究与发展
Journal of Computer Research and Development
https://crad.ict.ac.cn
62(3): 589−604, 2025

ScaleFS: 面向大语言模型的高性能可扩展元数据设计

尚碧筠[1] 韩银俊[2] 肖 蓉[2] 陈正华[2] 屠要峰[2] 董振江[1]
[1](南京邮电大学计算机学院 南京 210023)
[2](中兴通讯股份有限公司 南京 210012)
(shangbiyun@126.com)

ScaleFS: High Performance and Scalable Metadata Design for Large Language Models

Shang Biyun[1], Han Yinjun[2], Xiao Rong[2], Chen Zhenghua[2], Tu Yaofeng[2], and Dong Zhenjiang[1]
[1] (*School of Computer Science*, *Nanjing University of Posts and Telecommunications*, *Nanjing* 210023)
[2] (*ZTE Corporation*, *Nanjing* 210012)

Abstract In recent years, large language models (LLMs) represented by ChatGPT have developed rapidly. As the scale of model parameters continues to grow, building and deploying LLMs puts forward higher requirement for data scale and storage access efficiency, which poses significant challenges to traditional storage systems. This study first analyzes the storage access characteristics across the three critical stages of LLM workflows: data preparation, model training, and inference. It also explores in depth the major issues and bottlenecks faced by traditional storage systems in LLM scenarios. To address these challenges, the study proposes and implements ScaleFS, a high-performance and scalable distributed metadata design. ScaleFS adopts a decoupled design for directory tree metadata and attribute metadata, and combines with a hierarchical partitioning strategy that balances depth and breadth in the directory tree. This design enables efficient path resolution, load balancing, and system scalability, thereby making it capable of effectively managing hundreds of billions of files. Additionally, ScaleFS introduces fine-grained metadata structures, optimizes metadata access patterns, and develops a metadata key-value store tailored for file semantics. These innovations significantly improve metadata access efficiency while reducing disk I/O operations. The experimental results demonstrate that ScaleFS achieves operations per second (OPS) rates 1.04 to 7.12 times higher than HDFS, with latency reduced to only 12.67% to 99.55% of HDFS. Furthermore, at a scale of hundreds of billions of files, ScaleFS outperforms HDFS in most operations, even when HDFS operates at a billion-file scale. This demonstrates its superior scalability and access efficiency. ScaleFS is thus well-suited to meet the demands of LLM scenarios for managing and efficiently accessing massive file datasets.

Key words large language model; large model training; storage system; metadata management; scalability

摘 要 近年来,以 ChatGPT 为代表的大语言模型(large language model, LLM)技术发展迅速.随着模型参数规模的持续增长,构建和应用大模型对数据存储规模和存储访问效率提出了更高要求,这对传统存储系统带来了严峻挑战.首先分析了大模型在数据准备、模型训练和推理阶段的存储访问特征,深入探讨了传统存储系统在大模型场景下面临的主要问题和瓶颈.针对这些挑战,提出并实现了一种高性能、可扩展的分布式元数据设计 ScaleFS.通过目录树元数据与属性元数据解耦的架构设计,并结合深度与广度均

收稿日期: 2024-05-31; **修回日期**: 2024-11-08
基金项目: 国家重点研发计划项目(2021YFB3101101)
This work was supported by the National Key Research and Development Program of China (2021YFB3101101).
通信作者: 董振江(dongzhenjiang@njupt.edu.cn)

衡的目录树分层分区策略设计, ScaleFS 实现了高效的路径解析、负载均衡和系统扩展能力, 能够高效管理千亿级文件. 此外, ScaleFS 设计了细粒度元数据结构, 优化了元数据访问模式, 并构建了面向文件语义优化的元数据键值存储底座, 显著提升了元数据访问效率并减少了磁盘 I/O 操作. 实验结果表明, ScaleFS 的每秒操作次数 (operations per second, OPS) 是 HDFS 的 1.04~7.12 倍, 而延迟仅为 HDFS 的 12.67%~99.55%. 在千亿级文件规模下, ScaleFS 的大部分操作性能优于 HDFS 在十亿级文件规模下的表现, 展现出更高的扩展性和访问效率, 能够更好地满足大模型场景对千亿级文件存储及高效访问的需求.

关键词 大语言模型; 大模型训练; 存储系统; 元数据管理; 可扩展性

中图法分类号 TP391

DOI: 10.7544/issn1000-1239.202440373 **CSTR:** 32373.14.issn1000-1239.202440373

以 ChatGPT[1] 为代表的大语言模型(large language model, LLM)(以下统称为“大模型”)在文字生成和语义理解等领域表现卓越, 引发了工业界和学术界的广泛关注. 科技界掀起了一场大模型竞争的热潮, 数据成为新生产要素, 算力成为新基础能源, 大模型成为新生产工具, 各行各业从“+AI”向“AI+”的转变已势不可挡. 随着大模型参数量从千亿级别迈向万亿级别[2], 用于训练这些模型的数据文件规模也急剧增长. 此外, 大模型的能力正从单一模态快速向多模态发展, 训练所需的数据类型也变得更加多样, 从文本扩展到音频、图像和视频等, 这给传统存储系统带来了巨大挑战.

如图 1 所示, 构建和应用大模型的过程通常包括数据准备、模型训练和模型推理 3 个阶段, 并呈现出不同的数据存储和访问特征.

我们深入分析了大模型数据准备阶段所依赖的数据源以及数据流入、处理及流出方式, 发现这一阶段的核心需求在于数据的广泛性和高效流动. 数据系统需要频繁与各种数据协议和生态系统进行交互和衔接, 而原有的本地存储或小规模商用存储已无法满足这些需求. 例如, 公开数据集 LAION 5B[3] 的文件数已达到 58.5 亿, 而在数据采集、清洗、格式化等前置步骤中处理的数据量通常是正式训练数据集的 100 倍以上[4]. GPT-4 的训练使用了 13 万亿 token, 按每 token 4 B 计算, 总训练数据集大小约为 53 TB. 尽管这一数据集对存储系统的容量要求并不苛刻, 但在数据准备阶段, 所涉及的数据规模却达到了数百亿个文件, 总容量也达到了 20 PB. 而且这一过程还涉及多种模态的数据文件批量处理, 因此存储系统必须具备高效管理千亿级文件的能力, 并支持高并发的文件读写操作. 千亿级文件的数据湖存储已成为实际应用中的首选方案.

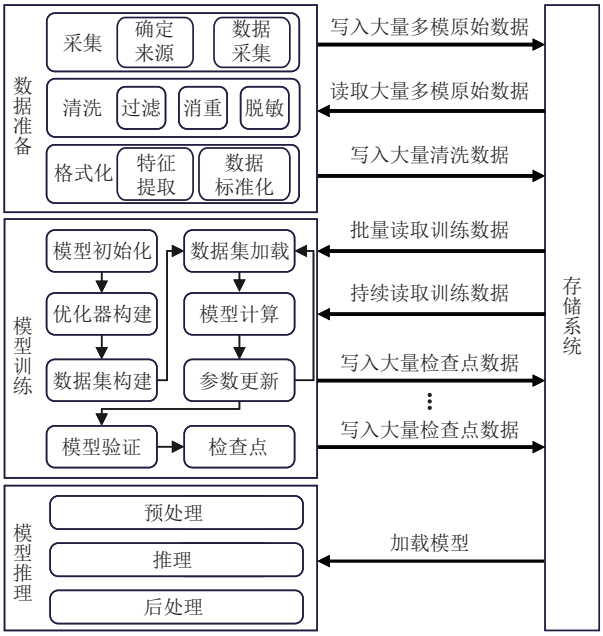


Fig. 1 I/O characteristics in LLMs construction and application

图 1 大模型构建及应用中的 I/O 特征

在模型训练阶段, 与存储紧密相关的有 2 点. 首先是训练数据集的加载, 每个 epoch 需要先对数据集进行 shuffle 操作, 将数据随机打散后再划分为若干个 batch, 每读取 1 个 batch 的数据进行 1 次训练迭代, 这对存储系统造成了大量的并发读写请求. 其次是检查点(checkpoint)机制, 大模型的训练过程通常长达数周至数月, 并且容易出错, 为确保训练的稳定性并便于故障恢复, 系统会定期保存 checkpoint.进一步分析发现, shuffle 操作主要依赖于元数据操作, 需获取大量文件列表, 而实际数据读取则涉及元数据和数据的双重操作. checkpoint 操作则主要依赖缓存加速层, 但在持久化存储时, 需要处理大量文件创建和高带宽的数据写入. 因此, 这一阶段要求存储系统具备高效的元数据访问和随机 I/O 能力.

模型推理阶段的主要工作负载是计算. 随着设备数量的增加, 模型文件并发读取对存储系统的读带宽有一定要求.

总之，虽然构建大模型在不同阶段对存储系统的要求各有不同，但面临的主要挑战是千亿级文件的集中存储和高效随机读写. 海量文件对应着海量的元数据，1 次文件操作通常涉及多次元数据操作，有文献指出元数据操作占据了文件操作的 50%~80%[5]. 对大模型训练常用的 KB 级小文件而言，元数据操作的耗时占比尤为显著. 因此，元数据处理效率对文件系统性能至关重要. 为千亿级文件提供高性能可扩展的元数据服务，成为大模型对存储系统的核心需求. 本文致力于解决元数据规模和访问效率的挑战.

处理千亿级文件的元数据的关键在于实现元数据的水平扩展和高效访问，同时最大化减少分布式架构引入的不利影响. HDFS[6]（Hadoop distributed file system）作为 Hadoop 生态的核心组件，主要负责海量文件数据的存储管理，是数据湖的主流存储方案，也是目前应用最广泛的分布式文件系统. 然而，当前的 HDFS 系统在可扩展性、数据规模和访问性能等方面远不能满足大模型对千亿级文件存储能力的需求. 目前，主流的分布式文件系统依赖子树分区或哈希分区来划分命名空间，但这些策略在优化访问性能和保持系统均衡方面存在局限. 为了克服这些限制，部分系统尝试将命名空间元数据下沉到通用数据库系统或者键值（key-value, KV）存储中. 这一做法虽然解决了扩展性问题，但同时也带来了事务管理开销增加、文件语义失配以及网络远程过程调用（remote procedure call, RPC）延迟增高等问题，这些都对系统的整体性能产生了负面影响.

针对上述难题，本文提出了一种高性能可扩展的分布式元数据管理机制 ScaleFS，在垂直扩展单点元数据管理规模并增强性能的同时可灵活地水平扩展节点，能够更好地满足大模型场景对千亿级文件存储和高效访问的需求. 主要贡献包括：

1）实现了将目录树元数据和属性元数据解耦的架构设计，设计了深度与广度均衡的目录树分层分区策略，实现了目录树的高效查找和元数据的高扩展性.

2）提出了一种访问模式友好的细粒度元数据结构设计，根据元数据属性的类型、访问频率以及访问相关性等因素，设计了精细化的存储结构和缓存策略，减少了磁盘 I/O 操作.

3）实现了面向文件语义的元数据键值存储底座，通过贴合文件系统语义的接口方式，支持键值部分读取和更新的设计，提升了元数据的访问性能.

1 相关工作

为了提升容量和访问性能，文件系统的架构不断演进. 最早的文件系统将元数据和数据集中存储[7]，然而，随着存储规模的增长，元数据的访问效率和扩展能力逐渐成为瓶颈. Gibson 等人[8] 提出了分离的元数据服务器架构，将元数据从数据存储中独立出来，从而进一步提升了文件系统的访问性能和可扩展性.

在分布式文件系统的初始阶段，系统通常依赖单一的元数据服务器来管理和存储元数据，比如 HDFS 和 GFS[9]. 然而，随着系统规模的扩大，元数据节点很快成为限制系统容量和服务能力的瓶颈[10]. 由此，业界开始研究分布式元数据技术. 命名空间的划分被视为分布式元数据架构的关键技术，其目的在于在保持命名空间局部性的同时，确保系统负载的均衡，从而提升元数据服务的整体性能和可靠性. 命名空间的划分主要分为静态划分和动态划分 2 大类.

静态划分方法分为静态子树划分和静态哈希划分. 静态子树划分常见于早期的分布式文件系统，比如 HDFS 引入联邦[11] 机制，将目录树归属到不同的 HDFS 集群管理，集群之间不共享目录，实现了元数据容量的扩展. 但是，单组 HDFS 集群容量和吞吐量均存在上限，当数据规模增长达到单组 HDFS 集群性能极限时，需要在集群之间进行数据迁移. 频繁的数据迁移不仅影响系统性能，还需要暂停上层业务以确保数据一致性，这对业务的连续性造成不利影响，特别是在管理海量数据时，迁移过程耗时的延长进一步加剧系统运维的复杂性，降低了服务的稳定性. 所以，尽管 HDFS 联邦在一定程度上缓解了元数据存储的瓶颈问题，但其在处理大规模数据时仍暴露出诸多不足，亟待进一步的研究和优化.

静态哈希划分[12] 能高效均衡元数据在集群中的分布，并允许通过文件名直接定位到相应的元数据节点，但这种方法由于牺牲了命名空间目录树的局部性，在处理列目录和文件路径重命名操作时显得尤为不便. 而且当集群中的节点发生变动时，这种划分方法要求对整个命名空间进行重新分布，这无疑带来了高昂的代价和复杂性.

动态划分方法[13-14] 能够根据系统实时负载情况动态地调整命名空间的划分，分为动态子树划分和动态哈希划分. 云计算生态 OpenStack 的主流后端存储系统 Ceph[13] 采用动态子树划分方法，将命名空间

树分解为若干子树,分布到不同的元数据服务器节点,并根据热点动态调整,这使得 Ceph 能够在运行时动态调整存储资源,以支持多变的数据存储和处理需求. 同时,这种机制能够根据节点的负载自动将数据和请求均衡地分配到各个节点上,避免单节点的过载和瓶颈问题,有助于提高整个集群的性能. 然而,动态子树仍存在一些不足. 首先是子树迁移过程可能会导致业务抖动,影响集群性能. 其次,负载较高的节点在迁移过程中负担加重,影响系统的稳定性. 而且,动态子树分区需要实时监控和调整,这增加了系统的复杂性和运维难度. 动态哈希划分[14]解决了静态哈希在节点变动时的重分布问题,但在支持命名空间局部性方面同样存在不足.

随着文件规模达到千亿级以上,为了高效、灵活地管理其庞大的元数据,业界开始探索元数据服务和元数据存储分离的架构. 一些研究工作[15-16]尝试引入分布式数据库来作为元数据存储底座. HopsFS[15]使用分布式数据库 NDB 来存储元数据,元数据按照父目录唯一标识符(ID)散列到不同的无状态命名空间节点(NameNode),移除了 HDFS 全局锁,相比于 HDFS 实现了 37 倍元数据存储和 16 倍吞吐量的提升. 但 NameNode 和后端数据库之间的交互致使延迟上升,同时哈希机制也使元数据的本地性被破坏,导致 listdir, rename 等操作性能不佳. FileScale[16]则采用了数据层、缓存层和代理层的 3 层分布式架构取代 HDFS 中的元数据管理. 在数据层,FileScale 将元数据存储在分布式数据库管理系统(distributed database management system, DDBMS)中,利用 DDBMS 完善的集群分区和数据复制功能,以确保数据的高可用性和一致性. 为了缓解客户端与 DDBMS 之间交互的网络延迟,FileScale 将 NameNode 作为缓存层. 代理层按照路径前缀进行分区,以减少单个分区中事务的数量,从而在缓存命中时实现了与 HDFS 相当的性能. 然而,当缓存未命中时,性能会出现明显下降,且分区事务的性能也不尽如人意.

由于键值存储具备高性能和可扩展性的优势,近年来一些研究工作[17-20]尝试使用键值存储作为元数据存储底座来提升元数据服务性能. Facebook 公司推出的 Tectonic[17]将元数据存储到强一致的分布式键值存储系统来支持水平扩展,元数据分为命名空间、文件和块 3 层,均使用哈希分区策略. Tectonic 使用分布式键的部分前缀做分区,使得同一目录下的所有元素都位于同一个分区下,便于执行 listdir 操作或目录内的元数据操作. 同时,同一个文件下所有的块的元数据也都位于同一个分区,进一步提高了操作的效率. 但是,Tectonic 并不提供跨目录 move 操作的原子性,在文件系统扫描操作的友好性方面也存在不足. 为了充分发挥键值存储的高性能优势,LocoFS[18]将文件元数据和目录元数据解耦,采用单独的目录管理服务器管理目录元数据以减少元数据内部网络延迟. 文件元数据散列到多个文件管理服务器,并且将文件元数据划分成访问元数据和内容元数据以便以键值友好的方式进行访问. InfiniFS[19]沿用了这一思想,并进一步将目录的访问元数据和内容元数据解耦,实行细粒度的哈希分区,从而解决了 LocoFS 单目录管理服务器的元数据瓶颈问题. 尽管这 2 个工作很好地发挥了键值存储的性能优势,但由于哈希分区的影响,其分布式事务的处理能力受到了一定程度的制约. CFS[20]在元数据解耦的基础上,将目录和文件元数据分别存放在分布式键值和本地键值存储中,通过范围分区策略来优化元数据写操作的临界区,并结合分区内的无锁化操作来消除分布式事务,除了跨分区的重命名操作外,大部分文件操作不再需要分布式事务的支持. 然而,CFS 的范围分区策略是基于目录的,这在处理多层级路径解析时会导致客户端与服务器之间进行多次交互,从而增加了请求延迟. 鉴于路径解析是大多数文件访问操作的必经步骤,这种延迟成为了阻碍系统性能提升的一个主要瓶颈. 上述这些研究工作[17-20]都聚焦于修剪元数据结构来适配通用键值存储系统,但它们缺乏针对文件语义的键值优化,通用键值系统的数据结构和接口不能很好地服务于文件系统. 更重要的是,采用分布式键值存储的系统还需要面对元数据服务与存储底座之间频繁交互带来的网络开销问题.

应对大模型训练场景下海量元数据问题的另一类方法是文件合并技术. 通过在数据准备阶段将多个小文件合并为较大的单一文件,可以减少数据集中的文件数量,从而降低训练阶段对文件元数据规模和访问性能的要求. Google 公司的 TFRecord[21]格式通过将多个训练样本封装成单一文件,减少了元数据访问和 I/O 操作,实现了训练数据的高效批量加载. Nvidia 提出的 WebDataSet[22]则通过将大型数据集分割打包成多个较小的 TAR 文件,并允许自定义数据加载和预处理流程,以适应不同的训练需求和模型架构. 然而,将现有数据集转换为 TFRecord 或 WebDataSet 需要额外的工作,并且会延长数据预处理时间. 此外,合并后的文件不支持随机访问和更新,

这增加了生产数据高速流动的难度，并使训练阶段的细粒度 shuffle 操作变得更加复杂.

综上所述，当前的元数据管理技术在不同方面均存在优势和挑战. 静态子树分区策略能较好地保留单机性能，但其可扩展性差且不支持跨分区操作，这在处理大规模数据时尤为不足. 动态子树分区策略通过动态调整提高了存储效率和性能，但局部热点的形成可能导致业务抖动，影响系统稳定性. 哈希分区策略虽然实现了负载均衡，但牺牲了目录树的局部性，导致了查找速度降低，影响用户体验，小文件合并方案不够灵活，而且会产生额外的开销. 在元数据存储底座的选择上，当前的研究趋势倾向于采用分布式数据库或键值存储. 这些系统能够平稳扩展元数据容量和访问性能，为大规模文件系统提供有力的支撑. 然而，网络延迟高的问题仍然是其面临的主要挑战之一. 此外，通用数据库和键值存储并非专为文件系统元数据设计，在满足元数据访问需求方面存在局限性. 针对上述问题，本文在元数据分区策略、元数据结构设计与缓存策略、元数据存储底座设计等方面进行了优化和改进，以支持大模型场景对千亿级文件存储和高效访问的需求.

2 系统架构

为了满足大模型场景对千亿级文件存储和高效访问的需求，本文实现了一种高性能可扩展的分布式元数据管理机制 ScaleFS.首先在架构上将元数据目录操作和属性访问功能解耦，把元数据划分为目录树元数据和属性元数据 2 部分，分别由不同的元数据服务器集群进行管理. 解耦的架构可以更好地对目录操作和属性元数据管理进行针对性优化，在实现可扩展的大规模元数据存储的同时，显著提升元数据访问性能.

ScaleFS 系统架构如图 2 所示，主要包含客户端、文件块存储服务、目录树服务（directory tree service, DTS）和属性元数据服务（meta data service, MDS）四个关键组件.

客户端是应用程序访问文件系统的代理，提供 HDFS 和可移植操作系统接口（portable operating system interface of Unix, POSIX）语义的文件操作接口，如 open, close, read 和 write 等. 客户端包含文件语义解析模块和缓存模块，其中语义解析模块负责将文件操作解析为内部指令，而缓存模块主要用于存储部分 DTS 的目录树信息，从而加速查找过程.

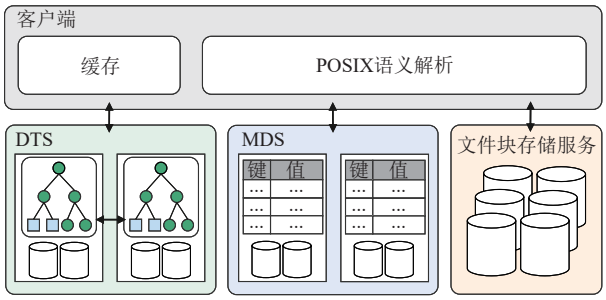


Fig. 2 ScaleFS system architecture

图 2 ScaleFS 系统架构

文件块存储服务集群负责实际文件的存储. 文件按照固定大小切块，并以块的形式存储在磁盘上. 这些数据块可以是原始文件块，也可以是经过纠删码处理的编码文件块. 文件块的位置等元数据信息存储在 MDS 集群中，文件块存储服务是无状态的，可通过增加节点数量实现容量和访问性能的线性扩展，为客户端提供可靠的文件读写服务.

DTS 集群负责目录树元数据的管理. 目录树元数据以“父目录 ID+文件名”为关键字，存储了目录或文件的自身 ID、权限和类型等核心信息. 对于目录，其元数据还额外包含了下一跳 DTS（*nextdts*）字段，用于记录其子目录和文件归属的 DTS 节点信息. 全局目录树依据路径深度分层分布到不同的 DTS 节点上，同时结合父目录关系和节点负载进行分区分配，实现了目录树的负载均衡和高扩展性，为客户端提供高效的目录解析和查找功能.

MDS 集群负责管理和存储除目录树元数据之外的所有目录和文件元数据，统称为属性元数据. 属性元数据根据目录或文件 ID 进行哈希分布，确保元数据在 MDS 集群中均匀分布，以支持海量元数据的扩展和高效访问. 对于新建的目录或文件，DTS 集群会分配 ID，并通过该 ID 将目录树元数据和属性元数据关联. MDS 通过精细化的元数据结构设计和优化的缓存机制，以及构建面向文件语义优化的键值存储系统，为客户端提供高性能、高并发和高可靠的元数据访问服务.

在系统的可用性和数据一致性方面，DTS 和 MDS 都采用了多节点主备架构设计. 数据分区以 3 副本的形式存储在不同节点，以确保数据的可靠性. 集群中的节点按数据分区组成复制组，每个节点承载多个复制组，维持节点间数据与负载的均衡. 在复制组内部，通过类 Raft[23] 协议机制实现同步，从而确保数据的强一致性.

通过上述精简的元数据架构设计，单台 DTS 节

点能够管理更多元数据,实现了目录树元数据单节点性能的垂直扩展,同时保留了目录树的本地性.在实际应用中,文件名长度在目录树相关元数据中占比较大,以文件名平均长度 64 B 为例,256 GB 内存可容纳 36 亿条元数据;即便在文件名平均长度为 256 B 的极端情况下,也可容纳 10 亿条元数据.为确保元数据的持久性和可靠性,ScaleFS 将元数据持久化存储于服务器本地磁盘,并通过键值接口访问.若将 10% 的热点数据加载至内存,则单台 DTS 服务器可支持数百亿条元数据的管理,而 ScaleFS 集群则能够满足千亿级规模文件的存储与管理需求.

目录树元数据和属性元数据解耦的设计,实现了目录树查找与属性访问的互不干扰,显著提升了二者的性能.将具有相同父目录的目录树属性放在同一 DTS 节点上,同时结合深度和广度对目录进行分层分区存放,不仅优化了目录路径的查找性能,还提升了目录树元数据的扩展性.此外,由于属性元数据被均匀哈希分布到多个 MDS 节点,单节点负载得到有效分担,从而避免了因访问超大目录引发的热点瓶颈问题.同时,ScaleFS 针对 MDS 不同的访问接口和元数据属性,设计了细粒度的缓存策略以提升缓存效率.这些设计确保了 ScaleFS 在处理大规模文件存储和访问请求时能够保持高效和稳定.

3 关键技术

本节逐一介绍 ScaleFS 的 3 个关键技术,包括深度与广度均衡的目录树管理策略、访问模式友好的细粒度属性元数据结构设计和面向文件语义优化的键值存储.

3.1 深度与广度均衡目录树管理策略

分布式元数据服务系统中,目录树结构和分区策略共同决定了路径解析的效率和负载的均衡.在 HDFS 的文件操作中,除了基本的 read, write, close 操作,其他操作的元数据操作均依赖于路径解析的先期处理.因此,提升目录树查找性能对整体元数据操作性能的提高至关重要.接下来详细阐述目录树在 DTS 集群的分层分区策略与路径解析机制.

3.1.1 分层分区策略

随着文件数规模的扩大,文件系统目录树深度显著增加,超过 10 级深度的目录占比高达 60%,甚至约 15% 的目录深度达到 15~20 级[19].传统的逐层查找的路径解析方法在这种情况下变得异常耗时,需要频繁地与分布式系统中的多个元数据服务器交互.静态子树方法在可扩展性上存在局限,而动态子树方法则可能导致性能波动.哈希虽然能够均衡分配数据,但却牺牲了数据局部性,从而增加了查找延迟.这些缺陷在路径深度增加时愈加突出.

针对这一问题,ScaleFS 设计了一种深度与广度均衡的分层分区管理策略,将目录路径进行分层管理,每层管理不同深度级数的连续路径,根据负载情况将不同分层分配到不同 DTS 节点上,这样就可将路径深度较大的目录树均衡分布在 DTS 集群节点.具体而言,当创建文件或者目录时,ScaleFS 首先查找其父目录所在的 DTS,然后根据其路径深度计算其是否超越分层.如果新创建的文件或目录没有超越分层,则直接将其分配到其父目录所在 DTS 服务器.若超越分层,则为其在集群中动态选择负载最轻的一个 DTS 服务器作为下一层,并将所选择的 DTS 记录到父目录元数据中.此后,该父目录下的所有子目录和文件均被分配到这个 DTS.按此方式,目录树被横向划分成小的子树分片,这样不仅保持了元数据的局部性,还实现了各节点间的整体负载均衡.

鉴于访问频率的非均匀性,接近根节点的目录比深层路径目录被访问得更频繁,ScaleFS 实现了一个非均匀的目录树分层结构,其中每层管理的路径级数会随着层数的增加而递增,旨在实现 DTS 集群访问负载的均衡.通过一个可调的系数 N, DTS 集群能够灵活而高效地管理目录树,N 值可以是当前层数,或者是层数与一个自然数的和,或是它们的乘积.如图 3 所示,设 N 等于当前层数,则每层管理的目录树深度为 2^N(N=1, 2, …).根目录位置固定,第 1 层 DTS 存储 2 级目录,第 2 层 DTS 存储 4 级目录,依此递增.根据路径可以确定任一目录或文件所在的层数.当在一个空目录下首次创建子目录或者文件时,若超过当前所管理的路径层次,其父目录所在的 DTS 将负责为其分配负载最轻的 DTS.此后,该目录下的其他子目录或者文件将直接存储到这个已分配的 DTS 中.例如,图 3 中目录 G 和文件 $F2$ 由于路径超过了第 2 层,被分配到第 3 层,并且因为它们属于同一个父目录 F,所以 G 和 $F2$ 均被分配到同一个 DTS3.N 的大小直接影响系统性能.N 值越大,路径解析所需的网络跳数越少,越能提高查找效率;而 N 值越小,则系统的均衡性越好,越有助于避免热点.在路径解析的过程中,从根目录查找是常见的操作,这容易导致近根热点,如图 3 中 DTS1.为了解决这一问题,ScaleFS 采用客户端缓存近根目录的策略,并利用最近最少使用(least recently used, LRU)淘汰机制进行管理.这

不仅缓解了近根热点问题，同时也减少了查找的网络延迟，从而提升了整体性能. 为了确保缓存的一致性，客户端消息中会携带用于路径解析的元数据版本号，版本号由服务端执行校验，如果版本号不匹配则返回失败，客户端则会失效该缓存记录并重新向根 DTS 发起查询. 由于近根目录的变化通常较少，重新查询的代价对整体性能的影响较小.

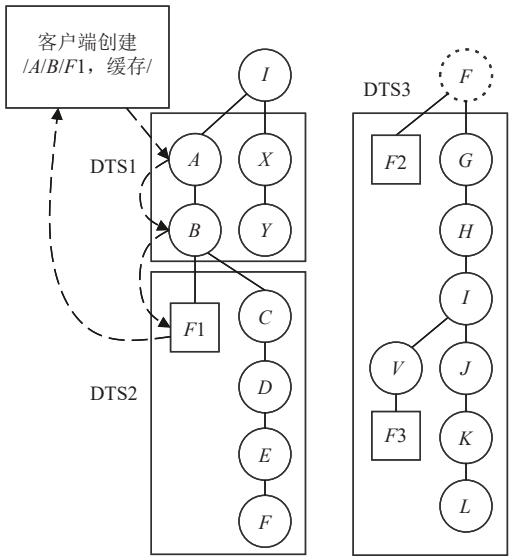


Fig. 3 Hierarchical partitioning strategy of directory tree

图 3 目录树分层分区策略

3.1.2 路径解析机制

传统文件系统在路径解析时，需根据父目录 ID 加子目录或文件名称逐级查找目录层级，直至找到所访问的目录或文件. 分布式文件系统由于目录树被分区存储，客户端和元数据服务器之间需要多次交互才能完成路径解析过程，如 HopsFS[15]，InfiniFS[19] 等. 图 3 所示客户端在/*A*/*B* 下创建文件 *F*1 时，需依次查找/，/*A*，/*A*/*B* 的 ID，创建/*A*/*B*/*F*1，与元数据服务器进行了多达 4 次交互，这导致了较大的时延开销.

相比之下，ScaleFS 采用了深度与广度均衡的目录树管理策略，并在路径解析中采用了全路径环路查找优化. 客户端与 DTS 之间建立了全链路链接，DTS 节点记录了其子目录的 DTS 地址，因此当 DTS 集群接收到客户端的全路径操作请求后，可以在 DTS 节点间进行接力查找，并由最终的 DTS 完成元数据操作响应客户端请求，从而避免了与客户端多次交互带来的延迟. 以创建文件/*A*/*B*/*F*1 为例，ScaleFS 的文件创建流程如算法 1 所示.

算法 1. 文件创建流程.

① *DTS.Create*(/*path*/*filename*)

② *subpath, nextdts* ← *Lookup*(*path*);

③ if *nextdts* ≠ *self* /**subpath* 不属于本 DTS*/

④ *nextdts.Create*(/*subpath*/*filename*);

⑤ else

⑥ *key* ← *InsertFile*(*filename*);

⑦ *Client.CreateOnMDS*(*key*);

⑧ end if

⑨ *Client.CreateOnMDS*(*key*)

⑩ *mds* ← *Hash*(*key*);

⑪ *mds.Create*(*key*);

⑫ *MDS.Create*(*key*)

⑬ *InsertFile*(*key*);

客户端被实现为始终缓存根目录及其子目录的 DTS 节点 ID，因此可以直接确定/*A* 位于 DTS1.客户端首先将创建消息发送给 DTS1，DTS1 收到 Create 消息(行①)，先在内部解析路径/*A*/*B*，直至路径最末端或者超出本 DTS 管理的目录层级，获取下一跳 DTS 节点 *nextdts* 和余下未解析的路径 *subpath*(行②)，比如本例中解析到 *B* 的 *nextdts* 为 DTS2，*subpath* 为空. 随后，判断 *nextdts* 是否为本节点，若不是(行③)，通知下一跳 DTS 创建文件(行④). 下一跳 DTS 将继续解析 *subpath* 并判定 *nextdts* 等于本节点(行⑤)，在本节点创建文件并分配 *key*(行⑥)，然后通知客户端在 MDS 创建文件(行⑦). 接着，客户端再根据 *F*1 的 *key* 进行哈希计算得出 MDS 的位置(行⑩)，并通知该 MDS 创建文件(行⑪). MDS 添加文件记录，创建文件完成(行⑫⑬). 整个过程中，客户端仅与 DTS 集群和 MDS 集群各进行 1 次 RPC 交互，DTS 集群节点间通过环路查找和闭环回复实现信息传递，只有在目录跨越层级时才向下一跳 DTS 节点发送消息. 因此，整个过程中客户端与 DTS 集群的交互次数明显减少，DTS 集群节点间的消息发送也更加高效，相较于传统的查找过程，显著减少了路径解析时延.

基于分层分区策略，路径解析的网络跳数是固定且可预测的. 这一策略在确保目录树分布均匀的同时保持了元数据的局部性. 通过合理调整配置的系数 *N*，显著降低了元数据操作跨 DTS 访问的频率，进而减少分布式事务的数量，从而在系统均衡性和局部性之间实现了理想的平衡. 此外，DTS 之间的环路查找与闭环回复的机制，也很大程度上减少了路径解析过程中的交互次数和耗时. 这一策略不仅提高了 ScaleFS 路径解析的效率，也增强了其可扩展性.

3.2 细粒度属性元数据设计

在单个元数据服务支持百亿级文件元数据时，由于内存容量的限制，只能缓存部分元数据，而大部

分元数据则存储在磁盘上. 当需要读取元数据时, 需要从磁盘加载到内存中, 缓存的命中率对元数据操作的延迟具有较大影响. 由于 MDS 承载了大量元数据的存储功能, 优化其缓存策略能很大程度上提升访问性能.

通过分析生产环境中多模态大模型训练数据加载过程的存储负载, 得到主要操作如表 1 所示, open, read, filestatus, listdir 操作占据了所有操作的 92%. 其中, open 和 read 操作主要访问的文件属性包括文件大小(*Size*)、访问时间(*Atime*)、块信息(*Block*)、软链接数(*Nlinks*)以及存取策略(*StoragePolicyId*). filestatus 操作则主要关注文件 *Size*、*Atime*、修改时间(*Mtime*)和创建时间(*Ctime*). 而 listdir 操作通常只需要访问 DTS 上的 ID 元数据, 只有在执行详细的目录列表(如 ls -l)时, 才会访问 MDS 上的目录属性元数据, 如 *Size*, *Atime*, *Mtime* 等.

Table 1 File Operations of Production Environment Load

表 1 生产环境负载的文件操作

操作	次数	占比/%
open	17 928 310	15.78
write	2 830 486	2.49
read	18 304 926	16.11
listdir	51 838	0.05
filestatus	68 271 469	60.10
delete	5 677 722	5.00

进一步分析各元数据属性的数据类型, 这些属性均为固定大小的整型数据, 占用内存较小. 然而, 这些属性仅是属性元数据的一小部分, 每次从磁盘读取完整元数据以访问这些属性的开销非常高. 同时, 将整条元数据全部保存在内存中则会造成内存空间的浪费. 因此, ScaleFS 将这些属性列为热属性, 并在内存中进行缓存, 以保持较高的元数据缓存命中率.

ScaleFS 采用本地键值存储来管理元数据, 并针对目录和文件的热属性构建了相应的热属性表, 这些表专为元数据操作设计, 能够实现快速访问. 完整的属性元数据则存储在磁盘上, 以确保数据的持久性. 通过这种设计, 频繁访问的热属性尽可能保留在内存中, 减少对磁盘的 I/O 访问, 从而加速读取和写入操作. 表 2 详细列出了 MDS 冷热元数据分类. 热属性共占用 31 B, 而冷属性则超过 78 B, 而且冷属性中还包含变长字段, 这些字段在实际应用中可能非常庞大, 例如 *Xattrs* 的最大值可达 16 KB.

Table 2 Classification for Hot and Cold Metadata of MDS

表 2 MDS 冷热元数据分类

热属性		冷属性	
名称	字长/B	名称	字长/B
Size	8	*Xattrs*	>32
Nlinks	4	*Acl*	>4
Block	4	*BlockSize*	8
Mtime	4	*FileUC*	4
Atime	4	*BlockType*	2
Ctime	4	*ECPolicyId*	4
Type	2	*NsQuota*	8
StoragePolicyId	1	*DsQuota*	8
		TypeQuotas	8
		⋮	
合计	31	合计	>78

从表 3 可见, 热属性覆盖了大多数文件操作涉及的元数据属性, 将这些属性加载到内存中可以显著提升元数据的访问速度. 同时, 系统确保所有属性都完整保存在磁盘上, 并通过键值接口灵活访问内存和磁盘中的元数据属性.

Table 3 Metadata Attributes Commonly Accessed in File Operations

表 3 常见文件操作访问的元数据属性

操作	热属性	冷属性
mkdir	√	
rmdir	√	√
filestatus	√	
delete	√	√
create	√	
open	√	
statdir	√	
listdir	√	
read	√	
write	√	√

当 MDS 需要读取元数据时, 如果读取的元数据属性均为热属性, 则系统会首先访问内存中的键值表. 若能在内存中命中, 则直接读取元数据, 实现快速响应. 若内存中未命中, 系统将转向磁盘读取, 并将该记录加载到内存热属性表中, 以便后续快速访问. 如果读取的元数据属性均为冷属性, 则系统会直接从磁盘读取, 避免不必要的内存查找. 如果读取的元数据同时涉及冷热属性, 系统首先尝试从内存中读取热属性. 若热属性命中, 再去磁盘读取冷属性;

若热属性未命中，则直接去磁盘读取全部属性. 这种策略确保了读取数据的可靠性和完整性，避免了因内存持久化周期等原因造成的数据不一致问题.

MDS 在更新元数据时，若更新的元数据属性均为热属性，系统会首先访问内存中的键值表. 如果命中，则直接在内存中更新相应属性，并将更改持久化到磁盘. 如果未命中，则从磁盘读取该记录到内存缓存中，更新相关属性后再持久化到磁盘. 若更新的元数据属性均为冷属性，则直接更新磁盘上的记录. 当更新的元数据同时涉及冷热属性时，系统会优先尝试访问内存中的键值表. 如果命中则直接更新内存中的热属性；如果未命中，则从磁盘读取该条记录的热数据进行更新并持久化，随后还需在磁盘上更新冷属性部分. 内存中的热属性还会定期刷写到磁盘以确保持久化，崩溃一致性是由键值操作日志来保障.

在保障元数据一致性方面，MDS 热属性中的 *Size*, *Nlinks*, *Block* 等需要保持绝对正确. 对于同一个文件 ID，同一个线程对这些属性的修改也必须保证时序的一致，ScaleFS 把这些属性放在相邻的位置，并对它们使用单独的锁进行管理. 而 *Ctime*, *Atime*, *Mtime* 仅需记录最后一次更新，因此无需加锁，可通过原子操作完成更新. 在文件读写时，如果涉及父目录的 *Atime* 或 *Ctime* 变更，可能需要多个 MDS 协同处理，但只需保持最后一次更新的值，这类属性可采取异步更新方法，同样不需要加锁. 其余冷属性主要在创建、删除目录/文件时涉及，访问频率较低，锁冲突的概率小，统一采用冷属性锁进行管理.

通过上述设计，ScaleFS 实现了基于访问模式和热度的元数据属性分级存储，使得同等大小的内存能够缓存更大规模的元数据，从而优化了缓存效率. 由于内存的读写速度远超磁盘，缓存操作显著提升了元数据操作的速度. 尽管在处理同时涉及冷热属性的元数据操作时可能增加 1 次磁盘操作，但由于此类操作所占比例极小，因此对整体性能的影响有限.

3.3 面向文件语义的键值存储

元数据存储底座的选择对元数据服务的性能有着重大的影响. HDFS 将元数据持久化到磁盘，并在启动时全部加载到内存，这使得内存容量成为瓶颈. 采用分布式数据库或者键值存储作为存储底座虽然具有可扩展性，但网络延迟开销较大. 此外，通用数据库系统并非面向文件系统语义设计，例如，它们需要对所有操作写日志以保证数据一致性，这增加了事务管理的开销. 而通用的键值存储既无法满足文件系统的特定语义需求，也不支持部分读取和更新，例如 RocksDB[24]，主要面向写优化，但读写放大问题仍未解决，性能难以达到预期. 为了克服这些挑战，ScaleFS 实现了一种面向文件语义的键值存储，不仅增强了文件系统语义的支持，还支持键值部分读取和更新，优化了对大规模元数据操作的磁盘访问效率. 通过这一改进，在保证数据可靠性的同时提升了元数据读写性能.

3.3.1 键值存储设计

ScaleFS 的元数据以键值对的形式存储在键值存储中，采用索引块和元数据块相结合的方式管理. 如图 4 所示，每个元数据块存储多条记录，块内按关键字排序，以支持范围查询. 为了支持大规模元数据的高速查找，ScaleFS 为元数据块建立了索引，维护关键字和元数据块的映射关系，索引也以块的形式存储在磁盘. 系统启动时，部分元数据块和索引块会预加载至内存，随后利用 LRU 策略管理缓存，确保元数据命中率. 由于索引只包含了关键字和元数据块位置，单条记录的长度较小，这允许它们以比元数据块更紧凑的形式存储. 为了优化索引块的查找过程，在内存中基于块的最小关键字构建索引块树. 检索元数据记录的过程为：通过查找索引块树检索目标索引块，然后在目标索引块中有序检索目标关键字（如文件 ID）所对应的元数据块位置，随后读取该元数据块，并在块内定位到目标关键字对应的记录. 由于索引块和元数据块均部分缓存在内存中，因此在最坏情况下，也仅需通过 2 次磁盘操作即可找到所需的元数据记录. 此外，ScaleFS 的元数据更新会写入预写日志（write ahead log, WAL），并同步至其他备用节点，

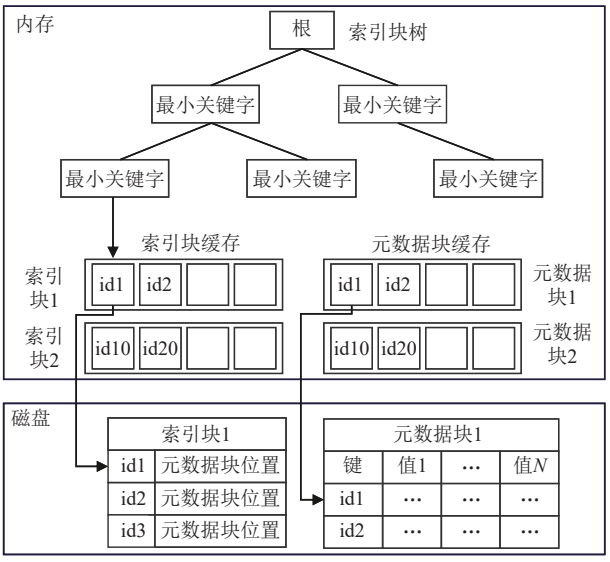


Fig. 4 key-value storage design

图 4 键值存储设计

确保主备节点间的元数据一致性, 从而提供稳定可靠的元数据服务.

3.3.2 支持键值部分读取和更新设计

在大模型数据准备和训练过程中, 涉及到的元数据更新操作主要包括原始数据的清洗转换、训练数据的读取、checkpoint 文件的读写以及日志文件的写入等. 这些操作通常会触发文件和目录的元数据更新, 如文件大小、访问时间等属性的变化. 传统的键值存储系统虽然能通过 Get 和 Put 操作根据键读取和更新值, 但通常不支持值的部分读取和更新. 这意味着, 即使只需要读取值的一小部分, 也必须从磁盘上把整个值读取下来, 导致不必要的磁盘 I/O 开销. 同样, 更新操作也需先读取整个值, 修改后再写回, 造成了读写放大, 这在处理长数据时尤为突出. 针对 MDS 元数据字段较多的特点, 对 ScaleFS 的键值存储底座进行了优化, 支持部分读取和更新功能, 允许元数据服务直接读取和修改指定属性, 同时支持一次读取和修改多个字段. 由于采用了细粒度的元数据设计, 并将定长且经常访问的属性放在一起, 按照 512 B 对齐, 以避免或减轻读放大和读改写带来的写放大问题, 减少频繁的 Put 操作对元数据的修改. 其部分接口示意为:

Get(*table*,*value_list*,*offset_list*,*size_list*),

Put(*table*,*value_list*,*offset_list*,*size_list*),

其中 *value_list* 为更新字段值序列, *offset_list* 为更新字段名偏移序列, *size_list* 为更新字段大小.

3.3.3 面向文件语义设计

在采用通用键值存储的系统中, 诸如树的递归查找等操作通常需要经过多次键值操作才能完成. 通过观察文件元数据服务与键值存储交互的过程, 发现有许多键值操作经常以固定组合的方式出现, 以完成特定功能. 例如, 在创建文件时, 系统需要逐级地通过父目录 ID 和文件名进行 Get 操作以定位最后一级父目录, 随后在该目录下插入一条记录. 同样, 在插入新纪录时, 系统首先会执行 Get 操作以检查记录是否存在, 若不存在则通过 Put 操作插入新纪录. 这些成组的操作涉及多次键值交互, 且频繁发生, 对系统时延产生很大影响.

ScaleFS 对这些流程进行了优化, 将部分文件语义下沉到键值存储层, 由键值层负责实现这些成组的操作对应的功能. 如表 4 所示, ScaleFS 的元数据服务提供了 5 类接口: 递归获取(Recursivefetch)、批量获取(Batchget)、批量删除(Batchdelete)、插入更新(Insertupdate)和批量更新(Batchupdate).

Table 4 Key-Vlue Interface for File Semantics

表 4 面向文件语义的键值接口

键值接口	功能	涉及文件操作
Recursivefetch	递归获取	create\open\filestatus\setacl \delete\statdir\rename 等
Batchget	批量获取	listdir
Batchdelete	批量删除	rmdir
Insertupdate	插入更新	create\mkdir\rename 等
Batchupdate	批量更新	chown

通过这些接口, ScaleFS 对元数据操作流程进行了优化和精简, 进一步提升了访问性能. 以跨分区 rename 操作为例进一步说明这些接口的使用方式, 具体流程见算法 2.

算法 2. 跨分区 rename 操作流程.

/*将/*src_parent*/*A* 改名为/*dst_parent*/*A*/

① *SrcDTS.Rename*(/*src_parent*/*A*, /*dst_parent*/*A*)

② *src_parent_part*1,*src_parent_part*2,*srcdts* ← *Recursivefetch*(*src_parent*); /**src_parent_part*1 为本 DTS 管理的路径片段, *src_parent_part*2 为不属于本 DTS 的路径片段, *srcdts* 为下一跳 DTS*/

③ if *srcdts* ≠ *self*

④ Lock *src_parent_part*1;

⑤ *SrcDTS.Rename*(/*src_parent_part*2/*A*, /*dst_parent*/*A*);

⑥ else

⑦ Lock *src_parent*;

⑧ Lock *A*;

⑨ *DstDTS.Rename*(*srcdts*, /*dst_parent*/*A*);

⑩ end if

⑪ *DstDTS.Rename*(*srcdts*, /*dst_parent*/*A*)

⑫ *dst_parent_part*1, *dst_parent_part*2, *dstdts* ← *Recursivefetch*(*dst_parent*);

⑬ if *dstdts* ≠ *self*

⑭ Lock *dst_parent_part*1;

⑮ *DstDTS.Rename*(*srcdts*, /*dst_parent_part*2/*A*);

⑯ else

⑰ Lock *dst_parent*;

⑱ *Insertupdate*(*key*(*dst_parent*, *A*), *A.value*);

⑲ UnLock *dst_parent*,*dst_parent_part*1;

⑳ *SrcDTS.Delete*(*key*(*src_parent*,*A*));

㉑ UnLock *A*;

㉒ UnLock *src_parent*,*src_parent_part*1;

㉓ end if

当客户端需要将/*src_parent*/*A* 重命名为/*dst_parent*/*A* 时，首先尝试从缓存中获取/*src_parent* 目录信息. 若缓存命中，则直接发送消息到相应的 DTS；若缓存未命中，则向根目录 DTS 发送消息. DTS 服务器收到 rename 消息后，首先通过 *Recursivefetch* 解析源父目录，直至路径最末端或者超出本 DTS 管理的目录层级，获取下一跳 DTS 和余下未解析的源父目录路径（行①②），若本 DTS 不是源父目录所在 DTS，则锁定本服务器所属的路径片段（行④），并通知下一跳 DTS 继续 rename 操作（行⑤）. 若本 DTS 是源父目录 DTS，则对源父目录和文件 *A* 加锁（行⑦⑧），随后从根目录或者缓存的目的 DTS 开始解析目的父目录（行⑨）. DTS 递归解析目的父目录并锁定所属的路径片段（行⑪~⑮），直至找到目的 DTS（行⑯），锁定目的父目录（行⑰），并添加文件 *A* 的记录（行⑱）. 之后，释放之前所有锁定的目的父目录或者路径片段（行⑲）. 源 DTS 删除源父目录下的文件 *A* 记录（行⑳），解锁文件 *A*（行㉑）并释放之前锁定的源父目录和路径片段（行㉒），rename 操作完成. 由于跨分区的目录树服务需要分布式事务管理，ScaleFS 采用了基于 RPC 的 2 阶段提交（two-phase commit, 2PC）协议来保证事务的原子性，从而保障数据的一致性和可靠性.

4 实验及分析

本节对 ScaleFS 与 HDFS 联邦、HopsFS 进行对比实验，以验证 ScaleFS 的优化效果.

4.1 实验环境

本实验所采用的环境配置如表 5 所示，使用 10 台中兴 R5300G4 存储服务器进行测试. 对比系统 HDFS 联邦版本为 Apache Hadoop 3.3.6. HopsFS 版本为最新发布的 HopsFS 3.2.0.4.在 HDFS 实验中，所有服务器均充当主 NameNode 角色. 在 HopsFS 实验中，每台服务器都运行 Hops-NameNode 进程和 NDB-Data 进程，即 Hopsfs 中的元数据代理层和元数据存储层. 而在 ScaleFS 的测试中，每台服务器均运行 1 个 DTS 进程和 1 个 MDS 进程. 所有系统的客户端均与元数据服务器部署在同一主机上，以尽可能在硬件限制下达到最大吞吐.

Table 5 Confiuration of Experimental Environment

表 5 实验环境配置

实验环境	配置
CPU	Intel® Xeon® Gold 6230N CPU @ 2.30GHz (×2)
DRAM	384 GB
NVMe SSD	Intel P4610 1.6 TB (×6)
Network	Mellanox ConnectX-5 100G RoCE
OS	CentOS Linux release 7.9.200 9

为全面评估不同元数据请求的性能，本实验采用元数据测试工具 NNThroughoutBenchmark[25]，并扩展了 setacl, listdir, statdir 等接口. 所有文件相关操作均在空文件上进行，以专注于元数据操作测试.

4.2 文件操作测试

为评估 ScaleFS 单机的文件元数据读写延迟表现，进行了文件操作测试，主要涉及路径解析和读取文件元数据操作. 为了更真实地反映文件元数据读写的延迟情况，使用倾斜负载的测试用例，使访问集中在固定目录，以利用缓存消除路径解析开销的影响. 在测试过程中，每个客户端均并发地访问所有元数据服务器，最终选取每个系统在达到最大吞吐量时的平均延迟作为评估指标.

实验结果如图 5 所示，HopsFS 在各项操作上的延迟均比 HDFS 联邦和 ScaleFS 高 1 个数量级. 该差距主要归因于 HopsFS 将元数据存储在分布式数据库 NDB 中，与 NDB 的频繁交互和 NDB 的事务都对元数据操作时延产生了显著影响. 这一测试结果与文献 [18−19] 中关于 HopsFS 的性能表现是一致的.

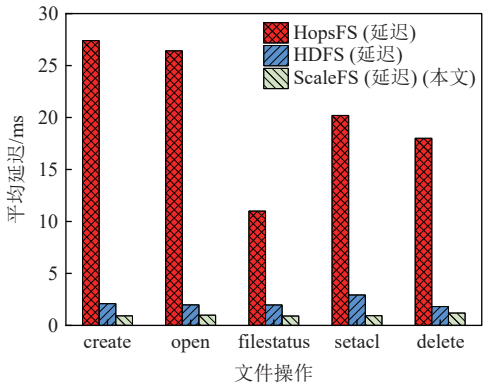


Fig. 5 Performance test of file operation

图 5 文件操作性能测试

相较于 HDFS 联邦，ScaleFS 在 create, setacl, delete 操作上的延迟分别降低了 55.9%, 68.1%, 34.2%. 这些操作涉及文件元数据更新. 此外，ScaleFS 在 open 和 filestatus 操作上的延迟也分别比 HDFS 降低了 50.3% 和 53.9%，这 2 个操作涉及文件元数据读取. ScaleFS 采用本地键值存储元数据，虽然磁盘读写速度要大幅低于内存，但是通过面向文件语义的优化减少了键值存取的次数，通过访问模式友好的缓存策略使

得大小相同的缓存能够容纳更多的元数据热属性,降低了从磁盘读写元数据的概率. 同时,由于 ScaleFS 对元数据热属性实施了细粒度的读写锁机制,进一步降低了整体延迟. 相比之下,HDFS 由于锁粒度过大和软件栈实现复杂,其整体性能受到了一定程度的影响.

4.3 目录操作测试

在千亿级规模的分布式文件系统中,由于元数据分区的存在,目录操作通常需要跨越多个元数据节点,从而显著增加了操作延迟. 常见的目录操作包括 statdir 和 listdir. 其中 statdir 操作主要包括路径解析和目录元数据的点查 2 个过程,其中路径解析的开销占很大,且受目录深度的影响显著. 因此,对不同深度的目录进行 statdir 操作测试可以反映 ScaleFS 的分区策略对路径解析效率的优化效果. 而 listdir 操作主要涉及范围查询,需要与多台服务器交互,通常导致较大的延迟. 为了评估 ScaleFS 的分区策略在路径解析和目录范围查找方面的优化效果,分别采用 statdir 操作和 listdir 操作进行测试.

首先测试了 statdir 操作在不同目录深度下的性能表现. 为了最大程度地减少目录下文件和子目录对测试的干扰,采用空目录进行均衡负载测试. 在测试过程中,关闭了所有系统的客户端缓存,迫使每次操作都必须进行路径解析. 测试中,将 ScaleFS 的 DTS 服务器集群的可调系数 N 配置为当前 DTS 所在层数加 1,每层 DTS 负责存储的路径层数为 2^N(N=2, 3, 4).

测试结果如图 6 所示,在目录深度小于 5 的情况下,ScaleFS 的吞吐量为 HDFS 的 2.24 倍,HopsFS 的 63.33 倍,延迟仅为 HDFS 的 72.04%,HopsFS 的 13.91%. 然而,随着目录深度的逐渐加深,ScaleFS 与其他 2 个系统之间的性能差距有所缩小. 当目录深度达到 20 时,ScaleFS 的吞吐量为 HDFS 的 1.12 倍,HopsFS 的 15.56 倍,延迟则接近 HDFS 的 99.01%,HopsFS 的 55.8%.

这主要是因为 HopsFS 在路径解析过程中需要向 NDB 发送大量查询消息,导致其延迟最高和吞吐量最低. 而 HDFS 由于将目录树完全保存在内存中,因此性能表现相对平稳,但在目录深度增加时仍会出现一定程度的下降. 对于 ScaleFS 而言,当目录深度小于 5 时,所有信息均存储在第 1 层 DTS 中,无需在 DTS 层次间跳转,因此性能表现接近. 但是当目录深度跨越 DTS 分层时,需要在不同层次的 DTS 服务器间跳转,这导致了性能的下降. 尽管如此,这种跳转次数是可以估计的,因为 ScaleFS 的分区策略一定

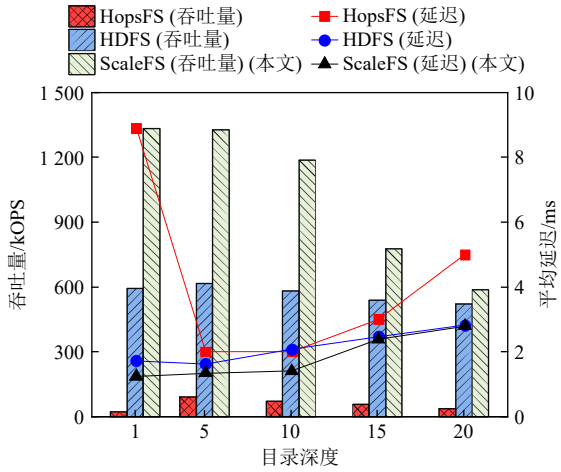


Fig. 6 Performance test of statdir operation

图 6 statdir 操作性能测试

程度上保留了目录树的局部性. 并且通过键值存储的优化,如使用 Recursivefetch 接口加速路径解析的过程,ScaleFS 的表现依然高于 HDFS.此外,ScaleFS 的缓存策略能够将更多的元数据保留在内存中,减少了读取磁盘的概率,从而进一步降低了延迟等待,展现出更优的性能.

为了准确测试 listdir 操作的性能,采用了倾斜负载测试. 总文件数不变、目录规模越小则目录树深度越大,目录规模从 10 到 100 000 进行了列根目录操作的测试. 测试结果如图 7 所示,当目录中包含 10 个文件时,3 个系统的性能均表现较好,且差距不大. HDFS 由于其目录树全部保存在本机内存中表现了较好的性能. ScaleFS 的性能略低于 HDFS,这主要是因为此场景下目录深度较大,ScaleFS 在路径解析过程中需要跳转多个 DTS. 尽管如此,ScaleFS 吞吐量是 HopsFS 的约 12.13 倍,延迟是 HopsFS 的 7.67%. HopsFS 的性能受其元数据哈希机制的影响,列目录操作较慢. 当

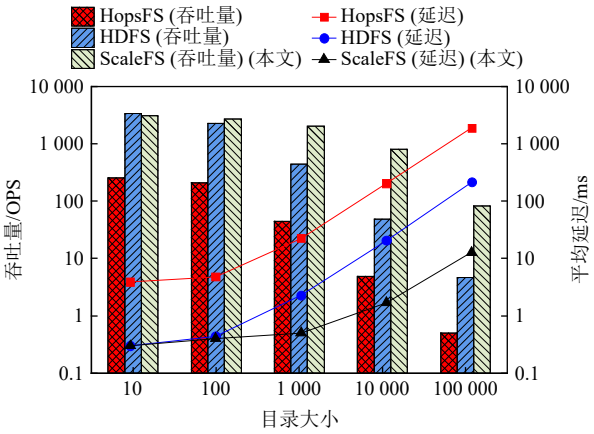


Fig. 7 Performance test of listdir operation

图 7 listdir 操作性能测试

目录规模从 100 增加到 1 000 时，3 个系统的性能均有所下降，但 ScaleFS 开始超越 HDFS. ScaleFS 的吞吐量达到了 HDFS 的 1.17~4.59 倍，HopsFS 的 12.86~410.62 倍；其延迟则相较于 HDFS 降低了 9.59%~78.07%，相较于 HopsFS 降低了 91.65%~97.78%. 这是因为随着目录规模的增加，目录深度相对变小，ScaleFS 路径解析时 DTS 跳转的影响逐渐减小，且采用 Recursivefetch 接口和 Batchget 接口进一步加速查找和读取元数据的过程.

在目录大小为 10 000~100 000 的范围内，3 个系统的性能均大幅下降. HopsFS 哈希对列目录的影响在大目录场景下更加突出. HDFS 和 ScaleFS 在列大目录时需要进行分包发送，这增加了消息交互和延迟等待的时间. 但是 ScaleFS 的吞吐量依然达到了 HDFS 的 16.65~18.04 倍. 这说明 ScaleFS 采用的深度与广度均衡的目录树分区策略，较好地保留了目录树局部性，再结合键值存储的优化，使得 ScaleFS 在目录操作方面相比分布式数据库方案存在显著的优势.

4.4 可扩展性测试

在可扩展性测试中，固定使用 10 个客户端进行测试. 在服务端，HopsFS 由于与 NDB 交互延迟巨大，单台 NameNode 创建 1 亿文件时间达到数小时. HDFS 由于系统局限单台 NameNode 支持文件数 5 亿左右. ScaleFS 单台服务器文件数达到了 100 亿左右. 为了公平比较，3 个系统以 HopsFS 为基准(单台服务器创建 1 亿文件)，逐渐增加服务器数量，采用均衡负载测试并记录各系统的吞吐量和平均延迟进行对比. 由于 HDFS，HopsFS 集群无法创建 1 000 亿文件，本实验仅测试了 ScaleFS 在 1 000 亿文件下的吞吐量和平均延迟.

测试结果如图 8 所示，ScaleFS 整体吞吐量最高，HDFS 次之，HopsFS 和前二者有明显差距. 随着元数据服务器数量的增加，3 个系统的吞吐量均有所提升. HDFS 联邦的目录树采用静态平均划分，服务端之间无需互相协同，因此其所有操作吞吐量均体现了较好的线性扩展趋势. HopsFS 采用分布式数据库 NDB 存储和管理元数据，也达到了线性增长的效果，但其性能受网络延迟影响较大，吞吐量较低、增长缓慢，需要多达数十倍的服务器数量才能达到 HDFS 和 ScaleFS 的水平. 对于 ScaleFS，当元数据服务器数量从 2 增加到 8 时，ScaleFS 所有操作的吞吐量均实现线性扩展. 随着服务器数量的进一步增加，除 filestatus 和 statdir 操作外，其他操作仍保持线性增长. 然而，由于 filestatus 和 statdir 操作的吞吐量较高，导致率先触及操作系统内核资源的上限，未能持续实现线性增长.

在文件操作方面，如图 8(a)~(e)所示，ScaleFS 的 open 操作吞吐量为 HDFS 的 1.06~2.06 倍，而延迟为 HDFS 的 43.84%~50.54%. filestatus 操作的性能表现与 open 操作类似. 这 2 个操作均涉及路径解析和文件元数据读取. 在路径解析阶段，ScaleFS 利用 Recursivefetch 接口显著提升了性能. 在元数据读取阶段，open 操作因需要获取块的位置，其性能略低于 filestatus 操作. 而 HDFS 由于元数据全内存存储，二者性能差异相对最小. 而 ScaleFS 由于在读取块的位置时需要额外访问 1 次磁盘，导致 open 操作与 filestatus 操作的性能差异相对较大. 此外，ScaleFS 的 create 操作吞吐量为 HDFS 的 1.52~4.04 倍，HopsFS 的 20.70~42.75 倍，而延迟为 HDFS 的 12.67%~44.11%，HopsFS 的 2.02%~3.36%. setacl 和 delete 操作也呈现出类似趋势，这些操作均涉及路径解析和文件元数据写操作. HDFS 联邦在元数据写操作时需提交操作日志并更新内存元数据，而 ScaleFS 则需要提交操作日志并将数据写到缓存键值，但 ScaleFS 面向文件语义设计的键值存储，降低了锁粒度和事务管理复杂性，所以提升效果显著.

在目录操作方面，如图 8(f)~(h)所示，ScaleFS 的 listdir 操作吞吐量为 HDFS 的 6.03~7.12 倍，而延迟为 HDFS 的 15.53%~27.90%，statdir 操作的性能提升也十分明显. 这 2 个操作主要涉及路径解析和目录元数据的读操作，这主要得益于 ScaleFS 在路径解析中使用了 Recursivefetch 接口. ScaleFS 的 mkdir 操作吞吐量为 HDFS 的 1.04~1.98 倍，延迟为 HDFS 的 44.83%~99.55%，性能提升不及其他操作. 这主要是因为 mkdir 操作需逐级创建目录并动态生成目录树分区，因此无法充分利用 Recursivefetch 接口加速. 但在创建目录的过程中，Insertupdate 接口仍然对性能提升起到了积极作用.

如图 8(a)~(h)中 ScaleFS 100b(吞吐量)和 ScaleFS 100b(延迟)所示，当集群文件规模增长至 1000 亿时，ScaleFS 的性能相较于 10 亿时有所下降，5 种文件操作、mkdir 操作和 statdir 操作的吞吐量下降了 19.7%~25.4%，延迟增加了 25.7%~34.6%.这些操作主要涉及路径解析和属性读写，随着集群文件规模的增大，缓存命中率降低，磁盘读写频率增加，从而导致性能下降. listdir 操作吞吐量相比 10 亿规模时下降了 50.1%，延迟增加了 189.1%. 这是因为 listdir 操作涉及范围查找，千亿规模时单个目录下文件数剧增，范围查找返回的结果集更大，导致性能下降更为明显. 相对于

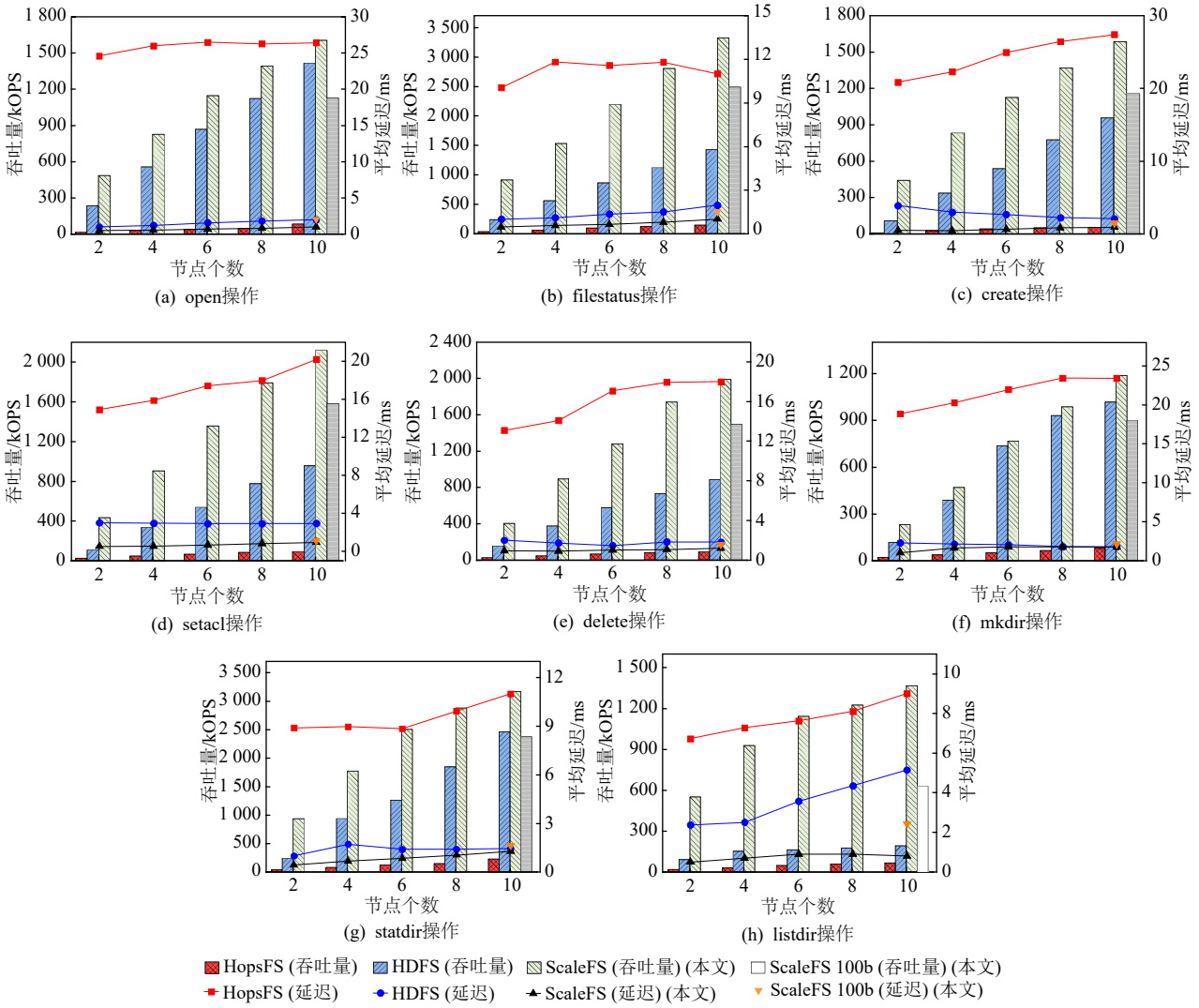


Fig. 8 Scalability test

图 8 可扩展性测试

HDFS 在 10 亿规模下的性能, ScaleFS 在千亿规模下的表现仍具有一定优势. 具体地, Open 和 mkdir 操作的吞吐量下降了 20.16% 和 11.68%, statdir 操作基本持平, 其余 5 种操作的吞吐量依然高出 HDFS 21.03%~228.06%. 因此, 尽管性能有所下降, ScaleFS 在千亿规模下大部分操作的性能优于 10 亿规模下的 HDFS.

总的来说, ScaleFS 在性能方面展现出显著优势, 其表现远超 HopsFS, 性能提升可达数倍至数十倍. 在与全内存的 HDFS 联邦对比中, ScaleFS 也展现了明显的优越性. 此外, ScaleFS 的元数据持久化存储于磁盘, 不受单机内存容量限制, 单台元数据服务器可容纳的元数据量是 HDFS 的数十倍. 通过增加节点数量, ScaleFS 可实现性能和容量的线性扩展, 成功解决了千亿级文件的元数据管理与存储问题. 这为面向大模型的海量数据存储与高效访问提供了坚实的基础.

4.5 数据加载性能测试

本节采用生产环境中的多模态大模型训练数据集(以 KB 级小文件为主)进行数据加载测试. 由于 HopsFS 在前期测试中读写较慢, 本测试主要以 HDFS 作为参照对象, 评估 ScaleFS 与 HDFS 在 1 个 epoch 训练周期下的性能表现. 此外, 在 HDFS 上补充测试了利用 WebDataSet 聚合小文件的场景, 将数据集预先打包为多个 100 MB 大小的 TAR 文件, 并通过 WebDataSet 插件在训练时读取并解压为原始文件.

如图 9 所示, 在端到端文件加载过程中, ScaleFS 的文件操作性能是 HDFS-WebDataSet 的 1.19 倍; 在元数据吞吐率方面, ScaleFS 展现出显著优势. 由于 HDFS 的设计偏向于面向大文件, 在小文件场景下, 其文件加载和元数据访问效率较低. 尽管 HDFS-WebDataSet 通过将小文件合并为 TAR 文件大幅减少

了文件数量，降低了对文件系统元数据的访问需求，但这一方案也引入了解压等额外开销，影响了文件加载性能. 相比之下，ScaleFS 在直接访问小文件时虽然产生了大量元数据操作，但得益于其高效的元数据访问机制，最终实现了较高的文件读取速度，获得了优于 HDFS-WebDataSet 的效果. 同时，ScaleFS 减少了数据预处理的复杂性和开销，在生产环境中具备更高的适用性和灵活性.

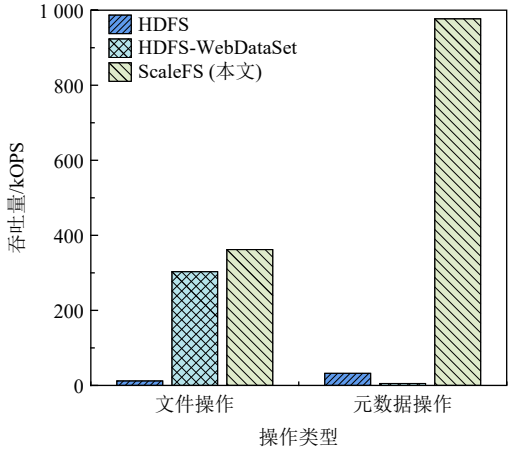


Fig. 9 Performance test of data loading

图 9 数据加载性能测试

5 总结与展望

本文探讨了大模型的构建和应用对存储系统在数据规模和访问性能方面带来的严峻挑战，分析了传统分布式文件系统在应对大模型场景时的局限性和瓶颈. 针对这些问题，提出并实现了一种高性能、可扩展的分布式元数据设计 ScaleFS. ScaleFS 通过解耦目录元数据与属性元数据的架构设计，并采用基于目录深度和广度均衡的分层分区策略，有效提升了目录访问局部性，减少了路径查找延迟，并实现了高效的负载均衡，避免了访问热点问题. 此外，ScaleFS 结合细粒度元数据设计、高效缓存策略以及面向文件语义的键值存储底座，大幅提升了元数据访问效率. 实验结果表明，ScaleFS 的吞吐量达到了 HDFS 的 1.04~7.12 倍，而延迟仅为 HDFS 的 12.67%~99.55%. 特别是在千亿级规模下，ScaleFS 的大部分操作性能优于 HDFS 在十亿级文件规模下的表现，能够更好地满足大模型场景对千亿级文件存储及高效访问的需求.

展望未来，随着硬件技术的快速发展，新型架构和存储介质将在大模型场景中发挥更重要的作用. 基于 CXL(compute express link)的硬件架构有望重塑 GPU 与本地存储设备之间的数据交互模式，RDMA (remote direct memory access)技术在分布式训练中的应用将进一步提高数据传输效率，而非易失性高速存储介质则可能为大模型训练中 checkpoint 容错机制及数据访问加速提供创新解决方案. 我们将持续致力于探索大模型场景下的技术创新，优化存储系统的性能和效率，以应对日益增长的计算和存储需求，推动大模型技术的进一步发展和落地应用.

作者贡献声明：尚碧筠、韩银俊提出论文思路，负责论文撰写和方案设计；肖蓉、陈正华参与方案实施和实验验证；屠要峰、董振江指导方案设计和论文撰写.

参 考 文 献

[1] Achiam J, Adler S, Agarwal S, et al. GPT-4 technical report [J]. arXiv preprint, arXiv: 2303.08774, 2023

[2] Feng Yangyang, Wang Qing, Xie Minhui, et al. From BERT to ChatGPT: Challenges and technical development of storage systems for large model training[J]. Journal of Computer Research and Development, 2024, 61(4): 809-823 (in Chinese)
(冯杨洋, 汪庆, 谢旻晖, 等. 从 BERT 到 ChatGPT: 大模型训练中的存储系统挑战与技术发展[J]. 计算机研究与发展, 2024, 61(4): 809-823)

[3] OpenDataLab. laion5b-downloader[EB/OL]. 2023[2024-05-20]. https://github.com/opendatalab/laion5b-downloader

[4] Young A, Chen Bei, Li Chao, et al. Yi: Open foundation models by 01. AI[J]. arXiv preprint, arXiv: 2403.04652, 2024

[5] Dai Hao, Wang Yang, Kent K B, et al. The state of the art of metadata managements in large-scale distributed file systems—Scalability, performance and availability[J]. IEEE Transactions on Parallel and Distributed Systems, 2022, 33(12): 3850-3869

[6] Shvachko K, Kuang H, Radia S, et al. The Hadoop distributed file system[C/OL] //Proc of the 26th Symp on Mass Storage Systems and Technologies(MSST). Piscataway, NJ: IEEE, 2010[2024-05-20]. https://ieeexplore.ieee.org/document/5496972

[7] Thekkath C A, Mann T, Lee E K. Frangipani: A scalable distributed file system[J]. ACM SIGOPS Operating Systems Review, 1997, 31(5): 224-237

[8] Gibson G A, Nagle D F, Amiri K, et al. A cost-effective, high-bandwidth storage architecture[J]. ACM SIGOPS Operating Systems Review, 1998, 32(5): 92-103

[9] Ghemawat S, Gobioff H, Leung S T. The Google file system[C] //Proc of the19th ACM Symp on Operating Systems Principles. New York: ACM, 2003: 29-43

[10] Wang Yijie, Sun Weidong, Zhou Song, et al. Key technologies of distributed storage for cloud computing[J]. Journal of Software, 2012, 23(4): 962-986 (in Chinese)
(王意洁, 孙伟东, 周松, 等. 云计算环境下的分布存储关键技术[J].

软件学报, 2012, 23(4): 962−986)
[11] Jin Guodong, Bian Haoqiong, Chen Yueguo, et al. Survey on storage and optimization techniques of HDFS[J]. Journal of Software, 2020, 31(1): 137−161(in Chinese)
(金国栋, 卞昊穹, 陈跃国, 等. HDFS 存储和优化技术研究综述[J]. 软件学报, 2020, 31(1): 137−161)
[12] Rodeh O, Teperman A. ZFS-A scalable distributed file system using object disks[C] //Proc of the 20th Symp on Mass Storage Systems and Technologies (MSST). Piscataway, NJ: IEEE, 2003: 207−218
[13] Weil S, Brandt S A, Miller E L, et al. Ceph: A scalable, high-performance distributed file system[C] //Proc of the 7th Conf on Operating Systems Design and Implementation (OSDI'06). Berkeley, CA: USENIX Association, 2006: 307−320
[14] Schmuck F, Haskin R. GPFS: A shared-disk file system for large computing clusters[C] //Proc of the 1st Conf on File and Storage Technologies (FAST'02). Berkeley, CA: USENIX Association, 2002: 231−244
[15] Niazi S, Ismail M, Haridi S, et al. HopsFS: Scaling hierarchical file system metadata using newSQL databases[C] //Proc of the 15th Conf on File and Storage Technologies (FAST'17). Berkeley, CA: USENIX Association, 2017: 89−104
[16] Liao Gang, Abadi D J. FileScale: Fast and elastic metadata management for distributed file systems[C] //Proc of the 2023 ACM Symp on Cloud Computing. New York: ACM, 2023: 459−474
[17] Pan S, Stavrinos T, Zhang Yunqiao, et al. Facebook's Tectonic filesystem: Efficiency from exascale[C] //Proc of the 19th Conf on File and Storage Technologies (FAST'21). Berkeley, CA: USENIX Association, 2021: 217−231
[18] Li Siyang, Lu Youyou, Shu Jiwu, et al. LocoFS: A loosely-coupled metadata service for distributed file systems[C/OL] //Proc of the Int Conf for High Performance Computing, Networking, Storage and Analysis. New York: ACM, 2017[2024-05-20]. https://doi.org/10.1145/3126908.3126928
[19] Lv Wenhao, Lu Youyou, Zhang Yiming, et al. InfiniFS: An efficient metadata service for large-scale distributed filesystems[C] //Proc of the 20th USENIX Conf on File and Storage Technologies (FAST'22). Berkeley, CA: USENIX Association, 2022: 313−328
[20] Wang Yiduo, Wu Yufei, Li Cheng, et al. CFS: Scaling metadata service for distributed file system via pruned scope of critical sections[C] //Proc of the 18th European Conf on Computer Systems. New York: ACM, 2023: 331−346
[21] Google. Tensorflow[EB/OL]. 2024[2024-09-13]. https://www.tensorflow.org/tutorials/load_data/tfrecord?hl=zh-cn
[22] Aizman A, Maltby G, Breuel T. High performance I/O for large scale deep learning[C] //Proc of the 2019 IEEE Int Conf on Big Data (Big Data). Piscataway, NJ: IEEE, 2019: 5965−5967
[23] Ongaro D, Ousterhout J. In search of an understandable consensus algorithm[C] //Proc of the 2014 USENIX Annual Technical Conf. Berkeley, CA: USENIX Association, 2014: 305−319
[24] Dong Siying, Kryczka A, Jin Yanqin, et al. RocksDB: Evolution of development priorities in a key-value store serving large-scale applications[J]. ACM Transactions on Storage, 2021, 17(4): 1−32
[25] Apache. NNThroughoutBenchmark[EB/OL]. 2024[2024-05-20]. https://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-common/Benchmarking.html



Shang Biyun, born in 2004. Undergraduate. Student member of CCF. Her main research interests include big data and machine learning.
尚碧筠, 2004 年生. 本科生. CCF 学生会员. 主要研究方向为大数据、机器学习.



Han Yinjun, born in 1977. Master, senior engineer. Senior member of CCF. His main research interests include storage systems and data management systems.
韩银俊, 1977 年生. 硕士, 高级工程师. CCF 高级会员. 主要研究方向为存储系统、数据管理系统.



Xiao Rong, born in 1983. Master, senior engineer. Her main research interests include storage systems and distributed systems.
肖　蓉, 1983 年生. 硕士, 高级工程师. 主要研究方向为存储系统、分布式系统.



Chen Zhenghua, born in 1983. Senior engineer. Member of CCF. His main research interests include storage systems and distributed systems.
陈正华, 1983 年生. 高级工程师. CCF 会员. 主要研究方向为存储系统、分布式系统.



Tu Yaofeng, born in 1972. PhD, professor. Distinguished member of CCF. His main research interests include distributed systems, big data and machine learning.
屠要峰, 1972 年生. 博士, 研究员. CCF 杰出会员. 主要研究方向为分布式系统、大数据和机器学习.



Dong Zhenjiang, born in 1970. PhD, professor and PhD supervisor. Distinguished member of CCF. His main research interests include artificial intelligence, big data, and information security.
董振江, 1970 年生. 博士, 教授, 博士生导师, CCF 杰出会员. 主要研究方向为人工智能、大数据、信息安全.