软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2020,31(1):137−161 [doi: 10.13328/j.cnki.jos.005872]

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

HDFS 存储和优化技术研究综述*

金国栋[1,2], 卞昊穹[1,2], 陈跃国[1,3], 杜小勇[1,2]

[1](数据工程与知识工程教育部重点实验室(中国人民大学),北京 100872)

[2](中国人民大学 信息学院,北京 100872)

[3](大数据系统软件国家工程实验室(北京理工大学),北京 100081)

通讯作者: 陈跃国, E-mail: chenyueguo@ruc.edu.cn

摘 要: HDFS(Hadoop distributed file system)作为面向数据追加和读取优化的开源分布式文件系统,具备可移植、高容错和可大规模水平扩展的特性.经过10余年的发展,HDFS已经广泛应用于大数据的存储.作为存储海量数据的底层平台,HDFS 存储了海量的结构化和非结构化数据,支撑着复杂查询分析、交互式分析、详单查询、Key-Value读写和迭代计算等丰富的应用场景.HDFS 的性能问题将影响其上所有大数据系统和应用,因此,对 HDFS 存储性能的优化至关重要.介绍了 HDFS 的原理和特性,对已有 HDFS 的存储及优化技术,从文件逻辑结构、硬件设备和应用负载这3个维度进行了归纳和总结.综述了近年来 HDFS 存储和优化相关研究.未来,随着 HDFS 上层应用的日益丰富和底层硬件平台的发展,基于异构平台的数据存储、面向应用负载的自适应存储优化以及结合机器学习的存储优化技术将成为未来研究的主要方向.

关键词: HDFS;分布式文件系统;存储系统优化;数据分析

中图法分类号: TP311

中文引用格式: 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

英文引用格式: 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). http://www.jos.org.cn/1000-9825/5872.htm

Survey on Storage and Optimization Techniques of HDFS

JIN Guo-Dong[1,2], BIAN Hao-Qiong[1,2], CHEN Yue-Guo[1,3], DU Xiao-Yong[1,2]

[1](Key Laboratory of Data Engineering and Knowledge Engineering of Ministry of Education (Renmin University of China), Beijing 100872, China)

[2](School of Information, Renmin University of China, Beijing 100872, China)

[3](National Engineering Laboratory of Big Data System Software (Beijing Institute of Technology), Beijing 100081, China)

Abstract: As an append-only and read optimized open-source distributed file system, HDFS (Hadoop distributed file system) provides portability, high fault-tolerance, and massive horizontal scalability. Over the past decade, HDFS has been widely used for big data storage, and it manages various data, such as text, graph, key-values, etc. Moreover, big data systems based on or compatible with HDFS have been prevalent in many application scenarios such as complex SQL analysis, ad-hoc queries, interactive analysis, key-value storage, and iterative computation. HDFS has been the universal underlying file system to store massive data and support manifold analytical applications. Therefore, it is of great significance to optimizing the storage performance and data access efficiency of HDFS. In this study,

* 基金项目: 国家重点研发计划(2018YFB1004401); 国家自然科学基金(U1711261, 61432006, 61732014)

Foundation item: National Key Research and Development Program of China (2018YFB1004401); National Natural Science Foundation of China (U1711261, 61432006, 61732014)

收稿时间: 2019-01-17; 修改时间: 2019-03-11; 采用时间: 2019-05-30; jos 在线出版时间: 2019-08-09

CNKI 网络优先出版: 2019-08-12 12:07:56, http://kns.cnki.net/kcms/detail/11.2560.TP.20190812.1207.002.html

the principles and features of HDFS are summarized and a survey on storage and optimization techniques of HDFS is carried out from three dimensions, including logic file structure, hardware, and application scenarios. It is also proposed that storage over heterogeneous hardware, workload-guided adaptive storage optimization, and storage optimization combined with machine learning technologies could be the most appealing research directions in the future.

Key words: HDFS; distributed file system; storage system optimization; data analysis

近年来,随着大数据技术在各行各业中的应用和发展,大量的实际应用中数据量都超出了常规单机的存储和计算能力.分布式水平扩展成为了大数据系统的必由之路.在面向分析的大数据系统中,由于需要分析和处理大批量的历史和新生数据,大规模的水平扩展更成为了最为迫切的需求.HDFS(Hadoop distributed file system)旨在解决大数据的分布式存储问题,它随着 Hadoop 生态圈一起出现并发展成熟,具有开源、可移植、高可用、高容错、可大规模水平扩展等特性.随着 Hadoop 生态圈的繁荣发展,HDFS 也在各行各业的大数据系统中得到了广泛的应用.

HDFS 作为底层的文件存储系统,其上支撑着非常丰富的应用场景,如复杂查询分析、交互式分析、详单查询、Key-Value 存储和查询、迭代计算等.上层的系统和应用从 HDFS 读取数据并向 HDFS 写入数据或者中间结果.HDFS 的性能问题将影响到其上所有大数据系统和应用,因此,HDFS 性能的优化变得至关重要.但 HDFS 的性能优化非常具有挑战性.作为一个通用的分布式文件系统,在优化 HDFS 时,要保证其稳定和通用的访问接口以及高可用性和高可扩展等其他特性,其性能优化存在很多限制.并且,由于硬件设备的发展和 Hadoop 上应用的丰富性,性能优化往往需要针对特定的硬件和应用.因此,在保证 HDFS 基本接口和特性不被破坏的前提下,如何最大限度地优化 HDFS 的存储性能,在过去 10 余年中一直都是学术界和工业界研究的重点.已有的相关综述大多着眼于大数据管理和大数据分析系统的研究进展,并从宏观角度分析和对比了部分基于 HDFS 的扩展系统,但没有聚焦到底层的 HDFS 存储和优化技术[1,2].为了更好地优化 HDFS 系统的存储性能,本文对大数据分析背景下 HDFS 的存储和优化的研究成果进行了系统的整理和分析,以供后续研究参考.

本文第 1 节概述 HDFS 的发展历程,介绍 HDFS 的系统架构和基本原理,总结 HDFS 存储系统的特点和面临的挑战,并给出本文的研究框架.第 2 节从文件逻辑结构的角度介绍文件存储格式、文件压缩和数据索引等基于文件逻辑结构的 HDFS 存储和优化技术.第 3 节从 HDFS 系统硬件设备的维度出发,研究针对外存、内存和网络传输中不同硬件的存储和优化技术.第 4 节从应用的维度出发,概括了 HDFS 上的主要应用场景以及针对不同应用的存储和优化方法.最后,第 5 节总结全文,指出 HDFS 存储系统当前面临的挑战,并对未来研究方向加以展望.

1 HDFS 存储系统概述

本节介绍 HDFS 存储系统的发展历程以及系统架构和基本原理,并总结和归纳 HDFS 的存储特点及面临的主要挑战.最后,基于对 HDFS 存储特点和挑战的分析,给出本文的研究框架,即:从 HDFS 上数据存储和访问性能的 3 个主要影响维度,对已有的存储和优化技术进行研究和综述.

1.1 HDFS存储系统的发展历程

HDFS 是 Hadoop 中的分布式存储系统.Hadoop 最早起源于 Cafarella 和 Cutting 开发的 Nutch[3]. Nutch 是一个开源的搜索引擎,在分布式存储和计算框架方面参考了谷歌发表的关于 GFS[4]和 MapReduce[5]的两篇论文.之后,Nutch 中的分布式文件系统 NDFS(Nutch distributed file system)和分布式计算框架 MapReduce 独立开源,成为一个新的项目,即 Hadoop.此后,Hadoop 受到开源社区的青睐,日益流行,成为了 Apache 顶级项目,并在许多公司内部得到了应用,形成了一个成熟的生态系统.

作为 Hadoop 的重要子系统,HDFS 自开源后有多次重要的版本更新,不断加入新的特性和功能,以支持更可靠、规模更大、更高效的数据管理和访问.表 1 列出了自开源来 HDFS 的重要版本更新.2010 年发布的 0.21.0 版本中加入了 Append 功能,即用户可以追加写 HDFS 上的文件.HDFS 的 2.x 系列属于新一代 Hadoop 系统,该

大版本除了进行大量代码实现上的优化外,还支持 HDFS Federation 来增强系统的扩展性.另外,2.x 系列还适应存储硬件的发展,增加了新特性以提高数据管理和访问的效率.例如,Hadoop 在 2.3.0 版本开始支持异构存储(heterogeneous storage)[6]和集中式缓存管理(centralized cache management)[7],以更好地利用除磁盘外的存储介质提高存储性能;随后,在 2.6.0 版本中,HDFS 加入了对 SSD、内存的层级化存储的支持,还引入了 Archival Storage[8]来分离存储“冷”“热”数据,用户可以配置存储规则指定文件的存储介质和类型.2017 年 12 月,Apache Hadoop 发布了 3.0.0 版本.作为 3.x 版本的第一个 GA(general available)版本,HDFS 开始支持纠删码(erasure coding)[9]作为冗余副本之外的另一种数据容错方式,减少数据容错机制对存储空间的占用.

Table 1 Descriptions of important versions of HDFS in Apache Hadoop

表 1 Apache Hadoop 中 HDFS 的重要版本更新

序号	版本号	发布时间	内容
1	0.21.0	2010.08	支持 file append
2	0.22.0	2011.12	支持 symlink;优化 file append 的实现
3	1.0.2	2012.04	内置支持数据压缩和解压缩算法
4	2.0.0-alpha	2012.05	支持 HDFS Federation 增强 NameNode 的扩展性
5	2.1.0-beta	2013.08	支持 HDFS Snapshots;调整默认数据块大小为 128MB
6	2.2.0	2013.10	Apache Hadoop 2.x 的第一个 GA 版本
7	2.3.0	2014.02	支持 HDFS 的异构存储和集中式缓存管理
8	2.6.0	2014.11	在 HDFS 的异构存储中加入 SSD、内存和 Archival Storage 作为层级化存储
9	3.0.0	2017.12	Apache Hadoop 3.x 的第一个 GA 版本;支持纠删码

1.2 HDFS存储系统的架构和基本原理

HDFS 存储系统被设计用来在大规模的廉价服务器集群上可靠地存储大规模数据,并提供高吞吐的数据读取和追加式写入.单个 HDFS 集群可以扩展至几千甚至上万个节点[10].本节从 HDFS 的集群架构和数据块的放置与容错两个方面介绍 HDFS 的架构和基本原理,以便更深入地分析 HDFS 存储系统的特点.

(1) 集群架构

HDFS 将所存储的文件划分为较大的数据块(data block,如 128MB),并将这些数据块分布式地存储于集群中的各个节点上.如图 1 所示,HDFS 集群中的 NameNode 节点负责管理集群中的元数据.

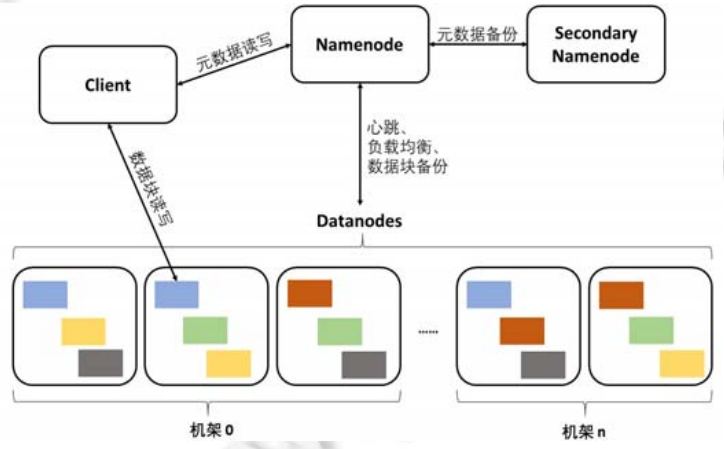

Fig.1 Architecture and data block replacement strategy of HDFS cluster

图 1 HDFS 集群架构和副本放置策略

元数据主要包括文件系统的命名空间和数据块的存储位置等.为防止 NameNode 的单点故障,HDFS 设计了 Secondary NameNode 作为 NameNode 的备份.另外,随着集群规模的扩大和存储的文件数量的增长,元数据的大小也随之增加,单节点的内存难以满足元数据的存储需求,HDFS 引入了 Federation 机制,允许一个 HDFS 集群中存在多个 NameNode,每个 NameNode 分管一部分目录,彼此独立.集群中负责管理节点数据的是 DataNode,每个

DataNode 负责节点本地文件系统中数据块的管理,并定时向 NameNode 发送心跳,反馈本节点状态.为提供存储的容错性,每个数据块都存有一定数量的副本,用户可以自行设置(默认为 3).所有的数据块(图 1 中的彩色方块,相同颜色的方块表示同一数据块及其副本)由 NameNode 决定存储的位置.

(2) 数据块放置与容错

HDFS 中,默认采用机架敏感(rack awareness)的副本放置策略.如图 1 所示,彩色方块代表数据块,相同颜色的色块表示同一数据块的副本.在副本数为 3 的情况下,HDFS 的默认副本放置策略将 3 个副本中的 2 个放在同一机架(rack)中的两个不同 DataNode 上,另外 1 个副本放置在不同机架的 DataNode 上.如果 HDFS 集群跨越多个数据中心,还可以配置更多的副本数,并调整副本放置策略,保证至少 1 个副本存储在不同的数据中心[11].这样的副本放置策略在容错和写入性能方面进行了有效的权衡:一方面保证了数据副本在多节点、多机架甚至多数据中心的冗余,提高了系统容错能力;另一方面,保证了尽可能多的副本优先写入到网络距离较近的节点(如同一节点、机架或数据中心)中,由于网络距离较近的节点间网络带宽较高、传输延迟较低,保证了 HDFS Client 向 HDFS 中写入数据块的性能.在进行集群的扩展、负载均衡调整和恢复丢失的数据块时,HDFS 会自动根据副本放置策略移动副本的存储位置和为恢复的数据块选择新的存储位置,并对上层应用透明[10,11].副本机制基于冗余存储来提供存储的容错,占用了数倍于原始文件大小的存储空间.为减少对存储空间的占用,HDFS 还支持基于纠删码 EC 的容错机制,可以在保证同等可靠性的情况下将存储利用率提高近一倍[12,13].纠删码是在存储空间和计算代价之间的权衡,通过增加编码和解码的计算代价来提高存储空间的利用率,适用于冷数据的存储.目前,HDFS 在同一集群同时支持副本和纠删码两种容错机制,用户可以根据应用需求使用.

1.3 HDFS存储系统的特点和挑战

依据上文对 HDFS 的系统架构和数据存储、容错等基本原理的分析,本节从系统的容错性、扩展性、可移植性和数据读写等方面总结出 HDFS 存储系统的特点,并针对当前应用场景的变化和硬件设备的发展分析 HDFS 存储系统面临的挑战.

HDFS 存储系统的特点包括:

(1) 高容错.HDFS 在设计之初就针对非高可靠的普通廉价服务器,充分考虑了集群的容错性,设计了心跳检测、副本、纠删码和 Secondary NameNode 等机制.基于这些机制,HDFS 能快速发现节点故障,并迅速恢复,在软件层提供高容错和高可靠的数据存储服务;

(2) 高可扩展.HDFS 拥有良好的可扩展性,集群可以扩展到数千甚至上万节点的超大规模,这保证了存储容量的动态扩展,突破了传统数据库系统和数据存储系统的限制,适合存储和管理超大规模的数据;

(3) 高可移植性.HDFS 用 Java 语言开发,屏蔽了底层硬件的细节,可以兼容不同的硬件设备,在不同平台上部署;

(4) 多数据模型.HDFS 中数据基本的组织结构是文件,不同类型的数据都可以抽象为文件来存储,包括文本、关系表、图等.因此,HDFS 可以灵活地支持结构化和非结构化等多种类型的数据存储;

(5) 侧重大文件顺序读写.HDFS 中,大量小文件会增加 NameNode 元数据管理的负担,并且不能很好地利用磁盘顺序读写的 I/O 性能,导致读写吞吐低.过大的文件也会导致数据迁移和恢复的延迟很高.因此,HDFS 一开始经验性地选择了 64MB 作为默认数据块大小,后来调整为 128MB,适合大文件的存储和高吞吐的数据访问;

(6) 侧重一次写入、多次读取的访问模式.HDFS 上的文件一旦被创建和写入完毕后,便不能更新已经写入的内容,只能截断或追加写文件.这种设计符合 HDFS 的设计初衷,即各类数据存储在 HDFS 之上,被不同的应用反复读取,且每个应用任务都会读取大部分甚至是全部的数据.

HDFS 存储系统的特点契合当前大数据存储大容量(volume)、生成快(velocity)和多类型(variety)的需求,在大数据系统,尤其是面向分析的大数据系统中得到了广泛的应用.但随着应用场景的日益多样化和硬件设备的飞速发展,HDFS 存储系统仍然面临着新的挑战,这些挑战主要包括:

(1) HDFS 不支持数据随机写入和修改.在此前提下,支持以随机写入为主的 Key-Value 等 NoSQL 存储系

统具有挑战;

(2) HDFS 主要面向高吞吐数据访问优化,低延迟访问没有足够保障.在此前提下,支持以数据的低延迟访问为主的交互式分析和迭代计算等具有挑战;

(3) HDFS 设计之初是针对传统磁盘优化的,随着新型硬件设备的发展,应用新型硬件设备进一步提高 HDFS 的性能具有挑战;

(4) HDFS 上汇聚的数据和支持的应用负载类型多样,针对不同负载自适应地优化存储性能具有挑战.

1.4 研究框架

结合上文对 HDFS 系统特点和挑战的分析可以看出:HDFS 系统在容错性、扩展性和可移植性等方面已经比较成熟,而数据读写的特点也契合当今大数据分析的需求.因此,当前对 HDFS 的研究重点应该是如何在维持自身特点的前提下克服其性能局限,最大程度地满足不同应用的需求,提高数据存储和访问的性能.这样在生产系统中无需维护多套存储系统和多份数据,降低了存储的成本和数据 ETL(extract-transform-load)的代价,也避免了多份数据之间的一致性问题和时效性问题.

而 HDFS 上数据存储和访问的性能主要受到 3 个维度的影响,即逻辑层面数据在文件内部如何组织、物理层面网络和存储设备的特性以及应用层面用户读写数据的方式.因此,本文从文件逻辑结构、硬件设备和应用场景这 3 个维度对 HDFS 上的存储优化技术进行研究和综述(研究框架如图 2 所示).

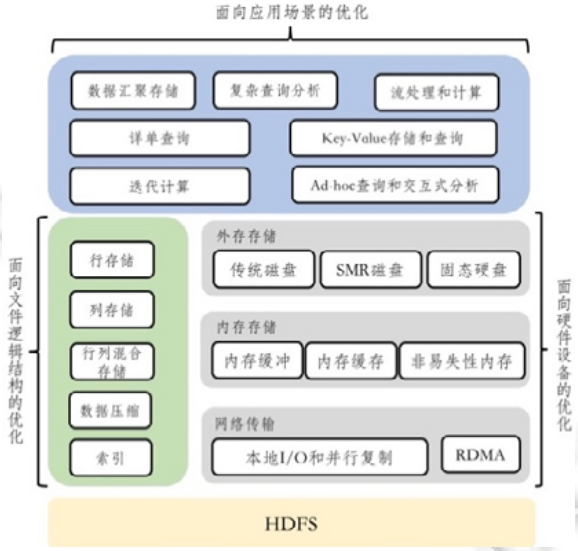

Fig.2 The research framework of storage optimization techniques on HDFS

图 2 HDFS 存储和优化技术研究框架

(1) 文件逻辑结构维度

文件是 HDFS 中数据组织的基本形式,HDFS 上数据的存储和读取都是围绕文件进行的.因此,如何基于文件组织数据,即文件的逻辑结构,会对 HDFS 的数据读写性能产生重大影响.目前,这方面的研究主要围绕文件存储格式、文件压缩方法和数据索引技术这 3 个方面展开,借鉴了很多数据库中成熟的优化设计,如列存储、数据压缩和聚簇索引等.未来,自适应行列混合存储、可直接查询的数据压缩方法以及基于机器学习模型的索引技术可能会是新的研究突破点.

(2) 硬件设备维度

随着硬件的发展,除传统磁盘外,大内存和固态硬盘在企业中的应用越来越多.非易失性内存作为一种新型硬件正受到越来越广泛的关注,并被认为可能对现有存储架构产生颠覆性的影响.在网络方面,RDMA 结合高速网络已经成为许多企业数据中心的标配.HDFS 的底层硬件平台正面临着从传统磁盘和低速网络走向异构存储

结合高速网络的转变.为适应这种变化,本文从内存存储、网络传输和外存存储这 3 个角度分析了现有基于不同硬件的 HDFS 存储和优化技术.当前研究主要集中在大内存、固态硬盘和 RDMA 技术在 HDFS 中的应用上,针对 SMR 磁盘和非易失性内存的研究还比较少,且非易失性内存尚未得到大规模的应用.集成这些新硬件的同时,未来还需进一步研究异构的存储技术,数据按需存储在不同类型的存储设备上,并动态迁移.

(3) 应用场景维度

存储优化技术的根本目的在于服务应用,提高实际应用场景中数据存储和访问的性能.因此,从应用场景的角度出发,更能看出 HDFS 上已有存储和优化技术取得的成果和存在的不足.该维度主要对 HDFS 上各类应用的典型 I/O 特征进行归纳总结,并分析不同 I/O 特征下适用的存储和优化技术,为实际应用和未来研究提供借鉴.当前,HDFS 上最大的不足是对以低延迟读取或实时更新为主的应用场景的支持,需要结合新的存储硬件进一步深入研究.

针对 HDFS 的存储和优化技术随着硬件技术的发展和应用场景的变化而不断发展.本文从文件逻辑结构的角度综述了各种文件存储格式、压缩方法和索引等优化技术,再从硬件特性的角度分析了各项优化技术与硬件特性之间的关系以及针对新型硬件的优化技术.最后,结合对应用场景的分析,从应用的角度对各种 HDFS 存储优化技术的出发点和优劣势进行分析总结.3 个维度的分析紧密结合,且大部分在前两个维度中被分析的工作会同时在应用场景的维度上再加以讨论.这样可以更加深入地理解 HDFS 上的存储和优化技术,为未来学术界和工业界的研究与开发提供参考.

2 面向文件逻辑结构的 HDFS 存储和优化技术

文件是 HDFS 中数据管理的基本逻辑单元,文件的逻辑结构决定了数据在文件内的组织和存储方式,对数据读写性能有着至关重要的影响.这种影响主要从文件存储格式、文件压缩和数据索引这 3 个方面体现.其中,文件存储格式定义了文件内部的数据组织方式,文件压缩决定了文件的物理存储大小,数据索引过滤掉不需要读取的文件内容.在实际应用中,这 3 个方面的优化技术是相互结合的.例如:在一个查询的执行过程中,数据索引对文件和文件中的数据块进行过滤,读取时按照文件格式的定义访问数据,最终读取到压缩后的数据并解压缩得到结果.

当前,HDFS 上流行的文件存储格式包括通用的行存储格式(如 TextFile、XML、JSON、CSV、Sequence File、Map File 和 Avro[14]等)和为关系数据优化的列存储格式(如 RCFile[15]、ORC[16]、Parquet[17]和 CarbonData[18]等).这些文件存储格式各有所长.行存储中的数据在文件内按行连续存储,方便流式追加写入和全量扫描,在 PageRank 等 MapReduce 计算任务中应用广泛.随着 SQL-on-Hadoop 的兴起,传统的行存储已经无法满足基于关系表的大数据分析的需求.此时,列存储打开了一扇新的大门.列存储将同列的数据连续存储,在数据访问时只读取需要的列,这样就避免了不必要的 I/O,并提高了数据压缩的效果.在行存储和列存储的基础上,行列混合存储根据查询负载将频繁一起被访问的列聚成列组,并调整列的物理存储顺序,进一步降低磁盘访问时的跳读代价,提升查询性能.

2.1 文件存储格式

2.1.1 行存储

行存储,也即 NSM(N-aray storage model)存储模型,将数据按行在磁盘页(page)上连续存储,是早期普遍采用的存储模型.HDFS 中的 Text File、XML、JSON、CSV、Sequence File、Map File 和 Avro 等文件存储格式也都采用行存储的思想.这些格式中,数据被水平划分为若干个数据块,每个数据块内按行存储.其中,Text File、XML、JSON 和 CSV 是常见的存储格式.因其格式简单灵活、可读性高,很多数据类型都基于这几种基本格式存储,如图数据存储中的 GraphSON 和 GraphML 分别基于 JSON 和 XML 格式,CSV 格式也在数据分析中被经常用来存储表格数据.除这几种常见格式外,Sequence File 是针对 Map Reduce 实现的二进制存储格式,它将数据按照⟨key,value⟩对的形式序列化到文件中,并且支持数据块级别的压缩.Map File 在 Sequence File 的基础上对数据按照 key 排序,并加入索引,用于快速查找.Avro 文件格式同 Sequence File 类似,都是存储的二进制的序列化内

容.但 Avro 文件格式中采用了更加高效的数据序列化技术,支持多语言的数据读写,并且在文件中包含数据的 schema,可以自描述.

行存储的优势在于数据可以流式追加写入,写入的延迟较低,且格式简单,易于编写 Map Reduce 程序进行处理.其缺点在于如果只需要访问行内的一小部分数据,也需要将整行数据读入内存,浪费了磁盘 I/O.因此,面向行的存储适合于按行扫描的情况.在上述行存储格式中,Text File、XML、JSON 和 CSV 都是常用的数据存储和交换格式,Sequence File 和 Map File 是针对 MapReduce 设计的存储格式,Avro 是面向复杂对象存储的数据格式.

2.1.2 列存储

列存储技术最早在面向分析的数据库系统中广泛应用.早期的关系数据库是面向写优化的,普遍采用 NSM(N-ary storage model)存储模型.直到 1985 年,Copeland 提出了 DSM 存储模型[19],将关系表按列划分,每列单独连续存储.DSM 证明了对于分析查询,列存储能帮助查询获得明显的性能提升.基于这个思想,Stonebreaker 等人设计和实现了 C-store[20],对后续研究和应用产生了较大影响.

列存储将同列的数据连续存储,在数据访问时只读取需要的列,避免了不必要的 I/O,并提高了数据压缩的效果,最终提高查询的 I/O 性能[21].当 HDFS 上 SQL 查询分析应用(尤其是 SQL-on-Hadoop 系统)的分析性能达到瓶颈后,列存储被很快从数据库中借鉴过来解决数据访问的性能问题.但按照 DSM 的思想在 HDFS 上将数据按列单独连续存储为文件或数据块,会导致各列数据被分散到集群的不同节点上,在恢复记录时,需要从不同节点读取多个数据块,造成额外的网络开销和计算开销,并且影响计算的并行度.

针对这个问题,HDFS 上流行的列存储文件格式 RCFile、ORC、Parquet 和 CarbonData 等借鉴了内存存储格式 PAX 存储模型[22]的思想,采用了如图 3 所示的方式实现列存储[23,24].数据文件被水平划分为行组(row group),行组内数据按列连续存储为列块(column chunk).一个列块中存储了行组内一个列上的所有数据项.一个列存储格式文件中包含一个或多个 HDFS 数据块,一个数据块中包含一个或多个行组(通常,一个数据文件只包含一个数据块,一个数据块只包含一个行组为最优配置[16−18]).这样的列存储格式保证了数据表中所有数据列存储在同一个数据块中,避免了跨节点的数据连接.在行组内部采用列存储,仍然可以发挥列存储在 I/O 和数据压缩方面的优势.同时,行组作为数据处理的并发单元,不影响 HDFS 环境下数据处理的并发度.因此,RCFile、ORC、Parquet 和 CarbonData 等文件格式在工业界得到了广泛应用.

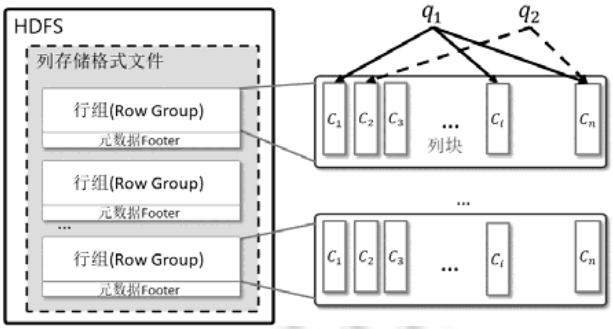

Fig.3 Standard columnar data storage layout on HDFS

图 3 HDFS 上的标准列存储布局

RCFile 是较早的 HDFS 列存储格式,一开始应用在 Facebook Hive[25]和 Yahoo Pig[26]中.它的默认行组大小只有 4MB,一个数据块中包含多个行组.每个行组中有一个元数据区域存储行组中各个列的位置偏移.

RCFile 支持对行组中的每列数据分别应用 RLE(run-length encoding)编码和 Gzip[27]压缩,并在查询时支持 Lazy Decompression,即压缩的数据在读入内存后不会立即全部被解压,而是被真正访问时才被解压.这种优化可以利用查询中的过滤条件减少不必要的数据解压,降低 CPU 和内存资源的消耗.

ORC 在 RCFile 的基础上进行了大量优化.

- 首先,ORC 在行组的元信息中添加了对统计信息和布隆过滤(Bloom filter)等的支持,并加入了谓词下推(predicate pushdown)的功能,这些优化对数据过滤起到了很好的效果;
- 其次,ORC 支持了字典编码、RLE 编码、位压缩(big packing)等多种编码方式,并且可以根据数据类型和列上的统计信息自动采用不同的压缩编码方式,提升了数据编码的效果;
- ORC 还支持将嵌套数据类型(如 Json,Map)分解为数据列存储在二维数据表中[16];
- ORC 中还有一个优化的细节增大了默认行组的大小,将其从 RCFile 的 4MB 提高到 256MB(Hive 中默认为 64MB),这个优化提高了 I/O 的连续性[23];
- 最后,ORC 能支持文件内记录的更新和删除.

Parquet 的数据组织方式和 ORC 类似,默认采用 128MB 的行组大小[17],并且同样支持根据列的数据类型和统计信息对数据进行自动的编码压缩.不同的是,Parquet 默认支持的数据模型就是嵌套结构的,它借鉴了 Google Dremel[28]中的嵌套数据分解技术对数据进行编码存储,从而实现高效的嵌套数据的存储和访问.

CarbonData 是 2016 年开源的一种列存储格式,并已成为 Apache 开源项目.CarbonData 中除了支持 ORC 和 Parquet 中的数据编码压缩和嵌套数据类型之外,还将列块划分为页面(默认一个页面包含 32 000 个数据项),在页面范围内对各个列上的数据分别排序,大幅提高数据压缩效果.由于页面是较小的存储单元,各个列上相对应的一组页面在查询执行时可以快速地在内存中进行连接、重建为元组.此外,CarbonData 提供了多种索引的支持,包括倒排索引和多维键值(multi dimensional key,简称 MDK)索引.CarbonData 还支持全局字典,同一个数据表内使用同一个字典对数据进行编码压缩,这样可以直接在编码后的数据上执行过滤和分组等计算,仅对最终的查询结果进行解压,无需对全部数据解压缩.

ORC、Parquet 和 CarbonData 是目前工业界普遍应用的 Apache 开源列存储格式,其中,ORC 和 Parquet 开源较早,发展比较成熟,在大部分应用场景中都不分伯仲,且大部分查询引擎,如 Spark SQL[29]、Presto[30]和 Hive[25]等都对两种存储格式有很好的支持.CarbonData 开源较晚,相比于 ORC 和 Parquet,加入了大量索引和复杂数据编码的支持,更适合对延迟较低的交互式查询的需求,目前支持 Hive、Presto 和 Spark SQL 作为查询引擎,并结合 SparkSQL 进行了深度优化.这几种流行的开源列存储格式的对比见表 2.

Table 2 Comparisons of popular open source columnar storage formats on HDFS

表 2 HDFS 上主要的开源列存储格式对比

存储格式	数据编码和压缩	数据索引	更新和删除	查询引擎支持
ORC	字典编码、RLE 编码、位压缩、zlib、Snappy[32]	统计信息和布隆过滤	支持	Pig、Hive、Presto、Spark SQL
Parquet	字典编码、RLE 编码、位压缩、GZip、LZO[33]、Snappy[32]	统计信息	不支持	Hive、Presto、Spark SQL、Impala[31]
CarbonData	全局字典编码、RLE 编码、位压缩、Snappy[32]、GZip	统计信息、聚簇索引、倒排索引、多维键值索引	支持	Hive、Presto、Spark SQL

除了以上的开源列存储格式,CIF[34]还提出了一个类似 DSM 的存储格式,即:将数据表中各个列分别存储在不同的数据块中,并通过修改 HDFS 的数据块放置策略将这些数据块存放在相同的 Datanode 上.对比 RCFile 在默认行组大小(4MB)下的性能,CIF 有明显的性能提升.但在后来的研究工作中,文献[23]证明:当 HDFS 列存储格式中采用了较大的行组之后,I/O 连续性得到了明显的提升,相比之下,CIF 没有性能收益.而且改变数据块放置策略需要对 HDFS 进行修改,工程实施难度较大,容易破坏 HDFS 的容错性和负载均衡.因此,现有 HDFS 上的主流存储格式都还是基于 PAX 的思想.

2.1.3 行列混合存储

通用的列存储格式适合大部分分析型负载,但在一些应用中,可以根据查询的访问特征进一步优化列存储的内部布局.文献[35,36]中提出:将频繁一起访问的列聚成列组,每个列组连续存储,构成行列混合存储.

文献[36]中采用枚举算法对列的所有组合进行枚举,从中选出适合当前负载的列组组合,每个 HDFS 文件副本都采用不同的列组组合,在查询执行时,从中选择读取代价最低的副本读取.并且为降低元组恢复(tuple

reconstruction)的代价,将列组中的数据按行存储.但该方案有 3 个缺点.

1) 需要修改 HDFS 的内部设计和实现,破坏了系统的通用性;
2) 枚举算法的计算复杂度随列个数的增加而呈指数级增长,只适合列个数较少的场景;
3) 元组恢复的代价在文献[23]中被证实并不是列存储中的主要性能开销,并且列组中的数据按行存储在 C-Store[21]中被证实不如列式存储更优.

列式存储可以避免读取列组中的部分列时的不必要 I/O,另外,文献[24]也表明,将列聚成列组的主要优势在于当一个列组中的列被连续访问时减少了磁盘跳读的代价.因此在实际的行列混合存储中,很少将列组内的数据按行存储.

针对上述问题,文献[24]提出了基于启发式算法的列排序的方法取代列组组合.相比于列组组合,列排序在提高 HDFS 列存储系统数据读取性能的前提下实现起来更加简单,且无需对现有的 HDFS 列存储系统作任何修改.通过分析查询负载对数据列的访问记录,文献[24]将列组的优化问题定义为列的物理存储顺序的排序问题,并证明该问题为 NP-Hard.实验结果表明:列排序技术在宽表(几百上千列)的场景下可以将查询性能提高 1 倍左右,比列组组合的效果更好.

2.2 文件压缩技术

除了在列存储格式中使用的数据编码算法之外,HDFS 上的数据可以采用一些无损压缩算法对文件进行进一步压缩,如 Snappy、LZO、Gzip、bzip2[37]等.数据压缩不仅可以节约存储,还可以提高数据处理的性能.HDFS 环境下的计算任务还可以对计算中间结果压缩后再写入磁盘,从而大幅度降低了中间结果的大小,提高了数据处理的性能[38].不同压缩算法之间的对比见表 3.

Table 3 Comparisons of common compression methods on HDFS

表 3 HDFS 中常用压缩算法的对比

压缩算法	是否可拆分	压缩比	压缩速率	解压速率	适应场景
Snappy	否	高	高	高	列存储格式等特定文件格式
LZO	是	中	高	高	文本等需要拆分解压的文件格式
Gzip	否	高	低	低	存储空间有限,解压速率要求不高(如冷数据)
bzip2	是	高	低	低	存储空间有限,解压速率要求不高(如冷数据)

HDFS 中常用的数据压缩算法分为可拆分(splittable)和不可拆分(non-splittable)两类.Snappy 具有高压缩速度和较好的压缩率,它在速度和压缩率之间作了较好的权衡[38].由于 Snappy 是不可拆分算法,即压缩后的文件不可拆分,它需要在一个特定的文件格式(如 Parquet,ORC)中使用.LZO 和 Snappy 类似,比较注重压缩速度.不同的是,LZO 压缩后的文件是可拆分的,因此相对于 Snappy,LZO 更适合用作一个独立的压缩格式来对 HDFS 上的文本格式的文件进行压缩.Gzip 提供了较高的压缩性能,平均达到 Snappy 的 2.5 倍,但其写入性能不如 Snappy.在读性能方面,Gzip 和 Snappy 接近.Gzip 同样是不可拆分算法,因此也需要嵌入在一个文件格式中使用.在部分情况下,Gzip 的压缩效果太好,导致压缩出的数据很小、数据块数很少,所以在执行数据处理任务时的并行度可能会偏低,从而导致数据处理的速度反而降低[38].这个问题可以通过使用较小的数据块来避免.bzip2 提供了非常好的压缩性能,但是其解压性能较差,通常只用于存储空间非常有限的情况.

2.3 数据索引技术

索引能够帮助查询快速定位要读取的数据,对于降低查询执行的延迟非常关键.但对于分布式的环境和海量的数据,传统数据库中的索引技术很难全都直接应用在 HDFS 上.目前,HDFS 上普遍采用的索引技术包括聚簇存储、Bitmap 和统计信息等,这些索引占用很小的存储空间,且维护简单.

Hive 中支持将数据表根据一个或多个属性进行分桶(bucket),如对数据按照生成日期的年、月、日划分,查询部分日期的数据可以直接利用 Bucket 过滤掉不相关的数据,相当于按照数据生成日期建立聚簇索引[39].此外,Hive 还支持利用 Bitmap 索引对数据进行过滤[39].

HAIL[40,41]提出在 HDFS 上利用数据块多副本,在每个副本上按照不同的属性对数据排序、建立聚簇索引,

从而利用多副本容错机制支持多个不同的聚簇存储,在查询时,根据查询条件选择合适的聚簇存储.这种方案利用了多副本机制来建立多个聚簇索引,可以应对不同的查询需求,但索引建立的开销也增加了,并且需要对HDFS 的内部设计和实现进行较大的改动,在实际应用中并不提倡.针对建立索引影响数据落地实时性的问题,文献[41]中提出了 Lazy Indexing 的方法,即:数据写入时不建立索引,而是在查询读取数据的同时动态建立索引,并在查询执行完成后将索引刷写到 HDFS 上.这种方式牺牲了数据首次查询的性能,换取了数据落地的实时性.

CarbonData[18]是 HDFS 上的新型列存储格式.CarbonData 中除了采用页面内数据按主键排序以提高压缩效果之外,还支持多维主键上的聚簇索引(CarbonData 中的 MDK,即 multi dimension keys),即多个维度的数据拼接成主键.与 HAIL 中的数据块内聚簇不同,CarbonData 采用数据段(segment)粒度的聚簇索引.向数据表中加载一次数据即形成一个数据段,CarbonData 对一个数据段内的数据按照主键排序后存储到 HDFS 中,并将各个数据块中的统计信息集中存储在元数据中.这样,在查询时可以通过集中存储的元数据快速进行数据块过滤、命中相关的数据块和块内数据页.但是数据段粒度的聚簇索引会对数据导入的性能有一定的影响.

文献[42,43]提出在聚簇索引的基础上,还可以在数据块内采用非聚簇索引,支持其他属性上的元组过滤.文献[43]中还提出了基于动态一致性哈希实现数据的高吞吐实时划分、索引和入库.文献[44]提出在 ORC 的基础上增加冗余的数据列支持聚簇索引,通过少量的存储空间开销实现索引属性上数据的快速查找.除显式的索引外,ORC、Parquet、CarbonData 等列存储格式在元数据中记录各个行组上的统计信息,查询可以根据统计信息过滤掉不相关的行组,这种 Data Skipping 方式可以看作是一种隐式索引.文献[42]中提出了一种细粒度的 Data Skipping 技术,通过提取历史查询中的数据过滤条件,将数据元组和过滤条件建立映射,为每个数据元组建立特征向量,并根据特征向量对元组进行聚集.这种方法可以将查询中被频繁共同访问的元组聚集到同一个数据块中,从而使得查询可以直接命中相关数据块,避免读取不必要的数据.实验结果表明,该方法在查询性能上比基于范围聚簇的数据块组织方式高 2~5 倍.但这种方法在大规模的数据集上建立特征向量的时间开销太大,且针对历史查询的优化改变了数据的聚集方式,可能会给新出现的查询带来负面影响.

2.4 小 结

本节从文件存储格式、文件压缩方法和数据索引技术这 3 个方面介绍了面向文件逻辑结构的 HDFS 存储和优化技术.目前,基于行存储的文件格式最多、适应面最广,涵盖了文本、关系表、图结构等多种数据类型,已经发展成熟,后续优化空间不大.面向关系表的列存储研究较多,且取得了很好的成果.ORC、Parquet 和 CarbonData 等几种常见的存储格式都得到了广泛应用,给查询带来了很大的性能提升.行列混合存储会是未来面向关系表的存储格式的主要发展方向,这方面目前研究较少,其中,结合应用负载的自适应优化仍是一个难题.文件压缩技术基本上是直接应用已有的成熟算法,没有大的改进和突破,但为了减少反序列化的开销和内存占用,可直接查询的编码和压缩技术在 HDFS 上的应用值得探讨,如 succinct 数据结构[45]等.数据索引技术当前的趋势主要是文件内部的数据索引和全局数据字典等,CarbonData 中引入了数据库中常用的聚簇和倒排等索引技术,未来可以探索更多数据库中索引技术在 HDFS 上的应用.另外,Kraska 等人提出 learned index structure[46]利用机器学习技术对数据建模替代传统索引,可以进一步在 HDFS 上进行尝试.

3 面向硬件设备的 HDFS 存储和优化技术

硬件是存储系统的基础,不同硬件具有不同的特性,需要不同的优化技术最大化地利用这些特性来提高 HDFS 的性能.本节从外存存储、内存存储和网络传输这 3 个层级,分析不同硬件设备的特点和基于这些硬件设备的存储及优化技术.

3.1 外存存储

HDFS 在设计之初是将传统磁盘作为主要存储介质.然而随着存储硬件的发展,SMR 磁盘和固态硬盘已经被广泛应用,HDFS 也开始集成多种不同的外存设备.本节介绍针对 HDFS 中不同外存设备的典型优化技术(见表 4).

Table 4 Storage and optimization techniques for HDFS based on external storage devices

表 4 基于外存存储设备的 HDFS 存储和优化技术

存储设备	设备原理	设备优点	设备缺点	优化技术
传统磁盘	依赖磁盘自身的旋转和磁头在磁盘上的移动来读写数据	存储密度高,寿命长,价格便宜	IOPS 低,随机读写性能差	缓存机制(见第 3.2 节) 文件逻辑结构优化(见第 2 节) 多磁盘并行读写[47] 磁盘服务线程[48] LSM-Tree[49−51]
SMR 磁盘	磁盘的相邻磁道相互重叠	存储密度非常高,寿命长,价格便宜	写放大严重,随机写性能很差	Archival Storage[8] ShingledFS[56] ManyLogs[57] SMR 内缓存和 S-block[58] 适合 SMR 的 STL 和 GC[59]
固态硬盘	Flash 存储阵列	高带宽和高吞吐,随机访问和顺序访问性能差别较小	有擦除次数的限制,寿命较短,价格较高	HDFS 异构存储[6] Alluxio[62,63] 固态硬盘的数据块放置策略[64−67] 高速网络[68−70] In-storage Computing[61,71,72] HDFS 数据块放置策略[70]

3.1.1 传统磁盘

传统磁盘依赖于磁盘自身的旋转和磁头在磁盘上的移动来读写数据,随机读写时需要频繁进行磁盘寻道,磁头在磁盘上来回移动,读写的性能差.因此,传统磁盘更适合顺序读写的操作.HDFS 的设计最初建立在传统磁盘上,对 HDFS 的存储优化都是从减少磁盘 I/O 和提高磁盘 I/O 效率两个方面进行的.

缓存机制在减少磁盘 I/O 方面有很好的效果,HDFS 及其上的计算框架和查询引擎都设计了不同的缓存机制(在第 3.2 节给出详细讨论).另外,第 2 节中讨论的列存储、数据压缩和数据索引都是减少磁盘 I/O 的常用方法.列存储可以避免读取不必要的数据列,数据块内部可以利用统计信息和索引进行过滤,并利用对数据的高效编码和压缩效果来减少数据分析时的磁盘 I/O[21,22].

在优化磁盘 I/O 效率方面,因为磁盘的随机跳读代价高,所以主要是减少磁盘的随机 I/O,提高 I/O 连续性.第 2 节中讨论的自适应列存储通过分析查询负载和数据列的访问特征,将被频繁一起访问的列在物理上相近地存储,提高了列存储的 I/O 连续性.另外,HDFS 支持将数据块均衡地存储在本地的多个数据目录下[47],每个目录可以挂载一块单独的磁盘,从而支持单机多磁盘的并行读写,提高数据读写的总带宽和吞吐量.但 HDFS 没有对磁盘上的读写操作进行细粒度的调度.由于磁盘是串行 I/O 设备,随机读写性能差,在大规模并行计算中可能发生大量读取进程或线程对磁盘的过度争用.因此,文献[48]提出在 Datanode 中为每一块本地磁盘启动一个服务线程,负责该磁盘上的读写操作,从而将磁盘上并发的读写变成串行的读写,减少磁盘随机跳转(seek)和磁盘调度的开销.但这种方式对 HDFS 系统的侵入较大,并未得到普遍采用,主要还是在应用层将大量小的随机读写尽可能合并为大的顺序读写.LSM-Tree(log-structured merge-tree)[49]便是采用这种思想的典型存储结构.LSM-Tree 在磁盘上采用 Copy-on-Write 的策略,将对文件上的随机写操作变为顺序的追加写入,之后再通过批量合并操作(合并操作本身也是顺序读写文件)消除数据冗余,极大地提高了数据的写入吞吐量.这种结构在 Key-Value 数据库中的应用十分广泛,BigTable[50]和 HBase[51]中也利用 LSM-Tree 优化数据的写入.除了随机的小 I/O 之外,HDFS 上可能存储了很多小文件,这些小文件降低了 I/O 的连续性,且增加了 NameNode 元数据管理的负担,因此,HDFS 中提供了 file archive[52]的功能自动合并小文件.

3.1.2 SMR 磁盘

SMR(shingled magnetic recording)磁盘是一种基于传统磁盘的存储密度更高的硬件,它对传统磁盘的结构进行了一个巧妙的调整,将磁盘的相邻磁道互相重叠来获得更高的存储密度[53].这种设计在提高磁盘容量的同时,使得在一个磁道上的写入会覆盖相邻磁道的数据.因此,SMR 磁盘上随机的数据写入会带来严重的写放大问题[54].SMR 磁盘在提高存储容量、降低单位存储价格的同时牺牲了随机写入的性能,适合批量顺序写入而随机

更新很少的应用场景[55],如企业的历史数据存储.现有的基于 SMR 磁盘的存储和优化技术主要包括两个方面:一是将 SMR 磁盘应用到 HDFS 中,二是进一步优化基于 SMR 磁盘的读写性能.

在 SMR 磁盘的应用方面,随着 Hadoop 集群存储数据量的不断增加,海量数据堆积在 HDFS 集群中,其中大部分是很少被访问的历史数据.为了降低历史数据存储的成本,Hadoop 引入了 Archival Storage[8]功能.该功能使得上层应用可以通过制定存储策略,将访问频次很低或不再被访问的文件存储在存储密度高、价格低廉的 Archive 类型的存储设备上.SMR 磁盘就是一种非常适合 HDFS 中 Archive 类型的存储设备,在 HDFS 中使用它能够减少历史数据对优质存储资源的占用,降低储存成本.

在针对 SMR 磁盘的优化方面,文献[56]针对 SMR 磁盘实现了 SMR 设备仿真器和对 SMR 感知的本地文件系统 ShingledFS.它优化了文件系统的写 I/O 来最大程度地发挥顺序写的性能,并提出了适应 SMR 磁盘特点的垃圾回收算法.ShingledFS 是基于 FUSE 实现的对叠瓦式磁盘感知的文件系统,可以应用到 HDFS 集群中.但该工作主要是验证 SMR 磁盘可以有效支持大数据负载,在 Hadoop 集群中作为高容量低成本的存储设备,并未对垃圾回收算法等进行深入研究,不能应用于对性能要求很高的场景.为进一步优化基于 SMR 磁盘的数据读写性能,ManyLogs[57]针对叠瓦式磁盘随机写性能差的特点,提出一种新的文件系统的日志记录方法,将许多小的随机日志写入转变为顺序写入.文献[58]提出基于磁盘内缓存和新的存储单元 S-block 的方式优化随机写入.文献[59]提出了适合叠瓦式磁盘的 STL(shingled translation layer)和 GC(garbage collection)方法.这些优化有助于提高基于叠瓦式磁盘的文件系统的读写性能,使叠瓦式磁盘可以更好地应用在 HDFS 中.

3.1.3 固态硬盘

随着固态硬盘(solid state disk,简称 SSD)容量的增加和成本的降低,越来越多的企业选用固态硬盘作为存储设备来加速数据访问.现代的固态硬盘一般由 Flash 阵列(flash memory array)持久化存储数据,DRAM 芯片配合 Flash 阵列缓存数据,并且内部还有一个作为控制器的处理芯片实现了接口协议和地址转换(flash translation layer,简称 FTL)等功能[60,61].相比于磁盘,SSD 不依赖于磁头的机械运动来读写数据,拥有更高的带宽和 IOPS.

除了磁盘以外,HDFS 可以兼容包括固态硬盘和 DRAM 在内的不同类型的存储设备,形成异构存储[6].其中,DataNode 可以将本地存储设备的类型和使用情况暴露给 NameNode,这样,NameNode 可以动态感知每个节点的存储类型.上层应用可以为存储的文件指定存储类型的偏好,如将文件优先存储在固态硬盘上,还可以设置存储策略,将一个或者所有副本存储在固态硬盘上.这样,上层应用可以利用固态硬盘存储一个副本作为磁盘的缓存或者存储所有副本来加速数据的读写.

除了 HDFS 的内置支持外,Hadoop 生态系统中常用的分布式缓存系统 Alluxio[62]也支持将固态硬盘作为层级化存储(内存、固态硬盘和磁盘这 3 种层级)的一部分.Alluxio 可以根据用户指定的策略将数据存储在合适的层级中.固态硬盘作为内存和磁盘之间的缓存,使得 Alluxio 可以在突破内存容量限制的同时,减少磁盘 I/O 带来的性能损失.对于频繁访问的较大的数据,应用也可以指定将它们缓存在固态硬盘中,提高 Spark 等上层应用的数据访问效率[63].

另外,hats[64]修改了 HDFS 原有的数据放置策略,将固态硬盘作为磁盘的上层存储,数据的副本存储到不同层级中.修改后,文件先在 SSD 中存储,再复制到磁盘中,读取时则优先从最上层读取.这种放置策略提高了数据写入的性能,但会造成数据访问的不均衡.类似地,Triple-H[65]提出基于 HPC(high performance computing)集群的 HDFS 文件放置策略,以利用高性能节点的内存和固态硬盘.hats 和 Triple-H 都是将文件的多个副本放置不同层级的存储设备中,没有结合文件的访问特征和设备底层的 I/O 特性进行动态优化,难以充分发挥固态硬盘的性能优势.文献[66]和 Venu[67]根据应用负载的访问特征预测未来的热点数据,自动地将“热”的数据存储到固态硬盘中,将“冷”的数据迁移到磁盘中.

为了进一步优化固态硬盘在 HDFS 中的使用,文献[68,69]发现,配置高速网络可以进一步发挥固态硬盘在 HDFS 中的性能优势.基于传统磁盘的 HDFS 集群在并发读写较高的时候性能会急剧下降[45],而固态硬盘在并发读写较高的情况下会获得更好的性能[68].文献[70]提出基于 Linux 异步 I/O 的精确预读取模型,更好地利用固态硬盘的 I/O 性能,能够带来 18%的性能提升.

除了作为比传统磁盘更高效的存储设备,固态硬盘还可以作为计算设备,即,利用 SSD 控制器中嵌入的低能耗处理器对本地数据进行简单计算(in-storage computing,简称 ISC).ISC 的思想是移动计算使之更加靠近数据,这种方式能提高计算性能和降低数据移动带来的能量消耗.Samsung 提出了 Smart SSD 模型[71],将 CPU 处理器和 DRAM 存储包装到 SSD 中,使得该 SSD 设备可以运行自定义的用户程序.文献[61]将 SQL Server 中关系查询的一些操作下推到 SSD 中执行,大幅度提高了查询的执行性能和降低了能量消耗.文献[72]利用固态硬盘的计算能力实现了适应 ISC 的 MapReduce 框架,将 Mapper 任务下推到固态硬盘中执行,这种设计给 MapReduce 带来了 2.3 倍的性能提升,并大幅降低了计算的能源消耗.

不同于传统磁盘,固态硬盘的数据块的擦除次数是有限制的,普通固态硬盘的擦除次数在 3000~5000 之间[73].为了尽可能延长 HDFS 中固态硬盘的使用寿命,文献[70]修改了 NameNode 中的数据块放置策略,在放置数据块时尽量平衡 DataNode 节点上固态硬盘的使用.

3.2 内存存储

HDFS 通常将数据块存储在磁盘文件系统中.由于磁盘的带宽和 IOPS 都较低,在大规模数据处理中通常利用内存缓冲(buffer)来将随机读写转换为顺序读写,以及利用内存缓存(in-memory cache)来弥补 CPU 和磁盘之间的速度差异.另外,非易失内存的出现也为 HDFS 的优化提供了新的方向.表 5 列出了典型的基于内存存储的 HDFS 存储和优化技术.

Table 5 Storage and optimization techniques for HDFS based on memory storage

表 5 基于内存存储的 HDFS 存储和优化技术

内存存储	存储设备	存储原理	优化技术
内存缓冲和缓存	DRAM	利用内存缓存数据,加速数据访问和将随机读写转化为顺序读写	LSM-Tree[49–51], HDFS CCM[7], Page Cache[74,75], Tachyon/Alluxio[62,76], Spark RDD[29,78], Flink DataFlow[79], Presto, Impala, Drill[81]
非易失性内存	STT-RAM、PCM、RRAM	性能介于 DRAM 和固态硬盘之间,支持以字节为单位的数据访问和持久化的数据存储	NVMFS[84]

3.2.1 内存缓冲和缓存

HDFS 上的 Key-Value 存储将内存作为数据的读写缓冲,将随机读写转化为顺序读写.HBase 作为 BigTable 的开源实现,采用 LSM-Tree 来解决磁盘 IOPS 低、随机写入延迟大的问题.LSM-Tree 将数据在内存和磁盘上分层存储,内存中缓存最近写入和修改的数据,然后批量写出到磁盘或固态硬盘.这种方式提高了写吞吐,但数据查询时需要对多层数据的查找结果进行汇总才能得到最终结果.为了保证数据查询的性能和控制写放大,LSM-Tree 对内存和磁盘上的数据进行批量归并(merge),从而在读写性能之间做出比较好的权衡.

除了利于加速数据写入外,内存更主要的是缓存文件来加速应用的数据访问性能.HDFS 从 2.3.0 版本开始增加了集中式缓存管理(centralized cache management,简称 CCM)功能[7].这是 HDFS 自身提供的一种显式缓存机制,允许用户指定 HDFS 中的一个存储路径并将其中的文件缓存在内存中.对于要缓存的文件,Namenode 通知存储该文件数据块的 Datanode 将数据块缓存在进程 JVM 的堆外内存中.堆外内存不受 Java 虚拟机垃圾回收机制的管理,避免了额外的垃圾回收开销.在集中式缓存管理功能出现之前,HDFS 依靠 Datanode 本地操作系统的 Page Cache 来进行文件内容的缓存.操作系统 Page Cache 不掌握文件系统上层应用的数据访问模式,只根据规则进行简单的预读和缓存置换[74,75].因此,需频繁访问的文件页面很容易因为其他文件上的大量顺序读操作而被驱逐出 Page Cache,不仅导致缓存命中率低,而且导致大量不必要的页面置换,浪费了 I/O 和计算资源.另外,由于 Page Cache 在操作系统内核空间中,当 Datanode 进程读取 Page Cache 中的内容时,需要进行一次内存复制,将 Page Cache 中的页面复制到 JVM 的进程空间中.对于频繁访问数据的迭代计算,内存复制消耗大量的 CPU 机器周期,影响计算性能.

相对于操作系统的 Page Cache,集中式缓存管理可以发挥分布式内存的优势.缓存由 Namenode 统一管理,同一数据块的副本只有部分会被缓存,避免多个副本都被读入操作系统 Page Cache,浪费内存空间.数据访问时,HDFS 上的应用可以查询到被缓存的数据块的位置,并将任务调度到缓存所在的节点上,提高数据访问的性能.另外,集中式缓存管理还实现了内存的零复制(zero-copy),降低了内存复制带来的 CPU 周期消耗.但 HDFS 的集中式缓存管理目前仅支持文件和目录级别的缓存,不支持数据块或用户自定义的缓存粒度,当文件较大而应用仅需访问文件中的部分数据时,缓存的效果并不明显,且被缓存的文件需要应用程序在运行时指定,不能根据查询负载动态选择.

除了 HDFS 内置的集中式缓存管理外,Tachyon[76]及其后续版本 Alluxio[62]也为 HDFS 上的应用提供了通用的分布式缓存管理,缓存的数据可以被不同应用共享.Tachyon/Alluxio 提供与 HDFS 兼容的文件访问接口,采用 Lineage 机制来支持数据快速写入和容错.计算任务将数据转换的结果写入一个新的文件,一系列计算过程中输出的文件就构成了 Lineage Graph.发生错误时,通过检查点和重计算来进行恢复.相对于基于副本的容错技术,基于 Lineage 的容错不需要在节点之间通过网络进行副本复制,实验结果表明:Lineage 的数据写入速度比内存文件系统上的 HDFS 还高 110 倍[76],从而保证了分布式缓存的写入性能.Alluxio 也支持磁盘、SSD、内存多级存储,默认根据 LRU 策略将内存中的数据淘汰到外存.另外,应用程序可以指定数据存储的介质,将频繁访问的数据缓存在 SSD 或内存中[62].

除了通用的文件缓存外,Apache Arrow[77]提供了针对列存储的内存缓存,支持不同系统之间共享数据,减少数据在不同系统之间移动的序列化和反序列化开销.此外,HDFS 上的一些计算框架和查询引擎也根据自身需要设计了专用的分布式内存管理机制.RDD(resilient distributed datasets)[78]是 Spark 中的分布式内存数据结构,分为若干个 Partition,分布式地存储在集群各个节点上,用于加速 Spark 中的迭代和交互式计算.每个 RDD 都是静态的(immutable),对 RDD 的转换操作都将产生新的 RDD.HDFS 上的文件可以作为 RDD 的数据输入和数据输出,Spark 计算的中间结果则以 RDD 的形式存储在内存中.RDD 的容错主要依赖于 Lineage 机制和计算过程中的检查点(checkpoint).尽管创建 RDD 需要进行大量的内存复制,RDD 的访问性能仍然远高于存储在磁盘上的 HDFS 文件.加之 RDD 与 HDFS 一样支持容错,而且 Spark 采用 Lazy Execution[29],仅在数据转换操作执行时才动态创建 RDD,一定程度上避免了不必要的内存开销,因此在大规模的迭代和交互式计算中具有很大的优势.

Flink[79]面向有状态的流处理,采用 DataStream 和 DataSet 作为抽象的内存数据结构.两种抽象数据结构在 Flink 运行环境中都通过 Dataflow Graph 实现,即:将一系列数据转换操作形成流水线,数据一边产生一边被消费,提高了内存利用率,缓解了因为内存不足而向磁盘刷写中间结果导致的存储性能下降问题.与 Spark 中的 RDD 不同, Flink 中的 DataFlow 并不是静态的,因此不需要进行频繁的内存复制.另外,Flink 采用异步 Snapshot 作为容错的方式[80],平衡了同步 Snapshot 导致的全局阻塞和 Lineage 机制的重新计算.类似 HDFS 的集中式缓存管理,Flink 运行环境将 Dataflow 中的数据序列化后存储在 JVM 堆外的内存段(memory segments)中,减少了 JVM 垃圾回收造成的开销.Flink 还支持直接在不反序列化的堆外内存段数据上执行排序和连接操作,降低了数据反序列化造成的额外开销.

HDFS 环境下的其他大数据分析引擎,如 Presto、Impala、Drill[81]也采用基于内存的流水线式数据缓冲区,将计算任务编译成有向无环图(DAG),在内存中不断产生和消费数据.这种方式避免了计算中间结果落地 HDFS,缓解磁盘性能瓶颈,为 HDFS 上的交互式数据分析提供了支持.此外,ElasticSearch[82]中也利用内存缓存和索引数据,提供实时的数据检索.

3.2.2 非易失性内存

非易失内存(nonvolatile memory,简称 NVM)作为一种新型存储受到了广泛的关注,它主要包括自旋矩传输磁存储器(spin-torque transfer RAM,简称 STT-RAM)、相变存储器(phase-change memory,简称 PCM)和电阻式存储器(resistive random access memory,简称 RRAM)等具有非易失性特点的存储设备[83],可以类似 DRAM 一样按字节寻址(byte-addressability),又不会在断电后丢失数据,因此被认为是一种具有广泛应用前景的新型存储[84].目前,市场上最新的非易失性内存产品是 Intel 的 Optane Apache Pass 内存条,该内存条基于 3D Xpoint 技术,单

条最大容量为 512GB.NVM 有着比磁盘和固态硬盘更高的读写性能,同时还能持久化地存储数据,因此,最近几年开始出现将 NVM 集成到 HDFS 中的研究工作.

文献[85]提出了 NVFS(NVM- and RDMA-aware HDFS),它基于现有的 HDFS,加入了对 NVM 和 RDMA 的支持.NVFS 中 NVM 提供两种访问模式:块访问和内存访问,分别对应 NVFS-BlkIO 和 NVFS-MemIO 接口.块访问模式下,NVM 可以作为块设备被加载到 DataNode 的本地文件系统中,文件系统的 I/O 操作调用 NVMe 接口通过底层的 NVMe 驱动实现.内存访问模式下,DataNode 的读写线程可以通过 NVM 的直接内存访问接口(direct memory interface)按照内存语义(memory semantics)访问 NVM.客户端的 DataStreamer 和 Responder 分别基于 RDMA 发送数据和接收响应消息,DataNode 端的 RDMA Receiver 会接收网络中的数据,数据的读写通过 NVFS-BlkIO 或 NVFS-MemIO 接口进行.针对上层应用,NVFS 也作了一些优化,如 HBase 中持久化 HFile 和日志文件的延迟比较高,影响 Put 操作的性能,NVFS 中将 WAL 日志存储在 NVM 中,其他数据存储在 SSD 中.

整体而言,当前对 NVM 的研究还处于起步阶段,且由于缺乏大规模量产的 NVM 存储设备,当前的研究主要基于 NVM 模拟器进行,一定程度上阻碍了相关研究的进行.未来,随着 NVM 设备的大规模应用,NVM 能为 HDFS 带来巨大的性能提升.但如何让 NVM 以 HDFS 可感知的方式集成进来,以及如何针对 NVM 设计有效的数据放置策略,这些仍需要进一步加以研究.

3.3 网络传输

HDFS 作为一个分布式的存储系统,数据读取和写入过程中都可以涉及网络传输.对网络传输的优化也是 HDFS 优化的一个重要部分.

3.3.1 本地 I/O 和并行复制

HDFS 支持短路本地读取(short-circuit local reads)[86].当 HDFS Client 在 Datanode 上读取本地数据块时,可以通过短路本地读取来直接读取本地磁盘上的数据,避免通过 TCP 套接字读取数据.这样节省了数据从磁盘读入 Datanode 进程空间,然后复制到 TCP 发送缓冲区,再通过 TCP 协议栈发送到 Client 进程空间的过程,对于提高 I/O 性能有一定的帮助.在 HDFS 数据写入时,数据块需要复制到集群中的其他节点作为副本,文献[87]将 HDFS 中默认的流水线的复制方式改为并行复制来优化数据块的复制,即 DFSClient 负责将副本并行复制到所有节点,提高写数据时数据复制的性能.

3.3.2 RDMA

InfiniBand[88]是一种通用的高速连接网络,实现了 RDMA(remote direct memory access)技术,在企业中被广泛使用.RDMA 允许一个进程直接读取远端进程的内存数据,远端进程不需要参与数据传输的过程,只需要指定内存读写地址,开启传输即可,这个特性极大地降低了网络传输的延迟和传输中的 CPU 占用,同时减少了应用态和内核态之间的内存复制以及 CPU 的上下文切换.基于 RDMA 的网络连接,HDFS 的随机和顺序写性能能够在传统磁盘上分别提高 30%和 100%;配合固态硬盘使用时,性能提高更为明显[71].可以预见地,RDMA 将成为数据中心的标准网络连接技术.

为了优化 HDFS 中的数据传输,文献[89]设计和实现了基于 RDMA 的 HDFS.

- 首先,在 HDFS 的客户端和 DataNode 实现中修改了数据传输相关的类,添加了对 RDMA 的支持.这些类通过底层的 JNI 接口利用 InfiniBand 网络进行通信,JNI 接口中封装了轻量级、高性能的 C 语言通信库 UCR(unified communication runtime)[90],为网络通信提供运行时环境.修改后的 HDFS 降低了节点间数据传输的延迟,尤其是写数据块时数据复制的网络延迟,实验结果表明,可以将 HBase 的 Put 性能提高 26%;
- 其次,在 DFSClient 和 DataNode 中加入了 Connection 的概念,每个 Connection 对象都会维护一个预分配的缓冲区,用来避免 JNI 和 UCR 之间的数据复制.

除了数据传输外,消息通信也是 Hadoop 中网络通信的一大开销,对此,文献[91]实现了基于 RDMA 的 Hadoop RPC 框架——RPCoIB,提高 Hadoop 中 RPC 的性能达到 50%.

HDFS 中,数据写入是 I/O 密集型的操作,也是主要的性能瓶颈之一.HDFS 数据块的写入分为 4 个步骤.

- 从网络中读取数据到 Java 的 I/O 流中;
- 处理读取到的数据;
- 复制数据到副本中;
- 写数据到本地磁盘.

默认地,HDFS 写入时每个数据块由一个线程负责,每个线程按步骤顺序执行,这样每次网络中的数据分片都需要等待上一个分片写入磁盘后才能被读取和处理.在基于 RDMA 的 HDFS 中,这个问题更加明显,因为写入的性能瓶颈从网络传输变为了本地磁盘的持久化.为了解决这个问题,文献[92]提出在基于 RDMA 的 HDFS 中利用 SEDA(staged event-driven architecture)架构[93],为每个步骤分配一个队列,将单线程的顺序执行变为多步骤的并发执行.

3.4 小 结

HDFS 一开始就是面向传统磁盘设计和实现的,基于传统磁盘的优化和应用已经趋于成熟.但高密度、低成本的 SMR 磁盘为 HDFS 引入了新的研究挑战,且目前相关研究还有较多不足,包括设计高效利用 SMR 磁盘特性的文件系统以及选择合适的数据放入 SMR 磁盘中.固态硬盘在 HDFS 之外的研究工作很多,且在 Key-Value 数据存储等场景中应用十分成熟,但出于成本考虑,目前在 HDFS 集群中并未普遍采用固态硬盘替换传统磁盘,而是作为磁盘的辅助存储设备.这就带来了新的挑战,即,固态硬盘和磁盘如何协同使用以及数据如何在固态硬盘和磁盘之间动态迁移.大内存和 RDMA 技术在现代的数据中心中应用十分广泛,大幅提高了数据读取和网络传输的性能.NVM 是未来的研究热点,目前受制于 NVM 设备的研发,尚未有成熟的应用,已有的研究工作主要关注如何设计文件系统在 HDFS 中集成 NVM 设备,而未对 HDFS 的存储性能作深入优化.

未来,HDFS 集群必然是构建在高速 RDMA 网络和各类存储设备之上的异构集群,如何在异构体系结构平台上构建一个高效的分布式异构存储方案,包括对不同设备 I/O 性能的感知、文件及副本的放置策略、数据在设备间的动态迁移等,还有很大的研究空间,且这些研究工作涉及对 HDFS 较大的改动,需要与开源社区紧密合作,才能真正在产业界产生影响.

4 面向应用场景的 HDFS 存储和优化技术

在当今大数据的环境下,HDFS 作为通用的分布式存储系统,汇集了关系、文本、图结构等各种各样的海量数据,并向上支撑了丰富的应用场景.本节总结 HDFS 存储系统的主要应用场景,并分析和概括不同应用场景下典型的 HDFS 存储及优化技术(见表 6).

Table 6 Typical storage and optimization techniques of HDFS tailored for different applications

表 6 不同应用负载的 HDFS 存储和优化技术

I/O 负载特征	应用场景	应用示例	典型系统	优化技术
批量写入	数据汇聚存储	电信运营商、互联网企业数据汇聚存储	Hive; Scoop; Spark Streaming	纠删码[9] 数据压缩(见第 2.2 节) SMR 磁盘(见第 3.1.2 节) 并发复制(见第 3.3.1 节) 并发数据装载[43,110] RDMA(见第 3.3.2 节)
高吞吐读取	复杂查询分析	数据报表和决策支持	SQL-on-Hadoop	列存储(见第 2.1.2 节) 行列混合存储(见第 2.1.3 节) 数据压缩(见第 2.2 节) 数据索引(见第 2.3 节) 内存缓存(见第 3.2.1 节) RDMA(见第 3.3.2 节)

Table 6 Typical storage and optimization techniques of HDFS tailored for different applications (Continued)

表 6 不同应用负载的 HDFS 存储和优化技术(续)

I/O 负载特征	应用场景	应用示例	典型系统	优化技术
低延迟读取	Ad-hoc 查询和交互式分析	数据分析师交互式查询和可视化应用	Drill; ElasticSearch	列存储(见第 2.1.2 节) 行列混合存储(见第 2.1.3 节) 数据索引(见第 2.3 节) 固态硬盘(见第 3.1.3 节) 内存缓存(见第 3.2.1 节) 非易失性内存(见第 3.2.2 节) RDMA(见第 3.3.2 节)
	详单查询	电信运营商对流量和通话记录进行精细化管理和分析	SQL-on-Hadoop	
	Key-value 查询	Facebook Messages; Facebook 社交网络用户信息	HBase	固态硬盘(见第 3.1.3 节) 内存缓存(见第 3.2.1 节) LSM-Tree(见第 3.2.1 节) 非易失性内存(见第 3.2.2 节) RDMA(见第 3.3.2 节)
	流处理和计算	移动网络和物联网实时监控; 证券交易监控; 在线视频播放优化; 地理位置信息记录	Spark Streaming; Flink; Kafka	内存缓存(见第 3.2.1 节) 非易失性内存(见第 3.2.2 节) RDMA(见第 3.3.2 节)
	迭代计算	广告推荐;社区发现;视频推荐	Hama; GraphX; Mahout; Mllib	
实时更新	Key-value 存储	Facebook Messages; Facebook 社交网络用户信息	HBase	固态硬盘(见第 3.1.3 节) LSM-Tree(见第 3.2.1 节) 内存缓存(见第 3.2.1 节) 非易失性内存(见第 3.2.2 节) RDMA(见第 3.3.2 节)

4.1 应用场景

(1) 数据汇聚存储

在国内,电信运营商在日常运营中积累了大量用户数据,包括用户真实、详细的个人基本信息、通话者的地理位置信息和通话信息等,这些每天 TB 级别增长的业务数据汇聚存储在 HDFS 中,用于支撑上层用户偏好分析、网络流量优化等各种分析应用[94].相比电信运营商,互联网企业对 HDFS 的应用更为广泛.百度一度是国内 Hadoop 的最大使用者之一,拥有的 Hadoop 集群节点总数超过万台,存储了爬取的网页数据、用户日志数据和广告数据等[95].阿里巴巴基于 Hadoop 研制了云梯(cloud ladder)系统,利用 HDFS 存储面向分析的淘宝用户数据和交易信息等[96].在国外,Amazon、Facebook、Twitter、Yahoo!和 Hulu 等互联网公司基于 HDFS 汇聚用户行为等业务数据,为用户提供精准的分析.其中,Twitter 基于 HDFS 存储和分析用户发布的动态、日志文件和中间数据,集群规模超过万台,存储了超过 300PB 的数据[97].

在这些应用中,HDFS 存储的数据多种多样,既有结构化的数据,也有非结构化的数据;既有最新的热点数据,也有压缩存储的历史数据.在数据的汇聚存储过程中,预处理是一个普遍的需求.相比于传统数据库的 ETL (extract,transform,load),HDFS 的数据导入性能更高.Hive 和 Sqoop[98]利用 MapReduce 将数据库中的数据导入 HDFS 上.此外,Spark Streaming 可以对流式数据进行 ETL 操作,然后持久化到 HDFS 中.

(2) 复杂查询分析

基于 HDFS 的 SQL 和类 SQL 分析系统,如 Pig、Hive、Presto、Impala、Spark SQL、HAWQ[99]、Kylin[100]等支持对海量数据的复杂查询分析.这一类系统将数据表以数据文件和数据块的形式存储在 HDFS 中,由 HDFS 负责数据存储的负载均衡、容错和提供数据的高吞吐读写,利用大规模集群的并行计算能力对海量的结构化数据进行批量的分析,提供数据报表和决策支持.这类应用读取的数据量较大,要求底层存储系统有较高吞吐的数据读取能力.

(3) Ad-hoc 查询和交互式分析

Ad-hoc 查询和交互式分析是数据分析过程中常用的方式.例如,数据分析师通过对移动应用中用户行为数

据的分析来洞悉应用设计和产品逻辑的缺点,改善应用可用性和产品功能.在这个过程中,分析师往往需要结合数据可视化的技术,对数据进行多轮交互式的查询.交互中的查询大多是 ac-hoc 的,分析师会根据上一轮交互的结果临时生成,或在数据集中随机探索.这类应用要求查询执行的延迟较低,能够快速生成可视化的结果,并且查询的多样性较高.Apache Drill 和 ElasticSearch 等都可以支持这类应用,并在查询执行时要求底层存储系统提供低延迟的数据访问[101,102].

(4) 详单查询

HDFS 上存储了海量的数据,很多应用中需要根据精确的查询条件对海量数据进行查询.例如,电信运营商需要对流量和通话记录等进行精细化分析,针对网络日志中的某 IP 地址查询某段时间的被访问记录,或从通话记录中查询某号码的通话记录等[43].在这类应用中,查询实际需要访问的数据量很小,要求很低的数据读取延迟.因此,为避免全表扫描,数据过滤机制的设计非常重要.

(5) Key-Value 存储和查询

Key-Value 存储提供基于键值的低延迟、高吞吐的数据 CRUD 操作.BigTable 提出了在 GFS 上构建分布式 Key-Value 存储的系统实现方法.作为 BigTable 的开源实现,HBase 在 HDFS 上构建了分布式 Key-Value 存储.Facebook 基于 HBase 支持 Messages 和社交网络中用户信息等数据的存储和分析[103].由于 HDFS 不支持数据的随机写操作,HBase 将写入和修改的数据缓存在内存中,批量地与 HDFS 上的数据文件进行合并.

(6) 流计算

在移动网络、物联网等应用中,大量数据会实时生成.在从不同数据源实时收集这些数据的同时,对这些数据进行流式处理和计算也十分重要.证券行业中,证券交易监管机构需要对实时发生的证券交易进行监控,并发现其中的违法违规行为.在线游戏平台和视频播放平台实时分析用户的访问延时,改善用户的游戏和视频观看体验.地图应用实时记录用户的地理位置数据,更新用户的行为轨迹.

HDFS 上的流计算引擎,如 Spark Streaming[104]、Flink 等为了降低计算延迟,仅将 HDFS 作为流计算中数据的落地存储,或将 HDFS 中的文件作为数据源,通过流式的计算模型进行处理.而流计算的中间结果则通过内存数据结构进行管理.分布式消息队列系统,如 Kafka[105]也支持在落地 HDFS 之前对消息进行流式的处理.

(7) 迭代计算

在 HDFS 存储的海量数据上,机器学习和图计算等迭代计算的需求非常旺盛.腾讯广点通利用 Spark 内存计算和快速迭代的优势,支持其广告推荐算法的实时训练和系统的实时预测.淘宝利用 GraphX[106]对用户图数据进行分析,发现用户社区.优酷利用 Spark 实现在线视频的推荐和广告投放.

HDFS 上的图计算引擎,如 Hama[107]、GraphX 等,机器学习框架,如 Mahout[108]、MLlib[109]等,均基于 HDFS 支持大规模的迭代计算.由于 HDFS 本身不支持随机低延迟的数据读写,因此,HDFS 上的迭代计算应用通常需要额外设计消息传输机制和基于内存的数据结构.

本小节总结了 HDFS 常见的应用,这些应用与 HDFS 之间有丰富的数据读写交互.在对这些交互行为进行分析后,本文将它们背后的 I/O 特征归纳分类为高吞吐读取、低延迟读取、批量写入和实时更新这 4 种模式.其中,复杂查询分析的 I/O 模式主要为高吞吐读取,Ad-hoc 查询和交互式分析、详单查询、Key-value 查询、流处理和计算和迭代式计算主要为低延迟读取,数据汇聚存储主要是批量写入,Key-value 存储主要为实时更新.不同的 I/O 模式需要应用不同的 HDFS 存储和优化技术.

4.2 高吞吐读取

高吞吐读取是 HDFS 的看家本领,也是 HDFS 上最为常见的数据访问模式.HDFS 本身的设计就是为了优化高吞吐的并行数据读取和计算.但随着基于 HDFS 的 SQL 查询分析系统,如 Hive、Presto、Impala、Spark SQL 等的广泛应用,针对批量查询分析的存储优化技术也不断出现和发展.

从文件逻辑结构的维度看,由于复杂查询分析中大部分查询通常只访问数据表中的部分属性,列式数据存储成为目前通用的存储方案.列存储可以支持仅读取需要列的数据,并且拥有比行存储更好的数据压缩效果,节省磁盘 I/O.行列混合存储根据查询负载中的数据访问特征进一步优化列存储的物理布局,如将一些被频繁访问

的列的组合在物理上存储在一起,可以有效减少数据读取时的跳读代价.列存储和行列混合存储中,数据压缩和索引是常用的减少数据读取 I/O 的方法,结合文件存储格式,能有效提高查询效率.

从硬件设备的维度看,随着大内存的普及,基于 Alluxio 的内存文件缓存和分析引擎内部的数据缓存机制等都能充分利用大内存的优势,加速复杂查询的执行.在存储介质方面,RDMA 可以降低大量数据网络传输时的 CPU 消耗,以及减少用户态和内核态之间的数据复制,提高集群中的网络效率.

以基于 Presto 的复杂 SQL 查询为例,HDFS 上的关系表存储为许多列存储文件,这些文件经过压缩,并在元数据中记录统计信息作为索引.HDFS 可以根据历史查询的访问特征对列存储文件的物理布局进行自适应优化.并且,基于大内存的硬件平台,Presto 可以利用 Alluxio 进行数据缓存,避免每次查询都从磁盘读取大量数据.远程读取数据和执行的查询任务还可以利用 RDMA 降低数据传输的延迟.

4.3 低延迟读取

低延迟读取是 HDFS 的弱项.但 HDFS 集群构建成本低,提供了良好的扩展性、容错性和高的可用性,汇聚了海量的各类数据.其中,对很多数据的访问要求 HDFS 提供低延迟的读取.

利用大内存、固态硬盘、非易失性内存和 RDMA 等硬件加速是降低数据读取延迟的主要方法.HDFS 支持集中式缓存管理,可以将指定的文件缓存到节点的内存中.查询引擎一般也利用内部的内存管理机制加速数据的访问,如 Spark 中的 RDD.此外,还可配合通用的分布式内存管理系统 Alluixio 一起使用,将不同应用频繁访问的数据自动缓存在内存中.随着固态硬盘容量的增加和成本的降低,固态硬盘的使用更加普遍.相比于磁盘,固态硬盘拥有更好的读写性能,适合低延迟的应用场景.非易失性内存能够基于字节访问数据,提供优于固态硬盘的数据访问性能,同时还能持久化地存储数据,非常适合低延迟读取的访问负载.当需要远程读取数据时,网络延迟也是数据读取延迟的重要组成部分.RDMA 可以在 InfiniBand 的基础上,允许一个进程直接读取远端进程的内存数据,而远端进程不需要参与数据传输的过程,只需要指定内存读写地址,开启传输即可.

在磁盘上,列存储和行列混合存储结合索引技术可以帮助快速定位要读取的数据的位置,也有助于降低查询的 I/O 延迟.ORC、Parquet、CarbonData 等列存储格式在元数据中记录各个行组上的统计信息,查询可以根据统计信息过滤掉不相关的行组.ORC 中还支持 bitmap 索引,记录列中的数据是否为空值.另外,CarbonData 支持建立多维聚簇索引,在 Key-Value 查询时,LSM-Tree 本身也是一种索引结构,可以降低查询的延时.

以交互式分析为例,应用程序利用 Alluxio 和查询引擎的内置缓存机制将聚合的中间结果缓存在内存中,还可以将部分数据持久化到非易失内存上减少对磁盘的访问.磁盘上,数据按列存储,并建立多种索引机制,加速数据从磁盘读取到内存.另外,节点间的网络通信利用 RDMA 技术加速.

4.4 批量写入

HDFS 作为文件系统,相对于传统的数据库系统,其本身具有很好的数据批量写入性能.在数据写入过程中,数据块需要复制到集群中的其他节点,此时,并发复制和 RDMA 技术都可以加速这个过程.另外,文献[43]和文献[110]针对日志数据,设计和实现了高效的并发数据写入的流水线,可以提高数据加载的性能,并允许应用访问正在加载中的数据.如今,数据分析越来越注重时效性,数据快速批量入库的需求越来越强烈,而目前相关的研究和系统实现还较少.

随着批量写入的数据量的不断增加,存储的代价也越来越高.一方面,数据压缩能够减少实际存储的数据大小;另一方面,能支持更高密度、更廉价存储的 SMR 磁盘受到大家的青睐.尽管由于其底层的叠瓦式设计,SMR 在随机更新时会有很高的写放大,但是在顺序写入时能保证高的写入性能,非常适合作为冷数据的存储设备.另外,冗余副本容错机制也是导致存储数据量变大的原因之一,HDFS 中引入的纠删码 EC 机制可以在保证同等可靠性的情况下,将存储利用率提高近 1 倍.

4.5 实时更新

HDFS 在数据写入方面的一个主要限制是只支持数据的追加式写入,而实时更新则要求对数据内容进行毫秒级延迟的更新操作,并支持较高的事务吞吐量.显然,在 HDFS 的文件系统上直接更新数据无法满足性能要求.

目前,HDFS 上的数据更新技术主要基于 LSM-tree 的思想,数据更新先记录在内存中,然后批量地写入文件系统.典型的系统实现是实现 Key-value 模型的 HBase.此外,HDFS 上的列存储格式,如 ORC 和 CarbonData,也支持对数据进行低延迟的更新,其实现原理与 LSM-Tree 类似.但由于列存储本身并不适合数据更新操作,所以更新操作的吞吐量很低,通常只有每秒几次至几十次.LSM-Tree 的思想是将数据更新操作暂存在内存中,对内存消耗较大,大内存可以将更多的数据缓存在内存中,同时,固态硬盘和非易失性内存可以提供远优于磁盘的随机访问性能,结合这些可以更好地发挥 LSM-tree 的优势.另外,RDMA 可以降低节点间网络传输的延迟,有助于提高数据更新的性能.

4.6 小 结

本节从高吞吐读取、低延迟读取、批量写入和实时更新这 4 种 I/O 模式分析了 HDFS 上常见的应用,及其对应的典型存储和优化技术.在很多实际场景中,同一个 HDFS 集群上往往运行着多种类型的应用程序,如同时存在需要高吞吐读取的复杂 SQL 查询和需要低延迟读取的图查询和 KV 查询,并且还有大量数据在不断批量写入.未来的 HDFS 存储更可能作为一个存储的基础设施,上层应用按照不同的抽象数据模型管理和分析数据.例如:数据以文本形式被批量装载到 HDFS 上,然后被抽取为关系表和图进行查询分析;同时,作为机器学习模型训练的输入,机器学习和查询分析的中间结果以 KV 的形式存储,以供未来使用.在对这种多样化应用的支持上,目前 HDFS 还有很多不足:首先,HDFS 不能很好地针对不同数据的访问特征进行定制化的优化;其次,由于主要依赖磁盘读写数据,HDFS 在面向低延迟的数据访问上还有很大不足;最后,HDFS 不支持文件随机更新,在实时更新方面存在很大的应用限制.这些问题还需要通过结合新的存储硬件进一步深入研究来加以解决.

5 总结及未来研究方向

本文从文件逻辑结构、硬件设备和应用场景这 3 个维度对现有的 HDFS 存储和优化技术进行了分析和总结.目前,HDFS 作为通用的分布式文件系统,强调高可扩展和低成本地提供高可靠的海量数据存储,保证副本的强一致,高吞吐地落地快速生成的数据,以及基于文件存储多样化的数据,包括结构化数据和文档、图等半结构化和非结构化的数据.HDFS 的这些特点契合了大数据的大容量、生成快和多类型的特性,未来更可能作为大数据存储和分析的基础设施,支持多模型和多应用的数据存储.另外,随着异构体系结构的大数据平台在工业界逐渐成为常态,新型的硬件设备也开始迅速在企业中得到推广和应用,HDFS 的底层硬件平台也将从传统磁盘和低速网络走向异构存储和高速网络连接.在 HDFS 未来的发展方向上,目前的研究工作虽然已经取得了一定的进展,但仍然存在很多问题值得深入探讨和研究.本节总结了未来研究可能的挑战和机遇.

1) 基于异构平台的数据存储

从硬件平台来看,存储设备的发展十分迅速.HDFS 的设计初衷是基于通用的廉价硬件提供可靠、高吞吐的数据存储和访问.但随着硬件的发展,传统的磁盘性能和存储容量都已经达到瓶颈,新的硬件,如固态硬盘、非易失性内存和 SMR 磁盘等受到广泛关注.目前,HDFS 已有的功能和研究着重于将不同硬件集成到 HDFS 中,但还没有很好的机制让 HDFS 能够智能感知不同设备的 I/O 特性,并根据数据的访问特征动态改变数据的存储方式,在异构的环境下最大程度地发挥各类硬件的性能优势.这里面临的挑战主要包括:设计 HDFS 对设备 I/O 性能的感知机制;结合设备 I/O 特点和应用需求动态放置和迁移文件及其副本;新硬件在集成时还需要针对硬件特点进一步优化,如叠瓦式磁盘的随机写放大、固态硬盘的寿命问题等.

2) 面向应用负载的自适应存储优化

从上层应用来看:

- 一方面,在大数据 Hadoop 生态系统不断发展的过程中,HDFS 因其自身的稳定可靠、简单易用、扩展性高等优点,使得越来越多的上层应用和系统将其作为统一的底层存储.HDFS 成为了事实上的“数据中心”,其上存储的数据类型和支持的分析负载越来越多元化;
- 另一方面,在企业中,不同部门和用户经常基于同一份全量数据进行查询分析,如不同用户分别在 Presto 和 Hive 上查询同一份关系数据,导致同一份数据服务多样的查询负载.在这种应用场景下,基于

人工制定策略的存储优化难以生效,需要研究根据应用负载的自适应优化技术,如行列混合存储中根据应用负载自动优化列的物理存储顺序[24].

自适应优化技术在传统数据库中早有研究[111],为 HDFS 上研究的开展提供了很好的借鉴.从 20 世纪 90 年代开始,用于决策支持的分析型数据库快速发展.相对于事务型数据库,分析型数据库中的查询通常更为复杂,查询优化技术也相对复杂,导致分析型数据库难以完全依靠人工调优[111].在索引选择、物化视图、数据划分等偏向存储方面的自适应优化变得非常关键.在自适应索引选择[112,113]、查询计划优化[114]、数据存储格式优化[115]等方面出现了很多相关的研究工作.

3) 结合机器学习的存储优化技术

由于数据存储优化问题复杂度非常高,早期的存储优化主要基于固定的规则和策略.随着大规模机器学习的发展,一些研究工作也尝试将深度学习等技术用于解决数据存储和查询执行的优化问题[46,116−118].HDFS 上的应用类型复杂多样,硬件平台也趋于多样化,对于 HDFS 上的存储优化问题,数据库优化的思想具有很好的借鉴意义,包括利用 RNN 模型对查询负载建模和预测[116]、利用神经网络构建数据索引[46]、利用强化学习解决复杂优化问题[119]等.

References:

[1] Karun AK, Chitharanjan K. A review on hadoop—HDFS infrastructure extensions. In: Proc. of the 2013 IEEE Conf. on Information & Communication Technologies. IEEE, 2013. 132−137.

[2] Du XY, Lu W, Zhang F. History, present, and future of big data management systems. Ruan Jian Xue Bao/Journal of Software, 2019,30(1):127−141 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/5644.htm [doi: 10.13328/j.cnki.jos.005644]

[3] Cafarella M, Cutting D. Building Nutch: Open source search. Queue, 2004,2(2):54.

[4] Sanjay G, Howard G, Shun-Tak L. The Google file system. In: Proc. of the SOSP. 2003. 29−43.

[5] Dean J, Ghemawat S. MapReduce: Simplified data processing on large clusters. Communications of the ACM, 2008,51(1):107−13.

[6] Hortonworks. Heterogeneous storages in HDFS. https://zh.hortonworks.com/blog/heterogeneous-storages-hdfs/

[7] Apache Hadoop. Centralized cache management in HDFS. http://hadoop.apache.org/docs/r2.3.0/hadoop-project-dist/hadoop-hdfs/CentralizedCacheManagement.html

[8] Apache Hadoop. Archival storage, SSD & memory. https://hadoop.apache.org/docs/r2.6.0/hadoop-project-dist/hadoop-hdfs/ArchivalStorage.html

[9] Apache Hadoop. Erasure coding. https://hadoop.apache.org/docs/r3.0.0/hadoop-project-dist/hadoop-hdfs/HDFSErasureCoding.html

[10] Konstantin S, Hairong K, Sanjay R, Robert C. The Hadoop distributed file system. In: Proc. of the MSST. 2010. 1−10.

[11] White T. Hadoop: The Definitive Guide. 4th ed., O'Reilly Media, Inc., 2015.

[12] Hakim W, John K. Erasure coding vs. replication: A quantitative comparison. In: Proc. of the IPTPS Workshop. 2001. 328−338.

[13] Xia MY, Mohit S, Mario B, David AP. A tale of two erasure codes in HDFS. In: Proc. of the FAST. 2015. 213−226.

[14] Apache Avro. Avro official site. https://avro.apache.org/

[15] He YQ, Lee RB, Huai Y, Shao Z, Jain N, Zhang XD, Xu ZW. RCFile: A fast and space-efficient data placement structure in MapReduce-based warehouse systems. In: Proc. of the ICDE. 2011. 1199−1208.

[16] Apache ORC. ORC official site. https://orc.apache.org/

[17] Apache Parquet. Parquet official site. http://carbondata.apache.org/

[18] Apache CarbonData. CarbonData official site. http://carbondata.apache.org/

[19] Copeland GP, Khoshafian SN. A decomposition storage model. ACM SIGMOD Record, 1985,14(4):268−279.

[20] Stonebraker M, Abadi DJ, Batkin A, Chen X, Cherniack M, Ferreira M, Lau E, Lin A, Madden S, O'Neil E, O'Neil P. C-store: A column-oriented DBMS. In: Proc. of the 31st Int'l Conf. on Very Large Data Bases. VLDB Endowment, 2005. 553−564.

[21] Abadi J D, Madden RS, Hachem N. Column-stores vs. row-stores: How different are they really? In: Proc. of the SIGMOD. 2008. 967−980.

[22] Ailamaki A, DeWitt DJ, Hill MD, Skounakis M. Weaving relations for cache performance. In: Proc. of the VLDB, Vol. 1. 2001. 169−180.

[23] Huai Y, Ma SY, Lee RB, O'Malley O, Zhang XD. Understanding insights into the basic structure and essential issues of table placement methods in clusters. In: Proc. of the VLDB. 2013. 1750−1761.

[24] Bian HQ, Yan Y, Tao WB, Chen JL, Chen YG, Du XY, Moscibroda T. Wide table layout optimization based on column ordering and duplication. In: Proc. of the SIGMOD. 2017. 299−314.

[25] Thusoo A, Sarma JS, Jain N, Shao Z, Chakka P, Anthony S, Liu H, Wyckoff P, Murthy R. Hive: A warehousing solution over a map-reduce framework. Proc. of the VLDB Endowment, 2009,2(2):1626−1629.

[26] Apache Pig. Pig official site. https://pig.apache.org/

[27] GZip. GZip official site. http://www.gzip.org/

[28] Melnik S, Gubarev A, Long JJ, Romer G, Shivakumar S, Tolton M, Vassilakis T. Dremel: Interactive analysis of Web-scale datasets. In: Proc. of the VLDB. 2010. 330−339.

[29] Armbrust M, Xin SR, Lian C, Huai Y, Liu D, Bradley KJ, Meng XR, Kaftan T, Franklin JM, Ghodsi A, Zaharia M. Spark SQL: Relational data processing in spark. In: Proc. of the SIGMOD. 2015. 1383−1394.

[30] Facebook. Presto official site. https://prestodb.io/

[31] Apache Impala. Impala official site. https://impala.apache.org/

[32] Google. Snappy official site. https://google.github.io/snappy/

[33] Oberhumer. LZO official site. https://www.lzop.org/

[34] Floratou A, Patel JM, Shekita EJ, Tata S. Column-oriented storage techniques for MapReduce. In: Proc. of the VLDB. 2011. 419−429.

[35] Guo SJ, Xiong J, Wang WP, Lee RB. Mastiff: A mapreduce-based system for time-based big data analytics. In: Proc. of the CLUSTER. 2012. 72−80.

[36] Alekh J, Jorge-Arnulfo Q, Dittrich J. Trojan data layouts: Right shoes for a running elephant. In: Proc. of the SOCC. 2011. 21.

[37] BZip. BZip2 official site. http://www.bzip.org/

[38] Shapira G, Seidman J, Malaska T, Grover M. Hadoop Application Architectures. O'Reilly Media, Inc., 2015.

[39] Apache. Hive language manual. https://cwiki.apache.org/confluence/display/Hive/LanguageManual

[40] Dittrich J, Quiané-Ruiz JA, Richter S, Schuh S, Jindal A, Schad J. Only aggressive elephants are fast elephants. In: Proc. of the VLDB. 2012. 1591−1602.

[41] Richter S, Quiané-Ruiz JA, Schuh S, Dittrich J. Towards zero-overhead adaptive indexing in hadoop. arXiv Preprint arXiv:1212.3480, 2012.

[42] He L, Chen JC, Du XY. Multi-layered index for HDFS-based systems. Ruan Jian Xue Bao/Journal of Software, 2017,28(3): 502−513 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/5161.htm [doi: 10.13328/j.cnki.jos.005161]

[43] Bian HQ, Chen YG, Qin XP, Du XY. A fast data ingestion and indexing scheme for real-time log analytics. In: Proc. of the APWeb. 2015. 841−852.

[44] Zhao LP. DuBAI: Duplication-based indexes for HDFS columnar storage [MS. Thesis]. Beijing: Renmin University of China, 2017 (in Chinese with English abstract).

[45] Agarwal R, Khandelwal A, Stoica I. Succinct: Enabling queries on compressed data. In: Proc. of the USENIX NSDI. 2015. 337−350.

[46] Kraska T, Beutel A, Chi EH, Dean J, Polyzotis N. The case for learned index structures. In: Proc. of the 2018 Int'l Conf. on Management of Data. 2018. 489−504.

[47] Apache Hadoop. HDFS disk balancer. https://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/HDFSDiskbalancer.html

[48] Shafer J, Rixner S, Cox AL. The Hadoop distributed filesystem: Balancing portability and performance. In: Proc. of the ISPASS. 2010. 122−133.

[49] O'Neil P, Cheng E, Gawlick D, O'Neil E. The log-structured merge-tree (LSM-tree). Acta Informatica, 1996,33(4):351−385.

[50] Chang F, Dean J, Ghemawat S, Hsieh WC, Wallach DA, Burrows M, Gruber RE. Bigtable: A distributed storage system for structured data. ACM Trans. on Computer Systems (TOCS), 2008,26(2):4.

[51] Apache. HBase official site. http://hbase.apache.org/

[52] White T. Small files problem. https://blog.cloudera.com/blog/2009/02/the-small-files-problem/

[53] Tim F, Gibson G. Shingled magnetic recording: Areal density increase requires new data management. ACM Trans. on Storage, 2013,38(3):22.

[54] Abutalib A, Shafaei M, Desnoyers P. Skylight—A window on shingled disk operation. In: Proc. of the TOS. 2015. 16.

[55] Amer A, Long DD, Miller EL, Paris JF, Schwarz ST. Design issues for a shingled write disk system. In: Proc. of the MSST. 2010. 1–12.

[56] Anand S, Gibson G, Ganger G. Shingled magnetic recording for big data applications. Carnegie Mellon University Parallel Data Laboratory Technical Report, 2012.

[57] Patana-anake T, Martin V, Sandler N, Wu C, Gunawi HS. Manylogs: Improved CMR/SMR disk bandwidth and faster durability with scattered logs. In: Proc. of the MSST. 2016. 1–16.

[58] Cassuto Y, Sanvido MA, Guyot C, Hall DR, Bandic ZZ. Indirection systems for shingled-recording disk drives. In: Proc. of the MSST. 2010. 1–14.

[59] He WP, Du DHC. SMaRT: An approach to shingled magnetic recording translation. In: Proc. of the FAST. 2017. 121–134.

[60] Wikipedia. Solid-state drive. https://en.wikipedia.org/wiki/Solid-state_drive

[61] Do J, Kee YS, Patel JM, Park C, Park K, DeWitt DJ. Query processing on smart SSDs: Opportunities and challenges. In: Proc. of the 2013 ACM SIGMOD Int'l Conf. on Management of Data. ACM Press, 2013. 1221–1230.

[62] Alluxio Inc. Accelerating On-demand Data Analytics with Alluxio. While Paper. 2016.

[63] Alluxio. Effective spark RDDs with Alluxio. https://www.alluxio.com/blog/effective-spark-rdds-with-alluxio

[64] Krish KR, Anwar A, Butt AR. hats: A heterogeneity-aware tiered storage for Hadoop. In: Proc. of the CCGrid. 2014. 502–511.

[65] Islam NS, Lu X, Wasi-ur-Rahman M, Shankar D, Panda DK. Triple-H: A hybrid approach to accelerate HDFS on HPC clusters with heterogeneous storage architecture. In: Proc. of the CCGrid. 2015. 101–110.

[66] Subramanyam R. HDFS heterogeneous storage resource management based on data temperature. In: Proc. of the 2015 Int'l Conf. on Cloud and Autonomic Computing (ICCAC). IEEE, 2015. 232–235.

[67] Krish KR, Iqbal MS, Butt AR. Venu: Orchestrating SSDS in Hadoop storage. In: Proc. of the 2014 IEEE Int'l Conf. on Big Data (big data). IEEE, 2014. 207–212.

[68] Sangwhan M, Lee J, Kee YS. Introducing ssds to the hadoop mapreduce framework. In: Proc. of the CLOUD. 2014. 272–279.

[69] Sur S, Wang H, Huang J, Ouyang X, Panda DK. Can high-performance Interconnects benefit Hadoop distributed file system. In: Proc. of the MASVDC Workshop. 2010.

[70] Hong J, Li L, Han C, Jin B, Yang Q, Yang Z. Optimizing Hadoop framework for solid state drives. In: Proc. of the Big Data. 2016. 9–17.

[71] Kang Y, Kee YS, Miller EL, Park C. Enabling cost-effective data processing with smart SSD. In: Proc. of the 29th IEEE Symp. on Mass Storage Systems and Technologies (MSST). IEEE, 2013. 1–12.

[72] Dongchul P, Wang JG, Kee YS. In-storage computing for Hadoop MapReduce framework: Challenges and possibilities. IEEE Trans. on Computers, 2016,PP(99):1–1.

[73] Colgrove J, Davis JD, Hayes J, Miller EL, Sandvig C, Sears R, Wang F. Purity: Building fast, highly-available enterprise Flash storage from commodity components. In: Proc. of the SIGMOD. 2015. 1683–1694.

[74] McCabe C, Wang A. New in CDH 5.1: HDFS read caching. http://blog.cloudera.com/blog/2014/08/new-in-cdh-5-1-hdfs-read-caching

[75] Mauerer W. Professional Linux Kernel Architecture. Wrox, 2008.

[76] Li H, Ghodsi A, Zaharia M, Shenker S, Stoica I. Tachyon: Reliable, memory speed storage for cluster computing frameworks. In: Proc. of the SOCC. 2014. 1–15.

[77] Apache Arrow. Apache arrow official site. https://arrow.apache.org

[78] Zaharia M, Chowdhury M, Das T, Dave A, Ma J, McCauley M, Stoica I. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In: Proc. of the NSDI. 2012. 2.

[79] Carbone P, Katsifodimos A, Ewen S, Markl V, Haridi S, Tzoumas K. Apache flink: Stream and batch processing in a single engine. Bulletin of the IEEE Computer Society Technical Committee on Data Engineering, 2015,36(4).

[80] Carbone P, Fóra G, Ewen S, Haridi S, Tzoumas K. Lightweight asynchronous snapshots for distributed dataflows. arXiv Preprint arXiv:1506.08603, 2015.

[81] Apache Drill. Apache drill official site. https://drill.apache.org

[82] Gupta P, Nair S. Survey paper on elasticsearch. Int'l Journal of Science and Research, 2016,5(1):333−336.

[83] Shu J, Lu Y, Zhang J, Zheng W. Research progress on non-volatile memory based storage system. Science & Technology Review, 2016,14:019 (in Chinese with English abstract).

[84] Pelley S, Wenisch TF, Gold BT, Bridge B. Storage management in the NVRAM era. In: Proc. of the VLDB. 2013. 121−132.

[85] Islam NS, Wasi-ur-Rahman M, Lu XY, Panda DK. High performance design for HDFS with byte-addressability of NVM and RDMA. In: Proc. of the Int'l Conf. on Supercomputing. 2016. 8−21.

[86] Apache Hadoop. HDFS short-circuit local reads. https://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/ShortCircuitLocalReads.html

[87] Islam NS, Lu XY, Wasi-ur-Rahman M, Panda DK. Can parallel replication benefit hadoop distributed file system for high performance interconnects? In: Proc. of the HOTI. 2013. 75−78.

[88] InfiniBand. InfiniBand official website. http://www.infinibandta.org

[89] Islam NS, Rahman MW, Jose J, Rajachandrasekar R, Wang H, Subramoni H, Panda DK. High performance RDMA-based design of HDFS over InfiniBand. In: Proc. of the Int'l Conf. on High Performance Computing, Networking, Storage and Analysis. 2012. 35−46.

[90] Jose J, Luo M, Sur S, Panda DK. Unifying UPC and MPI runtimes: Experience with MVAPICH. In: Proc. of the 4th Conf. on Partitioned Global Address Space Programming Model. 2010. 5.

[91] Lu XY, Islam NS, Wasi-Ur-Rahman M, Jose J, Subramoni H, Wang H, Panda DK. High-performance design of Hadoop RPC with RDMA over InfiniBand. In: Proc. of the ICPP. 2013. 641−650.

[92] Islam NS, Lu XY, Rahman MWU, Panda DK. SOR-HDFS: A SEDA-based approach to maximize overlapping in RDMA-enhanced HDFS. In: Proc. of the Int'l Symp. on High-Performance Parallel and Distributed Computing. 2014. 261−264.

[93] Matt W, Culler D, Brewer E. SEDA: An architecture for well-conditioned, scalable internet services. In: Proc. of the SIGOPS. 2001. 230−243.

[94] Yu F. Case analysis on telecom Carriers' application of big data. Proc. of the Systems & Solutions, 2014,8(6):63−69, 83 (in Chinese with English abstract).

[95] Xia JB, Wei ZK, Fu K, Chen Z. Review of research and application on Hadoop in cloud computing. Proc. of the Computer Science, 2016,43(11):6−11 (in Chinese with English abstract).

[96] Guo MJ. Research on big data and cloud computing platform applications. Proc. of the Modern Science & Technology of Telecommunications, 2014,44(8):7−11 (in Chinese with English abstract).

[97] Twitter. Hadoop filesystem at Twitter. https://blog.twitter.com/engineering/en_us/a/2015/hadoop-filesystem-at-twitter.html

[98] Apache Sqoop. Sqoop official site. https://sqoop.apache.org

[99] Chang L, Wang ZW, Ma T, Jian LR, Ma LL, Goldshuv A, Lonergan L, Cohen J, Sherry G, Bhandarkar M. HAWQ: A massively parallel processing SQL engine in Hadoop. In: Proc. of the SIGMOD. 2014. 1223−1234.

[100] Ranawade VS, Navale S, Dhamal A. Online analytical processing on Hadoop using apache skylin. Int'l Journal of Applied Information Systems, 2017,12(2):1−5.

[101] Hausenblas M, Nadeau J. Apache Drill: Interactive ad-hoc analysis at scale. Big Data, 2013,1(2):100−104.

[102] Elasticsearch BV. Elasticsearch-Hadoop: Best of two worlds for real-time analysis. https://www.elastic.co/products/hadoop

[103] Aiyer AS, Bautin M, Chen GJ, Damania P, Khemani P, Muthukkaruppan K, Ranganathan K, Spiegelberg N, Tang L, Vaidya M. Storage infrastructure behind Facebook messages: Using HBase at scale. IEEE Data Engineering Bulletin, 2012,35(2):4−13.

[104] Zaharia M, Das T, Li H, Hunter T, Shenker S, Stoica I. Discretized streams: Fault-tolerant streaming computation at scale. In: Proc. of the SOSP. 2013. 423−438.

[105] Kreps J, Narkhede N, Rao J. Kafka: A distributed messaging system for log processing. In: Proc. of the NetDB. 2011. 1−7.

[106] Gonzalez JE, Xin RS, Dave A, Crankshaw D, Franklin MJ, Stoica I. GraphX: Graph processing in a distributed dataflow framework. In: Proc. of the OSDI. 2014. 599−613.

[107] Siddique K, Akhtar Z, Yoon EJ, Jeong YS, Dasgupta D, Kim Y. Apache Hama: An emerging bulk synchronous parallel computing framework for big data applications. IEEE Access, 2016,4:8879−8887.

[108] Apache. Mahout official site. https://mahout.apache.org/

[109] Meng X, Bradley J, Yavuz B, Sparks E, Venkataraman S, Liu D, Xin D. Mllib: Machine learning in apache spark. The Journal of Machine Learning Research, 2016,17(1):1235−1241.

[110] Jin G, Wang Y, Qin X, Chen Y, Du X. Towards real-time analysis of ID-associated data. In: Proc. of the Int'l Conf. on Conceptual Modeling. 2018. 26−30.

[111] Chaudhuri S, Narasayya VR. Self-Tuning database systems: A decade of progress. In: Proc. of the VLDB. 2007. 3−14.

[112] Chaudhuri S, Narasayya VR. An efficient, cost-driven index selection tool for Microsoft SQL server. In: Proc. of the VLDB. 1997. 146−155.

[113] Bruno N, Chaudhuri S. An online approach to physical design tuning. In: Proc. of the ICDE. 2007. 826−835.

[114] Chakkappen S, Budalakoti S, Krishnamachari R, Valluri SR, Wood A, Zait M. Adaptive statistics in Oracle 12c. In: Proc. of the VLDB. 2017. 1813−1824.

[115] Alagiannis I, Idreos S, Ailamaki A. H2O: A hands-free adaptive store. In: Proc. of the SIGMOD. 2014. 1103−1114.

[116] Pavlo A, Angulo G, Arulraj J, Lin H, Lin J, Ma L, Santurkar S. Self-driving database management systems. In: Proc. of the CIDR. 2017.

[117] Arulraj J, Pavlo A, Menon P. Bridging the archipelago between row-stores and column-stores for hybrid workloads. In: Proc. of the SIGMOD. 2016. 583−598.

[118] Van Aken D, Pavlo A, Gordon GJ, Zhang B. Automatic database management system tuning through large-scale machine learning. In: Proc. of the SIGMOD. 2017. 1009−1024.

[119] Sharma A, Schuhknecht FM, Dittrich J. The case for automatic database administration using deep reinforcement learning. arXiv Preprint arXiv:1801.05643, 2018.

附中文参考文献:

[2] 杜小勇,卢卫,张峰.大数据管理系统的历史、现状与未来.软件学报,2019,30(1):127−141. http://www.jos.org.cn/1000-9825/5644.htm [doi: 10.13328/j.cnki.jos.005644]

[42] 何龙,陈晋川,杜小勇.一种面向 HDFS 的多层索引技术.软件学报,2017,28(3):502−513. http://www.jos.org.cn/1000-9825/5161.htm [doi: 10.13328/j.cnki.jos.005161]

[44] 赵丽萍.DuBAI:基于数据复制的 HDFS 列存储上的索引技术研究与实现[硕士学位论文].北京:中国人民大学,2017.

[83] 舒继武,陆游游,张佳程,郑纬民.基于非易失性存储器的存储系统技术.科技导报,2016,32(14):86−94.

[94] 余飞.电信运营商大数据应用典型案例分析.信息通信技术,2014,8(6):63−69,83.

[95] 夏靖波,韦泽鲲,付凯,陈珍.云计算中 Hadoop 技术研究与应用综述.计算机科学,2016,43(11):6−11.

[96] 郭敏杰.大数据和云计算平台应用研究.现代电信科技,2014,44(8):7−11.

金国栋(1993－),男,安徽安庆人,学士,CCF 学生会员,主要研究领域为数据库,大数据分析系统.

陈跃国(1978－),男,博士,教授,博士生导师,CCF 高级会员,主要研究领域为高性能大数据分析系统,知识图谱,语义搜索.

卞昊穹(1989－),男,博士,高级工程师,CCF 学生会员,主要研究领域为数据库,分布式存储系统.

杜小勇(1963－),男,博士,教授,博士生导师,CCF 会士,主要研究领域为数据库,大数据系统.