软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software, 2023,34(3):1451−1469 [doi: 10.13328/j.cnki.jos.006710]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

多模态医疗数据中海量小文件存储优化方法*

曾 梦[1,4], 邹北骥[1,4], 张文生[2], 杨雪冰[2], 朱承璋[3,4]

[1](中南大学 计算机学院, 湖南 长沙 410083)

[2](中国科学院 自动化研究所, 北京 100190)

[3](中南大学 文学与新闻传播学院, 湖南 长沙 410083)

[4](湖南省机器视觉与智慧医疗工程技术研究中心(中南大学), 湖南 长沙 410083)

通信作者: 朱承璋, E-mail: chzhzhu@csu.edu.cn

摘 要: Hadoop 分布式文件系统(HDFS)通常用于大文件的存储和管理, 当进行海量小文件的存储和计算时, 会消耗大量的 NameNode 内存和访问时间, 成为制约 HDFS 性能的一个重要因素. 针对多模态医疗数据中海量小文件问题, 提出一种基于双层哈希编码和 HBase 的海量小文件存储优化方法. 在小文件合并时, 使用可扩展哈希函数构建索引文件存储桶, 使索引文件可以根据需要进行动态扩展, 实现文件追加功能. 在每个存储桶中, 使用 MWHC 哈希函数存储每个文件索引信息在索引文件中的位置, 当访问文件时, 无须读取所有文件的索引信息, 只需读取相应存储桶中的索引信息即可, 从而能够在 O(1)的时间复杂度内读取文件, 提高文件查找效率. 为了满足多模态医疗数据的存储需求, 使用 HBase 存储文件索引信息, 并设置标识列用于标识不同模态的医疗数据, 便于对不同模态数据的存储管理, 并提高文件的读取速度. 为了进一步优化存储性能, 建立了基于 LRU 的元数据预取机制, 并采用 LZ4 压缩算法对合并文件进行压缩存储. 通过对比文件存取性能、NameNode 内存使用率, 实验结果表明, 所提出的算法与原始 HDFS、HAR、MapFile、TypeStorage 以及 HPF 小文件合并方法相比, 文件读取时间更短, 能够提高 HDFS 在处理多模态医疗数据中海量小文件时的整体性能.

关键词: 多模态医疗数据; HDFS; HBase; 小文件; 存储性能优化

中图法分类号: TP311

中文引用格式: 曾梦, 邹北骥, 张文生, 杨雪冰, 朱承璋. 多模态医疗数据中海量小文件存储优化方法. 软件学报, 2023, 34(3): 1451–1469. http://www.jos.org.cn/1000-9825/6710.htm

英文引用格式: Zeng M, Zou BJ, Zhang WS, Yang XB, Zhu CZ. Optimization Method for Storing Massive Small Files in Multi-modal Medical Data. Ruan Jian Xue Bao/Journal of Software, 2023, 34(3): 1451–1469 (in Chinese). http://www.jos.org.cn/1000-9825/6710.htm

Optimization Method for Storing Massive Small Files in Multi-modal Medical Data

ZENG Meng[1,4], ZOU Bei-Ji[1,4], ZHANG Wen-Sheng[2], YANG Xue-Bing[2], ZHU Cheng-Zhang[3,4]

[1](School of Computer Science and Engineering, Central South University, Changsha 410083, China)

[2](Institute of Automation, Chinese Academy of Sciences, Beijing 100190, China)

[3](School of Literature and Journalism, Central South University, Changsha 410083, China)

[4](Hunan Engineering Research Center of Machine Vision and Intelligent Medicine (Central South University), Changsha 410083, China)

Abstract: Hadoop distributed file system (HDFS) is used for the storage and management of large files, while storing and computing a large number of small files consume a lot of NameNode memory usage and access time. Therefore, the small file problem becomes an important factor that restricts HDFS performance. Aiming at the problem of massive small files in multi-modal medical data, a small file

* 基金项目: 科技创新 2030——“新一代人工智能”重大项目(2018AAA0102100); 湖南省科技计划项目 (2017WK2074); 湖南省高新技术产业科技创新引领计划 (2020GK2021)

收稿时间: 2021-06-17; 修改时间: 2021-11-25; 采用时间: 2022-02-23

storage method based on two-layer hash coding and HBase is proposed to optimize the storage of massive small files on HDFS. When merging small files, an expandable hash function is utilized to build an index file bucket to expand the index file dynamically as needed and realize the file append function. To read the file in O(1) time complexity and improve the efficiency of file search, the MWHC hash function is used to store the position of the index information of each file in the index file. There is no need to read the index information of all files, only need to read the index information of the corresponding bucket. To meet the storage needs of multi-modal medical data, HBase is used to store the index information and set the identification column to identify different modal medical data, which is convenient for storage and management of different modal data and improves file reading speed. To further optimize storage performance, the LRU-based metadata prefetching mechanism is established, and the LZ4 compression algorithm is utilized to compress the merged files. The experiment compares file access performance and NameNode memory usage. The results show that compared with the original HDFS, HAR, MapFile, TypeStorage, and HPF small file storage methods, the proposed algorithm has a shorter file access time, which can improve the overall performance of HDFS when processing massive small files in multi-modal medical data.

Key words: multi-modal medical data; HDFS; HBase; small files; storage performance optimization

随着大数据应用技术的飞速发展，单机存储系统已无法满足实际存储和计算的需求，分布式存储技术成为解决大数据问题的有效方式. 为了满足大数据存储的需求，Google 文件系统[1]、Hadoop 分布式文件系统(HDFS)[2]、Lustre 文件系统[3]等分布式文件系统被设计并应用于实际生产中. 其中，HDFS 作为 Hadoop 分布式系统的底层文件存储系统，随着 Hadoop[4]产生并不断发展，由于其具有开源、可移植、高可用、高容错以及可大规模水平扩展等特性[5]而得到广泛应用.

HDFS 作为一种通用的分布式文件系统，用于大规模数据存储并且能提供高密度数据访问，HDFS 集群可以扩展到数千甚至数万个节点. HDFS 的体系结构如图 1 所示，使用 master-slave 架构存储数据，主要由 4 个部分组成，即 Client、NameNode、DataNode 以及 Secondary NameNode.

- Client: 作为客户端与 NameNode 获取文件元数据信息，并与 DataNode 交互进行文件读写. 当文件上传到 HDFS 时，它可以将文件分割为多个块进行存储. 此外，它提供一些命令用于对 HDFS 进行管理或访问.
- NameNode: 作为 master 的角色，它管理 HDFS 的命名空间和数据块映射信息，并处理来自客户端的读写请求. 由于元数据的大小随着集群规模的扩大以及存储文件数量的增加而增大，有时单个节点的内存无法满足元数据存储需求，因此允许存在多个独立的 NameNode 节点，使每个节点都负责一部分数据.
- DataNode: 作为 slave 的角色，它存储实际的数据块并执行数据块的读写操作.
- Secondary NameNode: 为了防止 NameNode 单点故障，辅助 NameNode 而设计的 NameNode 的备份. 它分担了 NameNode 的工作，并在紧急情况下协助恢复文件信息.

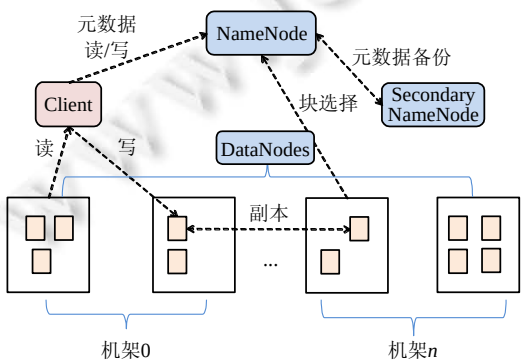


图 1 HDFS 架构

随着互联网的广泛应用，数据量的指数级增长使数据存储和计算的压力越来越大，HDFS 作为底层的存储和管理系统，能够有效地存储较大的文件，而社交媒体、医疗保健、电子商务等各个领域产生的大量小文件

则成为了 HDFS 的性能瓶颈. 小文件是指文件大小明显小于 HDFS 块大小(默认 128 MB)的文件. HDFS 存储小文件主要有以下问题.

- 第一, 所有文件的元数据信息必须保存在 NameNode 中, 当文件数量过多时, 会造成 NameNode 内存瓶颈. 根据实验验证: 存储单个文件的元数据信息需要大约 150 字节的内存, 假设使用默认的三副本策略存储每个文件, 则存储单个文件元数据信息将消耗 368 字节[6], 如果同一个文件夹有 10 亿个小文件, 则最少需要 300 GB 的内存空间来存储文件元数据信息.
- 第二, 当读取小文件较多时, 客户端需要不断地向 NameNode 发送请求并获得元数据信息, 然后在 DataNdoe 中读取小文件, 在节点间频繁通信将会消耗大量的时间.
- 第三, 小文件过多会导致大量的随机 I/O, 当运行 MapReduce 时, 需要大量的节点来处理这些小文件, 而并发的 mapper 数量有限. 当集群较小时, mapper 任务需要排队等候处理, 此时需要花费大量的时间, 这是影响 MapReduce 性能的重要因素之一. 因此, 较少数量的文件和更大的存储块, 将会减少 I/O 对性能的影响.

随着医疗信息化的飞速发展, 医疗设备也在不断地更新换代, 产生了海量且类型多样的医疗数据. 根据目前医疗数据展示的具体信息及表现形式, 可以将其大致分为 3 类.

- 第 1 类, 临床文本数据. 主要为结构化的检验数据, 如血常规. 同时也包含非结构化的文本数据, 如医生的诊断报告.
- 第 2 类, 影像、波形数据. 主要包括超声图像、CT 图像、核磁共振图像等影像数据, 以及心电图、脑电图等信号数据.
- 第 3 类, 生物组学数据. 这一类数据可以按照不同分子层面划分为基因组学、转录组学、蛋白组学等.

获取患者相关数据的一类方式即为一种数据模态, 不同模态则提供了患者不同角度的诊断信息, 从而形成患者的多模态医疗数据. 近年来, 有关多模态医疗数据的研究方法得到了国内外学者的广泛关注. 然而, 对于如何有效地存储和管理多模态医疗数据, 目前的研究很少.

为了解决多模态医疗数据中的海量小文件问题, 本文提出一种基于双层哈希编码和 HBase 的海量小文件存储优化方法, 主要贡献如下.

(1) 对多模态医疗数据中海量小文件进行合并, 使用可扩展哈希和 MWHC 哈希函数构建索引文件, 该索引文件为单层索引, 能够在 O(1)的时间复杂度内进行文件查询, 提高文件读取效率, 同时支持文件追加功能.

(2) 结合多模态医疗数据的特性, 利用 HBase 存储文件元数据信息, 并设置标识列用于标识不同模态的医疗数据, 便于对不同模态数据的存储管理, 且使文件索引读取不依赖于缓存容量大小, 进一步提高文件读取效率.

(3) 为了进一步优化存储性能, 建立基于 LRU 的预取机制, 将最近访问的文件索引信息存储在客户端内存中, 根据内存大小在客户端设定缓存的索引信息容量. 当 HBase 中无法获取所查询的索引信息时, 可在缓存中查询, 减少和 NameNode 交互. 同时, 采用 LZ4 压缩算法对合并文件进行压缩存储, 在充分考虑文件存取效率的同时节省存储空间.

本文第 1 节介绍目前海量小文件合并方法的国内外研究现状. 第 2 节介绍本文提出的多模态医疗数据中海量小文件存储优化方法. 第 3 节进行实验对比和结果分析. 第 4 节进行总结及未来工作的展望.

1 相关工作

如今, 针对 HDFS 的海量小文件存储优化问题的研究已有很多, 可以将这些方法分为 3 类[7].

第 1 类是将小文件合并成大文件, 以减轻 NameNode 存储元数据的负担, HDFS 默认的 HAR 文件、SequenceFile 以及 MapFile[8]就是典型的将小文件合并成大文件的方法, 文件结构如图 2 所示.

- Hadoop Archive (HAR)文件: 通过使用 MapReduce, 将大量的小文件打包到 HAR 文件中, 如图 2(a)所

示, har 文件使用_index 和_masterindex 构建两层索引文件存储元数据信息, 数据文件则以 part-*形式存储. HAR 存在很多问题: 首先, HAR 文件的源文件和目录在文件合并后不会自动删除, 这会浪费大量的磁盘空间; 其次, 一旦创建了 HAR 文件, 无法进行文件追加和部分删除; 第三, 当 MapReduce 访问 HAR 文件时, InputFormat 无法将多个文件划分为一个 InputSplit, 因此仍然需要处理多个小文件; 最后, 由于 HAR 构建的索引文件为双层索引, 可能导致 HAR 文件读取小文件的时间比原始 HDFS 读取时间更长, 并且 HAR 文件不支持文件压缩存储.

- SequenceFile 文件: 图 2(b)展示了 SequenceFile 的结构, SequenceFile 由一系列二进制 Key/Value 组成, SequenceFile 可以使用记录压缩和块压缩两种方式对文件进行压缩, 并且支持将 splittable 作为 MapReduce 的输入切片. 但是, 当读取文件时, 由于没有建立小文件和大文件之间的映射关系, 因此必须查找所有的 SequenceFile. 此外, SequenceFile 也不具备更新和删除功能, 且不考虑文件之间的关联性.
- MapFile 文件: 如图 2(c)所示, 它是基于 SequenceFile 的改进方法, 在 SequenceFile 的基础上增加了索引文件, 用于记录键值和偏移量. 因此, MapFile 的检索效率更高. 但 MapFilet 同样不具备追加和删除功能.

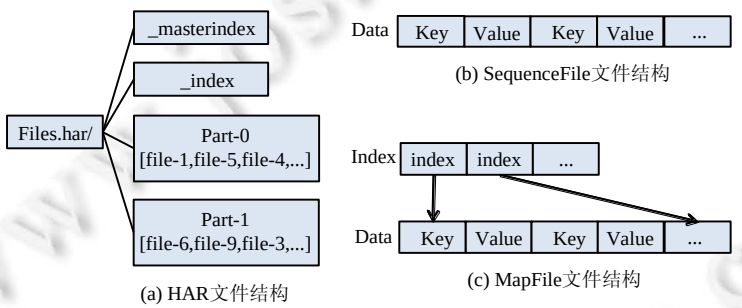


图 2 HDFS 默认小文件合并方法文件布局

为了进一步提升小文件存取效率, Zhai 等人[7]引入了 Hadoop Perfect File (HPF), 使用可扩展哈希函数和完美哈希函数构建单层索引, 对 Hadoop 存档文件进行优化, 从而提高文件的读取效率; Vorapongkitipun 等人[9]提出一种 New HAR (NHAR)的小文件合并方法, 该方法通过使用哈希函数将原有的 HAR 两层索引改为一层索引, 并且实现了文件追加功能, 但 NHAR 方法在 part files 大小达到 HDFS 默认块大小后, 需要重新对整个合并文件进行重组, 这会耗费大量的时间, 且在进行重组前需要更多的存储空间; Renner 等人[10]提出一种可扩展可追加的 Hadoop 存档文件; 与基于 Hadoop 存档文件的方式不同, Optimized Mapfile based Storage of Small files (OMSS)[11]是一种典型的基于 MapFile 的小文件合并方法, 该方法使用 worst fit 策略将小文件合并成大文件来减少数据中的内部碎片, 从而减少内存开销; TLB-MapFile[12]方法使用 fast table structure (TLB)建立数据块与小文件之间的映射关系, 从而提高小文件的读取效率. 这些方法虽然提升了海量小文件的存取效率, 但仍然存在一些与 HAR 和 MapFile 文件相同的问题, 如未考虑文件之间的关联性, 以及不同文件类型的分类存储管理等. 为了在特定应用场景中解决小文件合并问题, Dong 等人[13]提出了基于 BlueSky 系统的小文件处理方法, 将具有相关性的文件合并到同一个大文件中, 并引入两层的预取机制来优化算法; Liu 等人[14]提出一种使用 WebGIS 数据特征进行小文件存储的方法. 然而, 这些方法通常需要在特定的场景中使用, 不具有普适性. 基于数据库将小文件合并成大文件也是一种很好的改进方法, 郑通等人[15]将 HBase 和 HDFS 相合, 使用 HBase 存储小文件到合并文件的映射信息, 并在客户端设置预取机制; Liu 等人[16]提出将 RDBMS 和 Hadoop 云存储相结合来提升读写性能.

第 2 类方法使用专用的分布式文件系统来优化小文件处理框架. 典型的方法如: Twitter 的 Cassandra[17]; 淘宝设计了 TFS[18]文件系统, 该文件系统运用 IFLATLFS[19,20]解决小文件问题, 提供了高可靠性, 高可用性和

高性能; 同样地, Facebook[21]使用 Haystack 解决大量图片文件的问题. 然而, 专有的文件系统在系统设计上比较复杂并且花费的代价比较高.

第 3 类方法主要是改进处理框架. Extend Hadoop Distributed File System (EHDFS)[22]设计了索引和预取机制来提升系统的性能. CombineFile-InputFormat 方法在执行 map 任务之前, 将多个小文件划分到一个大的切片中, 以此来提升处理小文件的性能, 但是无法缓解 NameNode 的内存开销[23,24]. Choi 等人[25]通过结合 CombineFileInputFormat 和 Java 虚拟机(JVM)的重用功能来提高 CombineFileInputFormat 的性能. Zhou 等人[26]设计了 SFMapReduce 框架来解决小文件问题, 该框架使用 SFLayout 和 CMR 相结合.

此外, 一些学者针对小文件归类存储方面的进行研究, You 等人[27]研究了海量教育资源中的小文件问题, 根据文件之间的关联性进行文件归类存储; Zhao 等人[28]将海量 MP3 文件中相关性较强的文件进行归类, 并合并为大文件存储. 然而, 上述方法由于需要处理文件之间的相关性, 从而降低了文件的读写速率. Qin 等人[29]在大数据平台下, 根据不同文件类型进行小文件合并. 然而该方法依赖于缓存大小, 且只针对重复数据的读取时效率较高, 因此对于大规模数据存储仍具有局限性.

上述方法在一定程度上解决了 HDFS 存储海量小文件中的问题, 但是依然存在一些不足. 很多小文件合并方法并没有考虑大小文件以及不同类别文件的归类存储. 对于设计特定的分布式文件系统通常需要花费很大的代价, 并且只能满足一些特定的需求. 针对当前方法存在的一些问题, 本文提出一种用于多模态医疗数据中海量小文件的存储优化方法, 该方法基于双层哈希函数和 HBase, 结构简单, 无须修改 HDFS 底层设计, 文件读取速度快, 能够在 O(1)时间复杂度内进行文件读取, 提升文件读取效率. 同时, 使用 HBase 存储文件元数据信息, 结合多模态医疗数据特性设置文件标识列, 便于对多模态医疗数据的存储管理, 且使文件读取不依赖于缓存大小, 在文件分类存储的同时提升文件读取性能. 据目前的研究来看, 基于多类型存储的方法虽然兼顾文件的多类型, 但牺牲了文件的读取效率. 本文的方法能够在对文件进行归类的同时提升文件的读取速度. 值得注意的是, 本文针对医疗数据多模态的特点, 使用 HBase 的方法对文件进行分类存储, 该方法对于具有多模态特性的其他数据也具有一定的适用性. 此外, 为了避免在 HBase 中因无法查询索引信息而增加与 NameNode 的交互, 我们引入了索引预取机制, 充分优化文件的读取效率, 并采用 LZ4 压缩算法对合并后的文件进行压缩存储, 在充分考虑文件存取效率的同时节省文件存储空间.

2 多模态医疗数据中海量小文件存储优化方法

本文的海量小文件存储优化方法主要分为两部分: 文件上传和文件读取. 为了方便表达, 将本文所使用的方法命名为 Two-layer Hash File with HBase (THF-HBase).

文件处理框架如图 3 所示.

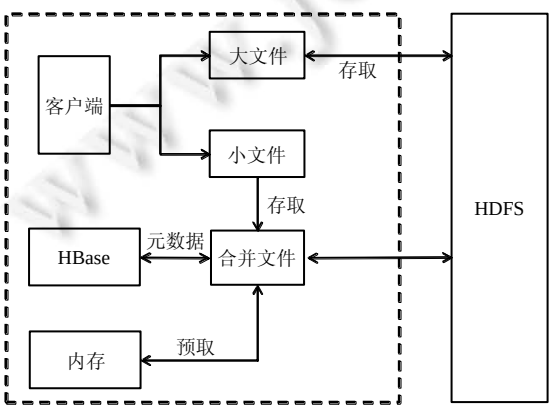


图 3 文件处理框架

针对文件上传, 首先判断文件大小.

- 如果是大文件, 则直接将文件上传至 HDFS.
- 如果为小文件, 则将小文件合并为大文件, 并采用可扩展哈希和 MWHC 哈希构建索引文件, 建立小文件到合并文件之间的映射关系. 将每个小文件的索引信息存入 HBase 中, 并在 HBase 中对设置标识列对每个小文件进行标识, 用于获取不同模态的文件信息. 为了节省文件存储空间, 使用 LZ4 压缩算法对文件进行压缩存储, 同时将合并文件上传至 HDFS 中.

针对文件读取, 首先判断文件大小.

- 如果为大文件, 则查看缓存中是否有读取文件的索引信息: 如果缓存中有索引信息, 则直接从缓存中读取文件索引信息, 再根据索引信息读取相应的文件; 如果没有索引信息, 则访问 NameNode 查找索引信息, 再根据索引信息读取相应的文件, 并将文件的索引信息存入缓存中.
- 如果为小文件, 则根据文件名在 HBase 中查找相应的索引信息, 再根据索引信息读取文件; 如果 HBase 中没有对应的小文件, 则通过缓存进行查找, 后续操作和大文件类似.

为了适用于多模态医疗数据的存储管理, 在 HBase 中设置标识列, 标识不同模态的数据信息. 当需要进行批量获取某种模态的数据文件时, 可以输入对应的标识信息进行批量读取. 这里需要注意的是, 由于在进行文件存储时, 每个小文件的索引信息都会存入 HBase 中, 因此一般情况下, 存储的小文件索引信息都会在 HBase 中获得. 对于文件大小的阈值判定, 目前没有一个准确的界定, 大多数学者认为, 小文件是指文件大小明显小于 HDFS 块大小(默认 128 MB)的文件. 在文献[19]中验证, 文件大阈值设置为 4.35 MB 时效果最佳. 对于特定的多模态医疗数据, 其中的医疗影像类数据的文件大小相对较大. 因此, 为了满足实际需求, 本文设置文件大小阈值为 HDFS 默认块大小, 即 128 MB. THF-HBase 文件布局如图 4 所示, _index-*i* 用于存储索引信息, _names 用于存储文件名称, part-*i* 用于存储实际的文件数据. 为了方便表述, 后续文件合并、文件读取、追加以及文件缓存和压缩操作都针对多模态医疗数据中海量小文件展开讨论.

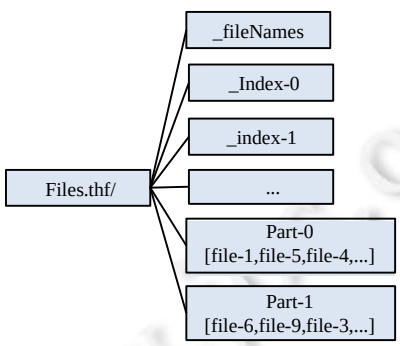


图 4 THF-HBase 文件结构

2.1 文件合并

2.1.1 索引文件构建

本文使用可扩展哈希函数和 MWHC 哈希函数构建单层索引文件. 可扩展哈希表(EHT)[30]是一个动态哈希方法, 如图 5 所示, 使用数组来存储桶指针目录, 哈希函数为键的最后几位. 假设全局位深度为 D, 则当位数为 i 时, 桶的容量为 i^D. 当发生溢出时, 溢出桶进行分裂, 其容量变为 i^{L+1}, 而其他桶的容量不变, 此时数据目录的深度变为 D+1. 如果删除了某个存储桶, 则使用该存储桶的目录将更改为引用相邻的存储桶. 使用可扩展哈希函数构建索引文件, 可以对索引文件进行动态扩展, 无须为后续索引条目增加而预留额外的存储空间, 有利于文件的追加和存储空间的使用.

MWHC 哈希函数[31]属于完美哈希函数, 是一种静态哈希函数. 完美哈希函数将 N 个键值映射到 S 个整数集合中(S>N), 且保证不会发生冲突. 当 N 等于 S 时, 称为最小完美哈希函数(minimal perfect hash function,

MPHF)[32]. MWHC 哈希函数使用 Majewski-Wormald-Havas-Czech 技术, 该方法通过实现 3-hypergraph 边缘排序过程进行哈希编码. 文献[33]中证实: MWHC 方法在实际应用中虽然空间使用量较大, 但具有速度快的优势. 由于 MWHC 哈希函数为静态哈希函数, 其关键字的集合是确定的静态集合, 因此可以有效缩减索引所需空间.

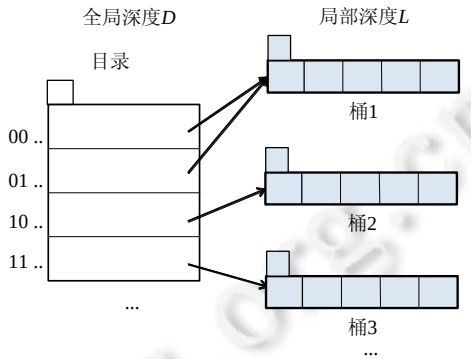


图 5 可扩展动态哈希结构

在本文方法中, 首先对每个桶中的文件进行排序, 确定每个桶中文件的索引位置, 因此文件的索引信息为固定值; 然后使用 MWHC 哈希函数存储索引位置信息. 通常对文件的操作为“存一次, 取多次”的方式, 因此文件的读取速度是需要关注的重点, 使用 MWHC 哈希函数能够很好地保证文件的快速读取. Zhai 等人[7]提出的 HPF 使用保序的完美哈希函数(order preserving perfect Hash function)对文件的索引信息进行排序后存储, 然而其存取效率并没有 MWHC 高, 将在第 3 节实验中给出相应的实验验证结果.

2.1.2 HBase 索引表构建

HBase[34]是一种开源的、面向列的分布式数据库, 存储在 HBase 中的表可以存储上亿行、上百万列. 由于 HBase 具备高可靠性、高性能、可灵活扩展且支持实时数据读写等特性, 因此得到广泛的应用. HBase 数据表主要由主键和列簇构成, 列簇中又包含一个到多个列.

在本文中, 使用 HBase 存储数据文件的元数据信息, 同时设置标识列用来标识不同模态的数据. 如表 1 所示, 创建一个列簇, 将文件元数据信息中的所有属性作为列属性放入同一列簇中, 同时在列簇中增加模态标识列. 这里使用文件后缀名作为标识信息, 如 DICOM 文件使用 dcm 作为标识, Text 文件使用 txt 作为标识. 当对某一模态的数据进行批量读取时, 只需输入对应类型即可实现批量读取. HBase 存储元数据信息如表 2 所示, 一共包含 4 个部分. 为了保证元数据信息的每部分都具有固定大小, 将文件名用哈希值代替, 每个文件的元数据信息共 24 bytes. 由于 HBase 可以实现高性能并发读写操作, 因此能够高效地查询所需读取的文件索引信息, 提高文件读取效率. 此外, HBase 中可直接存储数据文件位置、偏移量等信息, 在进行文件读取时, 直接根据文件名获取文件位置及偏移量等信息, 可以直接读取相应的小文件. 如果不使用 HBase 读取文件索引信息, 则需要查找对应的索引文件, 再根据索引文件中的文件位置计算偏移量, 进而读取对应的小文件, 则会消耗更多的读取时间. 具体读取流程和性能对比分别在第 2.2 节和实验部分进行详细论述.

表 1 HBase 表结构

Row key	Time stamp	列簇				
		模态标识	文件名(Hash 值)	数据文件位置	偏移量	文件大小
小文件 1	T_1	F_1	H_1	P_1	O_1	S_1
小文件 2	T_2	F_2	H_2	P_2	O_2	S_2
…	…	…	…	…	…	…
小文件 n	T_n	F_n	H_n	P_n	O_n	S_n

表 2 元数据信息汇总

名称	类型	大小(byte)
文件名(Hash 值)	long	8
数据文件位置	int	4
偏移量	long	8
文件大小	int	4

2.1.3 文件合并过程

合并过程如图 6 所示: 首先, 将小文件合并到 n 个大文件中, 使用 part-i 进行存储; 然后, 使用 EHT 构建 n 个存储桶, 每个桶中存储 k 个小文件的索引信息; 第三, 将每个桶中的文件索引信息按照文件名的哈希值字典序进行排序, 再使用 MWHC 哈希函数存储每个文件索引信息的位置信息, MWHC 哈希函数写在每个索引文件的开头; 第四, 将文件的索引信息存入 HBase 中, 并根据不同模态的数据设置相应的标识; 最后, 当所有的文件写入完成后, 删除_temporaryIndex 文件.

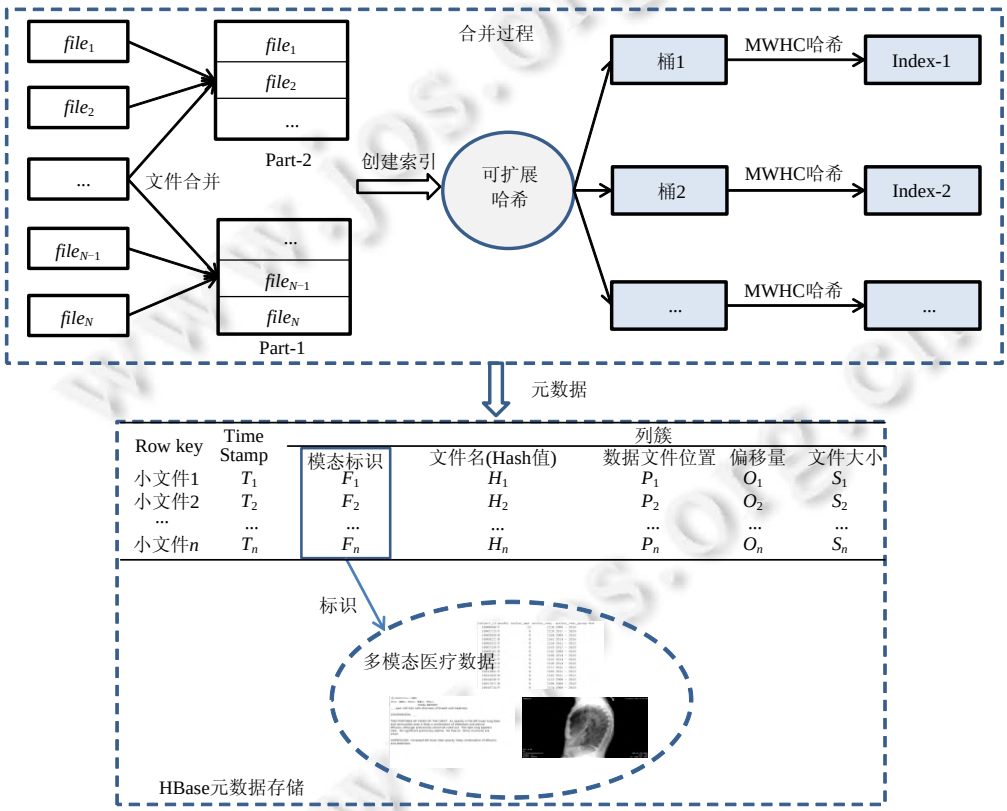


图 6 文件合并

在文件合并时, 会创建一个临时索引文件_temporaryIndex, 用于在合并文件过程中发生故障进行文件恢复使用. 当文件追加到 part-i 文件时, 将文件的元数据存储到临时索引文件中作为备份, 文件存储完成后, 再将_temporaryIndex 临时文件删除. 在文件合并时, 文件内容会加载到客户端内存中, 这时, 启用 LZ4 压缩方法[35]进行文件压缩后, 再将文件存入 part-i 中. 当然, 根据需求, 也可以选择不进行文件压缩存储. 当 part-i 文件达到设定的最大值时, 将创建一个新的 part 文件, 文件存储完成后, 创建每个文件的元数据信息并插入到索引文件中.

2.2 文件读取和追加

对于文件的读取操作分为两种情况, 读取流程如图 7 所示: 第一, 根据文件名在 HBase 中检索对应的元数据信息, 如果查询到结果, 则直接根据文件所在的位置信息找到对应的 part 文件并读取相应的文件内容; 第二, 如果 HBase 未查询到索引信息, 则在缓存中检索, 如果查询到结果, 则根据文件名的哈希值, 使用 EHT 计算对应存储桶的位置, 即对应的索引文件. 然后, 使用 MWHC 哈希函数计算文件索引信息所在的位置, 根据公式(1)计算出文件确切的偏移量 Y, 由于每个文件的元数据占 24 bytes, 因此要获取小文件的元数据, 只需从 Y 读到 Y+24; 根据文件的元数据信息, 查询对应的 part 文件并读取相应的文件内容; 如果缓存中未查询到索引信息, 则与 NameNode 交互, 得到索引信息, 后续操作与缓存中查询相同:

$$Y=x+h\times 24 \tag{1}$$

其中, x 表示 MWHC 在索引文件中的大小, h 表示文件名的哈希值.

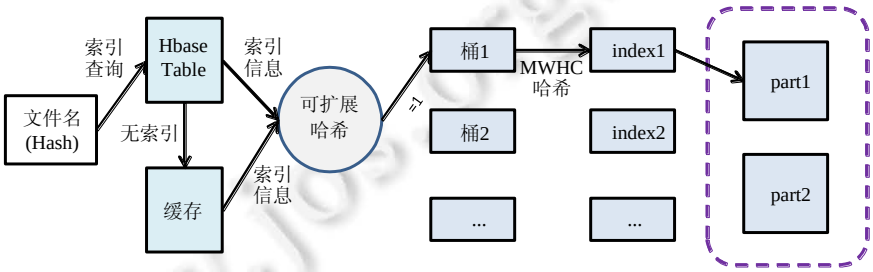
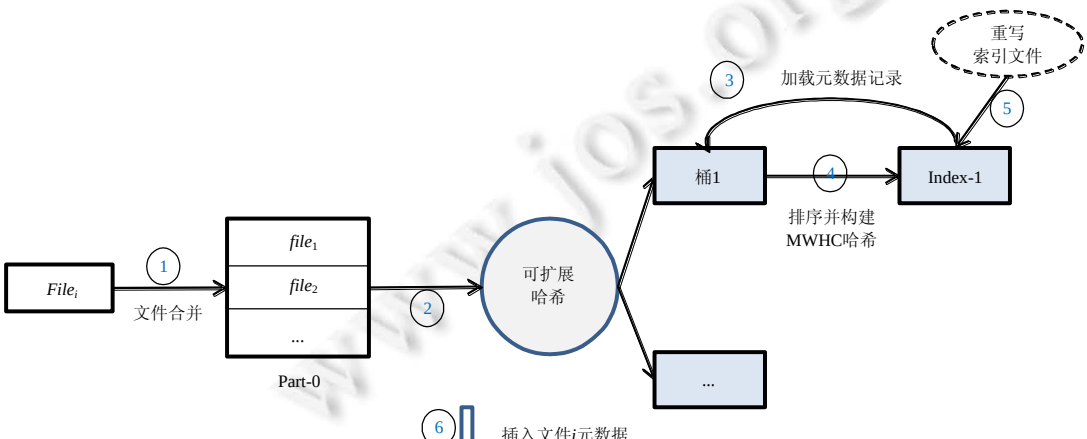


图 7 文件读取

对于文件的追加, 与文件合并过程类似, 如图 8 所示: 首先, 重新创建文件合并线程, 并追加小文件至 part-i 中, 追加文件名至_names 文件中, 在此过程中, 重新创建_temporaryIndex 文件; 然后, 使用可扩展哈希函数将文件索引放入对应的桶中; 第三, 追加文件的索引信息, 在其对应的存储桶中, 重新对索引信息按照文件名哈希值的字典序进行排序, 并使用 MWHC 哈希函数存储索引位置信息; 第四, 将文件索引信息追加至 HBase 中; 最后, 当所有的文件追加完成后, 删除_temporaryIndex 文件.



Row key	Time stamp	列簇				
		模态标识	文件名(Hash值)	数据文件位置	偏移量	文件大小
小文件1	T_1	F_1	H_1	P_1	O_1	S_1
小文件2	T_2	F_2	H_2	P_2	O_2	S_2
...	...	...	...	...	...	...
小文件n	T_n	F_n	H_n	P_n	O_n	S_n
小文件i	T_i	F_i	H_i	P_i	O_i	S_i

图 8 文件追加

由于 THF-HBase 支持文件追加，因此在文件数量增加的过程中，会不断发生存储桶中索引溢出并动态分裂的过程，该过程实际是可扩展哈希的操作. 如图 9 所示：首先，在插入文件时，先创建文件索引；然后，使用可扩展哈希函数将元数据放入相应的存储桶 i 中，图 9 所示例子为放入桶 2 中；第三，若存储桶已满，如桶 2，则创建新的索引文件和新的存储桶，如桶 3；最后，将新的索引信息存入 HBase 中，并根据不同类型的文件设置不同的标识信息，即可完成桶分裂过程.

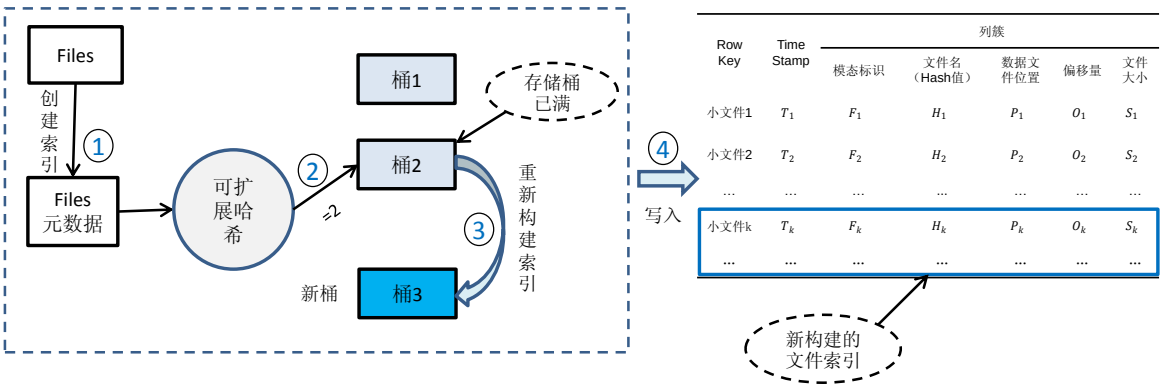


图 9 存储桶分裂

2.3 缓存机制

通常情况下，如果直接从 NameNode 中获取文件索引信息，读取相应的文件需要 6 步[7].

1) 客户端向 NameNode 发送读取文件内容的请求.
2) NameNode 通过内存查找文件索引信息.
3) NameNode 向客户端返回文件索引信息.
4) 客户端向 DataNode 发送读取文件请求.
5) DataNode 读取文件信息.
6) DataNode 返回文件内容给客户端.

根据上述步骤，可将 HDFS 读取文件分为两个阶段：第 1 阶段，从 NameNode 获取文件元数据信息；第 2 阶段，根据在 NameNode 中获取的元数据信息，客户端从 DataNode 中读取对应的文件内容. 由此可知，读取文件的操作需要耗费很长的时间. 若使用缓存机制将文件的索引信息存储在缓存中，则当读取文件内容时无须向 NameNode 发送请求，直接通过缓存中的索引信息在 DataNode 中查找对应文件内容即可，节省了客户端和 NameNode 通信的时间开销，从而有效减少了文件的读取时间.

本文使用 LRU (least recently used，最近最少使用)缓存机制缓存文件元数据信息. 在查询小文件时，若 HBase 中无法找到相应的文件索引信息，则通过缓存信息进行查找，因此保证文件的查询效率. LRU 算法的核心思想是：“如果数据最近被访问过，那么将来被访问的概率也更高”. 因此，依据最近被访问的情况，淘汰最长时间未访问的数据.

图 10 展示了 LRU 的缓存原理以及 LRU 缓存实例，从图中可以看出，当缓存空间未满时，缓存队列会一直加入新的数据；当缓存队列已满且此时缓存中的数据没有被访问时，淘汰最先进入缓存队列的数据；当缓存队列已满且缓存中的数据被访问时，将缓存中的数据放在队列头部并读取，其余数据依次往下压.

LRU 是一种常见的缓存算法，广泛运用于分布式存储框架中，其操作简单，当存在热点数据时，具有很高的效率. 然而，对于一些偶发性和周期性的批量操作，会导致 LRU 的命中率下降，出现“缓存污染”问题，这也是后续研究的问题.

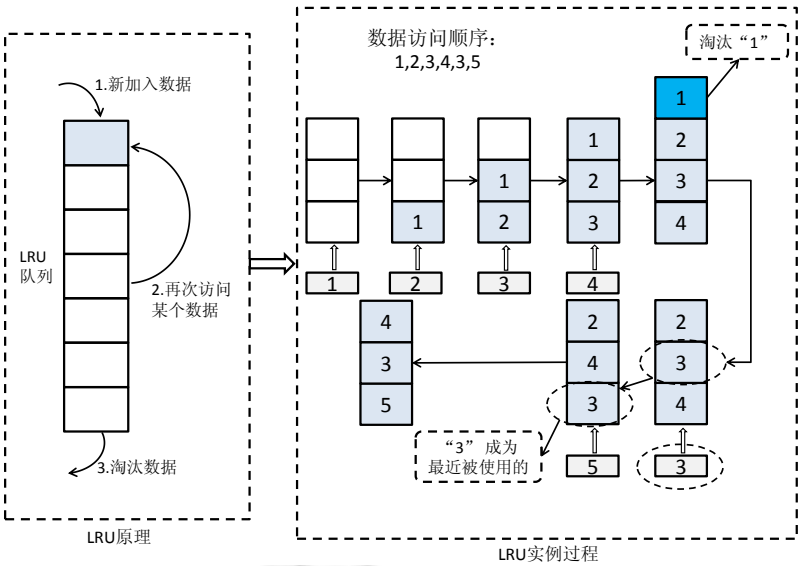


图 10 LRU 缓存机制

2.4 压缩机制

本文使用 LZ4 压缩算法对合并文件进行压缩. LZ4 是一种无损压缩算法, 着重于压缩和解压缩的速度, 因此对于压缩后文件读取效率上具有很大的优势. 它是 LZ77 压缩算法的变体, 核心为采用一系列查字典、匹配以及替换等操作达到去冗余的效果[36]. LZ4 压缩文件以 LZ4 序列组成[37], 序列格式见表 3.

表 3 LZ4 序列格式

令牌(token)	字符串长度(literal length)	字符串(literals)	偏移量(offset)	匹配长度(match length)
1 Byte	0–*N* Byte	0–*L* Byte	2 Byte	0–*M* Byte

LZ4 算法步骤如下.

- 第 1 步, 查字典. 初始情况下, 将字典和已处理区域大小设为 0, 数据流入为 4 字节. 由于没有匹配项, 因此直接写入字典中. 然后, 将字典后待处理区域的 1 个字符与字典后 3 个字符拼接成 4 个字符的字符串, 并在字典中搜索与之相同的字符串: 如果找到, 则进行匹配; 如果没有找到, 则字典后边界向后移动 1 个字符.
- 第 2 步, 匹配. 使用哈希方法来计算数据的哈希值, 并在字典中查找可能匹配的字符串. 匹配操作依次对比字典和待处理字符串, 若找到匹配, 则比较当前数据的后续数据与匹配数据的后续数据: 相同则继续向后匹配, 不同则终止匹配.
- 最后, 替换. 在匹配操作时, 会获取匹配长度和匹配距离. 匹配长度是正在处理的字符串能在字典中找到匹配的最大长度, 匹配距离指正在处理串的首地址和匹配的字典中字符串的首地址偏移量. 通过这两个值替换字典中匹配的字符串, 从而减少字符串占用的体积.

对于 MapFile 合并算法来说, 使用 Hadoop 默认压缩方法, 通过在 record 级别和 block 级别进行压缩, 在文件读取速度上较慢. LZ4 的文件压缩率较低, 因此压缩后文件大小相对于 MapFile 压缩后的文件较大, 但是其读取效率比 MapFile 要高. 由于文件合并后的操作主要为读取操作, 文件的存取效率是关注的重点. 综合来看, LZ4 压缩算法对于本文文件合并算法较为适用. 具体实验结果将在第 3 节实验中展示.

3 实 验

3.1 实验配置

实验使用 15 台服务器进行分布式集群的搭建, 服务器操作系统为 Ubuntu 20.04.1 LTS 系统, 处理器为 Intel(R) Core(TM) i7-10700 CPU@2.90 GHz, 内存大小为 32 GB, 采用 15 个节点的分布式集群, 其中一台作为 NameNode 节点, 其他作为 DataNode 节点. 客户端操作系统为 Windows10, 处理器为 Intel(R) Core (TM) i5-9500 CUP@3.00 GHz, 内存大小为 24.0 GB. 使用 Hadoop 版本为 Hadoop2.9, 客户端使用 Java 语言, JDK 版本为 jdk1.8.0_216. 文件存储副本数为默认的 3 副本策略, 块大小为默认 128 MB. 实验使用的数据集为 MIMIC IV[38,39]和 MIMIC Chest X-ray (MIMIC-CXR)[39−41]医疗数据.

MIMIC IV 数据集是 MIMIC III 数据集的延续, MIMIC III 数据集提供了 Beth Israel Deaconess Medical Center (BIDMC)重症监护病房超过 4 万名患者的重症监护数据, 在推动临床信息学、流行病学和机器学习的大量研究方面, MIMIC-III 提供了重要的帮助. 作为 MIMIC III 的延续, MIMIC IV 数据集采用一种模块化的方法来组织数据, 并且支持医疗保健领域的应用. 本文使用 MIMIC IV 数据集的结构化表格数据(csv 类型).

MIMIC-CXR 数据集是一个大型公共可用的胸片数据集, 以 DICOM 格式和文本格式的放射学诊断报告类型的数据组成. 该数据集包含 377 110 张图像, 对应于 Beth Israel Deaconess Medical Center 的 227 835 项放射学研究. 数据集旨在支持广泛的医学研究, 包括图像理解、自然语言处理和决策支持等. 本文使用 MIMIC-CXR 的两种模态数据, 分别为文本数据(txt 类型)以及医疗影像数据(dcm 类型).

本文使用上述 MIMIC IV 和 MIMIC-CXR 的表格数据(csv 类型)、文本数据(txt 类型)以及医疗影像数据(dcm 类型)这 3 种模态的医疗数据进行实验, 为了使数据规模更大, 且 3 种数据按一定比例分布, 根据 3 种医疗数据类型生成小文件. 由于 txt 类型的文件较小, 因此使用的 txt 类型文件数量最多. 这里使用原始医疗数据与生成医疗数据文件数量总和为 400 000 份, 文件大小在 1KB−15MB 之间. 为了更好地测试实验效果, 将 400 000 份文件划分为 4 批, 分别为 100 000, 200 000, 300 000 和 400 000 份用于所有实验中, 大小分别为 1.26 GB, 3.33 GB, 4.54 GB 和 6.02 GB. 3 种类型的文件个数分布见表 4. 对比方法为原始 HDFS、HAR、MapFile、TypeStorage[29]以及 HPF[7].

表 4 3 种类型文件个数分布(单位: 份)

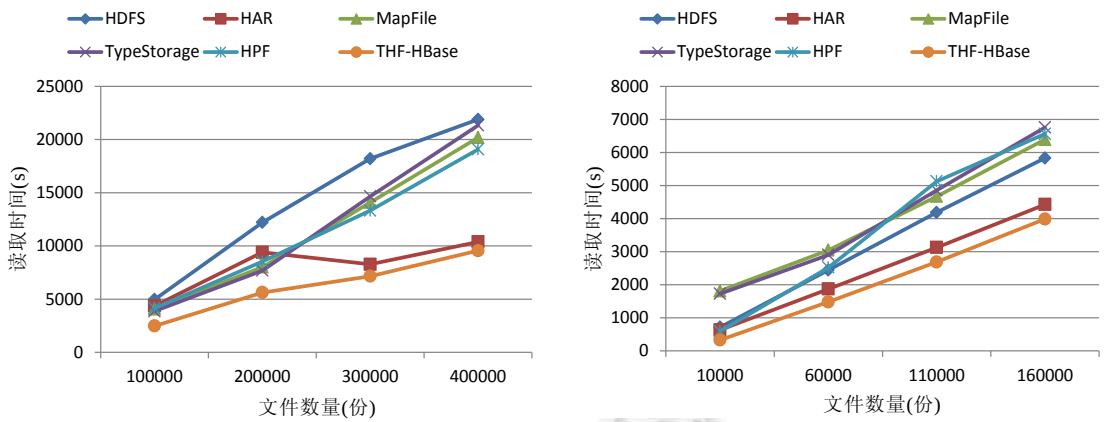
	txt类型	csv类型	dcm类型	总数
第1批	60 000	30 000	10 000	100 000
第2批	120 000	60 000	20 000	200 000
第3批	150 000	120 000	30 000	300 000
第4批	200 000	150 000	50 000	400 000

3.2 文件读取性能

在文件读取性能的比较中, 将实验分为两组.

- 第 1 组, 使用第 2 批文件, 总数为 200 000 份. 如表 4 所示, 其中, txt、csv 和 dcm 这 3 种类型文件分别为 120 000, 60 000, 20 000 份. 在 200 000 份文件中, 分别随机读取 10 000, 60 000, 110 000 和 160 000 份文件, 其中, 3 种类型文件的随机读取份数, 按随机读取总数占使用文件总数的比例读取.
- 第 2 组, 分别批量读取 4 批文件, 文件份数为 100 000, 200 000, 300 000 和 400 000, 3 种类型所占个数如表 4.

将批量读取文件时的读取时间进行 3 次实验, 取所有方法在 3 次实验中的平均时间作为最终结果, 所有对比方法使用其对应的缓存策略. 实验结果如图 11 所示, 从图中可以看出, 无论是随机读取一定量的文件, 还是读取所有文件, THF-HBase 方法的读取速度都比其他文件合并方法要快, 特别是在读取大量文件时表现出了很大的优势.



(a) 读取所有存储文件的读取时间对比 (b) 随机读取部分存储文件的读取时间对比

图 11 文件读取时间对比

THP-HBase 读取文件速度快的原因如下: 第一, 在创建索引文件时, THF-HBase 使用单层索引文件, 而 HAR 文件和 MapFile 文件使用两层索引, 因此需要更多的时间读取索引信息; 第二, 原始 HDFS 将文件元数据信息直接存储在 NameNode 中, 而 THF-HBase, HPF, TypeStorage, HAR 和 MapFile 使用文件合并策略, 减少了和 NameNode 交互, 降低了文件读取时间; 第三, 由于 MapFile 需要两层解压, 而 THF-HBase 只需要一层解压, 从而在读取文件时使用更多的时间; 第四, 虽然 HPF 文件使用单层索引, 但是需要根据文件的 hashname 值计算得到文件位置和偏移量等信息, 因此比直接从 HBase 中读取文件要慢; 第五, TypeStorage 虽然使用单层索引文件, 但在文件读取时过于依赖缓存的大小, 且会根据不同类型文件分类读取索引信息, 因此在随机读取不同类型文件时, 比 THF-HBase 速度更慢. 值得注意的是, THF-HBase 文件使用 MWHC 哈希函数对排序后的文件索引进行存储, 在读取时比 HPF 所用完美哈希方法速度快, 在后续实验中将会进行验证. 综上所述, THF-HBase 文件读取性能最佳.

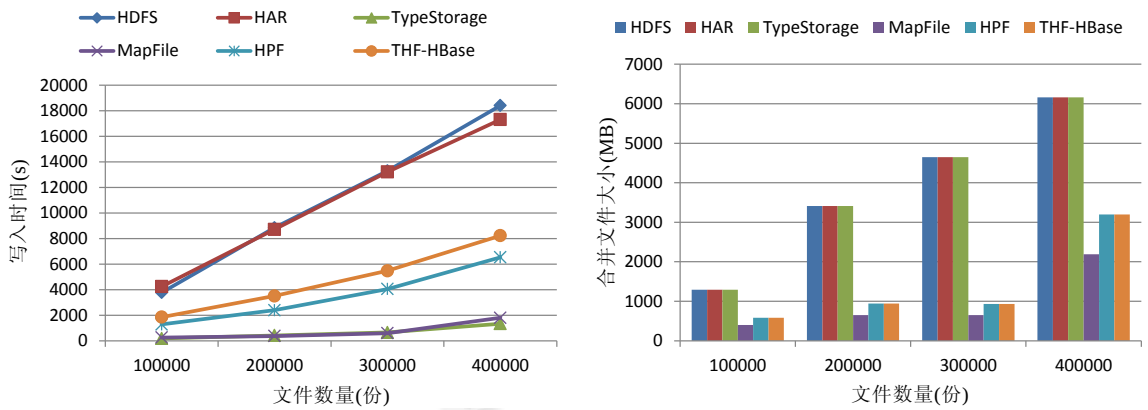
3.3 文件写入性能

对于文件写入 HDFS 的性能测试, 分别使用 4 批文件进行文件写入实验, 文件个数及文件类型占比如表 4 所示. 从文件的写入时间和创建的文件大小两方面进行实验, 这里设置 THF-HBase 的存储桶大小为 200 000 个记录. 文件写入时间的对比结果如图 12(a)所示, 从图中可以看出, MapFile 和 TypeStorage 文件写入时间最短, THF-HBase 方法和 HPF 方法文件写入时间仅次于 MapFile 和 TypeStorage 方法. 出现上述结果的原因如下: 首先, 由于 THF-HBase 在写入文件时, 需要将元数据信息写入到 HBase 表中, 因此会增加文件写入时间, 写入时间略大于 MapFile, HPF 以及 TypeStorage 方法; 第二, HAR 在上传过程中需要将所有小文件上传至 HDFS, 然后再进行合并, 且 HAR 文件使用 MapReduce 进行打包, 需要消耗很多时间从 NameNode 中获取文件元数据信息, 然后执行 map 任务, 而 THF-HBase 方法在文件创建过程中不需要先将小文件上传, 从而写入效率远高于 HAR 方法; 第三, 由于 HDFS 直接将大量文件上传, 增加了文件通过网络 I/O 传输以及写入 DataNode 的负担, 因此写入速度会远大于 HPF 文件、THF 文件和 MapFile 和 TypeStorage 文件.

然而, 如果不将 THF 文件的元数据信息存入 HBase 中, 文件的合并时间会比 HPF 短, 具体实验结果将在第 3.5.2 节进行详细说明. 此外, 虽然 THF-HBase 方法的文件合并时间略高于 TypeStorage, HPF 和 MapFile, 但是 THF-HBase 方法可以直接从 HBase 中读取文件索引信息, 文件读取性能会大幅度提升, 如第 3.2 节所示. 由于文件存储大多为“存一次, 取多次”, 综合来看, THF-HBase 的性能更佳.

为了进一步比较文件的写入性能, 将 6 种方法在 DataNode 中占用磁盘的大小进行比较. 从图 12(b)中可以看出, MapFile 文件占用空间最小, THF-HBase 文件和 HPF 文件次之且大小相同, 而 HDFS、HAR 和 TypeStorage 方法则和文件原始大小相同. 这是由于在文件创建过程中, THF-HBase, HPF 和 MapFile 采用压缩方式进行文

件存储，而 HAR 和 TypeStorage 文件在创建过程中并没有使用压缩方法，而是直接将小文件合并，因此文件大小和原始 HDFS 相似. 此外，由于 THF-HBase 文件和 HPF 文件在压缩过程中使用 LZ4 压缩方法来保证文件的存取效率，且仅在文件块级别上进行压缩，而 MapFile 在文件和元数据中都进行压缩，因此 MapFile 合并文件大小会比 THF-HBase 文件和 HPF 文件小. 但是由于 MapFile 采用两层压缩，因此所花费的读取时间最多，在整体性能上表现出劣势.



(a) 文件写入时间对比　　(b) 合并文件大小对比

图 12　文件写入性能测试

3.4 NameNode内存使用

表 5 展示了 6 种方法的 NameNode 内存使用情况.

表 5　NameNode 内存使用(KB)

文件数量(份)	HDFS	HAR	MapFile	HPF	TypeStorage	THF-HBase
100 000	61 600	2.83	3.428	2.83	11.65	**2.83**
200 000	123 200	4.294	4.892	2.83	23.728	**2.83**
300 000	184 800	5.392	4.892	3.446	30.682	**3.446**
400 000	246 400	5.392	13.676	5.892	32.512	**5.892**

为了在每个数据块中存储更多的文件，将 THF-HBase 文件的块大小设置为 512 MB，HAR、MapFile、TypeStorage 和 HPF 文件分别使用默认的 512 MB, 128 MB, 128 MB, 512 MB 块大小. 根据表中结果可以看出，HDFS 的 NamNode 内存使用最大，其他 5 种小文件合并方法的 NameNode 使用情况相差较小. 在文件创建过程中, THF-HBase、HPF、HAR、MapFile 和 TypeStorage 通过小文件合并的方式，大大减少了文件数量，从而减少了元数据信息. 对比其他小文件合并方法，THF-HBase 的 NameNode 内存使用最小. 此外，由于 MapFile 和 TypeStorage 方法默认使用 128 MB 的块大小，因此在 NameNode 内存使用上增长趋势较明显.

3.5 其他性能比较

THF-HBase 在索引构建与存储、缓存等方面使用了一些优化方法，为了更好地验证使用策略的有效性，采用以下实验进行验证，每一类进行 3 次实验取平均时间.

3.5.1 缓存性能测试

为了评估 THF-HBase 方法使用的 LRU 缓存策略的效果，在读取文件时，不通过 HBase 表进行索引的读取，而是通过读取缓存中的索引信息进行文件的读取操作. 将使用 LRU 缓存和不使用 LRU 缓存时文件读取时间进行比较. 读取文件分别为 100 000, 200 000, 300 000 和 400 000 份，结果如图 13 所示. 从图中可以看出，使用 LRU 缓存后，文件的读取效率有了显著的提升. 当文件读取数量为 400 000 份时，文件读取效率提升最明显，不使用缓存的读取时间为 61 864.616 s，而使用缓存的读取时间仅为 15 671.57 s，读取时间减少了 74.67%. 此外，不使用缓存的时间增长幅度比使用缓存的增长幅度要大. 将文件的索引信息存在缓存中，在读取文件

时, 无须向 NameNode 发送请求读取, 直接在缓存中查找对应的文件索引信息, 因此减少了文件查找时间. 当 HBase 中无法查询文件索引信息时, 可以在缓存中进行查询, 进一步提升文件查找效率.

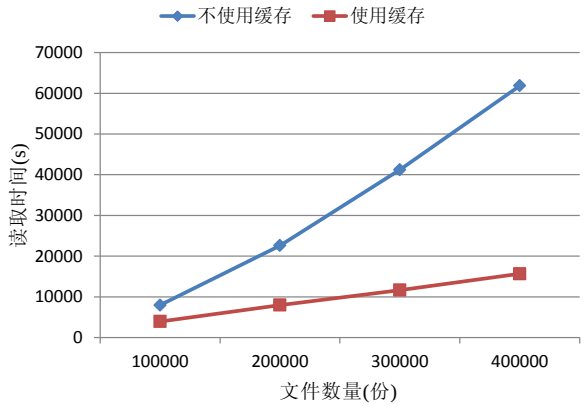
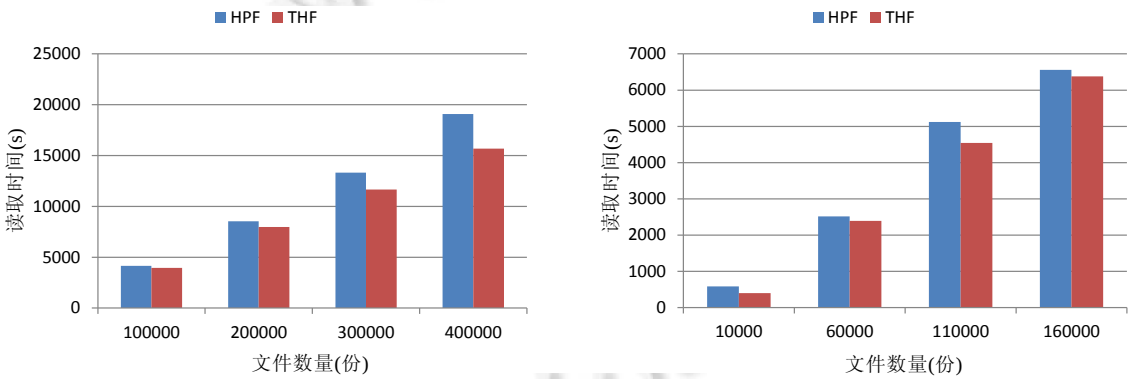


图 13 文件缓存性能测试

3.5.2 MWHC 哈希方法性能测试

本文在每个存储桶中, 使用 MWHC 哈希函数存储每个文件索引信息在索引文件中的位置信息. 为了比较保序完美哈希方法和 MWHC 哈希方法的文件读写效率, 在 THF-HBase 中不使用 HBase 存储文件元数据信息, 这里记为 Two-layer Hash File (THF). THF 和 HPF 的文件读写效率如图 14、15 所示.



(a) 读取所有存储文件的读取时间对比 (b) 随机读取部分存储文件的读取时间对比

图 14 使用 MWHC 哈希方法的读取性能测试

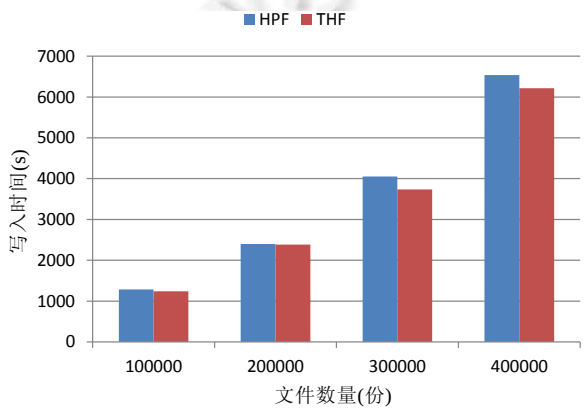
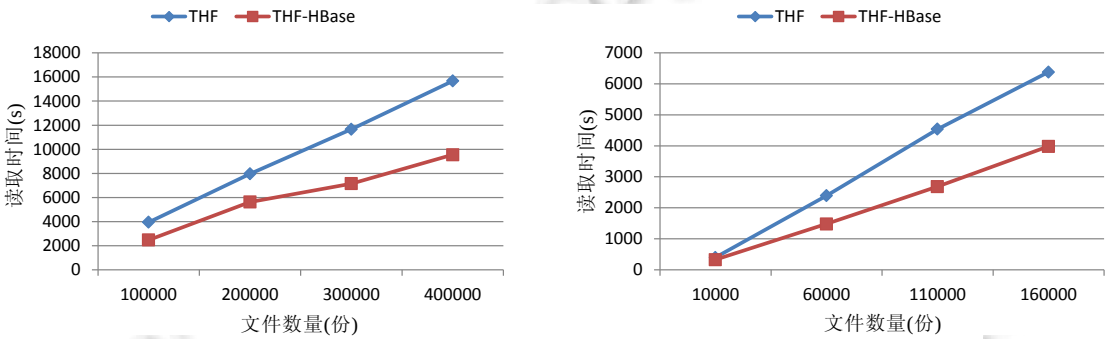


图 15 使用 MWHC 哈希方法的写入性能测试

对于文件读取，将实验分为两组，与第 3.2 节两组测试相同. 从图 14(a)、(b)中可以看出，无论是随机读取文件还是读取所有文件，THF 的时间都比 HPF 用时短. 文件写入时间对比如图 15 所示，写入文件分别为 100 000, 200 000, 300 000 和 400 000 个. 从图 15 中可以看出, THF 在文件写入上时间更短，而且存入文件数量越多，THF 在文件写入取效率上更具优势. 综合文件读取和写入效率来看，使用 MWHC 哈希函数能够提升文件存取速率.

3.5.3 使用 HBase 性能测试

为了验证使用 HBase 的效果，将不使用 HBase 存储索引与使用 HBase 的存取结果进行比较. 文件的读取效率如图 16 所示，测试分为两组，与第 3.5.2 节两组测试相同. 使用 HBase 进行文件读取，在读取效率上得到了很大的提升. 当读取文件数量增加时，使用 HBase 的读取优势更加明显. 在图 16(a)中，当随机读取文件为 400 000 份时，THF 的读取时间为 15 671.574 s，而 THF-HBase 的读取时间为 9 549.716 s，读取时间减少了 39.06%. 在图 16(b)中，当随机读取部分文件为 160 000 份时，THF 的读取时间为 6 380.999 s，而 THF-HBase 的读取时间为 3 983.734 s，读取时间减少了 37.58%.



(a) 读取所有存储文件的读取时间对比　　(b) 随机读取部分存储文件的读取时间对比

图 16　HBase 读取性能测试

文件写入效率如图 17 所示，使用 HBase 文件写入时间比不使用 HBase 所花费的时间多. 这是因为在文件进行写入时，创建的索引文件需要单独存入 HBase 中，因此时间消耗会更多. 但是从图 17 中可以看出，THF-HBase 与 THF 的文件写入时间的差距并不随写入文件数量的增加而增大. 由于文件操作大多为“存一次，取多次”，因此文件的读取效率对于文件合并至关重要. 综合文件的读写效率，使用 HBase 能够得到更好的效果.

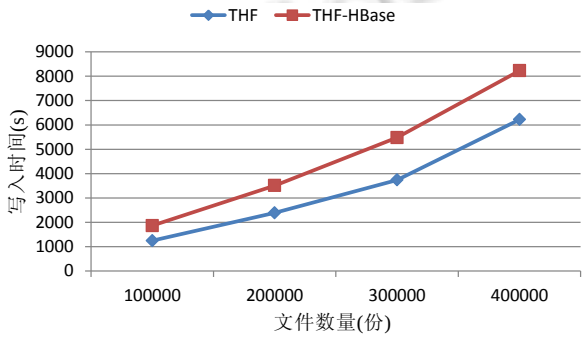


图 17　HBase 写入性能测试

4 结　论

HDFS 通常用于大文件的存储和管理，海量小文件的存储是制约 HDFS 性能的一个重要方面. 本文针对多模态医疗数据中海量小文件存储问题，提出一种基于双层哈希编码和 HBase 的优化方法. 该方法主要运用

EHT和MWHC哈希函数构建索引文件, 文件读取的时间复杂度为O(1). 同时, 使用HBase存储文件索引信息, 并构建标识列来标识不同模态的医疗数据, 方便多模态医疗数据的存储管理, 且使文件索引读取不依赖于缓存容量大小, 进一步提高了文件的读取效率. 此外, 建立了基于 LRU 的元数据预取机制, 将最近访问的元数据信息放入客户端内存中, 从而提高了文件的读取速度; 并采用 LZ4 压缩算法对合并文件进行压缩存储, 在充分考虑文件读取性能的同时, 减少文件的存储空间. 将本文算法和原始 HDFS、HAR、MapFile、TypeStorage 以及 HPF 文件存储方法进行对比, 结果表明, 本文算法在小文件读取效率、内存使用率、缓存性能等方面具有明显的优势.

在后续的工作中, 将会针对提出的算法进行一些改进, 使算法性能更优.

- 第一, 针对多模态医疗数据方面的优化, 如结合患者的统计信息、诊断报告、CT 图像等数据之间的关联性进行分类存储.
- 第二, 针对文件存取性能方面的优化, 如实现文件部分删除功能; 对缓存机制进行优化, 解决“缓存污染”问题.
- 第三, 针对多模态医疗数据存储的底层文件系统, 考虑使用其他方式进行数据存储.

References:

[1] Chen CT, Hsu CC, Wu JJ, Liu PF. GFS: A distributed file system with multi-source data access and replication for grid computing. In: Proc. of the Advances in Grid and Pervasive Computing. Springer, 2009. 119−130. [doi: 10.1007/978-3-642-01671-4_12]

[2] Shvachko K, Kuang H, Radia S, Chansler R. The Hadoop distributed file system. In: Proc. of the IEEE 26th Symp. on Mass Storage Systems and Technologies (MSST). IEEE, 2010. 1−10. [doi: 10.1109/MSST.2010.5496972]

[3] Braam PJ. The lustre storage architecture. arXiv:1903.01955, 2019.

[4] Apache Hadoop. 2020. https://hadoop.apache.org/

[5] Jin GD, Bian HQ, Chen YG, Du XY. Survey on storage and optimization techniques of HDFS. Ruan Jian Xue Bao/Journal of Software, 2020, 31(1): 137−161 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/5872.htm [doi: 10.13328/j.cnki.jos.005872]

[6] Srinithya L, Reddy G. Performance evaluation of Hadoop distributed file system and local file system. Int’l Journal of Science and Research (IJSR), 2015, 3(9): 1174−1183.

[7] Zhai YL, Tchaye-Kondi J, Lin KJ, Zhu LH, Tao WJ, Du XJ, Guizani M. Hadoop perfect file: A fast and memory-efficient metadata access archive file to face small files problem in HDFS. Journal of Parallel and Distributed Computing, 2021, 156: 119−130. [doi: 10.1016/j.jpdc.2021.05.011]

[8] Hadoop archives guide. 2020. https://hadoop.apache.org/docs/r1.2.1/Hadoop_archives.html

[9] Vorapongkitipun C, Nupairoj N. Improving performance of small-file accessing in Hadoop. In: Proc. of the 11th Int’l Joint Conf. on Computer Science and Software Engineering (JCSSE). IEEE, 2014. 200−205. [doi: 10.1109/JCSSE.2014.6841867]

[10] Renner T, Müller J, Thamsen L, Kao O. Addressing Hadoop’s small file problem with an appendable archive file format. In: Proc. of the Computing Frontiers Conf. ACM 2017. 367−372. [doi: 10.1145/3075564.3078888]

[11] Sethia D, Sheoran S, Saran H. Optimized MapFile based storage of small files in Hadoop. In: Proc. of the 17th IEEE/ACM Int’l Symp. on Cluster, Cloud and Grid Computing (CCGRID). IEEE, 2017. 906−912. [doi: 10.1109/CCGRID.2017.83]

[12] Bing M, Guo WB, Fan GS, Qian NW. A novel approach for efficient accessing of small files in HDFS: TLB-mapfile. In: Proc. of the 17th IEEE/ACIS Int’l Conf. on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD). IEEE, 2016. 681−686. [doi: 10.1109/SNPD.2016.7515978]

[13] Dong B, Qiu J, Zheng QH, Zhong X, Li JW, Li Y. A novel approach to improving the efficiency of storing and accessing small files on Hadoop: A case study by powerpoint files. In: Proc. of the IEEE Int’l Conf. on Services Computing. IEEE, 2010. 65−72. [doi: 10.1109/SCC.2010.72]

[14] Liu XH, Han JZ, Zhong YQ, Han CD, He XB. Implementing WebGIS on Hadoop: A case study of improving small file I/O performance on HDFS. In: Proc. of the IEEE Int’l Conf. on Cluster Computing and Workshops. IEEE, 2009. 1−8. [doi: 10.1109/CLUSTR.2009.5289196]

[15] Zheng T, Guo WB, Fan GS. Research on optimization method of merging and prefetching for massive small files in HDFS. Computer Science, 2017, 44(Z11): 516−519, 541 (in Chinese with English abstract).

[16] Liu XJ, Xu ZQ, Pan SM. A massive small file storage solution combination of RDBMS and Hadoop. Geomatics and Information Science of Wuhan University, 2013, 38(1): 113−115, 120 (in Chinese with English abstract). [doi: 10.13203/j.whugis2013.01.011]

[17] Lakshman A, Malik P. Cassandra: A decentralized structured storage system. Association for Computing Machinery, 2010, 44(2): 35−40. [doi: 10.1145/1773912.1773922]

[18] TFS. 2020. https://github.com/alibaba/tfs

[19] Dong B, Zheng QH, Tian F, Chao KM, Ma R, Anane R. An optimized approach for storing and accessing small files on cloud storage. Journal of Network and Computer Applications, 2012, 35(6): 1847−1862. [doi: 10.1016/j.jnca.2012.07.009]

[20] Fu SL, He LG, Huang CL, Liao XK, Li KL. Performance optimization for managing massive numbers of small files in distributed file systems. IEEE Trans. on Parallel and Distributed Systems, 2015, 26(12): 3433−3448. [doi: 10.1109/TPDS.2014.2377720]

[21] Beaver D, Kumar S, Li HC, Sobel J, Vajgel P. Finding a needle in haystack: Facebook's photo storage. In: Proc. of the 9th USENIX Conf. on Operating Systems Design and Implementation (OSDI 2010). ACM, 2010. 47−60.

[22] Chandrasekar S, Dakshinamurthy R, Seshakumar PG, Prabavathy B, Babu C. A novel indexing scheme for efficient handling of small files in Hadoop distributed file system. In: Proc. of the Int'l Conf. on Computer Communication and Informatics. IEEE, 2013. 1−8. [doi: 10.1109/ICCCI.2013.6466147]

[23] Bende S, Shedge R. Dealing with small files problem in Hadoop distributed file system. Procedia Computer Science, 2016, 79: 1001−1012. [doi: 10.1016/j.procs.2016.03.127]

[24] Phakade P, Raut S. An innovative strategy for improved processing of small files in Hadoop. Int'l Journal of Application of Innovation in Engineering and Management (IJAIEM), 2014, 3(7): 278−280.

[25] Choi C, Choi C, Choi J, Kim P. Improved performance optimization for massive small files in cloud computing environment. Annals of Operations Research, 2018, 265: 305−317. [doi: 10.1007/s10479-016-2376-0]

[26] Zhou F, Pham H, Yue JH, Zou H, Yu WK. SFMapReduce: An optimized MapReduce framework for small files. In: Proc. of the IEEE Int'l Conf. on Networking, Architecture and Storage (NAS). IEEE, 2015. 23−32. [doi: 10.1109/NAS.2015.7255218]

[27] You XR, Cao S. Storage research of small files in massive education resource. Computer Science, 2015, 42(10): 76−80 (in Chinese with English abstract).

[28] Zhao XY, Yang Y, Sun LL, Chen Y. Hadoop-based storage architecture for mass MP3 files. Journal of Computer Applications, 2012, 32(6): 1724−1726 (in Chinese with English abstract). [doi:10.3724/SP.J.1087.2012.01724]

[29] Qin JW, Liu H, Fang MY. Type-based small file merging method on big data platform. Software Engineering, 2020, 23(10): 12−14, 11 (in Chinese with English abstract). [doi: 10.19644/j.cnki.issn2096-1472.2020.10.004]

[30] Fagin R, Nievergelt J, Pippenger N, Strong HR. Extendible hashing—A fast access method for dynamic files. ACM Trans. on Database Systems, 1979, 4(3): 315−344. [doi: 10.1145/320083.320092]

[31] Majewski BS, Wormald NC, Havas G, Czech ZJ. A family of perfect hashing methods. Computer Journal, 1996, 39(6): 547−554. [doi: 10.1093/comjnl/39.6.547]

[32] CMPH—C Minimal Perfect Hashing Library. 2020. http://cmph.sourceforge.net/

[33] Belazzougui D, Boldi P, Pagh R, Vigna S. Theory and practice of monotone minimal perfect hashing. ACM Journal of Experimental Algorithmics, 2011, 16: Article No.3. [doi: 10.1145/1963190.2025378]

[34] HBase. 2021. https://hbase.apache.org/

[35] LZ4. 2021. https://github.com/lz4/

[36] Gu W. The optimization of LZ4 lossless compression algorithm based on FPGA [MS. Thesis]. Nanjing: Southeast University, 2017 (in Chinese with English abstract).

[37] Mei FQ. Research on big data encryption algorithm based on LZ4 data compression and lattice encryption [MS. Thesis]. Nanjing: Nanjing University of Aeronautics and Astronautics, 2019 (in Chinese with English abstract). [doi: 10.27239/d.cnki.gnhhu.2019.000241]

[38] Johnson A, Bulgarelli L, Pollard T, Horng S, Celi LA, Mark R. MIMIC-IV (Version 1.0). PhysioNet, 2021. [doi: 10.13026/s6n6-xd98]

[39] Goldberger A, Amaral L, Glass L, Hausdorff J, Ivanov PC, Mark R, Stanley HE. PhysioBank, PhysioToolkit, and PhysioNet: Components of a new research resource for complex physiologic signals. Circulation, 2000, 101 (23): 215−220.

[40] Johnson A., Pollard T, Mark R, Berkowitz S, Horng S. MIMIC-CXR Database (Version 2.0.0). PhysioNet, 2019. [doi: 10.13026/C2JT1Q]

[41] Johnson AEW, Pollard TJ, Berkowitz SJ, Greenbaum NR, Lungren MP, Deng CY, Mark RG, Horng S. MIMIC-CXR, a de-identified publicly available database of chest radiographs with free-text reports. Scientific Data, 2019, 6(1): Article No.317. [doi: 10.1038/s41597-019-0322-0]

附中文参考文献:

[5] 金国栋, 卞昊穹, 陈跃国, 杜小勇. HDFS 存储和优化技术研究综述. 软件学报, 2020, 31(1): 137−161. http://www.jos.org.cn/1000-9825/5872.htm [doi: 10.13328/j.cnki.jos.005872]

[15] 郑通, 郭卫斌, 范贵生. Hdfs 中海量小文件合并与预取优化方法的研究. 计算机科学, 2017, 44(Z11): 516−519, 541.

[16] 刘小俊, 徐正全, 潘少明. 一种结合 RDBMS 和 Hadoop 的海量小文件存储方法. 武汉大学学报(信息科学版), 2013, 38(1): 113−115, 120. [doi: 10.13203/j.whugis2013.01.011]

[27] 游小容, 曹晟. 海量教育资源中小文件的存储研究. 计算机科学, 2015, 42(10): 76−80.

[28] 赵晓永, 杨扬, 孙莉莉, 陈宇. 基于 Hadoop 的海量 MP3 文件存储架构. 计算机应用, 2012, 32(6): 1724−1726. [doi: 10.3724/SP.J.1087.2012.01724]

[29] 秦加伟, 刘辉, 方木云. 大数据平台下基于类型的小文件合并方法. 软件工程, 2020, 23(10): 12−14, 11. [doi: 10.19644/j.cnki.issn2096-1472.2020.10.004]

[36] 顾巍. 基于 FPGA 的 LZ4 无损压缩算法优化设计[硕士学位论文]. 南京: 东南大学, 2017.

[37] 梅发强. 基于 LZ4 数据压缩与格加密的大数据加密算法研究[硕士学位论文]. 南京: 南京航空航天大学, 2019. [doi: 10.27239/d.cnki.gnhhu. 2019.000241]

曾梦(1995－), 女, 博士生, 主要研究领域为分布式系统, 大数据存储管理, 机器学习.

杨雪冰(1991－), 男, 博士, 副研究员, 主要研究领域为机器学习, 大数据知识挖掘, 人工智能在智慧气象及辅助医疗的交叉应用.

邹北骥(1961－), 男, 博士, 教授, 博士生导师, CCF 杰出会员, 主要研究领域为医疗人工智能, 医学大数据分析.

朱承璋(1978－), 女, 博士, 副教授, CCF 高级会员, 主要研究领域为机器学习, 医学图像处理.

张文生(1965－), 男, 博士, 研究员, 博士生导师, 主要研究领域为人工智能, 跨模态数据标注, 医疗数据分析推理.