LPW: an efficient data-aware cache replacement strategy for Apache Spark

Hui LI, Shuping JI, Hua ZHONG, Wei WANG, Lijie XU,
Zhen TANG, Jun WEI & Tao HUANG1

摘要

缓存是流行的分布式大数据处理框架 Spark 最重要的技术之一。对于这种基于内存计算来支持各种应用的大数据并行计算框架来说，由于内存大小的限制，不可能缓存每个中间结果。缓存应用程序编程接口（API）使用的随意性、应用特性的多样性以及内存资源的可变性对实现高系统执行性能构成了挑战。低效的缓存替换策略可能会导致不同的性能问题，例如应用程序执行时间长、内存利用率低、替换频率高，甚至内存不足导致程序执行失败。作者提出了最小分区权重 LPW 缓存替换策略，在典型工作负载下大大减少了任务的执行时间。

关键词： Spark；内存；缓存替换；最小分区权重；数据感知

1 引言

近年来，大数据在各个领域得到广泛应用。同时，它也为社会带来了真正的价值。为了适应不同的大数据处理应用场景，人们开发了许多大数据处理框架，包括但不限于 Apache-Hadoop [2]、Tachyon、ApacheTez、Storm、GridGain 和 Spark。其中，Spark 因其高性能、良好的可扩展性和用户友好的界面而成为最流行的框架之一。Spark 是一种分布式内存计算解决方案。与基于磁盘的解决方案（例如 Apache Hadoop）相比，Spark 速度要快得多。这种差异很大程度上是因为 Spark 可以将中间计算结果缓存在内存中并消除频繁的磁盘访问和网络传输。这说明缓存机制对于提高 Spark 的性能起到了重要的作用。简而言之，Spark 用户通常会尝试将数据缓存在内存中以供重复使用，以加快任务执行速度。很多情况下，可能会缓存过多的冗余数据，导致内存逐渐耗尽。当内存不够的时候，Spark 的缓存替换机制 [1]，具体来说就是 LRU 策略开始工作。这个过程类似于 Java 虚拟机中的垃圾收集。然而，Spark 的问题更为复杂。因为 Spark 不仅需要收集未使用的内存，还需要收集将来有可能重用的数据。缓存使用的随机性给数据分析带来了混乱的局面。一个挑战是我们无法轻易知道哪些缓存数据最适合被替换。

Spark 采用的 LRU 策略简单，得到了广泛的应用。然而，它在很多场景下并不能很好地工作。LRU 算法存在局限性，因为它没有考虑影响缓存性能的多种因素。根据我们的观察，在决定更换哪个分区时至少需要考虑四个重要因素：(1) 频率，(2) 引用计数，(3) 占用空间和 (4) 计算成本。这里，频率是指数据被再次访问的周期。引用计数表示数据被访问的次数。占

用空间定义为块的大小。计算成本定义为块的处理时间。LRU 只考虑新近度。其他一些广泛使用的缓存替换算法，例如随机、先进先出（FIFO）、最不频繁使用（LFU）以及最近提出的最小引用计数（LRC [3]）算法，最多只考虑单个两个因素。如图 1 所示，Random 不考虑任何因素，FIFO 只考虑 recency，LFU 只考虑频率，LRC 只考虑引用计数。为了克服这些现有解决方案的局限性，我们提出了一种新的缓存替换算法，称为最小分区权重（LPW）。LPW 的目标是最小化不同应用程序的总执行时间。LPW 的关键思想是设计一种新颖的统一权重模型来评估每个分区缓存的必要性。权重模型的计算基于对计算成本、引用计数和分区依赖等各种因素的分析。LPW 动态计算权重来预测最合适的数据，从而做出最佳的缓存决策。可以根据当前内存资源做出在线替换决策，优化 Spark 的缓存管理机制。

Cache replacement	Frequency	Recency	Reference count	Occupied space	Computation cost
Random	✗	✗	✗	✗	✗
FIFO	✗	✓	✗	✗	✗
LRU	✗	✓	✗	✗	✗
LFU	✓	✗	✗	✗	✗
LRC	✗	✗	✓	✗	✗
LPW	✓	✗	✓	✓	✓

图 1. 流行的缓存替换策略图

2 相关工作

2.1 LRU

为了管理有限的可用内存，Apache Spark 使用 LRU 驱逐策略作为默认策略。LRU 的基本思想是记录每个 RDD 的访问次数。当新数据产生并且存储内存不足以缓存新数据时，LRU 算法将驱逐属于最近最少访问的 RDD 的缓存分区。值得注意的是，所选分区和新分区不能属于同一个 RDD。这样的设计是为了防止同一个 RDD 的分区循环进出。LRU 驱逐最近最少使用的数据的机制是基于最近使用的数据更有可能被再次重用的假设，然而这个策略不适用于许多应用。

2.2 LRC

最小引用计数（LRC），它利用应用特定的 DAG 信息来优化缓存管理。LRC 驱逐那些引用计数最小的缓存数据块。对于每个数据块，引用计数被定义为尚未计算的依赖子块的数量。LRC 策略在三个方面优于 Spark 自带的 LRU 策略。首先，LRC 能够及时检测到引用计数为零的非活动数据块。这些块可能在剩余计算中不会再次使用，并且可以安全地从内存中淘汰。其次，与历史信息（如块访问的新旧程度和频率）相比，引用计数更准确地指示未来数据访问的可能性。直观地说，一个块的引用计数越高，越多的子块依赖于它，该块在下游计算中越可能需要。第三，引用计数可以在运行时精确跟踪，开销极小，使 LRC 成为所有基于 DAG 的系统的轻量级解决方案。但是 LRC 只考虑引用计数，并不考虑其他因素，所以泛化性并不高，所以这个策略也不适用于相当多的应用。

3 本文方法

3.1 本文方法概述

原文设计了 LPW 算法来指导 Spark 选择合理的数据进行缓存替换。在该算法中，提出了权重的概念来衡量缓存块的价值。权重越大，表明该块在应用程序执行过程中更有价值，并且将来更有可能被重用。与权重值较高的块相比，值较小的块更有可能被驱逐。许多因素可能会影响 Spark 应用程序的执行时间。在我们的设计中，权重的计算考虑了几个重要因素：区块的引用计数、占用空间、计算成本、历史执行信息（pastmod）。这些因素是综合考虑的。与 Spark 当前采用的 LRU 算法相比，该算法仅考虑数据块的最近使用信息，我们基于权重的算法对许多不同的应用更为可行和有效。

3.2 考虑因素

引用计数（Reference count）。当 RDD 的引用计数较大时，越多的子块依赖于它，该块在下游计算中越可能需要，因此更需要缓存以供重用。

计算成本（Computation cost）。不同 RDD 之间的成本往往相差很大，较大的计算成本意味着需要较长的计算时间。从完成这些 RDD 计算的任务中可以很容易地计算出详细的执行时间，缓存计算成本较大的 RDD 更有利于提高执行效率.

占用空间（Occupied space）。RDD 被划分为多个分区，作为占用内存的块分布在每个节点上。分区占用的空间越大，缓存该块的必要性就越小。过多的数据会占用很大一部分存储空间，很容易浪费大量的内存资源，甚至降低执行效率。驱逐占用内存较大空间的分区有助于减少应用程序的执行时间。

历史执行次数（Pastmod）。在过去的作业中经常使用的块更有可能在后续的作业计算中被访问。如果数据在前一段时间内被更频繁地访问，则该数据更有可能在未来被再次使用。Pastmod 表示在已完成的作业中计算的分区的引用计数。

3.3 分区权重模型设计

本研究综合考虑了上一节提到的所有因素。为了确定需要保留在内存中的有价值的分区，需要计算分区的权重。对于一个分区 partition_i，计算其权重的公式为

$$\text{weight}_i = \frac{\text{cost}_i \times \text{ref}_i \times (1 + \text{pastmod}_i)}{\text{size}_i}, \tag{1}$$

3.4 LPW 算法

原文提出的 LPW 算法旨在适应不同的 Spark 应用，该算法是基于及时的数据感知和动态调整来执行的。图 2 说明了该算法的伪代码，用于识别内存中缓存块的合理分区。变量分区表示需要在内存中缓存的新块。变量 cachedParts 是已经缓存在内存中的数据块的列表。变量 freeMem 表示节点的剩余内存。变量 pWeight 表示内存中缓存的每个数据块的权重值。首先，如果在 cachedParts 中找到分区，则不需要缓存数据，因为该块已命中（第 1-4 行）。空闲内存和 pWeight 中的权重（第 13-15 行）已更新。其次，计算每个分区的权重值（第 5 行）。计算方法如 (3) 所示。在替换过程中，分区的权重值按升序排序，并将排序后的权重添加到

pQueue（第 6-8 行）。在做出缓存决定之前，当 currPart 从内存中删除时，LPW 会继续检查空间是否足够（第 9-12 行）。首先，删除内存中 pWeight 中权重最小的分区，并将空闲内存空间跟踪到缓存分区。如果有足够的空间可用于缓存分区，LPW 将停止从内存中逐出块并更新 pWeight 和 cachedParts 的值（第 13-15 行）。算法 1 可用于遍历缓存的分区，基于对哪些数据块有较高的重用概率的预测，找到最合适的待替换分区。

Algorithm 1 lpwRep(cachedParts, partition, freeMem)

```
1: if partition ∈ cachedParts then
2:     return cachedParts[partition];
3:     break
4: end if
5: weight ← Compute(partition);
6: if freeMem < partition.size then
7:     pQueue ← SortByWeight(cachedParts);
8: end if
9: while freeMem < partition.size do
10:    currPart ← pQueue.pop();
11:    freeMem + = currPart.size;
12: end while
13: cachedParts.add(partition);
14: freeMem - = partition.size;
15: pWeight.add(partition);
```

图 2. 替换算法

4 复现细节

4.1 与已有开源代码对比

图 3 显示了 LPW 的体系结构概述。我们扩展的模块显示在阴影框中。Manager 模块位于 Driver 节点上，Monitor 模块分布在 Executor 节点上。本次复现实验使用到了作者开源代码中的以下 API，第一个 API 是 putOrUpdateWholeBlock，作用是更新分区的权重值；第二个 API 是 updateRef，作用是更新分区引用计数；第三个 API 是 updateCost，作用是更新分区的计算成本；第四个 API 是 calculateWeight，作用是计算每个分区的权重；第五个 API 是 broadcastRef，作用是将 RefCount 发送到 worker 上的 BlockManagerSlave；第六个 API 是 profileRefCountStageByStage，作用是计算单个作业中需要缓存的 RefCount；第七个 API 是 profileRefCountOneStage，作用是单阶段计算需要缓存的 RefCount；第八个 API 是 handleJobSubmitted，作用是计算引用计数；第九个 API 是 calculateTrueTime，作用是获取分区的计算时间。

在此实验中，所有模块都基于现有的 Spark 模块。Driver 节点上的 DAGScheduler 模块学习 RDD 之间的依赖关系，使我们能够获取参考计数（RefCount），它定义了在计算过程中每个 RDD 被依赖的次数。RefCount 通过 BlockManagerMasterEndpoint 发送到 WorkerManager BlockManagerSlave。Worker 上的 BlockManagerSlave 会不断向 BlockManagerMaster 发送心跳以更新块（分区）信息。一旦创建了 BlockManager 对象，它还将创建一个 MemoryStore 来管理块信息。通过检测 MemoryStore.scala，可以获取分区的大小和计算所花费的时间。在完成作业计算时获取每个 RDD 被依赖的次数，并根据模型计算分区权重。权重被计算并保存在执行相应任务的 Worker 中。在每个作业完成后，当新作业的依赖计数信息传递给 MemoryStore 时，MemoryStore 将记录其当前依赖信息，如在前一个作业中引用 RDD 的次数，并接收新的依赖信息。每当在 Worker 上发现内存不足时，MemoryStore 将发送当前节点的数据，并使用

LPW 决策信息选择适当的分区进行替换。在集群中，具有多个分区的 RDD 分布在每个节点上。RDD 的生命周期是从第一个作业到最后一个作业，它依赖于分区完成计算。分区的计算成本包含计算时间和通信时间。集群中的工作节点上的 ShuffleMapTask 和 ResultTask 记录了每个任务的运行时间，通过代码插桩可以获取每个分区的计算成本。BlockManager 用于管理 Spark 中的数据。通过监控 API，我们可以获取由分区占用的内存空间的大小。每个工作节点上的 BlockManager 基于权重做出替换决策，并将信息发送给 Driver。在复现实验的过程中，因为原算法没有考虑最后使用 RDD 的时间对内存替换策略造成的影响，根据时间局部性原理，最近被访问的内存块很有可能下次被重新访问。所以，考虑将最后使用该 RDD 的时间这一影响因素也纳入内存替换策略制定范畴。实现思路如下，把 RDD 的访问顺序体现在队列中，先使用到的 RDD 的先进队列，后使用到的 RDD 后进队列，队头元素设置访问权重为 1，排在第二的元素设置访问权重为 2，以此类推。当 RDD 被重新访问时，将这个 RDD 弹出队列，置于队尾。因此，便可实现动态计算 RDD 访问权重，最后，将该权重与原文提出的权重进行综合考虑，得出一个总权重。复现实验使用选择流行的 Hibench 基准测试并评估系统性能，选择 sparkbench 框架作为复现实验的实验框架。HiBench (V7.0) 中总共有 19 个工作负载，选用了其中四个工作负载：LogisticRegression，ScalaSparkNWeight，SVM，PCA 进行系统评估。

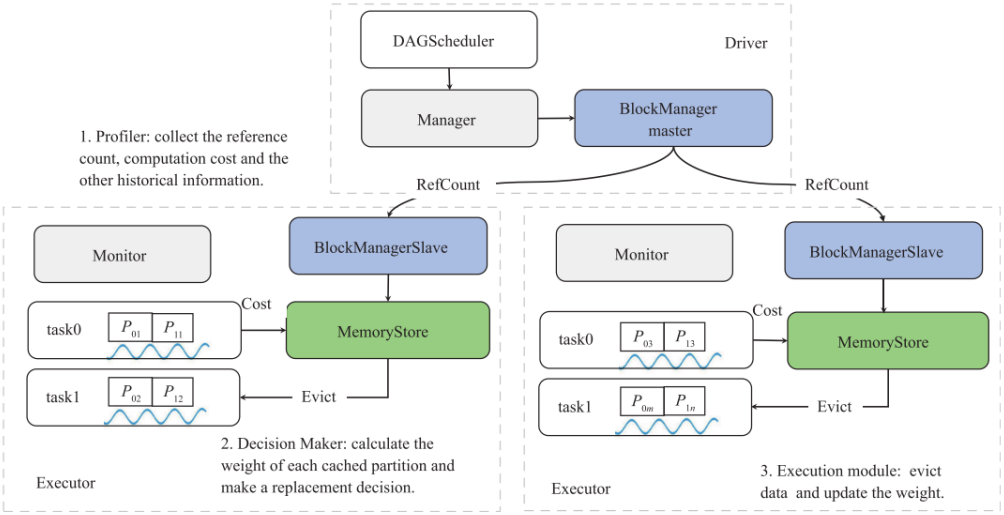

Figure 10 (Color online) System architecture of the LPW solution.

图 3. 体系架构

4.2 实验环境搭建

如图 4 所示在 VMware ESXi 上创建了 5 台虚拟机 gaia0 gaia4 作为本次实验的集群环境。每个节点分配 4 个 CPU，20GB 内存以及 200GB 的磁盘空间，但实际运行参数：executor.memory 为 6g，driver.memory 为 6g。安装的虚拟机版本为 Centos7.5，实验环境使用 JDK1.8，Scala2.11.12，Hadoop2.7.7，Spark2.2.3。实验中 Spark 运行模式均使用 spark-standalone 模式。以 gaia2 作为主节点部署 Hadoop 和 Spark，并配置开启相对应的服务如 HDFS、History Sever，图 5 为部署的 HDFS 服务的 Web 界面。并且使用选择流行的 Hibench 基准测试并评估系统性能，选择 sparkbench 框架作为复现实验的实验框架。

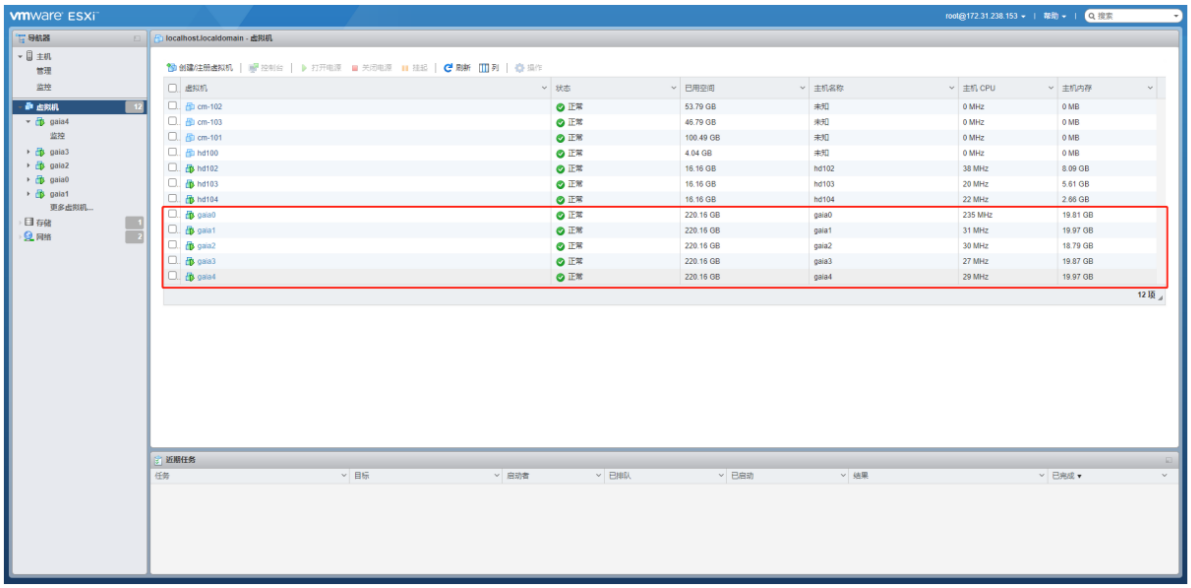

图 4. 集群搭建

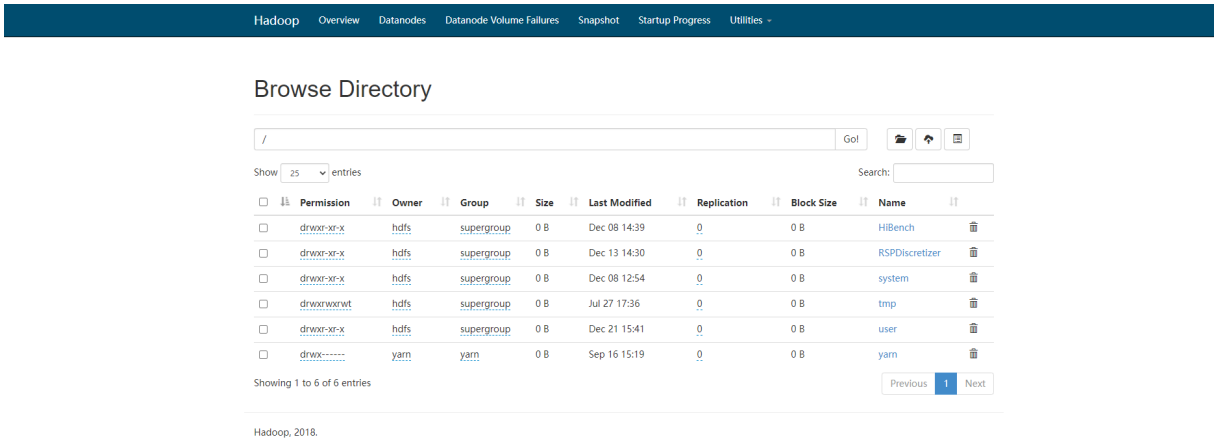

图 5. HDFS

4.3 界面分析与使用说明

本实验在 shell 上操作，操作安装的 Hibench 测试框架也是在 shell 上，图 6 为 Hibench 的主目录。图 7 为 Hibench 中 ml 下的不同工作负载。只需要进到工作负载目录下，执行 prepare 脚本生成指定规模的数据，再执行 run 脚本就能得到该工作负载下的实验数据。

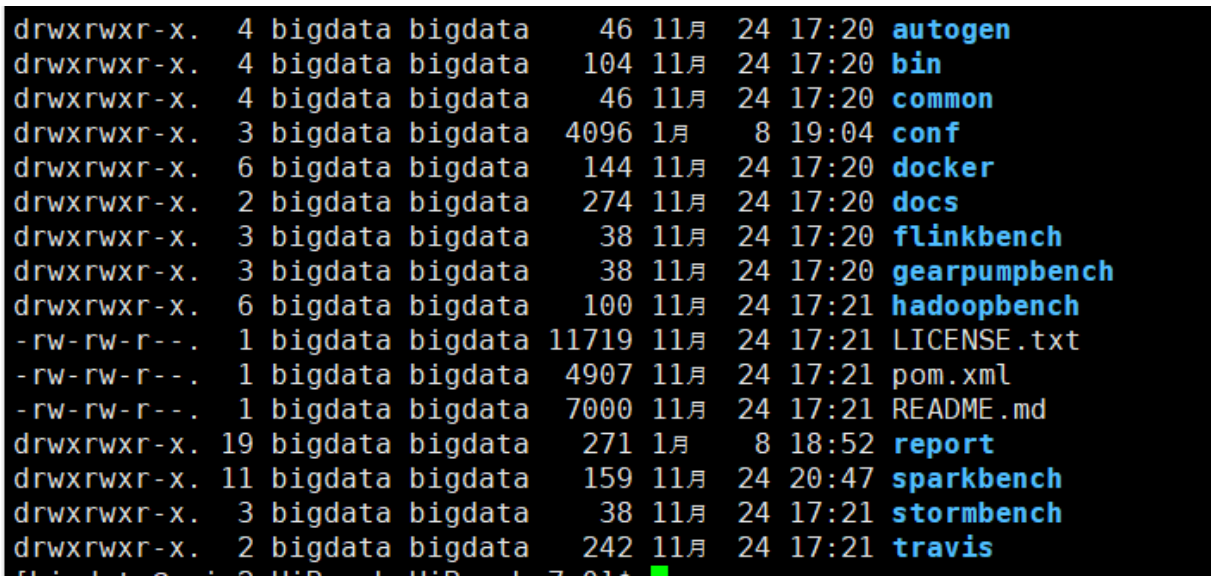

图 6. Hibench 主目录

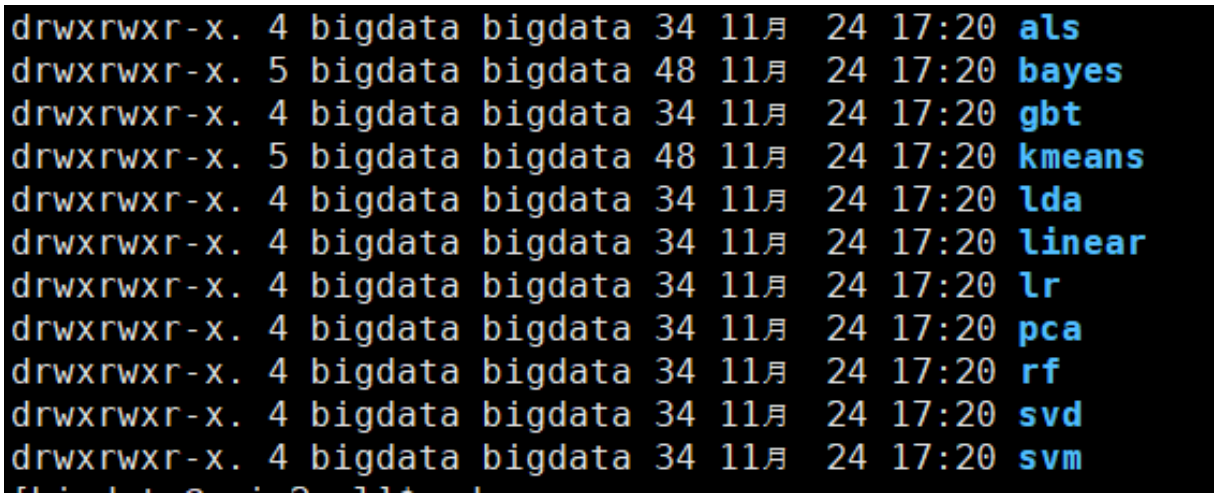

图 7. ml 下的工作负载

4.4 创新点

原算法没有考虑最后使用 RDD 的时间对内存替换策略造成的影响，根据时间局部性原理，最近被访问的内存块很有可能下次被重新访问。所以，考虑将最后使用该 RDD 的时间这一影响因素也纳入内存替换策略制定范畴。

5 实验结果分析

复现实验使用选择流行的 Hibench 基准测试并评估系统性能，选择 sparkbench 框架作为复现实验的实验框架。HiBench (V7.0) 中总共有 19 个工作负载，选用了其中四个工作负载：LogisticRegression，ScalaSparkNWeight，SVM，PCA 进行系统评估。其中设置 scale.profile 数据规模为 bigdata 进行测试，其中得到的结果如图 8 所示。其中每种负载类型都得到两条数据，其中这两条数据中的第一条为使用 LPW 策略得到的数据，下面为使用 LRU 策略得到

的数据。最终的结果表明使用 LPW 策略的 spark 框架在 LogisticRegression 这一工作负载下相对于使用 LRU 策略的 spark 框架表现出了良好的性能，在其余三种工作负载性能只得到略微提升，在这四种工作负载下实验得出的结果跟作者得出的实验结论基本吻合。

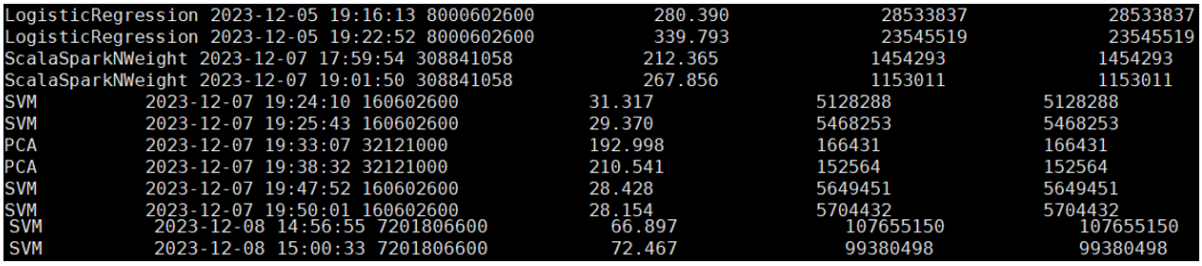

图 8. 实验结果

6 总结与展望

本次复现按照自己对原文的理解和对 Scala、Spark 的理解以及在作者源码的启发下复现了 LPW，结果上与原文工作存在一定的差距。复现工作中，为了实现其思想而简单的在小集群上进行运行。但是原文工作目的是为了解决大数据问题，大集群计算是需要的。在其基础上的时间开销会有明显的区别。并且在实验过程中并没有很好的控制变量，可能导致的实验结果跟原文有所差距。对于改进方向，可以再多考虑几种因素的影响，将 LPW 权重模型不断改变，以找到最优的权重模型。

参考文献

[1] Bikas Saha, Hitesh Shah, Siddharth Seth, Gopal Vijayaraghavan, Arun Murthy, and Carlo Curino. Apache tez: A unifying framework for modeling and building data processing applications. In *Proceedings of the 2015 ACM SIGMOD international conference on Management of Data*, pages 1357–1369, 2015.

[2] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *2010 IEEE 26th symposium on mass storage systems and technologies (MSST)*, pages 1–10. Ieee, 2010.

[3] Yinghao Yu, Wei Wang, Jun Zhang, and Khaled Ben Letaief. Lrc: Dependency-aware cache management for data analytics clusters. In *IEEE INFOCOM 2017-IEEE Conference on Computer Communications*, pages 1–9. IEEE, 2017.