DOI:10.3969/j.issn.1009-6868.2016.02.006
网络出版地址:http://www.cnki.net/kcms/detail/34.1228.TN.20160303.1005.002.html

Spark计算引擎的数据对象缓存优化研究

Data Object Cache in Spark Computing Engine

陈康/CHEN Kang
王彬/WANG Bin
冯琳/FENG Ling
(清华大学 计算机科学与技术系,北京 100084)
(Department of Computer Science and Technology, Tsinghua University, Beijing 100084, China)

中图分类号:TN929.5 文献标志码:A 文章编号:1009-6868 (2016) 02-0023-005

摘要: 研究了Spark并行计算集群对于内存的使用行为,认为其主要工作是通过对内存行为进行建模与分析,并对内存的使用进行决策自动化,使调度器自动识别出有价值的弹性分布式数据集(RDD)并放入缓存。另外,也对缓存替换策略进行优化,代替了原有的近期最少使用(LRU)算法。通过改进缓存方法,提高了任务在资源有限情况下的运行效率,以及在不同集群环境下任务效率的稳定性。

关键词: 并行计算;缓存;Spark;RDD

Abstract: In this paper, Spark parallel computing cluster for memory is studied. Its main work is about modeling and analysis of memory behavior in the computing engine and making the cache strategy automatic. Thus, the scheduler can recognize a valuable data object to be cached in the memory. A new cache replacement algorithm is proposed to replace least recently used (LRU) and have better performance in some applications. Thus, the performance and reliability of the Spark computing engine can be improved.

Key words: parallel computing; cache; Spark; resilient distributed dataset(RDD)

大数据处理的框架在现阶段比较有影响力的是基于Google所发明的MapReduce[1]方法及其Hadoop[2]的实现。当然,在性能上传统的消息传递接口(MPI)[3]会更好,但是在处理数据的方便使用性、扩展性和可靠性方面MapReduce更加适合。使用MapRecuce可以专注于业务逻辑,不必关心一些传统模型中需要处理的复杂问题,例如并行化、容错、负载均衡等。

由于Hadoop通过Hadoop分布式文件系统(HDFS)[4]读写数据,在进行多轮迭代计算时速度很慢。随着需要处理的数据越来越大,提高MapReduce性能变成了一个迫切的需求,Spark[5]便是在此种背景下应运而生。

Spark主要针对多轮迭代中的重用工作数据集(比如机器学习算法)的工作负载进行优化,主要特点为引入了内存集群计算的概念,将数据集缓存在内存中,以缩短访问延迟。

Spark编程数据模型为弹性分布式数据集(RDD)[6]的抽象,即分布在一组节点中的只读对象集合。数据集合通过记录来源信息来帮助重构以达到可靠的目的。RDD被表示为一个Scala[7]对象,并且可以从文件中创建它。

Spark中的应用程序可实现在单一节点上执行的操作或在一组节点上并行执行的操作。对于多节点操作,Spark依赖于Mesos[8]集群管理器。Mesos能够对底层的物理资源进行抽象,并且以统一的方式提供给上层的计算资源。通过这种方式可以让一个物理集群提供给不同的计算框架所使用。

Spark使用内存分布数据集,除了能够提供交互式查询外,它还可以优化迭代工作负载。使用这种方法,几乎可以将所有数据都保存在内存中,这样整体的性能就有很大的提高。然而目前Spark由于将缓存策略交由程序员在代码中手动完成,有可能会引起缓存的低效甚至程序出错,主要原因如下:

(1)程序员如果缓存无用的数据,将会导致内存未被充分利用,降低内存对程序性能的提升;

(2)错误的缓存甚至会产生内存溢出等严重后果,直接导致程序崩溃出错;

(3)程序中有具有缓存价值的数

收稿时间:2016-01-16
网络出版时间:2016-03-03
基金项目:国家高技术研究发展("863")计划(2013AA01A213);国家自然科学基金(61433008、61373145、61170210、U1435216);国家核高基重大专项(2013zx01039-002-002)

据得不到缓存，将使程序不能达到最高效率。

随着项目变大，代码量增加，这个问题会变得越来越严重。如果使用自动分析的方法，自动完成缓存的工作，无疑会降低程序员负担以及避免上述的问题。下面将对这方面进行初步研究，通过分析与建模，目的是使内存的使用更加智能有效，并加速任务的运行速度。

1 Spark中缓存优化研究

我们分3个方面对Spark中的缓存进行研究：一个是缓存自动化方法；一个是缓存替换方法改进；最后是程序执行调度顺序与缓存的关系。

1.1 Spark中的缓存

Spark通过将RDD数据块对象缓存在内存中这一方式对MapReduce程序进行性能上的提升改进。以经典的PageRank[9]算法为例，Spark比Hadoop快3倍左右。

以逻辑回归算法实验代码为例，如图1所示。

可以看出：在使用Spark时，从第2轮迭代计算开始，points的数据可以从缓存中直接读取出来，因此获得了极高的加速比。

1.2 数据对象的自动缓存

对于Spark来说，并不是每个RDD都要缓存到内存当中，需要进行筛选保留有价值的RDD存入内存。目前Spark中这种筛选的工作都交由程序员手动完成。以PageRank作为例子，如图2所示。

代码中有3个变量，且3个变量都是RDD类型，其中第1行最后的cache操作，会导致links被缓存到内存，在循环中可以从内存中直接读取，而ranks和contribs则没有被缓存，这就是Spark当前的缓存机制。

将缓存的工作交由程序员手动完成，对于系统本身的实现来讲，是简化了许多，但是对于程序员来讲则是极大的挑战。对于复杂操作，程序员通常很难直接分析出具有缓存价值的数据对象进行缓存。sortByKey是用来对数据依据其指定的键值进行排序的一个常用操作，具体如图3所示。

```
val points = spark.textFile(...).map(parsePoint).cache()
var w = Vector.random(D) // current separating plane
for (i <- 1 to ITERATIONS) {
    val gradient = points.map(p =>
    (1 / (1 + exp(-p.y*(w dot p.x))) -1) * p.y * p.x).reduce(_ + _)
    w -= gradient
}
println("Final separating plane: " + w)
```

▶图1 逻辑回归算法的Spark实现代码

```
val links = Spark.textFile("HDFS:...").map(...).cache()
val ranks = // RDD of (URL, rank) pairs
for (i <- 1 to ITERATIONS) {
            val contribs = links.join(ranks).flatmap {
            (url, (links, rank)) =>
            links.map(dest => (dest, rank/links.size))
    }
            ranks = contribs.reduceByKey((x,y) => x+y).
      mapValues(sum => a/N + (1-a)*sum)
}
ranks.save("HDFS:...")
```

▶图2 PageRank的Spark实现代码

可以看出，代码中的3个RDD均只使用了一次，在缓存links与不缓存links两种情况下，理论上两个时间应该相近，因为该RDD对象并未被再次利用，没有缓存的价值。然而实际中缓存links比不缓存快了近1倍。通过分析源码发现sortByKey的具体实现过程中有两个隐性任务，而这两个任务有数据相关性，通过缓存可以获得性能提升。这种数据相关性对于不了解sortByKey实现细节的程序员来说，是很难被察觉到的。

因此，将缓存的工作交由程序员手动完成，会让代码运行的效率随着程序员的水平不同而差别巨大，低效的缓存策略会让程序变慢，错误的缓存策略甚至会导致程序出错这一严重后果，由程序通过智能分析自动完成缓存策略则变得尤为重要。我们提出了一种缓存策略的自动化实现策略：

（1）在Spark源码中插入监听代码，当程序运行时记录程序中的关键信息，得到代码的无回路有向图（DAG）[10]，其中DAG图中的点对应程序中的RDD，边对应函数操作；

（2）统计DAG图中每个点的出度，即每个RDD将被使用到的次数；

（3）选出出度大于1的点，并将其对应的RDD进行缓存。

可以看出该方法需要运行2次程序：第1次为了分析出有价值的RDD进行缓存；第2次才是真正的作业运行。因此可以修改Spark的调度器，使其第1次运行时使用kB级别的小数据集，这样可以在很短的时间内运行完毕，相对于第2次真正运行时的大数据集（通常是GB或TB级别），额外一次运行不会影响性能。

1.3 数据对象缓存的调度策略

相对于MapReduce的计算框架，Spark的优势在于能将有价值的数据缓存到内存中，这样在迭代计算时能

```
val file = spark.textFile("hdfs:...")
val links = file.map(parseArctile _)
val sortlinks = links.sortByKey()
sortlinks.saveAsTextFile("hdfs:...")
```

▲图3 sortByKey的Spark实现代码

直接从内存读取数据，减少磁盘输入输出（IO），提高读写速度。然而实际生产中，由于内存大小有限，不可能缓存全部有价值的数据，只能部分缓存，在这种情况下，缓存替换算法的效率将极大影响作业运行效率。

目前，Spark 缓存替换算法使用的是近期最少使用算法（LRU）。基本方法是：当内存空间不足需要替换时，将缓存中距离当前时间最久没有被用到的数据替换出去；但在缓存空间不足时，Spark 的缓存调度器由于无法准确预测数据将来的使用顺序，导致LRU 替换算法效果欠佳。

实际上，针对缓存空间不足的情况，采用寄存器分配（RA）算法[11]会有更好的效果。已知内存容量的大小、RDD 的个数和 RDD 的访问顺序（由此可以得出 RDD 的生命周期以及每个 RDD 生命周期之间的关系），可以使用 RA 模型将 RDD 分配到内存的不同位置中。具体实施如下：

（1）通过分析程序得到每个 RDD 的生命周期，即开始使用时间和结束使用时间，并将该时间区间按照开始时间递增的顺序存入到 list 列表中；

（2）顺序遍历 list 列表，从缓存区中尝试为每个 RDD 分配可用空间；

（3）在遍历过程中持续维护一个列表，列表存放遍历的 RDD 的生存区间，并按照 RDD 的结束使用时间递增排序，同时删除当前时间已结束使用的 RDD，将其占用的空间释放返还到缓存区；

（4）如果缓存区已满，则将当前待分配空间的 RDD 加入上一步维护的列表中，并将结束使用时间最晚的 RDD 从列表中移除，将其占用的空间释放分配给当前待分配的 RDD；若需要移除的是当前 RDD，则不为其分配空间。

在 Spark 作业中，通过存储 RDD 的元数据信息来到达容错的目的。在执行到真实的数据读取与计算操作即 Action 操作之前，代码中计算的都只是 RDD 的元数据信息，并不会改变真实数据。只有遇到 Action 操作时才会进行真正的数据计算，并根据存储的 RDD 的元数据信息向前回溯，直至找到所需要的数据。因此，RDD 的访问顺序实质上是根据 Action 的顺序决定的。

低效的 Action 顺序，会导致低效的运行效率，以图 4 为例。图中的 R 为计算过程中产生的元数据 RDD，A 为触发真实数据的读写与计算的行为 Action。如果 Action 顺序为：A1、A4、A2、A5、A3，则 RDD 的访问顺序为：A、B、A、C、[A]、B、[A]、C、[A]、B、[A]、C。

可以看出，该访问顺序导致 RDD 频繁替换的概率要大出许多，因此 Action 顺序的优化变得尤为关键。

针对 Action 顺序优化问题，可以使用一种贪心算法来计算 Action 的顺序，具体如下：

（1）将初始的 Action 集合根据依赖关系进行聚类，形成新的 Action 集合 A，并建立一个空集合 R，一个空序列 S；

（2）从集合 A 中随机选出一个 Ai 加入 S 末尾，同时从 A 中剔除 Ai，并将 Ai 对应的所有 RDD 加入集合 R；

（3）遍历集合 A，选出当 Aj 对应的 RDD 集合与 R 交集最大时的 Action Aj，将 Aj 加入 S 末尾，同时从 A 中剔除 Aj，并将 Aj 对应的所有 RDD 加入集合 R；

（4）重复上述步骤，直至集合 A 为空，计算结束，序列 S 即为 Action 的优化后顺序序列。

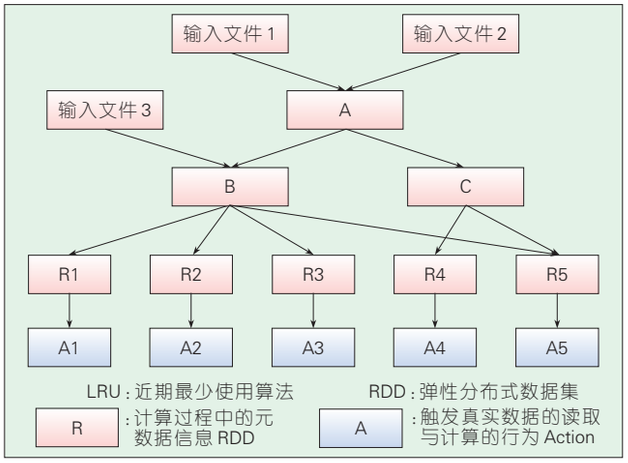
2 Spark 中缓存优化实验

2.1 实验环境

我们使用两套不同的集群配置运行不同的任务。第 1 部分使用 3 台节点搭建的小集群，配置如下：16 核中央处理器（CPU），48 G 内存，750 G 硬盘，网络带宽 1 000 M，操作系统为 64 位 CentOS 5.4；第 2 和第 3 部分实验使用 16 台节点配置的集群，配置如下：12 核 CPU，48 G 内存，880 G 硬盘，1 000 M 网络带宽，操作系统为 64 位 RedHat 6。使用的软件及版本为：Spark 0.5、Mesos 0.9.0、Hadoop 0.20.205。

2.2 实验 1——逻辑回归算法

我们以 1.1 节中的代码进行验证，该代码采用的是逻辑回归算法，是对数据的一种划分方式。计算方法为：通过多轮迭代计算来不断修正初始值 w，直至 w 的值在某轮计算后的改变小于设定阈值或迭代计算的次数达到设定次数上限。这里使用的是达到设定迭代次数上限的方式来停止计算。其中需要缓存的 RDD 为变量 points。实验主要观察在不同的数据大小的情况下，缓存 points 和不缓存 points 的运行时间的区别，实验结果如表 1 所示。



▶图 4 LRU 行为解析示例

▼表1 逻辑回归算法实验结果

数据大小/ G	迭代轮数	运行时间(缓存)/s	运行时间(未缓存)/s
2.8	迭代1	30	30
	迭代2～5	0.7	8
5.5	迭代1	66	72
	迭代2～5	0.9	12
14.0	迭代1	191	228
	迭代2～5	1.8	30

由实验结果可以发现：第1轮由于需要从HDFS中读取数据，缓存不能命中，因此使用缓存和不使用缓存对运行时间没有影响，二者时间基本相同；但从第2轮开始，由于可以从缓存中读取points数据，使用缓存将使性能极大提高，可以获得10～20倍左右的加速比。该实验证明Spark中的缓存机制在数据密集型计算中有巨大的提升效果。

2.3 实验2——sortByKey

我们以1.2中sortByKey的代码进行验证。上文中我们已经分析过，理论上两个时间应该相近，结果如图5所示。

使用缓存使运行时间缩短了一半。通过对实验过程的进一步细化拆分，我们发现整个任务又可以拆分成3个子任务，运行时间分别如图6所示。

可以发现：对于任务2和任务3，使用缓存使任务时间极大地缩短，任务2获得近200倍的加速比，任务3也有近5倍。sortByKey采用了sample的方法，因此两个数据相关的隐性任务2和任务3封装在代码当中。对于类似sortByKey这样的复杂操作，很容易出现这种具有隐藏的数据相关的情况，如果不了解这些复杂操作的具体实现细节，很难发现这种相关性加以利用。

2.4 实验3——不同缓存策略比较

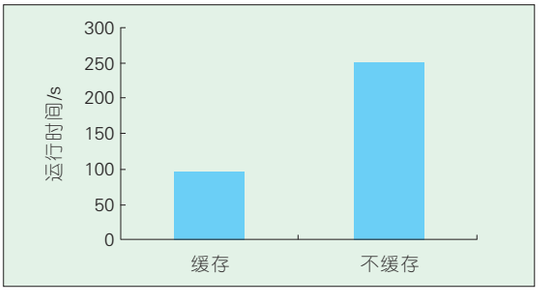
在该实验中，我们主要对比不同的缓存策略对任务执行效率的影响，使用的是图1所示的代码，一个PageRank的实现。下面我们将对比LRU替换算法和RA替换算法的运行时间的情况。代码中有多个RDD需要缓存，采用两种替换策略，在不同的缓存大小情况下，任务的运行时间差别很大，主要测试的是：在缓存空间极少（仅能缓存1个RDD）、较少（能够缓存一半的RDD）以及充足（能够缓存所有RDD）的情况下，在LRU算法和RA算法两种不同替换策略下，任务运行时间的变化，实验结果如图7所示。

从图7中我们可以看出：当所有RDD都缓存后，使用LRU替换算法与RA替换算法的运行时间都随可用缓存空间的增大而降低，当缓存空间足够大到能够缓存所有运行时的RDD时，两者的运行时间基本相同，均为120 s。但是对于缓存空间不足的情况，RA算法明显比LRU算法的性能要好：在缓存空间较少，仅能缓存运行时产生的一半的RDD的情况下，LRU算法运行时间为460 s，而RA算法运行时间只有350 s。RA替换算法更能适应不同的情况，在大多情况下都比LRU替换算法运行效率更高。

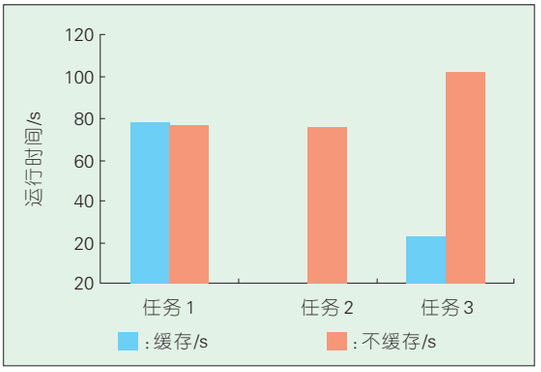
2.5 实验4——Action 顺序效果实验

该实验的目的是验证Action的顺序对于任务运行时间的影响。实验代码的核心部分代码如图8所示，其中1为原始代码，2为做Action顺序优化后的代码。

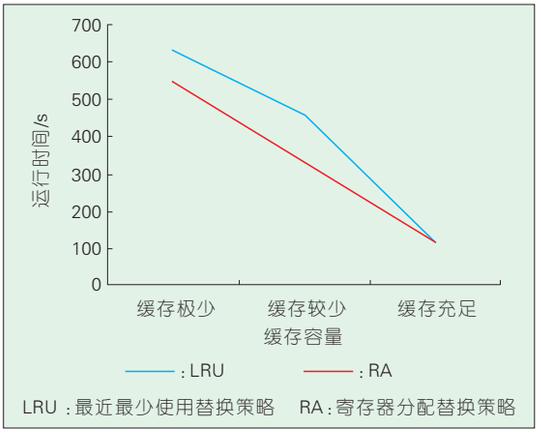
实验结果如图9所示：测试了在替换算法相同时，在内存极少以及内存较少的情况下，不同Action顺序的运行时间。从图中我们可以看出：Action顺序优化后的代码的运行时间在内存极少和内存较少的情况下均比优化前代码运行时间短，尤其在内



▲图5 sortByKey实验结果1



▲图6 sortByKey实验结果2



▲图7 两种缓存策略的结果比较

存较少的情况下，可以达到近4倍的加速比。Action顺序优化的作用十分明显。

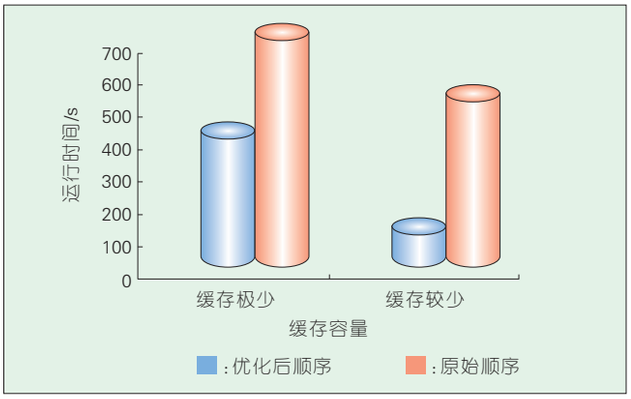
3 结束语

通过对并行计算框架Spark进行深入研究，并对Spark中的内存使用模型进行系统分析后，我们从多个角度对Spark中的缓存系统进行改进，使得系统具有更强的鲁棒性，并且在内存空间不足、执行的作业复杂等恶劣情况下依然有较高的性能。首先提出了自动化缓存策略的制定，将程序员从手动制定缓存策略中解放出来，降低程序员的编程时间以及出错的可能性；其次，对Spark原有的RDD缓存替换策略进行优化，使得新的替换策略在内存空间不足的情况下比原有策略性能更好，提高作业运行效率；最后，研究了Action顺序优化对作业运行效率的影响。通过这一系列的建模与改进工作，可以使Spark变得更有效率，最后也在实验的结果验证了这一观点。

理论建模是文中所涉及工作的重点内容，但为了体现出工作的实际效果，还需要大量的工程实践，将改进后的系统真正完善起来，才能真正投入应用，从而发挥出实际价值。

```
[1]
for(i <- 1 to 5) {
  links1.count
  links2.count
}
[2]
for(i <- 1 to 5){
  links1.count
}
for(i <- 1 to 5) {
  links2.count
}
```

▲图8 两种Action顺序的实现代码



◀图9 Action顺序结果

参考文献
[1] DEAN J, GHEMAWAT S. Mapreduce: Simplified Data Processing on Large Clusters [J]. Communications of the ACM 50th Anniversary Issue, 2008, 51(1): 107–113. DOI: 10.1145/1327452.1327492
[2] GABRIEL E, FAGG G, BOSILCA G, et al. Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation[C]// Proceedings European PVM/MPI Users' Group Meeting. Germany: Springer Berlin Heidelberg, 2004: 97–104. DOI: 10.1007/978–3–540–75416–9_35
[3] GHEMAWAT S, GOBIOFF H, LEUNG S T. The Google File System[C]//Proceedings of the Nineteenth ACM Symposium on Operating Systems. USA: ACM, 2003: 29–43
[4] CHANG F, DEAN J, GHEMAWAT S, et al. Bigtable: A Distributed Storage System for Structured Data[C]//Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation–Volume 7. USA: USENIX Association, 2006: 15–15
[5] JIANG Y. HBase Administration Cookbook [M]. UK: Packt Publishing, 2012
[6] ZAGARIA M, CHOWDURY 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. USA: USENIX Association, 2012:2–2
[7] OLIVEIRA B C, GIBBIONS J. Scala for Generic Programmers[C]//Proceedings of the ACM SIGPLAN workshop on Generic programming. USA: ACM, 2008: 25–36
[8] HINDMAN B, KONWINSKI A, ZAHARIA M, et al. Mesos: A Platform for Fine–Grained Resource Sharing in the Data Center[C]// Proceedings of the 8th Usenix Conference on Networked Systems Design and Implementation. USA: USENIX Association, 2011: 22–23
[9] ZHANG J, ZHOU H, CHEN R, et al. Optimizing Data Shuffling in Data–Parallel Computation by Understanding User–Defined Functions[C]//Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation. USA: USENIX Association, 2012: 22–23
[10] OLSTON C, REED B, SILBERSTEIN A, et al. Automatic Optimization of Parallel Dataflow Programs[C]//Proceedings of USENIX 2008 Annual Technical Conference on Annual Technical Conference. USA: USENIX Association, 2008: 267–273
[11] SMITH M D, RAMSEY N, HOLLOWAY G. A Generalized Algorithm for Graph–Coloring Register Allocation[C]//Proceedings of the ACM Sigplan 2004 Conference on Programming Language Design and Implementation.USA: ACM, 2004: 277–288

作者简介

陈康，清华大学计算机科学与技术系副教授；研究方向为分布式系统、存储系统等；先后主持和参加“863”、“973”等项目10余项；获得1项科研成果奖；已发表论文20多篇，其中被SCI/EI检索10余篇。

王彬，清华大学计算机科学与技术系硕士研究生；研究方向为分布式计算。

冯琳，清华大学计算机科学与技术系硕士；研究方向为分布式计算。