网络出版时间：2017-10-16 11:16:31
网络出版地址：http://kns.cnki.net/kcms/detail/11.2127.TP.20171016.1116.002.html

一种 Spark 集群下的 shuffle 优化机制

熊安萍[1,2]，夏玉冲[1]，杨方方[1]

XIONG Anping[1,2], XIA Yuchong[1], YANG Fangfang[1]

1. 重庆邮电大学 计算机科学与技术学院, 重庆 400065
2. 重庆市移动互联网数据应用工程技术研究中心, 重庆 400065
1. School of computer science and technology, Chongqing University of Posts and Telecommunications, Chongqing 400065, China
2. Chongqing Engineering Research Center of Mobile Internet Data Application Chongqing 400065, China

XIONG Anping, XIA Yuchong, YANG Fangfang. Shuffle Optimization for Spark Cluster. Computer Engineering and Applications, 2000, 0(0): 1-000.

Abstract：Spark is a distributed processing framework based on memory. The large amounts of data which generated by the shuffle process deeply affect the network transmission, has become one of the main bottleneck of the Spark performance. In order to solve the problem of unbalanced data distribution resulting in the I/O load imbalance in different nodes, a restart policy based on task local level is designed. Finally, the optimization mechanism is verified by experiments, which can reduce the execution time of task and improve the efficiency of shuffle process.

Key words：spark; shuffle; data transfer; locality; schedule strategy

摘 要：Spark 是基于内存的分布式数据处理框架，其 shuffle 过程中大量数据需要通过网络传输，已成为 Spark 最主要的瓶颈之一。针对 shuffle 过程中存在的数据分布不均、造成不同节点网络 I/O 负载不均的问题，设计了基于 task 本地性等级的重启策略，进一步提出了均衡的调度策略来平衡各节点的网络 I/O 负载。最后通过实验验证了优化机制能够减少计算任务的执行时间，提升整个 shuffle 过程的执行效率。

关键词：spark；shuffle；数据传输；本地性；调度策略
文献标志码：A **中图分类号**：**TP311** doi：10.3778/j.issn.1002-8331.1701-0238

1 引言

Apache Spark[1]是当前流行的并行计算框架之一，具有快速、通用、简单等特点。由于采用基于内存的集群计算，所以 Spark 的运算速度比基于磁盘的 Hadoop 快 100 倍[2]。Spark 针对 MapReduce[3,4]计算框架在迭代式机器学习算法和交互式数据挖掘等应用方面的低效率，提出了一种分布式内存抽象，称为弹性分布式数据集(RDD)[1]。它既保留了 MapReduce 的可扩展性、容错性、兼容性，又弥补了 MapReduce 的不足。同其它并行计算框架对比，Spark 对于大规模的数据处理具有更加高效的特性。

基金项目：重庆邮电大学博士启动基金(No. A2015-17)。

作者简介：熊安萍(1970—)，女，博士，教授，CCF 会员，研究领域为研究方向为高性能计算、海量信息处理、操作系统内核，E-mail：xiongan@cqupt.edu.cn；夏玉冲(1991—)，男，硕士研究生，硕士，研究领域为高性能计算，E-mail: 357917886@qq.com；杨方方(1991—)，男，硕士研究生，硕士，研究领域为高性能计算。

随着Spark在多个业务领域的应用越来越为广泛，框架本身存在的不足和瓶颈也逐步凸显。其中shuffle 过程中过多的临时文件、数据集分布不均、大量的数据拉取等问题，较大地影响了 Spark 执行效率。因此，shuffle 机制的优化成为 Spark 计算框架下急需解决的问题。

2 相关工作

针对 Spark 框架的 shuffle 机制已有大量研究。为解决 shuffle 过程中存在过多小文件、严重影响集群的磁盘 I/O 效率问题，文献[5]中提出 Shuffle File Consolidation 方法通过合并临时文件，减少 shuffle 过程中产生的大量临时文件；另外，为了解决由于分区策略中分区不均匀问题，Benjamin Gufler[6]等提出了 Fine Partitioning 和 Dynamic Fragmentation 算法，对 Map task 的输出进行重新分区，使得每一个 Reduce task 处理的分区大致相同，缓解了数据倾斜问题；陈英芝等在文献[7]中，提出了基于溢出历史的自适应内存调度算法，解决 shuffle 过程当中因 task 对内存的需求不均衡而造成 shuffle 过程当中出现的内存溢出错误；文献[8][9]分别提出了 SCID(Splitting and Combination algorithm)和 SASM (Spark Adaptive Skew Mitigation)自适应的策略，通过数据迁移，缓解了数据倾斜(Data-Skew)问题；Mosharaf Chowdhury[10]等提出 Orchestra 的调度框架，提出全局的网络调度策略，提高 shuffle 过程中网络的利用率；冯琳等在文献[11]实现了 RDD 的自动缓存策略以及内存替换策略的优化，提高了任务在有限资源情况下的运行效率；陈侨安等在文献[12]提出基于历史运行数据的优化方案，根据任务特征的相似性对相似的任务进行优化配置。

以上研究大都专注于调度和内存的优化，而忽略了数据拉取带来的网络对 shuffle 效率的影响。本文主要针对 shuffle 过程当中，数据过于集中，造成不同节点网络负载不同的问题，通过额外启动少量的 Map task 的方法，使得 Map task 的输出，能够更好、更均匀的分布于集群，并通过均衡调度策略来平衡各节点的网络负载。

3 Spark 的 Shuffle 机制

在 MapReduce 的计算框架中，shuffle 过程是连接 Map 和 Reduce 之间的桥梁，介于 Map 和 Reduce 之间。Map 的输出需经过 shuffle 过程传输到 Reduce，shuffle 的性能高低直接影响了整个任务的执行时间。Spark 作为 MapReduce 计算框架的一种实现，也实现了 shuffle 的逻辑，Spark 框架中 shuffle 过程如图 1 所示，将 Map 端划分数据、持久化数据的过程称为 shuffle write，而将 Reduce 端读入数据、聚合数据的过程称为 shuffle fetch：

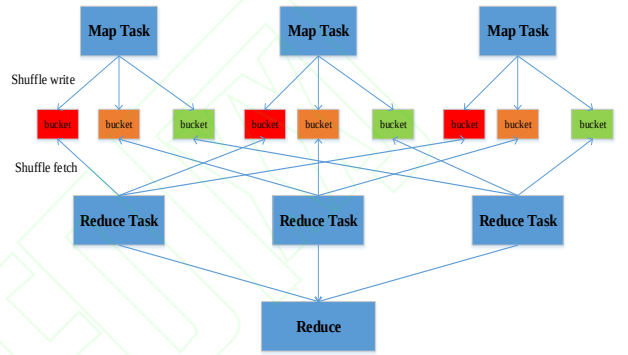

图 1 Spark shuffle 过程

首先，每一个 Map task 会根据 Reduce task 的个数创建相应的 bucket，bucket 的个数等于 M*R(M 为 Map task 的个数，R 为 Reduce task 的个数)；其次，Map task 的输出结果会根据设置的分区算法(默认为 Hash 分区)，放到对应的 bucket 中去；最后，当 Reduce task 启动时，会根据 task 所依赖的 Map task 的 id，从远端或者本地的 bucket 中拉取所需的数据，随后对数据进行 Reduce 操作。

不同于 Hadoop 框架[13]中的 shuffle 过程，在 Spark 框架当中，Reduce task 的启动需要等待所有的 Map task 执行完成。由图 1 可知，每一个 Reduce task 处理的数据集合的大小是由分区函数决定的。不合理的分区方法，可能造成不同 Reduce task 所处理的数据集合的大小存在较大的差异，造成数据倾斜，降低框架的执行效率。

在 Spark 的调度当中，使用延迟调度策略[14]，会根据上一个任务成功提交的时间，自动调整自身的本地性匹配策略。由于“传递数据不如传输计算”，故在调度的过程当中，会优先在数据存储的位置启动执行任务的 task，提高任务的本地性等级[15]。如图 1 所示，在 shuffle 过程当中，会伴随着大量的网络 I/O，如果 Map task 的输出结果在集群当中分布不合理，会造成部分 Map 节点负载过重，影响 shuffle 效率。

4 优化的 Shuffle 机制

4.1 问题分析

由第 3 节分析可知，在 shuffle fetch 过程当中，需要到对应的机器上拉取数据，所有数据的位置信息由 BlockManager 统一管理。

Shuffle 过程的执行效率直接决定了整个任务的执行时间。根据文献[10]在 Spark 集群中追踪的数据，计算分析绘制 CDF 图如图 2 所示。由图 2 可知，shuffle 过程当中所耗费的时间，平均占整个任务所用时间的 33%。更有甚者，在有些任务当中 shuffle 时间占整个任务时间的 50%以上。因此可见 shuffle 过程是整个任务执行效率的最主要的瓶颈之一。

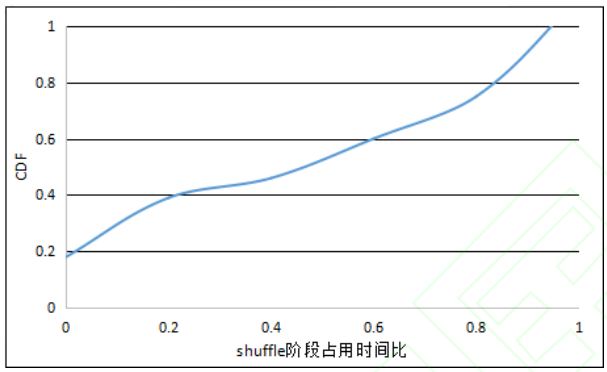

图 2 Shuffle 阶段占用时间比

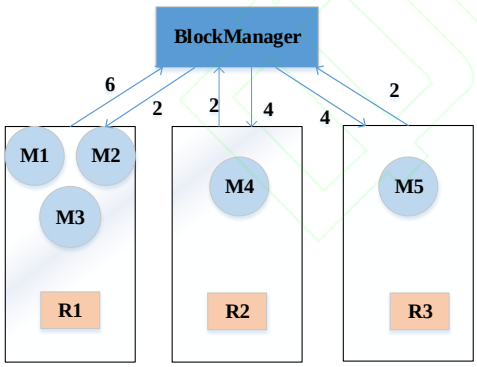

图 3 数据在不同机架之间的传输

图 3 显示了 shuffle 过程当中各节点网络负载不同的情况。由于 Map task 分布集中，造成大量 Reduce task 需要到左侧机器拉取数据(R2，R3 各自拉取 3 块数据)，造成左侧机器网络 I/O 负载过重，拥堵的网络资源会极大的限制网络利用率。因此合理的调整 Map task 的输出数据在集群中的分布，解决 shuffle 过程中数据过于集中的问题，能够显著的提高 shuffle 过程的执行效率。

造成各节点网络 I/O 负载不同的主要原因在于，Map task 分布集中在少量机器上，使得 Map task 输出数据不能够均匀分布在集群当中。可以通过合理安排 Map task 和 Reduce task 的分布来解决数据少量节点网络 I/O 拥堵的问题，解决方式如图 4 所示：

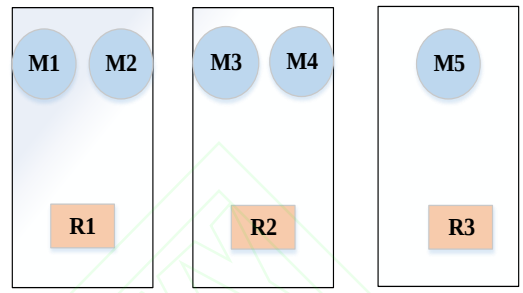

图 4 数据均匀分布

但是，此种方法在 Spark 的框架中不可行，主要原因如下：首先，在 Spark 的调度策略当中，Map task 的位置受限于 task 的本地性；其次，在 Map 阶段执行完毕之前，很难预测 Reduce task 在集群中的分布情况。另外，Spark 的调度器在分配 Reduce task 的时候，没有考虑 Reduce task 和 Map task 的依赖关系，而是随机调度 Reduce task。

4.2 Shuffle Fetch 优化

相关文献表明[10,16]，合理的利用网络资源能够很好的提升 shuffle 效率。本节提出优化机制包括两部分，首先通过启动少量额外的 Map task 来将 task 的输出均匀的分散到集群当中，其次，通过合理的选择策略来均衡各节点的网络 I/O 负载。

1、Map task 启动策略

shuffle 阶段大量数据通过网络拉取，中间数据过于集中会造成部分节点拥堵，降低网络的利用率。因此通过将中间数据均匀分散到集群的机器上，平衡各节点的网络负载能够有效的提升 shuffle 执行效率。

为了表明集群中各节点的负载情况，我们定义集群的网络负载倾斜比：集群当中节点最高的网络链接数和最低链接数之比。如图 3 所示左侧节点有六块数据需要通过网络传输，因此网络链接数为 6；右侧节点的网络链接数为 2，故集群的网络负载倾斜比为 3。由文献[17]在 Facebook 集群中的数据整理得图 5，由图可得在 Job 大小为 50-150 tasks 时，有超过一半任务网络倾斜率大于 5。由此可见网络负载的不均衡，存在于绝大多数的任务之中，且极大的限制了 shuffle 过程的效率。

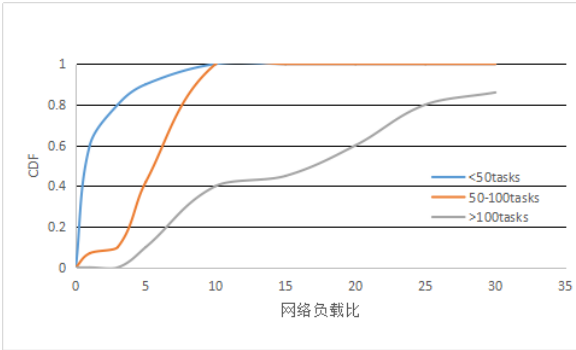

图 5 节点的负载倾斜比

为了将 Map 的输出更加均匀地分布在集群上，平衡 Map 节点负载，在启动 Map task 时，额外启动少量的 Map task 来分散中间数据在集群中的分布。少量额外的 Map task 可以增加数据在集群中均匀分布的概率。

首先需要确定额外启动的 task 的个数。如果启动过多的 Map task 势必会造成资源的竞争，多出的 task 会抢占系统资源，增加系统开销，使得整体性能下降。另外，对于本地性等级较高的 task 则不需额外启动，这样会使得新启动的 task 没有好的本地性，由于数据的 fetch 需要等到全部的 Map task 完成之后才开始，此种情况下启动的 task 会增加 Map 阶段的完成时间。

因此，本文给出需要额外启动 task 的条件是：如果同一个 TaskSet 在当前的 Executor 上已经有启动的 task，且当前调度的 task 的本地性等级为 ANY(即数据需要通过网络传输)。此时在不同的机架上另外启动处理该分区的 task(使用 index 将此两个 task 标志为同一个 task，使用 rack 来标志 task 所在的机架)。额外启动本地性等级为 ANY 的 task 可以大大减少需要额外启动的 task 数量，而且对 Map 的执行时间没有过多的影响(因为两个 task 的本地性等级都为 ANY)，算法如图 6 所示。

```
算法 1：
(1)  Input: TaskSet,WorkerOffer//任务集，可使用的 Executor 序列
(2)  Output: Map[Worker,Array[TaskDescription]] //返回调度结果
(3)  FOR(maxLocality <- taskSet.myLocalityLevels)//本地性策略由高到底
(4)    FOR(i <- 0 until WorkerOffer.size){//遍历所有 Executor
(5)
(6)      FOR(task <- TaskSet.resourceOffer(execId, host , maxLocality))//根据本
地性等级将 task 分配给 worker
(7)        Mp[WorkerOffer(i),task]//记录分配结果
(8)        IF(maxLocality == ANY)//本地性等级为 ANY 需要另外启动 task
(9)        val t = Select(WorkerOffer,i)//选择和当前被调度的 worker 不同机架
的 worker
(10)       Mp[WorkerOffer(t),task]//启动另外 task
(11)   }
(12)   RETURN Mp;
(13)   def Select(WorkerOffer,i)
(14)    FOR(t <- 0 until WorkerOffer.size)
(15)      IF(WorkerOffer(t).rack != WorkerOffer(i).rack)
(16)        RETURN t
```

图 6 task 启动策略

2、均衡调度策略

在以上数据分布的解决方案当中，输出的中间结果多于实际需要，合理的调度策略可以保证任务结果的一致性，同时能够平衡节点的网络 I/O 负载。均衡的调度策略思路如下：首先，将所有的 Map task 按照 index 排序，index 标识两个 task 是否处理同一分区，排序后可保障随后的选择过程中所有分区能被选择，从而保持任务的一致性；其次，若 task 处理分区相同，可根据 rack 中 task 的个数判定 rack 的负载，并选择负载较小的 rack，以可保障各节点负载的均衡。算法如图 7 所示。

```
算法 2：
(1)  Input:Maptask 列表
(2)  Output:M 个 task 的输出数据
(3)  MtaskCount = map()
(4)  FOR (task<-Maptask)
(5)    MtaskCount[task.index]++;
//对 task 列表按照 index 升序排序过 index 相同则按照 rack 的数目
进行降序排序
(6)  Maptask.sort()
(7)  FOR (i<-0 until Maptask.size())
(8)    IF(Maptask[i].index == Maptask[i+1].index)//处理分区相同
(9)      Maptask[i+1].remove();//选择负载较轻的
(10) RETURN Maptask
```

图 7 均衡调度策略

5 实验结果与分析

5.1 实验环境

本文搭建了 6 个节点组成的服务集群，集群的运行模式为 Standalone 模式，其中一台为 Master 节点，另外 5 台为 Slave 节点。其中节点的操作系统为 CentOS release 6.8，处理器的参数为 16 Intel(R) Xeon(R) CPU E5620@ 2.40GHz，4 核，主频为 16GHZ，内存大小为 24GB，节点之间通过千兆以太网连接。本文使用 Spark1.5.0 作为实验的分布式平台，Hadoop2.6 中的 YARN 作为分布式集群资源的调度器，HDFS 作为分布式存储系统。

5.2 实验结果与分析

实验中，采用不同大小的 Job size(Job 中 task 的个数)进行对比实验，比对 Job 在本文的均衡调度策略(Balanced –schedule)和原有调度策略(Normal-schedule)下的完成时间，以及 Job 当中 shuffle 阶段的时间，结果分别如图 8、图 9 所示。

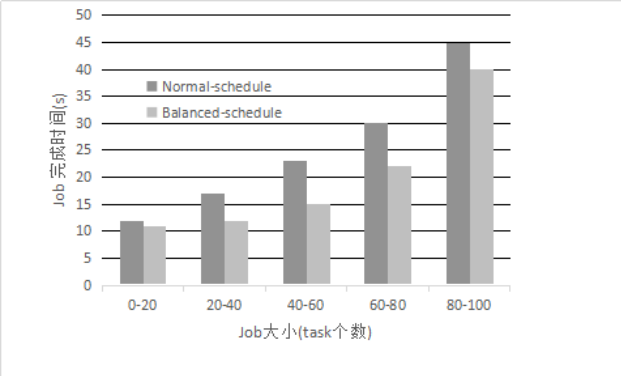

图 8　Job 完成时间对比

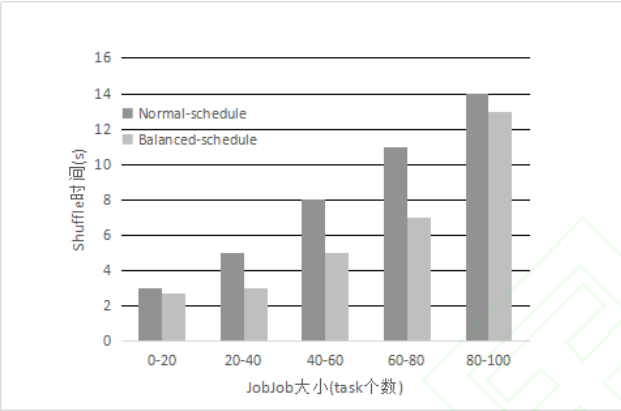

图 9　Shuffle 运行时间对比

从实验结果看出，本文 Balanced–schedule 策略，能够显著减少 shuffle 时间，从而提高整个任务执行时间。观察图 7、图 8 的变化趋势可知，对于中小型任务(20<Job size<80)，本文调度策略能够较大提升 shuffle 效率和任务完成时间。由表 1 可见，本文的优化策略在实际的执行过程当中，只启动了少量的 Map task。

表 1　Map 阶段额外启动的 task 个数

Job 大小	0-20	20-40	40-60	60-80	80-100
额外 tasks	0-2	0-5	3-7	5-10	8-12

综合图 8、图 9 和表 1 可得，随着任务量的增加，shuffle 时间会增加且在 Job size 较大的时候，由于 task 的增多，更多 task 的本地性等级会降低。按照本文的调度策略在 Map 阶段会有更多的 task 被启动。由于在 Spark 框架当中，采用延迟调度策略。因此在 Job size 变大时，Map 阶段有更多的 task 需要执行完毕，延迟了 shuffle fetch 的启动时间。由此可知，影响本文优化策略的主要因素为 Map task 的本地性等级。随着 Map task 本地性等级的降低，需要重启的 task 个数增多，shuffle 时间变长。

实验结果表明，本文优化机制能够在启动少量的 Map task 的情况下较好地提升 shuffle 执行效率，减少任务的执行时间。

6　总结

本文提出了基于 task 本地性重启策略的 shuffle 优化机制，通过均衡的调度策略平衡网络 I/O 负载，同时，保证了任务的一致性。实验表明，本文的优化机制，能够较好提升 shuffle 阶段的执行效率。在接下来的研究当中，将注重研究慢节点问题，因为额外启动的 task 可能延长 Map 阶段的执行时间。另外，根据在 Facebook 的一项统计[18]发现，即使在集群中多数机器是空闲的情况下，Spark 的实际的执行过程中，即便在集群相对空闲的情况下，只有不足 60%的 task 具有较高的本地性等级，进一步的工作需要提升 task 的本地性，减少任务的执行时间。

参考文献：

[1] 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 conference on Networked Systems Design and Implementation. USENIX Association, 2012: 2-2.

[2] Apache Spark[EB/OL]. https://spark.apache.org/

[3] Chen Q, Liu C, Xiao Z. Improving mapreduce performance using smart speculative execution strategy[J]. IEEE Transactions on Computers, 2014, 63(4): 954-967.

[4] Gunarathne T, Zhang B, Wu T L, et al. Scalable parallel computing on clouds using Twister4Azure iterative MapReduce[J]. Future Generation Computer Systems, 2013, 29(4): 1035-1048.

[5] Davidson A, Or A. Optimizing shuffle performance in spark[J]. University of California, Berkeley-Department of Electrical Engineering and Computer Sciences, Tech. Rep, 2013.

[6] Benjamin G, Nikolaus A, Angelika R, et al. Handling data skew in mapreduce[C]//Proc. Int. Conf. Cloud Comput. Serv. Sci. 2011: 574-583.

[7] 陈英芝. Spark Shuffle 的内存调度算法分析及优化[D]. 浙江大学, 2016.

[8] Tang Z, Zhang X, Li K, et al. An intermediate data placement algorithm for load balancing in Spark computing environment[J]. Future Generation Computer Systems, 2016.

[9] Yu J, Chen H, Hu F. SASM: Improving spark performance with Adaptive Skew Mitigation[C]//Progress in Informatics and Computing (PIC), 2015 IEEE International Conference on. IEEE, 2015: 102-107.

[10] Chowdhury M, Zaharia M, Ma J, et al. Managing data transfers in computer clusters with orchestra[C]//ACM SIGCOMM Computer Communication Review. ACM, 2011, 41(4): 98-109.

[11] 冯琳. 集群计算引擎 Spark 中的内存优化研究与实现[D].

清华大学, 2013.
[12] 陈侨安, 李峰, 曹越,等. 基于运行数据分析的 Spark 任务参数优化[J]. 计算机工程与科学, 2016, 38(1):11-19.
[13] Dean J, Ghemawat S. MapReduce: simplified data processing on large clusters[J]. Communications of the ACM, 2008, 51(1): 107-113.
[14] Zaharia M, Borthakur D, Sen Sarma J, et al. Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling[C]//Proceedings of the 5th European conference on Computer systems. ACM, 2010: 265-278.
[15] Ananthanarayanan G, Agarwal S, Kandula S, et al. Scarlett: coping with skewed content popularity in mapreduce clusters[C]//Proceedings of the sixth conference on Computer systems. ACM, 2011: 287-300.
[16] Ananthanarayanan G, Kandula S, Greenberg A G, et al. Reining in the Outliers in Map-Reduce Clusters using Mantri[C]//OSDI. 2010, 10(1): 24.
[17] Chen Y, Alspaugh S, Katz R. Interactive analytical processing in big data systems: a cross-industry study of MapReduce workloads[J]. Proceedings of the Vldb Endowment, 2012, 5(12):1802-1813.
[18] Venkataraman S, Panda A, Ananthanarayanan G, et al. The power of choice in data-aware cluster scheduling[C]//11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14). 2014: 301-316.