面向负载均衡的动态均衡分区策略

杨 迪 赵家伟 王 鹏* 赵建平
(长春理工大学 吉林 长春 130000)

摘 要 针对 MapReduce 计算框架处理倾斜数据集时造成 Reduce 端出现负载不均衡现象,提出一种动态均衡分区策略。在 mapper 阶段提出基于分治法的数据切分原则处理任务传入的数据组;结合最佳适应算法思想设计动态分配原则逐步将切分后的数据块均衡分配到预分区链表中;根据分区索引分配到各 Reduce 节点上实现负载均衡。实验结果显示,动态均衡分区策略与两个基准模型相比任务执行时长平均降低了 7.7%,表明动态均衡分区策略更好地解决了数据倾斜问题,降低了任务执行时间,验证了模型的有效性。

关键词 MapReduce 负载均衡 数据倾斜 数据分区

中图分类号 TP3 **文献标志码** A DOI:10.3969/j.issn.1000-386x.2024.08.007

DYNAMIC BALANCING PARTITION STRATEGY FOR LOAD BALANCING

Yang Di Zhao Jiawei Wang Peng* Zhao Jianping
(*Changchun University of Science and Technology*, *Changchun 130000*, *Jilin*, *China*)

Abstract In view of the unbalanced load on the reduce side caused by MapReduce computing framework processing inclined data sets, this paper proposes a dynamic balanced partition strategy. In the mapper stage, the data segmentation principle based on divide and conquer method was proposed to process the incoming data groups. The dynamic allocation principle was designed combined with the idea of the best adaptive algorithm to gradually allocate the segmented data blocks to the server. According to the partition index, it was allocated to each reduce node to achieve load balancing. The experimental results show that, compared with the two benchmark models, the average task execution time of the dynamic balanced partition strategy is reduced by 7.7%, which indicates that the dynamic balanced partition strategy can better solve the problem of data skew, reduce the task execution time, and verify the effectiveness of the model.

Keywords MapReduce Load balancing Data skew Data partition

0 引 言

MapReduce 是一种应用广泛的分布式计算框架,具有容错计算能力,可以并行处理 TB 级别的海量数据[1]。目前该框架正被广泛应用于地质勘探、电子商务、医疗影像、股票预测、气象分析等领域[2]。负载均衡问题在 MapRecuce 框架中一直是一个急需解决的关键性问题,Shuffle 分区结果会影响各个 Reduce 端接收处理数据的均衡性进而影响整个 MapReduce 的运行时间[3]。围绕如何解决负载均衡问题,国内外研究人员提出了多种研究方法与策略,可以归纳为数据采样法、针对连接操作方法、多策略融合方法、多阶段分区法。在数据采样方法中,Chen 等[3]通过采集整个 MapReduce 运行过程中的各个 Map 节点中间数据分布情况,并结合 Reducer 端的硬件资源去制定整个数据均衡分配策略。Tang 等[4]对待处理数据集合进行采样,预测 cluster 的大小,根据预测结果进行数据分割并填充到 Spark 的 Bucket 上进行分配。Elaheh Gavagsaz 等[5]通过对比中间数据集合进行抽样,此方法更加精确地观察到键值对的近似分布,然后应用可伸缩性的抽样算法对键值的整体分布进行评估。数据采样法通过判断待处理数据集的分布情况制定相应策略,可在一定程度上解决数据倾斜问题,但数据采样存在不确

收稿日期:2021 - 03 - 02。中央引导地方科技发展资金吉林省基础研究专项(202002038JC)。**杨迪**,讲师,主研领域:深度学习,大数据智能交通。**赵家伟**,硕士生。**王鹏**,教授。**赵建平**,教授。

定性,且采样过多会造成额外的时间开销影响任务运行的总体时间。数据库连接操作引发的数据倾斜问题是使 MapReduce 出现负载不均衡的常见问题,针对连接操作,赵宇兰等[6]利用复制、广播方法改进 Range-Partition 算法将数据发送到每个 Reduce 节点上,并通过一轮 MapReduce 任务完成所有连接操作。Wang 等[7]提出了一种基于聚类代价分区(CCP)的数据倾斜方法优化 MapReduce 中的并行链接。周娅等[8]针对传统的多表连接查询算法不能有效地解决数据倾斜导致的性能瓶颈问题,统计倾斜与轮询分区策略进而均衡地将数据分发到 Hadoop 集群的各个计算节点上。尽管上述方法能够解决由连接操作引起的数据倾斜问题,但不具备普遍适用性。

多策略融合方法是一种通过判断不同类型的数据倾斜问题采用不同策略的方法。Belussi 等[9]提出了一种基于一次计数的启发式方法,通过检验输入的空间数据集合的倾斜程度去决定使用哪个分区技术,以期提高后续的操作性能。梁俊杰等[10]对倾斜数据集合利用分区算法和广播算法实现数据重分布,从而消除数据倾斜的影响,而对非倾斜的数据集合采用自带的 Hash 算法实现数据分配。Berlińska 等[11]在文章中针对四种不同类型的数据倾斜提出了四种不同的算法去处理数据倾斜,并评估它们各自在不同程度的数据倾斜、不同系统参数下的性能。多种策略融合方法可处理多种类型的数据倾斜,但在判别倾斜类型阶段比较耗时最终影响 MapReduce 执行效率。

多阶段分区方法采用多轮次分区策略解决了一次分区造成的数据倾斜问题。高宇飞等[12]在 Map 阶段采用虚拟分区方式使得键值对分散存储,为后续重分区提供更好的分区组合,在 Reduce 阶段利用连续虚拟分区平衡重组方法将收集到的虚拟分区重新划分成与 Reduce 任务数相同的分区。王卓等[13]在 Mapper 运行过程时统计各个细粒度分区的数据量,然后由 JobTracker 按照全局分区信息筛选出部分未分配的细粒度分区,并用代价评估模型将剩余的分区分配到每个 Reducer 节点上。张元鸣等[14]将每个 Mapper 节点要处理的数据块细分再以迭代的方式循环处理,根据已迭代的微分区分配结果决定当前迭代轮次的微分区分配方案,并不断调整历次迭代产生的数据倾斜,逐步实现数据均衡分区。Zhou 等[15]通过跟踪监控全局分区信息并动态修改原分区函数,改变剩余分区索引使负载量较少 Reduce 端接收更多的数据。卞琛等[16]将 Spark 中间数据划分为原生区和扩展区,扩展区数据在特定时机下多次分配实现负载均衡。多阶段分区方法既在处理数据倾斜问题上节约了时间,又具有普遍适用性。

本文提出了一种基于多阶段分区思想的动态均衡分区策略。该策略通过结合分治法设计数据切分原则降低各 Reduce 节点所处理最大元组数据量,并采用最佳适应算法思想设计动态分配原则将数据均衡分配到各 Reduce 节点上。本文将提出的数据切分原则与动态分配原则应用于 MapReduce 中 Shuffle 阶段的数据分配,通过与两个基准模型进行实验对比考察该策略的有效性。

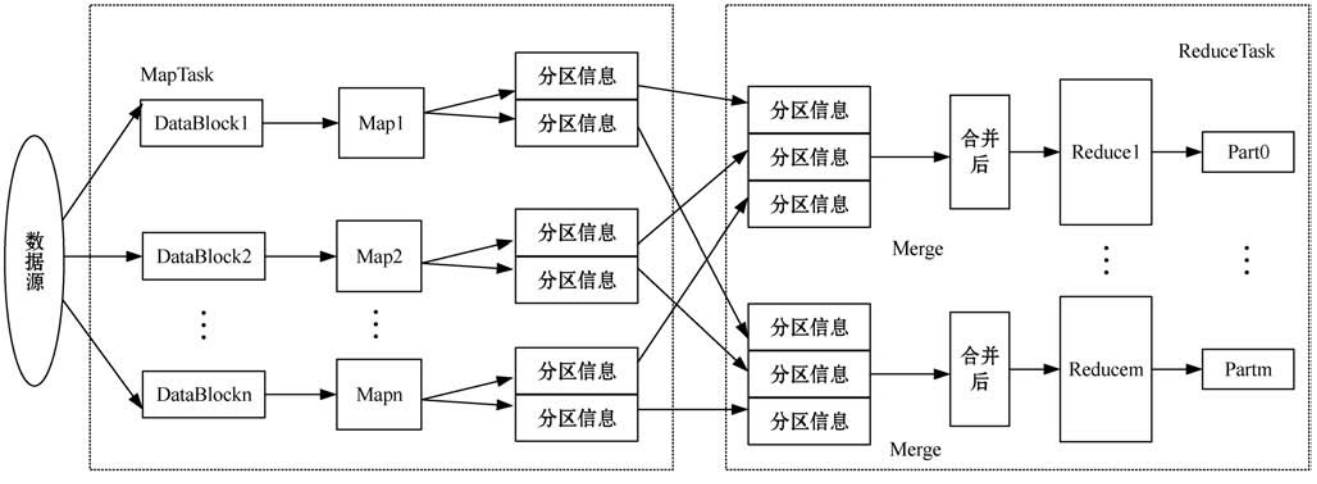

图 1　MapReduce 默认处理流程

1　MapReduce 分区策略

本节将介绍 MapReduce 框架的数据分区策略,其中 Shuffle 阶段作为 MapReduce 框架核心在实现数据区上起到了关键作用,Shuffle 在制定分区策略时分为 Map 阶段和 Reduce 阶段。

在 Map 阶段:Mapper 首先要进行数据处理,根据 Block 块的大小将传入的数据进行切分,并按照切分后产生 Block 的数量启动同一数量的 Mapper,Mapper 通

过 Map( )函数处理 Block 块,并输出成一个 key/reduce 的形式。接下来进行分区操作,MapReduce 采用默认的 Hash 分区,利用 Has(key) mod(Reduce 数目) 的方法进行计算,并为键值对分配分区信息将其序列化写入内存缓冲区内,缓冲区通过批量收集 Map 结果从而减少磁盘 IO 的影响。这个内存缓冲区在容量上存在一定限制,Hadoop 默认容量大小为 100 MB,当 MapTask 产生大量的输出结果时极有可能造成内存溢出,所以系统默认缓冲区的内存数据阈值为 80 MB,达到上限时会将缓冲区中的数据临时溢写入磁盘中,然后继续复用此缓冲区。在执行溢写的过程中,MapTask 的输出依旧会向剩下的 20 MB 内中存储,溢写线程会对 80 MB 空间中的序列化 Key 进行排序,方便接下来的操作。当 MapTask 完成后,内存缓冲区的数据也就全部溢写到磁盘中形成一个溢写文件。最终系统将溢写文件按照相同分区进行合并,再建立索引方便 Reducer 端对文件进行拉取。

在 Reduce 阶段:由于在 Map 阶段已经对溢写文件合并、建立索引,因此各个 Reducer 确定了其要去处理哪些分区文件。一个 Reducer 默认只能处理一个分区文件,Reducer 初始化完成后首先要对文件进行 Copy 操作,启动大量数据 copy 线程,通过 HTTP 方式请求从 TaskTracker 获取 Maptask 的输出文件。接下来进入 Reduce 阶段的 Merge 操作,此时的合并操作比 Map 端的合并操作要更加灵活,由于 Shuffle 阶段 Reducer 不运行,所以绝大部分内存都交给 Shuffle 使用。在不断进行 Merge 操作后,最后生成一个最终文件并执行 Reduce 操作。

2 动态均衡分区实现

本节首先对动态均衡分区策略的总体流程进行描述,然后分别对其中动态均衡分区原则、求解动态均衡分区代价评估模型算法进行详细介绍。

2.1 动态均衡分区策略

动态均衡分区策略与 Hadoop 默认处理数据的 Shuffle 一致,同样将分区策略分为 Map 阶段和 Reduce 阶段,整个分区策略的流程如图 2 所示,主要针对原分区策略的 Map 阶段改进,增加了基于分治法的数据切分原则和基于最佳适应算法的分区分配原则。

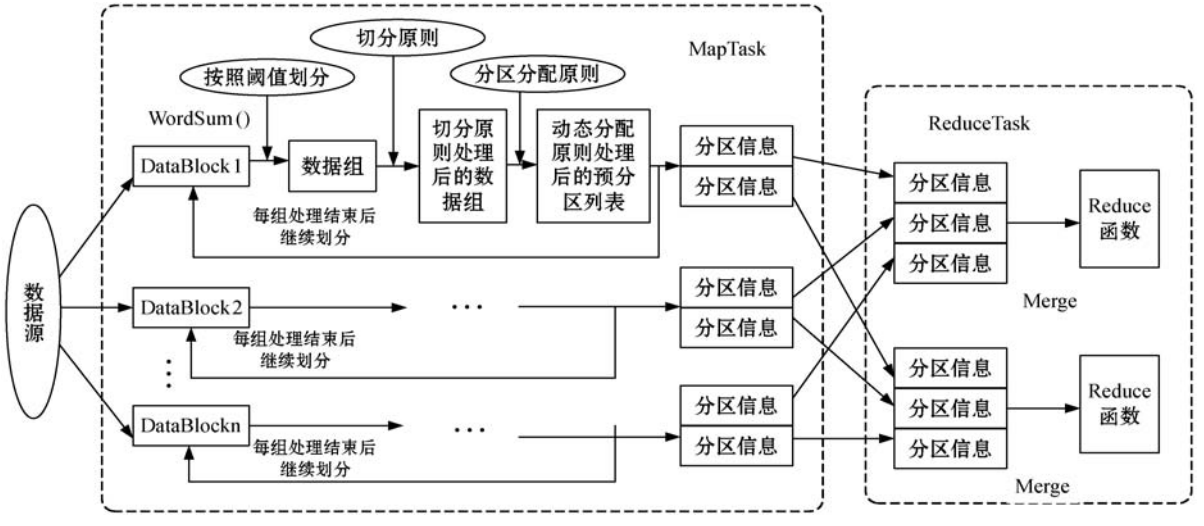

图 2 动态均衡分区的 MapReduce 处理流程

在 Map 阶段:将数据源切片生成若干个 DataBlock 后,首先统计传入 Map 中各元组的数量并存入 ListSum 链表中,为接下来的数据切分、数据分配做准备。待链表的数据达到规定的阈值上限时进行数据切分原则判断。在进行切分原则判断之前,设 Reducer 节点数为 N,新建 N 个预分区链表作为对应存储各分区信息的存储器,为这些链表建立索引使其与各分区保持一一对应关系。若 ListSum 中存在元素满足切分原则对满足的元素进行切分,再依次存入所建的预分区链表。ListSum 中的元素经过合理切分后,利用最佳适应算法思想,使用自建的分区分配原则去处理链表,使链表中的元素平均分配到预分区链表中,同时为预分区链表建立索引方便 Reduce 端拉取分区数据。当链表中的全部元素被分配完毕后清空 ListSum 链表,重新进行上述操作直到 DataBlock 中的数据全部被处理完成。

2.2 动态均衡分区原则

2.2.1 基于分治法的数据切分原则

一般不要超过两行。数据切分原则,利用 WordSum( )函数处理各数据组,并将统计得到的结果存入元组统计链表 ListSum 中等待进一步分配,当 ListSum 存储的元组数据量达到 *maxSize* 阈值时,先进行切分原则判断。若在 ListSum 链表存在某一个元素的数据量

远大于其余元素,那么在进行数据分配时,无论怎样分配数据也无法阻止数据倾斜的产生,因此本文引入了基于分治法思想的切分数据原则。如图 3 所示,将数据量较大的元组分解成 N 个数据量较少的元组,若这 N 个数据量较小的元组依旧不满足条件,则继续切分直到满足条件为止。切分原则首先会对链表中元组的数据量大小进行判断,在已经按数据量排序好的 ListSum 链表中取出第一个元素进行判断,若其数据量大于 *maxSize*/Reduce 节点个数,就代表进行预分区时一定存在其中一个预分区的元组数据量大于其余分区数据量,此时将会对其进行切分,按照 Reduce 节点的个数将其均等分配到各个预分区链表中。接下来继续从链表中拿出下一个元素判断,直到元素不满足条件后结束切分原则的判断进行第一轮数据分配。

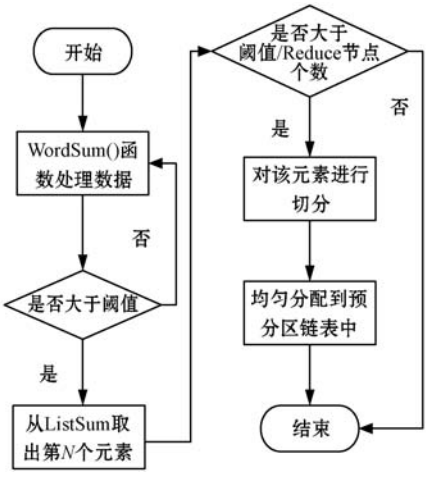

图 3　数据切分原则处理流程

2.2.2　基于最佳适应算法的分区分配原则

当切分原则判断结束后,引入基于最佳适应算法的分区分配原则将数据均匀分配到预分区链表中。系统生成任务后会向 Mapper 端传入数据,每个 Mapper 所处理的数据量大小由 Hadoop 的 BlockSize 来确定,默认的 BlockSize 大小为 128 MB,为了保证这 128 MB 的数据在分配时均匀分配到每一个分区中,我们引入最佳适应算法思想建立分区分配原则。最佳适应算法能把既满足分配要求又存在当前最小空闲空间的分区作为下一个处理数据的分区,避免了出现某个分区载入数据过多的现象。为了加速最佳适应算法寻找,新建一个空闲分区链 KP,该链上放置预分区的全部信息,其中包括预分区将处理的元组数量和数据量等。存储元组数据信息链表会占用大量的存储空间,进行取出、取入操作时会浪费大量内存,因此本文制定一个阈值 *maxSize* 作为每轮处理分配数据量的上限。通过调控 *maxSize* 的大小做到既实现 Reduce 节点的负载均衡,又在最大限度上节约时间和空间。

动态分配原则要对比每一个存储数据预分区的信息,在每次分配时选择剩余存储空间最大的预分区进行分配。因此按照 Reduce 节点数目 N 建立 N 个链表和微分区索引作为存储 N 个分区的数据信息和分区分配信息。在 ListSum 中的数据全部分配完成后,将此链表的数据清空,重新进行下一组分配。在对 ListSum 进行分配时,首先将 ListSum 中的元组按照元组数目大小进行排序,每次分配时优先将数据量大的元组进行分配,使数据量大的分区尽早进行 Shuffle,让数据量较少的分区稍后进行传输,减少了最后一个 Mapper 完成传输后的等待完成时间。第一轮分配时将元组数据量排名前 N 的元组依次分配到预分区链表中,在第二轮时由于各个存储预分区数据信息的链表已经发生了变化,因此要考虑这些链表的负载量,按照最佳适应算法思想进行后几轮的分配,最终实现预分区链表数据量的均衡。

目前其他策略如迭代式分区策略等在后几轮依旧采取与 Reducer 节点个数相同的元组数分配,在某些极端情况下不但起不到负载均衡作用反而会加剧数据倾斜。若第一轮分配某一元组的数据量远大于其余元组,在后几轮分配数据时依旧对其进行分配数据这样就会加剧数据倾斜的产生。因此本文采取了一种新的数据分配模式,如图 4 所示,若第一轮预分区链表分配的最大元组数据量 *MaxFst* 远大于第二大元组数据量 *MaxSec*,就代表下一轮若再对数据量为 *MaxFst* 的预分区链表分配元组将会增大各预分区链表中的数据差距。若发生上述情况在下一轮分配时将不会对此预分区链表分配元组,直到 *MaxFst* 与 *MaxSec* 相差不大为止。每轮分配时,将从链表中拿出的 N 个元组数据量最大的元组依次放入元组数据量最小的预分区链表中,直到 ListSum 链表中的全部数据被分配完成。

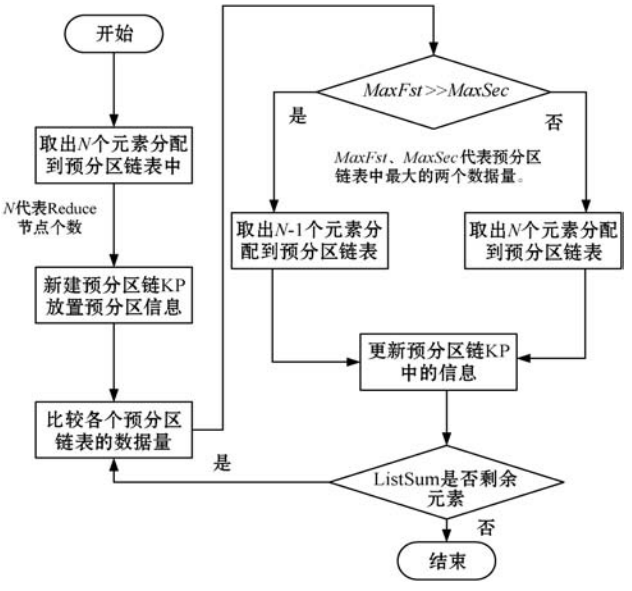

图 4　动态分配原则流程

2.3 动态均衡分区的算法设计

实现动态均衡分区策略需要实现两个原则,第一个原则是切分原则,其目的是为了解决要分配的元组存在数据量过大现象,在进行切分判断后,会将该元组均匀分配到 Reducer 端进行处理。第二个原则是分区分配原则,通过此原则将数据均匀地分配到 Reducer,实现负载均衡。

算法 1 实现的是数据切分算法,利用改进的分治法思想融合实现,属于低时间复杂度的算法。它的时间复杂度主要与链表中元素个数相关,若链表中元素的个数很多说明单个元素的 *count* 值较低,算法花费的代价相对较低。因此在进行数据切分时要先将链表元素按照 *count* 值进行排序,这样就不必遍历整个链表去寻找 *count* 值较大的元素,只需在排序后的链表中拿出前几个元素进行判断即可。

切分原则算法如算法 1 所示。

算法 1 切分原则算法

输入:未经过切分原则判断的 WordSum 类型的 ListSum 链表;Reducer 节点个数 *N*;规定 List 链表的阈值 *maxSize*。

输出:经过切分原则的链表 ListSumQ;经过切分原则后的预分配链表 a、b、c。

```
Sort(ListSum)
        //将 ListSum 中的元素按照元素的 Count 属性排序
FOR i in ListSum.length()
  IF i.count > maxSize/(N * 3)
    a.append(i/N)
      //如果大于规定值将此元素拆分到各个预分区链表
    b.append(i/N)
    …        //Reducer 节点有多少预分区链表就有几个
  ENDIF
ENDFOR
ListSumQ、a、b…        //拿到经过切分原则后的各链表
```

算法 2 是动态分配算法,该算法基于最佳适应算法 B-Fit 选择空闲空间较大、所存数据量较小的预分区链表作为优先分配对象,从经过切分原则处理后的 ListSumQ 取出元组进行分配。为了保证各预分区链表充分满足数据量均等思想,在每轮分配时进行判断确定此轮分配的元组个数。

算法 2 动态分配算法

输入:经过切分原则的链表 ListSumQ;经过切分原则后的预分配链表 a、b、…

输出:a、b、c。

```
INT Index                                    //定义分配轮次
IF Index == Fist                             //第一轮分配
  FOR i in N
        //将 ListSumQ 的前 N 个元素取出放入预分区链表
    X.insert(ListSumQ(i))                    //X 代表 a、b、c…
  ENDFOR
ELSE
  FOR i in ListSumQ.Length()
    MaxValue = maxSort(a.sum(),b.sum()…)
                //找出预分区链表中全部数据量最大值
    SecValue = maxSort(b.sum(),c.sum()…)
                //找出预分区链表中全部数据量第二大值
    IF ListMaxValue > ListMaxValue * 1.5
      Int DivideNum = N - 1
    ELSE
      INT DivideNum = N
    FOR i in DivideNum
      MinValueList = minSortList(a,b,c)
                //找出预分区链表中数据量最小的链表
      MinValueList.Insert(ListSumQ(i))    //从 ListSumQ
        //链表中取出最大值插入数据量最小的链表中
      Exclude(MinValueList)
                //下一轮比较将此链表排除在外
    ENDFOR
ENDFOR
```

3 实 验

本节通过两部分实验去验证动态均衡分区策略的性能,第一部分实验重点通过调节上文所规定的阈值参数大小对整个策略的影响,第二部分使用标准的 Zipf 数据集,将本文的动态分区策略与 Hadoop 原系统自带的 Hash 分区策略和增量式数据分区策略[12]进行对比实验。

3.1 实验设置

3.1.1 实验环境

实验中使用本地搭建的 Hadoop 虚拟集群去进行测试,使整个 Hadoop 集群由 5 个节点构成,其中包括 1 个 Master 节点和 4 个 Slave 节点,其中 Master 节点作为 NameNode 节点负责集群的任务调度和管理,不进行计算,Slave 节点作为 DataNode 节点进行数据存储与计算,每个节点的配置相同:硬盘 80 GB,内存 4 GB,处理器为酷睿双核 CPU,操作系统为 centOS7,安装 Hadoop 的版本为 3.1.3,采用 JDK1.8 进行编译。

3.1.2 实验数据

本文所使用的实验数据主要包括两类分别是合成数据集和真实数据集,其中合成数据集为 Zipf 数据集合,一共包括 10 个数据子集,各个子集中的存放数据的取值为 0 ~ 1 之间的小数,每个子数据集合中满足指数为 x 的标准 Zipf 分布,x 的增量为 0.1,取值范围从

0.6~1.5 之间的小数,当 x 的值越大,数据的分布越倾斜。

3.2　实验结果

3.2.1　阈值对动态均衡分区的影响

本文先通过实验评判文中规定每组处理数据阈值的大小对动态均衡分区的影响。实验使用倾斜度为 0.6,数据量大小为 400 MB 的 Zipf 数据集,其中约有 2 000 万个元组数目,Key 的取值范围是[0,1]。本次实验运行的是经典的 WordCount 实例。

阈值大小是指动态均衡分区策略中每组处理数据量大小,根据测试如果将阈值设置得很小,每组待处理数据的数据量就会很小,自建的动态分配分区策略就起不到相对的作用,若将阈值设置的特别大,每组待处理数据的数据量就会很大,作为存储每组数据的链表 ListSum 就会占用大量的内存空间,同时在进行动态分配原则时也会消耗大量的运行时间。因此选择大小适合的阈值可以更好地实现数据均衡、降低运行时间。

图 5 展示了设置不同的阈值大小对各个 Reduce 节点最长运行时间的影响。实验结果显示随着阈值的不断增加,Reduce 节点运行的最长时间不断降低,也就越接近负载均衡实现。而当设置的阈值达到 4 MB 时,Reduce 节点的最长运行时间趋于稳定,各节点数据趋于均衡状态,持续增加阈值对节点运行时间影响较小。

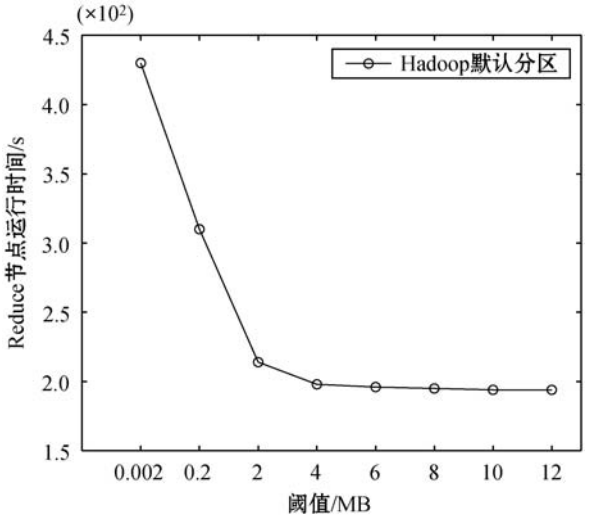

图 5　Reduce 节点最长运行时间对比

如图 6 所示展示了动态分配分区策略在设置不同阈值大小对作业的整体执行时间的影响。由于本文的动态均衡分区策略会随着阈值的增加进而增加 Map 的运行时间,减少 Reduce 的运行时间。当阈值达到一定程度节点就会无限趋近负载均衡状态,Reduce 的运行时间将不会随着阈值增加继续降低,而 Map 的运行时间还会保持增加趋势。实验表明随着阈值大小的增加,作业的整体运行时间先呈现递减状态再呈现递增状态。

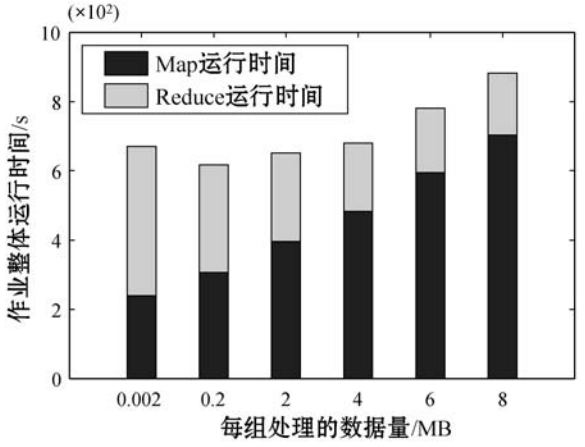

图 6　作业整体运行时间对比

3.2.2　动态均衡分区的性能比较

本节在可调节倾斜度的数据集合 Zipf 上运行 WordCount 实例对 Hadoop 默认的分区策略、增量式分区策略和本文的动态均衡分区策略进行比较分析。其中 Hadoop 采用 Hash 分区加随机分配原则,增量式分区策略定义分配轮数和放大系数均为 10,本文提出的动态均衡分区策略将阈值参数 MasSize 设为 4 MB。由于本文主要解决负载均衡问题,评判的性能指标是 Reduce 节点最长运行时间和作业整体运行时间。

如图 7 所示,本节使用大小为 400 MB 的 5 个倾斜指数不同的 Zipf 数据集合进行对比实验。实验结果表明随着倾斜指数的增加,Reduce 节点的最长运行时间也相应增加,而 Hadoop 默认 Hash 分区策略明显比增量式分区策略和动态均衡分区策略耗时更长。针对 Reduce 节点最长运行时间,处理数据倾斜严重的数据集合动态均衡分区策略明显优于增量式分区策略,动态分配分区策略的性能比默认分区策略性能平均提高了 42.7%,比增量式分区策略的性能平均提高了 3.3%。

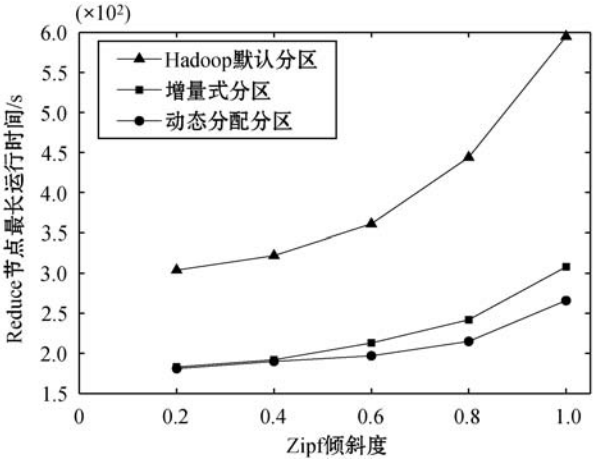

图 7　Reduce 节点最长运行时间对比

图 8 给出了上述实验中所使用的不同数据倾斜指数的 Zipf 数据集合在对比三个策略的任务整体运行时间、任务在 Mapper 端的运行时间、任务在 Reduce 端的运行时间。实验结果显示在倾斜指数较低的时候三种策略的任务整体运行时间相差不大,随着倾斜指数的增加,默认分区策略在 Reduce 端运行时间逐渐增加,任务整体运行时间也随之增加,与动态分配策略,增量式分区策略的差距也愈发明显。动态均衡分区策略的任务总体执行性能比默认分区平均提高了 13.26%,比增量式分区策略平均提高了 2.14%。

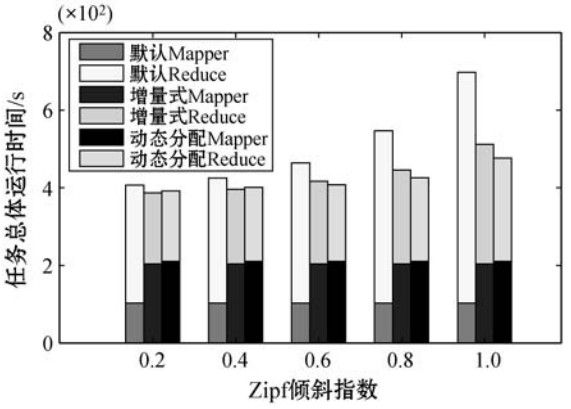

图 8 任务总体运行时间对比

4 结 语

本文提出了一种动态均衡分区策略用于解决 MapReduce 的数据倾斜问题,该策略在 Map 阶段新增数据切分原则与动态分配原则从而实现 Reduce 阶段的负载均衡。最后在不同倾斜指数的 Zipf 数据集上与两个基准模型进行对比实验,结果表明与默认的分区策略相比动态均衡分区策略的负载均衡显著提升。本文主要考虑数据量的均衡分配,未考虑节点计算量出现差异的情况,下一步将研究各 Reduce 节点在不同计算量的情况下如何更合理地分配数据。

参 考 文 献

[1] 程学旗,靳小龙,王元卓,等. 大数据系统和分析技术综述[J]. 软件学报,2014,25(9):1889 - 1908.

[2] Irandoost M A, Rahmani A M, Setayeshi S. A novel algorithm for handling reducer side data skew in MapReduce based on a learning automata game[J]. Information Sciences,2019,501:662 - 679.

[3] Chen Q, Yao J Y, Xiao Z. LIBRA: Lightweight data skew mitigation in MapReduce[J]. IEEE Transactions on Parallel & Distributed Systems,2015,26(9):2520 - 2533.

[4] Tang Z, Zhang X S, Li K L, et al. An intermediate data placement algorithm for load balancing in Spark computing environment[J]. Future Generation Computer Systems,2018,78:287 - 301.

[5] Gavagsaz E, Rezaee A, Javadi H. Load balancing in reducers for skewed data in MapReduce systems by using scalable simple random sampling[J]. The Journal of Supercomputing,2018,74(7):3415 - 3440.

[6] 赵宇兰. 基于 MapReduce 的两表数据倾斜连接的优化算法[J]. 吉林大学学报(理学版),2016,54(6):1383 - 1387.

[7] Wang Y, Zhong Y, Ma Q S, et al. Handling data skew in joins based on cluster cost partitioning for MapReduce[J]. Multiagent and Grid Systems,2018,14(1):103 - 123.

[8] 周娅,魏夏飞,熊晗,等. CSPRJ:基于数据倾斜的 MapReduce 连接查询算法[J]. 小型微型计算机系统,2018,39(2):367 - 371.

[9] Belussi A, Migliorini S, Eldawy A. Skewness-based partitioning in SpatialHadoop[J]. ISPRS International Journal of Geo-Information,2020,9(4):201.

[10] 梁俊杰,何利民. 基于 MapReduce 的数据倾斜连接算法[J]. 计算机科学,2016,43(9):27 - 31.

[11] Berlinska J, Drozdowski M. Comparing load-balancing algorithms for MapReduce under Zipfian data skews[J]. Parallel Computing,2018,72:14 - 28.

[12] 高宇飞,曹仰杰,陶永才,等. MapReduce 计算模型下基于虚拟分区的数据倾斜处理方法[J]. 小型微型计算机系统,2015,36(8):1706 - 1710.

[13] 王卓,陈群,李战怀,等. 基于增量式分区策略的 MapReduce 数据均衡方法[J]. 计算机学报,2016,39(1):19 - 35.

[14] 张元鸣,蒋建波,陆佳炜,等. 面向 MapReduce 的迭代式数据均衡分区策略[J]. 计算机学报,2019,42(8):1873 - 1885.

[15] Zhou H P, Liu G Z, Gui H X. A strategy to load balancing for non-connectivity MapReduce job[J]. IOP Conference Series Materials Science and Engineering, 2017, 231(1): 12038.

[16] 卞琛,于炯,修位蓉,等. 基于迭代填充的内存计算框架分区映射算法[J]. 计算机应用,2017,37(3):647 - 653.