软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2018,29(6):1770−1791 [doi: 10.13328/j.cnki.jos.005557]

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

MapReduce 与 Spark 用于大数据分析之比较*

吴信东[1,2], 嵇圣硙[1]

[1](合肥工业大学 计算机与信息学院,安徽 合肥 230009)

[2](School of Computing and Informatics, University of Louisiana at Lafayette, Lafayette 70504, USA)

通讯作者: 吴信东, E-mail: xwu@hfut.edu.cn

摘 要: 评述了 MapReduce 与 Spark 两种大数据计算算法和架构,从背景、原理以及应用场景进行分析和比较,并对两种算法各自优点以及相应的限制做出了总结.当处理非迭代问题时,MapReduce 凭借其自身的任务调度策略和 shuffle 机制,在中间数据传输数量以及文件数目方面的性能要优于 Spark;而在处理迭代问题和一些低延迟问题时,Spark 可以根据数据之间的依赖关系对任务进行更合理的划分,相较于 MapReduce,有效地减少了中间数据传输数量与同步次数,提高了系统的运行效率.

关键词: 大数据;MapReduce;Spark;迭代问题;非迭代问题

中图法分类号: TP311

中文引用格式: 吴信东,嵇圣硙.MapReduce 与 Spark 用于大数据分析之比较.软件学报,2018,29(6):1770−1791. http://www.jos.org.cn/1000-9825/5557.htm

英文引用格式: Wu XD, Ji SW. Comparative study on MapReduce and Spark for big data analytics. Ruan Jian Xue Bao/Journal of Software, 2018,29(6):1770−1791 (in Chinese). http://www.jos.org.cn/1000-9825/5557.htm

Comparative Study on MapReduce and Spark for Big Data Analytics

WU Xin-Dong[1,2], JI Sheng-Wei[1]

[1](School of Computer Science and Information Engineering, Hefei University of Technology, Hefei 230009, China)

[2](School of Computing and Informatics, University of Louisiana at Lafayette, Lafayette 70504, USA)

Abstract: This paper reviews two state-of-the-art algorithmic architectures, MapReduce and Spark, and compares them from their backgrounds, principles and application scenarios. The advantages and their corresponding limitations of these two algorithms are summarized. When dealing with non-iterative problems, MapReduce, by virtue of its task scheduling strategy and shuffle mechanisms, performs better than Spark in terms of intermediate data transfers and number of files. Spark can be used to deal with iterative problems and low latency issues, as it divides a computing task according to the dependencies between the data and the task. Compared with MapReduce, Spark can effectively reduce the number of intermediate data transmissions and the number of synchronizations, and improve the running efficiency of computing systems.

Key words: big data; MapReduce; Spark; iterative problems; non-iterative problems

随着互联网的持续发展,我们可收集获取的数据规模在不断增长.尽管数据的收集存储技术还在进步和日趋成熟[1],但是如何处理如此庞大的数据成为了新的研究问题.在分布式系统出现之前,只有通过不断增加单个处理机的频率和性能来缩短数据的处理时间.而分布式的提出则打破了这个传统的约束.所谓分布式,就是将一

* 基金项目: 国家重点研发计划(2016YFB1000901); 国家自然科学基金(91746209); 教育部创新团队项目(IRT17R3)

Foundation item: National Key Researh and Development Program of China (2016YFB1000901); National Natural Science Foundation of China (91746209); Program for Changjiang Scholars and Innovative Research Team in University (PCSIRT) of the Ministry of Education (IRT17R3)

收稿时间: 2017-10-19; 采用时间: 2018-01-15; jos 在线出版时间: 2018-02-08

CNKI 网络优先出版: 2018-02-08 11:56:05, http://kns.cnki.net/kcms/detail/11.2560.TP.20180208.1155.011.html

个复杂的问题切割成很多子任务,分布到多台机器上并行处理.在保证系统稳定性的同时,最大限度地提高系统的运行速度.

Google 在 2004 年提出了最原始的分布式架构模型 MapReduce[2],用于大规模的数据并行处理.MapReduce 模型借鉴了函数式程序设计语言中的内置函数 Map 和 Reduce,主要思想为:将大规模数据处理作业拆分成多个可独立运行的 Map 任务,分布到多个处理机上运行,产生一定量的中间结果,再通过 Reduce 任务混洗合并产生最终的输出文件.作为第一代分布式架构,MapReduce 已经比较好地考虑了数据存储、调度、通信、容错管理、负载均衡等很多问题,一定程度上降低了并行程序开发难度,也为后来的分布式系统的开发打下了很好的基础[3].然而它也存在很多不足:首先,为了保证较好的可扩展性,MapReduce 的任务之间相互独立,互不干扰,所造成的后果便是大量的中间结果需要通过网络进行传输,占用了网络资源,并且为了保证容错,所有的中间结果都要存储到磁盘中,效率不高;同时,在 MapReduce 中只有等待所有的 Map 任务结束后,Reduce 任务才能进行计算,异步性差,导致资源利用率低.

近年来,Zaharia 等人主导开发了新一代的大数据分布式处理框架 Spark[4].Spark 以其先进的设计理念,迅速成为热门课题,围绕着 Spark 推出了 Spark SQL,Spark Streaming,MLLib 和 GraphX 等组件,这些组件逐渐形成了大数据处理一站式解决平台.在处理迭代问题以及一些低延迟问题上,Spark 性能要高于 MapReduce.Spark 在 MapReduce 的基础上做了很多的改进与优化,使得在处理如机器学习以及图算法等迭代问题时,Spark 性能要优于 MapReduce.但是作为最经典的分布式架构,MapReduce 同样也有自身的可取之处,在特定的问题环境下,MapReduce 还是优于 Spark.

本文首先分析与介绍了 MapReduce 和 Spark 两种分布式架构,并在大数据评估指标和不同应用情景两方面对二者进行了比较和总结;最后,基于 WordCount 与 PageRank 问题从原理上对二者在算法执行时间、文件数目及同步次数等 3 方面进行了分析和对比,最后做出总结和展望.

本文第 1 节介绍 MapReduce 的背景、具体的模型结构以及作业的调度策略.第 2 节介绍 Spark 模型的具体思想以及与 MapReduce 的区别.第 3 节基于现有的研究对 MapReduce 和 Spark 在大数据评估指标和不同应用情景两方面进行比较和总结.第 4 节通过 WordCount 问题比较 MapReduce 与 Spark 在处理非迭代问题时各自的优缺点.第 5 节以 PageRank 问题为例,比较 MapReduce 与 Spark 在处理迭代问题时性能上的差异.第 6 节对本文内容进行总结并对大数据分析技术的发展趋势和前景做出展望.

1 MapReduce

1.1 MapReduce背景

在 Google,每天在需要处理大量的原始数据,比如网页爬取文件、网络日志文件等等.庞大的数据输入量和计算量是一台或者几台机器难以承受的,只有将任务分配到成百上千台处理机上去并行处理,才能在合理时间内完成计算.MapReduce 思想来源于 Lisp 函数式语言中的设计思想,提供了 Map 和 Reduce 两种函数来实现并行化操作.MapReduce 将分布式系统中如何分布、调度、监控以及容错等逻辑从复杂的细节中抽象出来,使得程序员不需要太多并发处理或者分布系统的经验,就可以处理超大的分布式系统的资源.MapReduce 是一种简化的分布式编程模型[5],它可以自动解决输入数据的分布细节,跨越机器集群的程序执行调度,处理机器的时效,并且管理机器之间的通信请求.

1.2 MapReduce模型

MapReduce 中的操作只有两种:Map 和 Reduce,而这两种操作都是由用户定义的.Map 和 Reduce 的输入与输出数据都是键值对⟨key/value⟩,可以用两个公式对它们进行简单的描述.

- Map $(k1,v1)\rightarrow list(k2,v2)$;
- Reduce $(k2,list(v2))\rightarrow list(v2)$.

在一个问题的计算过程中,Map 操作将数据自动地进行分区,并分布到多台处理机上进行并行处理.Reduce

操作会根据中间数据的键值key通过分区函数(如Hash函数)处理并分布到不同处理机上进行相同的计算,完成一个MapReduce计算过程.图1为完整的MapReduce计算流程图.

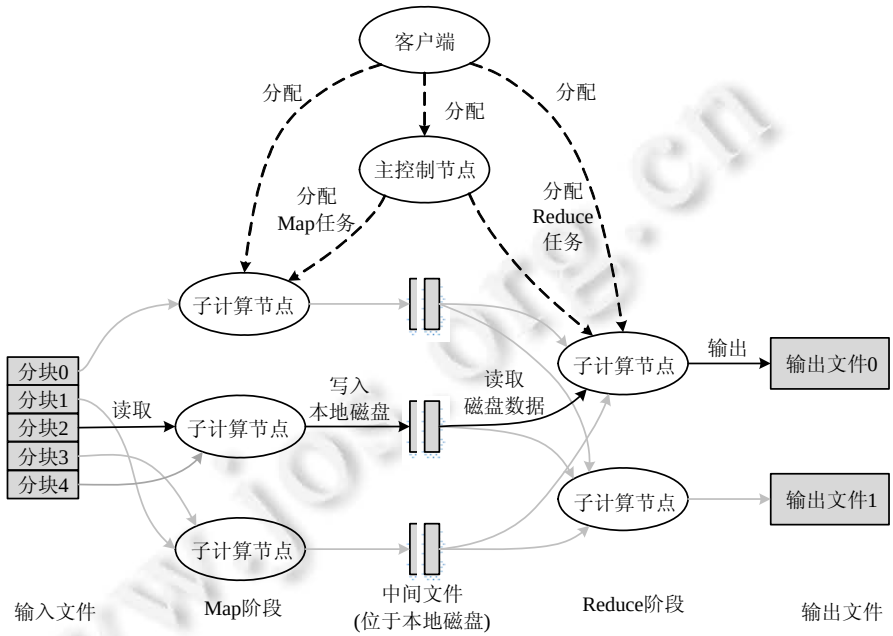

Fig.1 MapReduce flow chart[2]

图1 MapReduce流程图[2]

考虑在大量文档集合中统计单词“XindongWu”出现的次数问题,伪代码如下.

```
Map(String key,String value)
  //key:文档名
  //value:文档内容
  for each word “XindongWu” in value:
    EmitIntermediate(XindongWu,1);

Reduce(String key,Iterator values)
  //key:一个单词
  //value:值列表
  int result=0;
  for each v in values:
    result+=ParseInt(v);
  Emit(AsString(result));
```

在此问题中,Map 函数将所有文档进行分割,分布到多台处理机上进行处理.在并行处理时,只要遇到单词“XindongWu”便形成一个键值对(XindongWu,1).Reduce 函数将所有处理机上的键值对进行聚合并相加,最终得到单词“XindongWu”出现的次数.

除词频统计问题外,还有一些可以用MapReduce解决的问题.

- 分布式Grep(destributed grep):如果特定的行匹配到给定的模式,Map函数会将其传递给Reduce函数. Reduce函数负责将中间数据作为结果输出;

- URL 访问频率统计(count of URL):Map 函数首先处理网页请求日志并输出键值对〈URL,1〉.Reduce 函数对相同 URL 的键值对进行累加并输出为〈URL,total_count〉对;
- 主机词组向量(term-vector per host):在以〈word,frequency〉键值对列表表示的一个或一组文档中,词组向量是出现文档中最重要的词,以〈word,frequency〉对列表的形式表示.Map 函数处理输入文档并输出〈hostname,term vector〉键值对.Reduce 函数对具有相同的 host 键值对进行规约并去除非常用词,最终输出〈hostname,term vector〉键值对.

MapReduce 编程模型使得任务执行和任务通信之间变得简单且规范,任务并行化得以实现,扩展性良好;且每次 Map 操作的中间结果会被定期刷写到磁盘上,不会永久保留在内存中,这样做一是为了减少内存的消耗,避免内存溢出带来数据丢失;二是为了更好地容错,磁盘的稳定性和易恢复性让 MapReduce 的容错变得更加可行.但是,这样的设计也带来了一些缺点.

- 首先,一些问题并不适合用 Map 和 Reduce 操作来完成,如处理数据结构是图或者网络的问题时,因为图这样的数据结构中包含着各种隐性的关系:图的边、子树、节点之间的父子关系、权重等,而这些关系很难被分解为键值对在一次计算中完全表示.同时,任务依赖关系复杂时,会因为需要将任务分解以适应该模型而效率下降;
- 其次,在一个作业中,Reduce 任务必须要等待所有的 Map 任务完成后才能进行计算,异步性低,系统资源可能没有得到充分的利用.

1.3 MapReduce作业调度

有些问题可以通过一次 Map 和 Reduce 过程解决,比如词频统计(WordCount)问题.由于单个 MapReduce 过程能够完成的操作毕竟有限,有些复杂的问题就需要分解成多个 MapReduce 过程[6,7].每个 Reduce 操作的结果作为下一个 Map 操作的输入数据,以此过程来完成对复杂问题的处理[8].

在有多个 MapReduce 过程的问题中会涉及到作业的依赖关系问题,一个问题中,作业的依赖关系有可能为线性;但在更复杂的情况下,依赖关系为非线性.在 MapReduce 提出之初,考虑到多作业的情况,采用的是 FIFO (fist in fist out)方法.所有的作业被提交到一个队列中,按照优先级和提交时间的先后顺序扫描整个队列选择一个作业执行.允许执行的作业完全占用集群资源,等到当前作业执行完毕后再从队列中选择下一个执行的作业.FIFO 算法如今已经成为 MapReduce 默认的调度算法,但是它忽略了大作业和小作业之间的差异,通常会导致小作业的长时间等待影响到系统的吞吐率.公平调度算法(fair scheduler)[9]是 Facebook 公司开发的一种调度算法,该算法可以在一定时间内让所有的作业平等地共享集群资源.当集群只有一个作业在运行时,该作业独享整个集群的资源;当其他作业被提交到集群时,系统会将空余的任务槽分配给新提交的作业,使每个作业能够分到近似等量的系统资源.但是这种方法会减少每个作业能够分到的资源,增加作业的运行时间.文献[10]提出了一种计算能力感知的调度算法(capacity scheduler),该算法将系统资源分成多个队列,每个队列中的作业共享该队列的资源.2008 年,美国加州大学伯克利分校的 Andy Konwinski 等人提出了 LATE 调度算法(longest approximate time to end)[11].MapReduce 应用中一个作业会被划分成若干个任务并行运行,作业完成时间即最慢任务的完成时间,为提高作业完成速度,系统会将速度执行慢的任务在其他计算节点重新并行执行,该算法充分考虑了系统的异构性并借鉴了其他分布式系统备份执行机制[15].以上几种算法从根本上都是根据作业的提交先后顺序对作业进行调度,在作业之间的依赖关系为线性时,这些调度策略不用消耗太多资源在算法调度上,方便快捷;如在 PageRank 问题中,两次迭代过程被划分为 4 个作业,根据作业提交时间依次执行作业 1~作业 4.如图 2 所示.

但是以上调度策略没有考虑数据之间的依赖关系,仅根据作业的提交时间对作业进行调度,作业之间完全独立且中间结果需存入磁盘或通过网络进行节点传输,耗费了大量的系统资源.文献[12]提出了一种基于有向无环图的分布式通用执行引擎,能够较好地显示每步操作的依赖关系并且映射到物理资源上去.但是这些具体的依赖关系需要用户自行定义,系统无法自动判断.在复杂问题中也会带来很多麻烦,并且没有从根本上提高系统的执行速度.

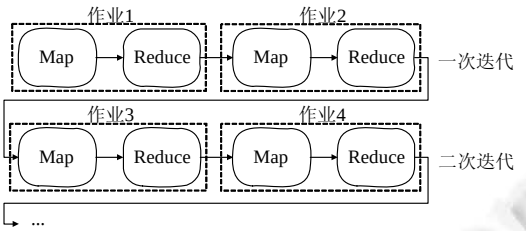

Fig.2 Job division of PageRank in two iterations

图 2 PageRank 两次迭代作业划分

1.4 MapReduce应用与发展

由于 MapReduce 易操作、可扩展及支持容错等特点,目前有越来越多的企业和应用都将 MapReduce 作为大数据处理平台的核心思想,其中最流行的为 Apache 研发的 HadoopMapReduce 平台.Hadoop[13]作为开源的分布式编程计算框架,其核心由 HDFS[14],MapReduce 和 YARN[15]组成,其中,HDFS(hadoop distributed file system)是适合搭建在廉价集群上可靠的分布式文件存储与传输系统.HDFS 采用 master/slave 架构,将集群中所有服务器存储空间连接到一起,构成了统一的、可以分布式存储海量非结构化数据的空间,具有成本低廉、可靠性强、吞吐量高等特点.YARN 是自 Hadoop 2.0 后提供的新型资源管理器.YARN 将 JobTracker 的系统资源管理调度和单个应用监控跟踪功能分离成两个独立的进程 ResourceManager 和 ApplicationMaster,更好地支持分布式编程计算框架.

随着以 MapReduce 为编程模型的应用越来越多,关于 MapReduce 模型的研究也逐渐增多,其中,主要研究方向集中在以下几方面.

- MapReduce 作业数据放置问题:在 MapReduce 中,数据块放置的方式都是采取随机放置,这种放置方式实现简单,但是可能会因为有关联的数据块放置不均匀,导致 MapReduce 执行效率不高的问题.文献[16]通过构建历史数据访问图得出最优数据放置策略;文献[17]根据被访问次数越多则被访问概率越大的原理提出了一种基于文件分组放置的方法;
- 异构环境下提高 MapReduce 性能问题:在大规模异构环境下,不同计算节点的性能差异会影响整个系统的计算效率.文献[18]提出了自适应的任务调度策略 SAMR(self-adaptive MapReduce),自动寻找执行较慢的节点并进行备份;文献[19]通过拆分较慢节点中的任务并分配给快节点执行的动态负载均衡策略 SkewTune 解决异构问题;文献[20]提出了多种优化方法,保证为 MapReduce 任务分得适当的资源;
- MapReduce 处理迭代问题性能不高问题:由于 MapReduce 每个作业中只包含一对 Map-Reduce 任务及中间数据必须存入磁盘的特性,MapReduce 在计算迭代问题时性能会显著降低.为此,在 MapReduce 基础上产生了很多针对迭代算法改进的类 MapReduce 框架,其中代表性的有 Twister[21],Haloop[22]和 iMapReduce[23]等.

2 Spark

2.1 Spark背景

Spark 是一种基于内存的开源计算框架,2009 年诞生于美国加州大学伯克利分校 AMPLab,在 2010 年正式开源,并于 2013 年成为了 Apache 基金项目,到 2014 年便成为 Apache 基金的顶级项目.自发布以来,Spark 已经被 Yahoo,eBay 和 Netflix 等多家公司在 8 000 多个节点的集群上处理了 PB 级的数据.

在 Spark 中,核心抽象概念就是弹性分布式数据集 RDD(resilient distributed datasets)[24],该抽象是分布在集群上的只读型可恢复数据集.用户可以利用 Spark 中的转换(transformation)和动作(action)对其进行操作,也可以长期保存在内存中不被回收,这样,再次使用这部分内容时,不需要再次创建此 RDD.这也是 Spark 在迭代问题中的性能表现要高于 MapReduce 的原因.RDD 通过一种血统(lineage)关系来完成容错:如果一个 RDD 丢失,那么

这个丢失的RDD有充足的信息知道自己是如何从其他RDD转换而来的,这样便可以通过再次计算得到丢失的RDD.Spark 是由 Scala 语言实现的,而 Scala 是一种基于 JVM 的函数式编程语言,提供了类似 DryadLINQ[25]的编程接口.同时,Spark 还通过修改版的 Scala 解释器提供交互式编程,用户可以自由定义集群中的 RDD、函数、变量以及类.

Spark 在处理迭代和低延迟问题时,能够在保证稳定性和容错性的同时提高系统的运行速度[26],适用场景包括:

- 迭代式算法:很多迭代式算法都会对相同的数据进行重复计算从而得到最优解.MapReduce 计算框架把每次迭代划分成一个或多个 MapReduce 作业(job),而每次迭代都要从磁盘重新加载数据,导致系统效率不高;而 Spark 可以把需要重复计算的数据缓存到内存中加快计算效率;
- 交互式数据分析:用户经常会用 SQL 对大数据集合做临时查询(ad-hoc query).Hive 把每次查询都当作一个独立的 MapReduce 作业,并且从磁盘加载数据,有很大的延迟;而 Spark 可以把数据加载到内存中,然后重复的查询;
- 流应用:即需要实时处理的应用,这类应用往往需要低延迟、高效率.

2.2 弹性分布式数据集(resilient distributed datasets,简称RDD)

RDD 是一种分布在集群中的只读型对象集合,Spark 将创建 RDD 的一系列转换记录下来,如果 RDD 的某个分区或者部分数据丢失,可以根据其父辈 RDD 重建来进行容错,这种策略称为血统(lineage).RDD 的优点有:

- RDD 只能从外部存储或转换(transformation)操作产生,相比于分布式共享内存(DSM),可以更高效地实现容错[27],对于丢失数据,只需根据血统就可重新计算出来,而不需要设置特定的检查点(checkpoint)[28];
- RDD 的只读不变性可以实现类 MapReduce 的预测式执行[3];
- RDD 基于数据的本地性的任务划分调度策略提高了系统性能;
- RDD 是可序列化的,当内存不足时,可自动把 RDD 存储于磁盘上.

创建一个新的 RDD 只能通过以下 4 种方法.

- 从内存集或者外部存储中读取数据创建一个初始 RDD;
- 通过并行化 Scala 集合,即,把一个集合切分成很多份并发送到各节点;
- 通过把 RDD 持久化.默认状态下,RDD 是惰性的,只有在遇到行动(action)操作时,RDD 才被物化,执行完后即被释放.用户可以通过缓存(cache)或保存(save)操作使 RDD 持久化;
- 对其他 RDD 进行转换(transformation),得到一个新的 RDD.在 Spark 中,不同于 MapReduce 只定义了 Map 和 Reduce 两种操作,Spark 定义了大量的操作供用户选择.而这些操作又分为两类:转换和行动,表 1 展示了部分转换和行动操作及其含义.

Table 1 Some transformations and actions of RDD in Spark[24]

表 1 Spark 中 RDD 的部分转换和行动操作[24]

转换 (transformation)	map(func)	原 RDD 每个元素通过 func 函数转换得到一个新的 RDD
	flatmap(func)	与 Map 转换类似,但返回一个 Seq 对象
	filter(func)	原 RDD 每个满足 func 函数的元素组成一个新的 RDD
	distinct([numTasks])	原 RDD 去重得到新 RDD
	union(other)	取两个 RDD 并集得到新的 RDD
	intersection(other)	去两个 RDD 交集得到新的 RDD
行动 (action)	count()	返回原 RDD 中元素个数
	collect()	返回原 RDD 中所有元素为数组
	reduce(func)	原 RDD 中所有元素通过 func 返回最终值
	SaveAsTextFile(Path)	将原 RDD 数据通过路径 Path 写入文件

转换和行动都是针对 RDD 的一系列操作.转换操作是将一个 RDD 转换成一个新的 RDD;而行动操作则是对 RDD 进行计算,然后将结果返回驱动程序.RDD 并不总需要被物化或者进行实际的操作,这是 RDD 最为重要

的一个特性之一.RDD 中只包含一些必须的信息:其对应的数据存放位置、该 RDD 的父辈 RDD 以及该 RDD 是如何从其父辈 RDD 转换而来的.在实际运算中,所有的转换操作都是惰性的,不会立刻执行,Spark 会记录下对 RDD 的全部转换操作,直到有行动操作要求返回结果时才会执行记录的全部操作.这种策略被称为懒惰(lazy).也就是说 Spark 中的转换操作并不是实时的,它需要一个触发因子,这个触发因子就是行动.不同于 MapReduce,Spark 中会最大化地利用集群中有限的资源.例如,Map 操作的结果往往是 Reduce 的输入.事实上,我们并不关心 Map 的结果,而是关心 Reduce 的返回值.懒惰策略减少了不必要的磁盘 I/O 以及返回.

以统计单词“XindongWu”出现的次数问题来解释 RDD 的概念,伪代码如下.

```
val WordCountResult=
  sc.textFile(“Introduction”,4)
  //从“Introduction”文档读取内容并分割
    .flatMap(line⇒line.split(“·”))
    //拆分数据,以空格为拆分条件
      .map(“XindongWu”⇒(“XindongWu”,1))
      //将单个单词转化成键值对形式
        .reduceByKey(_+_)
        //统计“XindongWu”出现次数
WordCountResult.saveAsTextFile(“wordcount_result”)
//输出结果
```

首先,由外部文件读取数据创建一个初始 RDD——MappedRDD;然后,由第 1 个转换操作 flatMap 得到 FlatMappedRDD;再由转换操作 map 形成 MappedRDD;最后,通过转换操作 reduceByKey 得到 ShuffledRDD.至此,所有转换操作不会被实例化,每一个 RDD 只会记录下其父 RDD 及相应的转换操作,当出现行动操作 saveAsTextFile 时,之前记录下的所有转换操作才会被真实地执行.在此例中,共有 3 次转换操作和 1 次行动操作并产生了 4 个 RDD.具体流程图如图 3 所示.

Fig.3 Counting process for word ‘XindongWu’

图 3 统计单词“XindongWu”流程图

2.3 RDD之间的依赖关系

在 MapReduce 中,不同作业中数据块的依赖关系是不同的,可能是一对一的依赖关系,也有可能是一对多或者其他更复杂的情况.在 Spark 当中也存在这个问题,为了让系统运行更加高效,Spark 将 RDD 的依赖关系做了具体的分类,即,窄依赖(narrow dependency)和宽依赖(wide dependency).窄依赖是指每一个 RDD 分区能且只能被一个子 RDD 分区所使用;相反地,宽依赖则指一个 RDD 分区可以被至少两个子 RDD 分区所使用.

如图 4 所示,每一个阴影矩形代表一个 RDD 分区,由多个 RDD 分区组成一个完整的 RDD.左边 3 类依赖关系属于窄依赖,右边两种属于宽依赖.这样划分的原因有两个.

(1) 在窄依赖中,因为每一个父 RDD 分区只能被一个子 RDD 分区所使用,所以父辈 RDD 分区中的数据全部是与对应的子 RDD 分区相关的,不存在冗余数据;相反,在宽依赖中,因为一个父 RDD 分区会被多个子 RDD 分区所使用,所以父 RDD 分区中所存储的数据除了和丢失的子 RDD 分区相关外,还会有部分数据是用来计算出其他的子 RDD 分区.所以在宽依赖中,为了更好进行容错,中间结果一般需要存入磁盘;

(2) 从定义和图 4 中可以看出:在窄依赖中每一个父 RDD 分区计算完成后可以直接将计算结果传给其对

应的子 RDD 分区,无需等待其他 RDD 分区的计算完成;而在宽依赖关系中,因为一个子 RDD 分区需要用到多个父 RDD 分区的计算结果,所以必须等待所有的 RDD 分区计算完成后才能进行子 RDD 的计算.这种无需等待的运算过程叫做管道化操作(pipeline).

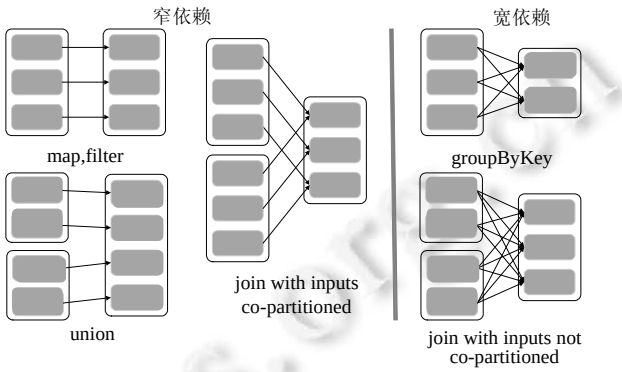

Fig.4 RDD dependency[24]

图 4 RDD 依赖关系[24]

2.4 Spark调度策略

MapReduce 中,将一个 Map 操作和 Reduce 操作称做一个作业(job),且每次作业的中间结果必须进行磁盘存取操作.在 Spark 中,对作业的定义则是:通过特定的行动触发所生成的步骤的集合称为一个 job.只有在遇到行动时才会触发之前所有转换的实际操作,而一个作业是指这一系列转换操作和最后的行动操作所组成的集合.

在一个 job 中,Spark 会根据宽窄依赖关系划分出不同的阶段(stage),然后再根据 RDD 自身保存的依赖关系形成一个有向无环图(DAG)进行调度.概括来说,阶段内部的 RDD 全都为窄依赖关系,阶段之间为宽依赖关系.在进行调度时,Spark 规定每一个阶段如果没有所依赖的父阶段,可以直接进行计算;否则必须等待父阶段计算完成后再进行计算.

2.5 Spark应用与发展

为了让 Spark 能够适应更多的应用场合,开发者以 Spark 为基础设计了不同的组件,包括 Spark SQL[29], Spark Streaming[30],Machine Learning(MLlib)[31]以及 GraphX[32,33],其中:Spark SQL 是结构化数据查询模型,即,支持分布式类 SQL 语句的数据管理平台;Spark Streaming 通过将输入数据进行切分处理方式,利用离散流数据(discretized steams)模型实现增量流数据计算.Mllib 是 Spark 的机器学习库,可以支持超过 50 种常见机器学习算法的分布式训练模型,如决策树模型、LDA 文档主题生成模型(latent dirichlet allocation)和交替最小二乘矩阵分解模型(alternating least squares matrix factorization)的常见分布式算法都可以利用 Mllib 完成;而 GraphX 提供类似 Pregel[34]和 GraphLab[35]的图计算接口,通过将图数据切分成 RDD 进行分布式计算.正是因为拥有众多的组件和完善的生态圈,Spark 得以在如大规模垃圾邮件检测(large-scale spam detection)[36]、图像处理(image processing)[37]和基因数据处理(genomic data processing)[38]等众多研究问题和领域中得到广泛应用.

3 相关工作

目前,在大数据领域已经有很多关于 MapReduce 与 Spark 比较的研究工作,现有比较研究工作主要集中在针对不同的大数据评估指标进行对比以及在各种应用情景中的适用性比较.笔者将现有的比较研究内容做出分析与总结,并针对不同情形下两种分布式模型的选择给出合理的建议.

3.1 大数据评估指标下的对比

(1) 运行时间

运行时间是在系统中运行相应的算法所耗费的总时长.在 MapReduce 与 Spark 架构上运行不同的算法,其运行时间差异也较大.因为 Spark 是轻量级的、基于内存计算的开源的集群计算平台,在默认情况下,Spark 中的数据都会基于内存进行存储;而 MapReduce 每一步 Map 或 Reduce 操作的结果都需要存入磁盘.所以相对而言,在一般情形中,基于内存存储的 Spark 运行速度较快.

文献[39,40]在 MapReduce 与 Spark 平台上运行了 PageRank,*K*-means 和 Grep 等算法并做了相应的实验对比,得出 Spark 在基于内存计算时效率要高于 MapReduce.但是过多的内存需求也会带来相应的问题,如在内存资源紧缺或者数据规模较大的情况下,Spark 的性能会出现大幅下降甚至无法完成计算[41].

(2) 资源消耗

在大数据领域中,不同的算法对计算机的性能要求不尽相同,如数据的排序与查询需要多次访问数据进行 I/O 密集型任务,而 *K*-means,PageRank 等算法需要大量的迭代计算,属于计算密集型任务.在不同问题中,磁盘读写速度、CPU 速度和网络带宽都可能是限制算法性能的瓶颈[42].当解决同一问题时,MapReduce 在内存、网络以及磁盘的占用率上都要低于 Spark.更低的资源消耗不仅可以降低系统对硬件的要求,也保证 MapReduce 可以在同一集群中与其他应用协作计算互不干扰[43].

(3) 容错性能

容错技术会占用计算机资源,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失败.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[44].在 MapReduce 中,每一步操作的结果都会被存入磁盘,虽然需要大量的磁盘 I/O 操作,但是在计算出现错误时可以很好地从磁盘进行恢复;而 Spark 所有计算都基于内存存储,因为内存中的数据会不定期进行清理,所以当某一步计算出现数据丢失时,Spark 需要根据 RDD 中的信息进行数据的重新计算,会耗费一定的资源.文献[45]比较了在 MapReduce 与 Spark 上运行 Sort 和 *K*-means 算法时的容错情况,无论是在单节点还是多节点情况下,MapReduce 的容错率都要优于 Spark.

3.2 不同应用情景下的对比

在大数据领域中,当遇到不同问题时,用户应该根据自身问题的需要选择合适的分布式模型.以下我们将通过对不同情况进行具体分析并给出相应的建议.

(1) 问题类型

从问题角度出发,根据 MapReduce/Spark 的特性,我们将常见的问题做以下划分.

- 批处理问题与交互式问题

批处理问题通常指针对静态离线的数据进行处理,在处理过程中,用户需求及数据不会产生变化的问题,如 WordCount、倒排索引和 URL 访问频率统计等问题都属于这一范畴;而交互式问题则要求在处理问题过程中用户可以和系统进行交互式访问并及时响应.针对批处理问题,通常可以选择 MapReduce 模型进行处理,Spark 则更多地应用在交互式问题上.因为 MapReduce 作业划分策略以及中间结果必须存入磁盘等特性,无法在很短的时间内对交互访问进行响应;而 Spark 的任务调度策略减少了大量不必要的磁盘操作,可以较好地进行交互式操作.

- 迭代问题与非迭代问题

迭代问题与非迭代问题的主要区别在于迭代问题需要对某部分数据进行重复的操作以得到最终的结果,如在 PageRank 问题中每次迭代都需要对 Link 和 Rank 数据进行相同的操作.由于 MapReduce 处理迭代问题时每次迭代过程都需要磁盘读取操作,效率较低;而 Spark 每次迭代结果无需存入磁盘,并且允许用户将常用数据存入内存,使得 Spark 在处理迭代问题时效率要高于 MapReduce.

(2) 数据类型

从数据角度出发,我们选取了以下几种数据类型进行说明.

- 键值对数据

在 MapReduce 模型中,所有的数据都被转化为键值对类型进行分布式处理,如 WordCount 算法中,每个单词

都被转化为⟨*key*,*value*⟩形式,所有的 Map 和 Reduce 操作都是建立在键值对形式的基础上.如果待处理数据能很好地以键值对形式进行表现和处理,那么 MapReduce 可以较好地胜任这类问题.

- 图数据

在社交网络和商品推荐场景中,数据通常是以图的形式表现出来,数据内部关联度可能较高,如社交网络不同用户之间的相关性.在对这样的数据进行处理时,如果转化成键值对形式会引入大量的聚集(aggregation)和连接(join)操作,带来大量的计算和数据迁移,导致效率不高.针对此类问题,Gonzalez 等人于 2013 年构建了基于 Spark 的高效图计算模型 GraphX.GraphX 利用 Spark 框架提供的内存缓存 RDD、任务调度策略以及基于依赖关系的容错等特性,实现了高效健壮的图计算框架.Spark GraphX 定义了图的数据模型以及一系列针对图的并行计算操作,使得图数据可以方便快速地进行存储与计算.

- 流数据

在一些特定的场景中(如实时用户推荐、用户行为分析)对实时性要求很高,要求系统能够快速地处理实时的数据流,通常每次数据处理的时间间隔在数百毫秒到数秒之间.显然,MapReduce 基于磁盘的计算模型不能满足这种实时的需求.而基于 Spark 的流式框架 Spark Streaming 则是专门用来处理此类问题.其原理是:将流数据分割成数据片段封装到 RDD 中,然后以类似批处理的方式对这些数据片段进行操作.Spark Streaming 在本质上仍然是一种批处理方式,但由于 Spark 可以基于内存达到较快的速度从而获得准实时的特性.

(3) 数据规模

在 MapReduce 中,一个完整的问题会被划分为一个或多个作业,每个作业运行时只会占用少量内存,且运行完成后会将结果写入磁盘并释放内存中的数据.而 Spark 在运行过程中会产生大量的 RDD,所有 RDD 都会默认先存入内存中.由于 RDD 只读不可修改的特性,随着计算的进行不断会有新的 RDD 生成并存入内存,所以在处理相同问题时,Spark 内存的需求量会远远大于 MapReduce.并且,Spark 的内存需求会随着问题的数据规模增大而增加,甚至在数据规模过大时,Spark 会无法正常完成运行过程,但是 MapReduce 可以很好地应对这种问题[41].所以在数据规模较大时,考虑到系统的稳定性以及内存消耗问题,应该选择对内存需求更小的 MapReduce 框架;而在处理轻量级数据时,Spark 可以根据其内存运算的优势达到更好的效果.MapReduce 与 Spark 在不同情境下的对比与选择见表 2.

Table 2 A comparison of MapReduce with Spark in different cases

表 2 不同情况下 MapReduce 与 Spark 比较

	问题类型				数据类型			数据规模	
	批处理	交互式	非迭代	迭代	键值对	图	流	大	小
MapReduce	✓		✓		✓			✓	
Spark		✓		✓	✓	✓	✓		✓

以上针对 MapReduce 与 Spark 的比较研究主要集中在实验性能对比方面,但是具体的实验对比结果只适用于其特定的参数配置,不具有普遍性,并且缺少对实验结果进行相应的原理分析.后文我们将以 WordCount 和 PageRank 算法为例对 MapReduce 与 Spark 进行原理分析和比较.

4 WordCount 问题的分布式处理

4.1 问题描述

WordCount 问题是分布式算法中最为经典的问题之一,其基本要求是,从给定文档中统计出每个单词出现的次数.在 MapReduce[2]与 Spark[4]的官方文献中也都提到了处理 WordCount 问题的基本思路.我们将通过 WordCount 问题分为以下 3 点比较 MapReduce 与 Spark 各自的优缺点.

- 算法执行时间:算法执行时间是算法最主要的影响因素之一,直接关系到算法的性能和效率;
- 文件数目:在分布式系统中,中间数据都是通过文件的形式进行传输,在总数据量相同的情况下,文件数目越多,文件系统的负担越大,并且随机磁盘 IO 也会严重影响程序的执行效率;

- 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.但同步是数据规约和汇总的前提,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,加快了算法的执行速度.即:在算法执行过程中,同步次数越少,越有利于提高算法的性能和效率.

假设给定文档为“CABDCABCDABCCACDBA”,为了方便表示,其中“A,B,C,D”分别代表不同的单词,要求从该文档中统计出 4 个单词出现的次数.

4.2 MapReduce处理WordCount问题

在第 2.2 节中,我们已经简单描述了 MapReduce 处理问题的大致步骤,但是具体如何进行任务的切分与重组、数据如何在各处理机之间计算与传递是我们所关心的.

图 5 为官方描述的一个 MapReduce 作业的流程图,通过分析该流程图,可以找出影响 MapReduce 作业完成的主要因素.

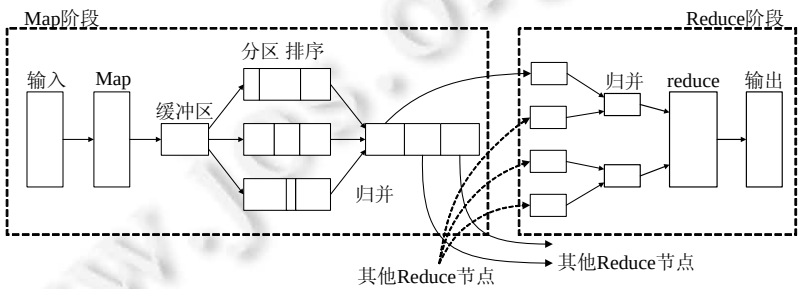

Fig.5 Implementation process of MapReduce

图 5 MapReduce 的执行过程

MapReduce 作业的执行过程主要分为 Map 阶段、中间结果排序与传递阶段和 Reduce 阶段.即,MapReduce 作业执行时间受读取输入文件时间 T_{read}、中间数据排序时间 T_{sort}、中间数据传输时间 T_{trans} 和写输出文件到 HDFS 时间 T_{write} 的影响.通过以上分析,MapReduce 的算法执行时间 T_{MR} 可以表示为

$$T_{MR}=T_{read}+T_{sort}+T_{trans}+T_{write}.$$

假设输入数据大小表示为$|f_{in}|$,输出数据大小表示为$|f_{out}|$,远程文件读写速度为 C_r,T_{read} 可以表示为

$$T_{read}=|f_{in}|\times C_r.$$

同理,T_{write} 可以表示为

$$T_{write}=|f_{out}|\times C_r.$$

在 MapReduce 与 Spark 针对 WordCount 问题的比较过程中,其输入/输出数据是相同的,也就是说,两者的$|f_{in}|$与$|f_{out}|$相同;且我们比较的是 MapReduce 与 Spark 运行机制以及调度策略不同所导致的时间快慢,所以不考虑网络传输速度以及文件读写速度所带来的误差,即:默认为在 MapReduce 与 Spark 中,C_r 的值相同.所以 T_{read} 与 T_{write} 在两种算法中的值近似相同,对两者算法执行时间的差异不会产生影响,我们主要关注 T_{sort} 与 T_{trans} 的比较.

在此例中,MapReduce 的执行过程为:

(1) 首先将数据分块,并转换为键值对形式;

(2) 每个 Map 任务根据 *key* 值对数据进行分区,即,同一个分区的数据将会被发送到同一个 Reduce 节点上;

(3) 对每一个分区进行排序,同时将可以合并的数据进行合并(如两个(A,1)将会被合并成(A,2))

(4) 每一个 Reduce 从各个 Map 节点通过网络传输复制相应的数据;

(5) 将不同 Map 节点的数据进行归并.

具体流程如图 6 所示.

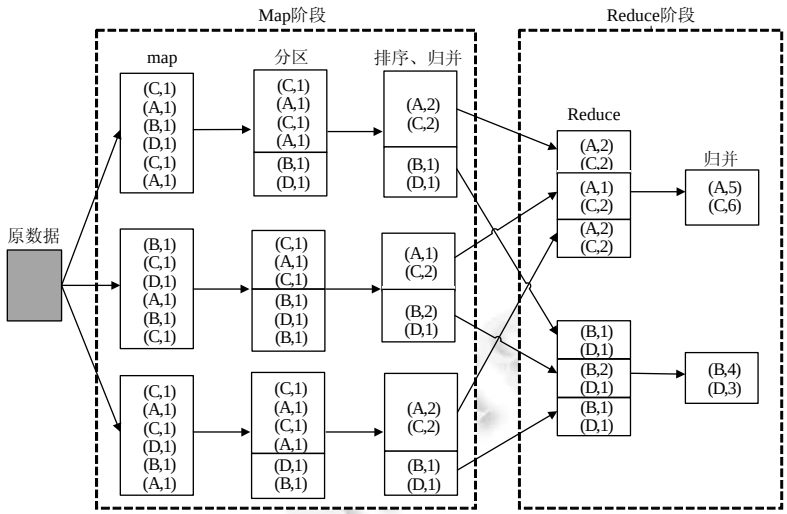

Fig.6 MapReduce flow chart for WordCount

图 6 MapReduce 处理 WordCount 问题流程图

4.3 Spark处理WordCount问题

Spark 处理 WordCount 问题的具体流程见图 7,过程如下.

(1) 第 1 次遍历:生成 RDD 及记录依赖关系

- 从外部文件中读取数据生成第 1 个 RDD(ParallelCollectionRDD),分区数为 3;
- 将每一个分区再进行哈希分块,分块数对应阶段 2 中的任务(task)数目,每一个分块单独形成一个文件进行数据传输;
- 将中间数据文件进行 shuffle 形成 ShuffledRDD,并在 ShuffledRDD 中记录下其父 RDD 以及依赖关系;
- 通过对 ShuffledRDD 进行转换操作生成 MapPartitionsRDD,并在 MapPartitionsRDD 中记录下其父 RDD 以及依赖关系;
- 因为在 MapPartitionsRDD 后为行动操作,则本作业中所有 RDD 生成完毕,第 1 次遍历结束.

(2) 第 2 次遍历:划分阶段与任务

- 根据第 1 次遍历每一个 RDD 中记录的父 RDD 信息及依赖关系,从 MapPartitionsRDD 开始向前进行第 2 次遍历;
- 首先进行阶段划分,若向前遍历时遇到窄依赖关系,则将依赖的 RDD 加入到当前阶段;若遇到宽依赖关系,则形成一个新的阶段继续向前遍历.在本例中,ParallelCollectionRDD 与 ShuffledRDD 为宽依赖关系,ShuffledRDD 与 MapPartitionsRDD 为窄依赖关系,所以会生成两个阶段,阶段 1 中包括 Parallel CollectionRDD,阶段 2 中包括 ShuffledRDD 和 MapPartitionsRDD;
- 在每一个阶段内部进行任务划分,任务数目为阶段内部 RDD 的分区数,在阶段 1 中有 3 个任务,阶段 2 中有 2 个任务;

(3) 第 3 次遍历:执行任务

根据第 2 次遍历划分的阶段与任务进行作业的具体执行,首先执行阶段 1 中的任务,执行完毕后再执行阶段 2 中的任务,输出数据.

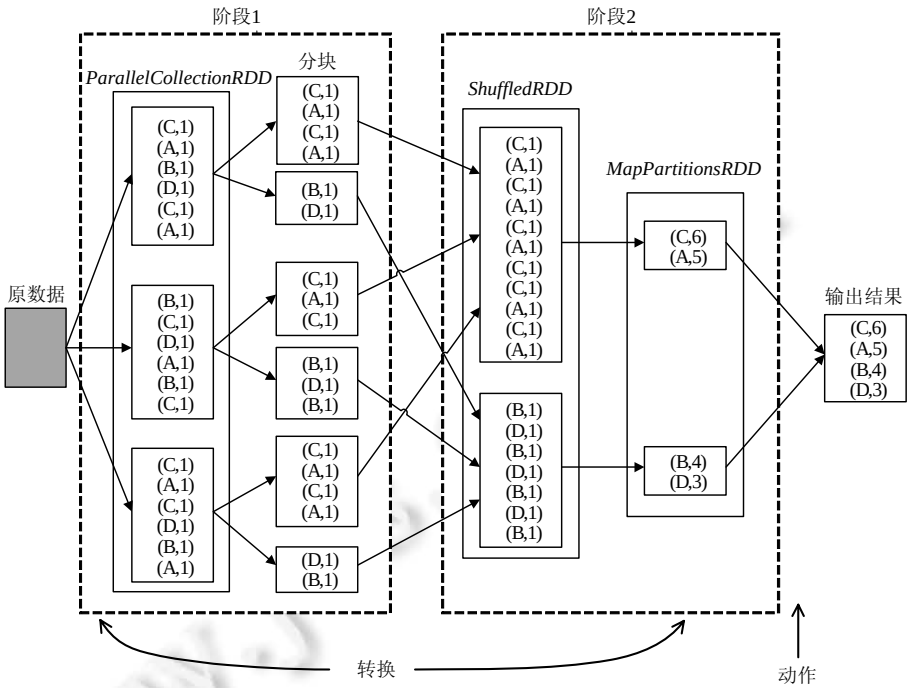

Fig.7 Spark flow chart for WordCount

图 7 Spark 处理 WordCount 问题流程图

4.4 WordCount问题分布式处理的比较与总结

在分析完 MapReduce 与 Spark 处理 WordCount 问题的流程后,我们从算法执行时间、同步次数和文件数目这 3 个方面进行 MapReduce 与 Spark 的比较,并进行总结.

- 算法执行时间

为了分析算法执行时间,我们需要讨论两者的中间数据排序时间 T_{sort} 与中间数据传输时间 T_{trans}.在分析两者的工作流程时可以看到:排序操作只有在 MapReduce 中存在,在 Spark 中并没有要求对中间数据进行排序操作.而 MapReduce 中规定每次 shuffle 必须对中间结果进行排序,主要是为了可以将中间结果进行初步的归并操作,使得需要传输的数据量减少,降低网络传输压力;并且可以保证每一个 Map 任务只输出一个有序的中间数据文件,减少文件数目.

在 MapReduce 中,在 Map 阶段对每一分区的数据进行排序,在 Reduce 阶段对不同 Map 任务的输出数据进行归并.共有 m 个 Map 任务,平均每个 Map 任务有 M 条数据,平均每个 Reduce 任务有 R 条数据.可以得到 $T_{sort\text{-}MR}$:

$$T_{sort\text{-}MR}=M\times\log M+R=O(M\log M).$$

针对此例,在 3 个 Map 任务中,第 3 个 Map 任务进行比较次数最多,共做了 4 次比较(快速排序).而在 Spark 中因为没有中间数据排序过程,所以,

$$T_{sort\text{-}Spark}=0.$$

再来讨论两者中间数据传输时间 T_{trans},中间数据传输是指由 Map 任务的执行节点发送到 Reduce 任务的执行节点的数据,所以 T_{trans} 由 Map 任务输出的中间数据大小$|D|$和网络文件传输速度 C_t 决定,即:

$$T_{trans}=C_t\times|D|.$$

在不考虑网络传输速度带来性能差异的情况下,默认在 MapReduce 与 Spark 中 C_t 大小相等,则 T_{trans} 与$|D|$成正比.在此例中,MapReduce 中间数据共有 12 条键值对数据,而 Spark 则有 18 条.Spark 中的$|D|$要大于 MapReduce,则相应的 Spark 的 T_{trans} 也要大于 MapReduce.

- 文件数目

在分布式系统中,中间数据是以文件的形式进行存储的.文件数目过多,会严重占用内存并影响磁盘的 IO 性能,所以在分布式系统中,应当尽量减少中间数据文件的数量.

对于 MapReduce 来说,每一个 Map 只会产生一个中间数据文件,不同分区的数据都会存在一个文件中,之所以可以做到这样,是因为 MapReduce 的排序操作使得分区内数据有序,不同的分区数据只需要通过增加一个偏移量便可以区分.所以在 MapReduce 中,文件数目等于 Map 任务数量 m;而在 Spark 中,因为没有对数据进行预排序,所以只能将不同分区的数据放在不同的文件中,则每一个 Map 任务都会生成 r 个文件,其中,m 为 Reduce 任务数量.则 Spark 总文件数目等于 $m\times r$.

此例中,MapReduce 的文件数量为 3,而 spark 的文件数量为 6.

- 同步次数

同步模型要求所有节点完成当前阶段后才可以进行下一阶段,这严重限制了计算性能.在 MapReduce 中,所有的步骤都严格遵守同步模型,即,Reduce 操作要在所有的 Map 操作结束后进行;

异步模型则可以使多个节点形成单独的流水线独立运行,大大加速了算法的执行.Spark 考虑到了这一点,它规定在无需进行网络传输的时候,有窄依赖关系的节点形成流水线独立运行;在需要网络传输的时候,遵守同步模型.但是这种策略所带来的缺点是额外的遍历次数.

在此例中,因为只存在一次 Map 与 Reduce 操作,只有一次 shuffle,所以 MapReduce 与 Spark 在同步次数上相同,皆为一次.但是 Spark 因为要建立操作之间的依赖关系,所以需要付出额外的代价,即,比 MapReduce 要多遍历两次.

通过表 3 可以看到:在处理 WordCount 问题时,MapReduce 在排序上消耗了更多的时间,而在中间数据传输、文件数目以及同步次数上效率都要优于 Spark.

Table 3 Performance comparison between MapReduce and Spark with WordCount

表 3 MapReduce 与 Spark 处理 WordCount 问题之性能比较

	算法执行时间				文件数目	同步次数
	T_{read}	T_{write}	T_{trans}	T_{sort}		
MapReduce	相同	相同	$C_t\times 2$	4 次比较	3	1(1 次遍历)
Spark			$C_t\times 18$	0 次比较	6	1(3 次遍历)

5 PageRank 问题的分布式处理

5.1 问题描述

迭代问题一直是大数据领域中最为重要的问题之一,如常见的 PageRank 算法以及 k-Means 算法,通常是将初始数据经过很多次的重复操作逼近所需的目标结果.如何在迭代问题中通过有效的数据存储优化达到加快系统效率的目的,是大数据问题中不可忽略的一项内容.PageRank 是由谷歌的创始人 Larry Page 发明,用来衡量网站页面的重要性[46].其基本思想来源于参考文献的引用网络,被引用越多的文献往往更加重要和权威.PageRank 基本思想基于两个前提.

(1) 一个网页在两种情况下可能是重要的网页:被很多网页所引用或者被一个或几个重要网页所引用.每一个网页的重要性被平均传递给其指向的网页,每个网页的重要性会被具体量化成一个权重(Rank);

(2) 假定每次开始时都会随机地访问页面,往后会根据网页的链接访问其他网页,访问下一个网站页面的概率是该网站页面的 Rank 值.而这也基本符合人们在日常生活中浏览网页的习惯.

基本的 PageRank 算法需要两个重要的数据结构:Rank(每一个网页的权重,初始值唯一)和 Link(每个页面之间的指向关系).PageRank 具体的算法可以描述如下.

(1) 将每一个网页的 Rank 值设为 1;

(2) 对于每一次迭代,将每一个网页的权重贡献(Rank/指向的网页数量)发送给指向的网页;

(3) 对于每一个网页,将收到的权重贡献相加得到 *contribs*,重新计算每一个网页的 *Rank*=(1−*p*)+*p**contribs(其中,*p* 为跳跃因子,可以根据具体情况设为[0,1]中的任意值);

(4) 循环操作第 2 步与第 3 步,直到每个网页的 *Rank* 值趋于收敛,结束.

为了可以有更直观的理解,图 8 举出了一个具体的例子,并且动态描述了 10 次迭代过程.在此例中,跳跃因子 *p* 的值取 0.8,所有结果保留 3 位有效数字.

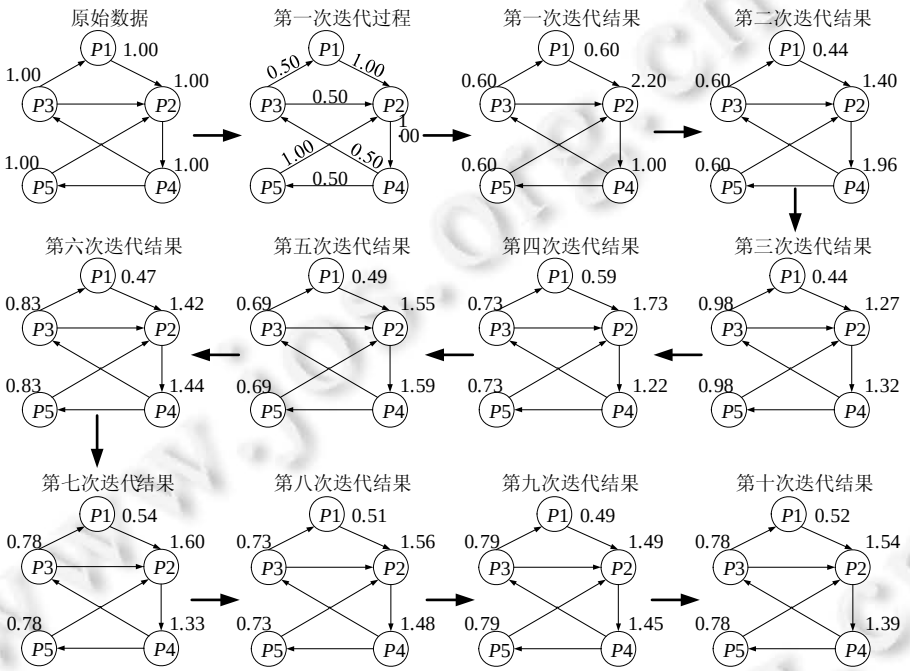

Fig.8 Ten iterations of PageRank

图 8 PageRank 的 10 次迭代

通过计算,此例中网页的权值是收敛的,经过 10 次迭代后的权值误差可以缩小到 0.1.第 10 次迭代计算后,网页 *P*1~*P*5 的权值依次为(0.52,1.54,0.78,1.39,0.78).

与 WordCount 问题类似,我们同样通过算法执行时间、文件数目和同步次数这 3 个方面来比较 MapReduce 与 Spark 在处理 PageRank 问题时性能的差异.因为 PageRank 是迭代问题,为了更好地分析迭代次数的变化给两种算法性能带来的影响,将分别讨论一次迭代、两次迭代和 10 次迭代情况下,MapReduce 与 Spark 在执行时间、文件数目和同步次数这 3 个方面的性能差异.

5.2 MapReduce处理PageRank问题

PageRank 问题在 MapReduce 中的操作步骤要比图 8 复杂很多,如何根据每个网页的 Rank 和 Link 数据计算出每个网页的最终 Rank 并且分布到多台处理机上去执行,是 MapReduce 所关心的.在图 9 中,我们展示了 PageRank 在 MapReduce 中的一次迭代过程,在此例中,我们每一步操作都假设分布到 3 台处理机上去操作,其他情况下可以根据具体需要进行变更.具体算法如下[41].

- Job1:通过一次 Map 和 Reduce 操作,将每个网页的 Rank 和 Link 数据进行合并,因为是第 1 次迭代,所以网页的初始 Rank 值设为 1;
- Job2:根据 Job1 的合并结果进行一次 Map 和 Reduce 操作,计算出每一个网页接收的权重贡献,并加入跳跃因子进行计算,得到每个网页一次迭代后的新 Rank 值.

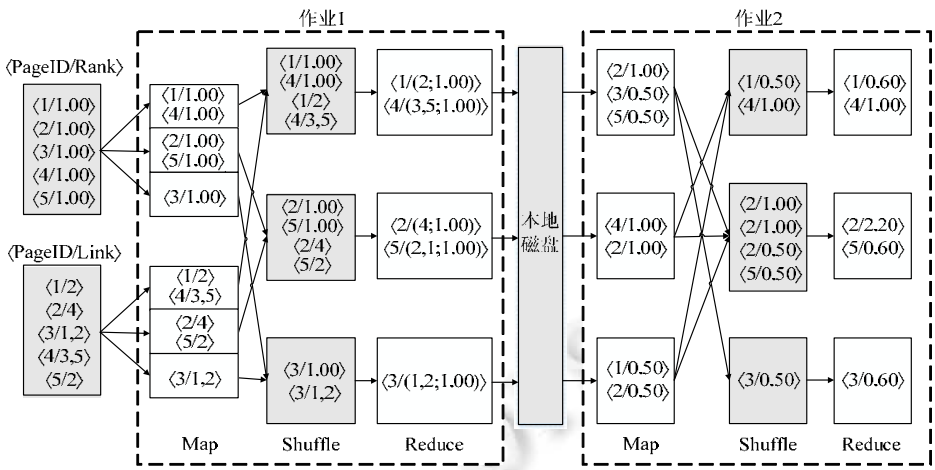

Fig.9 MapReduce for PageRank with one iteration

图 9 MapReduce 处理 PageRank 问题一次迭代

由图 9 可以看到,在 MapReduce 一次迭代过程中,需要注意以下几点.

- 与 WordCount 类似,MapReduce 中每个 Map 任务都需要在输出数据前对中间数据进行排序,使其分区内有序;
- 每一个 Map 节点与 Reduce 节点单独地作为一个任务,不同任务之间独立计算互不干扰,每一个任务的计算结果都要存入磁盘进行数据传输(图 9 中所有阴影部分表示需要存入磁盘的数据);
- 为了保证系统的可靠性和可扩展性,MapReduce 要求所有的 Reduce 需要在所有 Map 任务结束后才可以进行计算;并且只有在作业 1 所有任务完成后,作业 2 才能开始运行任务.

MapReduce 中,PageRank 的两次迭代过程被划分为 4 个作业进行计算,共有 23 个任务、4 次 shuffle 操作. 图 10 为 MapReduce 处理 PageRank 问题两次迭代的流程图.

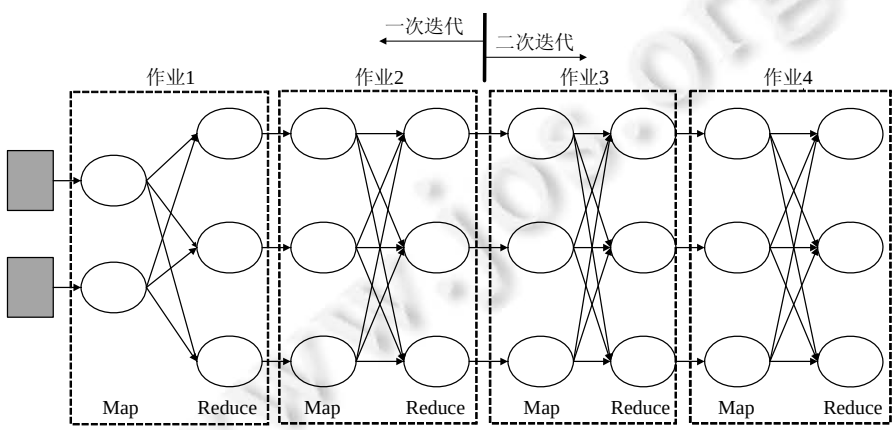

Fig.10 MapReduce for PageRank with two iterations

图 10 MapReduce 处理 PageRank 问题两次迭代

5.3 Spark处理PageRank问题

Spark 在处理迭代问题时,相较于 MapReduce 做了很多改进:首先,在内存足够的情况下,Spark 允许用户将常用的数据缓存到内存中,加快了系统的运行速度;其次,Spark 对数据之间的依赖关系有了明确的划分,根据宽依赖与窄依赖关系进行任务的调度,可以实现管道化操作,使系统的灵活性得以提高.

图 11 与图 12 分别是 Spark 处理 PageRank 问题的一次迭代与两次迭代的流程图.

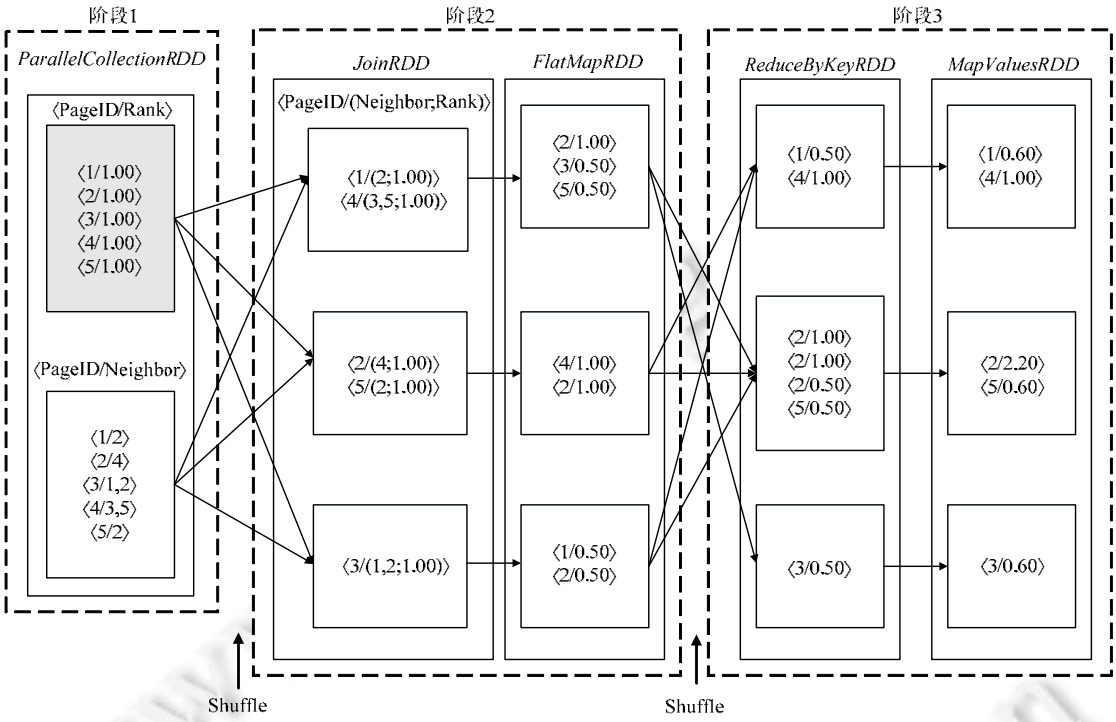

Fig.11 Spark for PageRank with one iteration

图 11 Spark 处理 PageRank 问题一次迭代

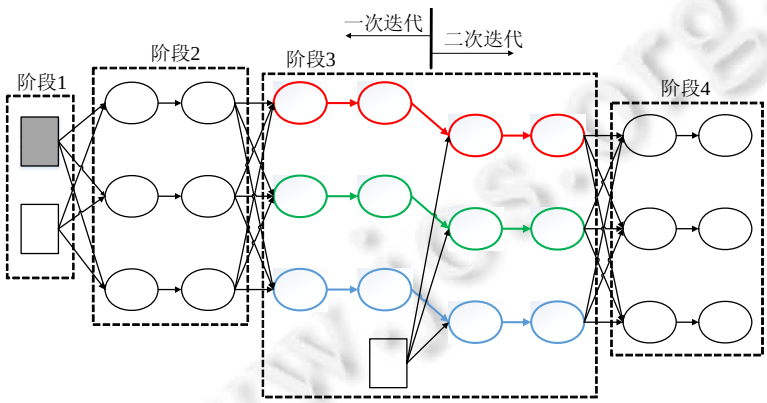

Fig.12 Spark for PageRank with two iterations

图 12 Spark 处理 PageRank 问题两次迭代

通过图 11 可以看到:在一次迭代过程中,Spark 需要 5 次转换(transformation)操作,共产生 5 个 RDD,与 MapReduce 一样需要进行两次 shuffle 操作,被划分为 3 个阶段.

与 MapReduce 不同的是,在图 12 的两次迭代中,Spark 可以将具有窄依赖关系的 RDD 分区分配到一个任务中进行管道化操作,任务内部数据无需通过网络传输且任务之间互不干扰.如在图 12 的阶段 3 中,所有分区节点根据依赖关系被分为 3 个任务(分别标注为红色、绿色和蓝色).在 PageRank 问题的两次迭代计算过程中,作业被划分为 4 个阶段,共有 11 个任务、3 次 shuffle 操作,Spark 相较于 MapReduce 有效地减少了任务之间的等

待时间以及中间数据传输数量.

5.4 PageRank问题分布式处理的比较与总结

与 WordCount 问题类似,我们将分别在一次迭代、两次迭代和 10 次迭代情况下,通过算法执行时间、文件数目和同步次数对 MapReduce 与 Spark 进行比较.

- 算法执行时间

首先讨论中间数据排序时间 T_{sort},从图 9 中可以看出:MapReduce 一次迭代需要两次 Map-Reduce 操作,第 1 次 Map-Reduce 操作中,每个 Map 都需要进行两次比较;而第 2 次 Map-Reduce 操作中,第 1 个 Map 任务进行的比较次数最多,为两次.则在一次迭代中,MapReduce 在中间数据排序上要进行 4 次比较,在两次迭代中,MapReduce,需要进行 8 次比较,在 10 次迭代中,MapReduce 需要进行 40 次比较.而 Spark 在 shuffle 时不要求中间数据有序,所以其比较次数为 0.

再讨论两者中间数据传输时间 T_{trans},在 WordCount 问题中,MapReduce 需要传输的中间数据量要小于 Spark;但在迭代问题中,MapReduce 的优势将不再明显.其原因是:Spark 根据依赖关系采用的任务调度策略,使得 shuffle 次数相较于 MapReduce 有了显著降低.

在 MapReduce 一次迭代过程中,需要进行两次数据传输,数据量分别为:$|D_1|=10$,$|D_2|=7$,$T_{trans\text{-}MR}=C_t\times 17$;在两次迭代中,MapReduce 需要进行 4 次数据传输,$|D_1|=10$,$|D_2|=7$,$|D_3|=10$,$|D_4|=7$,$T_{trans\text{-}MR}=C_t\times 34$;在 10 次迭代中,$T_{trans\text{-}MR}=C_t\times 170$.Spark 一次迭代过程中与 MapReduce 相同,需要进行两次传输:$|D_1|=10$,$|D_2|=7$,$T_{trans\text{-}Spark}=C_t\times 17$;在两次迭代中,需要进行 3 次数据传输,$|D_1|=10$,$|D_2|=7$,$|D_3|=7$,$T_{trans\text{-}Spark}=C_t\times 24$;而在 10 次迭代过程中,共进行了 11 次数据传输,$T_{trans\text{-}Spark}=C_t\times 80$.

- 文件数目

同 WordCount 类似,在需要进行网络传输的时候,MapReduce 的文件数量是 m,而 Spark 的文件数量是 $m\times r$,即,单次 shuffle 过程的文件数量 MapReduce 要少于 Spark.但从整个作业角度来看,Spark 的 shuffle 次数要少于 MapReduce,所以宏观上说,在迭代问题中,MapReduce 文件数量上的优势不如 WordCount 中那么明显.就此例而言,一次迭代过程中,MapReduce 总文件数为 2+3=5,Spark 总文件数为 6+9=15;两次迭代过程中,MapReduce 总文件数为 2+3+3+3=11,Spark 总文件数为 6+9+9=24;10 次迭代过程中,MapReduce 总文件数为 5+6×9=59,Spark 总文件数为 15+9×9=96.

- 同步次数

之前已经描述了同步模型与异步模型之间的区别,在算法执行过程中,同步次数越少、所占比例越小,越有利于算法的局部性能.虽然伴随着算法整体步骤的增加(如 Spark 中更多的遍历次数),但是总的来说会使算法性能得到有效的提升.在一次迭代过程中,MapReduce 与 Spark 同步次数皆为 2 次;在两次迭代过程中,MapReduce 中 4 次 Map-Reduce 操作全部遵守同步模型,同步次数为 4 次(即,每次都要等待上一步所有任务都执行完毕才会执行下一步操作),Spark 将两次迭代过程划分成 4 个阶段,每个阶段内遵守异步模型,阶段间为同步操作,所以同步次数为 3 次;在 10 次迭代过程中,MapReduce 共有 20 次同步操作,而 Spark 只有 11 次.

通过表 4 可以看到:在一次迭代过程中,MapReduce 与 Spark 在性能上并没有很大的差别;但是随着迭代次数的增加,两者的差距逐渐显现出来.

- 在算法执行时间方面,Spark 性能要好于 MapReduce.与 WordCount 问题不同,Spark 在 PageRank 问题中的中间数据传输数量要低于 MapReduce;
- 在同步次数方面,Spark 同步次数同样低于 MapReduce.

这些优势主要是因为 Spark 根据数据间的依赖关系对任务进行了更为合理的划分,有效地减少了问题中 shuffle 操作的次数.而因为 MapReduce 与 Spark 在 shuffle 操作中的差别,导致 Spark 文件数目要高于 MapReduce.而 Spark 也在不断地进行自我完善:在 Spark0.8.1 版本中引入了 shuffle consolidation[47]机制,有效地减少了 Spark 中间数据文件数量;从 Spark1.1 开始,Spark 开始将 MapReduce 的 Sort-BasedShuffle 机制作为一套备选 shuffle 机制供用户选择,Sort-BasedShuffle 机制能够保证每个 Map 任务节点只输出一个文件,且数据为分区内有序.

Table 4 Performance comparison between MapReduce and Spark with PageRank

表 4 MapReduce 与 Spark 处理 PageRank 问题之性能比较

		算法执行时间				文件数目	同步次数
		T_{read}	T_{write}	T_{trans}	T_{sort}		
一次迭代	MapReduce	相同	相同	C_t×17	4 次比较	5	2
	Spark			C_t×17	0 次比较	15	2
两次迭代	MapReduce	相同	相同	C_t×34	8 次比较	11	4
	Spark			C_t×24	0 次比较	24	3
10 次迭代	MapReduce	相同	相同	C_t×170	40 次比较	59	20
	Spark			C_t×80	0 次比较	96	11

6 总结与展望

本文首先分别阐述了 MapReduce 和 Spark 两种大数据分布式架构的背景、原理以及调度等方面,并从大数据评估指标和应用情景两个方面分别对二者进行对比和总结,针对不同问题需求给出相应的分析和选择.为了更好地理解 MapReduce 与 Spark 在原理上的区别,我们通过 WordCount 和 PageRank 算法分别分析 MapReduce 和 Spark 在处理非迭代和迭代问题时在算法执行时间、文件数目和同步次数上的差异.通过分析结果可以看出:在处理不同问题时,MapReduce 与 Spark 各有优点与不足.在处理 WordCount 问题时,MapReduce 在排序上消耗了更多的时间,而在中间数据传输、文件数目以及同步次数上效率都要优于 Spark;在处理 PageRank 问题时,Spark 通过更合理的任务调度机制,在算法执行时间和同步次数方面要优于 MapReduce.但是因为不同的 shuffle 机制,导致 Spark 的文件数目要高于 MapReduce,而 Spark 也在后期的版本中对此缺点进行了补充和完善.

MapReduce 作为第一代大数据分布式架构,让传统的大数据问题可以并行地在多台处理机上进行计算.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、实现简单和容错性强等特性[48].但是 MapReduce 并不适合处理迭代问题以及低延时问题,而 Spark 作为轻量、基于内存计算的分布式计算平台,采用了与 MapReduce 类似的编程模型,使用 RDD 抽象对作业调度、数据操作和存取进行修改,并增加了更丰富的算子,使得 Spark 在处理迭代问题、交互问题以及低延时问题时能有更高的效率.同样,Spark 也有其不足:如数据规模过大或内存不足时,会出现性能降低、数据丢失需要进行重复计算等缺点.总而言之,随着大数据领域的不断发展和完善,现有的大数据分析技术仍然有大量具有挑战性的问题需要深入研究,而作为大数据领域重要的两种分布式处理架构,MapReduce 与 Spark 都有着不可替代的地位和作用.

References:

[1] Gordon K. What is big data? Itnow, 2013,55(3):12−13.

[2] Dean J, Ghemawat S. MapReduce: Simplified data processing on large clusters. In: Proc. of the 6th Conf. on Symp. on Opearting Systems Design and Implementation. San Francisco: USENIX Association Berkeley, 2004. 137−149.

[3] Rao BT, Sridevi N, Reddy VK, Reddy L. Performance issues of heterogeneous hadoop clusters in cloud computing. Global Journal of Computer Science and Technology, 2011,6(8):1−6.

[4] Zaharia M, Chowdhury M, Franklin MJ, Shenker S, Stoica I. Spark: Cluster computing with working sets. HotCloud, 2010,10 (10-10):1−7.

[5] Dean J. Experiences with MapReduce, an abstraction for large-scale computation. In: Proc. of the 15th Int'l Conf. on Parallel Architectures and Compilation Techniques. Washington: ACM Press, 2006. 1−1.

[6] Srirama SN, Jakovits P, Vainikko E. Adapting scientific computing problems to clouds using MapReduce. Future Generation Computer Systems, 2012,28(1):184−192. [doi: 10.1016/j.future.2011.05.025]

[7] Elghandour I, Aboulnaga A. ReStore: Reusing results of MapReduce jobs in pig. In: Proc. of the 2012 ACM SIGMOD Int'l Conf. on Management of Data. Scottsdale: ACM Press, 2012. 701−704. [doi: 10.1145/2213836.2213937]

[8] Pansare N, Borkar VR, Jermaine C, Condie T. Online aggregation for large MapReduce jobs. Proc. of the Vldb Endowment, 2012, 4(11):1135−1145.

[9] Zaharia M, Borthakur D, Sarma JS, Elmeleegy K, Shenker S, Stoica I. Job scheduling for multi-user MapReduce clusters. Technical Report UCB/EECS-2009-55. Berkeley: EECS Department, University of California, 2009. 1−18.

[10] Zaharia M. Job scheduling with the fair and capacity schedulers. In: Proc. of the Hadoop Summit. 2009. 9.

[11] Nightingale EB, Chen PM, Flinn J. Speculative execution in a distributed file system. ACM SIGOPS Operating Systems Review, 2005,39(5):191−205. [doi: 10.1145/1095809.1095829]

[12] Isard M, Budiu M, Yu Y, Birrell A, Fetterly D. Dryad: Distributed data-parallel programs from sequential building blocks. In: Proc. of the 2nd ACM SIGOPS/EuroSys European Conf. on Computer Systems 2007. Lisbon: ACM Press, 2007. 59−72. [doi: 10.1145/1272998.1273005]

[13] Venner J. Pro Hadoop. Berkeley: Apress, 2009. 1−440. [doi: 10.1007/978-1-4302-1943-9]

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

[15] Vavilapalli VK, Murthy AC, Douglas C, Agarwal S, Konar M, Evans R, Graves T, Lowe J, Shah H, Seth S. Apache hadoop yarn: Yet another resource negotiator. In: Proc. of the 4th Annual Symp. on Cloud Computing. ACM Press, 2013. 1−16. [doi: 10.1145/2523616.2523633]

[16] Wang J, Shang P, Yin J. DRAW: A New Data-gRouping-AWare Data Placement Scheme for Data Intensive Applications with Interest Locality. New York: Springer, 2014. 149−174. [doi: 10.1007/978-1-4939-1905-5]

[17] Amer A, Long DD, Burns RC. Group-Based management of distributed file caches. In: Proc. of the 2002 22nd Int'l Conf. on Distributed Computing Systems. Vienna: IEEE, 2002. 525−534. [doi: 10.1109/ICDCS.2002.1022302]

[18] Chen Q, Zhang D, Guo M, Deng Q, Guo S. Samr: A self-adaptive MapReduce scheduling algorithm in heterogeneous environment. In: Proc. of the 2010 IEEE 10th Int'l Conf. on Computer and Information Technology (CIT). Bradford: IEEE, 2010. 2736−2743. [doi: 10.1109/CIT.2010.458]

[19] Kwon Y, Balazinska M, Howe B, Rolia J. Skewtune: Mitigating skew in MapReduce applications. In: Proc. of the 2012 ACM SIGMOD Int'l Conf. on Management of Data. Arizona: ACM Press, 2012. 25−36. [doi: 10.1145/2213836.2213840]

[20] Cherniak A, Zaidi H, Zadorozhny V. Optimization strategies for A/B testing on HADOOP. Proc. of the VLDB Endowment, 2013, 6(11):973−984. [doi: 10.14778/2536222.2536224]

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

[22] 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]

[23] 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]

[24] Zaharia M, Chowdhury M, Das T, Dave A, Ma J, McCauley M, Franklin MJ, Shenker S, Stoica I. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In: Proc. of the 9th USENIX Conf. on Networked Systems Design and Implementation. San Jose: USENIX Association Berkeley, 2012. 1−14.

[25] Yu Y, Isard M, Fetterly D, Budiu M, Erlingsson Ú, Gunda PK, Currey J. DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. In: Proc. of the 8th USENIX Conf. on Operating Systems Design and Implementation. USENIX Association Berkeley, 2008. 1−14.

[26] Lhoták O, Hendren L. Scaling Java points-to analysis using Spark. In: Proc. of the 12th Int'l Conf. on Compiler Construction. Berlin: Springer-Verlag, 2003. 153−169.

[27] Ananthanarayanan G, Ghodsi A, Shenker S, Stoica I. Disk-Locality in datacenter computing considered irrelevant. In: Proc. of the 13th USENIX Conf. on Hot topics in Operating Systems. California USENIX Association Berkeley, 2011. 12−17.

[28] Bhatotia P, Wieder A, Rodrigues R, Acar UA, Pasquin R. Incoop: MapReduce for incremental computations. In: Proc. of ACM Symp. on Cloud Computing. Cascais: ACM Press, 2011. 1−14. [doi: 10.1145/2038916.2038923]

[29] Armbrust M, Xin RS, Lian C, Huai Y, Liu D, Bradley JK, Meng X, Kaftan T, Franklin MJ, Ghodsi A. Spark sql: Relational data processing in Spark. In: Proc. of the 2015 ACM SIGMOD Int'l Conf. on Management of Data. Victoria: ACM Press, 2015. 1383−1394. [doi: 10.1145/2723372.2742797]

[30] Zaharia M, Das T, Li H, Hunter T, Shenker S, Stoica I. Discretized streams: Fault-tolerant streaming computation at scale. In: Proc. of the 24th ACM Symp. on Operating Systems Principles. Farmington: ACM Press, 2013. 423−438. [doi: 10.1145/2517349.2522737]

[31] Meng X, Bradley J, Yavuz B, Sparks E, Venkataraman S, Liu D, Freeman J, Tsai D, Amde M, Owen S, Xin D, Reynold X, Franklin MJ, Zadeh R, Zaharia M, Talwalkar A. Mllib: Machine learning in apache Spark. The Journal of Machine Learning Research, 2016, 17(1):1235−1241.

[32] Xin RS, Gonzalez JE, Franklin MJ, Stoica I. Graphx: A resilient distributed graph system on Spark. In: Proc. of the 1st Int'l Workshop on Graph Data Management Experiences and Systems. New York: ACM Press, 2013. 1−6. [doi: 10.1145/2484425.2484427]

[33] Gonzalez JE, Xin RS, Dave A, Crankshaw D, Franklin MJ, Stoica I. GraphX: Graph processing in a distributed dataflow framework. In: Proc. of the 11th USENIX Conf. on Operating Systems Design and Implementation. USENIX Association Berkeley, 2014. 599−613.

[34] Malewicz G, Austern MH, Bik AJ, Dehnert JC, Horn I, Leiser N, Czajkowski G. Pregel: A system for large-scale graph processing. In: Proc. of the 2010 ACM SIGMOD Int'l Conf. on Management of Data. ACM Press, 2010. 135−146. [doi: 10.1145/1807167.1807184]

[35] Low Y, Bickson D, Gonzalez J, Guestrin C, Kyrola A, 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]

[36] Thomas K, Grier C, Ma J, Paxson V, Song D. Design and evaluation of a real-time URL spam filtering service. In: Proc. of the 2011 IEEE Symp. on Security and Privacy (SP). Berkeley: IEEE, 2011. 447−462. [doi: 10.1109/SP.2011.25]

[37] Zhang Z, Barbary K, Nothaft FA, Sparks E, Zahn O, Franklin MJ, Patterson DA, Perlmutter S. Scientific computing meets big data technology: An astronomy use case. In: Proc. of the 2015 IEEE Int'l Conf. on Big Data. Santa Clara: IEEE, 2015. 918−927. [doi: 10.1109/BigData.2015.7363840]

[38] Nothaft FA, Massie M, Danford T, Zhang Z, Laserson U, Yeksigian C, Kottalam J, Ahuja A, Hammerbacher J, Linderman M. Rethinking data-intensive science using scalable analytics systems. In: Proc. of the 2015 ACM SIGMOD Int'l Conf. on Management of Data. Victoria: ACM Press, 2015. 631−646. [doi: 10.1145/2723372.2742787]

[39] Veiga J, Expósito RR, Pardo XC, Taboada GL, Tourifio J. Performance evaluation of big data frameworks for large-scale data analytics. In: Proc. of the 2016 IEEE Int'l Conf. on Big Data. Washington: IEEE, 2016. 424−431. [doi: 10.1109/BigData. 2016.7840633]

[40] Lee H, Kang M, Youn SB, Lee JG, Kwon Y. An experimental comparison of iterative MapReduce frameworks. In: Proc. of the 25th ACM Int'l on Conf. on Information and Knowledge Management. Indiana: ACM Press, 2016. 2089−2094. [doi: 10.1145/2983323.2983647]

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

[42] Kang M, Lee JG. An experimental analysis of limitations of MapReduce for iterative algorithms on Spark. Cluster Computing, 2017,20(4):3593−3604. [doi: 10.1007/s10586-017-1167-y]

[43] Mavridis I, Karatza H. Performance evaluation of cloud-based log file analysis with apache hadoop and apache Spark. Journal of Systems and Software, 2017,125:133−151. [doi: 10.1016/j.jss.2016.11.037]

[44] 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 with English abstract). http://www.jos.org.cn/1000-9825/5169.htm

[45] Shi J, Qiu Y, Minhas UF, Jiao L, Wang C, Reinwald B, Özcan F. Clash of the titans: Mapreduce vs. Spark for large scale data analytics. Proc. of the VLDB Endowment, 2015,8(13):2110−2121. [doi: 10.14778/2831360.2831365]

[46] Page L. The PageRank citation ranking: Bringing order to the Web. Stanford Digital Libraries Working Paper, 1998,9(1):1−17.

[47] Davidson A, Or A. Optimizing shuffle performance in Spark. Technical Report, University of California, Berkeley-Department of Electrical Engineering and Computer Sciences, 2013.

[48] 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 Int'l Journal on Very Large Data Bases, 2012,21(5):589−609. [doi: 10.1007/s00778-012-0279-5]

附中文参考文献:

[44] 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 [doi: 10.13328/j.cnki.jos.005169]

吴信东(1963－),男,安徽枞阳人,博士,教授,博士生导师,主要研究领域为数据挖掘,大数据分析,知识库系统,万维网信息探索.

嵇圣硙(1994－),男,硕士生,主要研究领域为数据挖掘,分布式数据库.