计算机系统应用 ISSN 1003-3254, CODEN CSAOBN
Computer Systems & Applications,2023,32(3):70−85 [doi: 10.15888/j.cnki.csa.008998]

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

混合存储模式下 MapReduce 作业调度①

杨振宇, 牛天洋, 吕　敏

(中国科学技术大学 计算机科学与技术学院, 合肥 230022)
通信作者: 吕　敏, E-mail: lvmin05@ustc.edu.cn

摘　要: 在异构 Hadoop 集群场景中, 为了缓和由于纠删码和副本存储模式混合使用, 以及服务器节点本身实时算力差异造成的 MapReduce 作业处理效率低下的问题, 本文实现了一种根据数据存储情况和节点实时负载来在多并发场景下动态调节 MapReduce 作业任务分配情况的调度策略. 该策略通过修改当前 Hadoop 框架中的数据存储选址策略并对节点任务并发量进行动态控制, 在多作业并发时实现更加均衡的作业间资源分配. 实验结果表明, 相较于 Hadoop 默认的两种作业调度策略, 本文提出的调度模式能够将作业完成时间缩短约 17%, 并有效避免部分作业面临的饥饿现象.

关键词: MapReduce; 作业调度; 纠删码; 异构集群; 混合存储; 云计算; 负载均衡; 大数据

引用格式: 杨振宇,牛天洋,吕敏.混合存储模式下 MapReduce 作业调度.计算机系统应用,2023,32(3):70–85. http://www.c-s-a.org.cn/1003-3254/8998.html

MapReduce Job Scheduling in Hybrid Storage Modes

YANG Zhen-Yu, NIU Tian-Yang, LYU Min

(School of Computer Science and Technology, University of Science and Technology of China, Hefei 230022, China)

Abstract: In a heterogeneous Hadoop cluster scenario, the hybrid use of erasure codes and replica storage modes, as well as the real-time computing capability difference of server nodes lead to the low efficiency of MapReduce job processing. To deal with this problem, this study implements a scheduling strategy that dynamically adjusts MapReduce job assignment in multi-concurrent scenarios according to data storage situations and the real-time load of nodes. This strategy dynamically controls the concurrent amount of tasks of each node by modifying data storage location strategies in the current Hadoop framework, so as to achieve more balanced resource allocation among jobs when multiple jobs are concurrent. The experimental results show that the scheduling mode proposed in this study can shorten the job completion time by about 17% and effectively avoid the starvation phenomenon faced by some jobs compared with the two default job scheduling strategies of Hadoop.

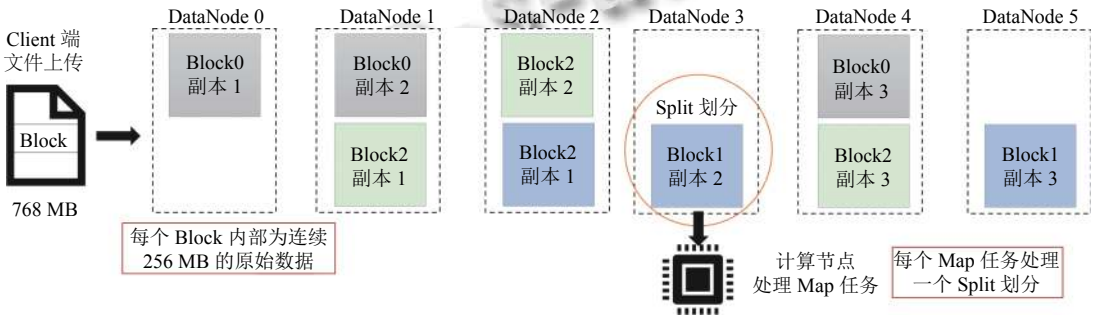
Key words: MapReduce; job scheduling; erasure code; heterogeneous cluster; hybrid storage; cloud computing; load balance; big data

MapReduce[1] 分布式计算框架作为 Hadoop 生态中的一个重要组成部分, 其核心思想是将大数据的处理过程抽象为 Map (映射) 和 Reduce (归约/化简) 两个阶段, 通过资源管理框架 YARN 来进行资源管理与作业调度, 实现对底层 HDFS[2,3] 文件系统中存储的大规模离线数据进行分析计算, 目前 MapReduce 被广泛应用于工业界的多种海量数据分析场景[4,5].

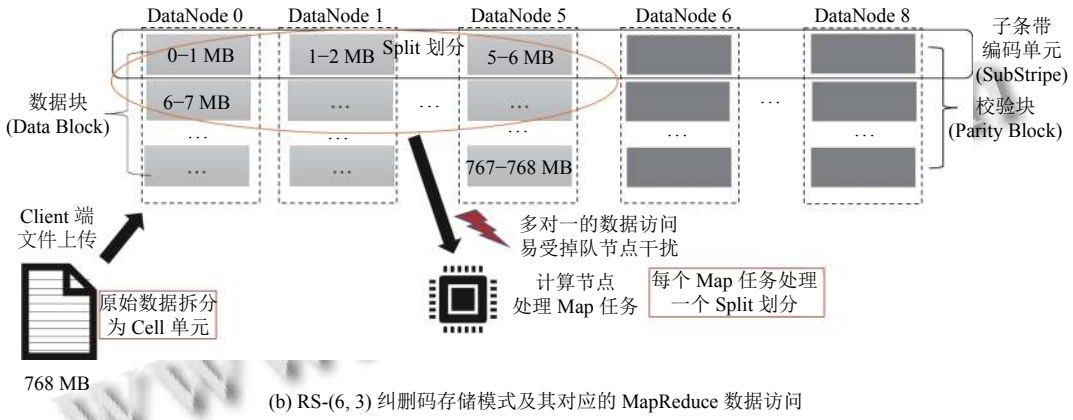
另一方面, 随着数据的爆发式增长[6], 数据中心的

① 基金项目: 国家自然科学基金重点项目 (61832011)
收稿时间: 2022-08-09; 修改时间: 2022-09-15; 采用时间: 2022-09-27; csa 在线出版时间: 2022-12-09
CNKI 网络首发时间: 2022-12-13

存储设备开销逐渐成为不可忽视的一部分[7]. 因此, 越来越多主流的分布式存储系统逐渐引入纠删码机制[8−11] 作为多副本容错机制的一种替代, 以在保障数据可靠性的同时进一步降低系统的存储开销. 当前 HDFS 中分别实现了上述两种容错机制, 其数据组织方式如图 1 所示, HDFS 的数据管理以 Block (块) 为单位, 将用户提交的文件划分为一个个 Block, 分散存储于各个 DataNode (用于存放数据的从节点). 在副本存储模式下, HDFS 按照 Block 的粒度对原始数据进行分割, 依照系统配置的数据块备份数量 (默认为 3) 和容单机架故障的数据块随机放置策略将数据块存放在不同的节点以满足节点以及机架级别的故障, 如图 1(a) 为 3 副本存储模式下 HDFS 中的数据存储情况. 而在纠删码存储模式下, 为了实现更好的在线编码性能和更少的存储开销, HDFS 在 Block 的数据划分逻辑上, 将原始数据拆分为了更细粒度的 Cell 单元, 默认为 1 MB 大小. 在进行编码操作时, 原始文件中连续的一部分将会按 Cell 进行划分和编码, 并随机分散放置在各个节点以保障机架级别的故障容错. 例如在图 1(b) 中原始文件的前 6 MB 通过 RS-(6, 3) 编码生成 3 个 Parity Cell, 分别放在 DataNode 0–8 这 9 个节点, 在外层, HDFS 仍然维护 Block 的划分逻辑, 当原始文件填满一组 Block 时, 则开始进行下一个条带的写入.



(a) 3副本存储模式及其对应的 MapReduce 数据访问



(b) RS-(6, 3) 纠删码存储模式及其对应的 MapReduce 数据访问

图 1 Hadoop 中不同存储模式及其对应的 MapReduce 数据访问情况

MapReduce 所处理的离线数据具有数据量大, 更新频率低等特性. 在 Map 阶段, 系统首先对原始数据进行分片 (Split) 从而确定 Map 任务的数量, 使每个任务都能处理原始数据集中连续的一部分内容. 资源管理框架 YARN 将根据集群各节点的配置文件, 以 container 为资源分配的基础单元管理各节点上的硬件资源 (CPU、内存等). 每个 Map 任务或 Reduce 任务都需要 YARN 为其分配 container 单元来进行计算.

如前文所述, 无论是原始数据还是经过计算后得到的输出结果, 底层 HDFS 采取纠删码机制能够成倍节省系统的存储占用. 但与此同时, 由于底层存储模式的改变, 以及异构环境中 YARN 资源管理和作业调度策略本身存在的不足, 使得在多作业并发场景下, MapReduce 作业的运行效率面临如下 3 个问题.

(1) 多种数据存储模式共存引起不同节点的数据访问热度倾斜. 在副本模式下, 如图 1(a) 所示, MapReduce

作业在进行数据分片时, 可以通过将分片大小和 Block 大小保持一致, 并在 YARN 进行任务分配时优先将任务分配到其对应数据的存放节点, 从而避免一个任务涉及多个远程节点的数据, 从而最大化利用数据的本地性. 而在纠删码存储模式下, 如图 1(b) 所示, 由于数据被拆分为更细粒度的存储单元, 因此一个 Map 任务往往需要从多个节点获取所需数据, 因此掉队节点 (straggler) 将会引发长尾任务影响 MapReduce 作业的运行效率. 此外, 在多作业并发的场景中, 由于不同作业处理的数据可能采用不同的存储模式, 因此集群中各节点的数据访问热度也会存在较大差异, 从而影响 MapReduce 框架的数据访问和处理效率.

(2) 静态的资源管理无法适应集群不同节点的负载变化. 多作业并发场景下, YARN 框架中静态的资源管理配置无法适应集群各节点动态变化的负载和本身的硬件异构. 例如 CPU 核心数较少, 主频较低的节点在本身 CPU 中高负载的情况下, 对本节点的任务并发量更加敏感. 同样地, 如果内存大小以及内存占用量不同的节点不能根据自身运行时情况决定节点上的任务并发数量, 也会造成任务运行缓慢甚至由于可用内存不足导致任务失败, 而失败任务的重试又将进一步影响 MapReduce 作业的完成速度.

(3) 作业间任务分布倾斜造成节点设备的有效利用率下降. 在多作业并发场景中, 由于作业提交存在时间差异, 在系统运行过程中存在任务的主资源需求和实际分配节点的硬件条件不匹配的情况. 例如在 CPU、内存等硬件性能存在差异的异构集群中, 先后向系统提交 A、B 两个大规模的 MapReduce 作业, 其中 A 作业对内存需求更加敏感, B 作业对 CPU 需求更加敏感, A 的部分任务在内存性能较好的节点上率先完成, 此时系统会将 B 作业中的部分任务分配到这些空闲节点上, 而如果这些节点中存在 CPU 硬件性能较差的情况, 将会造成 B 作业中部分任务的长尾效应, 影响作业完成时间.

针对上述 3 个问题, 本文从单个 MapReduce 作业和多 MapReduce 作业并发两个层面, 对当前 Hadoop 系统进行了对应优化, 主要贡献如下.

(1) 对于单个 MapReduce 作业, 通过修改当前 HDFS 的默认的随机数据放置策略和 YARN 的默认的根据配置文件静态确定 container 数量的资源管理方式, 提出 HDPA 纠删码数据放置策略和 DTAA 任务分配策略, 通过对异构集群中各节点的硬件信息和实时负载进行周期性采集, 以计算出合适的纠删码条带放置位置和 MapReduce 作业执行时各节点合适的任务并发量.

(2) 对于多作业并发场景, 提出动态平衡的公平作业调度方案 DB-Fair, 用来确定作业队列中各作业的计算资源分配和作业内任务的优先级划分, 并进行合适的任务分配位置选取.

通过上述两个层面的方案设计, 优化了前面所提到的 3 个问题. 实验结果表明, 本文提出的 DB-Fair 调度策略能够有效提升 Hadoop MapReduce 框架的处理效率, 此外在用户感知, 资源有效利用和集群负载均衡性等方面均要优于 YARN 的默认调度策略.

1 相关工作

1.1 研究现状

目前, 已有的 MapReduce 性能优化设计主要可以分为两类: 一类是从数据布局的角度出发, 通过更改 HDFS 中的数据放置, 优化 MapReduce 的数据访问效率. 另一类是从集群计算环境角度考虑, 通过对异构集群环境中的节点性能进行建模, 并分析各类 MapReduce 作业的特性以调整 MapReduce 作业中任务的分配位置, 从而提升 MapReduce 框架的计算效率. 在进行作业调度优化设计时, 不同学者聚焦的优化阶段也不尽相同, 同时针对具体场景, 在保证作业处理效率的同时, 部分学者也在考虑安全性、能耗控制等其他需求.

Wang 等人[12] 提出了一种基于数据局部性感知的放置策略, 该方案通过对集群数据的访问日志进行分析, 从而发掘数据本身的局部性和关联性, 在副本方式下实现局部关联数据在集群中的均匀分布, 在 MapReduce 作业运行过程中达到数据本地性最大化利用的效果. Bawankule 等人[13] 提出了基于历史数据的 Reduce 任务调度算法 HDRTS, 通过分析作业历史信息来确定每个节点的节点平均响应时间, 用来刻画异构集群节点的计算性能, 从而优化作业在 Reduce 阶段的计算效率. Jeyaraj 等人[14] 则是把优化 MapReduce 作业效率的目光聚焦于 Map 至 Shuffle 阶段之间, 提出了一种 TSMJS 调度方案, 其核心思想是对同一节点上的本地 Map 任务生成的中间数据进行预合并, 以减少 Reduce 阶段需要获取和合并的数据量, 从而缩短作业完成时间. Dai 等人[15] 基于真实场景下大量的 MapReduce 作业中 Map 任务相较于 Reduce 任务的数量更多[16] 的特征, 将优化重点放在了对 Map 任务的合理调度, 提出了一种具有

动态优先级的多级队列调度策略, 其核心思想是在副本存储模式下, 优先在各节点上运行数据存放在本地的 Map 任务, 并将作业对应的 Reduce 任务尽可能分配在离对应 Map 任务生成的中间结果较近的物理节点上, 以达到尽可能少的节点间通信. Maleki 等人[17]聚焦于云服务场景中, MapReduce 框架进行作业处理时面临的安全性保障和任务处理效率两方面的需求, 提出了 SPO 作业调度模型, 其核心思想是利用 MapReduce 本身两阶段的数据处理模式, 将当前系统内全部作业的 Map 任务和 Reduce 任务与集群计算资源进行映射, 使用 HEFT 算法在给定的安全性约束条件下进行任务级分配与调度, 从而尽可能提升集群中的 Map 任务或 Reduce 任务的处理效率. Chen 等人[18]则是聚焦于集群能耗和 MapReduce 性能的权衡, 在异构 Hadoop 集群中通过调度时间、能量消耗和执行成本来衡量一个机架合适的 container 数量, 而后在机架层面考虑多作业并发时的作业间资源分配, 提高集群的计算资源利用率并降低跨机架的数据传输和能耗.

1.2 存在的不足

根据上述相关工作可知, 国内外学者们通过刻画异构环境下的节点性能、调整数据放置、分析不同作业类型的资源需求偏好等各种方式来辅助 MapReduce 作业调度上的优化. 但基于底层采用纠删码存储模式的 MapReduce 作业优化的研究相对较少, 目前学界对纠删码存储的优化工作主要还是聚焦于如何提升存储侧本身的性能, 例如通过编码设计在保证低存储开销的前提下提升纠删码存储系统的数据修复性能[19,20], 引入新型硬件设备[21,22] 或调整编解码流程[23−25], 更改数据存储模式[26] 或数据更新、修复机制[27,28] 等方式来提升系统的数据更新和修复性能. 正如前文所述, 异构环境中, 当前 HDFS 的条带式纠删码在随机的数据放置下, 上层 MapReduce 作业的数据访问模式由过去的一对一变为了一对多, 在作业数据量大、任务高并发的场景下, 节点的数据访问热度差异和节点的计算负载偏斜将会大大降低 MapReduce 作业的运行效率. 此外, 相较于部分先前学者的工作, 本文考虑的混合存储模式和异构集群也更具有普遍性. 例如 Hashem 等人[29]虽然考虑到异构环境对 MapReduce 作业的影响, 但给出的调度优化仅针对单个作业的运行效率, 且未考虑底层存储模式的影响. Jiang 等人[30] 虽然提出了一种可有效缩短 MapReduce 作业运行时间的在线调度器设计, 但也仅考虑了同构的 Hadoop 集群场景. Naik 等人[31]虽然考虑了数据存储对异构集群中的 MapReduce 性能的影响, 但仍聚焦于单一的副本存储模式, 在其设计的作业调度模式中, 最大化利用数据本地性的策略无法应用于底层存储采用纠删码或混合存储模式的场景. Darrous[32] 引出了同构场景中单纠删码模式下由于数据放置的局部不均衡所产生的问题, 并给出了初步的解决方案, 但在更一般的场景中, 该方法并不能起到很好的优化作用.

1.3 混合存储模式下 MapReduce 作业优化的基本思路

真实场景下, 数据在集群中的存储模式往往是副本和多种条带宽度的纠删码混合存储. 因此, 为了使上层的计算应用更好地适应混合存储模式, 降低 MapReduce 作业中任务的长尾效应, 需要对当前异构环境中的各节点的数据访问负载 (主要是存储、网络资源) 和作业计算负载 (主要是 CPU、内存资源) 进行均衡. 而在多作业并发场景中, 如何保证各作业具有良好的执行效率也是需要解决的一个问题. 基于上述挑战, 本文通过两层设计, 在单个 MapReduce 作业运行时设计合理的作业内的数据放置和任务分配, 并在多作业并发场景下提出一种动态调节的作业调度方案, 实现对 MapReduce 框架计算效率的提升.

2 作业友好的数据放置和任务分配方案

本节将从优化当前 Hadoop 系统中默认的数据放置策略和资源管理策略等方面出发, 提出单 MapReduce 作业场景下的性能优化方案, 系统总体设计如图 2 所示. 图 2 中直观展示了优化方案的设计思路. 根据第 1 节对当前 MapReduce 作业在混合存储模式下面临的问题的分析和总结, 本文分别对存储侧 HDFS 文件系统和计算侧 YARN 框架进行部分工作流程的修改, 以实现对作业的数据访问效率和数据计算效率两方面的优化.

图 2 中的上半部分展示了本文对 HDFS 存储系统的修改设计: 考虑到异构场景中不同节点的磁盘、网络和后台数据访问负载存在区别, 因此, 通过在集群各数据存储节点中实现节点硬件信息采集以及数据访问历史负载分析的机制 (对应图 2 中 Devices info 模块), 对存储节点上的数据访问性能进行分析. 主节点则可根据各存储节点周期性更新汇报的数据访问性能情况, 按数据访问性能对节点进行分组, 在图 2 中以不同颜

色进行节点组区分, 相同节点组的节点间数据访问性能接近. 这样, 在向系统中写入以纠删码模式存储的数据时, 可以通过修改主节点的数据放置策略 (对应图中主节点的 HDPA 数据放置), 将同一条带尽可能放置在同一节点组内 (对应图中灰色的 Stripe 条带). 这样可以尽可能避免 MapReduce 作业的任务读取条带数据时, 由于节点数据访问性能差异过大产生掉队节点, 从而带来任务获取数据的效率下降问题.

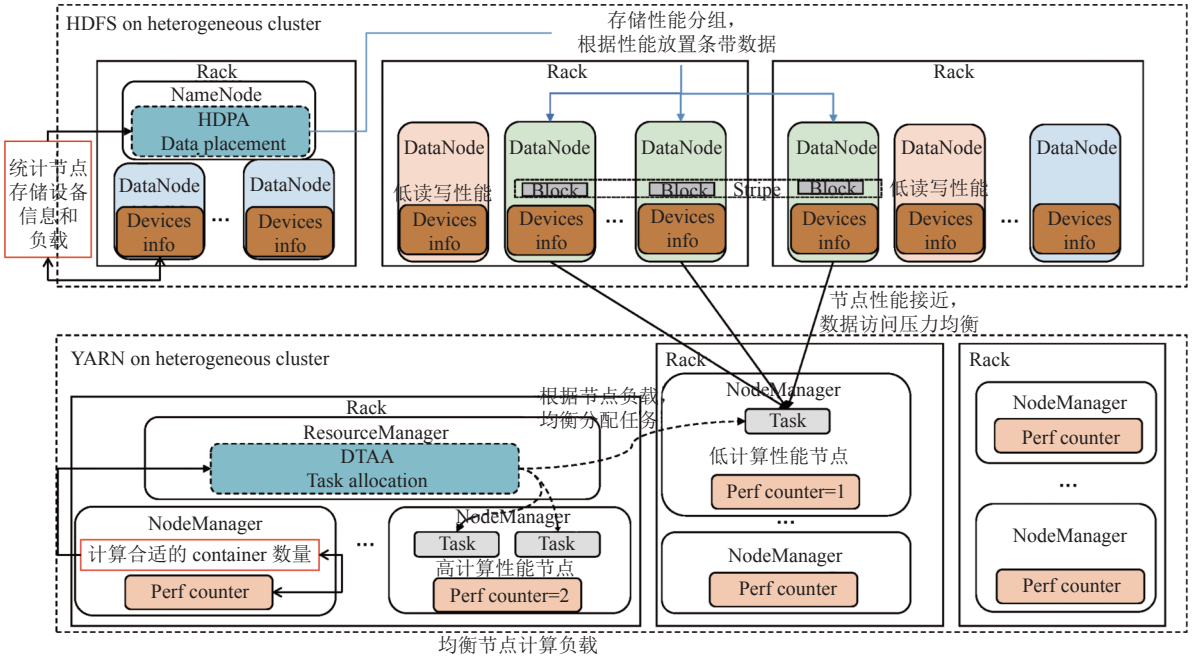


图 2 纠删码存储友好的优化方案设计

图 2 中的下半部分展示了本文对 YARN 计算框架的修改设计: 与 HDFS 中类似, 考虑异构场景中不同节点存在 CPU、内存等硬件设备性能差异和实时后台负载占用差异, 本文在每个计算节点中加入对应的计算设备信息统计机制 (对应图 2 中的 Perf counter 模块), 根据周期性更新的负载统计情况动态调节节点上的 container 计算单元数量并汇报给主节点, 以实现各节点上对任务并发度的动态控制 (对应图 2 中主节点的 DTAA 任务分配), 均衡计算负载, 避免由于部分节点上严重的计算资源竞争产生长尾任务, 从而影响作业的计算效率.

2.1 数据放置的算法设计

针对异构集群环境, 首先应对各节点影响 HDFS 数据访问的关键硬件参数进行统计, 而后根据各节点相关硬件的历史负载进行分析, 为各节点设备统计出一个常量负载因子, 基于上述两点对集群节点进行性能分组, 使同一个条带的数据块尽可能放置在同一个节点组内.

(1) 节点划分

由 MapReduce 的数据访问模式可知, 影响 Map 任务数据读取的因素主要有: 数据存放节点的磁盘顺序读写性能、网络上行带宽以及任务执行节点的网络下行带宽等. 本文基于影响任务数据读取的节点数据传输性能将所有节点进行分组, 数据传输性能将结合节点本身的硬件参数以及历史负载来衡量. 通过访问节点日志, 对节点相关硬件设备较长一段时间的历史负载进行分析并据此得出节点的长期后台数据访问负载情况, 用常量化的负载因子α来表示, 单位为数据传输速度 MB/s. 对于磁盘顺序读写, 统计节点磁盘大于 80% 时间中的负载峰值, 将其作为长期后台负载, 记作α_{diskIO}. 对于网络传输, 分别统计节点大于 80% 时间中的网络上、下行负载峰值, 将其作为网络传输的长期后台负载, 记作α_{NetUP}, $\alpha_{NetDown}$.

在获取到集群中各节点的数据传输性能情况后, 确定节点分组的阈值θ (单位为 MB/s), 如式 (1) 所示, 节点$node_i$将依据划分阈值被划分到对应的组$G_j = G(node_i)$

中, 节点组G_j的值越大, 代表了组内节点的数据访问性能越高.

$$Node_{\text{up}} = \min(R_{\text{disk}} - \alpha_{\text{diskIO}}, NetUP - \alpha_{NetUP})$$
$$Node_{\text{down}} = \min(W_{\text{disk}} - \alpha_{\text{diskIO}}, NetDown - \alpha_{NetDown})$$
$$G = \frac{\max(Node_{\text{up}}, Node_{\text{down}})}{\theta} \tag{1}$$

其中, R_{disk}、W_{disk}为磁盘的顺序读、写性能, $NetUP$、$NetDown$为节点上下行网络带宽. 通过取节点数据上传性能$Node_{\text{up}}$和数据下载性能$Node_{\text{down}}$中较大的值作为节点数据访问性能的表征, 每组内节点的数据访问性能差异不超过θ. 最终集群根据设定的阈值被分为G_{total}组. 基于上述划分方式, 进行后续的数据放置设计.

(2) 数据放置

在为条带选取放置位置时, 系统的可靠性要求也是必须满足的一个条件, 纠删码条带的放置要满足节点、机架等多级别的故障容错. 而在真实环境中单磁盘、节点以及单机架故障这类情况占到绝大多数[33], 因此, 为了更容易找到合适的条带存储节点, 同时尽量减少 MapReduce 作业运行造成的跨机架网络流量, 在符合实际需求的前提下, 保证单个机架的故障容错是足够的. 与此同时为了保证各节点在正常状态下的数据访问性能, 在选取条带放置位置时对条带的数据块和校验块进行区分, 并且在保证每个节点上数据块与校验块的相对均衡. 当集群同一节点组内无法找到足够存放完整条带的节点时, 可以考虑拆分条带, 将条带的数据块部分和校验块部分分别放置在不同的节点组中.

基于上述纠删码存储模式下的条带放置思想, 本文提出一种改进的数据放置策略 (heterogeneous-aware data placement algorithm, HDPA), 用来替代当前 HDFS 中默认的数据放置算法.

算法1. HDPA数据放置策略

```
Input: 写入文件F, 纠删码策略δ, 划分阈值, 集群各节点信息NI
1.  init: 初始化节点分组信息Node_Group = {}
2.  init: 计算写入文件的条带划分数量Stripe_Set_Compute(F,δ)
3.  for node_i in Cluster do:
4.      计算node_i的数据访问性能, 确定对应节点分组NG_tmp
5.      if NG_tmp not in Node_Group then:
6.          向Node_Group中添加新组
7.      endif
8.  end
9.  函数Refresh NG order(Node_Group)刷新节点组数据写入的优先级
10. for S_t in Stripe_Set do:
11.     for NG_i in Node_Group do:
12.         布尔变量place_flag 判断条带S_t能否写入节点组NG_i
13.         if place_flag = true then:
14.             选取节点组NG_i中合适的存储节点存储条带
15.             写入条带S_t
16.             刷新节点组数据写入的优先级
17.         else:
18.             布尔变量place_flag判断条带S_t数据块部分能否写入节点组NG_i
19.             if place_flag = true then:
20.                 选取节点组NG_i中合适的存储节点存储条带数据块
21.                 刷新节点组数据写入的优先级, 剔除为数据块分配的节点组
22.                 在剩余节点组中选取合适节点组存储条带校验块
23.                 写入条带S_t
24.                 刷新节点组数据写入的优先级
25.             endif
26.         endif
27.     end
28. end
```

如算法 1 中所示, HDPA 策略应用于 HDFS 存储系统中, 主要分为两部分: (1) 主节点周期性更新集群存储节点分组信息; (2) 文件写入时的数据放置位置选取.

首先是主节点周期性更新节点分组信息, 对应于算法 1 第 1 行的初始化, 主节点周期性初始化节点分组信息 (创建节点分组集合Node_Group), 并在后续流程中重新计算集群各节点的分组情况. 如算法 1 第 3–8 行所示, 主节点根据各存储节点汇报的数据访问性能信息和用户设置的划分阈值, 依次计算集群中各存储节点对应的分组情况. 如果计算出某节点的数据访问性能不属于当前主节点保存的任何一个已有分组, 则主节点为该节点创建一个新的分组并添加该节点. 当主节点更新完集群各节点的分组情况后, 在算法 1 第 9 行的函数中, 主节点通过统计每个节点组内的存储占用情况、数据块和校验块数量信息等, 对各节点组按存储占用率升序排序, 从而确定当发生数据写入, 主节点为条带选取放置节点时, 对节点组进行遍历的优先级顺序, 确保集群各节点组之间的存储均衡.

当有数据写入系统时, 对应于算法第 2 行的初始化, 主节点可以根据写入文件的大小、采取的存储策略来计算出将要写入的条带数量, 以便接下来为每一个纠删码条带选取放置位置 (对应算法 1 第 10–28 行的循环体). 在为每一个条带选取放置位置时, 首先按当前确定的节点组优先级进行遍历, 检查节点组能否放置当前条带, 如果当前节点组能够存放整个条带则

进行存放并更新节点组优先级 (对应算法 1 第 13–16 行). 否则进一步考虑将条带的数据块和校验块部分拆分存储 (对应算法 1 第 17–26 行的条件分支), 首先判断当前遍历的节点组能否容纳条带的数据块部分 (算法 1 第 18 行), 如果当前节点组也无法容纳条带的数据块部分, 则继续遍历下一个节点组 (对应于算法 1 第 19 行条件判断为假). 如果可以容纳, 则确定下来了条带的数据块部分存放位置, 接下来将存放条带数据块的该节点组从遍历节点组列表中剔除出去, 并重新刷新节点组优先级 (算法 1 第 21 行), 以相同的流程为条带的校验块部分选取存放的节点组 (算法 1 第 22 行). 直到条带的数据块和校验块均找到放置位置, 进行条带写入并结束当前条带选址流程. 依照上述流程直至所有条带写入完成.

通过上述 HDPA 算法, 在保证了 HDFS 集群正常读写性能和容错需求的同时, 进一步提高了系统在 MapReduce 这一应用场景下的数据读取性能.

2.2 单个 MapReduce 作业的任务分配

针对 YARN 框架当前采用的静态配置和集群的异构环境和负载变化不协调的问题, 在进行 MapReduce 作业的任务分配中, 需要解决如下两方面的问题.

(1) 计算资源的动态限制

主节点通过各从节点访问其上配置文件并向主节点发送心跳包来确定集群可用的计算资源情况. 当前 YARN 默认根据配置文件中的用户设置, 静态计算出各节点上的 container 数量并无法更改. 本文在系统默认计算出的 container 资源数量之外, 引入“弹性可用计算单元 (elastic-avail-container, EAC)”数量这一指标, 其含义是通过访问节点当前硬件负载情况, 结合硬件参数计算出在不影响节点运行效率的情况下, 节点上可用的计算资源数量, 用来实现计算资源的动态限制, EAC 的计算方式如式 (2) 所示:

$$RLB_{\mathrm{CPU}} = vCores \times \frac{Num_{\mathrm{core}} - \gamma_{\mathrm{CPUload}}}{Num_{\mathrm{core}}}$$

$$RLB_{\mathrm{MEM}} = vMEM \times \frac{RAM - \gamma_{\mathrm{RAMload}}}{RAM \times MEM_{\mathrm{task}}}$$

$$Num_{\mathrm{EAC}} = \min(RLB_{\mathrm{CPU}}, RLB_{\mathrm{MEM}}) \quad (2)$$

其中, $vCores$、$vMEM$是 YARN 框架节点配置文件设置的可用 CPU 核心数和内存分配容量; $\gamma_{\mathrm{CPUload}}$、$\gamma_{\mathrm{RAMload}}$是节点上的 CPU、内存的实时占用情况; Num_{core}、RAM是节点 CPU 的物理核心数和内存容量的硬件信息, MEM_{task}是每个任务所需的内存资源. 式 (2) 通过节点 CPU 的实时占用情况和 YARN 中的默认配置信息, 计算出为了保证节点不发生严重的 CPU 资源竞争, container 不应超过的数量RLB_{CPU}. 同理, 通过考虑节点内存的实时占用情况, 计算出为了保证节点不发生严重的内存资源竞争, container 不应超过的数量RLB_{MEM}, 并综合两者的计算结果得出最终的 EAC 数量Num_{EAC}. 根据上述 EAC 的计算公式, 可以周期性对集群各节点的任务并发度进行更新调整, 由此实现根据真实负载情况对集群计算资源的动态调节.

(2) 任务选取

在实现集群计算资源的动态限制后, 可以得知集群的两类关键信息: MapReduce 应用需要处理的数据集合以及当前集群的可用计算资源. 接下来需要解决的就是任务分配问题, 存在 3 种情况.

1) 作业对应的任务规模小, 需要分配的任务数量小于集群 EAC 总量.

2) 作业对应的任务规模超过了集群当前计算出的 EAC 总量, 但没有超过集群默认配置的 container 数量.

3) 作业对应的任务规模巨大, 需要分配的任务数量大于集群默认配置的 container 数量, 涉及节点计算资源的分配、释放与再分配.

与 HDPA 类似地, 本文提出一种动态的任务分配算法 (dynamic task allocation algorithm, DTAA), 用以解决上述两方面问题.

算法2. DTAA任务分配策略

Input: 提交作业$\mathcal{J}$, 集群各节点负载信息OL

1. init: 确定作业$\mathcal{J}$的Map, Reduce任务集信息MapTask_Set, ReduceTask_Set
2. init: 初始化各节点默认container数量$Num_{node_i_\mathrm{container}}$, 集群全部container数量$total_{\mathrm{container}}$
3. for $node_i$ in Cluster do:
4. 根据$node_i$的负载计算对应EAC数量$Num_{node_i_\mathrm{EAC}}$
5. end
6. 统计集群当前全部的EAC数量$total_{\mathrm{EAC}}$
7. if NUM(MapTask_Set) < $total_{\mathrm{EAC}}$ then:
8. 各节点按$Num_{node_i_\mathrm{EAC}}$占$total_{\mathrm{EAC}}$的比例确定Map任务的分配比例$\rho$
9. for $node_i$ in Cluster do:
10. 向$node_i$分配$Num_{node_i_\mathrm{EAC}} \times \rho$个Map任务
11. end
12. else if $total_{\mathrm{EAC}}$ < NUM(MapTask_Set)<$total_{\mathrm{container}}$ then:
13. 计算Map任务数量和集群默认container数量的比例ρ
14. for $node_i$ in Cluster do:
15. 以ρ为系数放大各节点$Num_{node_i_\mathrm{EAC}}$并分配Map任务

16. end
17. else
18. 在作业中选取$Num_{node_i_EAC}$个Map任务执行
19. while still MapTask to be allocated do:
20. 集群存在EAC被释放后分配给待处理的Map任务
21. endwhile
22. endif

如算法 2 中所示, 集群各节点定期采集自身 CPU、内存等负载情况, 根据式 (2) 结合默认配置文件计算自己的 EAC 数量, 附加在向主节点发送的心跳包信息中 (算法 2 第 2–6 行). 主节点获悉集群默认 container 数量$total_{container}$和 EAC 数量$total_{EAC}$. 当提交 MapReduce 作业后, 主节点启动应用服务进程, 后者开始计算 MapReduce 作业的资源需求并向主节点申请资源 (算法 2 第 1 行初始化). 当 MapReduce 作业的任务规模小于集群的 EAC 总数时, 主节点将按各节点的 EAC 数量$Num_{node_i_EAC}$等比例分配任务 (算法 2 第 7–11 行). 当 MapReduce 作业的任务规模超过集群 EAC 总数但未超过集群默认 container 数量时, 在单作业场景下各节点按自身 EAC 数量的相对比例进行放大 (上限为各节点默认的 container 数量), 从而将整个作业的任务尽快分配完成 (算法 2 第 12–16 行). 当任务规模已超过集群默认配置的 container 总数, 则需多次分配运行, 每一次任务分配数量均以各节点的 EAC 数量为限制, 直至所有任务分配完成 (算法 2 第 17–21 行).

通过上述 DTAA 算法, 能够有效避免在性能较弱、负载较高的节点上分配过多任务, 从而产生长尾任务影响作业的执行效率, 而在性能较高的节点上也能充分利用节点计算能力, 避免资源空闲浪费.

3 多作业并发下的 DB-Fair 调度方案设计

本节将站在多作业并发的角度, 分析当前 Hadoop 默认作业调度方案存在的不足, 并通过两个层面的优化, 来提升 MapReduce 作业并发场景下的性能, 系统总体设计如图 3 所示.

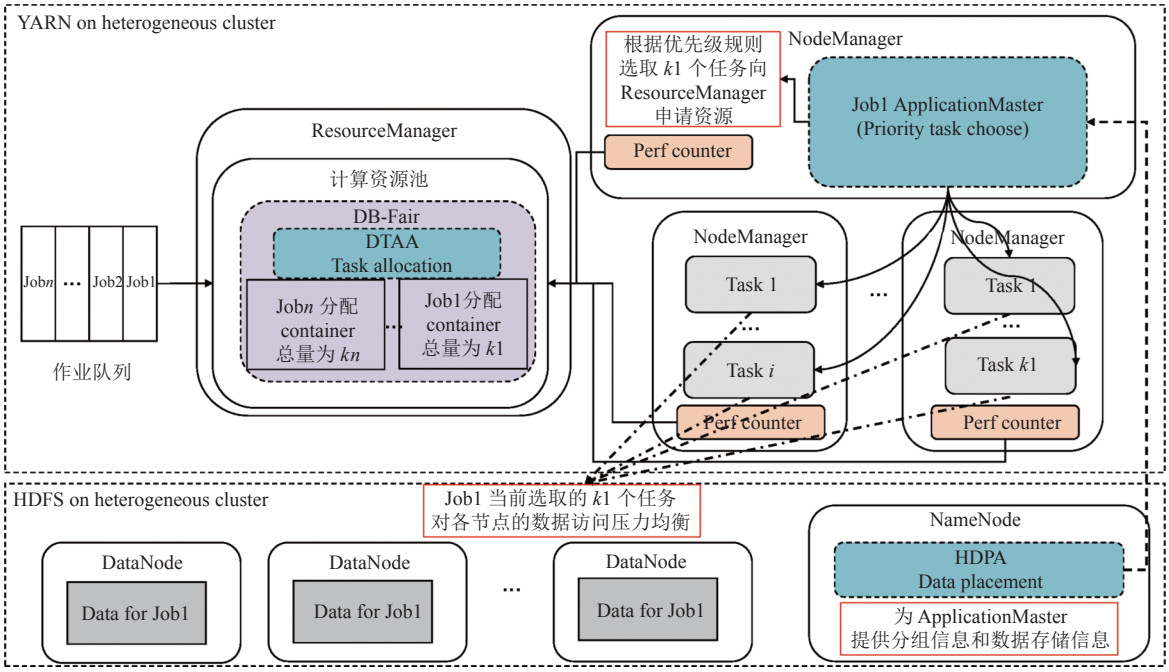


图 3 多作业并发场景下 YARN 调度优化方案设计

图 3 中展示了在多作业并发场景下, 本文对当前 Hadoop 系统中部分流程的优化设计. 首先方案在存储侧的 HDFS 中沿用图 2 中的 HDPA 数据放置策略, 在计算侧的 YARN 框架中也仍然采用周期性更新集群计算资源 (即 EAC 数量) 的设计. 当作业队列中有多个作业时, 考虑作业间资源分配和作业内任务分配两个层面的优化.

在进行作业间集群计算资源划分时, 主节点采用等分集群计算资源的方式为每个作业分配对应数量的 container 单元. 若部分作业的任务数量少于等分当前

集群可用 container 单元后可为其分配的资源量, 则其他作业可进一步等分这些空闲的计算资源. 作业间的资源划分由图 3 中主节点上的 DB-Fair 模块来实现.

在为每个作业划分好可用的计算资源后, 由每个作业自己的管理进程 (如图 3 中作业 1 的 Application-Master) 决定优先分配作业内的哪些任务. 考虑对 HDFS 存储系统的访问负载均衡, 每个作业管理进程可通过获取作业内任务处理的数据的存储情况, 并获取 HDPA 策略对当前节点分组情况的信息, 来选取一批在读取所需数据时, 对 HDFS 存储系统的访问压力均衡的任务. 如图 3 中所示, 作业 1 的管理进程选取的 $k1$ 个任务, 满足对 HDFS 各节点的数据访问压力 (具体为访问该节点上数据块的数量) 和存储节点本身的数据访问性能成正比, 以实现对存储侧的读取负载均衡.

3.1 作业并发调度的优化思路

在实际的异构 Hadoop 集群环境中, 用户往往不定时地向集群提交 MapReduce 作业, 因此系统中通常都是多作业并发的共享模式, 因此, 本文通过修改 YARN 中对共享队列中作业的调度, 和作业内任务的分配顺序两个层面去进行优化. 首先站在作业间的角度考虑, 保证资源共享的公平性和集群计算资源的负载均衡. 然后站在作业内任务分配的角度考虑, 保证作业内任务执行用时的均衡性.

(1) 各作业计算资源的分配

考虑作业提交序列$\mathcal{J}=\{J_1,J_2,\cdots,J_i,\cdots,J_n\}$, 假设该序列中各作业对应包含的任务数量为$\mathcal{T}_{\mathcal{N}}=\{TN_{J_1}, TN_{J_2},\cdots,TN_{J_i},\cdots,TN_{J_n}\}$, 若某一时间段内队列中作业为$J_m,\cdots,J_k$, 其中$1\leqslant m<k\leqslant n$, 系统在后续任务分配中应尽可能等分集群资源, 即保证为作业$J_m,\cdots,J_k$各分配$1/(k-m)$的集群资源. 这样做有利于保证短作业的快速执行, 而单纯基于作业规模或作业剩余任务数的比例来决定资源分配则会导致发生作业饥饿现象. 在此基础上, 增加两方面的修改:

1) 引入 DTAA 算法中的动态资源限制机制, 周期性地确认各节点的后台负载并计算合理的优先资源分配阈值, 并且当有新作业提交时实时更新, 以重新计算为各作业分配的计算资源量.

2) 区分作业的 Map 任务、Reduce 任务. 一般地, 在 MapReduce 作业中 Reduce 任务的数量相较于 Map 任务要小近乎一个量级, 并且不同作业对应的 Reduce 任务特征相较于 Map 任务差异更加明显 (即 Reduce 任务对于各类计算资源的敏感程度相较于 Map 任务更高), 因此, 在多作业并发场景下, 需要对 Map 任务和 Reduce 任务进行进一步区分. 这里有两方面的考虑: 一方面, 为了降低任务分配的复杂程度, 不区分不同作业的 Map 任务在集群不同节点上的分配比例; 另一方面, 注意同一作业内 Reduce 任务在集群中均匀分布, 避免引起节点间的负载倾斜. 假设当前队列中作业为$J_m,\cdots,J_k$, 其中J_l, 剩RTN_{J_l}个 Reduce 任务未分配, 假设RTN_{J_l}的值不超过集群为作业J_l分配的 container 资源数量$Num_{J_l}^{\text{container}}$, 则在新的作业提交或周期性更新 EAC 参数之前, 系统根据当前集群各节点的 EAC 数量的比例计算出各节点应该处理的作业J_l的 Reduce 任务数量$RTN_{J_l}^{N_i}$, N_i即集群第i个节点, $\sum_{N_i\in Cluster}RTN_{J_l}^{N_i}=RTN_{J_l}$. 若$RTN_{J_l}$的值超过了$Num_{J_l}^{\text{container}}$, 则从作业$J_l$的$RTN_{J_l}$个 Reduce 任务中选取$Num_{J_l}^{\text{container}}$个 Reduce 任务等待分配, $\sum_{N_i\in Cluster}RTN_{J_l}^{N_i}=Num_{J_l}^{\text{container}}$. 若集群中部分节点上的空闲 EAC 数量小于待为其分配的作业J_l的 Reduce 任务数量, 即使其他节点上存在可用 EAC 资源, 此时作业J_l的任务分配仍需要等待.

(2) 作业内任务优先级确定

动态更新集群总计算资源可以优化异构环境和负载波动对作业执行的影响, 而当作业分配的计算资源小于作业任务数量时, 优先选取作业内的哪些任务也会对作业完成产生不同影响. 多种存储模式混合场景下, 作业并发执行将会带来节点数据访问热度差异, 影响大量的 Map 任务执行效率. 这时可以考虑使用 HDPA 算法中的节点分组信息. 假设当前集群划分为 n 个节点组$\mathcal{G}=\{NG_1,\cdots,NG_i,\cdots,NG_n\}$, 相对数据访问性能比例为$P_1:\cdots:P_i:\cdots:P_n$, 当作业$J_i$进行任务选取时, 将优先以$P_1:\cdots:P_i:\cdots:P_n$的比例选取对应数据位于$NG_1,\cdots,NG_i,\cdots,NG_n$节点组的 Map 任务, 以保证接下来一段时间内数据访问负载的相对均衡.

3.2 算法流程设计

基于上述优化思路, 本文提出一种资源动态平衡的公平调度算法 (dynamic balanced fair scheduling algorithm, DB-Fair), 算法流程如算法 3.

算法3. DB-Fair作业调度策略

Input: 作业队列$\mathcal{J}$, 集群数据存储信息, 各节点硬件及负载信息

1. init: 根据DTAA中的方式计算$node_i$的EAC数量$Num_{node_i_\text{EAC}}$
2. while True do
3. if 作业队列$\mathcal{J}$更新或进行周期性负载采样then

4. 刷新各节点$Num_{node_i_EAC}$
5. end
6. if $\mathcal{J}$ not null then
/*等分集群EAC资源给作业队列$\mathcal{J}$中各作业*/
7. 为Job$i$分配$J_i^{cur}=total_{EAC}/|\mathcal{J}|$个container
8. for Job$i$ in $\mathcal{J}$ do
9. if 0 < rest MapTask < J_i^{cur} then
10. 分配集群可用container给作业Job$i$余下的Map任务
11. end
12. if rest MapTask > J_i^{cur} then
13. 根据Job$i$数据在HDPA存储策略下的存储信息, 确定任务优先级先分配J_i^{cur}个Map任务
14. end
15. if rest only ReduceTask then
16. 根据节点$Num_{node_i_EAC}$情况均匀分配Job$i$的Reduce任务
17. end
18. end
19. end
20. endwhile

如算法 3 中所示, YARN 框架中主节点统计各从的可用计算资源, 控制作业队列中的任务在当前集群中的并发度 (算法 3 第 1 行初始化). 各从节点周期性更新 EAC 数量$Num_{node_i_EAC}$, 当作业队列发生更新或主节点更新周期结束时重新计算集群 EAC 数量并重新为作业队列中的作业进行资源分配 (算法 3 第 3–5 行). 各作业对应的作业管理进程并发向主节点申请计算资源, 主节点根据作业当前剩余的作业类型进行 container 资源分配 (算法 3 第 6–19 行).

通过算法 3, 可以实现在底层采用混合存储模式时, 上层 MapReduce 计算框架在多作业并发场景下的效率提升, 在后续的实验分析中本文也将从多个角度论证方案的有效性.

4 实验结果及分析

4.1 实验环境与方案集成

本文通过在一个由 16 台服务器组成的真实的异构集群中部署 Hadoop 系统, 并对 HDFS、YARN 以及 MapReduce 框架中的部分模块进行如图 4 所示的修改以实现本文的设计方案, 节点硬件设备如表 1 所示.

集群拓扑如图 5 所示, 16 台服务节点分布在 3 个机架中, 每个机架内节点通过机架顶部的万兆交换机通信, 机架间通过一台万兆的汇聚交换机进行互联. 集群处于共享状态, 其上运行了多种常见的分布式系统及容器环境 (Ceph, Redis, Cassandra 等), 各种硬件设备具有一定的后台负载, 因此集群具有软硬件环境异构的特性.

表 1 集群服务器配置

设备	型号、性能参数
CPU	Intel(R) Xeon(R) CPU E5-2650 v4×2 NUMA
内存	DDR4 2400 MHz 16 GB×4
网卡	Intel Corporation 82599ES 10-Gigabit
HDD	SEAGATE ST1000NM0023 7200 RPM 1 TB
SSD	Samsung SSD 860 EVO 500 GB
操作系统	CentOS Linux release 7.9.2009
内核版本	Linux Kernel 3.10.0-1160.25.1.el7.x86_64

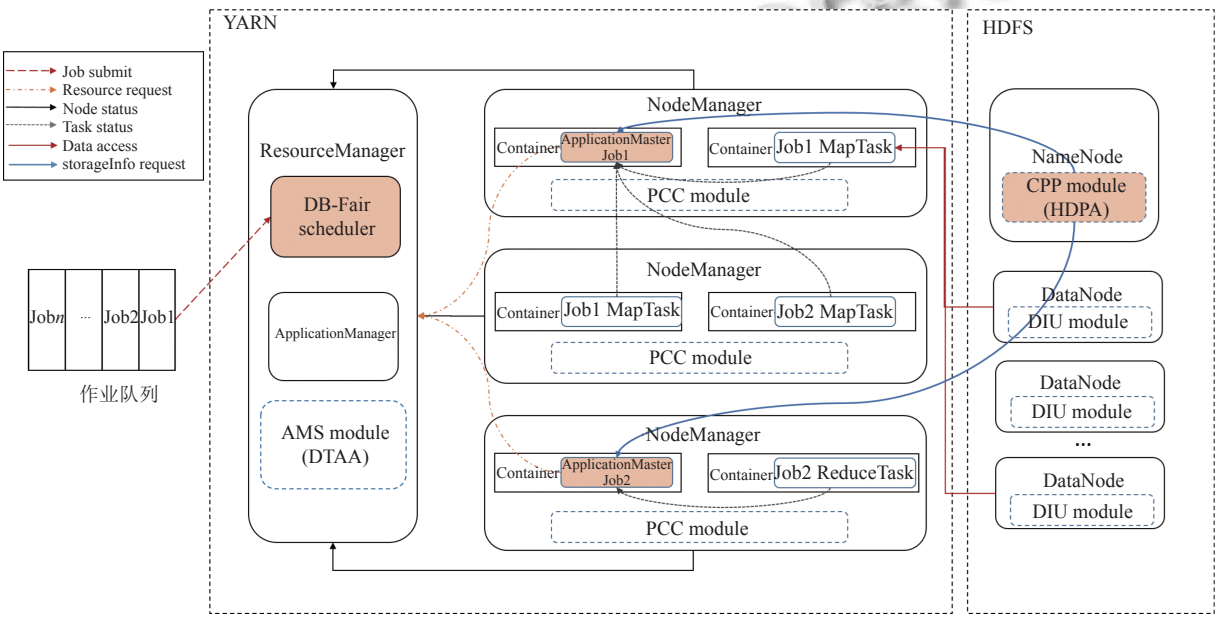


图 4 方案在 Hadoop-3.3.0 中的集成

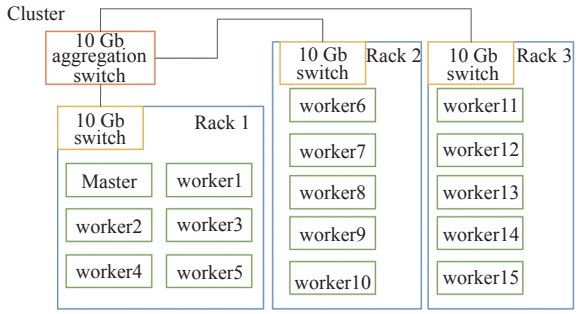


图 5 集群拓扑

4.2 实验设置

MapReduce 框架可以进行多种场景的离线批处理应用计算, 文献 [4,5,34] 对近 20 种常见的 MapReduce 应用进行了运行时资源占用分析, 给出了这些 MapReduce 应用处理过程中的关键路径, 即资源占用和运行时间占比较高的阶段. 结合 HiBench[35] 中的部分作业测试集以及 Hadoop 自带的部分测试用例, 本文选取了 11 种 MapReduce 应用作为 Benchmark 的测试集合, 如表 2 所示. 其中有工作负载集中在 Map 阶段, 对计算资源需求较高的 Wordcount、Histogram-Movies 等应用, 有在 Shuffle 阶段传输大量数据使得 Reduce 任务对内存有更高需求的 TeraSort、AdjList 等应用以及迭代式计算应用 Kmeans 和纯计算应用 MonteCarlo 等. 通过设置不同类型作业的数量以及作业提交顺序、时间间隔等参数, 本文构造出一个由 30 个作业组成的 Benchmark. 作业对应的输入数据来自真实环境或模拟生成的不同大小的数据集, 以 3 副本、RS-(3, 2)、RS-(6, 3) 以及 RS-(10, 4) 等不同存储模式随机存储. 实验的相关设置, 如作业的提交间隔、不同类型作业对应的数据集大小和类型等, 本文参考了以前学者[4,5]的配置方式, 能反映真实 Hadoop 集群中的作业运行场景. 本文通过对比 DB-Fair 作业调度策略和当前 Hadoop 中提供的两种调度策略 FIFO 先进先出调度和 Fair 公平调度策略, 验证方案的有效性.

4.3 实验结果与分析

4.3.1 评价指标选取

Benchmark 的完成速度能够最直观地体现不同调度策略的性能. 因此也是实验结果中最主要的评价指标. 除此之外, 在对比不同调度策略时, 也需要站在 MapReduce 框架性能、集群环境影响和用户服务质量等其他角度分析作业调度的影响. 因此, 本文在实验分析中除了对比不同调度策略下 Benchmark 的完成时间以外, 选取了作业任务分配、用户服务质量和节点计算资源负载等指标来验证 DB-Fair 调度的有效性.

表 2 测试程序集

测试程序	数据	Map&Reduce任务数量	关键路径	作业数
Wordcount Huge	Wikipedia 300 GB Text Data	1 140&80	Map Compute	2
Wordcount Tiny	Wikipedia 50 GB Text Data	144&30	Map Compute	3
TeraSort	30 GB Random Data of TeraGen	132&30	Shuffle、Reduce	3
Kmeans	NetFlix Movie Data 30 GB	108&0	Map Compute	5
Ranked-Inverted-Index	SequenceCount Output Data	128&15	Shuffle、Reduce	2
Grep	Wikipedia 50 GB Text Data	145&2	Map Compute	2
AdjList	30 GB Simulated Data	169&30	Shuffle	2
Histogram-Movies	NetFlix Movie Data 30 GB	108&1	Map Compute	3
Inverted-Index	Wikipedia 50 GB Text Data	144&30	Map Compute	3
SequenceCount	Wikipedia 50 GB Text Data	144&30	Balanced	2
Histogram-Ratings	NetFlix Movie Data 30 GB	108&1	Map Compute	1
MonteCarlo	None	100&1	Map Compute	2

4.3.2 Benchmark 完成用时

通过在异构集群和模拟后台负载的实验背景下测得了 FIFO、Fair 和 DB-Fair 这 3 种调度策略在给定 Benchmark 下的运行表现, 实验结果如图 6 所示. 从图 6 可以看出, FIFO 调度策略下 Benchmark 的完成时间最短, 其次是 DB-Fair, 相较于默认 Fair 调度速度提升约 17%. 值得注意的是, Benchmark 中的两个 AdjList 作业属于 Shuffle-Reduce Heavy 作业[5], 在 3 类调度策略中其 Shuffle、Reduce 阶段占据了相当长的运行时间. 由于 FIFO 按作业提交先后顺序优先为先提交的作业分配足够的计算资源, 因此在 Benchmark 设置的作业提交顺序下, FIFO 调度能够尽可能快地处理完

AdjList 的全部 Map 任务并为 AdjList 的全部 Reduce 任务分配所需 container 计算资源, 剩余大部分集群计算资源则可以分配给后面的作业, 这种 Overlapping 使得 AdjList 作业没有拖慢整个 Benchmark 的完成. 而在 Fair、DB-Fair 调度策略下 AdjList 作业的 Map、Reduce 任务需要和其他作业进行集群资源共享, 因此在作业调度后期集群中只剩下部分 AdjList 的 Reduce 作业正在处理, 拖慢了 Benchmark 的完成时间. 但是由于 FIFO 这种先来先服务原则对短作业以及较晚提交作业的处理并不友好, 本文将在第 4.3.3 节从用户感知的角度进行实验结果分析, 论证 FIFO 的不足. 而本文提出的 DB-Fair, 相较于默认的 Fair 调度策略有着较为明显的性能提升, 并且避免了 FIFO 调度存在的问题, 实验结果证明了方案的有效性.

4.3.3 任务分布的均衡性分析

进一步, 本文对比了 Benchmark 在 Fair、DB-Fair 调度模式下运行过程中, 某个 Histogram-Movies 作业对应在两种调度策略下的任务分布情况. 值得注意的是, 由于 DB-Fair 动态地调整各计算节点上的 container 数量配置, 因此任务分布的均衡性不能简单以作业对应任务在各节点上分布的数量差异来评估. 本文以不同时刻作业对各节点算力占用的比例, 即不同时刻作业在各节点上分配的任务数量除以各节点对应时刻的 EA 数量, 来刻画任务的分配情况, 实验结果如图 7 所示.

为了便于分析实验结果, 本文选取了 3 个具有代表性的节点, 观察作业中对应任务的分布情况. 从图 7 中可以明显看出作业在 Fair 调度策略下存在严重的长尾现象: 图 7(a) 的 node3 上, 作业中部分任务对节点计算资源的占用超过了 150 s, 这也意味着作业的完成时间要更久. 此外从图 7(a) 也可以看出, Fair 调度策略下作业对不同节点的计算资源占用比例差异较大, 体现出本文最开始提到的任务偏斜效应. 而在 DB-Fair 调度策略下, 如图 7(b) 所示, 任务在各节点上的完成用时和对节点计算资源的占用比例都体现出相较于 Fair 调度更加均衡的结果, 最高和最低节点算力占用的区间从 10%–40% 收窄为 17.5%–24%, 该作业的完成用时缩短了 57.9%, 验证了本文提出的方案在均衡作业内任务分布上的有效性.

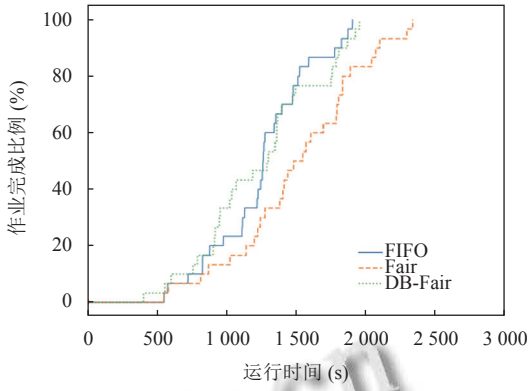
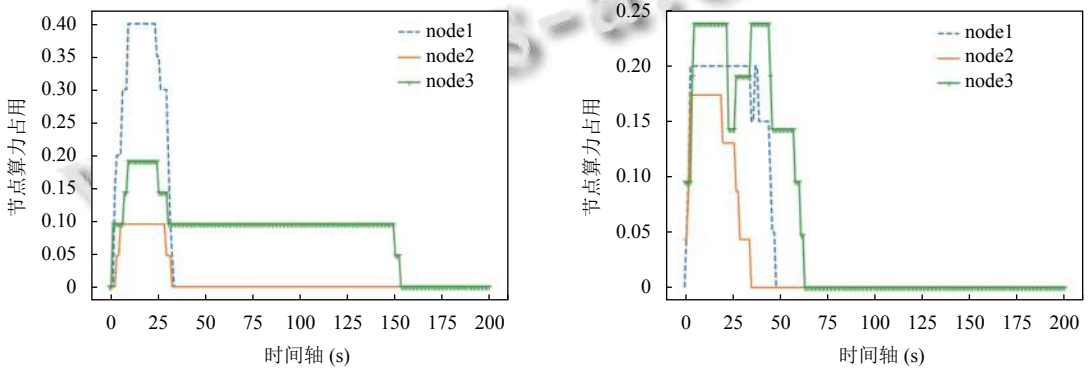


图 6 3 种调度策略下 Benchmark 完成用时



(a) Fair 调度策略 (b) DB-Fair 调度策略

图 7 Fair、DB-Fair 策略下作业的任务分布情况

4.3.4 用户感知分析

接下来本文将从用户感知的角度分析 3 种调度策略的服务性能, 站在作业提交者角度, 作业的完成速度是其唯一的评价指标. 本文以单个作业独占 YARN 集

群在 HDPA+DTAA 策略下的运行时间为基准J_{base}定义 Benchmark 中作业提交到完成用时除以J_{base}的比值作为作业的放大因子α (α越接近于 1 证明用户视角下作业执行速度越快, 用户对作业处理情况越满意), 对比在 3 种调度策略下 Benchmark 中各类型作业对应的α值, 从用户感知的角度对 3 种调度策略进行评估, 结果如图 8 所示. 从图中可以看出, FIFO 调度由于作业按提交顺序对集群资源的独占, 使得短作业必须经过长时间的等待才能运行, 具有代表性的如作业 12 为一个 Histogram-Ratings 作业, 由于前面作业 9 这一 300 GB Wordcount 作业在 FIFO 调度策略下长时间独占集群的全部计算资源而迟迟无法运行, 因此α值达到了 20, 由于作业 12 本身在独占集群资源的情况下运行时间仅为几十秒, 在 FIFO 调度下 20 倍的等待时间将会严重影响用户的实际体验, 无法满足用户需求. 而 Fair、DB-Fair 的放大因子则相对较低, 但由于默认的 Fair 调度没有考虑到集群异构情况、数据访问热度和任务类型区分, 部分作业的运行效率仍然较低, 如作业 3, 14, 23 等. 值得注意的是, 多用户共享集群时, 极端情况的发生对用户体验的影响程度要更大, 尽管 FIFO 调度下部分作业的α值要明显小于 Fair 和 DB-Fair, 但在 DB-Fair 调度策略下, 各作业的放大因子则更加均衡, 30 个作业中α的最大值为 10, 小于 FIFO 调度下的最大值 20 和 Fair 调度下的最大值 14.7, 虽然部分作业的α值相较于 FIFO 和 Fair 更大, 如作业 9, 21, 从用户服务感知的角度出发, DB-Fair 仍能体现出其自身优势.

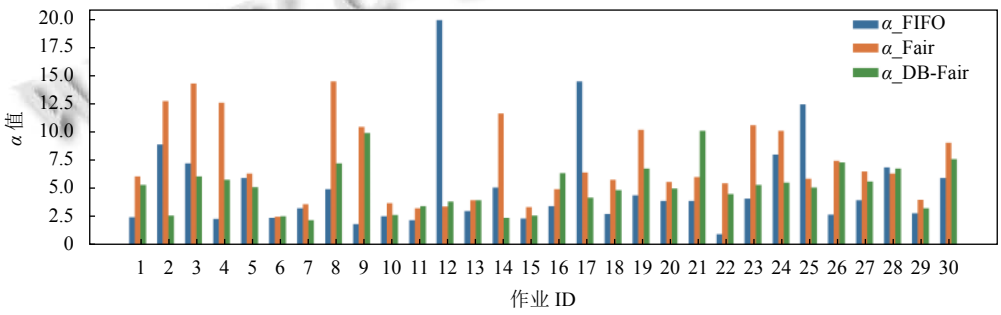


图 8 3 种调度策略下的用户感知情况

4.3.5 集群负载均衡性分析

本节将对不同调度策略下集群各节点的负载情况进行分析. 本文统计了 Benchmark 运行过程中集群各节点的 CPU 负载和内存占用的变化情况, 并展示了在 3 种调度策略下 CPU、内存负载差异最大的两个节点的具体负载信息, 实验结果如图 9, 图 10 所示.

从图 9, 图 10 可以看出, 在实验的统一设置下, 模拟集群后台负载时, Benchmark 开始运行前, 集群中 CPU 负载差异最大的两个节点的后台负载分别为 70% 和 5%, 内存占用差异最大的两个节点的后台负载分别为 60% 和 10%. 而正如前文所述, FIFO、Fair 这两种默认策略下 YARN 未考虑集群异构和运行时负载, 静态的计算资源划分和任务分配方式加剧了高负载节点的资源占用, 并且没有很好地利用低负载节点上的计算资源, 如在图 9 中, Benchmark 运行在 FIFO、Fair 调度策略下, 集群 CPU 负载最高的节点长期处于 100% 占用, 而 CPU 负载最低的节点则在 20%–60% 之间波动. 与之对应的, 在本文提出的 DB-Fair 调度策略下, Benchmark 运行期间 CPU 负载最高的节点虽然峰值也接近 100%, 但总体在 90%–100% 之间波动, 而 CPU 负载最低的节点的 CPU 利用率也得到了提升, 在 40%–80% 之间波动. 说明 DB-Fair 调度策略在一定程度上缓和了集群不同节点上的 CPU 负载差异. 同样地, 如图 10 中所示, 在 FIFO、Fair 调度策略下, Benchmark 运行期间内存占用最高的节点峰值分别达到了 98.8% 和 93.1%, 内存占用最低的节点则长期处于 40% 以下. 与之对应的, 在 DB-Fair 调度下, 高负载节点的内存占用维持在 70%–80%, 低负载节点的内存占用也在 50% 上下波动, 实现了一定程度上的负载均衡.

5 结论与展望

本文分析了当前异构集群环境下, 底层存储系统采用混合存储模式时上层 MapReduce 运行过程中面临的问题. 通过设计合适的数据放置、资源管理和作

业调度策略, 实现了多作业场景下的 MapReduce 计算效率的优化. 实验结果表明, 相较于 Hadoop 默认的调度策略, 本文的方案能在给定的 Benchmark 中提升约 17% 的性能, 并且在用户感知、资源有效利用和集群负载均衡性等方面均要优于默认调度策略. 此外还可将本文提出的方案部署在更大规模、异构情况更加复杂的集群环境中, 去验证方案的有效性并发掘新的问题. 当然, 本文仍然存在进一步的改进空间, 如对集群节点存储性能进行更细粒度的刻画, 细分不同类型 MapReduce 作业的特性等.

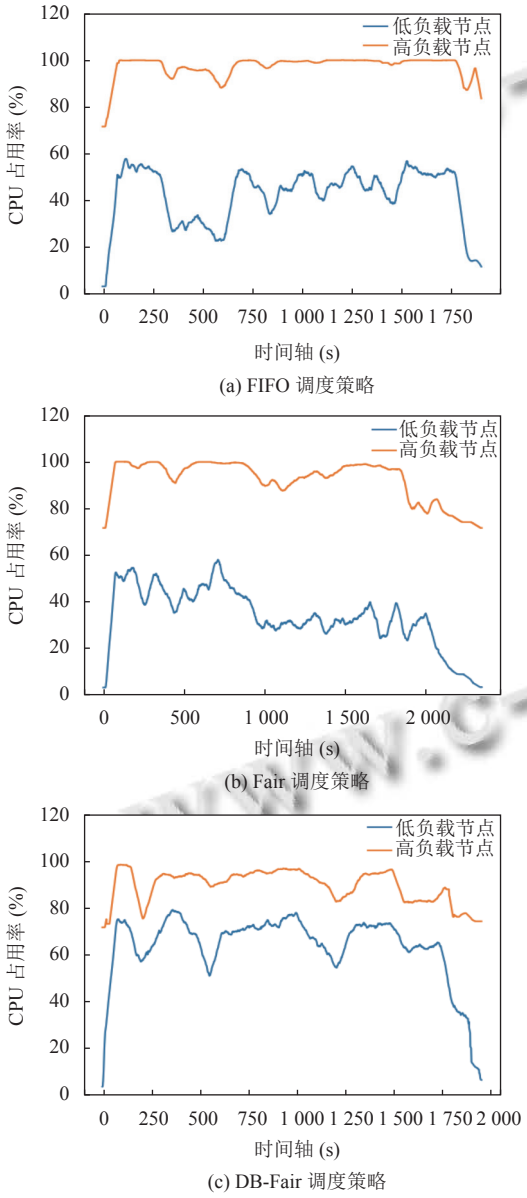


图 9 3 种调度策略下的节点 CPU 负载差异

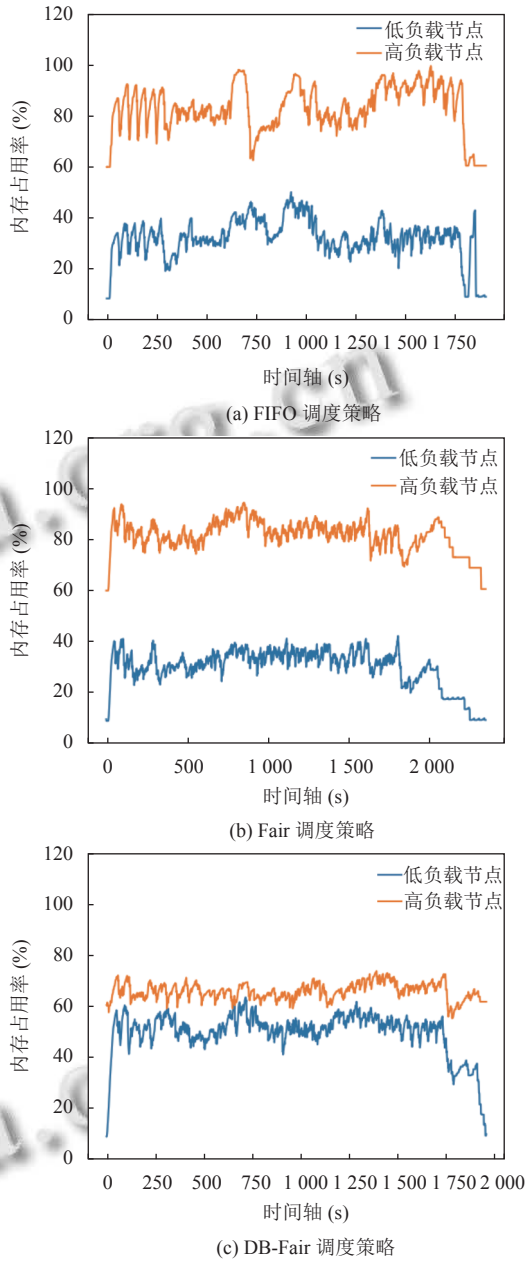


图 10 3 种调度策略下的节点内存负载差异

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. [doi: 10.1145/1327452.1327492]

2 徐鹏. Hadoop 2. X HDFS 源码剖析. 北京: 电子工业出版社, 2016. 1–25.

3 Apache. HDFS architecture. https://hadoop.apache.org/docs/r3.3.1/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html. (2021-06-15)[2022-08-03].

4 Chen YP, Ganapathi A, Griffith R, *et al*. The case for

evaluating mapreduce performance using workload suites. Proceedings of the 19th IEEE Annual International Symposium on Modelling, Analysis, and Simulation of Computer and Telecommunication Systems. Singapore: IEEE, 2011. 390–399.

5 Ahmad F, Chakradhar ST, Raghunathan A, *et al*. Tarazu: Optimizing mapReduce on heterogeneous clusters. ACM SIGARCH Computer Architecture News, 2012, 40(1): 61–74. [doi: 10.1145/2189750.2150984]

6 IDC. The digitization of the world from edge to core. https://www.seagate.com/files/www-content/our-story/trends/files/idc-seagate-dataage-whitepaper.pdf. (2018-12-04)[2022-08-03].

7 CAICT 中国信通院. 数据中心产业图谱研究报告. http://www.caict.ac.cn/kxyj/qwfb/ztbg/202201/P020220125529907466991.pdf. (2022-01-26)[2022-08-03].

8 Wikipedia. Era sure code. https://en.wikipedia.org/wiki/Era sure_code. (2022-06-26).

9 Ceph. Ceph documentation. https://docs.ceph.com/en/latest/rados/operations/erasure-code/. (2019-04-23)[2022-08-03].

10 Huang C, Simitci H, Xu YK, *et al*. Erasure coding in Windows azure storage. Proceedings of the 2012 USENIX Conference on Annual Technical Conference. Boston: USENIX Association, 2012. 15–26.

11 Muralidhar S, Lloyd W, Roy S, *et al*. F4: Facebook's warm BLOB storage system. Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation. Broomfield: USENIX Association, 2014. 383–398.

12 Wang J, Shang PJ, Yin JL. DRAW: A new data-grouping-aware data placement scheme for data intensive applications with interest locality. In: Li XL, Qiu J, eds. Cloud Computing for Data-intensive Applications. New York: Springer, 2014. 149–174.

13 Bawankule KL, Dewang RK, Singh AK. Historical data based approach to mitigate stragglers from the Reduce phase of MapReduce in a heterogeneous Hadoop cluster. Cluster Computing, 2022, 25(5): 3193–3211. [doi: 10.1007/s10586-021-03530-x]

14 Jeyaraj R, Ananthanarayana VS, Paul A. MapReduce scheduler to minimize the size of intermediate data in shuffle phase. Proceedings of the 18th IEEE/ACIS International Conference on Computer and Information Science. Beijing: IEEE, 2019. 30–34.

15 Dai XM, Bensaou B. Scheduling for response time in Hadoop MapReduce. Proceedings of the 2016 IEEE International Conference on Communications. Kuala Lumpur: IEEE, 2016. 1–6.

16 Kavulya S, Tan JQ, Gandhi R, *et al*. An analysis of traces from a production MapReduce cluster. Proceedings of the 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing. Melbourne: IEEE, 2010. 94–103.

17 Maleki N, Rahmani AM, Conti M. SPO: A secure and performance-aware optimization for MapReduce scheduling. Journal of Network and Computer Applications, 2021, 176: 102944. [doi: 10.1016/j.jnca.2020.102944]

18 Chen L, Liu ZH. Energy- and locality-efficient multi-job scheduling based on MapReduce for heterogeneous datacenter. Service Oriented Computing and Applications, 2019, 13(4): 297–308. [doi: 10.1007/s11761-019-00273-x]

19 Rashmi KV, Shah NB, Ramchandran K, *et al*. Regenerating codes for errors and erasures in distributed storage. Proceedings of the 2012 IEEE International Symposium on Information Theory. Cambridge: IEEE, 2012. 1202–1206.

20 Chen HCH, Hu YC, Lee PPC, *et al*. NCCloud: A network-coding-based storage system in a cloud-of-clouds. IEEE Transactions on Computers, 2014, 63(1): 31–44. [doi: 10.1109/TC.2013.167]

21 Shi HY, Lu XY. TriEC: Tripartite graph based erasure coding NIC offload. Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. Denver: ACM, 2019. 44.

22 Wang F, Tang YJ, Xie YW, *et al*. XORInc: Optimizing data repair and update for erasure-coded systems with XOR-based in-network computation. Proceedings of the 35th Symposium on Mass Storage Systems and Technologies. Santa Clara: IEEE, 2019. 244–256.

23 Mitra S, Panta R, Ra MR, *et al*. Partial-parallel-repair (PPR): A distributed technique for repairing erasure coded storage. Proceedings of the 11th European Conference on Computer Systems. London: ACM, 2016. 30.

24 Li XL, Yang ZR, Li JH, *et al*. Repair pipelining for erasure-coded storage: Algorithms and evaluation. ACM Transactions on Storage, 2021, 17(2): 13.

25 Xu LL, Lyu M, Li QL, *et al*. SelectiveEC: Towards balanced recovery load on erasure-coded storage systems. IEEE Transactions on Parallel and Distributed Systems, 2022, 33(10): 2386–2400. [doi: 10.1109/TPDS.2021.3129973]

26 Chan JCW, Ding Q, Lee PPC, *et al*. Parity logging with reserved space: Towards efficient updates and recovery in erasure-coded clustered storage. Proceedings of the 12th USENIX Conference on File and Storage Technologies. Santa Clara: USENIX Association, 2014. 163–176.

27 Rashmi KV, Shah NB, Gu DK, *et al*. A "hitchhiker's" guide to fast and efficient data reconstruction in erasure-coded data centers. Proceedings of the 2014 ACM Conference on SIGCOMM. Chicago: ACM, 2014. 331–342.

28 Rawat AS, Vishwanath S, Bhowmick A, *et al*. Update

efficient codes for distributed storage. Proceedings of the 2011 IEEE International Symposium on Information Theory. St. Petersburg: IEEE, 2011. 1457–1461.

29 Hashem IAT, Anuar NB, Marjani M, *et al*. Multi-objective scheduling of MapReduce jobs in big data processing. Multimedia Tools and Applications, 2018, 77(8): 9979–9994. [doi: 10.1007/s11042-017-4685-y]

30 Jiang YW, Zhou P, Cheng TCE, *et al*. Optimal online algorithms for MapReduce scheduling on two uniform machines. Optimization Letters, 2019, 13(7): 1663–1676. [doi: 10.1007/s11590-018-01384-8]

31 Naik NS, Negi A, Br TB, *et al*. A data locality based scheduler to enhance MapReduce performance in heterogeneous environments. Future Generation Computer Systems, 2019, 90: 423–434. [doi: 10.1016/j.future.2018.07.043]

32 Darrous J. Scalable and efficient data management in distributed clouds: Service provisioning and data processing [Ph.D. thesis]. Lyon: Université de Lyon, 2019.

33 Ford D, Labelle F, Popovici FI, *et al*. Availability in globally distributed storage systems. Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation. Vancouver: USENIX Association, 2010. 61–74.

34 Ahmad F, Chakradhar ST, Raghunathan A, *et al*. ShuffleWatcher: Shuffle-aware scheduling in multi-tenant MapReduce clusters. Proceedings of the 2014 USENIX Conference on Annual Technical Conference. Philadelphia: USENIX Association, 2014. 1–12.

35 Intel. Intel-bigdata/HiBench. https://github.com/Intel-bigdata/HiBench. (2020-06-20)[2022-08-03].

(校对责编: 孙君艳)