面向 MapReduce 的迭代式数据均衡分区策略

张元鸣 蒋建波 陆佳炜 徐 俊 肖 刚
（浙江工业大学计算机科学与技术学院 杭州 310023）

摘 要 MapReduce 是一种适用于大数据处理的重要并行计算框架. 然而，由于难以提前全面获得中间数据的分布规律，默认的数据分区策略往往会造成 Reducer 端的数据倾斜，会直接影响 MapReduce 的整体性能. 为了实现数据均衡分区，本文提出一种迭代式数据均衡分区策略，将每个 Mapper 节点要处理的数据块细分后以迭代方式循环处理，根据已迭代轮次的微分区分配结果决定当前迭代轮次的微分区分配方案，以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. 给出了迭代式数据分区策略的分配时机、分配准则、分配评价模型和分配算法. 基于公开的数据集，对迭代式数据均衡分区策略进行了详细测评，结果表明，该策略能够得到更均衡的数据分区结果，当数据集本身倾斜比较显著时，MapReduce 整体性能比默认分区策略平均提高了 11.1% 和 19.7%.

关键词 MapReduce；大数据；数据倾斜；迭代式数据分区；微分区；均衡分区
中图法分类号 TP311 **DOI 号** 10.11897/SP.J.1016.2019.01873

《计算机学报》

An Iterative Data Partitioning Strategy for MapReduce

ZHANG Yuan-Ming JIANG Jian-Bo LU Jia-Wei XU Jun XIAO Gang
(*College of Computer Science and Technology, Zhejiang University of Technology, Hangzhou* 310023)

Abstract MapReduce is an important parallel computing framework applied to big data. It can greatly improves data processing performance by running multiple tasks in parallel on a large number of cluster nodes. Intermediate data partitioning is a key problem that needs to be carefully handled in Shuffle phase, since the partitioning result will directly affect load balancing among Reducers. However, due to the difficulties of fully acquiring the distribution of intermediate data in advance, default data partitioning strategy will generally lead to data skew, which greatly limits the overall performance of MapReduce. This paper proposes an iterative data partitioning strategy for MapReduce. The data block processed in Map phase is further subdivided into fine-grained block (FGB) that will be iteratively processed. According to the partitioning results of previous micro-partitions (MPs), the partitioning plan of current MP is made. The data skew generated by previous MPs can be continuously eliminated, and balanced data partitioning can be finally achieved. In this strategy, data allocation begins after one FGB is completely processed by Mapper and all MPs contained in the FGB are generated. The condition to perform data allocation is that there are unallocated MPs in current iteration. The MPs with the same key will be allocated to the same Reducer in default. The MPs with larger factors are speculatively selected for data allocation in each iteration. The advantage of that is the MPs with larger factors contain more tuples and can be transmitted as early as possible. In addition, the Mapper also can work along with the data transmission in parallel. Vectors are defined to record profiling information of MP

收稿日期：2018-02-26；在线出版日期：2018-11-19. 本课题得到国家自然科学基金(61379017)、浙江省公益技术项目(2017C31014)和计算机体系结构国家重点实验室开发课题(CARCH201804)资助. **张元鸣**，博士，副教授，中国计算机学会(CCF)会员，主要研究方向为并行计算、云计算等. E-mail：zym@zjut.edu.cn. **蒋建波**，硕士研究生，主要研究方向为并行计算. **陆佳炜**，硕士，讲师，主要研究方向为服务计算. **徐 俊**，硕士，高级实验师，主要研究方向为服务计算. **肖 刚**(通信作者)，博士，教授，主要研究领域为云计算、智能信息系统等. E-mail：xg@zjut.edu.cn.

in every iteration and global profiling information of allocated MPs in all nodes. Furthermore, two algorithms are implemented to allocate the selected MPs to several Segments in current iteration based on existing partitioning results of previous MPs. These algorithms can make data balanced as well as possible after each iteration when future data distribution is unknown. With several public data sets of Zipf, BTS and Konect, We have evaluated the proposed strategy on a 16-node cluster considering four evaluation indicators: the maximum data amount, the standard data deviation, the longest execution time and the overall execution time. Experimental results show that our strategy can achieve much higher balanced data partitioning results than two existing data partition strategies. The average performance of MapReduce is improved to 11.1% and 19.7% for those data sets with obvious data skew.

Keywords MapReduce; big data; data skew; iterative data partitioning; micro-partition; balanced partitioning

1 引 言

MapReduce 是 Google 公司在 2004 年提出的一种适用于大数据处理的并行计算框架[1],通过在大量廉价的机群节点同时运行多个任务并行地处理海量数据,提高了大数据的处理能力,被广泛应用于数据挖掘、风险分析、客户行为分析等[2]场合. Spark① 是一个基于 MapReduce 并行计算框架的开源实现,由美国加州大学伯克利分校 AMPLab 实验室开发,基于内存计算对数据读写过程进行优化,具有更小的 IO 开销,比 Apache 的开源实现 Hadoop 具有更好的性能优势.

数据分区是 MapReduce 框架在 Shuffle 阶段需要解决的一个关键问题,其分区结果影响到各个 Reducer 接收数据的平衡性. Spark 系统默认采用的 Hash 分区方法,以及 Range 和自定义分区方法,都属于一次性分区方法,使分区与 Reducer 建立对应关系. 当数据集生成的中间数据元组分布均匀时,默认的一次性分区方法能够实现各个 Reducer 均衡地接收数据. 然而,已有研究表明[3-6]:中间数据元组不均衡是大规模数据集的一个普遍现象,当采用一次性分区方法时,在 92%的任务中出现了 Reducer 接收数据的不均衡,使整体性能降低了 22%到 38%. 由此可见,数据分区对 MapReduce 框架整体性能有重要影响,如何对中间数据进行均衡划分是近几年 MapReduce 并行计算框架研究的热点.

针对 MapReduce 默认分区方法所导致的数据不均衡性问题,研究者们提出了两阶段分区[7-9]、多阶段分区[10-11]、采样分区[12-24]、延迟分区[25-26]、迁移分区[27-29]等方法. 然而,这些方法在分区的时机、分区的适应性、额外开销等方面均存在一定的局限性. 针对这些问题,本文结合数据分区中潜在的迭代特征提出一种迭代式数据分区策略,以逐步寻优的方式实现数据均衡分区,最大化数据均衡性. 本文的主要贡献包括:

(1) 提出一种新的迭代式数据均衡分区方法,在分区过程中以迭代的方式逐步将数据进行分区,使各个 Reducer 能够均衡地接收数据.

(2) 给出了迭代式数据均衡分区策略的实现机制,包括分区时机、分区准则、评价模型和分区实现算法.

(3) 通过实验详细评价了迭代式数据均衡分区策略,证明了该策略能够实现数据均衡分区并能够提高 MapReduce 的整体性能.

本文第 2 节主要介绍 MapReduce 数据分区相关工作;第 3 节介绍 MapReduce 的默认数据分区策略和迭代式数据分区策略;第 4 节给出迭代式数据分区的实现机制;第 5 节对迭代式数据分区进行详细评价;第 6 节对本文进行总结.

2 相关工作

如何将 Mapper 端所产生的中间数据均衡地划分到各个 Reducer 端,是 MapReduce 并行计算框架面临的一个关键问题. 最早的数据划分方法是基于 Hash 划分方法[1],它以简单的方式将数据分配到各个 Reducer 上. 由于该方法的简单和通用性,在 Spark

① http://spark.apache.org/

《计算机学报》

平台上被采用为默认的分区策略，但这种一次性划分方法难以实现数据均衡分配，显著影响了平台的整体性能.

围绕数据分区的均衡性问题，研究者们提出了多种策略和方法，可以归纳为以下几类：

(1) 两阶段数据分区. 为了避免一次性分区带来的弊端，Gufler 等人[7-8]提出了两阶段数据分区策略，其先按照一次性分区策略生成数据分区，并在运行的过程中分析数据的分区情况，如果发生倾斜，则对数据量较大的分区进行拆分，并将拆分的数据调整到数据量较小的分区，从而实现数据均衡分区；高宇飞等人[9]提出一种基于 Hash 虚拟平衡重分区的数据倾斜处理算法 HVBR-SH，该算法在 Map 阶段采用虚拟分区，元组分散存储，然后在 Reduce 阶段对连续虚拟分区进行重组分配. 该策略的不足之处在于其有效性取决于调整分区的时机，过早拆分数据量较大的分区会增加误拆分的可能性，而过晚拆分数据量较大的分区会造成数据延迟传输.

(2) 数据多阶段分区. 该策略多次将分区分配到 Reducer 端，如王卓等[10]提出一种增量式分区策略，该策略通过在 Map 阶段生成细粒度分区，然后在运行的过程中通过定义的代价模型评估分区的平衡性，当满足某一条件时将选择的细粒度分区分配到 Reducer 上，经过多阶段的筛选和分配，使数据分配达到一定的平衡；卞琛等人[11]提出了一种迭代式填充的分区映射方法，将 Spark 中间数据划分为原生区和扩展区，扩展区的数据在特定时机下分多次进行分配. 然而，多阶段分区策略所依赖的分区时机难以把握，如增量式分区策略中规定最后一次的分区时间是在任务完成 80% 的时候，这不具有通用性，而且该策略也会造成部分数据的延迟传输，并且也未考虑数据延迟对性能的影响.

(3) 数据采样分区. 为了避免盲目数据分区导致的数据倾斜，人们提出了基于数据采样的分区策略，该策略通过采样进程分析数据的分布规律，再根据采样结果制定分区策略. Tang 等人[12]提出了一种 SCID 算法，通过采样并预测 cluster 大小，根据预测结果分割填充到 Spark 的 Bucket 上；Liu 等人[13]实现了一种面向 Spark 的 SP-partition 分区策略，通过采样计算得到 Bucket 的平均大小，将数据分配给可以容纳它们的 Bucket；Ramakrishnan 等人[14]提出一种将采样与数据拆分相结合的方法，在数据处理执行过程中增加一个额外的进程负责分析数据分布情况，当数据处理到一定比例后，根据采样进程的分析结果对数据进行拆分和合并，即将数据量大的分区拆分，并与数据量较小的分区合并；为了提高数据拆分与合并的效率，Kolb 等人[15-16]提出一种比元组粒度大的基于数据块的采样和拆分方法，该方法将元组集合合并为一个数据块，然后以数据块为单位进行分析、拆分和合并；Ibrahim 等人[17]结合数据本地特征提出了 LEEN 分区算法，该算法结合了数据统计情况和数据的本地特征，尽可能地将数据分区到最合适的节点上；Chen 等人[18]提出了一种轻量级数据平衡算法 LIBRA，该算法通过采集运行过程中的 Map 节点的中间数据分布情况，并结合 Reducer 端的硬件资源来制定数据均衡分配策略；韩蕾等人[19]专门对数据采样方法进行了深入研究，包括采样效果、采样规模以及节点个数之间的关系，采样效果与数据倾斜、数据集大小之间的关系，其目的是实现以较少的采样代价取得尽可能好的数据均衡；另外一些研究还包括频次分类连接算法[20]、多维度分区 MDRP 方法[21]、组合分区[22]、基于权重计算的负载平衡[23]、内存数据溢写统计方法[24]等. 该策略需要花费额外的时间开销，如启动采样进程，收集数据分布规律，这将增加数据访问和数据传输开销，而且，数据采样还存在某种不确定性，如果采样过少会导致准确性不足，采样过多又将增加时间开销.

(4) 数据延迟分区. 该策略[25]的思想是通过定义代价模型评估数据分区的大小，然后根据代价模型评估产生数据的大小，并在任务运行到某个时刻时启动数据分区. 例如，万聪等[26]提出了 RBPA 算法，将 Map 阶段产生的中间数据划分为更多的小分区，每次仅分配 Reducer 一个小分区，待 Reducer 完成所分配的小分区之后，再继续分配下一个小分区，直到所有的小分区都被分配完毕，实现了动态调节 Reducer 的负载. 但是，该方法要求数据分区完成之后才能进行数据传输，将会延迟数据传输时间，造成数据在 Mapper 端等待，使数据处理与数据传输不能够重叠进行.

(5) 数据迁移分区. 该策略的思想是允许出现数据倾斜，但在出现数据倾斜后通过数据迁移实现数据均衡. Yu 等[27]提出一种 SASM 策略，当新任务被注册时，将未处理完的任务拆分到其他空闲节点；Kwon 等人[28]提出 SkewTune 均衡调度方法，通过数据的迁移完成节点负载的平衡，通过制定代价模型，对尚未完成的 Reducer 的任务进行剩余代价评估，并在满足一定的条件时将该节点尚未被处

理的数据迁移到已完成的任务节点上，从而实现各节点执行任务的均衡；Martha 等人[29]提出了一种 h-MapReduce 分区策略，通过定义代价模型评价各个节点的负载情况，将负载过大的任务拆分为若干个新的子任务，并将其迁移到子节点上并行运行. 该策略的局限性在于需要增加额外的数据传输代价，在一定程度上延迟了作业完成的时间.

3 MapReduce 数据分区策略

3.1 默认数据分区策略

Shuffle 是连接 Mapper 和 Reducer 之间的桥梁，其任务是将 Mapper 端读取的原始数据经过处理后，将中间数据以某种分区方法分配给各个 Reducer 端并行处理. 如果分配到各个 Reducer 的数据是不均衡的，即产生了数据倾斜，将造成 Reducer 端负载的不平衡，影响整个作业的执行时间.

以 MapReduce 并行计算框架的开源实现 Spark 为例说明默认分区的实现机制. 由于 Spark 中定义的窄依赖转换操作可以看作是 Map 阶段的操作，宽依赖操作可以看作 Reduce 阶段前的 Shuffle 分区操作，因此本文以 Map 阶段和 Reduce 阶段对分区过程进行讨论. 图 1 给出了 Spark 在 Shuffle 阶段的基本原理，其采用基于排序的 Shuffle 操作机制，目的是能够在集群上处理大规模数据集，缓解 Shuffle 过程中产生过多的文件和缓存开销过大的问题.

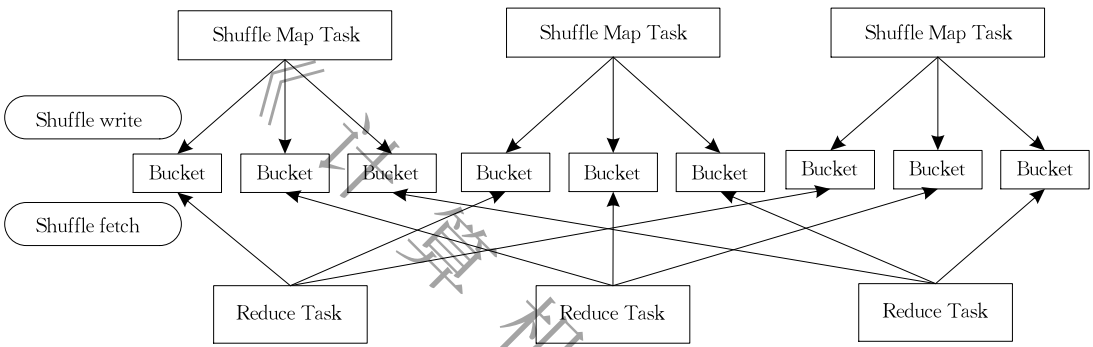


图 1 Spark 的 Shuffle 基本原理

在 Shuffle Write 阶段，每一个 Mapper 任务会创建出一个 Bucket 文件，然后将文件分为若干个段 Segment，Segment 的数量与 Reducer 任务的数量相同；Mapper 根据默认的 Hash 分区算法或自定义的分区算法将数据进行分区，根据需要进行聚合和排序，再将分区结果写入到各个 Segment 中；为了访问方便，对应生成 Index 索引文件，通过索引读取 Reducer 所依赖的 Segment.

在 Shuffle Read 阶段，每一个 Reducer 将根据各自的任务标识从它们所依赖的 Bucket 的相应 Segment 中拉取数据，然后进行 Key 的聚合或连接等操作，数据读取和聚合操作是并行进行的，每次都只能读取一定大小的数据，然后进行聚合等操作，直到所有的数据都从 Mapper 端读取和处理完毕.

3.2 迭代式数据分区策略

由于难以完全掌握大规模数据集的分布情况，默认分区策略往往会造成不均衡的分区结果. 为此，本文提出一种迭代式数据分区策略，图 2 给出了 4 个 Mapper 任务的迭代式数据分区过程，在该图中，Mapper 任务将 Block 处理得到的元组分给 4 个 Reducer 节点.

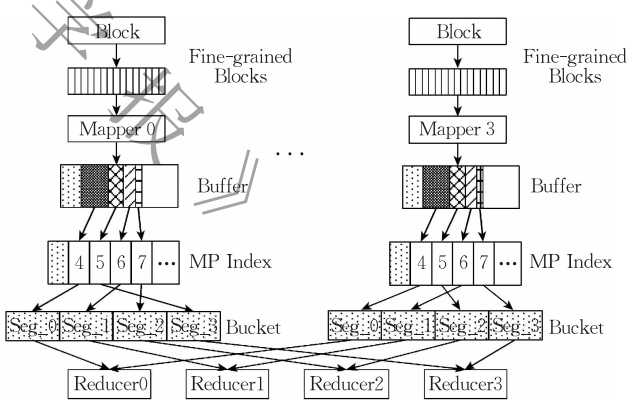


图 2 Mapper 任务的迭代式数据分区过程

首先将 Block 进行细分，得到细粒度数据块(Fine-grained Blocks，FGB)，它是 Mapper 任务处理的基本单位，以该 FGB 中各类 Key 的元组数量作为分区的主要依据，可以避免只根据 Key 值而不考虑元组数据量的一次性分区带来的缺陷. 同时，将 Block 细分为多个 FGB，使已经分区的元组可以更早地启动 Shuffle 操作和数据传输操作.

Mapper 任务以迭代的方式对 FGB 进行逐个处理，将每次迭代处理得到的元组集合存储到 Buffer 中，将 Key 值相同的元组集合在逻辑上作为本次迭

代分区的基本单位，称为微分区（Micro-Partition，MP）；同时创建微分区索引和微分区因子，分别用于标识该微分区将要被分配的位置，以及表示该微分区中的元组数量，微分区因子可以通过统计 Key 值相同的元组数量得到，是数据分区的主要依据.

为每个 Mapper 任务创建一个 Bucket，根据 Reducer 任务数量划分为若干个 Segments，Reducer 与 Segment 具有一一对应关系，即每个 Reducer 只能从依赖的 Segment 中读取数据；每个 Segment 包含多个槽 Slots，每个 Slot 存储 Key 值相同的 MP.

迭代分区的基本过程是：从当前迭代的 FGB 中按某种策略选择若干个 MP 分配给各个 Segment，目标是确保本轮次 MP 分配之后各个 Segment 数据均衡最大化，未被分配的 MP 在下次迭代时与新生成的 MP 再一起参与数据分配.

图 3 给出了前两轮迭代分配的示意图. 第一次迭代分配时，选择一定数目（图中选择的微分区数量与 Segment 数量相同）的 MP 进行分配，剩余的 MP 在后面的迭代中再进行分配；第二次迭代分配时，将新产生的 MP 与之前尚未分配的 MP 的元组数量进行汇总，选择一定数目的 MP 再次进行分配，其目标是确保本次迭代分区之后各个 Segment 数据均衡最大化. 之后的每轮迭代如同第二轮，通过再次分配新的 MP 使得已经分配的数据总体上处于平衡，从而以迭代的方式不断调整可能出现的数据倾斜.

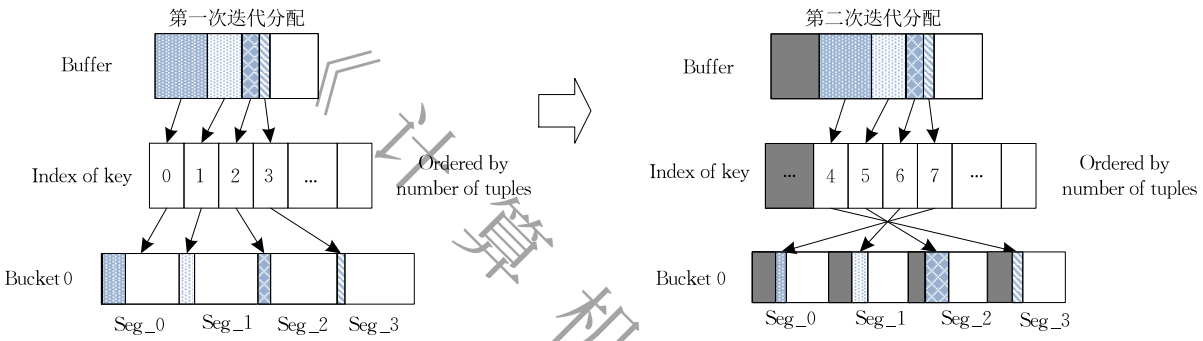


图 3　第一次迭代和第二次迭代的数据分配过程

假设迭代的轮次为 n，则第 i 次迭代需要综合考虑前面 $i-1$ 次的分区结果进行分区，使得第 i 次迭代之后数据尽可能平衡，n 次迭代结束后没有迭代的 MP 则按照一次性分区方法进行分区.

在上述机制中，每次迭代得到的 MP 在分区之前，需要将数据先送入 Buffer，这会引入一定的内存开销. 假设 FGB 的大小为 m 字节，迭代 n 次，则最坏情况下占用的内存约为 $m\times n$ 字节. 然而，在分区确定之后 MP 将被传输给 Reducer 端，并释放其所占用的内存空间，因此实际占用的内存空间要远小于上述的估算结果. 此外，未分配的 MP 与新产生的 MP 不会进行合并，仅重新统计相同 Key 的元组数量，由此产生的内存碎片可以通过逐步合并邻近的内存空间而进行重新使用.

迭代式分区策略不改变 Spark 默认的弹性分布式数据集 RDD 的依赖关系，通过将每个 Block 进一步划分为若干细粒度数据块，并构建出细粒度的 RDD，这些细粒度的 RDD 在进行 Shuffle 分区操作时，以迭代的方式进行处理，基于不同类型的 Key 的数量实现均衡分区，使得 Shuffle 分区之后得到的下一个 RDD 上各个分区的元组更加均衡.

迭代式分区策略与增量式分区策略[10]的不同点主要表现在：(1) 迭代式分区策略直接根据各类 Key 的元组数量以迭代的方式制定分区方案，分配的对象仅包含一种 Key，而增量式分区策略则先根据 Key 通过 Hash 函数分配到相应的子分区，其中包含多种 Key，再根据子分区中包含的元组数量分阶段制定分区方案；(2) 迭代式分区策略的分配时机是在 FGB 处理完成之后，以迭代的方式逐步进行分配，增量式分区策略的分配时机是根据特定的时间间隔启动，如第一次分配是在第一个 Mapper 任务结束之后，最后一次分配是在所有的 Mapper 任务完成 80%之后.

4　迭代式数据分区实现机制

迭代式数据分区策略在 Mapper 阶段对细分得到的 FGB 进行迭代处理，合并 Key 值相同的元组得到微分区 MP，并将选择的 MP 分配到各个节点上，以迭代的方式逐步调整已分配数据导致的数据倾斜. 为了实现该分区策略，需要解决的关键问题包括：(1) 制定数据分配的时机；(2) 确定每轮迭代分

《计算机学报》

配的微分区数量;以及(3)制定优化的分配算法使本轮分配后数据平衡最优化.

4.1 迭代式数据分区的时机

数据分区的时机是指执行数据分区的时间和条件.在迭代式数据分区策略中,将数据分配的时机定于每次 FGB 迭代结束时,即 Mapper 对每个 FGB 处理完成,创建得到微分区索引和微分区因子之后.此外,还需要满足一个必要条件,即还存在尚未分配的 MP.相同 Key 的微分区将被分配到同一个 Segment 上,如果本次迭代包含的微分区已经被分配,则不会被再次分配.这样,在迭代的过程中如果没有产生新的 Key,则不需要启动分区.

数据分区的数量是指每次执行数据分区时所分配的 MP 数量.记每轮选择的 MP 数量为 p,该值与 Segment 的数量 s 以及可供选择的 MP 的数量 n 相关.一种比较简单的方法是:选择的 MP 数量 p 等于 Segment 的数量 s,也就是 Reducer 数量.最后再将没有迭代分配的 MP 按照一次性分区的方法进行分区.

另外,如果某一个微分区因子远大于其他微分区因子,则将不对其进行拆分处理,这符合 MapReduce 框架对同一 Key 值的所有元组作为一个整体进行处理的要求.解决该极端情况的一个途径是通过另选属性作为 Key 值进行分区处理.

4.2 迭代式数据分区的准则

数据分区的准则是指从尚未分配的 MP 中选择分配的标准.本文选择 MP 的准则是优先选择微分区因子大的微分区.该准则的优势在于能够对包含元组数量较多的 MP 尽快进行分配,从而尽早启动数据传输,并让元组数量较少的 MP 稍候传输,使得 Mapper 任务和数据传输能够重叠执行,且有助于减少缓存占用的存储空间.

该准则的另外一个特点是实现简单,其先根据元组因子对尚未分配的 MP 进行排序,然后选择前面的 p 个 MP,一旦 MP 被分配,则后续迭代轮次将无法更改.结合图 3 进一步说明前面两轮迭代分区的过程.第一次迭代分区时,先按照微分区因子对 Buffer 中的微分区队列进行排序,设排在最前面的 4 个 MP 的索引为 $Index_0 \sim Index_3$,那么将这 4 个 MP 分配到 4 个 Bucket 中的 Segments 中.第二次迭代分区时,已分配的 MP 将不再参与分配,重新统计微分区因子,并对微分区队列重新进行排序,设排在最前面的索引分别是 $Index_4 \sim Index_7$,那么本迭代轮次的分区任务是根据各个 Segment 上的已有的分区结果,将 $Index4 \sim Index7$ 所指向的 MP 分配到各个 Segment 的 Slot 上,如图 3 所示,这 4 个 MP 被分配到 4 个 Segment 的对应 Slot 上.

4.3 迭代式数据分区的评估模型

此外,还需要记录每次迭代的所有 MP 的元组数量,以及所有 Bucket 的 Segment 中已分配的全局的元组分区情况.

在微分区索引与微分区因子之间建立关联关系,创建一个记录微分区索引及微分区索引因子的微分区向量:

$$\boldsymbol{\alpha}=(t_1,t_2,t_3,\cdots,t_n),$$

其中 $t_i=(index,factor,seg_no,slot_no)(i\in[1,n])$,$index$ 为微分区索引,$factor$ 为微分区因子,seg_no 为 Segment 的编号,$slot_no$ 为 slot 的槽号.通过该向量记录该 Mapper 当前迭代的 MP 情况,如果 seg_no 与 $slot_no$ 均为 -1,则表示 MP 尚未分配.

当有 m 个 Mapper 同时运行时,将每个 Mapper 上的 $\boldsymbol{\alpha}$ 中对应的分量进行累加,得到全局的数据分布情况,用二维向量 $\boldsymbol{AM}$ 表示:

$$\boldsymbol{AM}=\sum_{k=1}^{m}\boldsymbol{\alpha}_k(k\in[1,m]).$$

Mapper 每次迭代结束时,将微分区向量 $\boldsymbol{\alpha}$ 发送给 Master,然后将 $\boldsymbol{\alpha}$ 累加到二维向量 $\boldsymbol{AM}$ 中,即可得到截至当前迭代轮次全局的元组分配情况.

由于 Reducer 与 Bucket 的 Segment 是一对一的,因此各个 Reducer 上分配的元组数量可以通过统计 Bucket 的 Segment 上已分配的元组数量计算得到,定义一个 Reducer 任务元组分配向量 $\boldsymbol{AB}$:

$$\boldsymbol{AB}=(\boldsymbol{R}_1,\boldsymbol{R}_2,\boldsymbol{R}_3,\cdots,\boldsymbol{R}_r),$$

其中 r 是与 Bucket 的数量相等的 Reducer 任务数,$\boldsymbol{R}_b=(f_1,f_2,f_3,\cdots,f_n)(b\in[1,r])$,$\boldsymbol{R}_b$ 向量用于记录各个 Bucket 上给该 Reducer 任务分配的元组数量,其中的 f_1 表示第一个 Bucket 的 Segment 上已分配的元组数量.

向量 $\boldsymbol{AB}$ 的每一行表示 Reducer 任务在不同的 Bucket 上分配的元组数量,每一列表示在同一个 Bucket 上给各个 Segment 上分配的元组数量.通过累加向量 $\boldsymbol{AB}$ 上每一行各个 Segment 的已被分配的元组总数即可得到每个 Reducer 任务分配的元组数量.

4.4 迭代式数据分区的优化算法

迭代式数据分区优化算法解决的问题是:在未来元组数据及分布情况未知的情况下,根据已经分配的元组数量,将当前迭代所选择的 p 个微分区分配给 b 个 Segment,并使分配后的数据尽可能平衡.

《计算机学报》

算法 1 是实现的微分区选择算法，是一种基于时间复杂度低的堆排序算法，它的时间复杂度为 $O(n\log n)$，这与未分配微分区的数量相关，作业刚开始执行时，处理花费代价低，当未分配的微分区数量增大，处理花费的代价也逐渐增加. 实际处理时，没有必要对所有的微分区进行完全排序，只需要将最大的前 p 个微分区排序筛选出来即可.

算法 1.　微分区选择算法.

输入：全局数据分布向量 ***AM***

输出：本轮需要分配的微分区编号集合 C

```
FOR i in AM
    IF i.index not in Dic
      X.add(i) //筛选出已分配的微分区
    ENDIF
ENDFOR
Sort(W) //根据微分区因子排序
C←W[1:p] //从 W 中取出前 p 个最大的微分区
```

算法 2 是微分区分配优化算法，该算法基于贪心策略选择元组数据量最小的 Bucket 作为分配对象，使得各个 Bucket 上的数据量趋于均等. 获取 Segment 上总体的元组数量并排序，然后将选择的 p 个 MP 分配到 Segments 上，采用冒泡排序，时间花费代价低.

算法 2.　微分区分配算法.

输入：数据分配向量 ***AB***，全局数据分布向量 ***AM***

输出：数据分配向量 ***AB***

```
FOR i in AM
    IF i.index in Dic
      Y.add(i) //筛选出已分配的微分区
      Seg[Dic[i.index]][slot++]=i
    ENDIF
ENDFOR
C←Y //选择本轮迭代分配的微分区，即算法 1
S←Count(Seg) //计算各个 Bucket 上的数量
FOR i in C //将所有的 C 分配到 S 中
    a←Sort(S) //选择数据量最小的 Bucket 的编号
    Seg[a][slot++]=C[i]
    Dic.add(C[i],a)
    S[a]+=C[i].num //更新微分区 S[a]的数据量
ENDFOR
AB←Seg //更新 AB
```

迭代式数据分区策略只对 Shuffle 阶段的分区过程进行了改进，通过对细粒度数据块的迭代处理，逐步根据 Key 的元组数量进行分区，在分区的过程中，如果出现内存不足的情况，将会开启内存溢写，尽管会牺牲一定的性能，但是由于没有改变 Spark 默认的读写机制，不会影响其容错性.

5　实验结果与分析

实验环境由 16 个机群节点组成，每个节点是一个 4 核处理器，主频为 2.5 GHz、16 GB 内存、40 GB 系统盘和 40 GB 数据盘，操作系统是 64 位 Ubuntu 16.04，Spark 版本是 2.1.1.

实验数据是公开的数据集，包括 Zipf 数据集[30]、BTS 数据集和 Konect 数据集，如表 1. Zipf 数据集为合成数据，主要包括 10 个子数据集，每个数据集都满足指数为 α 的标准 Zipf 分布，在有 N 个元素的 Zipf 分布中，数量排名第 k 位的元组出现频率 f 为

$$f=\frac{k^{-\alpha}}{\sum_{i=1}^{N} i^{-\alpha}}.$$

参数 α 表示数据的倾斜程度，α 越大则数据集的分布就越倾斜，α 的取值范围为[0.2，1.2]，增量为 0.2，数据的取值范围为[1，1000]，实验过程中通过调整 α 改变数据倾斜程度.

表 1　实验数据集统计信息

数据集	运行实例	大小/GB	元组数/亿	描述
Zipf	WordCount	2.20	1.20	标准 Zipf 分布
BTS	WordCount	11.10	1.88	US 航空数据
Konect①	PageRank	1.26	1.90	网络数据集
Konect	InvertIndex	1.26	1.90	网络数据集

BTS 数据集是真实的航天航空数据，一共包含约 500 个文件，平均每个文件包含记录数为 20 万条. Konect 数据集包含了各种类型的网络大数据，本文选取了其中的维基百科链接数据集.

实验通过以下 4 个指标来评价迭代式数据分区：(1) 节点最大数据量. 该指标衡量分区之后分配到元组数量最大的 Reducer 节点，值越大说明该节点分配的数据量也越大；(2) 分区标准方差. 该指标衡量各个 Reducer 分区之后数据的均衡情况，值越大说明数据分布越不均衡；(3) 节点最长执行时间. 该指标衡量分配数据量最大的 Reducer 节点的执行时间；(4) 作业整体执行时间. 该指标衡量不同分区情况下作业总的执行时间.

5.1　迭代式数据分区性能的影响因素

本文先通过实验来评价迭代轮数、微分区数和 FGB 大小 3 个因素对迭代式数据分区的影响. 实验

① http://konect.uni-koblenz.de/networks/wikipedia_link_en

《计算机学报》

的数据集为 Zipf 数据集，倾斜指数 α 为 0.6，数据量大小为 450 MB，包含 1.2 亿个元组，Key 值的取值范围为[1,10000]，Key 的数量为 330 个.

5.1.1 迭代轮数对性能的影响

迭代轮数是指迭代分区的总轮数，如前所述，如果迭代轮数为 1，则仅将第一个 FGB 按照迭代式数据分配策略进行分配，之后的所有数据按照一次性分区策略进行直接分区；如果迭代轮次为 2，则对前面 2 个 FGB 按照迭代式数据分配策略进行分配，剩余的数据按照一次性分区策略进行分区. 迭代轮数一方面影响分区之后数据的均衡性，另一方面由于迭代时产生的开销也对作业整体执行时间产生一定影响. 实验中将 FGB 大小设为 3.5 MB，迭代的 MP 数据量为 32，Reducer 节点数为 60，实验结果如图 4 所示.

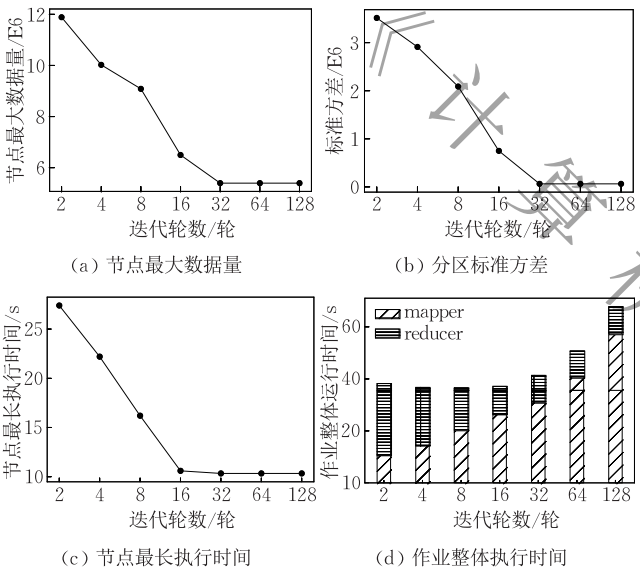


图 4 迭代轮数对 4 个评价指标的影响结果

图 4(a)给出了不同迭代轮次的节点最大数据量. 实验结果表明，随着迭代轮数的增加，节点最大数据量逐步减小，主要原因是迭代轮数的增加使参与分配的 MP 数量增加，能够不断调整之前轮次的分配结果，避免将更多的 Key 分配到同一个 Reducer 节点，从而减少 Reducer 节点的最大数据量. 当迭代轮数大于 32 时，节点最大数据量趋于稳定.

图 4(b)给出了不同迭代轮次的分区标准方差. 实验结果表明，随着迭代轮数的增加，分区标准方差逐步减少，即分配到各个 Reducer 节点的元组数越来越平衡；另外，当迭代轮数大于 32 轮时，对数据均衡度影响逐渐变小，这表示当迭代轮数足够大时，基于之前的 MP 进行数据分配足以得到较好的分区结果.

图 4(c)给出了不同迭代轮次的节点最长执行时间. 实验结果表明，随着迭代轮数的增加，节点最长执行时间呈下降趋势，并且当迭代轮数大于 32 时，节点最长执行时间变化不再明显，这与图 4(a)和(b)的实验结果相一致.

图 4(d)给出了不同迭代轮次的作业整体执行时间. 实验结果表明，随着迭代轮数的增加，作业整体执行时间先略微减少然后再增加，原因在于 Map 阶段的执行时间在不断增加，而 Reduce 阶段的执行时间趋于平稳，即分区一方面减少了 Reducer 执行时间且在超过一定的迭代轮次后趋于稳定，另一方面又增加了 Mapper 执行时间，因此太多的迭代轮数并不利于减少作业整体执行时间.

5.1.2 迭代微分区数对性能的影响

迭代微分区数是指每次迭代选择的微分区数量，实验将 FGB 大小设为 3.5 MB，迭代轮数设为 8，实验结果如图 5 所示.

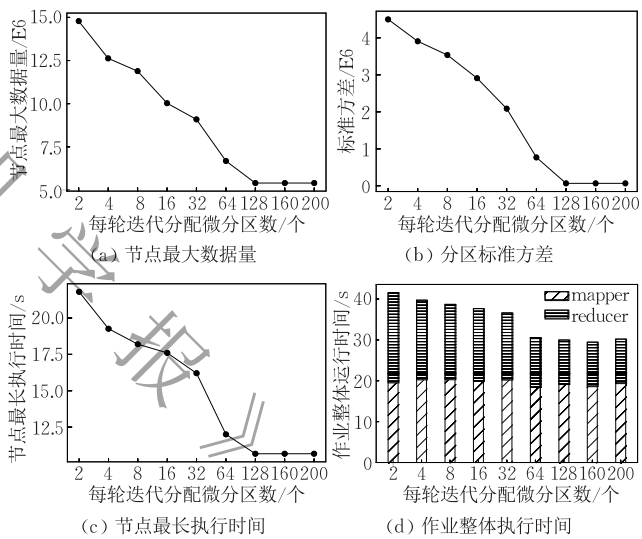


图 5 迭代微分区数量对 4 个评价指标的影响结果

图 5(a)给出了不同迭代微分区数的节点最大数据量. 实验结果表明，随着迭代微分区数的增加，节点最大数据量逐渐下降，主要原因是迭代微分区数越大，每次迭代参与统计的 Key 数量越多，越能够得到更好的分区结果；此外，当 MP 数量在 128 以上时，节点最大数据量逐渐变得稳定，表明当分配的微分区数足够大时，可以得到较好的数据分区结果.

图 5(b)给出了不同迭代微分区数的分区标准方差. 实验结果表明，随着迭代微分区数的增加，分区标准方差逐渐减少，主要原因也是分配更多的微分区能够得到更好的分区结果，且当迭代微分区数足够大时，如当 MP 数量为 128 时，各个 Reducer 上

《计算机学报》

所分配的数据量趋于稳定，方差变化不明显.

图 5(c)给出了不同迭代微分区数的节点最长执行时间. 实验结果表明，随着迭代微分区数的增加，节点最长执行时间逐步下降，即增加迭代微分区数能够减少节点的最长执行时间；当每轮 MP 数超过 128 时，节点最长执行时间波动变化小，这与图 5(a)和(b)的实验结果相一致.

图 5(d)给出了不同迭代微分区数的作业整体执行时间. 实验结果表明，Map 阶段的时间变化不大，Reduce 阶段时间逐步下降，作业整体执行时间总体呈现下降趋势.

5.1.3　FGB 大小对性能的影响

FGB 大小是指迭代处理的细粒度数据块大小，实验将迭代轮数设置为 8，将每轮迭代的微分区数设置为 64.

图 6 给出了 FGB 大小不同时的节点最大数据量、分区标准方差、节点最长执行时间和作业整体执行时间. 从分区结果的平衡性来看，随着 FGB 增大，节点最大数据量基本保持不变，分区标准方差变化波动较小，节点最长执行时间变化也较小，这表明 FGB 粒度对分区的结果影响不明显；从分区结果对作业整体执行时间看，如图 6(d)所示，随着 FGB 增大，作业整体执行时间呈下降趋势，其原因在于分区结果对 Reducer 任务的执行时间影响较小，而 Mapper 任务的执行时间却随着 FGB 增大而逐步减少，使得作业整体执行时间逐步减小.

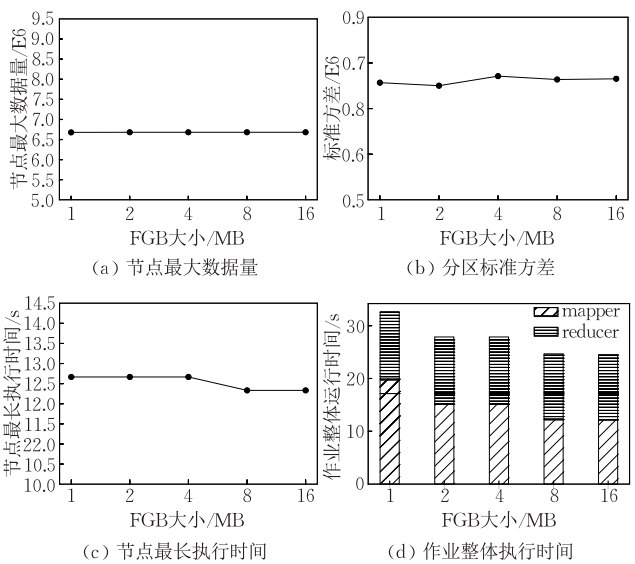


图 6　FGB 大小对 4 个评价指标的影响结果

5.2　迭代式数据均衡分区的性能比较

本节分别在 3 个公开的数据集 Zipf、BTS 和 Konect 上对默认分区策略、增量式分区策略[11]和本文的迭代式分区策略进行了对比分析，评价指标仍然是节点最大数据量、分区标准方差、节点最长执行时间和作业整体执行时间.

5.2.1　在 Zipf 数据集上的性能比较

Zipf 数据集的主要特点是可以通过设置数据倾斜指数 α 改变数据本身的倾斜程度，实验将 FGB 大小设置为 1MB，迭代轮数设置为 8，迭代的微分区数为 64，实验结果如图 7 所示.

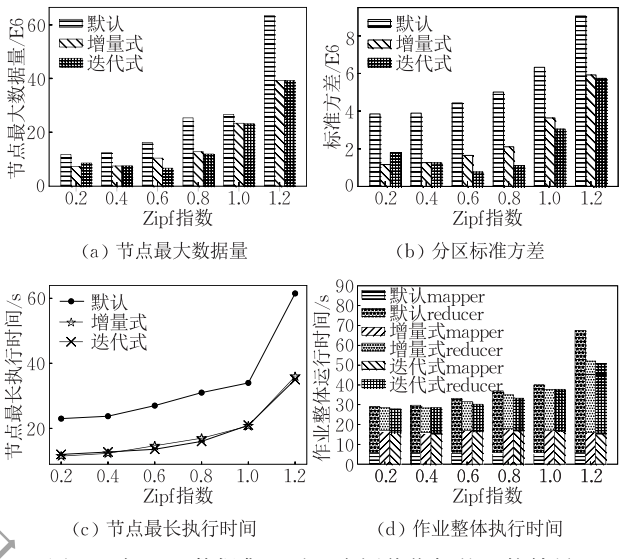


图 7　在 Zipf 数据集上对 4 个评价指标的比较结果

图 7(a)给出了 6 个不同数据倾斜指数 α 的 Zipf 数据集所对应的节点最大数据量. 实验结果表明，随着 α 从 0.2 增加到 1.2，默认分区策略的节点最大数据量逐渐增大；增量式分区策略和迭代分区策略在 α 从 0.2 增加到 0.8 时的节点最大数据量没有明显变化，但是当 α 超过 0.8 后开始呈现递增趋势，其主要原因在于原始数据变得更加倾斜，同一 Key 必须被分配到同一 Reducer 上进行处理. 迭代式分区策略的节点最大数据量比默认分区策略平均小 9.73×10^6 个，比增量式分区平均小 0.57×10^6 个.

图 7(b)给出了 6 个不同数据倾斜指数 α 的 Zipf 数据集的节点标准偏差. 实验结果表明，随着 α 的增大，默认分区策略的标准方差在逐步增加，分区结果变得更加倾斜；增量式分区策略和迭代式分区策略的标准方差也在增大，但其值均小于默认分区策略. 迭代式分区策略的标准方差比默认分区平均小 3.14×10^6，比增量式分区平均小 0.33×10^6.

图 7(c)给出了 6 个不同数据倾斜指数 α 的 Zipf 数据集的节点最长执行时间. 实验结果表明，随着 α 的增大，节点最长执行时间都相应增加，增量式分区策略和迭代式分区策略比默认分区策略增加的要

《计算机学报》

少.针对节点最长执行时间,迭代式分区策略的性能比默认分区策略的性能平均提高了 84.6%,比增量式分区策略的性能平均提高了 1.01%.

图 7(d)给出了 6 个不同数据倾斜指数 α 的 Zipf 数据集的作业整体执行时间.实验结果表明,迭代式分区策略的作业整体执行时间要小于默认分区策略和增量式分区策略.迭代式分区策略的作业整体执行性能比默认分区策略平均提高了 11.1%,比增量式分区策略的平均提高了 2.2%.

5.2.2 在 BTS 数据集上的性能比较

BTS 数据集是真实的航天航空数据,该数据集的 Key 及其元组个数的统计结果如图 8 所示,横坐标是 Key 值编号,纵坐标是每个 Key 值对应的元组数量,共有 3300 多个 Key 值,其中元组个数超过 2×10^6 的 Key 有 10 个.实验将 FGB 大小设置为 16MB,迭代轮数设置为 5,迭代的微分区数为 200,实验结果如图 9 所示.

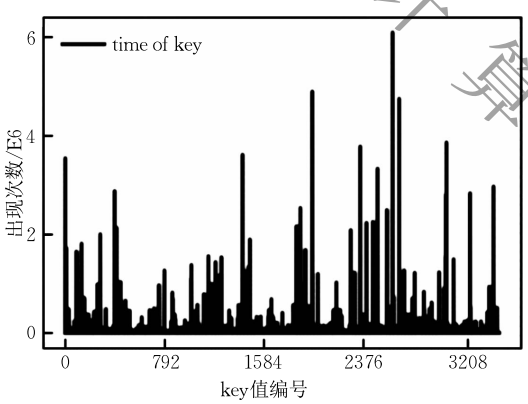
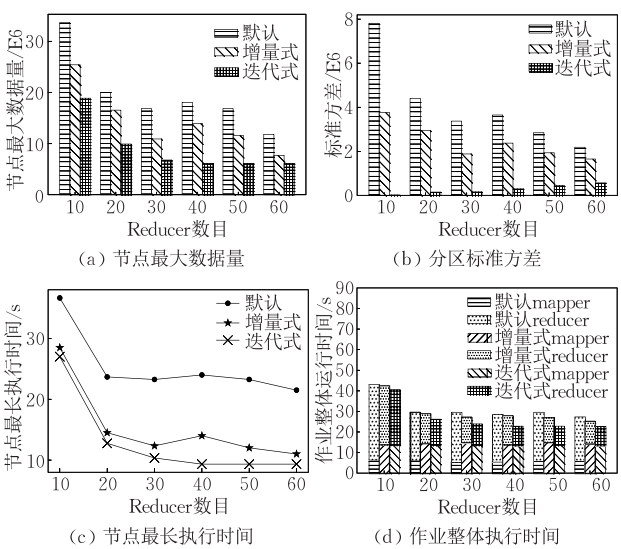


图 8 BTS 数据集的元组数量分布情况



(a) 节点最大数据量 (b) 分区标准方差

(c) 节点最长执行时间 (d) 作业整体执行时间

图 9 在 BTS 数据集对 4 个评价指标的比较结果

图 9(a)给出了 Reducer 数量不同时的节点最大数据量.实验结果表明,随着 Reducer 数量的增加,三种分区策略的节点最大数据量都在逐渐减少,但默认分区策略的节点最大数据量始终是最大的,迭代分区策略的节点最大数据量最小.迭代式分区策略的节点最大数据量比默认分区策略的平均小 10.5×10^6 个,比增量式分区策略的平均小 5.3×10^6 个.

图 9(b)给出了 Reducer 数量不同时的节点标准方差.实验结果表明,随着 Reducer 数量的增加,默认分区和增量式分区的标准方差呈下降趋势,迭代式分区的标准方差缓慢增长.迭代式分区策略的标准方差比默认分区平均小 3.7×10^6,比增量式分区平均小 2.1×10^6.

图 9(c)给出了 Reducer 数量不同时的节点最长执行时间.实验结果表明,随着 Reducer 数量的增加,三种策略的节点最长执行时间也都逐渐呈下降趋势;迭代式分区策略与增量式分区策略的节点最长执行时间均小于默认分区策略,其中迭代式分区策略的节点最长执行时间最短.针对节点最长执行时间,迭代式分区策略的性能比默认分区策略的性能平均提高了 114%,比增量式分区策略的性能平均提高了 22%.

图 9(d)给出了 Reducer 数量不同时的作业整体执行时间.实验结果表明,迭代式分区策略和增量式分区策略在 Mapper 阶段的执行时间要大于默认分区策略,但它们在 Reducer 阶段的执行时间要明显小于默认分区策略.迭代式分区策略的作业整体执行性能比默认分区策略提高了 19.7%,比增量式分区平均提高了 13.5%.

5.2.3 在 Konect 数据集上的性能比较

在 Konect 维基百科链接数据集上基于 PageRank 算法和 Inverted Index 算法对三种分区策略进一步分析和比较,该数据集的一个特点是 Key 元组分布相对比较均衡,PageRank 算法用来评价网页的重要性,Inverted Index 算法用于全文检索系统中的一种单词文档映射结构.实验将 FGB 大小设置为 2MB,迭代轮数设置为 5,迭代的微分区数为 200.

图 10 给出了在 Konect 数据集上基于 PageRank 算法对 4 个评价指标的比较结果,横坐标为 Reducer 数量,纵坐标为评价指标.从数据分区均衡性方面看,当 Reducer 数量变化时,默认分区策略的节点最大数据量、节点标准方差、节点最长执行时间 3 个指标波动较为明显,如当 Reducer 数量为 30 和 60 时,默认分区策略都产生了显著的数据不均衡;而增量

计算机学报

式分区策略和迭代式分区策略随着 Reducer 数量的增加 3 个指标都相应减少，表现了较为稳定的均衡分区结果. 对比 3 个分区策略，迭代式分区策略比默认分区策略和增量式数据策略都有较为显著的优势.

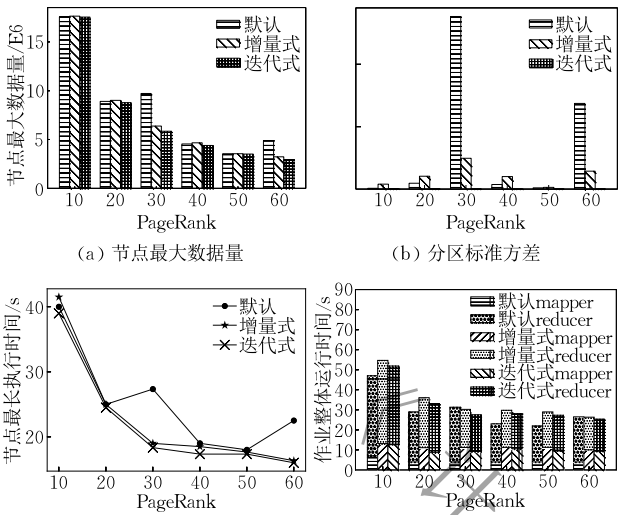


图 10　在 Konect 数据集上基于 PageRank 算法对 4 个评价指标的比较结果

从作业的整体执行时间看，如图 10(d)所示，当 Reducer 数目为 10、20、40、50 时，默认分区策略的整体执行时间最小，这是由于三种分区策略都没有产生明显的数据倾斜，而且默认分区策略的 Mapper 阶段执行时间最小；当 Reducer 数量为 30 和 60 时，增量式分区策略和迭代式分区策略的整体执行时间比默认分区策略的整体执行时间要小，其原因是增量式分区策略和迭代式分区策略没有产生数据倾斜，从而减少了 Reducer 阶段的执行时间.

图 11 给出了在 Konect 数据集上基于 Inverted Index 算法对 4 个评价指标的比较结果，横坐标为 Reducer 数量，纵坐标为评价指标. 从数据分区均衡性方面看，与图 10 的实验结果类似，当 Reducer 数量变化时，默认分区策略的节点最大数据量、节点标准方差、节点最长执行时间 3 个指标波动仍然较为明显；增量式分区策略和迭代式分区策略产生了较为稳定的均衡分区结果. 从作业整体执行时间看，如图 10(d)所示，默认分区策略的作业整体运行时间比增量式分区策略和迭代式分区策略的整体执行时间都要短，主要原因是该 Inverted 算法的 Reducer 的操作简单，数据不均衡对该阶段的执行时间影响不明显，而增量式分区策略和迭代式分区策略却增加了一定的 Mapper 执行时间，因此导致作业的整体执行时间相对较长.

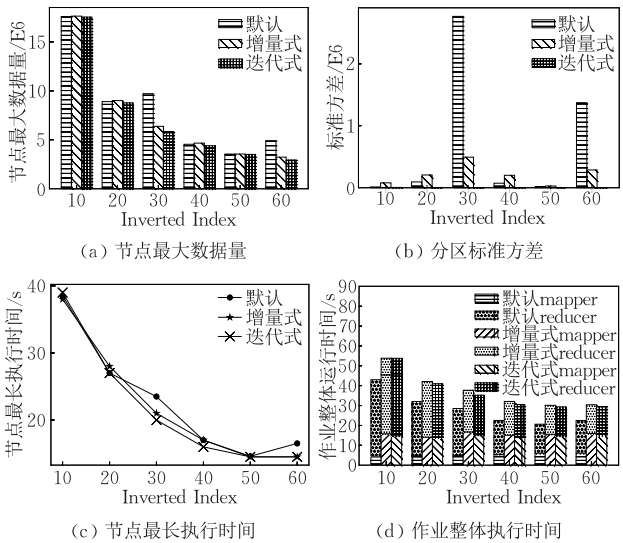


图 11　在 Konect 数据集上基于 Inverted Index 算法对 4 个评价指标的比较结果

6　总结与展望

数据不均衡分区对 MapReduce 并行计算框架的整体性能具有重要影响，针对该问题，本文提出了一种迭代式数据均衡分区策略，该策略先将每个 Mapper 节点要处理的数据块细分为细粒度数据块 FGB，以迭代方式循环处理 FGB，根据已迭代轮次的微分区分配结果制定当前迭代轮次的微分区分配方案，从而逐步调整已处理数据块的数据倾斜问题. 详细给出了迭代式数据分区策略的实现机制，并在三个公开的数据集对分区策略进行了评价，与默认的分区策略相比，不仅能够显著提高数据分区的均衡性，而且能够显著提高 MapReduce 的整体性能，因此，本策略能够为 MapReduce 中间数据分区提供一种更加均衡的数据分配方案. 特别地，如果数据集中存在某个 Key 的元组数量远远超过其他所有 Key 的元组数量的情况，本方法将不再适用.

参　考　文　献

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

[2] Wang Shan, Wang Hui-Ju, Qin Xiong-Pai, Zhou Xuan. Architecting big data: Challenges, studies and forecasts. Chinese Journal of Computers, 2011, 34(10): 1741-1752(in Chinese)

计算机学报

(王珊，王会举，覃雄派，周煊. 架构大数据：挑战、现状与展望. 计算机学报，2011，34(10)：1741-1752)

[3] Rasmussen A，Lam V T，Conley M，et al. Themis：An I/O-efficient MapReduce//Proceedings of the ACM Symposium on Cloud Computing. CA，USA，2012：1-14

[4] Ren K，Gibson G，Kwon Y C，et al. Hadoop's adolescence：A comparative workloads analysis from three research clusters//Proceedings of the High Performance Computing，Networking Storage & Analysis. Denver，USA，2013：1452-1452

[5] Lin Jimmy. The curse of Zipf and limits to parallelization：A look at the stragglers problem in MapReduce//Proceedings of the Workshop on Large-Scale Distributed Systems for Information Retrieval. Boston，USA，2012：57-62

[6] Kwon Y，Balazinska M，Howe B，et al. A study of skew in MapReduce applications. Open Cirrus Summit，2011：1-5

[7] Gufler B，Augsten N，Reiser A，Kemper A. Handing data skew in MapReduce//Proceedings of the International Conference on Cloud Computing and Services Science. Noordwijkerhout，Netherlands，2011，146：574-583

[8] Gufler B，Augsten N，Reiser A，Kemper A. Load balancing in MapReduce based on scalable cardinality estimates//Proceedings of the 2012 IEEE 28th International Conference on Data Engineering. Washington，USA，2012：522-533

[9] Gao Yu-Fei，Cao Yang-Jie，Tao Yong-Cai，Shi Lei. Handling data skew in MapReduce programming model based on virtual partitioning method. Journal of Chinese Computer Systems，2015，36(8)：1706-1710(in Chinese)

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

[10] Wang Zhuo，Chen Qun，Li Zhan-Huai，et al. An incremental partitioning strategy for data balance on MapReduce. Chinese Journal of Computers，2016，39(1)：19-35(in Chinese)

(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. 计算机学报，2016，39(1)：19-35)

[11] Bian Chen，Yu Jiong，Xiu Wei-Rong，et al. Partition and mapping algorithm for in-memory computing framework based on iterative filling. Journal of Computer Applications，2017，37(3)：647-653(in Chinese)

(卞琛，于炯，修位蓉等. 基于迭代渐进填充的内存计算框架分区映射算法. 计算机应用，2017，37(3)：647-653)

[12] Tang Z，Zhang X，Li K，et al. An intermediate data placement algorithm for load balancing in Spark computing environment. Future Generation Computer Systems，2016，78(1)：287-301

[13] Liu G，Zhu X，Wang J，et al. SP-Partitioner：A novel partition method to handle intermediate data skew in spark streaming. Future Generation Computer Systems，2017，86：1054-1063

[14] Ramakrishnan S R，Swart G，Urmanov A. Balancing reducer skew in MapReduce workloads using progressive counter//Proceedings of the ACM Symposium on Cloud Computing. New York，USA，2012：1-14

[15] Kolb L，Thor A. Block-based load balancing for entity resolution with MapReduce//Proceedings of the 20th ACM International Conference on Information and Knowledge Management. Glasgow，UK，2011：2397-2400

[16] Kolb L，Thor A，Rahm E. Load balancing for MapReduce-based entity resolution//Proceedings of the IEEE International Conference on Data Engineering. Washington，USA，2012：618-629

[17] Ibrahim S，Jin H，Lu L，et al. Handling partitioning skew in MapReduce using LEEN. Peer-to-Peer Networking and Applications，2013，6(4)：409-424

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

[19] Han Lei，Sun Xu-Zhan，Wu Zhi-Chuan，Chen Li-Jun. Optimization study on sample based partition on MapReduce. Journal of Computer Research and Development，2013，50(s2)：77-84(in Chinese)

(韩蕾，孙徐湛，吴志川，陈立军. MapReduce 上基于抽样的数据划分最优化研究. 计算机研究与发展，2013，50(s2)：77-84)

[20] Liang Jun-Jie，He Li-Min. Join algorithm in skewed datasets based on MapReduce. Computer Science，2016，43(9)：27-31(in Chinese)

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

[21] Myung J，Shim J，Yeon J，et al. Handling data skew in join algorithms using MapReduce. Expert Systems with Applications，2016，51：286-299

[22] Geng Yu-Jiao. The Research of Skew with Sampling Technique in MapReduce[M. S. dissertation]. Dalian Maritime University，Dalian，2013(in Chinese)

(耿玉娇. MapReduce 中基于抽样技术的倾斜问题研究[硕士学位论文]. 大连海事大学，大连，2013)

[23] Fu Jie，Du Zhi-Hui. Load balancing strategy on periodical MapReduce job. Computer Sciencs，2013，40(3)：38-40(in Chinese)

(傅杰，都志辉. 一种周期性 MapReduce 作业的负载均衡策略. 计算机科学，2013，40(3)：38-40)

[24] Li Hang-Chen，Qin Xiao-Lin，Shen Yao. Load balancing strategy on MapReduce with locality-aware. Computer Science，2015，42(10)：50-56(in Chinese)

(李航晨，秦小麟，沈尧. 数据本地性感知的 MapReduce 负载均衡策略. 计算机科学，2015，42(10)：50-56)

[25] Kwon Y，Balazinska M，Howe B，Rolia J. Skew-resistant parallel processing of feature-extracting scientific user-defined functions//Proceedings of the 1st ACM Symposium on Cloud Computing. Indianapolis，USA，2010：75-86

《计算机学报》

[26] Wan Cong, Wang Cui-Rong, Wang Cong, Jia Shuo. Research on reduce stage load balanced partition algorithm based on MapReduce. Journal of Chinese Computer Systems, 2015, 36(2): 240-243(in Chinese)
(万聪，王翠荣，王聪，贾朔. MapReduce 模型中 reduce 阶段负载均衡分区算法研究. 小型微型计算机系统，2015，36(2)：240-243)

[27] Yu J, Chen H, Hu F. SASM: Improving spark performance with adaptive skew mitigation//Proceedings of the IEEE International Conference on Progress in Informatics and Computing. Nanjing, China, 2016: 102-107

[28] Kwon Y, Balazinska M, Howe B, Rolia J. SkewTune: Mitigating skew in MapReduce applications//Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data. Scottsdale, USA, 2012: 25-36

[29] Martha V S, Zhao W, Xu X. h-MapReduce: A Framework for Workload Balancing in MapReduce//Proceedings of the IEEE International Conference on Advanced Information Networking and Applications. Barcelona, Spain, 2013: 637-644

[30] Wyllys R E. Empirical and theoretical bases of Zipf's law. Library Trends, 1981, 30: 53-64

ZHANG Yuan-Ming, Ph.D., associate professor. His main research interests include parallel computing and cloud computing.

JIANG Jian-Bo, M. S. candidate. His main research interest is parallel computing.

LU Jia-Wei, M. S., lecturer. His main research interest is services computing.

XU Jun, M. S., senior experimentalist. His main research interest is services computing.

XIAO Gang, Ph. D., professor. His main research interests include cloud computing and intelligent information system.

《计算机学报》

Background

MapReduce is an important parallel computing framework applied to big data. Intermediate data partitioning is a key problem that needs to be carefully handled in Shuffle phase, since the partitioning results will directly affect load balancing among Reducers. Default data partitioning approaches adopted in MapReduce, such as hash partitioning, range partitioning and self-defined partitioning, all belong to one-time partitioning strategy in essence. Studies have shown that the distribution of intermediate data is generally imbalanced for most of big data sets. And therefore, the one-time partitioning strategy generally leads to imbalanced data partitioning results, and has great impact on the overall performance of MapReduce.

In this paper, we propose an iterative data partitioning strategy for MapReduce. The data block processed in Mapper phase is further subdivided into fine-grained block (FGB) that will be iteratively processed. According to the partitioning results of previous micro-partitions (MPs), the partitioning plan of current MP is made. The data skew generated by previous MPs can be continuously eliminated, and balanced data partitioning can be finally achieved. The partitioning timing, partitioning criteria, implementation model and partitioning algorithms of the proposed strategy are given in detail. Experimental results show that our proposed strategy can achieve much higher balanced data partitioning results than two existing data partition strategies, and can also achieve higher overall performance on those data sets with obvious data skew.

This work is supported by National Natural Science Foundation of China (61379017), the Public Project of Zhejiang Province (2017C31014), and the Open Projects Funding of State Key Lab of Computer Architecture (CARCH201804), ICT, CAS.