引用格式：桂静，王永滨．基于Alluxio的新闻文本存储优化方法[J]．中国传媒大学学报（自然科学版），2023，30(06)：12-18.
文章编号：1673-4793(2023)06-0012-07

基于Alluxio的新闻文本存储优化方法

桂静[1,2]，王永滨[1*]
（1.中国传媒大学媒体融合与传播国家重点实验室，北京 100024；
2.中国传媒大学计算机与网络空间安全学院，北京 100024）

摘要：新闻数据的指数级增长对新闻的高效存储和管理提出了重大挑战。为解决存储与计算分离模式下新闻文本的存储与读访问瓶颈问题，提出了一种基于数据编排平台Alluxio的新闻文本存储优化方法。首先，利用Alluxio对不同存储系统中的新闻文本进行缓存并提供统一命名空间，加速了计算应用的数据访问。其次，为了解决新闻文本存储在远程场景下所面临的读访问性能瓶颈问题，对新闻文本进行了基于目录聚合的合并，并利用最小完美哈希算法对新闻文本元数据构建索引，实现了新闻文本的快速检索。

关键词：分布式系统；新闻数据；小文件存储；缓存；Alluxio
中图分类号：TP31　**文献标识码**：A

An optimization method of news text storage based on Alluxio

GUI Jing[1,2], WANG Yongbin[1*]
(1.State Key Laboratory of Media Convergence and Communication,Communication University of China, Beijing100024,China; 2.School of Computer and Cyber Sciences,Communication University of China, Beijing100024, China)

Abstract: The exponential growth of news data has presented a significant challenge to efficiently storing and managing news. To address the bottleneck issues of news text storage and access in the separation mode of storage and computation, we proposed an optimization method for news text storage based on Alluxio, a data orchestration platform. Firstly, Alluxio was utilized to cache news texts across various storage systems and provided a unified namespace, enhancing data access for computational applications. Secondly, to overcome performance limitations in reading and accessing remotely stored news text, we incorporated directory aggregation for merging the news text and emploied the Minimum Perfect Hash algorithm to index metadata associated with the news text. This enables rapid retrieval of relevant information from the stored news texts.

Keywords: distributed system; news text; small files storage; cache; Alluxio

1　引言

随着互联网以及网络媒体技术的高速发展，各媒体机构每天都产生大量的新闻数据。海量文本、图片、音频和视频新闻数据被存储在不同的存储集群或云服务器上，以便后续结合数据分析以及人工智能等技术应用于网络舆情监控、新闻推荐、虚假新闻检测等领域。

作者简介(*为通讯作者)：桂静(1993-)，女，博士研究生，主要从事分布式文件系统研究。Email：guijing_b@cuc.edu.cn；王永滨(1963-)，男，教授、博士生导师，主要从事网络新媒体技术研究。Email：ybwang@cuc.edu.cn

在计算应用与存储系统的传统紧耦合模式下，媒体应用往往不能访问多个存储系统，从而形成数据竖井[1]（Data Silos）。这种架构现如今已不能够满足大数据的计算需求，于是计算与存储分离的架构模式应运而生。这种模式下，数据竖井不断扩散，新闻数据变成分布式的、解耦的、去中心化的、分散而且混乱的。如何管理存储在不同存储系统中的新闻数据，就成为了新的挑战。数据编排[1]（Data Orchestration）就是为了打破数据竖井而生，抽象了跨存储系统的数据访问，虚拟化了所有数据，并将多源数据整合在一起，通过全局标准化 API 提供给上层应用，极大节约了存储成本以及人工成本。

本文针对新闻文本所面临的“数据竖井”问题以及读访问性能瓶颈，提出了一种基于 Alluxio 的新闻文本存储优化方法。该方法利用数据编排平台 Alluxio 为不同存储系统中的新闻文本提供统一访问接口，同时本文也在 Alluxio 上对新闻文本进行基于目录聚合的合并，并利用最小完美哈希算法[2]（Minimal Perfect Hashing, MPH）对新闻文本元数据进行索引，实现新闻文本的快速检索。

2 研究背景

2.1 Alluxio 概述

Alluxio 是世界上第一个开源数据编排平台，同时又是一个虚拟分布式文件系统[3]（Virtual Distributed File System, VDFS）。它诞生于加州伯克利大学分校的 AMPLab 实验室。Alluxio 与其它分布式文件系统的不同之处是其常被当作分布式共享缓存系统。与它通信的计算应用可以透明地缓存被频繁访问的数据，并享受到其提供的内存级 I/O 速度。它在计算应用（Spark，Tensorflow，PyTorch 等）和各种存储系统（HDFS，Ceph，AWS，OSS 等）中间提供了统一命名空间，使用户能通过统一接口直接访问底层数据。开发人员则不用针对不同存储系统重复编写访问接口。

Alluxio 的系统架构采用了主从（Master/Slave）结构，包含一个 Master 节点和多个 Worker 节点。Alluxio的文件系统服务主要由三个组件完成，分别是：主节点 AluxioMaster、从节点 AlluxioWorker、客户端 AlluxioClient。AluxioMaster 负责响应客户端的访问请求、元数据管理以及数据节点的管理；AlluxioWorker 负责存储数据块；AlluxioClient 负责为应用程序诸如 Spark 或 Tensorflow 提供交互入口。

Alluxio 的系统架构如图 1 所示，AlluxioClient 与 AlluxioMaster 之间的消息通信主要是发送请求获取文件元数据等信息，然后与相应的 AlluxioWorker 进行通信，以执行文件读、写或者删除操作。当进行文件读取操作时，如果被访问的文件正好被缓存在 AlluxioClient 所在本地 Worker 节点，则将会触发本地“短路读”模式，加速数据访问。如果本地 Worker 不存在此文件，则从其它 AlluxioWorker 读取，并将所请求的文件内容复制到本地 Worker。如果 Alluxio 中不存在所请求的文件，则从底层存储系统加载并返回给客户端。Alluxio 除了使用内存作为缓存空间外，还可以使用 SSD 和 HDD 作为挂载的外部缓存空间，从而达到层次存储的目的。

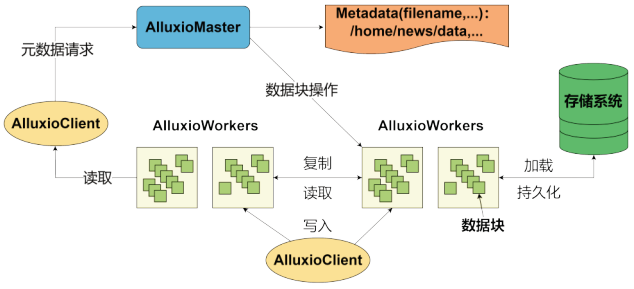

图 1 Alluxio 架构图

2.2 问题分析

新闻文本一般属于小文件类型数据。当前的分布式存储系统大多数都是为了存储大文件而设计的，在存储小文件方面表现不佳，虚拟分布式文件系统 Alluxio 也存在同样问题。据官方数据统计，Alluxio 中的文件元数据对象大小约为 2-4KB，那么 1 亿小文件将产生 200-400GB 的文件元数据。根据 Alluxio 的文件管理机制，所有的文件元数据都被存储在 AlluxioMaster 中。缓存大量小文件，会严重占用 AlluxioMaster 节点的内存，从而导致读性能瓶颈。同时在远程场景下，数据通过网络传输，多客户端频繁地访问文件也会导致 AlluxioMaster 节点的性能瓶颈。因为计算作业任务使用的数据大多是几十、几百甚至是几 KB 的新闻文本数据，与传输大文件（音频、视频）相比，传输同等数据量的小文件时，系统的寻址次数要远远大于大文件，从而产生大量的 I/O 操作。

表 1 验证了 Alluxio 读写不同大小（2KB-4GB）文件时的 I/O 吞吐量变化。

表1 Alluxio文件读写吞吐量

文件大小	读吞吐量(MB/s)	写吞吐量(MB/s)
2KB	4.2	1.3
16KB	32.7	3.7
128KB	321.5	54.8
1MB	1018.0	334.5
8MB	3315.8	1042.2
64MB	5107.7	2795.2
512MB	5769.0	4322.3
4GB	5851.8	4092.8

在验证实验中,使用测试工具DFS-Perf[4],将写模式设置为MUST_CACHE,读取模式设置为NONE_CHCHE。从表中可以看出,读写小文件时,Alluxio的吞吐量相对较低。随着文件大小的增大,吞吐量随之增大,但增幅逐渐减小,达到峰值后下降。在写测试中,4GB文件的吞吐量是2KB文件的3147倍。读测试的吞吐量提升较低,但仍然是原来的1393倍。

对新闻文本进行合并,中间多使用了一次哈希操作,请求文件就多了一次查询操作,从而降低了文件读访问效率。而且当系统响应文件读访问请求时,为了检索单个文件的元数据信息而将整个索引文件加载到Master节点的内存中,这也是访问过程变慢的主要原因。

2.3 相关研究现状

传统的分布式文件系统,例如GFS[5]、Lustre[6]以及HDFS[7],都是为大文件存储而设计的。目前还没有一套通用的用来构建存储海量小文件的分布式文件系统的标准。淘宝的TFS[8]、Facebook的Haystack[9]都是为处理小文件定制的分布式文件系统。TFS基于平面轻量级文件系统IFLATLFS[10],主要是存储小于1MB的小文件,并将文件所需元数据减少至能够完全存放在内存中。Haystack主要是为了解决海量图片的存储问题,将用户图片合并到大文件中,并将文件元数据存储在索引文件中,而索引文件存储在主内存。除了定制分布式文件系统,合并小文件也是优化小文件存储的主要方式。针对小文件问题,HDFS提供了三种默认解决方案:HAR[11]、SequenceFile[12]、MapFile[13]。HAR是对文件进行合并,并通过两个索引文件保存元数据:_index和_masterindex。SequenceFile主要用来解决二进制日志问题。在此格式中,数据被记录为键值对序列。MapFile是一个排序的SequenceFile,其索引允许按键查找。MapFile在查找过程中使用二进制搜索,以将复杂度降低到O(log(n))。但MapFile一旦创建了存档,就不可能添加具有任意名称的文件。由于Alluxio本身的写机制(数据写入后不被修改且静态数据集不能继续添加或删除文件),本文中的文件合并转换过程就借鉴了HAR的思想。由于文件访问多了一层索引查询操作,所以HAR提供了相对较差的访问性能。

LHF[14]是一个在HDFS中处理大规模小文件的解决方案,通过将小文件合并成大文件,并建立一个基于线性散列的可扩展索引来加速查找小文件的过程。但是在进行小文件合并的过程中,需要进行文件排序。HPF[15]将小文件合并成大文件,并直接从索引文件访问元数据,将访问开销降至最低,算法时间复杂度为O(1)。索引文件的构建使用了两个哈希函数。元数据记录通过使用扩展哈希函数分布在索引文件中,元数据记录在索引文件中的位置通过使用最小完美哈希函数存储。本文在索引构建上借鉴了HPF的思想,将此应用于新闻文本存储场景下,并在Alluxio中实现。与HPF不同的是,在元数据的索引构建上我们同时比较了多种最小完美哈希算法:CHD[16]、GOV[17]、BBHash[18],且根据实验结果,从中选择了最优方案。

3 本文方法

在计算与存储分离的集群架构模式下,大数据计算应用的数据一般存储在HDFS、Ceph等分布式存储系统或者AWS、S3和OSS等云服务器中。使用Alluxio作为缓存系统,可以为上层应用提供全局统一的访问命名空间。在这种存储架构下,上层应用只需要访问Alluxio就可以获得所需要的计算数据,而不需要关心数据来自于底层的哪一个存储系统,从而做到了底层数据的资源整合。

图2展示了本文提出的远程应用场景下的新闻文本存储架构,并说明了在Alluxio组件上添加的服务模块,主要包括:文件合并模块、元数据管理模块以及合并文件存储模块。

AlluxioClient是处理用户请求的入口,所以将合并生成模块放在客户端中。合并生成模块中还包括一个文件过滤器,它的作用是识别文件大小小于2KB的文本文件,这类文件都被称为超小文件,这类文件将会与元数据合并存储。因为超小文件的大小与文件元数据大小基本相等,所以将其与元数据一同存储,可以减少文件访问时间,提高文件访问效率。除此之外,剩下的新闻文本则会以目录聚合的方式被合并。

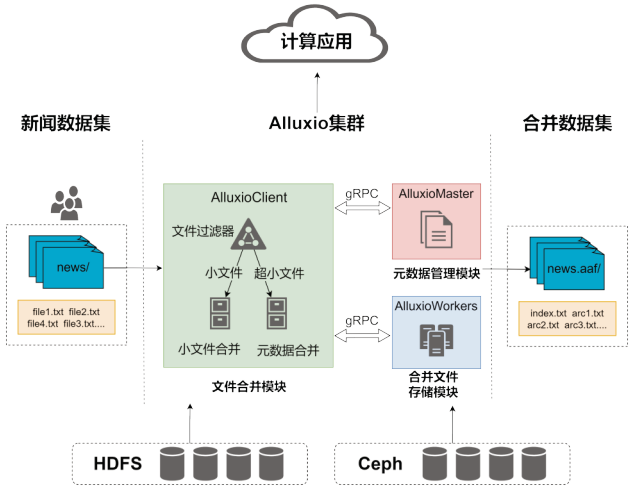

图 2 新闻文本存储架构图

AlluxioWorkers 是实际的数据缓存节点，在每个 AlluxioWorker 中增加一个合并文件存储模块。为了高效管理和快速定位原文件，在 AlluxioMaster 中设计了合并元数据管理模块。通过这些模块，可以轻松实现对合并文件的基本文件操作，如读、写、删除等操作，且所有文件操作都以合并文件为粒度进行。AlluxioMaster 不再需要为每个小文件创建元数据，可以显着减少 AlluxioMaster 的内存使用。如果一个计算节点需要从底层存储读取一个目录，它应该为每个文件发送请求。但有了合并文件，一个请求就可以完成整个过程。批处理中的文件可能不会按顺序存储，甚至分布在不同的 AlluxioWorker 上。合并文件保证内部文件存储在同一个 AlluxioWorker 上，并且位置连续，可以减少 AlluxioWorker 之间数据传输的开销。

3.1 文件合并

文件合并的实现方式是在 Alluxio 客户端实现用户命令 ArchivedCommand，该命令是将同一目录中的新闻文件合并为大文件，以达到提高元数据管理节点内存使用率的目的。文件合并的转换过程如图 3 所示：

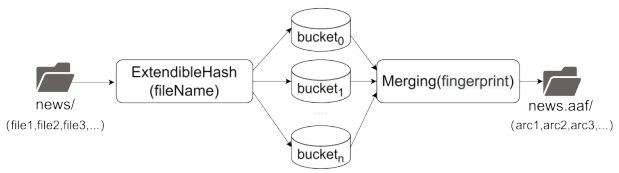

图 3 文件合并转换过程

其中，news/是存储新闻数据的文件目录，经过文件合并，生成新的文件目录 news.aaf/。新的文件目录包括索引文件以及数个合并文件。合并之后，用于检索原新闻文本的信息被称为元数据记录。在创建索引文件之前，需要完全完成合并过程，以便拥有所有元数据记录。此外，在等待合并完成时只将这些记录保存在内存中是不安全的，因为可能会发生网络错误而导致合并失败。为了避免出现这种情况，元数据记录被临时存储在临时索引文件中。当一个新闻文件被合并到一个大文件时，它的元数据记录也被附加到临时索引文件中。因此，临时索引文件在整个合并过程中都保存着元数据记录，还用于确保故障时的恢复。合并过程在所有新闻文件合并完成后停止。

3.2 元数据管理

小文件合并为大文件时，将生成新的文件元数据信息，这些扩展的元数据是原文件在合并后文件中的标识信息，可以称作合并元数据。合并元数据包含四个字段：*file_name_hash*、*file_position*、*offset*、*size*。每个字段的长度是固定的，包含信息如表 2 所示：

表 2 合并元数据信息

字段	长度(byte)	含义
file_name_hash	128	小文件在合并文件中的唯一标识
file_position	16	标记小文件所在的 bucket
offset	16	从合并文件读取小文件内容的精确偏移量
size	16	文件大小

3.3 索引构建

使用最小完美哈希算法对新闻文件元数据条目构建索引，通过访问索引文件直接访问新闻文本元数据，提升读访问性能。

3.3.1 MPH 算法

最小完美哈希是计算机领域的一个基础研究问题。MPH 被广泛地应用于解决内存高效存储以及静态集快速检索等问题，通常被应用于压缩全文索引、计算机网络、数据库、语言模型、Bloom 过滤器[19]等应用。

MPH将集合S的n个键映射到集合$[n]=\{0,...,n-1\}$。生成的数据结构实现了S和$[n]$中的整数之间的一一对应。这样的函数 $f:U->\{0,...,n-1\}$称为最小完美哈希函数(Minimal Perfect Hash Function, MPHF)。

已经有多种MPHF构建被提出应用于大集合,例如:CHD、GOV和BBHash等。这些算法都旨在保持快速计算时间的同时占用很少的空间。

3.3.2 构建过程

新闻文件合并元数据索引构建流程如图4所示。在扩展哈希表构建完成后可根据文件名的哈希值直接定位该文件合并元数据所处的bucket,该bucket中具有多个文件的元数据项,若直接遍历则具有O(n)的时间复杂度。本文对桶中的合并元数据以文件名的哈希值作为键,利用MPHF进行索引构建,在通过MPHF后,将最小完美哈希表和元数据记录项并称为索引项(index),并保存在索引文件index*中。这一部分主要由AlluxioClient端的合并生成模块完成。

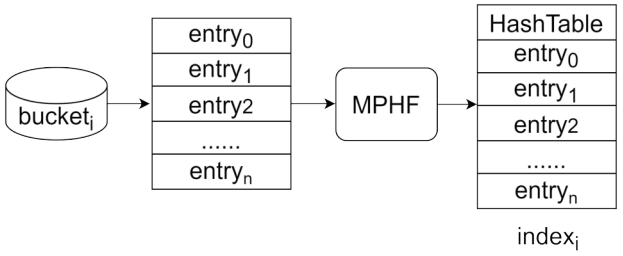

图4 索引构建过程

4 实验

4.1 实验设置及集群配置

4.1.1 实验环境

实验环境具体配置如表3所示:

表3 实验环境配置

名称	参数
CPU类型	Intel® Xeon
CPU个数	4
内存大小	16GB
硬盘大小	1TB
操作系统版本	Linux(Centos7)
JDK版本	jdk-8u144
Hadoop版本	2.7.3
Spark版本	2.4.7
Alluxio版本	2.7.0

4.1.2 集群配置

为了验证方法的有效性,我们以存储与计算分离模式部署了实验集群。其中,计算集群由四台物理机组成,存储集群由三台物理机组成。计算集群负责执行计算任务和新闻文本的缓存,存储集群则负责新闻文本的持久化存储服务。为了模拟远程场景,中间网络带宽被设置为1 M/s。集群拓扑如图5所示:

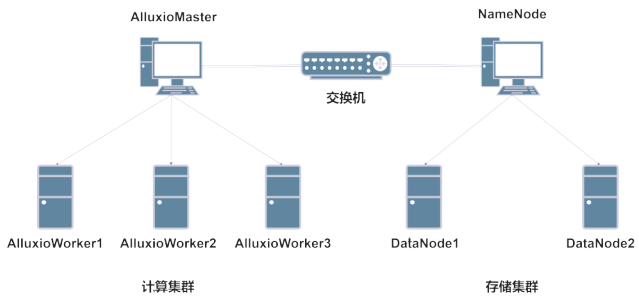

图5 集群拓扑图

4.2 实验结果分析

实验数据集为文件数分别为100000、200000、300000、400000的自建新闻数据集:News_1、News_2、News_3、News_4,每个新闻文件的平均大小为200KB。

4.2.1 MPHF模型验证

针对数据集News_4我们使用不同MPHF构建元数据索引,并比较了不同算法构建索引时的构建时间、查询时间以及空间占用情况,如表4所示。其中,CHD算法中λ设置为4,BBhash算法中γ设置为2。

表4 MPHF比较

算法模型	构建时间(s)	查询时间(ns/key)	空间占用(bits/key)
CHD	1261	431	2.17
GOV	**621**	**183**	**2.24**
BBHash	672	213	3.72

由表看出,作为保序最小完美哈希的代表算法GOV,在构建时间、查询时间以及空间占用方面都明显优于其它两种算法。

4.2.2 内存占用

实验主要是测试数据文件合并前后,AlluxioMaster节点的内存占用情况。实验结果如表5所示,文件合并后,文件元数据对AlluxioMaster节点的内存占用呈现指数级减少。这是因为合并以后,小文件元数据主要存放在索引文件中,索引文件则被存储在AlluxioMaster节点的外部缓存空间中,必要时也会缓存在AlluxioClient节点。而AlluxioMaster的文件树只存储合并后的合并文件的元数据。所以,存储元数据对AlluxioMaster的内存压力将会大大减少。

表5 AlluxioMaster 内存占用表

数据集	合并前内存占用(KB)	合并后内存占用(KB)
News_1	97700	2.0
News_2	195000	3.8
News_3	293100	5.7
News_4	390500	7.6

4.2.3 读性能

本实验主要是测试使用不同MPHF对合并元数据条目进行索引，对小文件读性能的影响，实验数据集为News_4。随机读取100个文件，比较文件读取时间。实验结果如图6所示。不同MPHF对文件读性能都有一定的提升，其中GOV算法的性能最优，是原生系统的5.32倍。

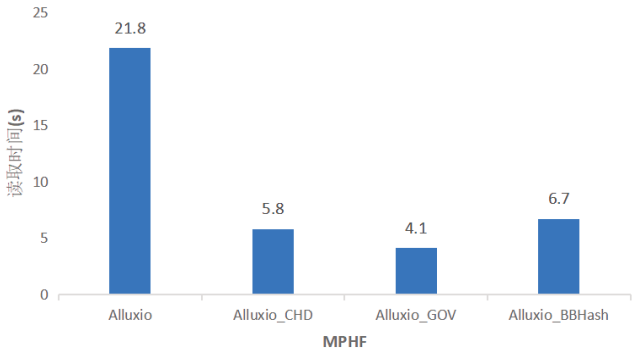

图6 文件读性能

4.3 性能分析

从合并文件中访问一个文件所花费的总时间 T_{Access} 由公式(1)定义：

$$T_{Access} = T_M + T_C \quad (1)$$

其中，T_M 是合并文件从其索引文件中检索文件元数据所需的时间，T_C 是从合并文件中恢复文件内容所需的时间。

假设，T_{M-CHD} 代表从CHD算法构建的索引中检索出文件元数据的时间，T_{M-GOV} 代表从GOV算法构建的索引文件中检索出文件元数据的时间，$T_{M-BBHash}$ 代表从BBHash算法构建的索引文件中检索出文件元数据的时间。由上述实验结果可知，

$$T_{M-GOV} < T_{M-CHD} < T_{M-BBHash} \quad (2)$$

由于 T_C 受各种因素影响，包括seek时间以及是否使用压缩算法改变合并文件大小。但是对于三种算法来说，T_C 是相同的。因此 T_{Access} 只受 T_M 的影响。所以根据公式(1)、(2)得出：

$$T_{Access-GOV} < T_{Access-CHD} < T_{Access-BBHash} \quad (3)$$

5 结论

本文针对存储与计算分离模式下新闻文本的存储与读访问瓶颈，提出了基于Alluxio的新闻文本存储优化方法，解决了新闻文本的存储与访问隔离问题，并对新闻文本的存储以及读访问进行了优化。通过实验验证，对新闻文本进行基于目录聚合的合并能够指数级提高缓存系统元数据管理节点的内存利用率。同时使用基于最小完美哈希算法的元数据索引构建方法将新闻文本的读访问性能提升了数倍，从而达到新闻文本快速检索的目的。本文目前以新闻文本的存储作为主要研究对象，后续可以直接应用到新闻图片等小文件类型数据的存储场景下，具有可扩展性。

参考文献(References)：

[1] WekaIO, Inc. What is data orchestration? a guide to handling modern data [EB/OL]. (2021-08-09) [2023-10-12] https://www.weka.io/blog/data-orchestration/.

[2] Belazzougui D, Boldi P, Pagh R, et al. Theory and practice of monotone minimal perfect hashing [J]. ACM Journal of Experimental Algorithmics, 2008, 16: 3.1-3.26.

[3] 顾荣，刘嘉承，毛宝龙. 分布式统一大数据虚拟文件系统：Alluxio原理、技术与实践[M]. 北京：机械工业出版社，2023.

[4] Gu R, Dong Q, Li H, et al. DFS-PERF: a scalable and unified benchmarking framework for distributed file systems [R/OL]. (2016-07-27) [2023-10-12] https://www2. eecs. berkeley.edu/Pubs/TechRpts/2016/EECS-2016-133.html.

[5] Ghemawat S, Gobioff H, Leung S T. The Google file system [J]. ACM SIGOPS Operating Systems Review, 2003, 37(5): 29-43.

[6] Schwan P. Lustre: building a file system for 1, 000-node clusters [C]//Proceedings of the Linux Symposium, 2003: 380-386.

[7] Borthakur D. HDFS architecture guide [EB/OL]. (2022-05-18) [2023-10-12] https://hadoop. apache. org/docs/r1.2.1/hdfs_design.html.

[8] Alibaba Inc. Alibaba/tfs [DB/OL]. (2014-01-04) [2023-10-12] https://github.com/alibaba/tfs.

[9] Beaver D, Kumar S, Li H C, et al. Finding a needle in haystack: Facebook's photo storage [C]//9th USENIX Symposium on Operating Systems Design and Implementation (OSDI'10), 2010: 47-60.

[10] Fu S, He L, Huang C, et al. Performance optimization for managing massive numbers of small files in distributed file systems [J]. IEEE Transactions on Parallel and Distributed Systems, 2014, 26(12): 3433-3448.

[11] Apache Software Foundation. Hadoop Archives Guide[Z/OL]. [2023-10-12] https://hadoop.apache.org/docs/r2.7.5/hadoop-archives/HadoopArchives.

[12] White T. Hadoop: The Definitive Guide[M]. Sebastopol, CA: O'Reilly Media Inc, 2012.

[13] Meng B, Guo W, Fan G, et al. A novel approach for efficient accessing of small files in HDFS: TLB-MapFile[C]// 17th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD), 2016: 681-686.

[14] Tao W, Zhai Y, Tchaye-Kondi J. LHF: A new archive based approach to accelerate massive small files access performance in HDFS[C]//2019 IEEE Fifth International Conference on Big Data Computing Service and Applications (BigDataService), 2019:40-48.

[15] Zhai Y, Tchaye-Kondi J, Lin K J, et al. Hadoop perfect file: a fast and memory-efficient metadata access archive file to face small files problem in HDFS[J]. IEEE Journal of Parallel and Distributed Computing, 2021, 156: 119-130.

[16] Belazzougui D, Botelho F C, Dietzfelbinger M. Hash, displace, and compress[C]//European Symposium on Algorithms, 2009: 682-693.

[17] Genuzio M, Ottaviano G, Vigna S. Fast scalable construction of ([compressed] static| minimal perfect hash) functions[J].Information and Computation, 2020, 273: 104517.

[18] Limasset A, Rizk G, Chikhi R, et al. Fast and scalable minimal perfect hashing for massive key sets[DB/OL]. arXiv:1702.03154, 2017.

[19] Tarkoma S, Rothenberg C E, Lagerspetz E. Theory and practice of bloom filters for distributed systems[J]. IEEE Communications Surveys & Tutorials, 2011, 14(1): 131-155.

编辑:王谦