基于优先填补策略的Spark数据均衡分区方法

何玉林[1,2],吴东彤[1,2],Philippe Fournier-Viger[2],黄哲学[1,2]
(1. 人工智能与数字经济广东省实验室(深圳),广东深圳 518107;2. 深圳大学计算机与软件学院,广东深圳 518060)

摘 要: Spark作为基于内存计算的分布式大数据处理框架,运行速度快且通用性强. 在任务计算过程中,Spark的默认分区器HashPartitioner在处理倾斜数据时,容易产生各个分区数据量不平衡的情况,导致资源利用率低且运行效率差. 现存的Spark均衡分区改进方法,例如多阶段分区、迁移分区和采样分区等,大多存在尺度把控难、通信开销成本高、对采样过度依赖等缺陷. 为改善上述问题,本文提出了一种基于优先填补策略的分区方法,同时考虑了样本数据和非样本数据的分配,以便实现对全部数据的均衡分区. 该方法在对数据采样并根据样本信息估算出每个键的权值后,将键按照权值大小降序排列,依次将键在满足分区容忍度的条件下分配到前面的分区中,为未被采样的键预留后面的分区空间,以获得针对样本数据的分区方案. Spark根据分区方案对样本中出现的键对应的数据进行分区,没有出现的键对应的数据则直接映射到可分配的最后一个分区中. 实验结果表明,新分区方法能够有效实现Spark数据的均衡分区,在美国运输统计局发布的真实航空数据集上,基于该方法设计的优先填补分区器的总运行时间比HashPartitioner平均缩短了15.3%,比现有的均衡数据分区器和哈希键值重分配分区器分别平均缩短了38.7%和30.2%.

关键词: 均衡分区;优先填补策略;数据倾斜;Spark算子;大数据

基金项目: 深圳市科技重大专项项目(No.202302D074);广东省自然科学基金面上项目(No.2023A1515011667);深圳市基础研究面上项目(No.JCYJ20210324093609026);广东省基础与应用基础研究基金粤深联合基金重点项目(No.2023B1515120020)

中图分类号: TP311 **文献标识码:** A **文章编号:** 0372-2112(2024)10-3322-14

电子学报URL:http://www.ejournal.org.cn **DOI:**10.12263/DZXB.20240094

First Filling Strategy-Based Partitioning Method to Balance Data in Spark

HE Yu-lin[1,2], WU Dong-tong[1,2], Philippe Fournier-Viger[2], HUANG Zhe-xue[1,2]
(1. *Guangdong Laboratory of Artificial Intelligence and Digital Economy (Shenzhen), Shenzhen, Guangdong* 518107, *China*;
2. *College of Computer Science and Software Engineering, Shenzhen University, Shenzhen, Guangdong* 518060, *China*)

Abstract: Spark is a distributed big data processing framework based on in-memory computing, which has the advantages of fast running speed and strong versatility. When conducting the computation task, Spark's default partitioner HashPartitioner is easy to generate data skewing among partitions. It results in low resource utilization and poor operating efficiency. Most of the existing Spark balanced partitioning methods, such as multi-stage partitioning, migration partitioning, and sampling partitioning, have defects of scale control difficulty, high communication overhead, and excessive sampling dependence. In order to solve the above-mentioned problems, we propose a partitioning method based on first filling strategy, which considers the allocations of sample data and non-sample data at the same time, so as to achieve a balanced data partitioning. After sampling the data and estimating the weight of each key according to the sample information, the keys are sorted in descending order according to the weights. The keys are in turn assigned to the previous partitions if their additions can satisfy the partition tolerance, and the space of the last partition is reserved for the keys that are not sampled, so as to obtain the partitioning plan for the sample data. Spark partitions the data corresponding to the keys that appear in the sample according to the partitioning plan, and the data of other keys that do not appear is directly allocated to the last data partition available. The experimental results show that the new method can effectively achieve balanced partitioning for Spark data. On the real datasets from Bureau of Transportation Statistics, compared with HashPartitioner, the total running time of first filling partitioner (FFP), designed based on the proposed method, is shortened by 15.3% on average. In addition, FFP's

收稿日期:2024-01-22;修回日期:2024-07-17;责任编辑:覃怀银

total running time is on average 38.7% shorter than balanced Spark data partitioner and 30.2% shorter than hash based key reassigning partitioner.

Key words: balanced partitioning; first fill strategy; data skew; Spark operator; big data

Foundation Item(s): Science and Technology Major Project of Shenzhen (No.202302D074); Natural Science Foundation of Guangdong Province (No.2023A1515011667); Basic Research Foundation of Shenzhen (No.JCYJ20210324093609026); Guangdong Basic and Applied Basic Research Foundation (No.2023B1515120020)

1 引言

社会的快速发展,科技的日新月异,使得产生的数据与日俱增,大数据计算概念随之被提出[1]. 在大数据背景下,分布式计算应运而生,主流的分布式计算平台有 Hadoop[2]、Spark[3]和 Flink[4]等. Hadoop 主要由分布式文件系统(Hadoop Distributed File System,HDFS)[5]和并行计算框架MapReduce[6]组成,用于对大量数据进行分布式处理. Spark 是一个快速通用的计算引擎. 与 Hadoop 和 Flink 不同,它是基于内存计算的,有效避免了频繁访问磁盘的I/O开销,运行速度更快,且有着更丰富的算子库支持,是当前大数据领域最热门和通用的分布式计算框架.

Spark 将任务运行划分为多个阶段(stage),每个 Stage 由多个作业(task)构成,每个 Task 对应一个分区,同一个 Stage 的 Task 可以并行执行. 因此,分区数据量会影响其所在 Task 的运行时间,进而影响其所在 Stage 的运行时间. 如果各个分区的数据量存在严重的不平衡,那么在相同条件下,分区数据量多的 Task 的运行时间就会远远大于其他 Task,会导致其他 Task 对应的资源闲置,整个 Stage 的运行时间变长. 因此,我们需要对 Spark 任务中的数据均衡分区,提高运行效率[7].

Spark 是在 Hadoop MapReduce 计算框架的设计思想上发展而成的,故两者有着相似的洗牌(shuffle)过程以及分区方法,也就有着相似的数据倾斜问题. 面向 MapReduce 的数据倾斜问题,前人提出了很多均衡分区的方法[8~19],但面向 Spark 的均衡分区方法却相对较少. 虽然理论上具有相似性,但实际中两种方法之间难以通用. Spark 的计算以算子为单位,Map 类算子底层是已经包装好的 scala 文件,无法像 MapReduce 一样对映射的中间过程做额外改动,故面向 MapReduce 的数据均衡分区方法并不适用于 Spark 环境. 由此,提出新的面向 Spark 的数据均衡分区方法,具有重要的意义.

文献[20]提出了一种中间倾斜数据块的拆分合并算法,将簇迭代分配到各个桶中,使得 Reduce 作业中各个桶负载均衡. 文献[21]针对 Spark 在 Shuffle 阶段一次性分区导致的数据倾斜问题,提出了基于迭代填充的分区映射算法. 文献[22]在 Spark 框架下实现了迭代式数据均衡分区策略,将数据块细分成微分区并迭代循环处理,以不断调整产生的数据倾斜. 但这种多阶段分区方法中,分区的时机和选取数据的比例难以决策,尺度很难把控,不具通用性.

文献[23,24]针对 SQL 提出动态优化方法,引入工作窃取机制,自适应地将大任务的数据迁移到其他已完成任务的空闲节点中. 文献[25]提出一个 Spark 自适应倾斜缓解系统,利用预先收集的元数据动态缓解倾斜分布,将掉线任务块重新划分给其他空闲节点. 文献[26]提出一种基于线性回归分区预测的均衡负载机制,利用线性回归预测分区大小,识别倾斜分区,再根据细粒度资源分配算法调整数据的分区分配. 这些方法可以归类为迁移分区,在检测到数据倾斜后,将重负载节点的数据迁移到轻负载节点上,以更好地利用空闲资源. 但同时这类方法会导致计算过程中断,数据传输延迟,大大增加分区间的数据传输负载量.

鉴于上述两类分区方法中显著的缺陷和难以消除的漏洞,很多研究者使用了相对通用且高效的采样分区方法. 文献[27~29]所提出的分区方法使用贪心策略的思想,将样本数据依次分配给当前负载最小的节点或分区,从而减缓数据倾斜. 文献[30,31]都是先对样本数据进行哈希分区,若原哈希分区的剩余容量不满足要求,则按照算法重新将数据分配到其他分区. 文献[32]提出了键重分配和分割的分区算法,并设计了基于哈希键值重分配的分区器(Key Reassigning Hash-base Partition,KRHP),在对数据拒绝采样并预估权值分布后,先对样本数据执行哈希分区,然后将导致分区过载的数据拉取到负载量较小的其他分区,对于未被采样的数据则按照哈希分区分配. 文献[33]则针对异构环境对文献[32]的分区算法做了优化,将数据重新分配到资源充足的节点. 文献[34]针对以往方法动态适应性差和粒度不足的问题,提出了一种基于蓄水池抽样的动态平衡分区方法,针对不同类型应用的每个子阶段自适应地更新分区方案. 然而,这些采样分区方法把更多的重点放在样本上,专注于对样本数据的均衡分区,忽略了其他未被采样的数据类别,或者只能增加通信成本来对数据进一步均衡分区.

在实际应用中,存在很多包含大量低频数据的应用场景,如每天正常起飞的航班与不同原因延迟的航班,如文献网站,不同领域的专业术语多且杂. 此时,采样难以全覆盖所有类别,而未被采样的数据类别总量

又无法忽视,仅针对样本均衡分区的方法很难获得好的均衡分区效果. 针对这一问题,本文提出了基于优先填补策略的Spark数据均衡分区方法,并设计了一个高效的优先填补分区器(First Filling Partitioner,FFP). 该方法的主要操作是对所有待分配数据采样估计后,在容忍度允许的条件下,将样本数据中出现的类别优先分配到前面的分区中,把未被采样的类别直接指向最后一个分区. 相较其他采样分区方法,这一方法提出了分区容忍度与分区范围的概念,利用容忍度控制分区范围的大小进而约束每个分区的数据量,从而达到数据均衡分区的效果. 另外,这一方法将样本中出现的已知类别优先分配到前面分区中,为未知类别在后面分区中预留了一定空间,便于实现全部数据的均衡分区. 最后,该方法将非样本类别直接分配到最后一个分区,消除了其他采样分区方法在分区阶段采集各个分区信息的环节,不仅避免了主节点与从节点之间的额外通信开销,也节省了不必要的存储成本和计算成本. 通过在仿真数据集和真实数据集上实验验证,与默认分区方法和其他采样分区方法相比,本文分区方法能够有效缓解Spark计算过程中的分区数据倾斜,缩短程序的运行时间,提高Spark任务的运行效率.

2 预备知识

2.1 Spark的默认分区器

Spark提供了两种分区器,分别是哈希分区器(HashPartitioner)和范围分区器(RangePartitioner).

HashPartitioner,顾名思义就是根据哈希值进行分区. 它是最广泛使用的分区器,也是Spark中非排序类算子默认使用的分区器. HashPartitioner首先获取可用的分区数目,然后判断待分区的键值对是否为空,若为空则返回整数0,否则计算该键值对的键哈希值. 接着,将得到的哈希值对分区数目取模,若结果值为正整数则直接返回,若结果值为负数,则需要另外加上分区数目使其变为正数后再返回. HashPartitioner的优点就是运行速度快,简单易懂,且能够保证具有相同键的所有数据分配到同一个分区. 但它也有一个明显缺点,一旦待分区的数据确定,那么各个分区的数据量就会确定,如果指向同一个哈希值或者取模结果值的键值对过多,就会导致某个分区分配到的数据量过大,进而导致数据倾斜,具体情况在下一小节中详细描述.

RangePartitioner就是对数据采样,根据样本信息划定范围,将在相同范围内的数据分配到相同的分区中. RangePartitioner需要键值可排序的输入数据,从而获得有序的输出结果. 由于本文研究的是没有有序输出需求的场景以及键值不一定可排序的数据,故后续讨论的默认分区器都是HashPartitioner.

2.2 Spark的数据倾斜

Spark以弹性分布式数据集(Resilient Distributed Datasets,RDD)[35]作为核心数据结构,RDD也是多个分区的集合. RDD之间通过算子运算建立血统关系和依赖关系. 算子包括转换算子和行动算子. 以是否产生Shuffle操作作为判定标准,不同的转换算子使RDD之间存在不同的依赖,包括宽依赖和窄依赖. 窄依赖是指两个RDD的数据分区一一对应. 宽依赖,也称Shuffle依赖,指上游RDD的数据重新分区,同一个分区的数据可以被下游RDD的多个分区继承.

Spark将任务运行过程以宽依赖为界划分为多个Stage[36],相邻Stage之间会发生Shuffle,每个Stage中有多个Task并行计算,且要等上一个Stage完成后才能开始. 每个Task都需要在执行器(executor)上运行,一个Task计算一个分区的数据. 因此在计算环境相同的情况下,分区的数据量是影响其对应Task运行时间的关键因素,进而也是影响所在Stage运行时间的重要变量. 遇到类别分布均匀的数据时,默认的HashPartitioner在Shuffle阶段能够很好地实现数据的均衡分区,此时下一个Stage中每个Task需要计算的数据量没有明显差距,那么在executor充足的情况下各个Task的运行结束时间也会比较接近,资源利用率高. 然而在处理各种类别分布极度不均匀的数据时,使用HashPartitioner可能会导致Shuffle后各个分区的数据量悬殊,造成数据倾斜,此时如果没有额外干预,下一个Stage中高负载量分区对应的Task所需的运行时间就会明显更长,会出现一部分executor还在运行而其他executor只能长时间空闲等待的场景. 因此,在Spark计算过程中,分区的数据倾斜会导致资源浪费,增长Stage的运行时间,降低程序的运行效率.

图1是词频统计(WordCount)程序执行的一个数据倾斜例子. 该程序先后使用了map、reduceByKey两个转换算子连接不同的RDD,其中reduceByKey会导致Shuffle操作,以它为边界前后分为两个Stage,分别为虚线圆角矩形内的Stage1和Stage2. 设置每个Stage的分区数为4,executor数量也为4. 在Stage1中,程序从HDFS中读取数据保存在RDD1中,RDD1通过map算子将数据扩展并保存为RDD2,这一过程各个分区的数据量均匀,4个Task的运行时间一致. 接着,RDD2调用了reduceByKey算子,发生了Shuffle操作,RDD2的数据需要根据键值重新分区获得RDD3. 默认情况下,HashPartitioner会将具有相同哈希值的键的对应数据映射到相同的分区中. 假设RDD2中键的哈希值为1的数据非常多,则Shuffle后分区1的数据量会远远大于分区2、3、4的数据量,就如图1中的RDD3所示,分区1的数据量是其他分区的两到三倍,这就是Spark计算过程中分

区的数据倾斜. 在Stage2的每个executor性能相同的情况下,分区1对应的Task的运行时间会更长,分区2、3、4的Task运行完成后对应的executor只能空闲等待分区1的Task完成,而不能提前进入下一个Stage,也就是数据倾斜会导致资源利用率低且任务执行效率低.

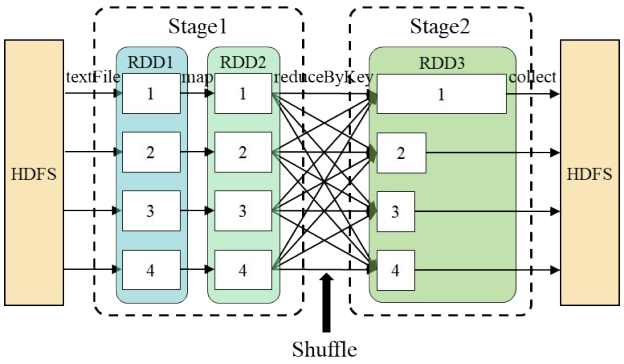


图1　WordCount程序的数据倾斜例子

3　基于优先填补策略的数据分区方法

3.1　问题定义

为了更好地阐述本文设计的优先填补策略分区方法的原理,本小节将对Shuffle过程的分区问题进行数学化描述和分析. 表1是对后续使用符号的简要说明,key表示待分区数据的键.

表1　符号说明

符号	含义
n	下一个Stage中的分区数
N	待分区的数据总量
P	各个分区分配到的数据量的数组
p_i	第i个分区分配到的数据量
V	各个key对应的数据量的数组
K	各个分区分配到的key数组的集合
C	各个分区分配到的key个数的数组

假设Shuffle前一个Stage生成的数据总量为N,下一个Stage可分配的分区数量为n,那么Shuffle过程就是将这N个数据重新分配到这n个分区中,即$N=\sum_{i=1}^{n}p_i$. 在executor充足且性能相同时,下一个Stage的运行时间取决于最大Task的运行时间,也就取决于最大分区的数据量. 用T表示运行时间,用f表示运行时间与分区数据量的函数关系,可以获得式(1):

$$T(\text{Stage})=f(\max P)=f(\max\{p_1,p_2,\cdots,p_n\}) \tag{1}$$

因此,为了能够有效缩短下一个Stage的运行时间,均衡分区方法应该最小化最大分区的数据量,也就是计算怎么分配数据能够使得数组P最大元素的值最小. 故目标函数g可以表示为:

$$g(V)=\min\max\{p_1,p_2,\cdots,p_n\} \tag{2}$$

$$p_i=\sum_{j=1}^{C_i}V(K_i^j) \tag{3}$$

其中,K_i为第i个分区分配到的所有key的数组,K_i^j表示第i个分区中的第j个key,C_i表示第i个分区分配到的key的个数,$V(x)$表示key为x的数据量.

此外,对于是否有效均衡分区,可以通过计算Shuffle后各个分区数据量的倾斜程度来验证. 本文引用了LIBRA算法[13]的变异系数来度量分区倾斜程度,Mean(P)表示Shuffle后所有分区的平均数据量,Stddev(P)表示Shuffle后所有分区数据量的标准方差,两者分别作为分母和分子,计算获得分区数据量的变异系数S,具体公式如下:

$$\text{Mean}(P)=\frac{\sum_{i=1}^{n}p_i}{n} \tag{4}$$

$$\text{Stddev}(P)=\sqrt{\frac{\sum_{i=1}^{n}[p_i-\text{Mean}(P)]^2}{n}} \tag{5}$$

$$S=\frac{\text{Stddev}(P)}{\text{Mean}(P)} \tag{6}$$

其中,变异系数S的范围为$[0,+\infty)$. S值越小,表示数据分配得越均匀,分区数据量的均衡程度越高;反之S值越大,说明数据分配情况不良好.

当一个key被分配到不同的分区,即执行不同的分区方案时,会获得不同的分区大小,最大分区的数据量也可能发生变化. 因此,计算目标函数$g(V)$,获得最佳的分区方案,使得变异系数几乎等于0,这是一个计算量相当大的问题,同时也是一个NP-hard问题. 在不知道数据实际分布的情况下,我们基本不可能通过计算得到最佳结果. 因此,现有的分区方法都是确定一份近似最优的分区方案,使得最大分区数据量尽可能小,各个分区的数据量尽可能地均衡.

3.2　方法原理

本文所提出的分区方法主要分为两个阶段,获取分区方案和对数据分区. 获取分区方案也可以看作预分区,指的是根据样本数据的分布情况,预先对样本数据中出现过的key指定分区,获取已知key的分区方案,分区方案仅包含key及其对应的分区号. 对数据分区,也可以看作实际分区过程,就是Shuffle过程中的每个key通过调用分区函数获取分区号,进而将数据传输到对应分区中.

在获取分区方案时,根据样本的预估结果,对样本中的key依照权值降序排列,然后在满足条件的情况下,依次优先分配到前面的分区中,样本中所有key分配完成后,可获得分区方案. 图2展示了基于优先填补策略的分区方法在获取分区方案阶段的主要流程,具

体分为以下几个步骤：

（1）将样本数据中的key按照权值降序排列. 估算样本中每个key在全量数据中的权值（记为weight），获得数组 $W=\{(\text{key}_1,\text{weight}_1),\cdots,(\text{key}_{\text{Nc}},\text{weight}_{\text{Nc}})\}$，对数组 W 按照权值降序，获得有序数组 $W_{\text{sorted}}=\{(\text{key}_1',\text{weight}_1'),\cdots,(\text{key}_{\text{Nc}}',\text{weight}_{\text{Nc}}')\}$，其中Nc为样本中不同key的个数. key的权值可以看作key对应的数据量.

（2）初始化分区数组，计算分区数据量的平均值和阈值. 先初始化一个大小为 n、元素全为0的数组 P，即 $P=\{p_1,p_2,\cdots,p_n\}=\{0,0,\cdots,0\}$，用来记录当前各个分区已分配的数据量. 计算有序数组 W_{sorted} 中所有key的权值平均值mean. 获取用户定义的分区容忍度tolerance，取值范围为（1.0，2.0），表示用户可接受的分区倾斜程度，将容忍度与平均值相乘获得阈值threshold. 具体计算如下：

$$\text{mean}=\text{Mean}(\text{weight})=\frac{1}{\text{Nc}}\sum_{j=1}^{\text{Nc}}\text{weight}_j' \tag{7}$$

$$\text{threshold}=\text{mean}\times\text{tolerance} \tag{8}$$

（3）顺序获取有序数组 W_{sorted} 的下一个元素. 通过循环遍历数组 W_{sorted} 的每一个元素并赋值给指定变量，即 $(k,w)=(\text{key}_j',\text{weight}_j')$，其中 j 的取值范围为 $[1,\text{Nc}]$，初始值为1. 当所有key都分配完成时，保存好分区方案，退出循环.

（4）获取下一个分区的信息. 按照索引从小到大依次访问数组 P，获取分区信息 p_i，其中 i 的取值范围为 $[1,n]$，初始值为1. 当 $i>n$ 即后续没有其他分区时，执行步骤（7）.

（5）判断当前key能否放入对应分区. 从步骤（3）和步骤（4）中分别获取待分配的key信息 (k,w) 和分区信息 p_i，判断该key是否适合放入该分区中：

$$p_i<\text{mean} \tag{9}$$

$$p_i+w<\text{threshold} \tag{10}$$

若不满足式（9），表明当前分区数据量已经不小于平均值，对应图2中的情况1. 若不满足式（10），则说明加上当前key的权值后，该分区的数据量会超过阈值，对应图2中的情况2. 当且仅当式（9）和式（10）都能满足时，将该key分配到这一分区中，执行下一步；否则令 $i=i+1$，返回步骤（4）.

（6）更新分区信息. 通过步骤（5）确定了key分配的分区后，需要在分区方案中将该key指向分配的分区号，并及时更新当前分区的数据量. 接着令 $j=j+1$，$i=1$，再次执行步骤（3）.

（7）当所有分区都无法同时满足式（9）和式（10）时，则直接将该key分配到当前数据量最小的分区中，并更新相关信息. 然后令 $j=j+1$，$i=1$，再次执行步骤（3）.

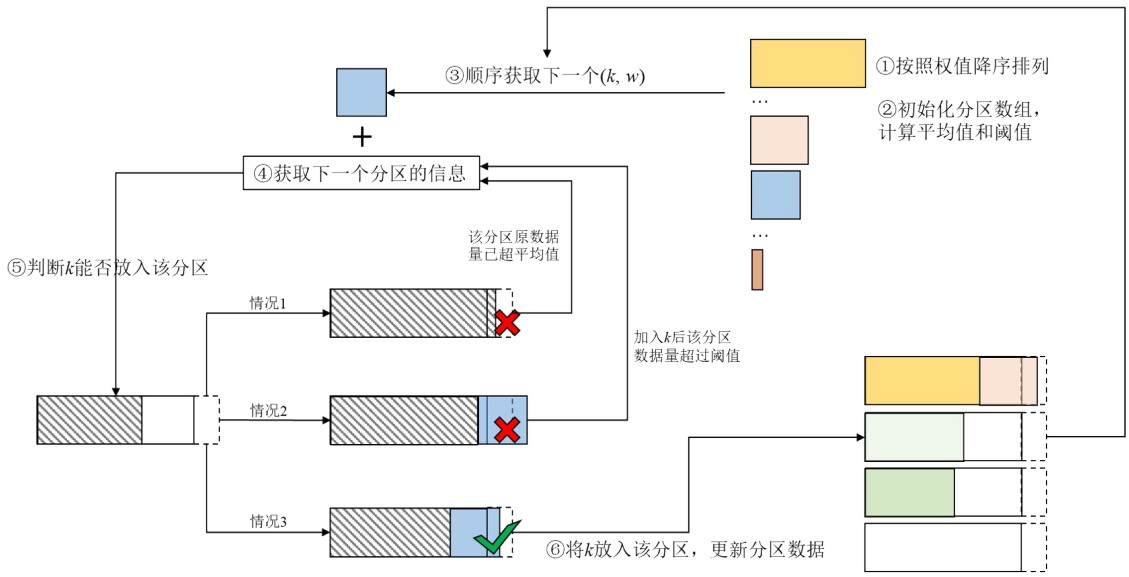


图2 获取分区方案的流程图

在获取分区方案的过程中，通过两层循环，逐一将样本中出现的key分配到适当的分区中，获得针对样本的分区方案. 算法1是获取分区方案的伪代码.

在对全量数据正式分区时，先判断每一个待分区数据的key是否包含在已有分区方案中. 若包含，则返回分区方案中该key指向的分区索引号；若不包含，则直接返回最后一个分区的索引号. 算法2是对数据正式分区的伪代码.

优先填补策略的分区方法有两个关键变量，一个是样本数据中所有key的权值平均值，一个是用户可容忍范围内的分区阈值，将平均值和阈值分别作为分区条件的下限和上限. 首先，平均值代表着完全理想的分

算法1　获取分区方案

输入:样本数据W,分区容忍度tolerance
输出:分区方案keyToPartition
1:对W按照weight降序排列,获得W_{sorted}
2:mean←Mean(W_{sorted},weight):
3:threshold←mean×tolerance
4:P←zeros(n)
5:FOR (k,w) IN W_{sorted}:
6:　FOR i=1 to i=n:
7:　　p←P(i)
8:　　IF p=0 OR p满足式(9)和式(10):
9:　　P(i)←p+w
10:　　keyToPartition中添加(k,i)
11:　　退出该层循环
12:　END IF
13:　END FOR
14:　IF i>n:
15:　　获取P中最小值的索引minIndex
16:　　P(minIndex)←P(minIndex)+w
17:　　keyToPartition中添加(k,minIndex)
18:　END IF
19:END FOR

算法2　对数据分区

输入:待分区数据的key,分区方案keyToPartition
输出:分区索引号
1:IF keyToPartition中包含key:
2:　返回keyToPartition.getValue(key)
3:ELSE
4:　返回最后一个分区的索引号n
5:END IF

区状态下各个分区的数据量大小,也就是在实际分区中,最大分区的数据量不可能低于平均值的大小.因此,当分区数据量还未达到平均值时,向该分区继续分配数据不会影响最后的最大分区数据量;反之则不适合再继续分配数据.其次,该方法通过阈值来限制优先填补策略下分区的数据倾斜程度.如果能一直限制分区数据量在阈值范围内,就能控制最终各个分区数据量还是相对平衡的,没有出现哪一个分区负载过大的情况;相反,如果某个分区数据量超出了阈值范围,表明破坏了分区平衡.当分区数据量居于平均值至阈值这一范围内时,各个分区相对均衡且为后续非样本数据预留了空间,故将平均值和阈值分别作为在获取分区方案时每个分区数据量的上下限.

而这个上下限的范围大小由用户定义的分区容忍度来确定,对最终的均衡分区效果至关重要.当容忍度的取值较大时,分区数据量的上限也就变大,在获取分区方案时更多的数据会被分配到前面的分区中,后面的分区分配到的数据少,为非样本的数据预留的空间大.当容忍度的取值较小时,分区数据量的上限会接近下限值,那么在获取分区方案阶段,样本中出现的键的数据就会相对均匀地被分配到各个分区中,此时为非样本的数据预留的空间小.可见,容忍度的取值大小对均衡分区效果的影响主要在于为非样本数据的预留空间的大小,以及前面分区与后面分区之间的数据量均衡.因此,在常见的采样方法下,对于低频数据较多的数据集,非样本类别的数据会比较多,需要预留的空间多,分区容忍度取值可以大一点;当数据量较小或低频数据比例低时,非样本类别的数据会比较少,容忍度取值可以相对小一些.

针对现有采样分区方法存在的对采样过于依赖、数据传输量大、通信开销高等问题,基于优先填补策略的分区方法对Shuffle阶段的数据分区做了改进和优化.首先,该方法使用分区容忍度来确定上下限,约束获取分区方案阶段的每个分区数据量大小;其次,在获取分区方案阶段,该方法将样本中的键优先分配到前面的分区中,为非样本的键预留后面分区的空间,进而实现Shuffle后全部数据的均衡分区;最后,在对数据分区时,将非样本的键直接分配到最后一个分区中,简单方便,避免了主程序中额外的存储、计算和通信开销.

3.3　具体实现

我们基于上述提出的优先填补策略分区方法设计了新的分区器FFP,它主要包含四个组件,分别是采样、权值评估、获取分区方案和对数据分区.图3展示了FFP的架构以及Spark程序使用FFP的计算流程.

采样模块引用了文献[32]提出的采样方法,即基于步伐的拒绝采样算法.它在参与Shuffle操作的每个RDD的每个分区中分别采样,统计样本中每个key的频率,并计算分区的采样率.

在权值评估模块中,将每个分区的样本统计结果除以采样率,然后将所有分区的计算结果按照不同key分别汇总求和,获得每个key在全量数据中的权值.

获取分区方案模块和对数据分区模块是对本文3.2节所提出的分区方法的具体实现.在获取分区方案模块中,先对权值评估模块获得的结果根据权值进行降序,再通过两层循环将有序数组中的key逐一分配到各个分区中,获得分区方案.分配时遵循优先填补策略的原则,设置分区数据量的上下限,在条件范围内将key尽可能地填补到靠前的分区中;若出现所有分区都未能满足条件的情况,则将key分配到当前数据量最小的分区中.

在对数据分区模块中,对于每个传入参数key,倘若分区方案中包含该key,则返回分区方案中该key指向的分区号,反之则返回最后一个分区的分区号.

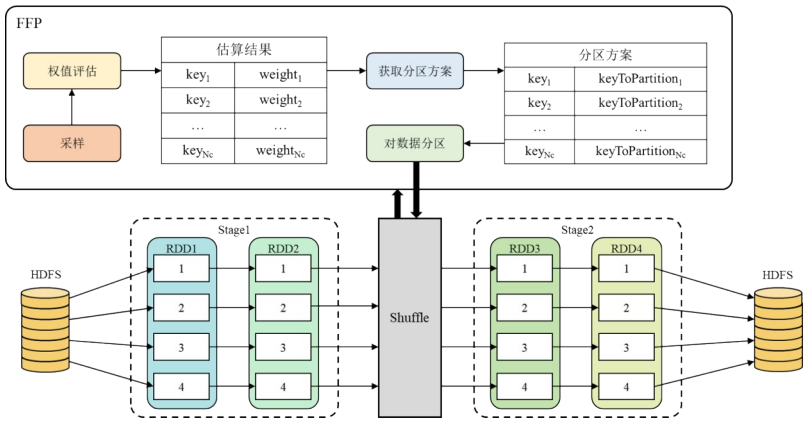


图3 FFP的架构及计算流程

如图3所示,Spark任务计算过程中,在Shuffle阶段调用FFP对数据进行分区. FFP先对数据采样,然后根据样本数据估算每个key在全量数据中的权值,基于估算结果获取样本中的key的分区方案,最后在分区方案的辅助下,对Shuffle阶段的所有数据进行均衡分区,分区后的数据作为下一个Stage的输入数据.

4 实验与分析

4.1 实验设置

实验环境是由5个节点构成的集群,总内存大小为446 GB,总核数为480个,操作系统是Centos Linux 7(Core),Hadoop的版本是2.7.4,Spark的版本是2.4.0,Scala的版本是2.11.12. 本文执行Spark任务时,统一设置8个executor,每个executor设置2个核,Driver内存设置为16 GB. 由于分布式环境下运行具有不稳定性,本文的实验结果都是重复运行5次结果的平均值.

实验数据包含仿真数据集和真实数据集. 仿真数据集是存在分区倾斜的Zipf分布数据集[37~39],一个Zipf数据集分别包含10个子数据集,每个子集的数据为单列整数,满足指数为γ的标准Zipf分布,γ的取值范围是[0.5,5.0],增量为0.5. 指数γ的值越大,表明数据的倾斜分布越明显. 真实数据集使用的是美国运输统计局发布的航空数据(Bureau of Transportation Statistics,BTS)[40],实验选取了2018年1月至2023年12月的US航班数据,属性值包括年份、月份、起点和终点的相关信息、是否延误以及延误原因等,测试时按照年份不同划分为多个子集. 仿真数据集的实验都是在Shuffle前分区数目为32,Shuffle后分区数为8的条件下完成. 真实数据集的实验中Shuffle前分区数目为24,Shuffle后分区数为8. 采样比例都设为0.1.

测试程序选取了三个应用程序,分别是WordCount(词频统计)、Join(表格连接)和Subtract(去除交集),对应产生Shuffle的算子分别是groupByKey、join和subtractByKey. 表2展示了不同应用程序和对应数据集的具体信息.

表2 应用程序和数据集的具体信息

应用程序	数据集	总大小	子集数据量
WordCount	Zipf	4.13 GB	1亿+
Join	Zipf	371 MB	1千万+
Subtract	Zipf	371 MB	1千万+
WordCount	BTS	2.72 GB	500万~800万

4.2 评估指标

均衡分区的主要目的是使得Shuffle后各个分区的数据量相对平衡,提高整个Spark任务的资源利用率和计算效率. 鉴于该目标,实验设置了以下4个指标来评估不同分区器的性能:

(1)任务的总运行时间. 均衡分区的最终目标就是缩短数据倾斜分布下的Spark任务运行时间;

(2)下一个Stage的运行时间. 指Shuffle后紧接着的Stage的运行时间,该值可以更直观地展现均衡分区的效果;

(3)最大分区的数据量. 由本文3.1小节的分析可知,该值越小,说明均衡分区的效果越好;

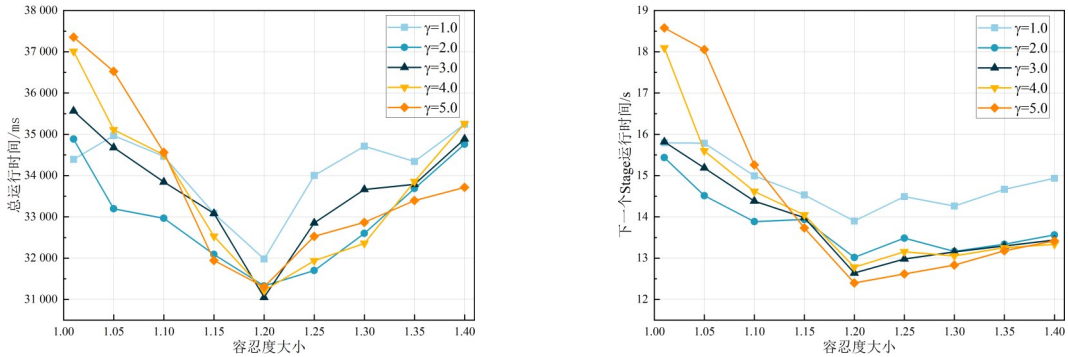
(4)分区数据量的变异系数. 根据本文3.1节的式(4)~(6)计算Shuffle后各分区数据量的变异系数,该值越小,说明均衡分区效果越好.

4.3 不同容忍度对FFP性能的影响

为了验证基于优先填补策略的均衡分区方法的可行性,本文在仿真数据集和真实数据集上测试了不同容忍度对FFP性能的影响. 从本文3.2节可知,容忍度的大小应适宜,才能获得好的分区效果. 因此,本文在相同数据集中选取不同的容忍度值进行实验,范围为1.01和[1.05,1.40],增量为0.05,对比程序的运行时间,测试容忍度值的增减对FFP性能的影响,以获得合适的容忍度取值. 为了使优先填补策略发挥作用,给非样本数据预留空间,规定容忍度值应大于1.0,故实验最低值取1.01. 容忍度值不宜过大,否则Shuffle后各分区数

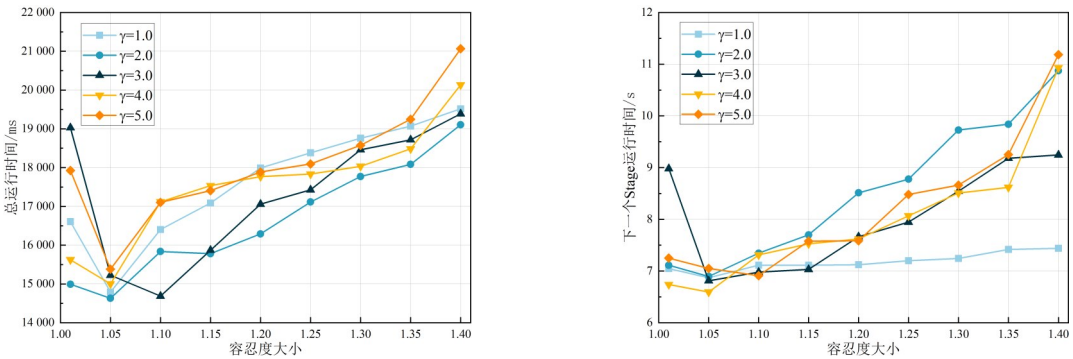
据量仍然倾斜,故实验最高值取1.40,更高的取值对FFP的性能测试没有意义.

为了验证在不同倾斜程度的数据集中容忍度取值对FFP性能的影响是否一致,本文在5个不同γ值的Zipf数据集上做对照实验,γ值的取值范围为[1.0,5.0],增量为1.0. 图4、图5和图6分别展示了不同容忍度下WordCount程序、Join程序和Subtract程序在Zipf数据集上使用FFP的运行结果. 其中,横坐标都为容忍度大小,图4(a)、图5(a)和图6(a)的纵坐标为总运行时间,单位为ms,图4(b)、图5(b)和图6(b)的纵坐标是下一个Stage的运行时间,单位为s. 可以看出,在不同γ值的Zipf数据集中,三个应用程序的总运行时间和下一个Stage运行时间的变化趋势基本一致,容忍度取值过大或过小时,程序运行效率都会降低. 在图4中,随着容忍度的增大,WordCount的运行时间先逐步减少;容忍度取1.20时,在不同γ值的Zipf数据集上程序的总运行时间和下一个Stage运行时间都达到最小值;当容忍度超过1.20后,程序的运行时间又逐步增大,运行效率逐渐降低. 如图5



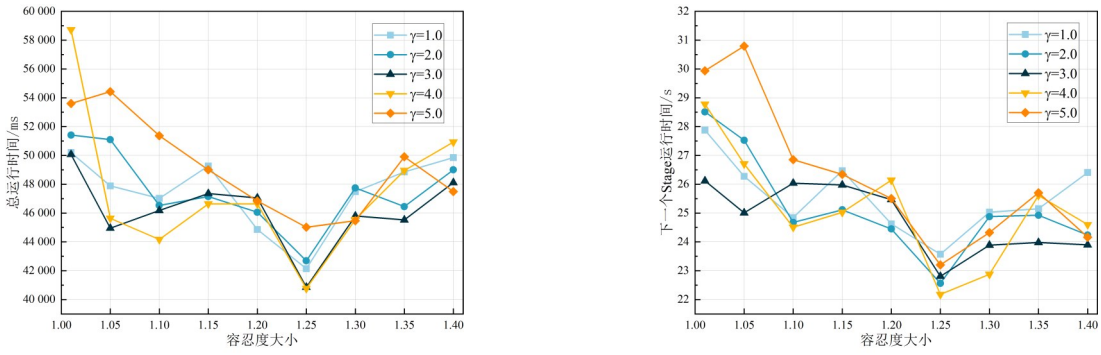
(a) 总运行时间　　(b) 下一个Stage运行时间

图4　不同容忍度下FFP在Zipf数据集的WordCount运行结果



(a) 总运行时间　　(b) 下一个Stage运行时间

图5　不同容忍度下FFP在Zipf数据集的Join运行结果

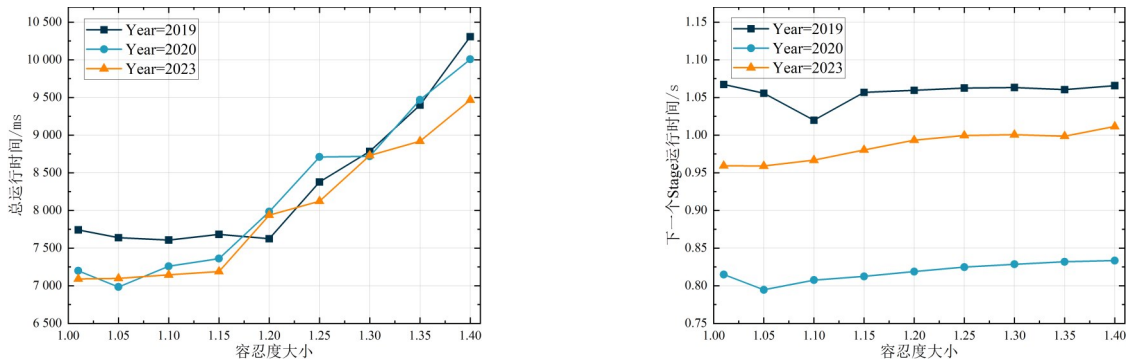


(a) 总运行时间　　(b) 下一个Stage运行时间

图6　不同容忍度下FFP在Zipf数据集的Subtract运行结果

所示，在Join程序中，当容忍度为1.05时，总运行时间和下一个Stage运行时间基本达到最低点，而后随着容忍度大小的递增，两个指标值也都不断增长. 由图6可以发现，在不同γ值的Zipf数据集中，Subtract程序的总运行时间和下一个Stage运行时间大多有两个极小值点，第一个极小值点大概在容忍度为1. 10处取得，第二个极小值点在容忍度为1. 25处取得，后者也是在同一γ值数据集上Subtract程序运行时间的最小值点.

为了验证不同容忍度下FFP在真实数据集的运行效果，本文在BTS数据集上进行了WordCount程序测试，选取了2019、2020和2023年的子数据集，分别代表了BTS中数据量较多、较少和中等三个等级. 图7(a)和图7(b)分别是不同容忍度下FFP在三个子集上的总运行时间和下一个Stage运行时间. 可以看出，当容忍度取值在1.01~1.10内时，FFP的总运行时间和下一个Stage运行时间都非常地接近. 而当容忍度取值大于1.10时，FFP的运行时间随着容忍度增大而逐渐增加.



(a) 总运行时间 (b) 下一个Stage运行时间

图7 不同容忍度下FFP在BTS数据集的WordCount运行结果

从Zipf数据集的运行结果可以看出，在数据量为1亿且低频数据较多的情况下，FFP在WordCount程序中的最佳容忍度为1.20；1千万左右的数据量下，FFP在Join程序中的最佳容忍度为1.05，在Subtract程序的最佳容忍度为1.25. 而在数据量为百万级且低频数据较少的BTS数据集上，FFP在WordCount程序中的最佳容忍度范围为1.01~1.10. 假定某个类别的数据量低于总数据量的0. 01%时，该类别数据看作低频数据，那么WordCount程序的Zipf数据集的低频数据量占比大约为6.0%，而BTS数据集的低频数据量平均占比为1.32%，故FFP在Zipf数据集的最佳容忍度比BTS数据集的略高.

上述实验验证了本文第3.2节的分析，在优先填补策略下，分区容忍度的取值对分区效果具有重要影响. 正确的容忍度值可以获得更好的均衡分区结果，大大提高基于优先填补策略的分区方法下的Spark任务运行效率. 对于类别分布和特征信息等完全未知的数据，可以将分区容忍度的取值设为经验值1.10，或者随机选取一小块数据做参数测试，选择效果最好的一个作为最终容忍度的取值.

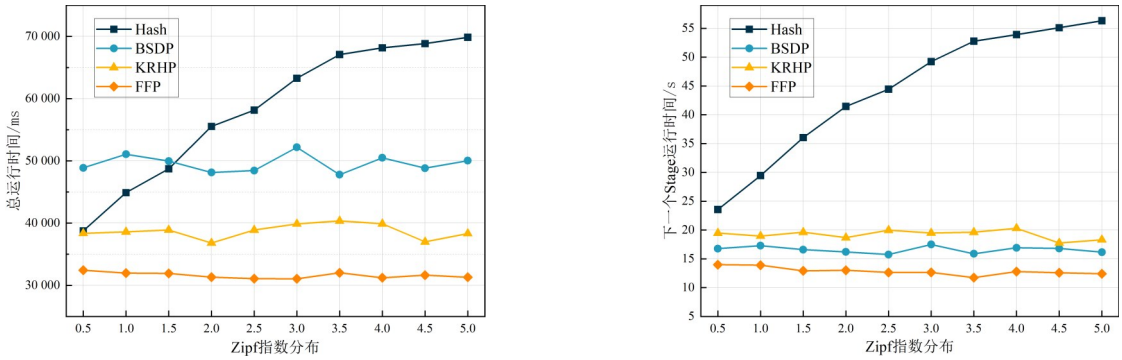
4.4 不同分区器在Zipf数据集上的对比

为了证明基于优先填补策略的分区方法的优越性，本文将FFP与默认分区器HashPartitioner(图表中简写为Hash)、分区器BSDP[28]、分区器KRHP[32]在三个应用程序上进行了对比. 本文的均衡分区主要针对的是存在哈希倾斜的数据，故将Spark的HashPartitioner作为基准分区器进行对比. BSDP是基于贪心策略而提出的，KRHP是基于倾斜重分配策略而提出的，两者使用的分区方法都是现有采样分区方法中较为先进且具有代表性的方法.

图8~10依次是WordCount、Join和Subtract程序使用不同分区器在不同γ值的Zipf数据集上的运行结果. 其中，横坐标为Zipf分布指数γ，取值范围为[0.5，5.0]，增量为0.5. 图8(a)、图9(a)和图10(a)的纵坐标是程序的总运行时间，单位为ms，图8(b)、图9(b)和图10(b)的纵坐标是下一个Stage的运行时间，单位为s. 从三个应用程序的运行结果看，使用HashPartitioner的程序的总运行时间和下一个Stage运行时间都随着γ的增长而增长，而使用另外三种均衡分区器的程序运行时间相比之下较为平稳，基本在一个范围内波动且远远小于HashPartitioner的运行时间，其中FFP基本是三者中总运行时间和下一个Stage运行时间最短的. 在图8的WordCount程序中，FFP的总运行时间比BSDP平均缩短了36.2%，比KRHP平均缩短了18.3%；而在下一个Stage运行时间这一指标上，FFP比BSDP平均降低了22.4%，FFP比KRHP平均降低了33.0%. 在图9的Join程序中，BSDP的总运行时间与KRHP的差别不大，而FFP的总运行时间很明显低于前两者，分别平均缩短了

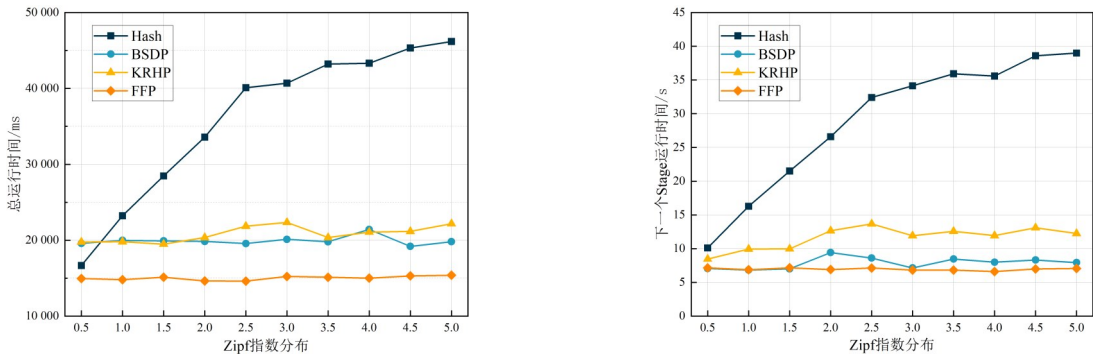
24.6%和27.8%;在下一个Stage运行时间上,FFP比BSDP平均缩短了10.9%,比KRHP平均缩短了39.0%.在图10的Subtract程序中,三个均衡分区器的运行时间都随着γ值增长有些起伏,但基本限制在一定范围内上下波动.在Subtract程序的总运行时间上,FFP比BSDP平均缩短了21.5%,比KRHP平均缩短了16.3%;在下一个Stage运行时间上,FFP与BSDP之间差别不大,FFP较KRHP平均缩短了20.7%.

从结果分析可以看出,FFP在WordCount、Join和Subtract这三个程序上都具有明显优势.在使用HashPartitioner的程序的运行时间随着数据倾斜度的增长而增长时,使用FFP的程序的运行时间始终维持在一个较低的范围内,这表明本文提出的分区方法和FFP有效缓解了Spark任务计算过程中分区数据倾斜的问题.与BSDP和KRHP的结果相比,使用FFP的应用程序的总运行时间平均缩短了27.4%和20.8%,这表示FFP比其他分区器能够更好地实现Spark任务计算过程中数据的均衡分区,更好地提高程序的运行效率,具有更优



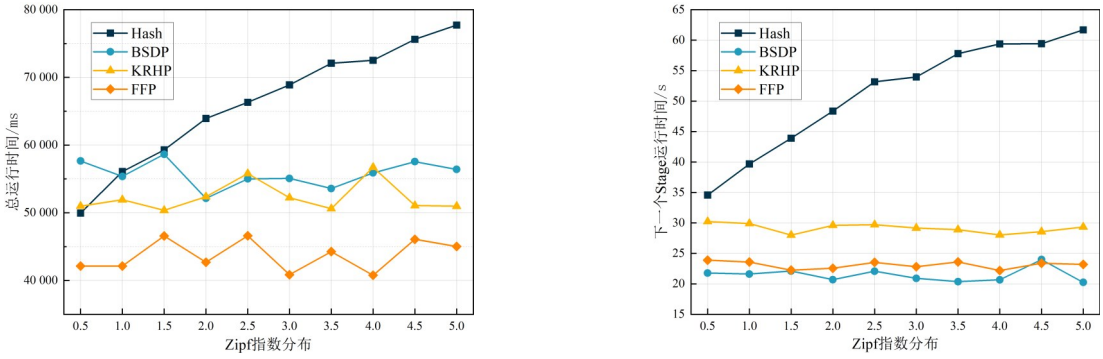
(a) 总运行时间　　(b) 下一个Stage运行时间

图8　不同分区器的WordCount运行结果



(a) 总运行时间　　(b) 下一个Stage运行时间

图9　不同分区器的Join运行结果



(a) 总运行时间　　(b) 下一个Stage运行时间

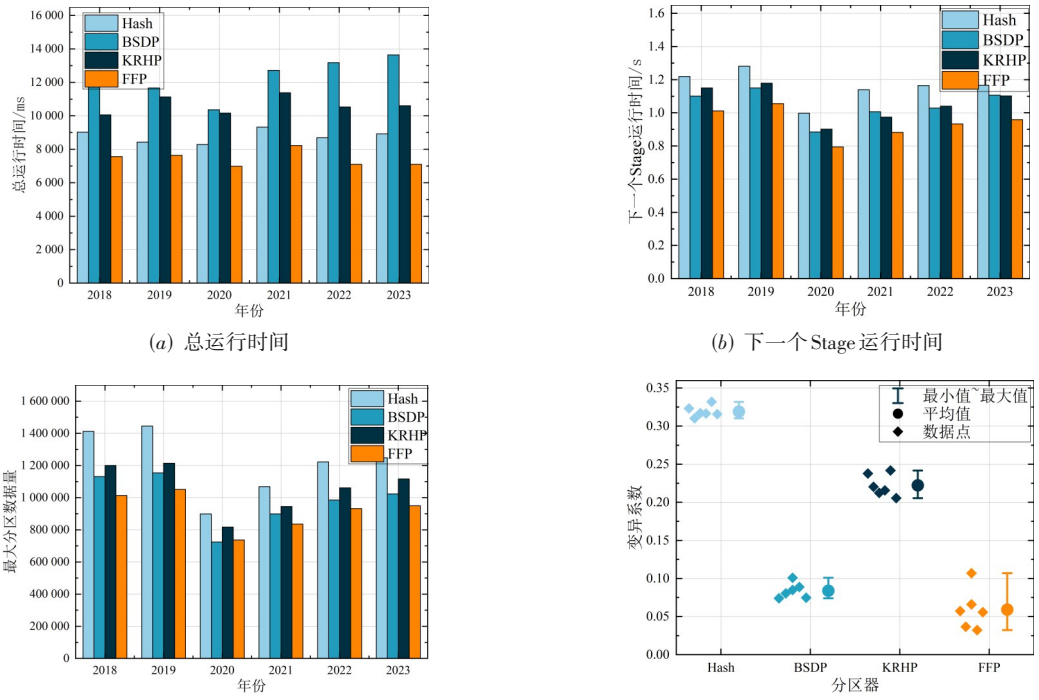
图10　不同分区器的Subtract运行结果

越的大数据计算性能.

4.5 不同分区器在真实数据集上的对比

为了更好地展示基于优先填补策略的分区方法在实际场景中的应用效果,本文在真实数据集BTS上进行WordCount程序实验,将FFP与HashPartitioner、BSDP[28]、KRHP[32]作对比,比较分区效果. 从4.3节可知,当容忍度为1.01~1.10时,FFP在BTS数据集上的均衡分区效果最好,故为了更好地展现FFP的性能优越性,对比实验中FFP的容忍度取该范围的中间值1.05.

图11为在BTS数据集上使用不同分区器执行WordCount程序的运行结果. 其中,图11(a)~(c)的横坐标为BTS统计的US航班年份,每个年份为一个子集. 图11(a)的纵坐标为总运行时间,单位为ms;图11(b)的纵坐标为下一个Stage运行时间,单位为s;图11(c)的纵坐标为Shuffle后最大分区数据量. 从图11(a)~(b)看,使用FFP的程序的运行效率明显高于使用其他三种分区器的程序的运行效率,FFP的总运行时间比HashPartitioner、BSDP和KRHP分别平均缩短了15.3%、38.7%和30.2%,FFP的下一个Stage运行时间则分别平均缩短了19.2%、10.3%和11.2%. 图11(c)~(d)则从各分区数据量这一方面展示FFP与其他分区器的性能. 从图11(c)可以看出,使用FFP的程序,Shuffle后最大分区数据量明显低于其他三种分区器的程序,FFP的最大分区数据量比HashPartitioner平均降低了23.8%,比BSDP平均降低了6.2%,比KRHP平均降低了12.9%. 图11(d)的横坐标是不同分区器,纵坐标是变异系数. 每个分区器的结果包含6个数据点(左)和1个箱体(右),数据点为分区器在不同子集上的具体变异系数,箱体显示了所有变异系数的最小值、最大值和平均值. 可以看到,FFP的变异系数明显是最低的,平均值只有0.059,比HashPartitioner、BSDP和KRHP的分别降低了0.260、0.025和0.163. 从上述分析可知,FFP在真实数据集上较其他分区器的分区效果更好,使得Shuffle后各个分区的数据量更加均衡,能够明显缩短任务的运行时间.



(a) 总运行时间 (b) 下一个Stage运行时间

(c) 最大分区数据量 (d) 变异系数

图11 不同分区器在BTS数据集的WordCount运行结果

5 结束语

针对Spark任务计算过程中分区间数据倾斜的问题,本文对其进行了定义及建模分析,提出了基于优先填补策略的Spark数据均衡分区方法. 通过对所有参与Shuffle的数据进行采样来估算每个键的权值,将样本的键依据权值大小降序排列后,按照优先填补策略逐步分配到各个分区中,生成针对样本的分区方案,然后将未被采样的键直接指向最后一个分区,实现Shuffle后各个分区的数据量相对均衡. 通过在标准Zipf分布数据集和真实数据集BTS上验证得到,与默认分区方法和其他均衡分区方法相比,基于优先填补策略的分区方法能够为Shuffle过程提供更为高效合适的数据分区方案,获得更好的均衡分区效果,能够更加有效地提高Spark任务的运行效率.

在未来的工作中,针对在线环境和异构环境下的倾斜数据任务,我们将考虑使用粒计算的策略[41]对优先填补分区器进行更加深入的功能完善与性能提高,实现分区容忍度的自适应确定方法等.

参考文献

[1] KATAL A, WAZID M, GOUDAR R H. Big data: Issues, challenges, tools and Good practices[C]//2013 Sixth International Conference on Contemporary Computing (IC3). Piscataway: IEEE, 2013: 404-409.

[2] DITTRICH J, QUIANÉ-RUIZ J A. Efficient big data processing in Hadoop MapReduce[J]. Proceedings of the VLDB Endowment, 2012, 5(12): 2014-2015.

[3] ZAHARIA M, CHOWDHURY M, FRANKLIN M J, et al. Spark: Cluster computing with working sets[C]//2nd USENIX Workshop on Hot Topics in Cloud Computing (HotCloud). Berkeley: USENIX Association, 2010: 1-7.

[4] CARBONE P, KATSIFODIMOS A, EWEN S, et al. Apache flink: Stream and batch processing in a single engine[J]. Bulletin of the IEEE Computer Society Technical Committee on Data Engineering, 2015, 38(4): 28-38.

[5] SHVACHKO K, KUANG H R, RADIA S, et al. The hadoop distributed file system[C]//2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST). Piscataway: IEEE, 2010: 1-10.

[6] DEAN J, GHEMAWAT S. MapReduce: Simplified data processing on large clusters[J]. Communications of the ACM, 2008, 51(1): 107-113.

[7] GEETHA J, HARSHIT N G. Implementation and performance comparison of partitioning techniques in apache spark[C]//2019 10th International Conference on Computing, Communication and Networking Technologies (ICCCNT). Piscataway: IEEE, 2019: 1-5.

[8] BAERT Q, CARON A C, MORGE M, et al. An adaptive multi-agent system for task reallocation in a MapReduce job[J]. Journal of Parallel and Distributed Computing, 2021, 153: 75-88.

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

[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.
LIANG J J, HE L M. Join algorithm in skewed datasets based on MapReduce[J]. Computer Science, 2016, 43(9): 27-31. (in Chinese)

[11] LU W, CHEN L, WANG L Q, et al. NPIY: A novel partitioner for improving mapreduce performance[J]. Journal of Visual Languages & Computing, 2018, 46: 1-11.

[12] 陶永才, 丁雷道, 石磊, 等. MapReduce在线抽样分区负载均衡研究[J]. 小型微型计算机系统, 2017, 38(2): 238-242.
TAO Y C, DING L D, SHI L, et al. Research on MapReduce on-line load balancing based on sample partition[J]. Journal of Chinese Computer Systems, 2017, 38(2): 238-242. (in Chinese)

[13] CHEN Q, YAO J Y, XIAO Z. LIBRA: Lightweight data skew mitigation in MapReduce[J]. IEEE Transactions on Parallel and Distributed Systems, 2015, 26(9): 2520-2533.

[14] SINGH B, VERMA H K. IMSM: An interval migration based approach for skew mitigation in MapReduce[J]. Recent Advances in Computer Science and Communications, 2021, 14(1): 71-81.

[15] 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.
WANG Z, CHEN Q, LI Z H, et al. An incremental partitioning strategy for data balance on MapReduce[J]. Chinese Journal of Computers, 2016, 39(1): 19-35. (in Chinese)

[16] ALAMRO S, LAN T, SUBRAMANIAM S. Forseti: Dynamic chunk-level reshaping for data processing on heterogeneous clusters[J]. Journal of Parallel and Distributed Computing, 2023, 171: 14-23.

[17] 侯震梅, 杨玉莹. 分布式数据流数据倾斜均衡方法研究[J]. 长春大学学报(自然科学版), 2020, 30(5): 11-20.
HOU Z M, YANG Y Y. Research on data skew equalization method for distributed data streams[J]. Journal of Changchun University, 2020, 30(5): 11-20. (in Chinese)

[18] KWON Y, BALAZINSKA M, HOWE B, et al. SkewTune: Mitigating skew in mapreduce applications[C]//Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data. New York: ACM, 2012: 25-36.

[19] HOU X F, ASHWIN K T K, THOMAS J P, et al. Dynamic workload balancing for Hadoop MapReduce[C]//2014 IEEE Fourth International Conference on Big Data and Cloud Computing (BDCloud). Piscataway: IEEE, 2014: 56-62.

[20] TANG Z, ZHANG X S, LI K L, et al. An intermediate data placement algorithm for load balancing in Spark com-

puting environment[J]. Future Generation Computer Systems, 2018, 78: 287-301.

[21] 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.
BIAN C, YU J, XIU W R, et al. Partitioning and mapping algorithm for in-memory computing framework based on iterative filling[J]. Journal of Computer Applications, 2017, 37(3): 647-653. (in Chinese)

[22] 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.
ZHANG Y M, JIANG J B, LU J W, et al. An iterative data partitioning strategy for MapReduce[J]. Chinese Journal of Computers, 2019, 42(8): 1873-1885. (in Chinese)

[23] HE Z Y, HUANG Q L, LI Z F, et al. Handling data skew for aggregation in spark SQL using task stealing[J]. International Journal of Parallel Programming, 2020, 48(6): 941-956.

[24] SHEN Y J, XIONG J, JIANG D J. SrSpark: Skew-resilient spark based on adaptive parallel processing[C]//2020 IEEE 26th International Conference on Parallel and Distributed Systems (ICPADS). Piscataway: IEEE, 2020: 466-475.

[25] YU J D, CHEN H P, HU F. SASM: Improving spark performance with Adaptive Skew Mitigation[C]//2015 IEEE International Conference on Progress in Informatics and Computing (PIC). Piscataway: IEEE, 2015: 102-107.

[26] HUANG Z C, WEI W G, XIE G Y. Load balancing mechanism based on linear regression partition prediction in spark[J]. Journal of Physics: Conference Series, 2020, 1575(1): 012109.

[27] WANG S Z, JIA Z T, WANG W L. Research on optimization of data balancing partition algorithm based on spark platform[M]//Lecture Notes in Computer Science. Cham: Springer International Publishing, 2021: 3-13.

[28] SONG A B, PENG B W, QIU J Y, et al. BSDP: A novel balanced spark data partitioner[C]//2021 IEEE 27th International Conference on Parallel and Distributed Systems (ICPADS). Piscataway: IEEE, 2021: 556-566.

[29] GUO W X, HUANG C J, TIAN W H. Handling data skew at reduce stage in Spark by ReducePartition[J]. Concurrency and Computation: Practice and Experience, 2020, 32(9): e5637.

[30] ZVARA Z, SZABÓ P G N, LÓRÁNT B B, et al. System-aware dynamic partitioning for batch and streaming workloads[C]//Proceedings of the 14th IEEE/ACM International Conference on Utility and Cloud Computing (UCC). New York: ACM, 2021: 1-10.

[31] 阎逸飞, 王智立, 邱雪松, 等. Spark环境下基于数据倾斜模型的Shuffle分区优化方案[J]. 北京邮电大学学报, 2020, 43(2): 116-121.
YAN Y F, WANG Z L, QIU X S, et al. A shuffle partition optimization scheme based on data skew model in spark[J]. Journal of Beijing University of Posts and Telecommunications, 2020, 43(2): 116-121. (in Chinese)

[32] TANG Z, LV W, LI K L, et al. An intermediate data partition algorithm for skew mitigation in Spark computing environment[J]. IEEE Transactions on Cloud Computing, 2021, 9(2): 461-474.

[33] SHI X J, QIAN Y Q. An algorithm of data skew in spark based on partition[C]//2020 International Conference on Computers, Information Processing and Advanced Education (CIPAE). Piscataway: IEEE, 2020: 217-222.

[34] LI C L, CAI Q Q, LUO Y L. Data balancing-based intermediate data partitioning and check point-based cache recovery in Spark environment[J]. The Journal of Supercomputing, 2022, 78(3): 3561-3604.

[35] ZAHARIA M, CHOWDHURY M, DAS T, et al. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing[C]//Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI). Berkeley: USENIX Association, 2012: 15-28.

[36] 卞琛, 于炯, 英昌甜, 等. 并行计算框架Spark的自适应缓存管理策略[J]. 电子学报, 2017, 45(2): 278-284.
BIAN C, YU J, YING C T, et al. Self-adaptive strategy for cache management in spark[J]. Acta Electronica Sinica, 2017, 45(2): 278-284. (in Chinese)

[37] ILGEN B, KARAOGLAN B. Investigation of Zipf's 'law-of-meaning' on Turkish corpora[C]//2007 22nd international symposium on computer and information sciences (ISCIS). Piscataway: IEEE, 2007: 1-6.

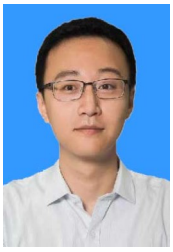
[38] LIN J. The curse of Zipf and limits to parallelization: A look at the stragglers problem in MapReduce[C]//Proceedings of the 7th Workshop on Large-Scale Distributed Systems for Information Retrieval (LSDS-IR), Boston: CEUR-WS, 2009: 57-62.

[39] AHMAD F M. Statistical universals of language: Mathematical chance vs. human choice[J]. Technometrics, 2022, 64(3): 432-433.

[40] United States Department of Transportation. Bureau Of Transportation Statistics (BTS) [DS/OL]. (2024-04-24) [2024-04-24]. https://www.transtats.bts.gov.

[41] 智慧来, 张丽, 李金海. 旁观者视角下粒的多层次描述[J]. 电子学报, 2022, 50(11): 2568-2574.
ZHI H L, ZHANG L, LI J H. Multi-level description of granules from an outsider's perspective[J]. Acta Electronica Sinica, 2022, 50(11): 2568-2574. (in Chinese)

作者简介



何玉林　男,1982年4月出生于河北衡水. 现为人工智能与数字经济广东省实验室(深圳)研究员、高级工程师. 主要研究方向为大数据系统计算技术、多样本统计分析理论与方法、数据挖掘与机器学习算法及应用. 中国电子学会会员编号:E190029878M.
E-mail: yulinhe@gml.ac.cn

吴东彤　女,1999年9月出生于广东汕头. 现为人工智能与数字经济广东省实验室(深圳)硕士研究生. 主要研究方向为大数据智能处理与分析技术.
E-mail: 2210273127@email.szu.edu.cn

Philippe Fournier-Viger　男,1980年8月出生于加拿大蒙特利尔. 现为深圳大学计算机与软件学院特聘教授、博士生导师. 主要研究方向为数据挖掘、人工智能、知识表示和推理、认知模型建构等.
E-mail: philfv@szu.edu.cn

黄哲学　男,1959年07月出生于黑龙江哈尔滨. 现为深圳大学计算机与软件学院特聘教授、博士生导师. 主要研究方向为数据挖掘、机器学习、大数据系统计算技术.
E-mail: zx.huang@szu.edu.cn