计算机系统应用 ISSN 1003-3254, CODEN CSAOBN
Computer Systems & Applications,2019,28(6):189−197 [doi: 10.15888/j.cnki.csa.006985]

E-mail: csa@iscas.ac.cn
http://www.c-s-a.org.cn
Tel: +86-10-62661041

广播机制解决 Shuffle 过程数据倾斜的方法①

吴恩慈

(上海淇毓信息科技有限公司, 上海 200120)
通讯作者: 吴恩慈, E-mail: kaneeasy@163.com

摘　要: 在 Spark 计算平台中, 数据倾斜往往导致某些节点承受更大的网络流量和计算压力, 给集群的 CPU、内存、磁盘和流量带来了巨大的负担, 影响整个集群的计算性能. 本文通过对 Spark Shuffle 设计和算法实现的研究, 深入分析在大规模分布式环境下发生数据倾斜的本质原因. 提出了广播机制避免 Shuffle 过程数据倾斜的方法, 分析了广播变量分发逻辑过程, 给出广播变量性能优势分析和该方法的算法实现. 通过 Broadcast Join 实验验证了该方法在性能上有稳定的提升.

关键词: 数据倾斜; 分区策略; 洗牌算法; 广播机制

引用格式: 吴恩慈.广播机制解决 Shuffle 过程数据倾斜的方法.计算机系统应用,2019,28(6):189–197. http://www.c-s-a.org.cn/1003-3254/6985.html

Method Research to Solve Shuffle Data Skew Based on Broadcast

WU En-Ci

(Shanghai Qiyu Information Technology Co. Ltd., Shanghai 200120, China)

Abstract: In the Spark computing platform, data skew often causes some nodes to withstand greater network traffic and computing pressure, which imposes a huge burden on the cluster's CPU, memory, disk, and traffic, affecting the computing performance of the entire cluster. Through the research on Spark Shuffle design and algorithm implementation, and deep analyses on the essential reasons of data skew in large-scale distributed environment, this study proposes a method to avoid data skew in shuffle process through the broadcast mechanism, analyzes the process of broadcast variable distribution logic, and gives the algorithm implementation and performance advantage analysis of the method. The performance of the method is improved by the Broadcast Join experiment.

Key words: data skew; partition; shuffle; broadcast

Spark 由加州大学伯克利分校 AMP 实验室开发, 是基于 RDD 的内存计算框架, 在流式计算场景中表现良好. 在 Spark 分布式流计算集群的实践过程中, 往往出现数据分配不均匀的现象, 某个分区的数据显著多于其它分区, 该节点的计算速度成为整个集群的性能瓶颈. 任务执行慢的节点往往导致内存溢出, 服务器 CPU 使用率在短时间内急剧增加.

发生数据倾斜时, 某个任务处理的数据量远大于其他分片, 从而增加了整个阶段的完成时间. 由于原始数据源的分布不均匀, 每个 Reducer 在分区映射过程中计算的数据量也不相同, 任务执行时间不同, 增加宽依赖阶段的延迟, 降低集群作业执行的效率. 虽然支持用户定义分区, 但真正的数据分布难以预测, 无法保证自定义分区功能的合理性和准确性. 数据倾斜问题很难规避, 为有效改善数据倾斜问题, 本文在分析和研究国内外对该问题的研究成果和实践经验的基础上, 主要做了以下工作.

1) 研究了 Spark Shuffle 设计和算法实现, 分析了

① 收稿时间: 2018-12-19; 修改时间: 2019-01-15; 采用时间: 2019-02-19

哈希和排序两类 Shuffle 机制的实现过程, 深入分析在 Shuffle 过程发生数据倾斜的本质原因.

2) 分析了 Spark 流计算集群中, 发生数据倾斜常见业务场景, 以及数据倾斜问题的原因和发生过程, 提供了问题定位的方法和步骤.

3) 提出了 Broadcast 机制避免某些场景下 Shuffle 导致的数据倾斜问题的方法, 给出广播变量分发机制和算法实现. 通过 Broadcast 实现 Join 算子的实验, 验证了该方法在性能上有较大提升.

1 相关工作

MapReduce 框架数据倾斜解决方案包括静态和动态自适应调整. 静态自适应是事先分析数据集中键的分布特征, 选择适当的分区方法. 该方法需要总结出一套和业务相关的数据集抽象规则算法, 可以通过机器学习的方式来训练算法, 提高对各种类型数据集的适配度. 文献[1]提出的 LEEN 方法, 在 Skew Reduce 的基础上将分析过程移至 Map 完成后, 考虑 Reduce 端输入数据的公平性和数据本地性, LEEN 方法在缓解慢任务和 Shuffle 引起的网络拥塞方面有较大的性能提升[1].

动态自适应的思想是在应用运行时, 实时检测当前各个 Partition 中的数据填充情况. 如果发现存在严重的数据倾斜问题, 在下一步进行调整, 以避免原始 MapReduce 任务之间的不能全局优化的问题. 文献[2]提出了 Skew Tune 方法, 推测当前任务需要完成时间, 确定最可能的慢任务[2]. 扫描慢任务要处理的输入数据, 在数据偏斜场景中性能有较好的表现. 文献[3]提出在执行流程中插入一个 Intermediate Reduce(IR) 的新阶段, 用于并行处理动态的中间结果, 使所有键值对都能够利用到所有的节点资源[3]. 文献[4]提出 Adaptive MapReduce 策略, 动态地调整每个 Mapper 处理的输入数据量, 在 Mapper 端使用一个固定大小的哈希表执行数据本地聚合[4]. 文献[5]提出了任务重分割方法, 通过监视任务的执行进度, 根据预定策略直接切割任务本身[5].

本文与上述研究成果的不同之处在于通过 Broadcast 机制将数据分发到计算节点中, 实现数据本地性, 从根本上避免 Shuffle 过程, 不需要做额外的开发和部署. 能够快速解决特定场景下的数据倾斜问题, 具有较高的实用性.

2 分区策略和 Shuffle 算法

2.1 分区策略

分区策略 (Partition) 的主要作用是确定 Shuffle 过程中 Reducer 的数量, 以及 Mapper 侧数据应该分配给哪个 Reducer. 分区程序可以间接确定 RDD 中的分区数和分区中的内部数据记录数. Spark 内置了 Hash Partition 和 Range Partition, 支持自定义分区器. Hash Partition 的算法实现是获取 Key 的 HashCode 除以子 RDD 分区的数量取余. 哈希分区器易于实现并且运行速度快. 但有一明显的缺点是不关心键值的分布情况, 其散列到不同分区的概率会因数据而有差别.

Range Partition 在一定程度上避免了该问题. 范围分区器根据父 RDD 的数据特征确定子 RDD 分区的边界, 通过对数据进行采样和排序, 将父 RDD 数据划分为 M 个数据块. 如果数据均匀分布, 则为每个分区提取的样本大小都很接近. 如果所有数据都分配给每个分区, 则每个分区都会提取相同的数据量, 并统计每个分区的实际数据量. 若出现数据倾斜的情况, 则对个别分区重新采样.

2.2 Hash-Based Shuffle 算法

Spark 作为 MapReduce 框架的一种实现, 本质上将 Mapper 端输出的数据通过 Partition 算法, 确定发送到相应的 Reducer, 该过程涉及网络传输和磁盘读写. Spark 提供了哈希和排序两类 Shuffle 机制. ShuffleManager 主要作用是提供 Shuffle Writer 和 Shuffle Reader 过程.

Hash-Based Shuffle 机制的实现过程包括聚合和计算, 使用哈希表存储所有聚合数据的处理结果. 聚合和计算过程不进行排序, 分区内部的数据是无序的, 如果希望有序就需要调用排序操作. 哈希 Shuffle 强制要求在 Map 端进行聚合操作, 对于某些键值重复率不高的数据会影响效率.

如图 1 所示, 哈希 Shuffle 算法实现过程. 每个 Mapper 都会根据 Reducer 的数量创建一个相应的桶 (Bucket). Mapper 生成的结果将根据设置的分区算法填充到每个桶中. 当 Reducer 启动时, 它将根据自己的 TaskId 和它所依赖的 MapperId, 从远程或本地 Block Manager 获取相应的 Bucket 作为 Reducer 的输入. Bucket 是一种抽象概念, 可以对应于某个文件或文件的一部分. 哈希 Shuffle 算法实现中, Mapper 会针对每个 Reducer 生成一个数据文件, 当 Mapper 和 Reducer 数量比较多的时候, 生成大量的磁盘文件.

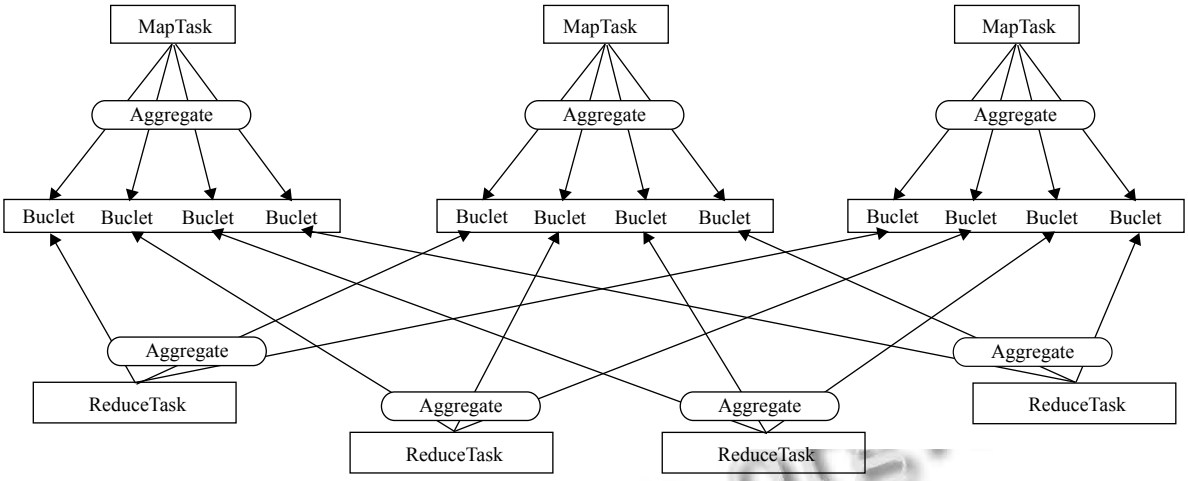


图 1 Hash-Based Shuffle 算法实现过程

开启 consolidate 优化机制后, Shuffle Writer 过程中 Task 不是为下游 Stage 的每个 Task 创建一个文件. 每个 ShuffleFileGroup 对应一批磁盘文件, 磁盘文件数与下游 Stage 中的 Task 总数相同. Executor 分配多少个 Core 就可以并行执行多少个任务, 第一批并行执行中的每一个任务都创建一个 ShuffleFileGroup, 并将数据写入相应的磁盘文件. 当执行器执行下一批任务时, 复用以前存在的 ShuffleFileGroup 的磁盘文件, Task 将数据写入现有磁盘文件. consolidate 机制允许不同的任务复用同一批磁盘文件, 减少了文件数量并提高了 Shuffle Writer 的性能.

2.3 Sort-Based Shuffle 算法

如图 2 所示, Sort-Based Shuffle 算法实现过程. 排序 Shuffle 相比于哈希 Shuffle, 两者的 Shuffle Reader 过程是一致的, 区别在 Shuffle Writer 过程. 排序 Shuffle 允许 Map 端不进行聚合操作, 在不指定聚合操作的情况下, 排序 Shuffle 机制 Mapper 端用数据缓存区（Buffer）存储所有的数据. 对于指定聚合操作的情况下, 排序 Shuffle 仍然使用哈希表存储数据, 聚合过程与哈希 Shuffle 的基本一致. 无论是 Buffer 还是 HashMap, 每更新一次都检查是否需要将现有的数据溢存到磁盘中, 需要对数据进行排序, 存储到一个文件中. 更新完所有数据后, 将多个文件合并为一个文件, 并确保每个分区的内部数据都是有序的.

排序 Shuffle 机制包括普通和 bypass 两种运行模式. 当 Shuffle Reader Task 的数量小于等于 bypassMergeThreshold 的值时就会启用 bypass 模式. 在普通操作模式中, 首先将数据写入内存数据结构, 根据不同的 Shuffle 运算符选择不同的数据结构. 如 ReduceByKey 之类的聚合运算符选择哈希数据结构, Join 类的普通运算符使用数组数据结构. 在将每条数据写入内存之后, 确定是否已达到临界阈值. 如果达到临界阈值则将内存中的数据写到磁盘, 然后清空内存中的数据.

在溢出到磁盘文件之前, 根据 Key 对内存中已有的数据进行排序, 排序后数据将批量写入磁盘文件. 排序后的数据以每批 10 000 个批量写入磁盘文件, 从而有效减少磁盘 IO 数量. 将 Task 的所有数据写入内存数据结构的过程中, 会发生多次磁盘溢出操作, 生成多个临时文件, 最后合并所有先前的临时磁盘文件. 由于一个 Task 仅对应于一个磁盘文件, 因此将单独写入索引文件以标识文件中每个下游 Task 数据的开始和结束位置, 磁盘文件合并过程减少了文件数量.

在 bypass 模式下, 为每个下游 Task 创建一个临时磁盘文件, 并根据 Key 的 HashCode 写入相应的磁盘文件. 写入磁盘文件时先写入内存缓冲区, 然后在缓冲区满后溢出到磁盘文件. 它还将所有临时磁盘文件合并到一个文件中, 并创建一个单独的索引文件. 此过程的磁盘写入机制与未优化的哈希 Shuffle 相同, 但是 Shuffle Reader 性能会更好. 这种机制和普通的排序 Shuffle 之间的区别是磁盘写入机制不同, 不会被排序. 启用 bypass 机制的最大优点是在 Shuffle Writer 过程中不需要执行数据排序操作, 节省了部分性能开销.

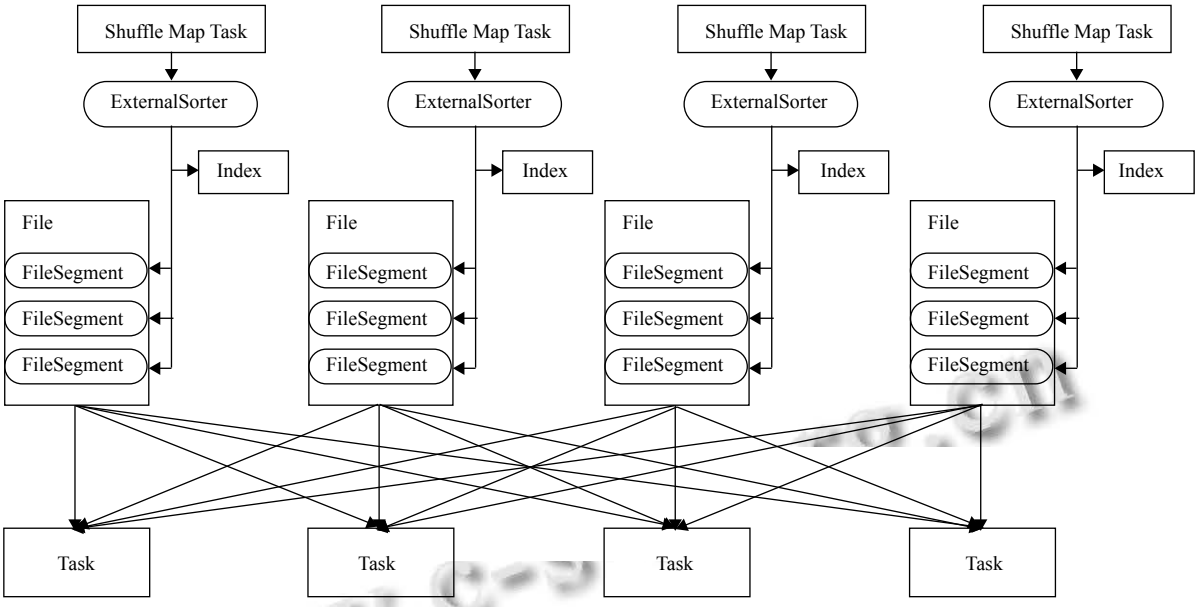


图 2 Sort-based Shuffle 算法实现

3 数据倾斜问题分析和定位

3.1 数据倾斜原因分析

发生数据倾斜的原因主要包括输入数据源分布不均匀, 以及计算过程中数据拉取时间不均匀. 输入数据分布不均匀的一般表现为原始数据, 或者中间临时数据中 Key 分布不均匀[6]. 例如 Spark Streaming 通过 Direct Stream 读取 Kafka 数据. 由于 Kafka 的每个分区都对应于 Spark 的相应任务, Kafka 相关主题的分区之间的数据是否平衡, 直接决定 Spark 在处理数据时是否会产生数据偏斜. 如果使用随机分区, 概率上分区之间的数据是平衡的, 不会生成数据倾斜. 但很多业务场景要求将具备同一特征的数据顺序消费, 需要将具有相同特征的数据放于同一个分区中, 一个典型的场景是相同的用户相关的 PV 信息放置在相同的分区中, 很容易导致数据倾斜.

数据拉取时间不均匀一般是硬件计算能力不均匀, 或者网络传输能力不均匀造成的[7]. 比如 PageRank 算子分为三个 Stage 运行, 由于第二个 Stage 产生了 Shuffle 是最容易发生数据倾斜, 每个 Task 处理分区数据绑定了各个顶点权重, 然后收集其邻接节点的权重. 由于 Executor 需要从非本地节点上拉取上一个 Stage 中得到的节点信息. 如果数据分布不均匀, 某些节点会比其他节点承受更大的网络流量和计算压力. 数据倾斜的计算时间主要花费在 Shuffle 上, 提高 Shuffle 性能有利于提高应用程序的整体性能.

3.2 数据倾斜定位方法

通过 Spark 界面观察每个阶段任务当前分配的数据量, 进一步确定数据的不均匀分布是否导致了数据倾斜[8]. 只要在代码中看到 Shuffle 类操作符, 或在 Spark SQL 语句中看到导致 Shuffle 的语句, 就可以确定划分 Stage 的边界. Stage 1 的每个任务开始运行时, 将首先执行 Shuffler Reader 操作, 从 Stage 0 的每个任务中提取需要处理的 Key. 比如 Stage 1 在执行 ReduceByKey 操作符之后计算出最终 RDD, 然后执行收集算子将所有数据拉到 Driver 中.

数据倾斜往往发生在 Shuffle 过程中, 可能会触发 Shuffle 操作的算子包括 GroupByKey、ReduceByKey 和 AggregateByKey 等. 在执行 Shuffle 时必须将每个节点上的相同 Key 拖动到同一个节点上的 Task 进行处理. 如果某个 Key 对应的数据量特别大就会发生数据倾斜, Job 运行得非常缓慢, 甚至可能因为某个 Task 处理的数据量过大导致内存溢出.

4 广播机制避免 Shuffle 的策略

4.1 Torrent Broadcast 原理

通过广播机制将只读变量从一个节点发送到其他 Executor 节点, 进程内运行的任务属于同一个应用程序, 在每个执行器节点上放置广播变量可以由该节点的所有任务共享[9]. Torrent Broadcast 算法的基本思想是将广播变量划分为多个数据块. 当某个执行器获得

数据块时, 当前执行器被视为数据服务器节点. 随着越来越多的执行器获得数据块, 更多的数据服务器节点可用. 广播变量可以快速传播到所有节点. Torrent Broadcast 读取数据的方式与读取缓存类似, 使用 Block Manager 自带的 NIO 通信方式传递数据, 存在的问题是慢启动和占内存[10]. 慢启动指的是刚开始数据只在 Driver 节点上, 要等执行器获取多轮数据块后, 数据服务节点才会变得可观, 后面的获取速度才会变快. 执行器在获取完所有数据分块后进行反序列化时, 需要将近两倍的内存消耗.

4.2 Broadcast 变量性能优势

Driver 先把广播变量序列化为字节数组, 然后切割成 BLOCK_SIZE 大小的数据块. 在数据分区和切割之后, 数据分区元信息作为全局变量被存储在 Driver 节点的 Block Manager 中. 之后每个数据分块都做相同的操作, Block Manager Master 可以被 Driver 和所有 Executor 访问到. 执行器反序列化 Task 时, 先询问所在的 Block Manager 是会否包含广播变量, 若存在就直接从本地 Block Manager 读取数据. 否则连接到 Driver 节点的 Block Manager Master 获取数据块的元信息.

广播机制把只读变量通过共享的方式有效的提高了集群的性能. 大多数 Spark 作业的性能主要消耗在 Shuffle 环节, 该环节包含了大量的磁盘 IO、序列化、网络数据传输等操作. 通过广播机制避免 Shuffle, 可显著提高应用运行速度. 例如普通 Join 操作会产生 Shuffle, RDD 中相同的 Key 需要通过网络拉取到同一个节点上. 在算子函数使用广播变量时, 首先会判断当前 Task 所执行器内存中是否有变量副本. 如果有则直接使用, 如果没有则从 Driver 或者其他 Executor 节点上远程拉取一份放到本地执行器内存中. 广播变量保证了每个执行器内存中只驻留一份变量副本, Executor 中的 Task 执行时共享该变量副本, 减少变量副本的数量和网络传输的性能开销, 降低了执行器内存的开销, 降低 GC 的频率, 会极大地提升集群性能.

4.3 Broadcast Join 代码实现

如图 3 所示, Broadcast Join 算子 Scala 代码实现. 当在 RDD 上使用 Join 类操作或者在 Spark SQL 中使用联接语句时, 普通联接运算符会产生 Shuffle 过程, 并将相同的 Key 数据拉入 Shuffle Reader Task 进行联接操作. 如果连接操作中 RDD 或表的数据量相对较小, 则不使用连接运算符而是使用广播变量和映射类运算符来实现 Join 操作, 从而完全避免了 Shuffle 过程和数据歪斜. 较小的 RDD 中的数据通过收集操作直接拉入 Driver 节点的内存中, 创建广播变量. 然后从广播变量中获取较小 RDD 的数据, 并在另一个 RDD 上执行映射类运算符. 根据连接键比较当前 RDD 的每个数据. 如果连接键相同则将两个 RDD 的数据连接在一起, 实现 Join 操作的效果. 该方法不足在于 Driver 和 Executor 节点都要存储广播变量的全部数据, 比较消耗内存.

```
/** 通过broadcast+map实现join操作 */
def broadcastJoin(conf: SparkConf) = {
  val sc = new SparkContext(conf)
  /** littleRDD数据结构(cust_no,repay_amt) */
  val littleRDD = sc.textFile("customer_repay")
    .map(e => (e.split(",")(0), e.split(",")(1)))
  /** largeRDD数据结构(cust_no,create_id)  */
  val largeRDD = sc.textFile("action_record")
    .map(e => (e.split(",")(0), e.split(",")(1)))
  /** 将数据量比较小的RDD的数据collect到Driver中 */
  val data = littleRDD.collect()
  /** 使用Spark广播功能将小RDD的数据转换成广播变量 */
  val broadcast = sc.broadcast(data)
  /** 通过广播变量获取到本地Executor中的rdd1数据 */
  val joinRDD = largeRDD.flatMap(e => {
    val list = broadcast.value
      .filter(b => b._1.equals(e._1))
    list.map(tuple2 => (e._1, (e._2, tuple2._2)))
  })
  /** Join数据结果保存 */
  joinRDD.saveAsTextFile("action_result")
  sc.stop()
}
```

图 3 Broadcast Join 算子 Scala 代码实现

5 实验和分析

5.1 集群硬件环境配置

本文使用的实验环境是一个由四个节点组成的服务器集群. 集群采用主-从体系结构, 其中一个是主节点, 其他三个是从节点. Spark Job 和 Hadoop 文件系统部署在同一节点上. 大部分的 Spark 作业会从外部存储系统读取输入数据, 比如 Hadoop 文件系统, 将其与存储系统放得越近越好. 在相同的节点上安装 Spark Standalone 模式集群, 并单独配置 Spark 和 Hadoop 的内存和 CPU 使用以避免干扰[11]. 实验服务器集群环境的硬件配置根据数据量调整, 参考官方建议初始配置如下.

单个节点服务器最初配置四个磁盘. Spark 很多计算都在内存中进行, 但当数据在内存中装不下的时候, 仍然使用本地磁盘来存储数据, 以及在不同阶段之间保留中间的输出. 在实验中每个计算节点有 4–8 个磁盘. 集群网络最初配置为万兆网卡. 当数据在内存中时,

许多 Spark 应用程序都与网络密切相关. 使用万兆或更高的网络是使这些应用程序更快的比较好的方法. 分布式 Reduce 应用程序尤其适用, 比如 group-by、reduce-by 等操作.

单个节点服务器的初始内存配置为 16GB. Spark 应用分配 75% 的内存, 剩下的部分留给操作系统和缓冲区缓存. 使用 Spark 监视 UI 的 Storage 选项卡查看内存使用情况, 内存使用情况受存储等级和序列化格式的影响很大[12]. 单个计算节点的 CPU 核心数量最初配置 8 个 Core. Spark 在每台机器上可以扩展到数几十个 Core. 测试阶段在每台机器上提供 8–16 个内核, 根据服务器负载的消耗可以配置更多.

5.2 系统配置和源数据

如图 4 所示, 集群运行时软件版本信息, 图 5 是集群系统配置信息, 都是当前业界使用的稳定版本和配置. 实验数据来自于某金融机构的客户消费贷款逾期信息, 包括逾期客户还款信息, 以及催收行动信息. 选取逾期客户还款信息数据量相对较少, 10 万条记录, 数据结构中包括记录 ID、还款时间、还款金额、银行编码、还款卡号、客户姓名、客户 ID 等信息. 逾期客户机构催收行动数据量相对较大, 1000 万条记录, 数据结构包括记录 ID、催收行动码、行动描述、客户电话、客户关系、通话时长、客户 ID、客户姓名、行动时间、催收员、催收机构等信息. 根据客户 ID 进行 Join 操作, 计算指标包括机构催收员工作量和催收员工作业绩.

Runtime Information

Name	Value
Java Version	1.8.0_45 (Oracle Corporation)
Scala Version	version 2.11.7
Hadoop Version	version 2.7
Spark Version	version 2.1.3

图 4 集群软件版本信息

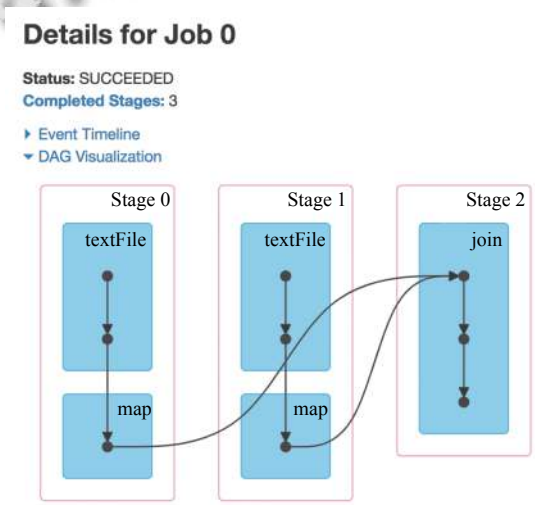
5.3 实验结果分析

如图 6 所示, Broadcast Join 算子 DAG 视图中不存在 Shuffle, 普通 Join 算子 DAG 视图的复杂度明显高于 Broadcast Join 算子. 图 6(a) 是普通 Join 操作的 DAG 视图, 根据 RDD 的宽依赖关系分为三个阶段, 有向无环图描述了阶段之间的依赖关系, 当前 Stage 只能在父 Stage 之后执行. 从 DAG 视图清晰的看到普通 Join 算子存在 Shuffle 过程. 图 6(b) 是 Broadcast Join 算子有向无环图只有一个阶段, 逻辑过程相对简单.

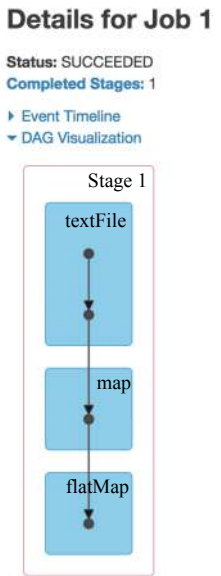
System Properties

Name	Value
spark.scheduler.mode	FIFO
OS	CentOS release 6.6 (Final)
java.vendor.url	http://java.oracle.com/
java.vm.name	Java HotSpot(TM) 64-Bit

图 5 集群系统配置信息



(a) 普通Join操作DAG视图



(b) Broadcast Join操作DAG视图

图 6 Join 算子 DAG 视图

如图 7 所示, Broadcast Join 算子中 Task 统计数据表明性能上存在明显的优势. 图 7(a) 是普通 Join 算子某阶段 Task 性能统计数据, 包括 Task 持续时间、GC 执行时间和 Shuffle 数据量等 3 个方面的统计信息. 图 7(b) 是 Broadcast Join 算子 Task 性能统计数据, 相比于普通 Join 算子的 Task 性能统计信息, 在 3 个方面都存在明显的优势.

如图 8 所示, Broadcast Joins 算子各个 Stage 的磁盘读写和网络流量、任务持续时长等都存在明显的优势. 图 8(a) 是普通 Join 算子各个阶段执行情况, 共有 3 个阶段, 整个过程耗时 6–7 秒, Shuffle 并行度为 9, 涉及 Shuffle Reader 数据量 61.3MB, Shuffle Writer 数据量 61.3 MB. 如图 8(b) 所示, Broadcast Join 算子只有 1 个阶段, 没有涉及 Shuffle 数据读写过程, 数据输入 117.4 MB, Task 总数量明显小于普通 Join 算子的数量, 执行时间是 3 秒, 和普通 Join 算子相比在性能上有较大的提升.

Summary Metrics for 9 Completed Tasks

Metric	Min	25th percentile	Median	75th percentile	Max
Duration	0.5 s	1 s	1 s	1 s	2 s
GC Time	0 ms	0.1 s	0.1 s	0.7 s	0.8 s
Shuffle Read Size / Records	6.8 MB / 521290	6.9 MB / 528201	8.9 MB / 531465	7.3 MB / 552584	7.8 MB / 573713

Aggregated Metrics by Executor

Executor ID	Address	Task Time	Total Tasks	Failed Tasks	Succeeded Tasks	Shuffle Read Size / Records
driver	localhost:51409	11 s	9	0	9	64.1 MB / 4869604

(a) 普通Join操Task性能统计

Summary Metrics for 9 Completed Tasks

Metric	Min	25th percentile	Median	75th percentile	Max
Duration	0.8 s	1 s	1 s	1 s	1 s
GC Time	0 ms	7 ms	7 ms	28 ms	28 ms
Input Size / Records	13.0 MB / 539503	13.0 MB / 539648	13.0 MB / 539695	13.0 MB / 539759	13.3 MB / 552244

Aggregated Metrics by Executor

Executor ID	Address	Task Time	Total Tasks	Failed Tasks	Succeeded Tasks	Input Size / Records
driver	localhost:51610	11 s	9	0	9	117.4 MB / 4869504

(b) Broadcast Join操作Task性能统计

图 7 Join 操作 Task 性能数据

Completed Stages (3)

Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
2	count at JoinBroadcastAction.scala:118 +details	2018/12/14 17:25:22	3 s	9/9			61.3 MB	
1	map at JoinBroadcastAction.scala:101 +details	2018/12/14 17:25:20	26 ms	2/2	3.0 KB			3.8 KB
0	map at JoinBroadcastAction.scala:107 +details	2018/12/14 17:25:20	3 s	9/9	117.4 MB			61.3 MB

(a) 普通Join算子Stage执行情况

Completed Stages (1)

Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
1	count at JoinBroadcastAction.scala:83 +details	2018/12/14 17:30:01	3 s	9/9	117.4 MB			

(b) Broadcast Join Stage执行情况

图 8 Join 算子各个 Stage 执行情况

如图 9 所示, Broadcast Joins 算子不存在数据倾斜问题. 图 9(a) 是普通 Join 操作任务数据分配和执的详细数据. 从列 Shuffle Read Size 可以看出, 任务分配出现了数据倾斜问题, 被分配数据量较大的 Task 执行时间明显高于其他任务的持续时间, 消耗更大的资源和网络流量, 其他已经完成计算的节点处于等待状态. 图 9(b) 是 Broadcast join 算子任务数据分配和执行情况明细, 从持续时间列和 Input Size 列看出, 数据几乎是均匀分配, 8 个任务的持续时间是 1 秒, 1 个是 0.9 秒, 充分发挥了数据本地性特性, 每个节点的计算资源都被有效利用.

如图 10 所示, Broadcast Join 算子在高并发的应用情况下性能上存在稳定的提升. 如图 10(a) 所示, 普通 Join 算子压测统计, 通过压测工具 10 万次的统计结果, 统计了平均持续时间、中位数和偏差情况. 从图中看出, 持续时间绝大部分相对集中和稳定在 7 秒左右. 如图 10(b) 所示, Broadcast join 算子压测情况统计, 比较相同的统计指标存在明显的优势, 持续时间基本集中和稳定在 4 秒左右.

6 总结

本文研究了 Spark Shuffle 设计和算法实现, 分析了哈希和排序两类 Shuffle 机制的实现过程, 深入分析在 Shuffle 过程发生数据倾斜的本质原因. 进一步分析了 Spark 流计算集群中, 发生数据倾斜常见业务场景, 分析数据倾斜问题的原因和发生过程, 提供了问题定位的方法和步骤. 提出了广播机制避免某些场景下的数据倾斜问题, 给出广播变量分发机制和算法实现. 通过 Broadcast 实现 Join 算子的实验, 相对于直接操作

Join 算子, 通过 DAG 视图、任务持续时间、Shuffle 读写数据量等指标的比较和分析, 验证了广播机制在性能上有较大提升, 压力测试进一步验证了在大规模应用的情况下性能有稳定的改善.

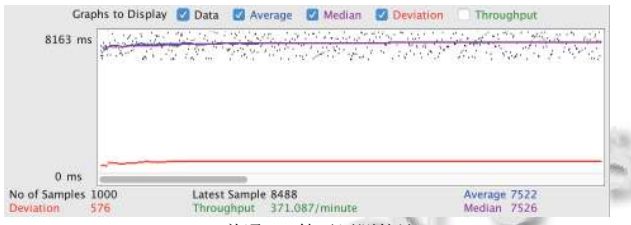
Index ▲	ID	Attempt	Status	Locality Level	Executor ID / Host	Launch Time	Duration	GC Time	Shuffle Read Size / Records	Errors
0	11	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2018/12/14 17:25:22	2 s	0.3 s	7.5 MB / 573708	
1	12	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2018/12/14 17:25:22	2 s	0.3 s	6.6 MB / 528201	
2	13	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2018/12/14 17:25:22	3 s	0.3 s	18.0 MB / 1552584	
3	14	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2018/12/14 17:25:22	2.8 s	0.3 s	15.5 MB / 1025129	
4	15	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2018/12/14 17:25:24	0.7 s	70 ms	6.6 MB / 528782	
5	16	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2018/12/14 17:25:24	0.3 s	70 ms	3.4 MB / 121290	
6	17	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2018/12/14 17:25:24	0.3 s	70 ms	2.5 MB / 173713	
7	18	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2018/12/14 17:25:24	0.6 s	70 ms	6.6 MB / 531465	
8	19	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2018/12/14 17:25:25	0.4 s		6.7 MB / 534732	

(a) 普通Join算子数据分配和执行

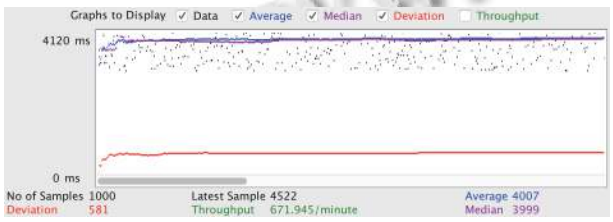
Index ▲	ID	Attempt	Status	Locality Level	Executor ID / Host	Launch Time	Duration	GC Time	Input Size / Records	Errors
0	2	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2018/12/14 17:39:30	1 s	25 ms	13.0 MB / 539664	
1	3	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2018/12/14 17:39:30	1 s	25 ms	13.0 MB / 539648	
2	4	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2018/12/14 17:39:30	1 s	25 ms	13.0 MB / 539507	
3	5	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2018/12/14 17:39:30	1 s	25 ms	13.0 MB / 539779	
4	6	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2018/12/14 17:39:32	1 s	6 ms	13.3 MB / 552244	
5	7	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2018/12/14 17:39:32	1 s	6 ms	13.0 MB / 539705	
6	8	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2018/12/14 17:39:32	1 s	6 ms	13.0 MB / 539503	
7	9	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2018/12/14 17:39:32	1 s	6 ms	13.0 MB / 539695	
8	10	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2018/12/14 17:39:33	0.9 s	2 ms	13.0 MB / 539759	

(b) Broadcast Join数据分配和执行

图 9 Join 算子数据分配和执行



(a) 普通Join算子压测统计



(b) Broadcast join压测统计

图 10 Join 算子压测数据统计

参考文献

1 Chen Q, Yao JY, Xiao Z. LIBRA: Lightweight data skew mitigation in MapReduce. IEEE Transactions on Parallel and Distributed Systems, 2015, 26(9): 2520–2533. [doi: 10.1109/TPDS.2014.2350972]

2 Al Hajj Hassan M, Bamha M. Towards scalability and data skew handling in GroupBy-joins using MapReduce model. Procedia Computer Science, 2015, 51: 70–79. [doi: 10.1016/j.procs.2015.05.200]

3 de Oliveira PM, Allison PM, Mastorakos E. Ignition of uniform droplet-laden weakly turbulent flows following a laser spark. Combustion and Flame, 2019, 199: 387–400. [doi: 10.1016/j.combustflame.2018.10.009]

4 Tang Z, Zhang XS, Li KL, *et al*. An intermediate data placement algorithm for load balancing in Spark computing environment. Future Generation Computer Systems, 2018, 78: 287–301. [doi: 10.1016/j.future.2016.06.027]

5 Mustafa S, Elghandou I, Ismail MA. A machine learning approach for predicting execution time of spark jobs. Alexandria Engineering Journal, 2018, 57(4): 3767–3778. [doi: 10.1016/j.aej.2018.03.006]

6 Tang JC, Xu M, Fu SJ, *et al*. A scheduling optimization technique based on reuse in spark to defend against APT

attack. Tsinghua Science and Technology, 2018, 23(5): 550–560. [doi: 10.26599/TST.2018.9010022]

7 Ye XM, Chen XS, Liu DH, *et al*. Efficient feature extraction using apache spark for network behavior anomaly detection. Tsinghua Science and Technology, 2018, 23(5): 561–573. [doi: 10.26599/TST.2018.9010021]

8 廖旺坚, 黄永峰, 包从开. Spark 并行计算框架的内存优化. 计算机工程与科学, 2018, 40(4): 587–593. [doi: 10.3969/j.issn.1007-130X.2018.04.003]

9 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. 电子学报, 2017, 45(2): 278–284. [doi: 10.3969/j.issn.0372-2112.2017.02.003]

10 李俊丽. 基于 Spark 平台的离群数据并行挖掘算法. 计算机与数字工程, 2018, 46(11): 2175–2178. [doi: 10.3969/j.issn.1672-9722.2018.11.003]

11 朱继召, 贾岩涛, 徐君, 等. SparkCRF: 一种基于 Spark 的并行 CRFs 算法实现. 计算机研究与发展, 2016, 53(8): 1819–1828.

12 谭亮, 周静. 基于 Spark Streaming 的实时交通数据处理平台. 计算机系统应用, 2018, 27(10): 133–139.