软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2017,28(3):514−543 [doi: 10.13328/j.cnki.jos.005169]
©中国科学院软件研究所版权所有.
E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

MapReduce 大数据处理平台与算法研究进展*

宋 杰[1], 孙宗哲[1], 毛克明[1], 鲍玉斌[2], 于 戈[2]

[1](东北大学 软件学院,辽宁 沈阳 110819)

[2](东北大学 计算机科学与工程学院,辽宁 沈阳 110819)

通讯作者: 宋杰, E-mail: songjie@mail.neu.edu.cn

摘 要: 综述了近年来基于 MapReduce 编程模型的大数据处理平台与算法的研究进展.首先介绍了 12 个典型的基于 MapReduce 的大数据处理平台,分析对比它们的实现原理和适用场景,抽象其共性;随后介绍基于 MapReduce 的大数据分析算法,包括搜索算法、数据清洗/变换算法、聚集算法、连接算法、排序算法、偏好查询、最优化算法、图算法、数据挖掘算法,将这些算法按照 MapReduce 实现方式分类,分析影响算法性能的因素;最后,将大数据处理算法抽象为外存算法,并对外存算法的特征加以梳理,提出了普适的外存算法性能优化方法的研究思路和问题,以供研究人员参考.具体包括优化外存算法的磁盘 I/O、优化外存算法的局部性以及设计增量式迭代算法.现有的大数据处理平台和算法研究多集中在基于资源分配和任务调度的平台动态性能优化、特定算法并行化、特定算法性能优化等领域,所提出的外存算法性能优化属于静态优化方法,是现有研究的良好补充,为研究人员提供了广阔的研究空间.

关键词: 大数据;MapReduce;外存算法;大数据处理;算法性能优化

中图法分类号: TP311

中文引用格式: 宋杰,孙宗哲,毛克明,鲍玉斌,于戈.MapReduce 大数据处理平台与算法研究进展.软件学报,2017,28(3): 514−543. http://www.jos.org.cn/1000-9825/5169.htm

英文引用格式: Song J, Sun ZZ, Mao KM, Bao YB, Yu G. Research advance on mapreduce based big data processing platforms and algorithms. Ruan Jian Xue Bao/Journal of Software, 2017,28(3):514−543 (in Chinese). http://www.jos.org.cn/1000-9825/5169.htm

Research Advance on MapReduce Based Big Data Processing Platforms and Algorithms

SONG Jie[1], SUN Zong-Zhe[1], MAO Ke-Ming[1], BAO Yu-Bin[2], YU Ge[2]

[1](Software College, Northeastern University, Shenyang 110819, China)

[2](School of Computer Science and Engineering, Northeastern University, Shenyang 110819, China)

Abstract: This paper introduces the research advance on MapReduce based big data processing platforms. Frist, twelve typical MapReduce based data processing platforms are descripted, their implementation principles and application areas are compared, and their commonalities are concluded. Second, the MapReduce based big data processing algorithms, including search algorithms, data cleansing/transformation algorithms, aggregation algorithms, join algorithms, sorting algorithms, optimization algorithms, preference query algorithms, graph algorithms, and data mining algorithms, are studied. These algorithms are classified by their MapReduce implementations, and the factors that affect their performance are analyzed. Finally, big data processing algorithms are abstracted as the out-of-core algorithms whose performance features are well analyzed. The considerations, ideas and challenges of universal optimizations on the performance of out-of-core algorithms are proposed as references for researchers. These optimizations include optimizing algorithms' I/O cost and locality, and designing incremental iterative algorithms. Comparing the current topics, such as resource allocation

* 基金项目: 国家自然科学基金(61672143, 61433008, 61402090, 61502090)
Foundation item: National Natural Science Foundation of China (61672143, 61433008, 61402090, 61502090)
收稿时间: 2016-08-01; 修改时间: 2016-09-14; 采用时间: 2016-11-01; jos 在线出版时间: 2016-11-29
CNKI 网络优先出版: 2016-11-29 13:35:10, http://www.cnki.net/kcms/detail/11.2560.TP.20161129.1335.011.html

and task scheduling based dynamic optimizations on platform, parallelization for specific algorithms, and performance optimizations on iterative algorithms, the proposed static optimizations serve as complements that highlight new areas for the researchers.
Key words: big data; MapReduce; out-of-core algorithm; big data processing; performace optimization on algorithms

近年来,伴随着信息技术、互联网和物联网技术的不断发展,数据采集终端迅猛增加,人们步入信息爆炸的大数据时代.正如麦肯锡所说:“数据,已经渗透到当今每一个行业和业务职能领域,成为重要的生产因素.人们对于海量数据的挖掘和运用,预示着新一波生产率增长和消费者盈余浪潮的到来.”在大数据时代,商业、经济及其他领域中的决策将不再基于经验和直觉,而是基于大数据分析结果,因此,大数据分析处理技术已经成为一个重要的研究和应用领域.同时,业界对于该技术的急切需求以及云计算技术的成熟,促使各种基于大规模分布式系统的大数据处理平台和处理算法如雨后春笋般涌现[1].大数据处理采用分治法,将大数据问题分解成规模较小的子问题求解,然后合并子问题的解,从而得到最终解.基于此,Google 公司研发的 MapReduce 是一种专门处理大数据的编程模型和实现框架,具有简单、高效、易伸缩以及高容错性等特点[2].

Google MapReduce 在设计之初致力于通过大规模廉价服务器集群实现大数据的并行处理,它优先考虑系统的伸缩性和可用性,用于处理互联网中海量网页内容数据,通过存储、索引、分析以及可视化等处理步骤,实现用户对网页内容的搜索和访问[2].MapReduce 在一个简单的库中隐藏分布式执行、容错、数据分发、任务调度以及负载均衡等难点,隐藏远程数据访问、节点失效和任务间通信等细节.而 MapReduce 之所以能够迅速成为大数据处理的主流计算平台,得力于其自动并行、自然伸缩、实现简单和支持商用硬件等特性[3].现如今,MapReduce 已是成熟的 TB/PB 级大数据处理平台广泛地应用于社交网络、科学数据分析、传感器数据处理、医疗和电子商务应用中,并拥有各种不同版本的实现.本文第 1 节将分析和对比典型 MapReduce 大数据处理平台,介绍其优劣势以及适用范围.

大数据处理平台是一种计算平台,计算平台泛指支持算法执行的硬件系统、操作系统和运行库[4],那么大数据处理平台则泛指可以支持大数据处理算法执行的平台.MapReduce 平台广泛地支持大数据处理算法,包括数据清洗、排序、统计分析、连接查询、图分析、PageRank、分类、聚类、最优化、机器学习、自然语言处理算法等.MapReduce 为上述算法提供了编程模型和分布式并行的运行环境[5].大数据处理算法是以大数据为输入,在给定资源约束内处理数据,并计算出给定问题结果的算法[6].大数据处理算法读写数据时间长、数据难以放入内存、待处理的数据无法存储在一台机器上,因此多为外存算法[7].外存算法是指算法所处理的数据过大而无法一次放入内存,必须借助外存反复读写数据的算法.外存算法不再基于无限大内存这一假设,不采用随机读写内存的数据访问方式,而通过传输大规模连续的数据块来平摊巨大的 I/O 代价[8].大数据处理算法的研究现在仍旧处于初始阶段,现有研究多集中在如何采用 MapReduce 改写传统算法、特定算法的优化这类问题上,对算法自身的、且具有一定普适性的优化研究较少[8,9].本文第 2 节将基于 MapReduce 编程模型,按 Maps 算法、Reduces 算法和迭代算法这 3 种分类来例举大数据处理算法,并分析影响算法性能的因素.本文第 3 节总结外存算法模型,并提出此类算法的优化思路,重点从算法磁盘 I/O、算法局部性和增量式迭代算法这 3 个角度阐述.

综上所述,基于 MapReduce 技术,大数据处理平台采用集群系统作为硬件环境,分布式中间件作为数据存储和计算平台.采用无共享体系结构,数据处理程序部署在每个节点之上.数据保存在分布式文件系统中.而大数据处理算法以大数据作为输入,在大数据处理平台上执行.在算法执行阶段,平台将算法分解为一种或多种类型的任务,任务的实例会分发到多个节点上并行执行,多节点上并行执行的实例相互独立,实例间不存在远程调用.实例会访问本地或远程节点的数据.大数据处理算法分解为多种任务,每种任务都视为一个外存算法,每个节点上运行的实例都视为执行态的外存算法.本文将综述基于 MapReduce 的大数据处理平台和大数据处理算法,分析它们的性能特征;与传统的优化任务间逻辑关系、资源分配、数据布局、任务调度等动态优化方法不同,本文将从外存算法角度提出静态的算法性能优化思路和挑战.

1 大数据处理平台

由于 MapReduce 具有简单、易伸缩性及高容错的特点,对大数据具有高效批处理能力,Yahoo,Facebook,Amazon 和 IBM 都将 MapReduce 作为大数据处理平台.Apache 研发的 Hadoop MapReduce 最为流行,并以此引出完善的 Hadoop 生态圈.作为 Hadoop MapReduce 的良好补充,业界和学界针对不同设计目标,实现了多种 MapReduce 平台.本节将分析对比典型的 MapReduce 大数据处理平台,如 Hadoop[10],GridGain[11],Mars[12],Phoenix[13],Disco[14],Twister[15],Haloop[16],iMapReduce[17],iHadoop[18]和 PrIter[19]以及类似 MapReduce 的 Dryad[20],Spark[21].

1.1 平台描述

(1) Hadoop

Hadoop[10]能够很好地支持 Java 语言编写的 MapReduce 作业,通过 Hadoop Streaming 或 Hadoop Pipes 工具,也能支持 C/C++或其他语.

Hadoop 为大规模并行数据处理算法提供运行环境,其工作原理为:将作业分解成更小的任务,将数据进行分区,每一个任务实例处理一个不同的分区,任务实例并行执行[22],这充分体现了分治的思想.

Hadoop 把 MapReduce 作业分解成顺序执行的 Map 阶段和 Reduce 阶段,Map/Reduce 阶段包含一个 Map/Reduce 任务,Map/Reduce 任务的实例(简称实例)部署到 Map/Reduce 节点并行执行,当所有 Map/Reduce 实例执行结束后 Map/Reduce 阶段才结束.MapReduce 程序仅包含两个函数,即 Map 函数和 Reduce 函数,它们定义了用户处理键值对数据的 Map 任务和 Reduce 任务.程序的输入数据集位于分布式文件系统中,采用迁移运算而非迁移数据的方式,Map/Reduce 任务被下载到每个数据节点并执行,输出结果仍保存在分布式文件系统中[23].

Map 和 Reduce 函数的输入和输出都是用户定义格式的键值对形式.Map 函数输入 $\langle Key_M^{In}, Value_M^{In}\rangle$,输出 $\langle Key_M^{Out}, Value_M^{Out}\rangle$,Reduce 函数的输入为 $\langle Key_R^{In}, Value_R^{In}\rangle$,输出为 $\langle Key_R^{Out}, Value_R^{Out}\rangle$,其中,$Key_M^{Out}$ 需要隐式地转换为 Key_R^{In},$Value_R^{In}$ 是保存 $Value_M^{Out}$ 的集合.

MapReduce 将作业(job)分解为任务(task)并在每个节点上并行地执行.MapReduce 程序首先将输入数据分割成 M 份(通常 M 大于节点个数),作为 M 个 Map 实例的输入.Map 函数从一份输入中读取每一条记录,对其进行必要的过滤和转换,然后输出 $\langle Key_M^{Out}, Value_M^{Out}\rangle$ 格式的中间结果.这些中间结果被 Hash 函数按 Key_M^{Out} 分割为 R 个不相交的组,每个组被写入到处理节点的本地磁盘中.所有 Map 函数终止时,M 个 Map 实例把 M 份输入文件映射成 $M\times R$ 个中间文件.由于所有 Map 函数的分割函数都一样,因此,相同键 Hash 值(设为 j)的 Map 函数输出结果被存在文件 $F_{ij}(1\leqslant i\leqslant M,1\leqslant j\leqslant R)$中.

MapReduce 的第 2 阶段执行 R 个 Reduce 实例,R 通常是节点的数量.每个 Reduce 实例 R_j 输入文件为 F_{ij} $(1\leqslant i\leqslant M)$.这些文件从各个节点通过网络传输汇聚到执行节点.Reduce 函数输入 F_{ij} 的键 Key_M^{Out} 和对应的一组 $Value_M^{Out}$ 值,并向分布式文件系统输出若干 $\langle Key_R^{Out}, Value_R^{Out}\rangle$ 格式的记录.所有的从 Map 阶段产生的具有相同 Hash 值的 $\langle Key_M^{Out}, Value_M^{Out}\rangle$ 输出条目均被相同的 Reduce 实例处理.

输入数据集以集合的形式存在于分布式文件系统中的一个或多个分区中.MapReduce 的调度器决定执行多少个 Map 实例以及如何把它们分配给可用的节点;同样,调度器还必须决定运行 Reduce 实例的数量和执行位置(Hadoop 早期版本让用户指定 Map 和 Reduce 实例的数目).MapReduce 中央控制器协调每个节点上的运算,一旦最终结果以新数据的形式写入分布式文件或数据库系统中,MapRedcue 作业执行完毕.MapReduce 作业的执行过程如图 1 所示.

除支持 MapReduce 外,Hadoop 的最新版本 YARN[24]提供了强大的资源管理功能.YARN 将 JobTracker 分解为两个独立的服务:全局的资源管理器 ResourceManager 和应用程序特有的 ApplicationMaster.其中,ResourceManager 负责整个系统的资源管理和分配,而 ApplicationMaster 负责单个应用程序的管理.YARN 采用 Master/Slave 架构,在整个资源管理框架中,ResourceManager 为 Master,NodeManager 为 Slave,ResourceManager

负责对各个 NodeManager 上的资源进行统一管理和调度.当用户提交一个应用程序时,需要提供一个用以跟踪和管理该程序的 ApplicationMaster,它负责向 ResourceManager 申请资源,并通知 NodeManger 启动任务和占用资源.由于不同的 ApplicationMaster 被分布到不同的节点上,因此它们之间不会相互影响.因为资源管理器 ResourceManager 和应用程序之间去耦合,所以 YARN 具有更好的可扩展性、更加高效,大量不同组件能够有效共享同一个框架.

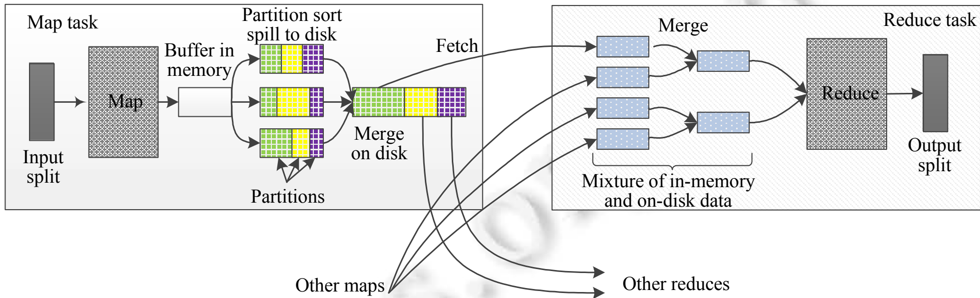


Fig.1 Execution process of job on Hadoop MapReduce

图 1 Hadoop MapReduce 作业执行流程

(2) GridGain

GridGain[11]是另一种 MapReduce 的开源实现,GridGain 仅支持 Java 语言,它与 Hadoop DFS 相兼容.GridGain 提供了一种分布的、基于内存的、实时和易伸缩的数据网格,将数据源和各类数据处理程序连接起来.GridGain 与 Hadoop 相比较最大的差异在于:前者是一个内存版本的 MapReduce,而后者能够支持超大规模数据集的处理.两者的具体差异在于:① GridGain 的作业仅包含单一的 Reduce 实例,因此在 Reduce 阶段,GridGain 不具备并行性;② GridGain 的 Map 任务仅返回单一值,若要返回多个值,则需要将多个值封装为集合;③ GridGain 对 Map 任务输出的中间结果不排序,也不会合并;④ GridGain 的作业管理策略灵活,用户可以随意创建和终止作业,作业的输入和输出格式也较 Hadoop 更为自由;⑤ GridGain 的 Map 任务强制在节点本地执行,本地性强,任务执行效率高,网络 I/O 的开销小,但容易导致木桶效应,在同步时,执行快的节点会等待执行慢的节点.综上所述,GridGain 和 Hadoop 在 MapReduce 模型上并无本质差异,两者不同之处主要在于任务调度、接口定义和附功能细节.

(3) Mars

Mars[12]是一种基于 GPU 的 MapReduce 框架,能够在 GPU 上正确、有效、简易地执行数据密集型和计算密集型作业.Mars 简洁的编程接口向用户隐藏了 GPU 编程的复杂性,并且能够自动地实现 CPU 和 GPU 的任务分割、数据分发、并行化和线程管理.Mars 在 GPU 内启动大量的线程,负载均衡地分配任务实例至每线程,每线程执行一个 Map/Reduce 实例.Mars 以小数量的键值对数据作为任务实例的输入,并通过一种无锁的方法来管理多任务实例对数据的并发写操作.

Mar 作业执行的各个阶段如图 2 所示,包括 Map,Group 和 Reduce 这 3 个阶段,每个阶段对应多个任务实例,调度器将 Map 和 Reduce 实例调度至 GPU 上运行,并将结果返回给用户.

① 首先,数据存储在磁盘中,在 Map 阶段之前,Mars 会对磁盘数据进行预处理,在主存中将输入数据转换成键值对格式;

② 接着,调度器初始化线程配置,定义 GPU 线程组的数量和每个线程组中线程的数量;

③ 随后,Map 阶段开始,MapSplit 将输入数据由内存调度至 GPU 线程,平衡所有线程的负载,每个线程执行 MapCount 函数来计算 Map 实例输出的中间结果的数量(种类)和规模,并统计出一个局部直方图,然后在局部直方图上执行前缀和(prefix sum)函数,获得每个线程的输出大小以及写入位置.每一个 GPU 线程执行用户自定义的 Map 函数并且输出中间结果至缓存(内存).正因为所有线程的输出位置是经过预

计算而确定的,因此不会产生写冲突,也不需要并发读写锁机制;

④ 下一个阶段是 Group 阶段,该阶段可对中间结果进行排序和 Hash 分组,也可以实现先分组再组内排序;

⑤ 在 Reduce 阶段,ReduceSplit 将具有相同 Key 值的分组调度到同一 GPU 线程,同 Hadoop MapReduce 一样,这种方式也会导致负载不均衡.

由此可见,Mars 和 Hadoop 对 MapReduce 作业的执行方式基本相同,但前者可以以最小的代价快速部署代码至分布式环境,并充分整合 GPU 资源.

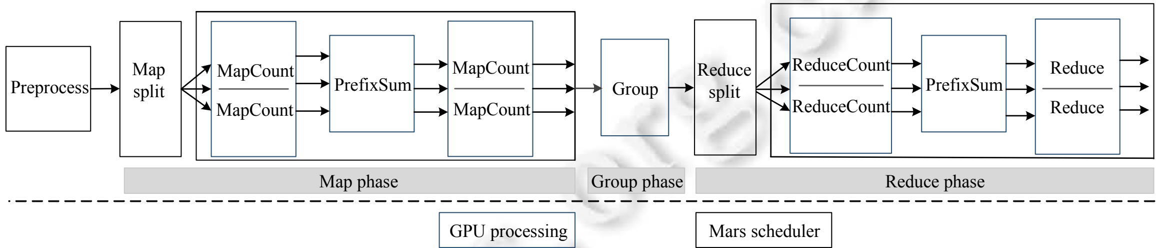


Fig.2 Phases of a job executing on Mars

图 2 Mars 作业执行的各个阶段

(4) Phoenix

Phoenix[13]的 MapReduce 实现方法和 Hadoop MapReduce 基本相同,其特点在于 Phoenix 是一种共享内存的分布式计算平台,该特点能最小化任务分发和数据通信所导致的网络 I/O 代价.Phoenix 采用纯 C++编写,提供 C/C++的应用程序接口.Phoenix 适用于多核或多处理器系统,能够对用户屏蔽多核编程、并行化、资源管理以及自动容错等功能的复杂性[25].Phoenix 通过共享内存的多线程执行 Map/Reduce 实例实现并行数据处理,其原理如图 3 所示.

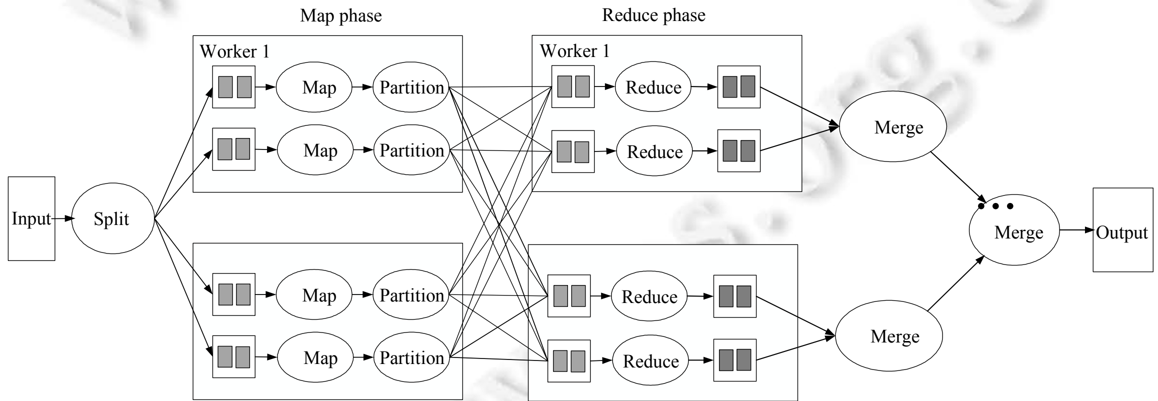


Fig.3 Data flow of Phoenix

图 3 Phoenix 数据流

用户首先编写 Map/Reduce 函数,并指定待处理的数据;随后,Phoenix 在多个 CPU 上启动多个线程.在 Map 阶段,输入数据会被划分成多个块,在每个块上调用 Map 函数处理数据,将键值对中间结果输出至内存.在 Reduce 阶段,将相同键对应的所有值传递给 Reduce 函数,Reduce 函数将输入约减为单一的键值对,所有 Reduce 实例的输出结果最终被合并、排序和输出.

(5) Disco

Disco[14]是 Nokia 研究中心研发的 MapReduce 轻量级开源实现,其核心组件采用 Erlang 语言开发,外部编程接口为 Python 语言.Disco 支持大数据集的并行处理,能运行在低可靠性的集群系统之上,也可以部署在多核计

算机、Amazon EC2 等云平台之上.

Disco 采用简单的主从结构,图 4 展示了主节点(master)服务器控制多台从节点(slave)服务器的系统架构.在 Disco 中:用户使用 Python 脚本启动作业;作业请求通过 HTTP 协议发送至主节点;主节点通过 SSH 启动每个从节点,从节点在 Worker 进程中运行 Disco 任务,从节点之间的通信也基于 HTTP 协议.

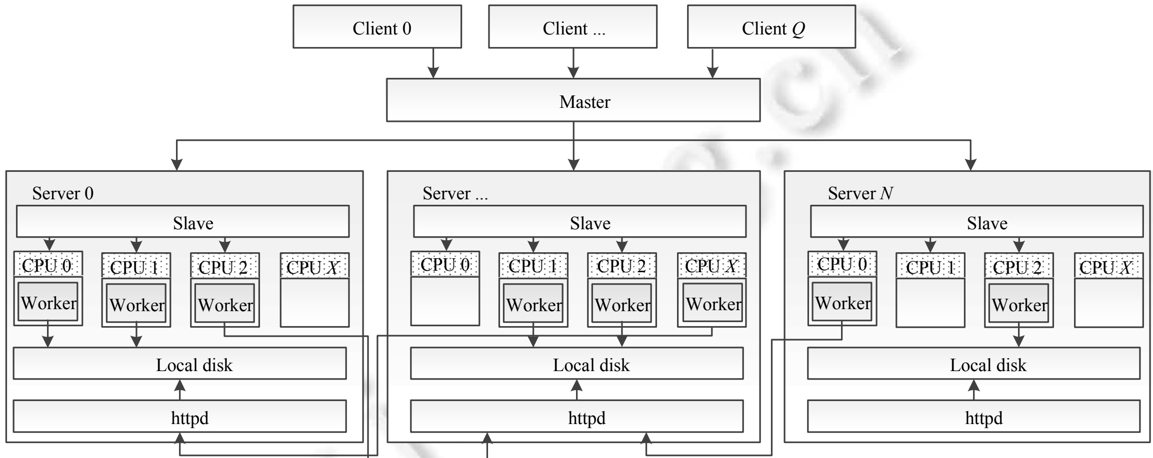


Fig.4 Disco architecture

图 4 Disco 架构图

(6) Twister

Twister[15]是一个可以支持迭代算法的轻量级 MapReduce 实现.Twister 的输入数据由两部分组成,即,静态数据和动态数据.一般的,静态数据比动态数据要大很多,静态与动态数据完成一轮运算后产生新的动态数据,然后,新动态数据与静态数据再次进行运算.如此循环,直到迭代终止条件满足,输出最终结果.Twister 修改 Hadoop MapReduce 使其适应迭代算法:① Hadoop MapReduce 一个作业仅能包含一对 Map-Reduce 任务,而 Twister 的作业可以包含多对 Map-Reduce 任务,减少了迭代算法执行时作业启动和关闭的代价;② Map/Reduce 任务静态数据采用内存或本地磁盘缓存,通过运行时任务的本地特性,采用合理的数据布局和任务调度,提高任务本地性,加快任务执行;③ 在 Reduce 任务后增加 Combine 任务,用以判断迭代结束条件是否满足,在作业内完成迭代控制.Twister 执行迭代算法的工作流程如图 5 所示.

Twister 不是自动地将大文件分割为数据块,而是依赖用户自定义的分割算法,任务实例输入的是数据块,因此,用户自定义的分割算法将影响任务的本地性和负载均衡.Twister 采用 Broker Network[26]存储迭代中间结果和动态数据,Broker Network 可以将 Map 实例的输出结果推送给 Reduce 实例.此外,Twister 还提供迭代中间结果的本地磁盘备份,当某个任务计算失败,则可以从最近的备份中恢复,避免重新迭代.

(7) HaLoop

HaLoop[16]是在 Hadoop MapReduce 基础上扩展的迭代计算框架.HaLoop 对 Hadoop 的扩展包括:① 编程接口更加适用于迭代算法;② 将原生的 Hadoop 中每一个作业对应一对 Map-Reduce 任务改成对应多对 Map-Reduce 任务,以复用作业;③ 框架实现迭代终止条件的检测;④ 任务尽量满足数据本地计算的特性,两次迭代任务尽量使用相同的数据,对没有变化的数据进行本地缓存,如增加了 Reduce 任务的输入缓存和输出缓存;⑤ 增加索引机制以优化数据访问.但是 HaLoop 的动态和静态数据无法分离,且没有一个客观的停止迭代的标准.图 6 为 HaLoop 的架构图,图中清晰地标明了 HaLoop 扩展 Hadoop 的模块以及新增模块.

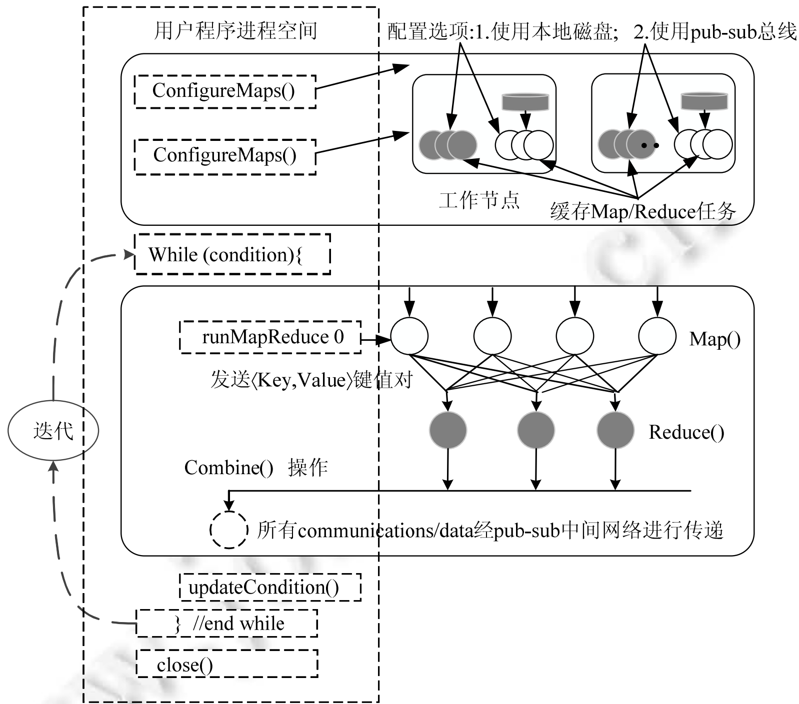


Fig.5 Execution flow of iteration on Twister

图 5 Twister 迭代执行流程

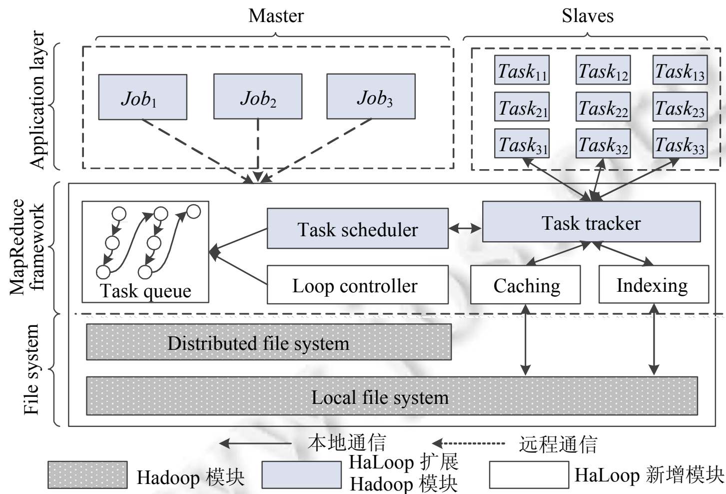


Fig.6 HaLoop architecture

图 6 HaLoop 架构图

(8) iMapReduce

iMapReduce[17]是一种基于 Hadoop MapReduce 的迭代计算模型和实现.由于 MapReduce 采用批处理模型,在执行迭代计算过程中存在 3 个重要开销,即反复的作业调度开销、反复的数据加载和传输开销、反复的任务同步开销.iMapReduce 改进了传统 MapReduce 的批处理模型,它针对上述不足,将 Reduce 任务的处理结果回传给 Map 任务开始下一轮的计算,避免反复的作业调度开销,通过维护本地静态数据来避免反复的加载传输静态

数据的开销,并在一次迭代内允许异步执行 Map 任务来避免反复的任务同步开销.iMapReduce 可以有效提升迭代算法性能.iMapReduce 同时提供迭代收敛检测、容错、负载均衡等功能,提供类似于 Hadoop 的编程接口方便用户实现迭代算法.图 7 比较了 Hadoop MapReduce 处理流程与 iMapReduce 处理流程.

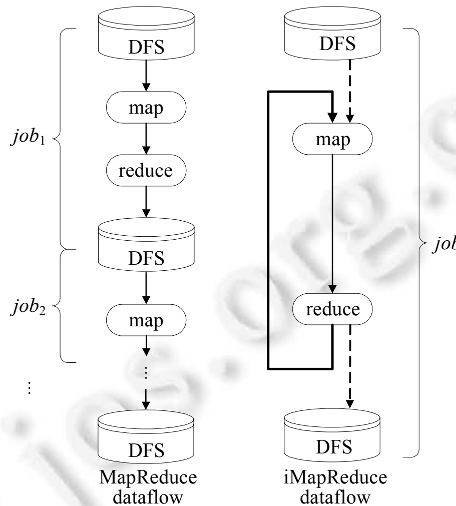


Fig.7 Difference on processing flow of Hadoop MapReduce and iMapReduce

图 7 Hadoop MapReduce 处理流程与 iMapReduce 处理流程比较

(9) iHadoop

iHadoop[18]是一个类似于 iMapReduce 的迭代计算平台,同样也扩展了 Hadoop 平台.首先,iHadoop 实现了 MapReduce 的异步迭代;其次,iHadoop 没有静态数据和动态数据的组合和管理,所以更容易调度.其调度器提供一种数据定位机制,从而减少数据的冗余传输以及磁盘 I/O 和网络资源的浪费,但是随之而来的代价是无法避免传输和处理静态数据的开销.

(10) PrIter

PrIter[19]是基于 Hadoop 的,支持优先级的迭代计算框架,能够保证迭代的快速收敛,适合交互式查询需求.PrIter 摒弃了传统的迭代模型,提出优先级迭代概念.部分迭代算法的执行过程中,少数数据单元的更新将对迭代算法收敛起决定性作用,而大多数数据单元的更新对收敛的贡献很有限.利用这个特点,优先级迭代对数据单元加以区分,让那些对算法收敛作用更大的数据单元执行更频繁的更新运算,而忽略那些无关紧要的数据单元.如图 8 所示,PrIter 在 Hadoop 中的 Reduce 任务内部加入了优先级计算执行引擎,StateTable 里维护了各计算单元键值对的优先级信息.PrIter 支持多个优先级迭代算法,包括 PageRank,SSSP,WCC,Adsorption 等优先级迭代算法,并且支持 Top-k 结果的在线反馈、收敛检测、负载均衡和容错控制等重要功能.

(11) Dryad

Dryad[20]的设计目标同样是为了降低大规模分布式编程的难度,为用户提供一个简单通用的大数据处理平台.Dryad 没有采用 MapReduce 模型,而是提供一种通用的、粗颗粒度的、批处理式的计算模型.Dryad 的核心数据模型由 Vertex 计算节点和 Channel 数据通道两部分组成,用户自定义 Vertex 节点来实现运算逻辑,而节点之间通过各种形式的数据通道传输数据,用户的运算逻辑通常是顺序执行的,而分布式逻辑则由 Dryad 框架实现.Dryad 模型和平台架构如图 9 所示.

Dryad 和 MapReduce 的概念模型十分相似,不同之处在于:MapReduce 较细粒度地将数据处理逻辑分为 Map 和 Reduce 两个阶段,而 Dryad 粗粒度地提供不分阶段的 Vertex 模型.MapReduce 强制定义了 Map 和 Reduce 两个任务以及两任务之间的数据输入/输出格式,优势在于程序设计人员可以通过套用 Map/Reduce 任务来抽象

自身的运算逻辑,简化了用户编程接口,降低编程难度,也有利于 MapReduce 框架对任务的管理.缺点在于固定编程模型一定程度上限制了 MapReduce 的通用性,比如 MapReduce 模型中,所有的计算节点只能接受统一格式的一组输入数据,也只能输出一组数据,无论是否需要,用户逻辑都必须由匹配的 Map 和 Reduce 任务组成.而 Dryad 则提供了更为灵活的编程模型以及完善的任务管理和容错机制.

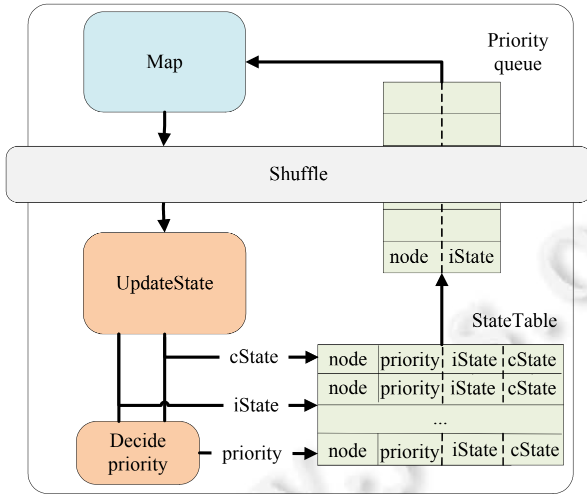


Fig.8 PrItre architecture
图 8 PrIter 架构图

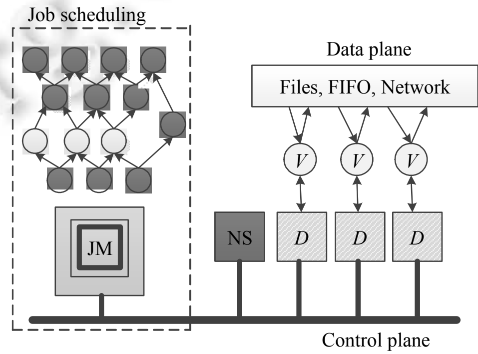


Fig.9 Dryad architecture
图 9 Dryad 架构图

(12) Spark

Spark[21]是轻量的、基于内存计算的开源集群计算平台.Spark 采用了与 MapReduce 类似的编程模型,并且支持 Java,Scala,Python,Spark Shell 等多种编程语言.为适用不同的应用场景,Spark 存在多种运行模式,如单节点的本地运行模式、单机的伪分布运行模式、基于 Standalone Deploy 的分布式模式、基于 Hadoop Yarn 或 Mesos 的分布式模式.其中,前两者主要用于开发和调试,后两者则用于生产环境.运行模式的差异主要在于资源申请和管理,前两者由 Spark 自己申请并占用系统资源,后两者则是由 Spark 向 Hadoop Yarn 或 Mesos 的中央资源调度器申请资源,并在使用完后归还.

在 Spark 中,数据被高度抽象且存储在弹性分布式数据集(resilient distributed dataset,简称 RDD)中.RDD 支持粗粒度写操作,对于读操作,RDD 可以精确到每一条记录,这使得 RDD 可用作分布式索引.在 Spark 中,所有的操作被称为算子.Spark 不仅实现了 MapReduce 的类 Map 函数算子和类 Reduce 函数算子,还提供了更多丰富的算子,如 Filter,Join,Groupbykey 等算子.Spark 主要将算子分为两类:一类是 Transformation 算子,另一类为 Action 算子.Transformation 主要为数据项的转换操作,而 Action 算子负责数据的汇总和保存等操作.此外,Spark 还实现了任务调度、RPC、序列化和数据压缩.相对于 Hadoop MapReduce,Spark RDD 可以缓存到内存中,每次 RDD 数据集的操作结果都可以存储在内存中,下一个操作可以直接从内存中读取数据,节省了大量磁盘 I/O 操作.Spark 对于迭代运算效率明显.

Spark 架构如图 10 所示.Spark 作为计算平台可以运行多种资源管理器,如 Spark 自备的管理器、Mesos 系统、Hadoop Yarn 等.近年来,由于 Spark 的研究热度高,相继涌现了多种基于 Spark 的大数据处理平台.例如,Spark SQL 为支持类 SQL 语句的数据管理平台[27],GraphX 为类似于 Pregel 的图计算平台[28],Streaming Spark 为类似于 Storm 的流计算平台[29],MLLib 为机器学习平台[30].Spark 引入了 RDD(resilient distributed dataset)模型,中间数据都以 RDD 的形式存储,而 RDD 分布存储于从节点的内存中.对比 Hadoop MapReduce,Spark RDD 可以缓存到内存中,每次 RDD 数据集的操作结果都可以保存至内存,下一个操作可以直接从内存中读取数据,省去了 MapReduce 中大量的磁盘 I/O 操作,明显提升迭代算法中常见的机器学习算法和交互式数据挖掘算法的性能.

所以,Spark 更适合迭代算法.

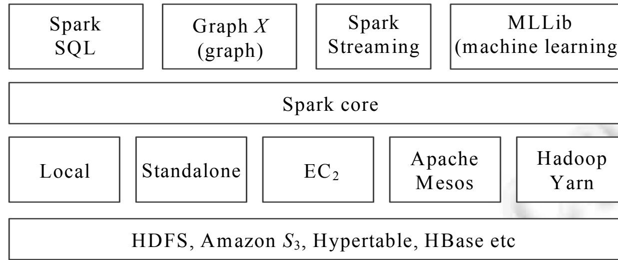


Fig.10 Spark architecture
图 10 Spark 架构图

1.2 平台对比

12 个典型的 MapReduce 大数据处理平台各具特点,适用于不同的硬件环境,如多线程、多核、GPU、商用机集群、内存计算.开发人员可以根据实际需求选择合适的平台,研究人员也可以针对它们的优缺点展开具体研究.大数据处理平台可以按多种角度分类:按照平台规模以及开发复杂度可以将平台分为轻量级与重量级平台;按照平台基于的计算模型可以分为 MapReduce、迭代 MapReuce、类似 MapReduce 这 3 种;按照其支持语言有 Java,C,C++,Scala,Erlang,其中,基于 Java 实现的平台较多.由于设计目标、用途和实现方式的差异,所以优缺点各不相同,具体对比见表 1.

Table 1 Comparison of the big data processing platform
表 1 大数据处理平台比较

名称	量级	计算模型	支持语言	用途	优点	缺点
Hadoop[10]	重	MR	Java	大数据集上的数据密集型和计算密集型任务	同时提供数据存储和计算能力,伸缩性好,适合超大数据集的分析处理	集群可用性差,缺少安全模型,对小文件支持差,需要全局同步,处理小规模数据的速度不一定比串行程序快
GridGain[11]	重	MR	Java	基于内存的大数据处理	对任务无特定要求,适用性好.可以在网格上执行	不支持任何非 Java 应用;只提供了分布式计算支持,没有分布式文件系统,Reduce 阶段前缺少数据预处理
Mars[12]	轻	MR	C++	利用 GPU 进行大数据处理	支持 GPU;允许在单机上利用不同的处理器;在 Map 和 Reduce 阶段之前存在两次预处理过程	GPU 线程不支持动态调度;不支持运行时内存分配空间;易造成写错误;预处理操作昂贵
Phoenix[13]	轻	MR	C/C++	基于内存的大数据处理	独立运行,无需提前部署;利用共享内存缓冲区实现通信,避免因数据复制产生开销	不能自动执行迭代算法;无高效的异常处理机制;对内存数据结构敏感;较差的伸缩性
Twister[15]	轻	迭代 MR	Java	迭代算法,大数据处理	有效支持迭代算法,提供数据管理工具;在 Combine 阶段,收集所有 Reduce 实例输出结果,用户可以通过本地磁盘访问数据	任务调度机制不如 Hadoop 有效;需要把大数据文件分成多个小文件
Disco[14]	重	MR	Erlang, Python	HTTP 协议下的大数据处理	采用轮询的通信机制,通过 HTTP 的方式传输数据,适用于 Web 环境	轮询时间间隔难以确定,降低算法执行性能
HaLoop[16]	重	迭代 MR	Java	Hadoop 的迭代计算优化版本	更好的支持迭代算法,减小作业开销,增加迭代终止条件的判定	静态数据和动态数据不能完全分离,导致无效 I/O;模型较复杂,抽象程度不高

Table 1 Comparison of the big data processing platform (Continued)

表 1 大数据处理平台比较(续)

名称	量级	计算模型	支持语言	用途	优点	缺点
iMapReduce[17]	重	迭代MR	Java	Hadoop的迭代计算优化版本	迭代处理模型优化,避免反复的作业调度开销;优化的动态数据和静态数据管理;避免反复的数据加载和传输开销;支持任务的异步执行,避免同步开销	要求Map实例和Reduce实例数量一样,Map阶段和Reduce阶段绑定,调度缺乏灵活性
iHadoop[18]	重	迭代MR	Java	Hadoop的迭代计算优化版本	支持Map任务和Reduce任务的异步执行,减少同步代价	缺少静态数据和动态数据的组合管理,无法避免静态数据的传输和处理开销
PrIter[19]	重	迭代MR	Java	适用于部分迭代算法的迭代计算	高效的优先级调度可以加速迭代收敛,通过理论推导保证算法执行的准确性	部分算法存在优先级,因此支持算法有限
Dryad[20]	轻	类似MR	Java	大数据处理可高度自定义处理算法	简化大规模分布式编程的难度,提供给用户一个简单通用的分布式运算框架	由于任务形式自由,因此任务管理并不高效
Spark[21]	轻	类似MR	Scala, Java Python, R	基于内存的大数据处理,迭代计算	可与Hadoop完整结合;保证容错的前提下,内存存储数据,数据访问速度变快	数据分区能力有限,各台机器计算任务分配不平均,负载不均衡

2 大数据处理算法

大数据处理的核心是数据处理算法,数据处理算法是一个广泛的概念,查询、统计、聚集、连接、排重、分析、挖掘、优化算法都可以视为数据处理算法[31].本文按照这些算法的 MapReduce 实现方法分类,将其分为 Maps 算法、Reduces 算法和迭代算法:Maps 算法仅包含 Map 任务(一个或多个),Reduces 算法包含至少一对 Map-Reduce 任务,迭代算法则是一组或多组 Map-Reduce 任务对反复执行直至收敛.本节分别介绍上述算法,并且在每一类算法的小节部分,介绍 MapReduce 是如何保证该类算法的性能以及影响算法性能的关键因素.

2.1 Maps算法

Maps 算法仅包含 Map 任务,算法将执行至少一个 Map 任务,每个 Map 任务将处理逻辑应用在数据集之上.常见的 Maps 算法包括搜索算法、数据清洗/变换算法等.

(1) 搜索算法

搜索算法是一种简单的 Maps 算法,采用单个 Map 任务实现.典型的为线性搜索(linear search)算法[32],Map 任务检查每条数据是否匹配查询条件:若匹配,则找到该条数据;若扫描整个数据集结束仍未匹配,则搜索失败.线性搜索按遍历数据集的方向,又可以分为正向搜索和逆向搜索.除此之外,二分搜索(binary search)算法也可以仅采用单个 Map 任务实现,但是它适用于频繁查询而不频繁改变的有序列表.二分搜索虽以递归形式定义,但是尾递归可改写为循环,因此可以用 Map 任务实现.内存版本的线性搜索算法时间复杂度为 $O(N)$,二分搜索算法的时间复杂度为 $O(\log N)$,而外存版本的线性搜索算法 I/O 复杂度为 $O(N/B)$,二分搜索算法的 I/O 复杂度为 $O(\log_B N)$,其中,B 为块大小,N 为数据量[33].

(2) 数据清洗/变换算法

大数据分析往往需要相对高昂的硬件成本和时间成本,因此,学界更多关注分析平台和算法,但数据质量同样对处理结果的精度产生影响[34].数据清洗/数据变换都是数据预处理方法:数据清洗能够去除数据中的噪声和无关数据、填充缺失值;数据变换则是将数据变换为适合处理算法的格式[35].数据清洗/变换算法是大数据处理

算法中较为简单的算法,其特点是逐条对数据按清洗/变换规则处理.用 MapReduce 实现的数据清洗/变换需要多个 Map 任务,每个清洗/变换规则都由一个或多个 Map 任务来实现.文献[35]中,通过减少 Map 任务的个数以及 I/O 读写的次数对数据清洗算法进行优化.

(3) 算法小结

Maps 算法的时间复杂度多为线性,因此,影响性能的主要因素是磁盘 I/O 次数和节点数量.处理数据量越少,参与计算的节点越多,Maps 算法的性能越好.MapReduce 通过移动处理逻辑至数据端,并且并行地处理数据.由于数据处理逻辑是独立的,因此不需要同步操作,处理节点相互等待的情况不会发生.因此,Maps 算法的并行性很好.由于没有 Reduce 任务,因此 Maps 算法不能将各个节点的计算结果聚集,所以此类算法的适用范围较窄.

2.2 Reduces算法

Reduces 算法的 MapReduce 实现中至少包含一对完整的 Map-Reduce 任务,数据经过处理、汇聚、分组、排序、约减等操作,最后获得处理结果.常见的 Reduces 算法包括聚集算法、连接算法、排序算法、偏好查询等:

(1) 聚集算法

聚集算法将一组数据聚集成一个或多个值(分组聚集).求和、极值、均值都是简单的聚集算法.聚集算法分为分布、代数和整体这 3 种.设数据被划分为 n 个集合,聚集算法在每一部分上计算得到一个聚集值,如果将算法用于 n 个聚集值得到的结果与将算法用于所有数据得到的结果一样,那么该聚集算法是分布的.如果聚集算法能够由一个具有 M 个参数的代数函数计算,而每个参数都由分布聚集算法求得,那么该聚集算法是代数的.描述整体的聚集算法的子聚集所需的存储没有一个常数界,也即不存在一个具有 M 个参数的代数聚集算法计算得出[36].分布的和代数的聚集算法都可以采用 MapReduce 模型实现,属于 Reduces 算法;而整体算法则无法采用 MapReduce 实现,但整体算法可以采用近似计算的方法,进而转化为代数的或是分布的聚集算法[37].

实现单词词频统计的 WordCount 算法[38]是一种典型分组聚集算法.以 Hadoop MapReduce 实现为例,词频统计发生在 Map,Shuffle 以及 Reduce 这 3 个阶段:Map 阶段首先将文本分词,形成键值对的中间值,其中,键为单词本身,值为 1;Shuffle 阶段将中间值按照键进行排序,使所有具有相同键的值分组存储;在 Reduce 阶段对数据进行聚集,将具有相同键的值进行加和,输出结果.算法流程如图 11 所示.聚集算法有很多,大部分的实现方法与 WordCount 算法类似,本文不再赘述.

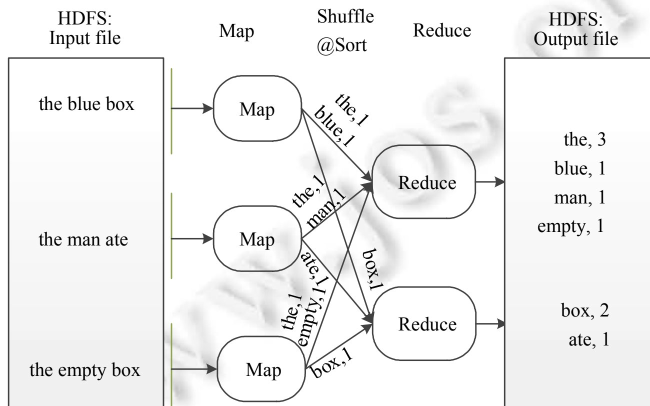


Fig.11 MapReduce implementation of WordCount

图 11 WordCount 算法 MapReduce 实现

(2) 连接算法

连接运算把两个或多个数据集中的记录按条件组合为一个结果数据集[39].连接运算在数据分析中非常常见,TPC-H 提供的 22 个查询用例中,有 16 个涉及到连接运算.连接运算有多种方式.

- 内连接结果集仅包含满足条件的行,是大部分数据库系统默认的连接方式.根据所使用的比较算法不同,内连接又分为等值连接、自然连接和不等连接 3 种;
- 交叉连接的结果集包含两个数据集中所有行的组合,又称笛卡尔连接;
- 外连接的连接结果集中既包含那些满足条件的行,也包含其中某个数据集的全部行,有 3 种形式的外连接:左外连接、右外连接、全外连接.

在上述连接种类中,最为常用的是等值连接和自然连接.

Reduce-Join 是连接操作基于 MapReduce 的经典实现:对于给定的两个数据集 x 和 y,Map 任务分别读取两个数据集的各个部分,从每个记录中按查询条件(C_x 和 C_y)抽取连接属性值($x.a$ 和 $y.a$),作为 Map 任务的键输出,这样,具有相同连接属性值的记录就会汇总到一个 Reduce 实例中,从而在 Reduce 任务中对两个数据集的记录进行笛卡尔积运算,完成连接过程.Reduce-Join 具有普适性,当 x 和 y 数据集都很大时,会产生大量网络 I/O[40].此外,在 Reduce-Join 连接算法基础上提出了基于半连接的 Semi-Join 算法.Semi-Join 在连接操作之前提取出用于连接的连接属性数据集($x.a$ 和 $y.a$),然后用此数据集对需要参加连接的数据集进行过滤,从而减少传输数据.半连接的实现方式很多,可以采用普通的 HashSet 来存储连接属性数据集;也可以采用 BloomFilter 技术,用连接属性数据集过滤数据.Semi-Join 需要额外的 MapReduce 作业来完成半连接,如果连接属性数据集的记录数和原数据集的记录数相差无几,那么 Semi-Join 的优势将不会明显.排序合并连接(sort-merge join)算法[41]将输入的两个数据集分别进行排序并划分区间,生成相应的桶,每个桶对应一个 Reduce 实例;然后,Reduce 实例从每个 Map 实例中读取桶,每实例读取的桶是同一个区间内的数据,利用 Reduce 任务进行合并.排序合并连接算法的 MapReduce 实现如图 12 所示.排序合并连接较 Reduce-Join 性能更好[42].哈希连接算法将排序合并连接中的区间划分改为哈希划分,然后,Reduce 实例从每个 Map 实例中读取桶,连接桶内记录.文献[43]中,在多核处理器上改善了哈希连接算法的性能.

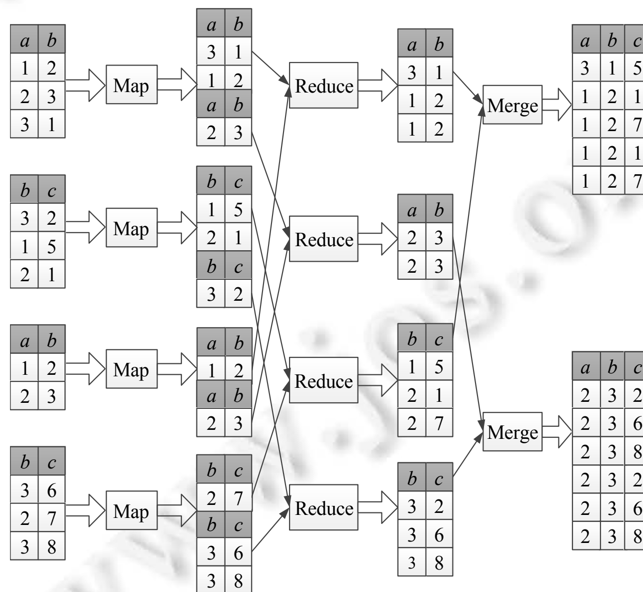


Fig.12 MapReduce implementation of sort-merge join

图 12 排序归并连接算法 Mapreduce 实现

并非所有的连接算法都为 Reduce 算法,Map-Join 是基于 MapReduce 的、仅采用 Map 任务的连接算法.Map-Join 是 Reduce-Join 的一种改进,减少了 Reduce 任务,从而消除了数据从 Map 任务进入 Reduce 任务的网络传输过程,但需要在 Map 任务开始前增加一个分发任务.对于给定的两数据集 x 和 y,当数据集 y 较小时,系统将 y 分

发至每一个节点内存,或分发至分布式缓存中供每个节点访问.这样,在每个 Map 任务就可以参照 y 完成数据连接.当 x 数据量很大,而 y 数据量很小时,Map-Join 方式的效率非常高;但是当 y 的数据量非常大时,分发阶段的 I/O 代价会很高.

除等值连接算法外,朴素 MapReduce 相似连接算法同样采用 MapReduce 编程模型实现[33].如图 13 所示:算法将两个数据集通过 Map 任务映射到相应的不同键值上,然后对它们的组合进行合并.在 Reduce 任务中完成连接,判定相似性是否大于阈值,然后把大于阈值的结果输出.基于 MapReduce 的相似连接算法还有多重集合 MapReduce 相似连接算法[33].

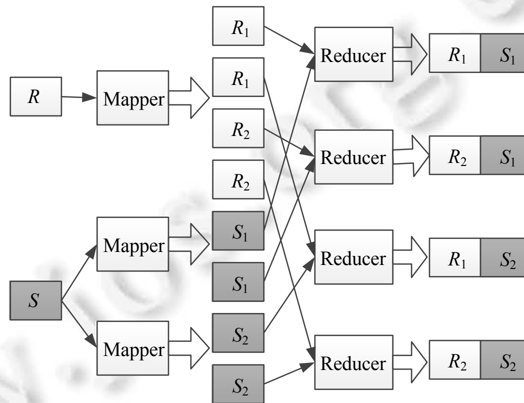


Fig.13 MapReduce implementation of similarity join

图 13 朴素相似连接算法的 Mapreduce 实现

更多 MapReduce 连接算法请见文献[40].在本节中,主要就 Map-Join,Reduce-Join,Sort-Merge Join 这 3 种连接方式的适用场景以及优缺点进行比较分析,具体分析见表 2.

Table 2 Comparison of the different join algorithms

表 2 不同连接算法比较

名称	适用场景	优点	缺点
Map join	两个待连接表中,一个表非常大,而另一个表非常小,小表可以直接加载到内存中	Join 算子执行在 Map 端,无需经历 Shuffle 和 Reduce 等阶段,效率高	数据集有序,不同数据集 Partition 方式一致.若数据规模较大,需对多个参数进行调优,否则可能会造成内存不足
Reduce join	两个数据集非常大,难以将某一个存放到内存中	相对于 Map Join,不需要每个 Map 任务实例都去读取所有的数据集,降低计算量	Shuffle 阶段要进行大量的数据传输,效率低.数据规模较大,需对多个参数进行调优,否则可能会造成内存不足
Sort-Merge join	数据集已排序	较 Reduce Join 性能更好,可用于不等值连接(如<,≤,>,≥)	排序合并连接执行效率不如 Map-Join,不适合 OLTP 类型的系统

(3) 排序算法

MapReduce 可以实现大部分排序算法,例如 MapReduce 的 Reduce 任务溢出文件合并过程默认使用的就是归并排序.但有一种排序算法是专门为 Hadoop MapReduce 设计的,这就是 TeraSort[44]算法.TeraSort 是一种海量数据集上的排序算法.2008 年,Terasort 用时 209s 完成了 1TB 的排序,赢得 Sort Benchmark 的桂冠;2009 年,TeraSort 在 1 460 节点的集群上用 62s 完成了 1TB 数据的排序;2013 年,TeraSort 在 2 100 节点的集群上将 1.42TB 数据的排序时间降至 1m.Terasort 算法中主要由 3 步组成:采样、Map 任务标记数据、Reduce 任务局部排序.采样在 Map 阶段开始之前由客户端完成,客户端首先从输入数据中抽取一部分数据,将这些数据进行排序,然后将它们划分成 r 个数据块,找出每个数据块的数据上限和下限(称为分割点),并将这些分割点保存到分布式缓存中;Map 阶段,每一个 Map 实例把数据分为 r 个块,其中,块内数据满足“第 i 块数据要大于第 i+1 块数据”,这是通过构建 Trie 树实现的,树的叶子节点上保存有该叶子对应的 Reduce 实例编号;Map 实例首先从分布式缓存中读

取按分割点建立的Trie树,对于每条数据,Map实例在Trie树中查找它所属的Reduce实例编号并保存;在Reduce阶段,每个Reduce实例从每个Map实例中读取其对应的数据进行局部排序,最后将Reduce实例处理后结果按Reduce实例编号依次输出即可.TeraSort算法流程如图14所示.

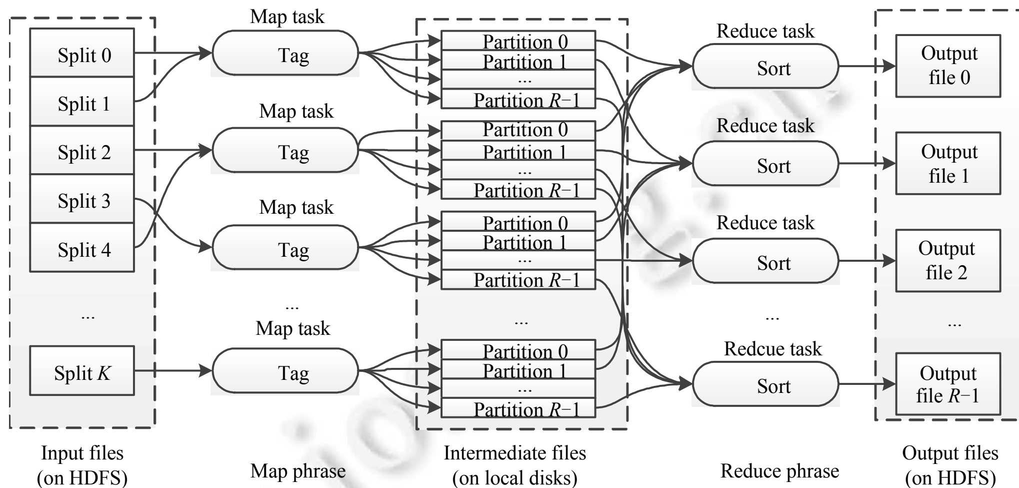


Fig.14 MapReduce implementation of Terasort

图14 Terasort的MapReduce实现

(4) 偏好查询

Top-*K*算法根据给定的评分函数在潜在的海量数据中返回*k*个分值最高的元组.Top-*K*的MapReduce实现是一种Reduces算法.在Map阶段,首先把数据Hash后分配到不同的节点上,每一个节点负责处理不同的数值范围;在Reduce阶段,选出各个节点中的前*K*个元组,然后进行汇总.文献[45]考虑到MapReduce的特点,首先采用水平划分的方式对数据进行划分,将数据以数据块的形式存入HDFS中,然后提交查询;主节点利用元数据对区间进行筛选,确定参与最终计算的数据块;最后,MapReduce任务只对涉及到的区间进行计算,返回Top-*K*值.对Top-*K*算法而言,其保存前*K*个值,则算法时间复杂度为$O(n\times\log k)$.除此之外,文献[46]介绍了不确定性Top-*K*查询.Top-*K*查询框架如图15所示.

Skyline查询的主要目标是在给定的数据集中搜索不被其他数据对象支配的数据对象[47].一个数据对象支配另一个数据对象指的是:该数据对象在所有维度都不比另一个数据对象“差”,并且至少有一个维度比另一个数据对象“优”.这里,“优”和“差”的定义需要根据具体标准而定.采用MapReduce实现Skyline的方法很多,最基本的是块嵌套循环算法(map-reduce based block-nes ted loops,简称MR-BNL)[48]:首先将输入数据划分为若干块,每个块传入Map实例,得到块内的Skyline查询结果;随后进行排序和分组后,发送给Reduce任务,任务接收到所有中间结果后产生最终Skyline查询结果.针对MapReduce Skyline优化算法很多,例如,延迟Skyline查询算法的基本原理是:在所有Map实例完成后,将自身的局部过滤值发送给Master节点;Master节点在接收到所有Map实例的局部过滤值之后产生全局过滤值,再将其发送给每个Map实例;每个Map实例接收到全局过滤值之后,运用该全局过滤值对自身结果进行过滤,并产生各自Skyline查询结果;Reduce任务以过滤后的中间结果为输入,产生最终的Skyline查询结果.贪婪Skyline查询算法的基本原理是:当一个Map实例结束后,将其局部过滤值发送给Master节点并获得一个全局过滤值,如果这个全局过滤值为空,那么该Map实例将其所有Skyline查询结果输出;否则,该Map实例用这个从Master节点获得的全局过滤值对自身Skyline查询结果进行过滤[49].

(5) 算法小结

Reduces算法中其主要包含聚集算法、连接算法、排序算法以及偏好查询算法.与Maps算法不同的是:在Reduces算法经过Map任务产生的数据仅是中间结果,需要Reduce任务对每一份中间结果进行合并处

理.Reduces 算法的基本数据流总结如图 16 所示.

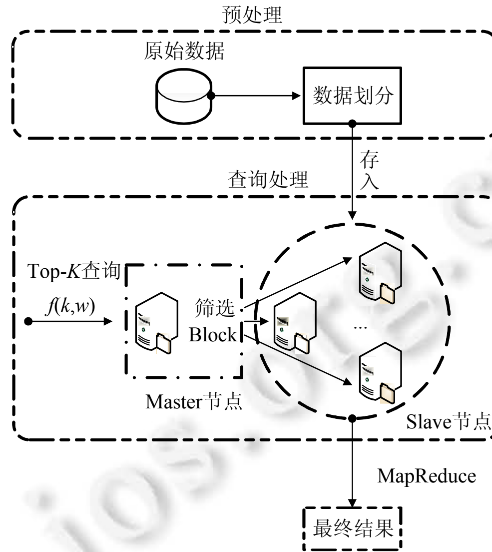


Fig.15 MapReduce implementation on top-K query

图 15 Top-K 查询的 MapReduce 实现

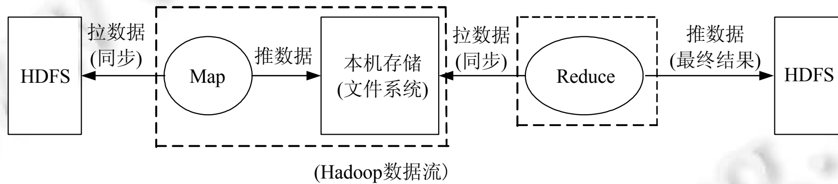


Fig.16 Data flow of Reduces algorithm

图 16 Reduces 算法的数据流

Reduces 算法鲜有线性算法,除算法本身的复杂程度、磁盘 I/O 次数和节点数量因素,性能还与以下 4 个因素有关.

① Reduce 实例的并行性.由于 Reduce 实例的并行性部分取决于节点数量,又部分取决于 Map 任务输出数据的特征,因此,合理设计 Map 任务的数据分组,让分组数量远大于 Reduce 节点数量,并且负载均衡地将每一组数据分发给 Reduce 实例,可以提高 Reduce 阶段执行性能;

② Map 任务和 Reduce 任务同步带来的性能损失.由于 Reduce 任务要等待所有 Map 任务结束才能开始,因此,该同步过程将浪费计算资源.如果能通过调度算法提高 Map 任务的并行性,减少木桶效应,或者采用异步执行的方式解除 Reduce 任务对 Map 任务的等待,将会有效提高数据处理性能;

③ 算法与数据结构的设计,Reduces 任务中涉及了大量的排序、Hash 分组、判定等算法,高效的算法和数据结构,如 TeraSort 算法中的 Trie 树,能够有效提高算法性能;

④ 从算法设计角度,尽量减少 Reduces 算法不可并行的部分,例如可以将整体聚集算法近似为可以并行执行的代数聚集算法,可以充分利用分布式并行计算能力,优化算法性能.

对于 MapReduce 平台,除提供 Map 任务和 Reduce 任务的数据传递和并行执行,因素①和因素②也是平台保证 Reduce 算法效率的核心设计.而对于 MapReduce 算法,因素③和因素④是其执行性能的关键.

2.3 迭代算法

迭代计算是应用相同的计算逻辑根据某一初值反复处理同一个输入数据的过程,而这一初值也在迭代计算中不断精准,逐步接近于最优解.迭代计算广泛存在于数据挖掘和机器学习算法中.迭代算法的 MapReduce 实现多包含一组或多组反复执行的 Map-Reduce 任务对,通过不断重复用 Map/Reduce 任务处理数据,来逼近目标或结果.迭代数据分为不变的静态数据和在迭代过程中频繁改变的动态数据.利用 MapReduce 实现迭代算法需要从以下 3 个方面考虑.

① 确定迭代变量.至少应有一个直接或间接地不断由旧值递推出新值的变量,该变量就是迭代变量;

② 建立迭代关系式.迭代关系式的建立保证了能够从当前变量推出下一个变量,该关系式需可并行,可用 Map/Reudce 任务实现;

③ 控制迭代过程,确定迭代过程终止条件和终止时机,以及如何在 Map/Reduce 任务中加入该条件.

MapReduce 并不原生支持迭代算法.编程人员通过程序设计,精心地布置 Map/Reduce 任务,同时编写一些特殊的程序来支持迭代算法.此外,很多大数据处理平台都针对迭代算法特点进行优化,一般称为迭代计算平台.常见的迭代算法包括最优化算法、图算法、数据挖掘算法等.

(1) 最优化算法

优化算法通过不断进行迭代运算,处理数据以获得最优解.文献[50]介绍了模拟自然进化过程搜索最优解的并行遗传算法.遗传算法是根据适者生存、优胜劣汰的遗传机制演化而来的随机搜索方法,并行遗传算法的迭代过程不能直接用一对 Map/Reduce 任务表达,因此需要在每次迭代的最后阶段添加一个全局选择阶段,具体实现方法是,在迭代后添加第 2 次 Reduce 任务判断结果是否满足条件.同时,算法通过一个协调器来协调迭代过程.

如图 17 所示:首先,Coordinator 产生后代并进行变异,然后其将后代发送到 Master 以供评价和选择.Master 将后代分割成 m 个独立的组,并将组分配给 m 个 Mapper,其中,m 的值大于机器的数量,每组后代与 Mapper Worker 一一对应.Mapper Worker 遍历该组的所有后代,产生中间结果保存在本地磁盘.第一 Reduce 阶段中,Reducer Worker 都会被分配到 Reduce 实例,Reduce 函数用来选择局部最优的数据,并将其保存至本地磁盘;第二 Reduce 阶段,Reducer Worker 收集前一 Reduce 阶段产生的结果,Reduce 函数用于产生全局最优的结果作为最终结果,该结果被发回 Coordinator,作为下一次迭代的输入.通常情况下,遗传算法时间复杂度为 $O(n^2)$.

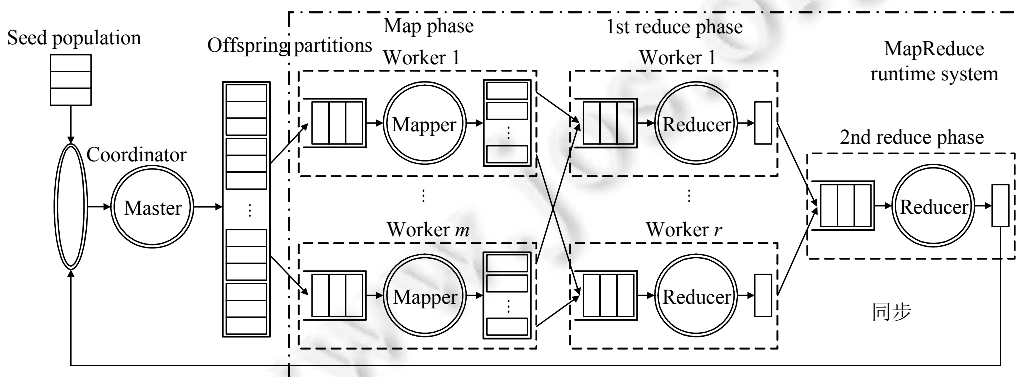


Fig.17 MapReduce implementation on genetic algorithm

图 17 遗传算法的 MapReduce 实现

类似的迭代最优化算法还有粒子群优化算法[51],该算法从随机解出发,不断进行迭代来寻找最优解,同时通过适应度评价解的品质.它没有遗传算法的交叉和变异操作,由当前搜索到的最优值来寻找全局最优.该算法具有简单、收敛快等优点.此外,文献[52]介绍了一种新兴启发式算法——布谷鸟搜索算法,有效地求解最优化问题.布谷鸟搜索算法的实现结合分而治之的算法思路和 MapReduce 编程模型,实验结果表明:在处理大数据集

时,该算法具有更好的性能[53].

(2) 图算法

图算法也是迭代算法中比较常见的算法,包括图查询、图索引、图匹配、可达性算法等.由于 MapReduce 没有提供获得全局状态或全局数据的机制,而只能在 Map 任务和 Reduce 任务之间传输局部信息,所以图算法的 MapReduce 实现都遵循以下模式:首先计算节点信息,然后将更新数据传递到它的邻居再次计算,多次迭代后,收敛至最终结果.MapReduce 适用于处理稀疏图,中间结果数量与节点的邻居节点数目成正比;而对于稠密图,若仍然使用 MapReduce 实现算法,则需要设计更为成熟的数据划分方法,比如最小生成树算法或图割算法[33].典型的图算法包括 PageRank 算法、Descendant-Query 算法、Dijkstra 算法、最小生成树算法.

PageRank[54]是 Google 创始人于 1997 年构建早期的搜索系统原型时提出的链接分析算法,目前很多重要的链接分析算法都以 PageRank 算法为基础.PageRank 现用于标识网页的等级/重要性,在揉合了诸如标题标识和关键字标识等因素之后,Google 通过 PageRank 使那些更具等级/重要性的网页排在搜索结果的前面,从而提高搜索质量.PageRank 基于两个前提,给定网页 A,$L(A)$表示链入 A 的网页集合,$B(A)$表示 A 所链接的网页集合, $R(A)$表示 A 的 PageRank 值,则:① 数量前提,$|L(A)|$越大,即链入 A 的网页越多,那么 A 越重要;② 质量前提,$L(A)$中,页面自身的质量不同,质量高的页面,在计算 $R(A)$时的权重越高.所以越是质量高的页面指向 A,则 A 越重要.基于以上两个前提,PageRank 计算过程如下:

① 初始化.网页通过链接关系构建出 Web 图,在图中,每一个网页是一个节点,网页之间的链接关系作为节点之间的有向边,每个页面设置相同的 PageRank 初始值;

② 更新.在一轮更新计算中,Map 任务将每个页面的 PageRank 值平均分配到本页面的链出页面上,这样,当前页面每个链接获得了相应的贡献度;Reduce 任务将每个页面所有链入页所传入的贡献度求和,即可得到新的 PageRank.当每个页面都更新了 PageRank 值,就完成了一轮 PageRank 计算;

③ 终止.当更新过程中 PageRank 的值不发生变化时该算法终止,并得到最终的 PageRank 值[55].

Descendant-Query 是社交网络中的常用算法.Descendant-Query 基于用户关系图,通过迭代的方式计算某个用户所相识的所有人.算法抽象描述为:假定用户的样本为 U,在 U 中包含了所有直接相识用户所组成的序偶.即 A 和 B 是两个直接相识的用户,则$\langle A,B\rangle\in U$.与 A 所有相识的用户集合为 R.Descendant-Query 算法过程表示如下.

① 初始化:$R=\{\langle A,A\rangle\}$;

② 更新:通过 R 与 U 的连接运算以及去重复运算,增加 R 内元素,更新所得的结果即为下一轮迭代初始值 R;

③ 终止:当 R 不再发生变化时,该算法运行终止[56].

BSP 模型(bulk synchronous parallel model)是一种基于消息通信的并行计算模型[57].一个 BSP 作业由若干个串行执行的超步 $S_1,S_2,...,S_n$ 组成,对应于 n 次迭代.并行任务按照超级步组织,在超步 S_i 内,各任务异步接收来自 S_{i-1} 的消息,执行本地计算并发送消息给下一个超步 S_{i+1}.在下一个超步执行之前,通过显式地同步确保所有任务均已完成上一个超步.该同步方式可避免死锁和数据竞争.文献[58]对比 BSP 和 MapReduce 在图处理算法上的优缺点.

① 在执行机制方面:MapReduce 是一种数据流模型,每个任务只是对输入数据进行处理,产生的输出数据作为另一个任务的输入数据,并行任务之间独立地进行,串行任务之间以磁盘和数据复制作为交换介质和接口.而 BSP 是状态模型,各个子任务在本地的子图数据上进行计算、通信、修改图的状态等处理,并行任务之间通过消息通信交流中间计算结果,不需要像 MapReduce 那样对全体数据进行复制;

② 在迭代处理方面:MapReduce 模型理论上需要连续启动若干作业才可以完成图的迭代处理,相邻作业之间通过分布式文件系统交换全部数据.BSP 模型仅需启动一个作业,利用多个超级步就可以完成迭代处理,两次迭代之间通过消息传递中间计算结果.由于减少了作业启动、调度开销和磁盘存取开

销,BSP 模型的迭代执行效率较高;

③ 在数据分割方面:基于 BSP 的图处理框架需要对加载后的图数据进行一次再分布的过程,以便于消息通信时路由地址的确定.例如,各任务并行加载数据过程中,根据一定的映射策略,将读入的数据重新分发到对应的计算任务上(通常是存放在内存中),既有磁盘 I/O 又有网络通信,开销很大.但是一个 BSP 作业仅需一次数据分割,在之后的迭代计算过程中除了消息通信之外,不再需要进行数据的迁移.而基于 MapReduce 的图处理模型,一般情况下不需要专门的数据分割处理,但是 Map 阶段和 Reduce 阶段存在中间结果的 Shuffle 过程,增加了磁盘 I/O 和网络通信开销.

(3) 数据挖掘算法

K-Means[59]算法是典型的基于距离的聚类算法,采用距离作为相似性的评价指标,即认为:两个对象的距离越近,其相似度就越大.该算法认为簇(cluster)是由距离靠近的对象组成的,因此把得到紧凑且独立的簇作为迭代目标.*K*-Means 算法抽象为:假定训练样本为 $X=\{x^{(1)},x^{(2)},\ldots,x^{(m)}\}$,其中,$x^{(i)}\in R^n(i\in[1,m])$,且要将样本聚类成 k 个簇,算法过程如下.

① 初始化.随机选取 k 个聚类质心点(cluster centroids),$\mu_1,\mu_2,\ldots,\mu_k$,其中,$\mu_i\in R^n(i\in[1,k])$;

② 更新.给定任意样例 $i(i\in[1,m])$,根据 $c(i)=\min(\|x(i)-u_j\|^2),j\in[1,k]$,Map 任务计算其所属于的分类 $c^{(i)}$,给定任意类别 $j(j\in[1,k])$,Reduce 任务根据 $\mu_i'=\frac{\sum_{j:c(j)=i}x^{(j)}}{|c(j)|}$ 更新该类别的质心 μ_j;

③ 终止.在更新过程中,质心点不在发生变化时,算法终止.

K-Means 运行过程如图 18 所示:在图中一共有 $A\sim E$ 这 5 个数据以及 2 个随机的质心点(灰色点).图 18(a)为算法经过初始化后的状态,图 18(b)和图 18(d)是两个更新过程,图 18(c)和图 18(e)是两个更新后的状态.经过 2 轮更新该算法达到稳定如图 18(e)所示.文献[60]中,在传统的 *K*-means 基础之上提出了一种 *K*-means 集群优化算法,通过减少迭代的次数以及提高每次迭代的速度来提高算法效率.*K*-means 集群优化算法与传统算法比较有较高的处理速度和稳定性,*K*-Means 聚类算法的时间复杂度是 $O(nkt)$,其中,n 代表数据集中对象的数量,t 代表着算法迭代的次数,k 代表着簇的数目.同样对 *K*-means 算法进行优化的论文还有文献[61,62].类似基于 MapReduce 的聚类算法还有层次聚类算法、FCM 聚类算法和 SOM 聚类算法.决策树算法是最经典的分类算法.例如,ID3 可以通过反复执行一对 Map-Reduce 任务实现决策树的构建,其中,

- Map 任务计算属性之间的信息增益,并按〈属性名,信息增益〉的键值对形式输出中间结果;
- Reduce 任务汇总中间结果,并选择信息增益最大的属性构建决策树:如果当前属性是决策树的叶子节点,则算法结束;否则,需要对数据执行分裂操作,重新按属性将数据集划分,并将子集数据传递给下一次迭代轮的 Map 任务.

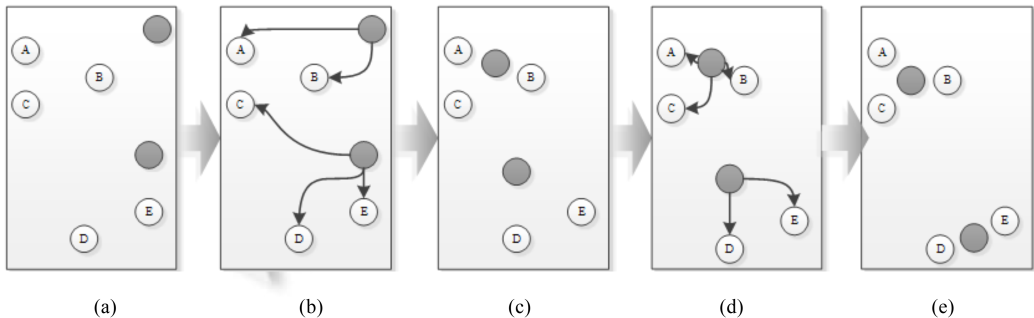


Fig.18 *K*-Means example

图 18 *K*-Means 算法过程示例

支持向量机 SVM[63]算法同样采用迭代的方式对数据进行分类,通过寻求结构风险最小化来提高模型的泛

化能力,实现经验风险和置信范围之和最小化,从而达到在统计样本量较少的情况下,也能获得良好统计规律的目的.

数据挖掘算法可以通过机器学习算法包实现.本文将其分为分布式与非分布式算法包,同时,在分布式算法包中又可以将其分为基于 MapReduce 的和非 MapReduce 的算法包.本节将介绍基于 MapReduce 的机器学习算法包.Mahout 是 Apache Software Foundation 旗下的开源项目,提供一些可扩展的机器学习领域经典算法的实现,它提供了用于建立可伸缩算法的简单和可扩展的编程环境和框架,对迁移到 Hadoop 上的应用程序或从 Hadoop 上剥离成为独立应用程序的项目有着重要作用.Mahout[64]实现的算法包括:分类算法,如随机森林、支持向量机和贝叶斯算法;聚类算法,如 *K*-Means,Canopy 和 Mean Shift 算法;降维算法,如 Lanczos, Stochastic SVD 和 PCA(主成分分析)算法;主题模型中 LDA 算法;协同过滤算法,如 User-Based Collaborative Filtering,Item-Based Collaborative Filtering 和 SVD 算法;还有 Frequent Pattern Mining,Row Similarity 等其他机器学习算法.除 Mahout 之外,还有 Oryx2[65],利用 Oryx2 可以构建并部署基于 Hadoop 的机器学习算法,如回归、分类、聚类和协同过滤算法.随着数据的不断流入,Oryx2 还将支持自动更新.此外,H2O[66]提供了适合大数据处理的数据结构和机器学习算法,其同样能分类、聚类、深度学习等一系列算法.作为 MapReduce 机器学习算法库的良好补充,MLlib[30]是基于 Spark 的机器学习算法库,其特点是规模大和性能好,同样包含大量常见的算法和数据类型.MLlib 支持 Java,Python 和 Scala 等编程语言.

(4) 算法小结

在大数据处理算法中,迭代算法的实际应用更加广泛.除上述算法之外,HITS(超文本主题检索)、递归关系查询、神经网络分析、社会网络分析以及网络流量分析同样需要进行迭代算法.然而,MapReduce 框架并不直接支持迭代算法,因此需要精妙设计的多轮 Map/Reduce 任务,这会引入一些新的问题:① 静态数据及在迭代过程中保持不变的数据如果无需在迭代任务间传递,能够最小化网络带宽;② 每一轮计算都会输出大量的中间数据,需要判断中间数据后才能决定迭代是否收敛,收敛判定程序无法以一种分布式的方式执行,而且需要每一轮运算节点同步后(运算结束)才可以进行收敛性检查,上述操作开销显著.

迭代算法的性能除算法本身的复杂程度、磁盘 I/O 次数和节点数量、并行性以外,还与以下 5 个因素有关:

① 静态数据的缓存效果.如何避免静态数据在任务间传递,是迭代算法性能优化的关键因素之一;

② 任务管理和调度的效率.由于迭代算法涉及大量任务的反复调度执行,因此,任务调度的成本和效果不可忽视;

③ 算法的局部性.迭代算法会反复访问数据,因此,数据访问局部性高的算法能够充分利用计算节点内存,减少磁盘 I/O;

④ 算法同步次数和收敛速度.迭代算法的每一轮内 Map 任务和 Reduce 任务都需要同步,且迭代轮和迭代轮之间也需要同步,同步会影响算法并行性,导致节点间等待.如果算法的收敛速度快,能够减少同步次数.算法的收敛速度与很多因素有关,如每轮迭代的复杂度、初始点的选择、终止条件的设计、任务调度等;

⑤ 由于 Reduces 算法可以视为迭代算法的一轮,因此,前节 Reduces 算法分析中提到的性能因素同样适用于迭代算法.

基于 Spark 的迭代算法较基于 Hadoop 的迭代算法更有性能优势[67],原因是前者支持内存计算而后者是磁盘计算.在迭代算法中,每轮迭代过程从磁盘中重新加载中间数据的 I/O 代价过高,因此,Hadoop MapReduce 并不适合迭代算法;而 Spark 具有全局缓存机制,在整个迭代过程中,Spark 把中间数据保存在内存中,在每轮迭代中,直接从内存读取数据,这极大提高了迭代算法的运行速度,但是需要大量内存空间.因此,当迭代算法是时间敏感性的,Spark 是一个很好的选择[21],但是需要有大量的内存空间保证.值得一提的是:通常情况下,我们很难估算也难以保证 Spark 运行迭代算法时需要的内存大小.因此,对于非实时的、面向大数据的迭代算法,Hadoop 依然具有优势.在实践中,还可以将 Spark 与 Hadoop 相互结合来提高算法运行效率和处理规模,如 Lamda 架构[68].

2.4 算法分析

本节按大数据处理算法的 MapReduce 实现特征,将其分为 Maps 算法、Reduces 算法以及迭代算法,并对 3 类算法进行了细分,例举若干典型算法的 MapReduce 实现.分析影响算法性能的因素,总结为以下几点.

- 算法复杂度:包括时间复杂度和空间复杂度,时间复杂度是算法的天然属性,不会因为并行化而改变;大数据处理算法为外存算法,数据无法一次性放入内存,因此,传统的空间复杂度难以度量算法性能;此外还应该考虑算法的 I/O 复杂度,因为 I/O 代价是外存算法的主要性能代价,I/O 复杂度表征算法读取磁盘块的次数和输入规模之间的关系;
- 算法处理数据方式:不同的数据处理算法模式不同,可以分为批处理式、交互式处理和流式处理,MapReduce 适用于批处理式的算法,也能胜任交互式处理,但难以实现流式处理算法.交互式处理算法对性能要求高,需要在用户可以接受的时间内响应用户请求,而批处理式算法是性能松散的;
- 算法访问数据频率:部分数据处理算法通过大量计算获得最终结果,对同一数据访问次数少甚至仅一次,数据访问频率低;反之,部分算法则需要反复读取同一数据,在连续读取过程中通过对数据进行分析比较,获得最终结果,数据访问频率高;数据访问频率会影响 CPU 的使用率、缓存失效率,进而影响算法性能;
- 算法并行性:可以定义为算法并行部分占算法的比例.阿姆达尔定理指出:系统某一部件由于采用某种更快的执行方式后,整个系统功效的提高与这种执行方式使用频率占总时间的比例有关.并行性高的算法更适用于 MapReduce 计算模型;
- 算法局部性(locality):指算法倾向于访问最近访问过的数据项本身或临近于该数据项的数据项[69].掌握算法局部特征,则有利于充分利用缓存减少外存访问,减少 I/O 代价,优化算法性能;
- 算法磁盘 I/O 代价:对于大数据处理算法,算法的磁盘 I/O 代价是算法性能的决定性因素;
- 算法网络 I/O 代价:对于大数据处理算法,所有 I/O 操作都可以归结为磁盘 I/O,但部分 I/O 操作还伴随着网络 I/O.以下 3 种情况大数据处理算法需要远程访问数据:① 输入数据集有一部分存储在远程位置;② 访问位于远程的全局数据;③ 访问其他节点的中间结果.在大部分硬件环境中,访问本地磁盘和访问远程磁盘的性能有着明显的差距;
- 数据源:对大数据处理算法的影响主要取决于是集中的还是分布的.若数据是集中存储的,或者是由外部数据源提供的,那么算法需要读入数据,并将数据分发至各个运算节点,带来大量的网络 I/O 代价;反之,若数据是分布存储的,视存储节点为运算节点,就可以将远程数据访问优化为本地数据访问;
- 数据布局(data placement):分布式文件系统中,数据布局是指数据在节点间按特定目标的分布状态[70],良好的数据布局可以提高数据处理算法性能.在 MapReduce 以及基于其分布式文件系统(如 Hadoop HDFS)中,数据布局有 3 个特点:数据是分布的;数据在节点之间复制;各个节点中的数据被并行地处理.数据布局决定了节点的数据特征(如数据量、值域分布等).由于迁移计算而非迁移数据,计算被移动到距离数据最近的节点上运行,不同节点执行的任务算法相同,因此,任务的执行特征取决于输入的数据量以及数据特征,也即数据布局决定 Map/Reduce 任务的分发方式和执行效率.由此可见:要想提高任务的执行效率和并行性,减少节点间的等待,就必须尽可能地实现负载均衡的数据布局;
- 同步次数:同步模型要求所有节点完成当前阶段后才可以开始下一阶段,这严重限制了计算性能.但同步是数据归约和汇总的前提,也是下一个阶段的开始时机.异步模型则可以使各个节点之间独立运行,加速算法执行,但无法有效地汇总中间结果,无法获得算法全局信息.算法执行过程时,同步次数越少越有利于算法的局部性能,但有可能导致算法整体执行步骤的增加;
- 缓存管理:缓存是减少磁盘 I/O、提高算法性能的重要手段.在大数据处理算法执行过程中,何种数据需要缓存、如何设计缓存、采用本地缓存还是分布式缓存、集中管理还是分散式管理,这些策略都会影响算法性能.缓存管理指定缓存大小、需缓存的数据、缓存保存的位置、缓存保存和清除时机、缓存生命周期、任务访问缓存的优先级等等;

- 任务管理:包括:① 任务调度,即调整同时执行任务的执行顺序;② 任务分发,即选择合适的节点执行任务实例;③ 资源管理,即为任务实例合理分配计算资源和.对于不同的算法,任务管理策略应不相同,有效的任务管理策略能够提高算法执行性能;
- 容错机制:容错技术会占用计算资源,在一定程度上影响算法性能;但另一方面,算法在具有良好容错机制的平台上执行时,能够快速从故障中恢复而不会导致执行失败,这本身就是一种性能优化.大数据处理平台的容错机制与算法性能息息相关,合理的容错机制是算法执行的保证;
- 增量计算支持:当新增数据或数据集发生改变时,大数据处理算法需在新的全集数据上重新处理而无法复用现有数据处理结果,这将浪费大量的计算资源.增量计算则是对处理结果的复用,是指算法能够通过新增数据集和已知的处理结果完成增量式的数据处理,可以很大程度上提高数据处理的效率.

针对 Maps 算法、Reduces 算法和迭代算法,本文对上述算法性能影响因素进行总结分析,具体见表 3.

Table 3 Comparison of the big data processing algorithms

表 3 大数据处理算法比较

	Maps 算法	Reduces 算法	迭代算法
Map/Reduce 任务	一个或多个 Map 任务,无 Reduce 任务	至少一对 Map/Reduce 任务	至少一对 Map/Reduce 任务反复执行
算法复杂度	算法简单直接,多为线性算法	算法复杂度低,算法容易实现	算法复杂度高,实现算法需要程序设计技巧
算法处理数据方式	批处理式和交互式	批处理式和交互式	批处理式
算法访问数据频率	一次访问数据	一次或少量次数访问数据,访问频率低	对静态数据反复访问,访问频率高
算法并行性	算法并行性高,很多算法可以 100%并行处理数据	算法具有天然的并行性,不可并行的部分仅占算法的小部分,无需并行化处理	算法可并行的部分通常只占算法的一部分,需要算法设计人员进行特定的并行化处理
算法局部性	由于对数据仅扫描一次,因此无法评价也无需考虑算法局部性	算法局部性较好,同一数据被同一任务重复访问,不存在不同任务,如 Map 任务和 Reduce 任务中,或不同节点先后访问同一数据的情况	算法局部性差,同一数据访问间隔长,如不同的迭代轮,且不同的运算节点会先后访问同一数据
算法磁盘 I/O 代价	相对于处理的大数据量,算法磁盘 I/O 较低,优化空间小	高磁盘 I/O,除访问被处理的数据外,平台的任务管理、对数据的 Hash,排序等操作也产生额外的磁盘 I/O 代价	高磁盘 I/O,数据被反复读取,平台的任务管理和迭代控制耗费大量磁盘 I/O
算法网络 I/O 代价	网络 I/O 代价很低,数据无需远程传递	网络 I/O 代价低,Map 节点输出的中间结果需经网络传输到 Reduce 节点	网络 I/O 代价高,迭代的静态数据和动态数据都会在网络中传递
数据源	分布存储	分布存储	集中存储或分布存储
数据布局	数据布局对算法执行性能影响较小	由于 Map 任务和 Reduce 任务之间的同步,原始数据的数据布局对 Map 阶段执行时间有显著影响,中间结果的数据布局对 Reduce 阶段的负载均衡有显著影响	数据布局随着迭代过程的改变而改变,每一轮迭代,动态数据都会重新布局,而静态数据的布局会影响每一轮迭代的性能
同步次数	无同步,每个节点独立地处理数据	Map 和 Reduce 阶段之间同步	Map 和 Reduce 阶段之间同步,每一个迭代轮之间均需同步
缓存管理	任务本地缓存	任务本地缓存,Shuffle 阶段的缓存能够提高算法性能	任务本地缓存,静态数据缓存,动态数据缓存,迭代中间结果缓存.合理的使用分布式缓存能够显著提高迭代算法性能
任务管理	任务管理策略对算法的性能影响较小	任务管理策略会影响算法性能,如负载均衡的 Map 任务分发能够提高其并行性,或将 Reduce 任务分发至中间数据所在节点	任务管理策略复杂,且贯穿整个迭代过程,需要为不同迭代算法,甚至不同迭代轮设计不同的任务管理策略
容错机制	采用任务副本的方式容错,容错机制对计算资源的占有量较小,对性能影响不大	采用任务副本的方式容错,同时也可以采用数据副本,Map 阶段的输出结果容错,同容错机制对计算资源的占有量较小,对性能影响不大	每一轮迭代采用检查点的方式容错,何种任务失效,以及失效的时机都会对整个迭代的性能产生不同的影响
增量计算支持	算法天然支持增量计算	部分算法支持增量计算	大部分算法均不支持增量计算,需要重新设计

3 外存算法优化思路

前文介绍了 MapReduce 大数据处理平台和处理算法,并且分析了算法执行过程中与性能相关的因素,为进一步优化算法性能提供依据.本节从 4 个层面提出算法优化思路、可供研究的问题以及现存挑战.算法可以分为内存算法和外存算法,前者假设算法处理的数据可以一次装入内存,而后者则不能.传统的算法分析和优化均基于内存计算模型,该模型有无限的存储,随机访问数据,且每条数据的访问代价是一致的.在大数据处理算法中,大部分算法需要对磁盘进行多次读写操作,即为外存算法.由于磁盘读写为外存算法的瓶颈,因此外存算法注重I/O能力,而非FLOPS的计算能力[7].图19的外存模型是分析外存算法时计算环境的抽象模型,包括一个无限空间的外存磁盘、一个空间为 M 的内存和一个 CPU.每次 I/O 读写操作都在外存和内存之间传输连续的数据块 B,B 是内存和外存数据传输时最小单位块的大小,显然,$1\le B\le M$.对于外存算法,算法性能以 I/O 量来度量,而 I/O 量以读写文件块 B 的次数来度量,忽略 CPU 运算和访问内存的时间.

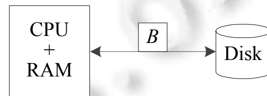


Fig.19 Model of out-of-core algorithm
图 19 外存算法模型

本节分析大数据处理算法的特性,并从外存算法的角度提出优化思路.优化思路并不针对特定算法,而是具有普适性,应用在特定算法时需要考虑算法特性.大数据处理算法以大数据作为输入,在大数据处理平台上执行;在算法执行阶段,平台将算法分解为一种或多种类型的任务;任务的实例会分发到多个节点上并行执行;每种任务都视为一个外存算法,每个节点上运行的实例都视为执行态的外存算法.大数据处理平台多采用资源管理和任务调度等动态性能优化方法,与其不同的是,本文从外存算法角度提出的性能优化是一种静态优化技术.静态优化方法,是在算法设计之时即被采用的,在算法运行时无法改变的优化方法.本节将从算法的磁盘I/O、算法的局部性和算法增量执行特性这 3 个角度提出优化思路和挑战,采用不同的算法结构和外存数据访问策略,降低算法运行时间.例如:从算法设计上减少磁盘 I/O,保证平台的资源利用率最高,降低因 I/O 阻塞带来的性能损失;将算法改写为局部性最优的算法,提高缓存命中率,减少外存访问等;减少算法的输入数据,仅将变化的数据作为输入,或用中间结果代替原始数据,避免数据的重复处理.

3.1 算法磁盘I/O优化

在大数据处理平台中执行外存算法,若运算和 I/O 之间能够良好地重叠,那么将减少 I/O 阻塞和 CPU 空闲,优化算法性能.基于这一思路,可研究如何通过算法设计减少 I/O,甚至略微增加运算量以降低 I/O 代价.外存算法的 I/O 代价主要来自于磁盘读写操作,且无论是远程数据读写还是本地数据读写,最终都会反映为磁盘 I/O.因此,可以从算法设计的角度降低磁盘I/O总量,本文提供以下研究思路.

- 首先,可研究如何从算法设计上减少磁盘 I/O.算法的磁盘 I/O 代价取决于如下因素:① 输入数据的数据量; ② 输出数据的数据量;③ 数据被访问次数.很多算法都需要反复访问同一数据,如连接算法和排序算法.基于比较的排序算法每个输入数据项要至少被访问 $\log N$ 次.重点研究如何优化因素①和因素③.因此,针对输入数据规模重新设计算法,研究如何利用采样(sampling)技术缩小数据规模,且满足算法精确性要求;或利用近似算法或概率算法改进,或采用更为高效的外存数据结构.针对数据访问次数,研究如何减少每次访问时的数据量.若算法在当前遍历数据时均计算概要信息以缩减数据,则可以在下一次遍历时减少磁盘 I/O.例如,统计分析算法中可采用频数统计的方法简化原始数据,图算法时可删除孤立节点并将子图替换为节点以缩小图规模;
- 其次,可研究数据压缩技术减少磁盘I/O提升性能.采用运算换存储的方法减少磁盘I/O可以降低CPU被动空闲.数据压缩适用于有中间结果输出的外存算法,以一种压缩格式将中间结果写入磁盘,并以解

压的方法读出数据.在密集的磁盘 I/O 操作时,CPU 大量空闲,数据压缩和解压带来的 CPU 运算量若小于因等待 I/O 时 CPU 被动空闲浪费的运算量,数据压缩和解压方法就有可能提高性能.那么,何种算法相关的数据需要压缩存储;如何确定压缩和解压的代价以及性能优化的收益;如何设计性能感知的压缩和解压算法;如何设计外存数据结构或文件结构;是否可以设计无需解压缩就可以读取文件的技术及该技术适用何种算法.这些都是可供研究的问题;

- 可研究如何从算法设计上减少远程磁盘访问.在大数据处理平台中,以下 3 种情况中算法需要远程访问数据:① 输入数据集有一部分存储在远程位置;② 访问位于远程的全局数据;③ 访问其他节点的中间结果.对于情况①,可通过数据布局或任务调度等方法减少远程数据访问,但这些技术并非面向算法本身,可以通过平台优化来实现,现有研究已经证明了这一点[44,71].对于情况②和情况③,则可以从算法设计角度加以优化.那么,如何调整算法结构,减少算法访问远程存储的中间结果,例如将同步迭代算法改为异步迭代计算;创建何种全局数据结构能够将全局数据或中间结果保存在内存中;如何利用分布式缓存技术,将单一远程节点的磁盘数据访问优化为多个节点的内存访问;如何将全局地址空间映射到本地磁盘或分布式内存的地址空间.这些都是可供进一步研究的问题.

算法磁盘 I/O 优化的现有研究很多,文献[72]介绍了基于 Lustre 文件系统的 Cannon 算法并行 I/O 优化,Cannon 算法是一个典型的用于二维网格矩阵乘法的分布式算法.然而,Cannon 算法在 Lustre 文件系统环境中 I/O 性能差,随着矩阵乘法规模的增加,I/O 代价成为影响算法性能的关键因素.文中提出一种连续条带式聚集模式(stripe-continuous aggregation pattern),充分考虑了 Lustre 文件系统的隔离机制和锁协议,提升 Cannon 算法的 I/O 性能.文献[73]提出了一种函数语言表达的代价模型,用以分析算法的缓存(内存)使用效率,指出算法实现采用的数据结构,如链表和树,会对内存效率产生影响.文献[74]采用分区的方法,在算法运行的同时即可输出部分结果,以此优化海量 Skyline Points 计算算法的 I/O 代价.文献[75−79]分别提出了一种 I/O 优化的矩阵运算算法、前缀树构建算法、格图(grid graph)上的算法、Skyline 查询算法和数据集连接算法,他们都是通过改变算法细节来优化算法 I/O.

3.2 算法局部性优化

若大数据处理平台中执行的外存算法能够尽量使用高速存储,那么将降低 CPU 因 I/O 阻塞而导致的被动空闲时间,优化算法性能.多级存储模型自上而下分为寄存器→缓存(L_1~L_3)→内存→磁盘→网络,每一级存储相对处理器的距离、读写速度、价格均是递减的,而容量是递增的.一个局部性优化的算法执行时,CPU 所访问的存储单元都趋于聚集在一个较小的连续区域中.由于局部性的作用,从算法的执行效果看,层次化存储体系的读写速度接近寄存器的读写速度,但容量是整个外存的容量.算法局部性优化能够有效地减少访问低速存储的可能,缩短 CPU 等待数据读写的时间,优化算法性能.因此,可以从外存算法设计角度优化其局部性.

- 首先,可研究外存算法中的数据访问重新排序,最大化磁盘顺序读写,尽量避免随机读写.局部性包含空间局部性.空间局部性满足处理器即将访问的存储单元通常是它当前访问存储单元邻近的单元,因此将这些单元一起调度.对于磁盘存储,若采用顺序读写的方式来代替随机读写的方式,会优化空间局部性.调整内存算法的内存数据结构,使原本连续访问的不相邻数据项相邻,以优化空间局部性.但这种技术不适用于外存算法,因为根据算法更改外存数据结构、数据布局和文件格式的代价过高,且仅对当前算法有效,不具有普适性.因此,依据外存数据结构的特点调整算法结构,重排数据访问顺序,力争最大化磁盘顺序读写的可能;
- 其次,可研究外存算法中数据重用距离的优化.局部性还包含时间局部性.时间局部性满足处理器最近访问过的存储单元通常在短期内会再次被访问,因此,它们应该尽量保留在离处理器较近的存储层次.从存储角度,数据缓存是利用算法时间局部性来减少 CPU 被动空闲的有效技术.从算法角度,应该尽量缩短数据项被再次访问的时间,缩短数据重复访问的距离,充分利用数据缓存.在大数据处理平台中,数据缓存可以通过内存或分布式内存的方式实现,也可以是位于本地磁盘的远程数据缓冲.分析外存算法中各个数据之间的依赖性,提出数据重用距离,即,从数据首次读取位置至再次读取位置之间的距离,

该距离与传统内存算法的距离表达不同.如何根据不同的数据缓存度量该距离?如何通过算法的缩小这个距离,如何数据项距离的整体距离度量?如何优化这个度量?这些都是可供研究的问题;

- 最后,可研究如何将算法改写为局部性最优的算法.无论是空间还是时间局部性,若算法采用批处理的方法处理数据,满足:① 每数据项逻辑上访问且仅访问一次;② 数据项之间没有依赖关系,可以任意调整访问顺序.那么该算法将是局部性最优的算法.可研究局部性最优算法的特征:何种算法可以满足局部性最优或是可以变换为局部性最优算法、是否可以通过适当改变算法功能,如采用近似算法或概率算法,使其满足局部性最优算法、如何将一个算法改写为局部性最优的算法等一系列问题.

外存算法的局部性优化研究的现有成果较少,大部分都是研究算法本地性优化及算法访问本地数据而非远程数据.我们考虑更一般的程序局部性研究.文献[69]通过数据重用距离来评价程序局部性,该文利用基于训练的程序分析对数据重用距离加以建模和预测,可以量化地表征程序局部性.文献[80]进一步分析了数据重用距离和程序局部性在多核系统上的适用性和特征,提出了并发重用距离的概念,并建立了并发重用距离和程序局部性之间的关系.文献[81]研究了图遍历算法的局部性特征,提出顶点距离(vertex distance)的概念.文献[82]分析了以循环为主体的程序的局部性特征,采用数据重用距离来度量程序局部性,并提出一个 MemAddIn 工具对源码进行优化,充分利用其局部性减少内存失效的概率.文献[83]采用条件概率这一数学模型来量化程序局部性.文献[84]研究了 Load-and-compute 风格程序的局部性,Load-and-compute 是指装载数据随后完成数据的计算.该文提出了重用能力的概念,并采用重用能力来表征程序局部性;随后,根据局部性来管理内存资源,提高程序运行性能.

3.3 增量式迭代算法

大数据是数据量庞大且高速增长的数据,当数据量增加后,原始数据处理结果将不再适用,数据处理算法需要在数据全集上重新运行,这将浪费大量的计算资源.若能够通过新增数据集和已知的处理结果完成增量式的数据处理,则可以在很大程度上提高数据处理的效率.这就要求大数据处理算法支持增量式数据处理,也即设计增量式外存算法.由前文分析得知:大数据处理算法中的 Maps 算法可以很好地支持增量计算,而部分 Reduces 算法可支持增量计算,或通过近似算法支持增量计算.迭代算法作为大数据处理算法中运用最广泛且算法最复杂的一类,尚难以支持增量计算,因此,增量迭代算法则是亟待研究的问题.

增量式迭代算法已经是研究的热点.例如,文献[85]给出了增量式的迭代计算的相关条件,同时给出了相关增量式的迭代计算的例子以及反例.文献[86]研究了当迭代结构发生改变时,如何继续进行迭代.但这些研究工作较早,并未采用分布式计算模型.在分布式算法层面,文献[87]研究了 *K*-means 收敛的情况,并基于 Distortions Reduction 提出增量式 *K*-means 算法,解决 *K*-means 收敛到局部极值问题.文献[88]提出了基于手机轨迹数据的紧凑表示法和轨迹相似性度量,基于上述理论,该研究又提出了增量式的聚类方法用于发现空间中相似的移动终端.文献[89]基于现有的增量式神经网络(IGNG)提出了其改进算法 I2GNG 用于证券分类(invoice classification).文献[90]提出了增量式分类方法用于在线文档分类,但是其研究重点是相似性度量和文档结构的抽取.文献[91]属于增量式的 *K*-means 算法,且可以移植到 MapReduce 框架中,且移植后的算法与本文实验中所采用的 *K*-means 全量迭代算法是一致的.平台层面,Google 的 Percolator[92]系统可以以增量的方式更新索引;Incoop[93]系统可以增量地执行 MapReduce 作业;以及我们前期研究提出的 ΔHaLoop 能够以增量的方式执行迭代算法[16].

上述研究或是针对特定算法的增量实现,或是从平台作业管理角度对现有平台的改造,但都未能提出一般性的增量式迭代计算模型,如增量式 MapReduce.普适的增量式迭代计算模型可作为进一步研究问题.分析其关键挑战有以下两点.

(1) 如何对增量式迭代计算模型进行数学抽象才具有普适性.迭代算法普遍抽象为某类数学运算,如 PageRank 算法抽象为矩阵运算,*K*-means 算法抽象为函数运算,Descendant Query 算法抽象为关系运算.为使增量计算模型能够适用于大多数迭代算法,采取何种数学模型可适用、可抽象或可转化为算法数学模型,是一个颇具挑战的难题;

(2) 如何量化数据以及数据间关系对迭代结果的影响.众多迭代算法可抽象为根据数据和数据间关系,通过反复计算,累积各数据对每轮结果的贡献度,直至收敛至最终结果这一过程.因此,数据及数据间关系能够影响迭代结果.若能量化数据和关系对迭代结果的影响,则能准确评估增量迭代结果的精确度,且可以在迭代前,根据影响程度划分数据,对数据进行预处理,例如:对增量数据与原数据之间的关系做出取舍,适当采样原始数据以参与增量计算.

4 结 论

本文对近几年国内外在大数据处理平台和处理算法领域的主要研究成果进行了综述.总结了基于 MapReduce 的多种大数据处理平台的原理和实现方法,分析对比了它们的优缺点,并提出了它们的计算模型的一般性表达;随后,按大数据处理算法的 MapReduce 模型实现,将其分为 Maps 算法、Reduces 算法和迭代算法,又将每一类按算法功能进一步分类,描述每一小类的典型算法的设计思路和实现方法并分析这些影响这些算法的性能因素;最后,为了提出具有一般性的大数据处理算法优化思路,本文将其抽象为外存算法,并对外存算法的特征加以梳理.

本文提出 3 种外存算法优化思路:优化外存算法的磁盘 I/O、优化外存算法的局部性以及设计增量式迭代算法.每个优化思路均详述了可供研究的问题以及解决问题的基本思路,以供研究人员参考.总而言之,大数据处理算法的优化研究仍然处于刚刚起步的阶段,现有研究多关注平台优化、特定算法在大数据集上的适应性问题、特定大数据算法的优化研究,仍然有大量具有挑战性的关键问题需要深入研究,为大数据处理平台和算法领域的算研究者提供了广阔的研究空间.

References:

[1] Wu L, Yuan L, You J. Survey of large-scale data management systems for big data applications. Journal of Computer Science and Technology, 2015,30(1):163−183. [doi: 10.1007/s11390-015-1511-8]

[2] Dean J, Ghemawat S. MapReduce: Simplified data processing on large clusters. Communications of the ACM, 2008,51(1): 107−113. [doi: 10.1145/1327452.1327492]

[3] Wolf J, Balmin A, Rajan D, Hildrum K, Khandekar R, Parekh S, Wu KL, Vernica R. On the optimization of schedules for MapReduce workloads in the presence of shared scans. The VLDB Journal—The Int'l Journal on Very Large Data Bases, 2012,21 (5):589−609. [doi: 10.1007/s00778-012-0279-5]

[4] Computing platform. 2016. https://en.wikipedia.org/wiki/Computing_platform

[5] Yang H, Luan Z, Li W, Qian D. MapReduce workload modeling with statistical approach. Journal of Grid Computing, 2012,10(2): 279−310. [doi: 10.1007/s10723-011-9201-4]

[6] Kimura K, Nomura Y, Tanaka Y, Kurihara H, Yamamoto R. Runtime composition for extensible big data processing platforms. In: Proc. of the 2015 IEEE 8th Int'l Conf. on Cloud Computing. 2015. 1053−1057. [doi: 10.1109/CLOUD.2015.151]

[7] Out-of-Core algorithm. 2016. https://en.wikipedia.org/wiki/Out-of-core_algorithm

[8] Low Y, Gonzalez J, Kyrola A, Bickson D, Bickson D, Guestrin C, Hellerstein JM. Distributed graphLab: A framework for machine learning and data mining in the cloud. Proc. of the VLDB Endowment, 2012,5(8):716−727. [doi: 10.14778/2212351.2212354]

[9] Zhang J, Xiang D, Li T, Pan Y. M2M: A simple Matlab-to-MapReduce translator for cloud computing. Tsinghua Science and Technology, 2013,18(1):1−9.

[10] Liu Y, Li M, Alham NK, Hammoud S. HSim: A MapReduce simulator in enabling cloud computing. Future Generation Computer Systems, 2013,29(1):300−308. [doi: 10.1016/j.future.2011.05.007]

[11] GridGain in-memory data fabric. http://go.gridgain.com/rs/491-TWR-806/images/GridGain_Product_Datasheet_070416.pdf

[12] Fang W, He B, Luo Q, Govindaraju NK. Mars: Accelerating mapreduce with graphics processors. IEEE Trans. on Parallel and Distributed Systems, 2011,22(4):608−620. [doi: 10.1109/TPDS.2010.158]

[13] Yoo RM, Romano A, Kozyrakis C. Phoenix rebirth: Scalable MapReduce on a large-scale shared-memory system. In: Proc. of the IEEE Int'l Symp. on Workload Characterization (IISWC 2009). IEEE, 2009. 198−207. [doi: 10.1109/IISWC.2009.5306783]

[14] Mundkur P, Tuulos V, Flatow J. Disco: A computing platform for large-scale data analytics. In: Proc. of the 10th ACM SIGPLAN Workshop on Erlang. 2011. 84−89. [doi: 10.1145/2034654.2034670]

[15] Ekanayake J, Li H, Zhang B, Gunarathne T, Bae S, Qiu J, Fox G. Twister: A runtime for iterative MapReduce. In: Proc. of the 19th ACM Int'l Symp. on High Performance Distributed Computing. ACM Press, 2010. 810−818. [doi: 10.1145/1851476.1851593]

[16] Bu Y, Howe B, Balazinska M, Ernst MD. HaLoop: Efficient iterative data processing on large clusters. Proc. of the VLDB Endowment, 2010,3(1-2):285−296. [doi: 10.14778/1920841.1920881]
[17] Zhang Y, Gao Q, Gao L, Wang C. Imapreduce: A distributed computing framework for iterative computation. Journal of Grid Computing, 2012,10(1):47−68. [doi: 10.1007/s10723-012-9204-9]
[18] Elnikety E, Elsayed T, Ramadan HE. iHadoop: Asynchronous iterations for MapReduce. In: Proc. of the 3rd IEEE Int'l Conf. on Cloud Computing Technology and Science (CloudCom). IEEE, 2011. 81−90. [doi: 10.1109/CloudCom.2011.21]
[19] Zhang Y, Gao Q, Gao L, Wang C. PrIter: A distributed framework for prioritized iterative computations. In: Proc. of the 2nd ACM Symp. on Cloud Computing. ACM Press, 2011. 13. [doi: 10.1145/2038916.2038929]
[20] Isard M, Budiu M, Yu Y, Birrell A, Fetterly D. Dryad: Distributed data-parallel programs from sequential building blocks. Proc. of the ACM SIGOPS Operating Systems Review, 2007,41(3):59−72. [doi: 10.1145/1272998.1273005]
[21] Zaharia M, Chowdhury M, Franklin MJ, Shenker S, Stoica I. Spark: Cluster computing with working sets. HotCloud, 2010.
[22] Rasooli A, Down DG. Guidelines for selecting hadoop schedulers based on system heterogeneity. Journal of Grid Computing, 2014, 12(3):499−519. [doi: 10.1007/s10723-014-9299-2]
[23] Karun AK, Chitharanjan K. A review on hadoop—HDFS infrastructure extensions. In: Proc. of the 2013 IEEE Conf. on Information & Communication Technologies (ICT). IEEE, 2013. 132−137. [doi: 10.1109/CICT.2013.6558077]
[24] Vavilapalli VK, Murthy AC, Douglas C, Agarwal S, Konar M, Evans R, Graves T, Lowe J, Shah H, Seth S, Saha B, Curino C, O'Malley O, Radia S, Reed B, Baldeschwieler E. Apache Hadoop YARN: Yet another resource negotiator. In: Proc. of the 4th Annual Symp. on Cloud Computing. 2013. 16. [doi: 10.1145/2523616.2523633]
[25] Ranger C, Raghuraman R, Penmetsa A, Bradski G, Kozyrakis C. Evaluating mapreduce for multi-core and multiprocessor systems. In: Proc. of the 2007 IEEE 13th Int'l Symp. on High Performance Computer Architecture. IEEE, 2007. 13−24. [doi: 10.1109/HPCA. 2007.346181]
[26] Pietzuch PR, Bacon J. Peer-to-Peer overlay broker networks in an event-based middleware. In: Proc. of the 2nd Int'l Workshop on Distributed Event-based Systems. ACM Press, 2003. 1−8. [doi: 10.1145/966618.966628]
[27] Armbrust M, Xin RS, Lian C, Huai Y, Liu D, Bradley JK, Meng X, Kaftan T, Franklin MJ, Ghodsi A, Zaharia M. Spark SQL: Relational data processing in spark. In: Proc. of the 2015 ACM SIGMOD Int'l Conf. on Management of Data. ACM Press, 2015. 1383−1394. [doi: 10.1145/2723372.2742797]
[28] Gonzalez J, Xin R, Dave A, Stoica I. GraphX: Graph processing in a distributed dataflow framework. In: Proc. of the Int'l Conf. on Operating Systems Design and Implementation. 2014. 599−613.
[29] Matei Z, Tathagata D, Haoyuan L, Timothy H, Scott S, Ion S. Discretized streams: Fault-Tolerant streaming computation at scale. In: Proc. of the SOSP. 2013. 423−438. [doi: 10.1145/2517349.2522737]
[30] Meng X, Bradley J, Yuvaz B, Sparks E, Venkataraman S, Liu D, Freeman J, Tsai DB, Made M, Owen S, Xin D, Xin R, Franklin MJ, Zadeh R, Zaharia M, Talwalkar A. Mllib: Machine learning in apache spark. Journal Machine Learning Research, 2016,17(34): 1−7.
[31] Qiu J, Wu Q, Ding G, Xu Y, Feng S. A survey of machine learning for big data processing. EURASIP Journal on Advances in Signal Processing, 2016,2016(1):1−16. [doi: 10.1186/s13634-015-0293-z]
[32] Martins R, Manquinho V, Lynce I. Improving linear search algorithms with model-based approaches for MaxSAT solving. Journal of Experimental & Theoretical Artificial Intelligence, 2015,27(5):673−701. [doi: 10.1080/0952813X.2014.993508]
[33] Wang HZ. Big Data Algorithms. Beijing: China Machine Press, 2015 (in Chinese).
[34] Ding XO, Wang HZ, Zhang XY, Gao H. Association relationships study of multi-dimensional data quality. Ruan Jian Xue Bao/ Journal of Software, 2016,27(7):1626−1644 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/5040.htm [doi: 10. 13328/j.cnki.jos.005040]
[35] Yang DH, Li NN, Wang HZ, Li JZ, Gao H. The optimization of the big data cleaning based on task merging. Chinese Journal of Computers, 2016,39(1):97−108 (in Chinese with English abstract).
[36] Han JW, Kamber M, Pei J. Data Mining: Concepts and Techniques. 3rd ed., Morgan Kaufmann Publishers, 2011.
[37] Wang Y, Su Y, Agrawal G. A novel approach for approximate aggregations over arrays. In: Proc. of the 27th Int'l Conf. on Scientific and Statistical Database Management. ACM Press, 2015. [doi: 10.1145/2791347.2791349]
[38] Issa JA. Performance evaluation and estimation model using regression method for hadoop WordCount. IEEE Access, 2015,3: 2784−2793. [doi: 10.1109/ACCESS.2015.2509598]
[39] Han XX, Yang DH, Li JZ. Approximate join aggregate on massive data. Chinese Journal of Computers, 2010,10:1919−1933 (in Chinese with English abstract).
[40] Song J, Li TT, Zhu ZL, Bao YB, Yu G. Research on I/O cost of MapReduce join. Ruan Jian Xue Bao/Journal of Software, 2015, 26(6):1438−1456 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/4586.htm [doi: 10.13328/j.cnki.jos.004586]

[41] Asiri N, Alsulim R. Non-Recursive approach for sort-merge join operation. In: Proc. of Int'l the Conf. on Beyond Databases, Architectures and Structures. Springer Int'l Publishing, 2015. 216−224. [doi: 10.1007/978-3-319-34099-9_16]

[42] Chen M, Zhong Z. Block nested join and sort merge join algorithms: An empirical evaluation. In: Proc. of the Int'l Conf. on Advanced Data Mining and Applications. Springer Int'l Publishing, 2014. 705−715.

[43] Tong Y, Liu ZJ, Liu H. Optimizing Hash join with MapReduce on multi-core CPUs. IEICE Trans. on Information and Systems, 2016,99(5):1316−1325. [doi: 10.1587/transinf.2015EDP7306]

[44] Song J, Xu S, Zhang L, Pahl C, Yu G. Performance and energy optimization on terasort algorithm by task self-resizing. Information Technology and Control, 2015,44(1):30−40.

[45] Ci X, Ma YZ, Meng XF. Method for top-*K* query on big data in cloud. Ruan Jian Xue Bao/Journal of Software, 2014,25(4): 813−825 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/4564.htm [doi: 10.13328/j.cnki.jos.004564]

[46] Li WF, Peng ZY, Li DY. Top-*K* query processing techniques on uncertain data. Ruan Jian Xue Bao/Journal of Software, 2012,23(6): 1542−1560 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/4200.htm [doi: 10.3724/SP.J.1001.2012.04200]

[47] MacLean B, Tomazela DM, Shulman N, Chambers M, Finney GL, Frewen B, Kern R, Tabb DL, Liebler DC, MacCoss MJ. Skyline: An open source document editor for creating and analyzing targeted proteomics experiments. Bioinformatics, 2010,26(7):966−968. [doi: 10.1093/bioinformatics/btq054]

[48] Zhang B, Zhou S, Guan J. Adapting skyline computation to the mapreduce framework: Algorithms and experiments. In: Proc. of the Int'l Conf. on Database Systems for Advanced Applications. Berlin, Heidelberg: Springer-Verlag, 2011. 403−414. [doi: 10.1007 /978-3-642-20244-5_39]

[49] Ding LL, Xin JC, Wang GR, Huang S. Efficient skyline query processing of massive data based on MapReduce. Chinese Journal of Computers, 2011,34(10):1785−1796 (in Chinese with English abstract). [doi: 10.3724/SP.J.1016.2011.01785]

[50] Jin C, Vecchiola C, Buyya R. MRPGA: An extension of MapReduce for parallelizing genetic algorithms. In: Proc. of the 4th IEEE Int'l Conf. on eScience (eScience 2008). IEEE, 2008. 214−221. [doi: 10.1109/eScience.2008.78]

[51] McNabb AW, Monson CK, Seppi KD. Parallel pso using mapreduce. In: Proc. of the 2007 IEEE Congress on Evolutionary Computation. IEEE, 2007. 7−14. [doi: 10.1109/CEC.2007.4424448]

[52] Li H, Wei X, Fu Q, Luo Y. MapReduce delay scheduling with deadline constraint. Concurrency and Computation: Practice and Experience, 2014,26(3):766−778. [doi: 10.1002/cpe.3050]

[53] Xu X, Ji Z, Yuan F, Liu X. A novel parallel approach of cuckoo search using MapReduce. In: Proc. of the 2014 Int'l Conf. on Computer, Communications and Information Technology (CCIT 2014). Atlantis Press, 2014. [doi: 10.2991/ccit-14.2014.31]

[54] Whang JJ, Lenharth A, Dhillon IS, Pingali K. Scalable data-driven pagerank: Algorithms, system issues, and lessons learned. In: Proc. of the European Conf. on Parallel Processing. Berlin, Heidelberg: Springer-Verlag, 2015. 438−450. [doi: 10.1007/978-3-662 -48096-0_34]

[55] Song J, Guo CP, Zhang YC, Zhang YF, Yu G. Research and implemental incremental iterative model. Chinese Journal of Computers, 2016,39(1):109−125 (in Chinese with English abstract).

[56] Bu Y, Howe B, Balazinska M, Ernst MD. The HaLoop approach to large-scale iterative data analysis. The VLDB Journal—The Int'l Journal on Very Large Data Bases, 2012,21(2):169−190. [doi: 10.1007/s00778-012-0269-7]

[57] Valiant LG. A bridging model for parallel computation. Communications of the ACM, 1990,33(3):103−111. [doi: 10.1145/79173. 79181]

[58] Yu G, Gu Y, Bao YB, Wang ZG. Large scale graph data processing on cloud computing environments: Challenges and progress. Chinese Journal of Computers, 2011,34(10):1753−1767 (in Chinese with English abstract). [doi: 10.3724/SP.J.1016.2011.01753]

[59] Mohanavalli S, Jaisakthi SM, Aravindan C. Strategies for parallelizing *k*-means data clustering algorithm. In: Proc. of the Information Technology and Mobile Communication. Berlin, Heidelberg: Springer-Verlag, 2011. 427−430. [doi: 10.1007/978-3-642-20573-6_76]

[60] Liao Q, Yang F, Zhao J. An improved parallel *K*-means clustering algorithm with MapReduce. In: Proc. of the 15th IEEE Int'l Conf. on Communication Technology (ICCT). IEEE, 2013. 764−768. [doi: 10.1109/ICCT.2013.6820477]

[61] Li ZH, Song XD, Zhu WH, Chen YX. *K*-Means clustering optimization algorithm based on MapReduce. In: Proc. of the 2015 Int'l Symp. on Computers & Informatics. 2015. 198−203.

[62] Li Q, Wang P, Wang W, Hu H, Li Z, Li J. An efficient *K*-means clustering algorithm on MapReduce. In: Proc. of the Int'l Conf. on Database Systems for Advanced Applications. Springer Int'l Publishing, 2014. 357−371. [doi: 10.1007/978-3-319-05810-8_24]

[63] Çatak FÖ, Balaban ME. A MapReduce-based distributed SVM algorithm for binary classification. Turkish Journal of Electrical Engineering & Computer Sciences, 2016,24(3):863−873. [doi: 10.3906/elk-1302-68]

[64] Rong C. Using Mahout for clustering Wikipedia's latest articles: A comparison between *K*-means and fuzzy C-means in the cloud. In: Proc. of the IEEE 3rd Int'l Conf. on Cloud Computing Technology and Science (CloudCom). IEEE, 2011. 565−569. [doi: 10.1109/CloudCom.2011.86]

[65] Pop D, Iuhasz G, Petcu D. Distributed platforms and cloud services: Enabling machine learning for big data. In: Proc. of the Data Science and Big Data Computing. Springer Int'l Publishing, 2016. 139−159. [doi: 10.1007/978-3-319-31861-5_7]

[66] Dino K. H2O persistence framework for column oriented distributed (NoSQL) databases. In: Proc. of the 3rd Int'l Symp. on Sustainable Development. Sarajevo, 2012.

[67] Gu L, Li H. Memory or time: Performance evaluation for iterative operation on Hadoop and spark. In: Proc. of the 10th IEEE Int'l Conf. on High Performance Computing and Communications & 2013 IEEE Int'l Conf. on Embedded and Ubiquitous Computing (HPCC_EUC). IEEE, 2013. 721−727. [doi: 10.1109/HPCC.and.EUC.2013.106]

[68] Marz N, Warren J. Big Data: Principles and Best Practices of Scalable Realtime Data Systems. Manning Publications Co. Greenwich, 2015.

[69] Zhong Y, Shen X, Ding C. Program locality analysis using reuse distance. ACM Trans. on Programming Languages and Systems (TOPLAS), 2009,31(6):20. [doi: 10.1145/1552309.1552310]

[70] Song J, Wang Z, Li TT, Yu G. Energy consumption optimization data placement algorithm for MapReduce System. Ruan Jian Xue Bao/Journal of Software, 2015,26(8):2091−2110 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/4802.htm [doi: 10.13328/j.cnki.jos.004802]

[71] Song J, He HY, Wang Z, Yu G, Pierson JM. Modulo based data placement algorithm for energy consumption optimization of MapReduce system. Journal of Grid Computing, 2016. 1−16. [doi: 10.1007/s10723-016-9370-2]

[72] Li Y, Li H. Optimization of parallel I/O for Cannon's algorithm based on lustre. In: Proc. of the 11th Int'l Symp. on Distributed Computing and Applications to Business, Engineering & Science (DCABES). IEEE, 2012. 31−35. [doi: 10.1109/DCABES.2012.61]

[73] Blelloch GE, Harper R. Cache and I/O efficent functional algorithms. Proc. of the ACM SIGPLAN Notices, 2013,48(1):39−50. [doi: 10.1145/2480359.2429077]

[74] Talebi M, Razzazi M. An I/O cost optimal and progressive algorithm for computing massive skyline points. In: Proc. of the 35th Int'l Convention (MIPRO). IEEE, 2012. 333−338.

[75] Mohanty SK. I/O efficient algorithms for matrix computations. arXiv preprint arXiv:1006.1307, 2010.

[76] Ghoting A, Makarychev K. I/O efficient algorithms for serial and parallel suffix tree construction. ACM Trans. on Database Systems (TODS), 2010,35(4):25. [doi: 10.1145/1862919.1862922]

[77] Haverkort H. I/O-Optimal algorithms on grid graphs. arXiv preprint arXiv:1211.2066, 2012.

[78] Gui X, Zhang Y, Hao X. An almost linear I/O algorithm for skyline query. Journal of Software, 2010,5(2):235−242. [doi: 10.4304/jsw.5.2.235-242]

[79] Ramaswamy S, Suel T. I/O-Efficient join algorithms for temporal, spatial, and constraint databases. CiteSeer, 1996.

[80] Jiang Y, Zhang EZ, Tian K, Shen X. Is reuse distance applicable to data locality analysis on chip multiprocessors? In: Proc. of the Int'l Conf. on Compiler Construction. Berlin, Heidelberg: Springer-Verlag, 2010. 264−282. [doi: 10.1007/978-3-642-11970-5_15]

[81] Lezos C, Dimitroulakos G, Masselos K. Reuse distance analysis for locality optimization in loop-dominated applications. In: Proc. of the 2015 Design, Automation & Test in Europe Conf. & Exhibition (DATE). IEEE, 2015. 1237−1240. [doi: 10.7873/DATE.2015.0442]

[82] Yuan L, Ding C, Zhang Y. Modeling the locality in graph traversals. In: Proc. of the 41st Int'l Conf. on Parallel Processing. IEEE, 2012. 138−147. [doi: 10.1109/ICPP.2012.40]

[83] Gupta S, Xiang P, Yang Y, Zhou H. Locality principle revisited: A probability-based quantitative approach. Journal of Parallel and Distributed Computing, 2013,73(7):1011−1027. [doi: 10.1016/j.jpdc.2013.01.010]

[84] Yuan L, Zhang Y. A locality-based performance model for load-and-compute style computation. In: Proc. of the 2012 IEEE Int'l Conf. on Cluster Computing. IEEE, 2012. 566−571. [doi: 10.1109/CLUSTER.2012.25]

[85] Ryder BG, Marlowe TJ, Paull MC. Conditions for incremental iteration: Examples and counterexamples. Science of Computer Programming, 1988,11:1−15. [doi: 10.1016/0167-6423(88)90061-5]

[86] Burke M. An interval-based approach to exhaustive and incremental interprocedural data-flow analysis. ACM Trans. on Programming Languages and Systems, 1990,12:341−95. [doi: 10.1145/78969.78963]

[87] Pham DT, Dimov SS, Nguyen CD. An incremental *K*-means algorithm. Journal of Mechanical Engineering Science, 2004,218: 783−95. [doi: 10.1243/0954406041319509]

[88] Elnekave S, Last M, Maimon O. Incremental clustering of mobile objects. In: Proc. of the 23rd IEEE Int'l Conf. on Data Engineering Workshop. IEEE, 2007. 585−592. [doi: 10.1109/ICDEW.2007.4401044]

[89] Hamza H, Belaïd Y, Belaïd A, Chaudhuri BB. Incremental classification of invoice documents. In: Proc. of the 19th Int'l Conf. on Pattern Recognition (ICPR 2008). IEEE, 2008. 1–4. [doi: 10.1109/ICPR.2008.4761832]

[90] Khy S, Ishikawa Y, Kitagawa H. A novelty-based clustering method for on-line documents. World Wide Web, 2008,11(1):1–37. [doi: 10.1007/s11280-007-0018-9]

[91] Chakraborty S, Nagwani NK. Analysis and study of incremental *K*-means clustering algorithm. In: Proc. of the Int'l Conf. on High Performance Architecture and Grid Computing. Berlin: Springer-Verlag, 2011. 338–341. [doi: 10.1007/978-3-642-22577-2_46]

[92] Daniel P, Frank D. Large-Scale incremental processing using distributed transactions and notifications. In: Proc. of the 9th Symp. on Operating Systems Design and Implementation. 2010. 137–49.

[93] Bhatotia P, Wieder A, Rodrigues R, Acar UA, Pasquini R. Incoop: MapReduce for incremental computations. In: Proc. of the 2nd ACM Symp. on Cloud Computing. ACM, 2011. [doi: 10.1145/2038916.2038923]

附中文参考文献:

[33] 王宏志.大数据处理算法.北京:机械工业出版社,2015.

[34] 丁小欧,王宏志,张笑影,李建中,高宏.数据质量多种性质的关联关系研究.软件学报,2016,27(7):1626–1644. http://www.jos.org.cn/1000-9825/5040.htm [doi: 10.13328/j.cnki.jos.005040]

[35] 杨东华,李宁宁,王宏志,李建中,高宏.基于任务合并的并行大数据清洗过程优化.计算机学报,2016,39(1):97–108.

[39] 韩希先,杨东华,李建中.海量数据上的近似连接聚集操作.计算机学报,2010,10:1919–1933.

[40] 宋杰,李甜甜,张莉,朱志良,鲍玉斌,于戈.MapReduce 连接查询的 I/O 代价研究.软件学报,2015,26(6):1438–1456. http://www.jos.org.cn/1000-9825/4586.htm [doi: 10.13328/j.cnki.jos.004586]

[45] 慈祥,马友忠,孟小峰.一种云环境下的大数据 Top-*K* 查询方法.软件学报,2014,25(4):813–825. http://www.jos.org.cn/1000-9825/4564.htm [doi: 10.13328/j.cnki.jos.004564]

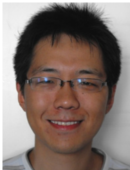
[46] 李文凤,彭智勇,李德毅.不确定性 Top-*K* 查询处理.软件学报,2012,23(6):1542–1560. http://www.jos.org.cn/1000-9825/4200.htm [doi: 10.3724/SP.J.1001.2012.04200]

[49] 丁琳琳,信俊昌,王国仁,黄山.基于 Map-Reduce 的海量数据高效 Skyline 查询处理.计算机学报,2011,34(10):1785–1796.

[55] 宋杰,郭朝鹏,张一川,张岩峰,于戈.增量式迭代计算模型研究与实现.计算机学报,2016,39(1):109–125.

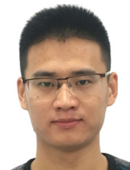
[58] 于戈,谷峪,鲍玉斌,王志刚.云计算环境下的大规模图数据处理技术.计算机学报,2011,34(10):1753–1767.

[70] 宋杰,王智,朱志良,李甜甜,于戈.一种优化 MapReduce 系统能耗的数据布局算法.软件学报,2015,26(8):2091–2110. http://www.jos.org.cn/1000-9825/4802.htm [doi: 10.13328/j.cnki.jos.004802]



宋杰(1980－),男,安徽淮北人,博士,副教授,CCF 高级会员,主要研究领域为大数据存储与管理,迭代计算,高能效计算.

鲍玉斌(1968－),男,博士,教授,CCF 高级会员主要研究领域为数据仓库,图数据处理.



孙宗哲(1991－),男,硕士生,主要研究领域为大数据处理.

于戈(1962－),男,博士,教授,博士生导师,CCF 会士,主要研究领域为数据库理论和技术,分布与并行系统.

毛克明(1981－),男,博士,讲师,主要研究领域为社会媒体数据处理.