CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2016; **28**:2473–2486
Published online 28 August 2015 in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/cpe.3584

SPECIAL ISSUE PAPER

Selection and replacement algorithms for memory performance improvement in Spark

Mingxing Duan[1], Kenli Li[1,2,*,†], Zhuo Tang[1,2], Guoqing Xiao[1] and Keqin Li[1,2,3]

[1]*College of Information Science and Engineering, Hunan University, Changsha, Hunan, China*
[2]*National Supercomputing Center in Changsha, Changsha, Hunan, 410082, China*
[3]*Department of Computer Science, State University of New York, New Paltz, NY 12561, USA*

SUMMARY

As a parallel computation framework, Spark can cache repeatedly resilient distribution datasets (RDDs) partitions in different nodes to speed up the process of computation. However, Spark does not have a good mechanism to select reasonable RDDs to cache their partitions in limited memory. In this paper, we propose a novel selection algorithm, by which Spark can automatically select the RDDs to cache their partitions in memory according to the number of use for RDDs. Our selection algorithm speeds up iterative computations. Nevertheless, when many new RDDs are chosen to cache their partitions in memory while limited memory has been full of them, the system will adopt the least recently used (LRU) replacement algorithm. However, the LRU algorithm only considers whether the RDDs partitions are recently used while ignoring other factors such as the computation cost and so on. We also put forward a novel replacement algorithm called weight replacement (WR) algorithm, which takes comprehensive consideration of the partitions computation cost, the number of use for partitions, and the sizes of the partitions. Experiment results show that with our selection algorithm, Spark calculates faster than without the algorithm, and we find that Spark with WR algorithm shows better performance. Copyright © 2015 John Wiley & Sons, Ltd.

Received 29 January 2015; Revised 14 June 2015; Accepted 16 June 2015

KEY WORDS: iterative computation; memory; resilient distribution datasets; selection algorithm; spark; weight replacement algorithm

1. INTRODUCTION

Processing amounts of data-intensive applications in the era of big data has presented a challenge issue [1–4]. Apache Spark [5] is an open-source parallel computation framework that computes in-memory based on resilient distribution datasets (RDDs) to speed up data processing [6], [7]. As a key abstraction in Spark, RDD is a collection of objects partitioned across nodes in Spark cluster, and all partitions can be computed in parallel. More importantly, as a crucial abstraction, RDD leverages the distributed memory to cache the intermediate results, with which, Spark has a huge advantage over other parallel frameworks in terms of processing iterative machine learning. At the same time, users can control storage strategies of Spark (e.g., in-memory storage and so on) and based on that, Spark can cache many multiple occurrences of RDDs, so that users can reuse intermediate results across multiple computations.

However, when Spark processes large-scale intensive applications, multiple storage strategies are selected for the whole process of the computation. If users select storage strategies for the

*Correspondence to: Kenli Li, College of Information Science and Engineering, Hunan University, Changsha, Hunan, 410082, China.
†E-mail: lkl@hnu.edu.cn

computations, the results are full of uncertainties. Different results will be caused by different levels of programmers. Whether the process should cache the variables is completely according to the experience. The uncertainty cannot adequately exploit the strengths of the high performance of Spark. Poor storage strategies not only lower the efficiency of the program but also lead to errors during the process of implementation. When useless intermediate results are kept in distributed memory according to the storage strategy, it is likely to waste memory and degrade the executing speed, even cause out-of-memory errors. Meanwhile, this situation misses the useful intermediate results which will be reused in the subsequent computations. Therefore, they should be recomputed each time when the other computations need them again, which lower the performance of Spark. More importantly, we must recognize the fact that memory is limited, and it does not make sure each computation has enough memory.

In this paper, we put forward a novel selection algorithm, by which Spark selects the reasonable intermediate results to be cached in memory instead of being decided by users. The algorithm reduces many unnecessary mistakes caused by programmers and makes sure that the useful RDDs are kept in memory. We leverage the algorithm to choose the value RDDs to cache their partitions in memory according to the number of use for RDDs during the process of computation. Experiments show that the algorithm reduces overhead and speeds up the whole computation process.

When limited memory is full of the partitions, there are many new value RDDs partitions to be cached in memory. Spark selects the unused for longest time cache partitions to be replaced according to the least recently used (LRU) replacement policy. However, the LRU algorithm just considers whether the partitions are used recently, and it does not make sure the chosen one is valueless. If the computation cost of the chosen one is higher, we should spend much cost to recompute it in the future. In addition, if the size of the chosen one is smaller than the other one cached in memory, which is not used recently as the chosen one, it is reasonable to select the larger one to be replaced, which means more space to be provided for new value RDDs partitions. How to take comprehensive consideration of these factors is another important problem in Spark.

In this paper, we propose a new novel replacement algorithm called weight replacement (WR) algorithm. The weight of RDDs in our algorithm contains three parts: the partitions computation cost, the number of use for partitions, and the sizes of the partitions. When the computation cost of partition is high, it is a value partition. At the same time, if different partitions have the same computing cost while the number of use of one partition is the largest among them, then its weight is the largest. Considering the above factors, we select the reasonable partitions to be replaced, which may speed up the process of Spark computation.

The remainder of this paper is organized as follows. Section 2 reviews the related work. Section 3 introduces preliminary information. Section 4 gives models and problem formulation. In Section 5, we discuss the proposed algorithms. Experimental evaluation is illustrated in Section 6. Finally, we make a conclusion in Section 7.

2. RELATION WORK

Because of characteristic of *persistence*, Apache Spark [5] provides multiple storage strategies for users to cache the intermediate results, such as memory-only, memory-only-2, and so on. In order to avoid computing the same RDDs repeatedly, programmers can choose one or several kinds of storage strategies for procedure. However, when several storage strategies are chosen for procedure, different programmers will select different strategies, which cannot make sure the useful RDDs cached in memory or make full use of *persistence* characteristic. The results are wasting memory or degrading the performance, which cannot adequately exploit the strengths of the *persistence* characteristic of Spark.

In our work, instead of the storage strategies chosen by users, Spark keeps the multiple occurrences of RDDs in distributed memory according to our selection algorithm. That reduces the mistakes or errors caused by programmers, improves the procedure reliability, reduces many overhead, and speeds up the whole process of calculation.

There are many widely used replacement algorithms such as first in first out (FIFO), LRU [8], least frequently used, frequency-based replacement [9], least recently used and least frequently used [10], [11], Landlord [12], [13], adaptive weight ranking policy (AWRP) [14]. Landlord algorithm is an online mechanism, which combines replacement cost, cache object size, and locality by extending both LRU and FIFO to include cost and the sizes of variable cache object within a cache [13]. However, the credit or cost of object given by LANDLORD algorithm is not ideal, because the algorithm does not take a comprehensive consideration of the number of use of the object and the computation cost of the object. Swain *et al.* proposed an AWRP and it assigned weight to each object in the buffer and ranked each object as per the weight. Nevertheless, the weight assigned by AWRP only considers the frequency of object while ignores the sizes of the object. LRU is widely used in operating systems as a buffer replacement policy because of its simplicity, and in LRU policy, the unused for longest time cache block will be replaced. The LRU policy is implemented on Spark framework to manage the limited memory available. When no enough space is provided for a new RDD partition, the system evicts a partition from the least recently accessed RDD, unless this is the same RDD as the one with the new partition. Although the LRU algorithm allows Spark to manage the memory, it only considers whether the partitions are used recently and cannot make sure the selected one is the valueless.

Compared with the replacement algorithms discussed previously, our work selecting the replacement parts not only considers the number of use of partitions but also takes the partitions computation cost and the sizes of the partitions into account. That makes sure the selected partitions be reasonable replacement parts and reduces many overhead.

3. PRELIMINARY INFORMATION

3.1. Resilient distribution datasets

As a user, we can think RDD as a handle for a collection of individual data partitions, and it is the critical part of Spark. Each RDD is characterized by five main properties [6]:

(1) A list of partitions. RDD is an abstraction in Spark and it contains a list of partitions, which are distributed on different nodes;
(2) A function for computing each split;
(3) A list of dependencies on other RDDs, we call that *lineage* and the DAGScheduler forms a Directed Acyclic Graph (DAG) of stages for each job according to the *lineage*;
(4) Optionally, a partitioner for key-value RDDs (e.g., to say that the RDD is hash-partitioned);
(5) Optionally, a list of preferred locations to compute each split (e.g., block locations for an Hadoop Distributed File System (HDFS) file).

Resilient distribution dataset is a read-only dataset, which can only be created by some operations (e.g., sc.textFile (hdfs://... /data.txt)) based on data in stable storage or other transformation operation in Spark (e.g., map, join, and so on). Transformations and actions are two kinds of operations. There are no actual operations to be carried out in process of transformation until action operations happened. During transformation operation process, RDD is transformed into different RDDs, and this process does not execute until it meets the action operation. That means there are many RDDs transforming into other kinds of RDDs, and each RDD remembers their parents and their children. If RDDs appear repeatedly, we can cache their partitions in memory during iterative computation process. It provides fault tolerance without requiring replication, by tracking how to recompute lost data starting from the base data on disk. This lets RDDs be read and created to 40 times faster than a typical distributed file system. Spark runs programs up to 100 times faster than MapReduce in memory, or 10 times faster on disks [15].

It can be seen in Figure 1 that RDD includes a list of partitions, and all partitions are distributed among the cluster nodes, which can be computed in parallel. By default, RDDs are *transient*, and they should be recomputed each time during the action computation [16]. All partitions have serial number, and in this paper, we use P to denote a partition, and P_{ij} means the j-th partition of the i-th RDD.

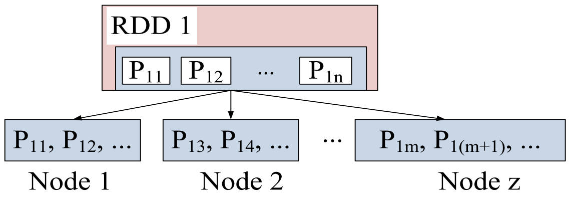


Figure 1. The distribution partitions of resilient distribution dataset (RDD) in cluster.

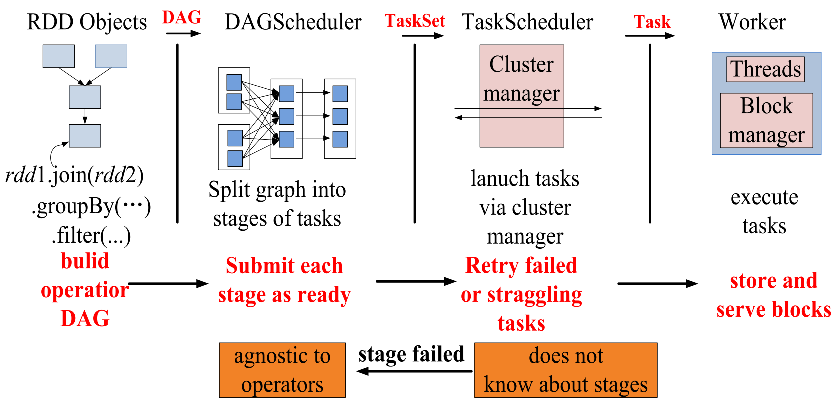


Figure 2. System scheduling in Spark.

3.2. *System scheduling in Spark*

All operations in Spark are based on RDDs, and there are two kinds of operation: transformation operations (e.g., map, join, groupBy, and filter) and action operations (e.g., reduce, count, collect, and save). During transformation operation process, RDD is transformed into different RDDs, and this process is not executed until its action operation happened. According to the dependencies between RDDs, DAGScheduler forms a DAG of stages for each job, and then it submits tasks to run on the cluster. So our selection algorithm selects the value RDDs to cache their partitions in memory according to the DAG before tasks are performed. In this paper, we use $G(R, E)$ as DAG model where $R = \{RDD_1, RDD_2, \ldots, RDD_n\}$ is the set of RDDs and E is the set of directed edges. This process is shown in Figure 2.

3.3. *Cache mechanism in Spark*

All iterative computations are computed in memory, and the multiple occurrences of RDDs are not cached in memory by default. We use a selection algorithm to select the multiple occurrences of RDDs to cache their partitions in memory according to the DAG, and the scheduling process in memory is illustrated in Figure 3.

It can be seen in Figure 3 that when RDD partitions have been cached in memory during the iterative computation, an operation which needs the partitions will get them by CacheManager. Moreover, the partitions are cached by CacheManager, and all operations including reading or caching in CacheManager mainly depend on the API of BlockManager. BlockManager decides whether partitions are obtained from memory or disks, while MemoryStore determines whether partitions are cached in memory. When in-memory is full of partitions, the system will use LRU replacement algorithm to select LRU partitions to be replaced. However, the LRU algorithm does not consider the sizes of RDDs partitions and the computation cost of RDDs partitions, and cannot make sure the chosen partition to be the reasonable one. Our proposed WR algorithm makes full consideration of the three factors and tries its best to select the reasonable RDDs partitions to be replaced.

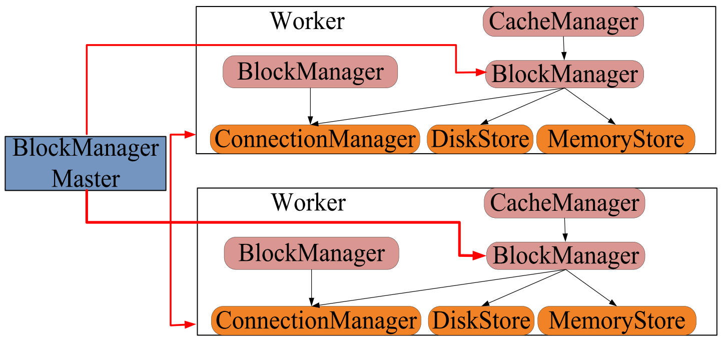


Figure 3. Cache mechanism in Spark.

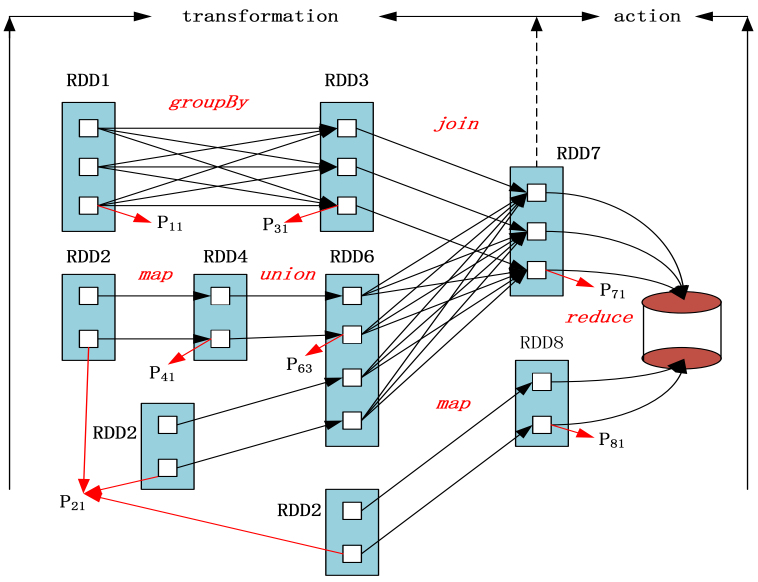


Figure 4. The DAG graph for this process.

4. MODEL AND PROBLEM FORMULATION

4.1. Architecture model

In this paper, the $G(R, E)$ is modeled as a DAG, and an edge e_{ij} of the form (RDD_i, RDD_j) denotes the dependency between RDD_i and RDD_j, in which RDD_i denotes the parent and RDD_j is the child. According to this definition, a child RDD cannot work until all of its parent RDDs are finished completely. Figure 4 is a DAG which stands for dependencies between RDDs in Spark.

We use CloudStack to manage our servers. Based on it, we deploy several virtual machines, and all of them are of the same type. So the processing capacity of each virtual machine is the same. Because memory in our paper is variable, the processing capacity changes with different memory, and we use P_{memory_k} to denote the processing capacity. In this paper, 1G, 2G, 4G, three different kinds of memory will be used, so k=1, 2, 3. We use P_{memory_k} to calculate the execution time of RDDs. The execution time of P_{ij} is estimated using the size of the jth partition of RDD_i, which is called S_{ij}, in terms of floating point operations [17]. $\{P_{i1}, P_{i2}, \ldots, P_{ih}\}$ are the partitions of RDD_i, and we can approximate partition execution time $ET_{P_{ij}}$ as follows:

$$ET_{P_{ij}} = S_{ij} / P_{memory_k}. \tag{1}$$

All partitions are computed in parallel, so the longest execution time of RDD_i partitions is roughly the execution time of RDD_i. Equation (2) describes the execution time ET_{RDD_i}:

$$ET_{RDD_i} = \max\left\{ET_{P_{i1}},\ ET_{P_{i2}},\ \ldots,\ ET_{P_{ih}}\right\}. \tag{2}$$

4.2. Scheduling model

The reason why Spark deals with big data faster is that it builds a DAG of stages for each job, which reduces many overheads during the process. When an iterative algorithm is carried out, there are many variables which are recurring in an iterative process or multiple processes. When that happens, it is important to cache the recurring variables in memory to speed up the whole computation. We select the multiple occurrences of RDDs to cache their partitions in memory according to our proposed algorithm, which speeds up the whole computation.

When there are many partitions cached in memory, the LRU partitions are chosen to store in disks based on LRU algorithm. However, the LRU algorithm only considers whether those partitions are recently used while ignores the partitions computation cost and the sizes of the partitions. When the sizes of two partitions and their number of use are the same, one needs more time or overhead to recompute it, which means that its computation cost is higher. There is no doubt that the higher computation cost of partition should be cached. In addition, if the computation cost and the number of use for two partitions are the same, the larger one needs more space. In this condition, caching the smaller one means that there are more opportunities to cache more partitions.

All transformation operations are not executed until action operations work. So the number of use for partitions can be known from the DAG before tasks are performed, and N_{ij} is the number of use of j-th partition of RDD_i. We use S_{ij} to express the size of j-th partition. Except execution time, all partitions need communication with each other. So communication time is also an important part. Each partition of RDD_i starting time ST_{ij} and finishing time FT_{ij} can roughly express its execution and communication time. We can approximate the computation cost of the partition as follows:

$$Cost_j = FT_{ij} - ST_{ij}. \tag{3}$$

After that, we set up a scheduling model and obtain the weight of P_{ij}, which can be expressed as Equation (4):

$$V_{P_{ij}} = k \times \frac{Cost_j \times N_{ij}}{S_{ij}}, \tag{4}$$

where k is a correction parameter, and it is set to a constant. We use V_P to denote the weight of partition, because it considers the computation cost of partition, the number of use for partitions, and the sizes of the partitions. There is no doubt that V_P is proportional to the partitions computation cost, and it is necessary to cache the partitions with higher computation cost. The number of use for partitions is also an important part in this equation and the frequently used partitions should be cached in memory instead of disks to reduce the I/O spending and communication overhead. As we know, the larger partitions cached in memory need more space. When different partitions have the same computation cost and the number of use, it is reasonable to choose the largest one to be stored in disk, which means more space in memory to be provided for new value RDDs partitions.

Finally, we assume that there are h partitions in RDD_i, and we can approximate the weight of RDD_i as follows:

$$V_{R_i} = \sum_{j=1}^{h} V_{P_{ij}}. \tag{5}$$

We use the time lag of partitions to roughly represent the computing cost and the complexity. Because all nodes are the same, and we can roughly think that their processing capacities are the same. There is no doubt that the partition spends more time to finish the task, which means that its computation cost is higher. In that condition, we can roughly use partition time lag as its computation cost. When many partitions are cached in memory which is near saturation, we can select the valueless partitions to be replaced based on $V_{P_{ij}}$ and V_{R_i}.

4.3. Problem definition

Spark is a memory consumption computation framework which is suitable for iterative algorithms. When the partitions are cached in memory, this can speed up the computation. We assume $\{RDD1, RDD2, \dots, RDDn\}$ are the whole datasets in the computation. All RDDs have many partitions. So we expect that the system spends less time or overhead to finish computation with our proposed algorithms. That is depicted in Equation (6):

$$\text{Minimize} \sum_{i=1}^{n} ET_{RDD_i}. \tag{6}$$

All nodes have the same memory, which is used to cache partitions, and the sizes of all the cached partitions should be less than the whole memory, which is used to cache partitions in cluster. Because of exceeding the memory, the partitions should be stored in disks or removed. We use S_{cache} to represent the whole memory which is used to cache partitions. Because different RDDs have different number of partitions, we use different parameters to denote corresponding RDDs partitions number (e.g., $h_1, h_2, \dots, h_n$). So we can depict the following:

$$\sum_{j=1}^{h_1} S_{1j} + \sum_{j=1}^{h_2} S_{2j} + \dots + \sum_{j=1}^{h_n} S_{nj} < S_{\text{cache}}. \tag{7}$$

5. PROPOSED ALGORITHMS

5.1. Selection algorithm

Figure 4 is an iterative computation process, and we can see this as a DAG graph. We assume that $\{P_{11}, P_{21}, P_{31}, P_{41}, P_{61}, P_{63}, P_{71}, P_{81}\}$ are in one node, and $P_{11} \Longrightarrow P_{31} \Longrightarrow P_{21} \Longrightarrow P_{41} \Longrightarrow P_{63} \Longrightarrow P_{21} \Longrightarrow P_{61} \Longrightarrow P_{71} \Longrightarrow P_{21} \Longrightarrow P_{81} \Longrightarrow DataCollection$ are execution sequences during the iterative process. When we do not cache the multiple occurrences of RDDs, we approximate the execution time in one node as Equation (8):

$$ET_P = ET_{P_{11}} + 3ET_{P_{21}} + ET_{P_{31}} + ET_{P_{41}} + ET_{P_{61}} + ET_{P_{63}} + ET_{P_{71}} + ET_{P_{81}}. \tag{8}$$

From Equation (8), we find it has to calculate P_{21} three times. So that process has to spend extra $2 \times ET_{P_{21}}$ finishing this computation. That process is a small part during the iterative process. Therefore, the whole computation has to spend many extra overheads to calculate the same variables. If we cache the multiple occurrences of RDDs, the system performance would be improved.

In this paper, we use our selection algorithm to select the reasonable RDDs to be kept in memory. During the transformation operations, RDDs are transformed into other kinds of RDDs, and that process is not executed until action operations are executed. So we can use our selection algorithm to select the value RDDs to cache their partitions in memory before action operations happen. The transformation operations form a DAG graph, and if the RDD appears again, it is cached in memory. The chosen RDDs will be kept in collection M, and N_{RDD_i} denotes the number of use for RDD_i. We assume all RDDs have the same partitions and we set the parameter as h. The free memory which is used to cache partitions denotes $S_{\text{cache}}^{\text{empty}}$. The pseudocode for the selection algorithm is described in Algorithm 1. This algorithm proceeds as follows:

Step 1: We should know whether the RDD_i has already been in memory.
Step 2: We select the reasonable RDD according to its number of use, while the sizes of the whole RDD partitions should be less than the free memory which is used to cache partitions.

5.2. Replacement algorithm

As we know, weight is used to measure the importance of the part in a graph, and in this paper, we use it to evaluate the importance of the partitions. We use V_P to denote the weight of partition.

Algorithm 1 Selection Algorithm

```
Require:
    Execution sequence $\{RDD_1, RDD_2, ...RDD_n\}$.
Ensure:
    Cached sequence.
1:  $M = \emptyset$;
2:  for ($i$ = 1 to $n$)
3:      if ($M \cap RDD_i \neq \emptyset$)
4:          continues;
5:      else if ($N_{RDD_i} > 2$ && $\sum_{j=1}^{h} S_{ij} > S_{cache}^{empty}$)
6:          Call Algorithm 2;
7:      else if ($N_{RDD_i} > 2$ && $\sum_{j=1}^{h} S_{ij} < S_{cache}^{empty}$)
8:          $M = M \cup RDD_i$;
9:      end if
10: end for
```

When we deal with large amounts of RDDs, especially in the process of iteration, data processing speed is an important performance evaluation. If the data used frequently is stored in the memory, next time you can access it much more quickly. Generally speaking, the memory which is used to cache partitions is finite and when many partitions are cached in memory, the memory is close to saturation. At that time, Spark will select LRU partitions and store them in disks according to the LRU algorithm.

When this algorithm is adopted to manage the memory, we find a problem that LRU cannot make sure LRU partition to be a reasonable one. When the system uses LRU algorithm to select the unused cache partition from memory for the longest time , and if P_{ij} is the result, then P_{ij} will be replaced. Although P_{ij} is used least recently, it is most likely that the cost of recomputing is significantly high. So, we can see LRU cannot make sure the P_{ij} to be the most valueless. In order to solve this problem, in this paper, we put forward a novel algorithm: we replace the partitions according to their weights. We take many experiments to compare the system with the LRU algorithm or WR algorithm. Most results show that when experiment is under the condition of limited memory, WR algorithm will have an advantage over the LRU algorithm.

When many partitions are cached in memory, we use Quicksort algorithm to sort the partitions according to the value of partitions. So, all partitions in one node are sorted according to the ascending order. Then we give new serial number according to the new order (e.g., $P^1, P^2, \ldots, P^q$). The pseudocode for WR algorithm is described in Algorithm 2. The chosen partitions which are stored in disk will be kept in collection Q. The free memory which is used to cache partitions in one node denotes $S_{\mathrm{node}}^{\mathrm{empty}}$. The algorithm consists of two steps and the algorithm proceeds as follows:

Step 1: We should know whether the new P_m has already been in memory.

Step 2: We traverse cached sequence and find the suitable P^i to be replaced, and P^i is stored in disk.

We give a simple example to explain the detailed steps of Algorithm 2 in Figure 5.

6. EXPERIMENTS

In this section, we use several experiments to evaluate the performance of the proposed algorithms mentioned previously. We deploy CloudStack to manage five servers, then we create six virtual machines. Each virtual machine has 100 G disk, 2.5 GHZ and runs Ubuntu 12.04 operation system while memory is variable, and we set it as 1G, 2G, or 4G in different conditions. The version for Scala is 2.10.4 while the Java Development Kit version is 1.8.0 25. We use Hadoop-2.4.0, Scala-2.10.4, and Spark-1.1.0 for all experiments, and use ganglia to observe the memory usage.

Algorithm 2 Weight Replacement Algorithm

```
Require:
    Cached sequence $\{P^1, P^2, ..., P^q\}$;
    The sizes of partitions $\{S_{P^1}, S_{P^2}, ..., S_{P^q}\}$;
    The weight of RDD $\{V_{P^1}, V_{P^2}, ..., V_{P^q}\}$;
    The free memory which is used to cache partitions in one node: $S_{node}^{empty}$;
    New partition: $P_m$;
    The size of new $P_m$: $S_{P_m}$.
Ensure:
    New cached sequence.
 1: $Q = \emptyset$;
 2: for ($i$ = 1 to $q$)
 3:     if($P_m$ equals to $P^i$)
 4:         break;
 5:     else if ($i == q$)
 6:         for ($i$ = 1 to $q$)
 7:             if (($V_{P^i} < V_{P^m}$) && (($S_{P^i} < S_{P^m}$ && ($S_{node}^{empty} + S_{P^i} > S_{P^m}$)) || $S_{P^i} > S_{P^m}$ ))
 8:               $P_m$ replaces $P^i$;
 9:             else
10:               $Q = Q \cap P^i$;
11:             end if
12:         end for
13:     end if
14: end for
```

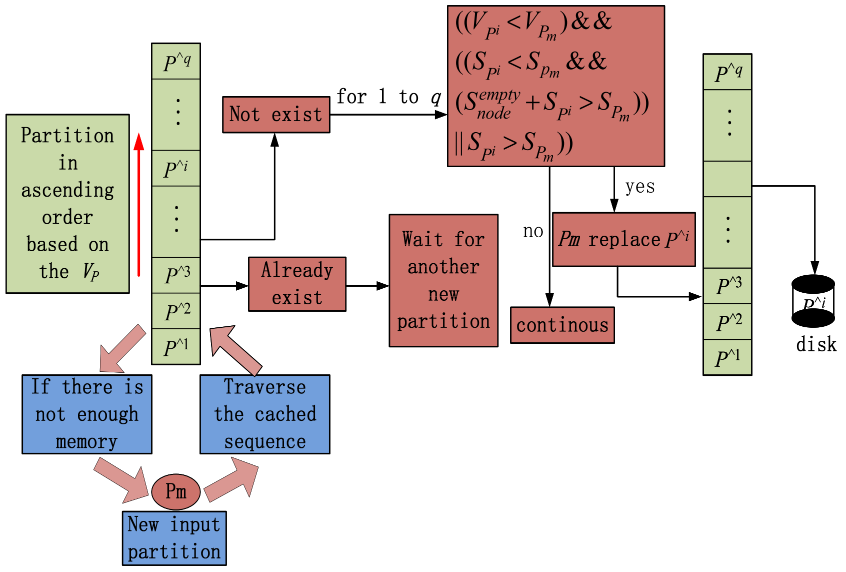

Partition in ascending order based on the V_P
P^q
P^i
P^3
P^2
P^1
Not exist
for 1 to q
$((V_{Pi} < V_{P_m})\&\& ((S_{Pi} < S_{P_m} \&\& (S_{node}^{empty} + S_{Pi} > S_{P_m})) \| S_{Pi} > S_{P_m}))$
yes
no
Already exist
Wait for another new partition
Pm replace P^i
continous
disk
P^i
If there is not enough memory
Traverse the cached sequence
Pm
New input partition

Figure 5. The operation of Spark computing process.

We choose 17 real graph datasets to do comparative experiments. All datasets will be showed in Table I. We use detailed format to express them. The 17 real datasets are obtained from SNAP [18]. The numbers of nodes and edges have a great influence on the execution time and memory usage. The reason why we choose the 17 real datasets is that they are directed and different among each other. More importantly, PageRank algorithm is running on directed graphs. The whole iterative process does not stop until the ranking process is convergence. These datasets are described in Table I.

Table I. Detailed information for datasets.

Name	Nodes	Edges	Description
p2p-Gnutella04	10,876	39,994	Gnutella peer to peer network from August 4, 2002
p2p-Gnutella24	26,518	65,369	Gnutella peer to peer network from August 24, 2002
wiki-Vote	7115	103,689	Wikipedia who-votes-on-whom network
p2p-Gnutella31	62,586	147,892	Gnutella peer to peer network from August 31, 2002
Cit-HepTh	27,770	352,807	Arxiv High Energy Physics paper citation network
soc-sign-Slashdot081106	77,357	516,575	Slashdot Zoo signed social network from November 6, 2008
Cit-HepPh	34,546	421,578	Arxiv High Energy Physics paper citation network
soc-sign-Slashdot090221	82,144	549,202	Slashdot Zoo signed social network from February 21, 2009
Soc-sign-epinions	131,828	841,372	Epinions signed social network
Slashdot0902	82,168	948,464	Slashdot social network from November 2008
Amazon0302	262,111	1,234,877	Amazon product co-purchasing network from March 2, 2003
Web-Stanford	281,903	2,312,497	Web graph of Stanford.edu
Amazon0312	400,727	3,200,440	Amazon product co-purchasing network from March 12, 2003
Wiki-Talk	2,394,385	5,021,410	Wikipedia talk (communication) network
web-Google	875,713	5105,039	Web graph from Google
cit-Patents	3,774,768	16,518,948	Citation network among US patents
soc-Pokec	1,632,803	30,622,564	Pokec online social network

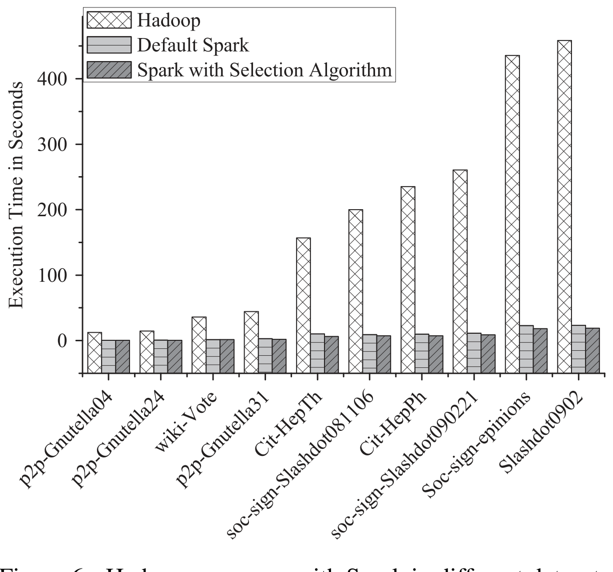
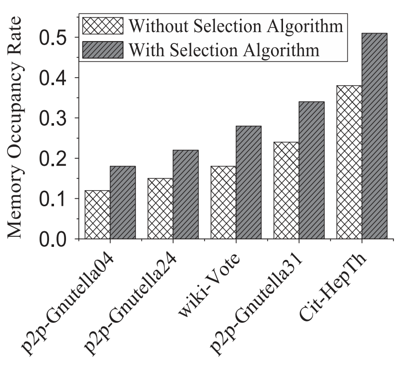


Figure 6. Hadoop compares with Spark in different datasets.

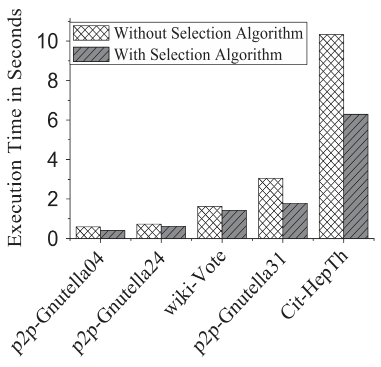
6.1. The performance of Spark with selection algorithm compared with Hadoop

First, we use some datasets to prove that Spark dealing with iterative computing is faster than Hadoop in the same condition. Our experiments are carried out on six virtual machines and their memory is 2G. In Figure 6, we use ten different datasets to compare their performance with PageRank algorithm, and each experiment has been performed more than three times, then we compute their average.

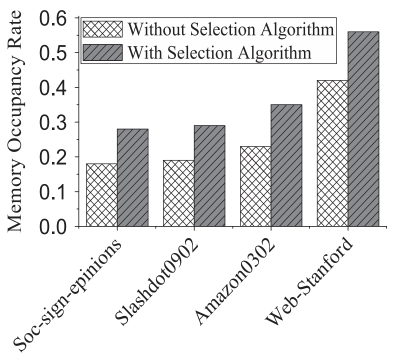
From the graph, we can see that when there is enough memory, Spark with our selection algorithm shows a significant advantage on iterative computation, and outcomes Hadoop by 20× to 30× because iterative computations are carried out in memory on Spark, while the computation on Hadoop is based on disk computation, and it should constantly read from or write into HDFS,



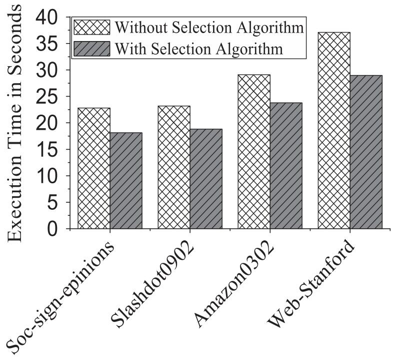
(a) Memory Occupancy Rate in 1G memory



(b) Execution Time in 1G memory



(c) Memory Occupancy Rate in 2G memory



(d) Execution Time in 2G memory

Figure 7. Execution time with selection algorithm in different memory.

which heavily increases the overhead. Meanwhile, compared with default Spark, we find that Spark with our selection algorithm spends less time to finish the computation. Because of small datasets and enough memory, our selection algorithm does not have an obvious effect on the performance of Spark.

6.2. *Selection algorithm used in enough memory*

Computation carried out in memory is a huge advantage in Spark framework. When we cache the repeated RDDs partitions, it will enhance the system performance and minimize the execution time. We will use the PageRank algorithm to test the system performance under different conditions. We use nine datasets to verify whether the selection algorithm will boost the system performance in different memory, and we record the execution time. Figure 7(a) and (c) gives each dataset average memory usage rate based on their usage curve from ganglia when each curve is in a stable condition. At the same time, by using ganglia, we can make sure each dataset computes with enough memory. Figure 7(b) and (d) gives each dataset corresponding execution time in 1G and 2G memory.

When each dataset is computed in enough memory, a replacement algorithm will not play an important role, while our selection algorithm will enhance the system performance. According to the scheduling model, it chooses the multiple occurrences of RDDs to cache their partitions in memory. If the identical RDDs appear again, it does not need to recompute them, which reduces the system computation cost and speeds up the whole iterative computation.

It can be seen in Figure 7(a) and (c) that when increasing the number of nodes and edges, memory occupancy rate gets higher because Spark computation is carried out in memory and memory is not released until the last iterative computation. More importantly, PageRank is a data-intensive application, which needs large amounts of overhead including taking up much memory. In addition, the iterative computation with selection algorithm selects multiple occurrences of RDDs to cache their partitions in memory, so the memory occupancy rate of Spark is higher than it without the algorithm.

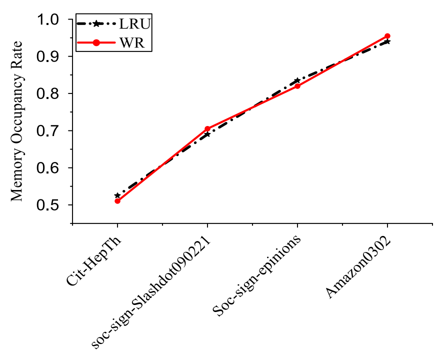
In Figure 7(b) and (d), it can be seen that an iterative calculation process with selection algorithm spends less time, while the same dataset without the algorithm costs more time to finish computation. The main reason is that our selection tries its best to select more recurring variables to be cached, so that the subsequent computations which demand the variables can read the data from memory instead of obtaining those through the remote transmission during the computation process. Thus, our algorithm speeds up the whole computation. Compared with our selection algorithm, that default Spark caches the recurring variables is decided by programmers, which is full of uncertainties. Different programmers will choose different strategies, causing different results. It is likely to cache many frequently used variables while miss many high frequency used variables. More importantly, whether the process should cache the variables is mainly based on the experience. The uncertainty cannot adequately exploit the strengths of the *persistence* characteristic of Spark. Although Spark with our algorithm likely uses more memory, which seems more cost, relative to the uncertainties of default Spark, even errors caused by programmers, our selection algorithm improves the system efficiency and enhances the system reliability. Meanwhile it makes the system run faster, and improves its performance. Relative to the more cost caused by our selection algorithm, Spark with our algorithm gains much larger value.

6.3. WR algorithm in different conditions

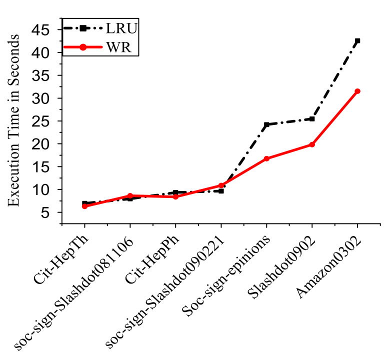
When Spark uses selection algorithm to select many RDDs to cache their partitions in memory while the memory is approach to saturation, it uses LRU replacement algorithm to manage and reallocate memory. Replacement algorithm plays an important role in this condition, and we compare LRU replacement algorithm with our WR algorithm. We use all datasets listed to test with PageRank algorithm under the condition of 1G, 2G, and 4G. We observe the memory usage rate in real time from ganglia. After we conduct experiments repeatedly, we find that the plots of five slaves are roughly the same. The reason for it is that a Spark platform adopts load balancing policy to manage the resource, and all the workers use the uniform distribution for datasets. So we only show the memory usage of one node, and we just illustrate the average memory usage rate during the stable process of iterative computation. In Figure 8 (a),(c), and (e), we will show memory occupied in 1G, 2G, and 4G conditions, and at the same time, we observe the execution time of iterative computation process by using the LRU algorithm and the WR algorithm in Figure8 (b),(d), and (f).

It can be learnt in Figure8(a),(c), and (e) that when the iterative computation is with WR policy, the memory occupancy rate is almost the same as the computation with LRU policy. This is primarily due to the fact that whether Spark runs with the LRU algorithm or the WR algorithm, it uses selection algorithm to select the value RDDs, and the LRU algorithm or the WR algorithm is the replacement algorithm which does not have a great effect on memory size.

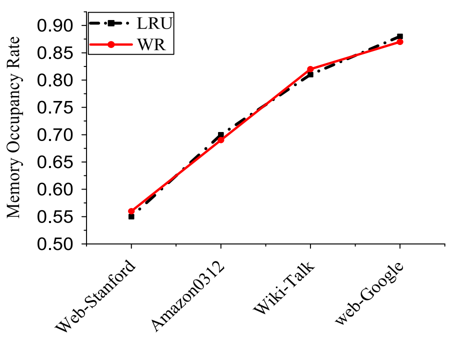
In Figure 8(b),(d), and (f), we can see that the execution time with LRU algorithm or WR algorithm is roughly close to each other, because the replacement policies do not have much effect on the performance in enough memory. With increasing the number of nodes and edges, there will be no enough memory to cache the partitions, and replacement policies will play an important role in this computation process. The three figures show that Spark handles computation with WR algorithm faster than it with LRU algorithm in insufficient memory. The most important reason is that LRU algorithm only takes an attention on whether the partitions are recently used, while WR algorithm considers the sizes of partitions, the computation cost for partitions, and the number of use for partitions. The partitions which are not used recently do not mean that they will not be used frequently in the future. When the weights of replaced partitions are higher, it needs more time to read them from disk or recompute them according to the *lineage*. The partitions chosen by LRU algorithm cannot make sure it is the suitable one to be replaced while WR algorithm which considers partition size, its computation cost, and its number of use makes sure the replaced partition to be more valueless. Because its reasonable consideration makes Spark with WR algorithm runs faster than it is with LRU algorithm.



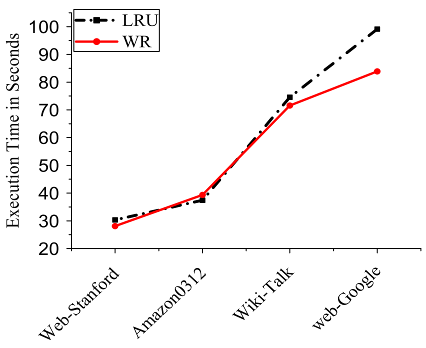
(a) Memory occupancy Rate in 1G memory



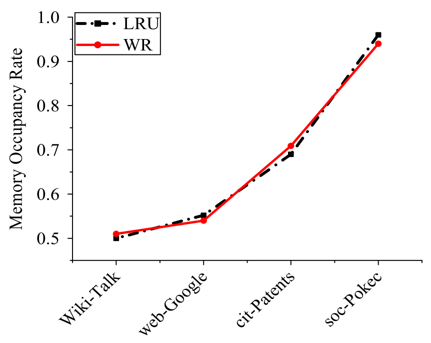
(b) Execution Time in 1G memory



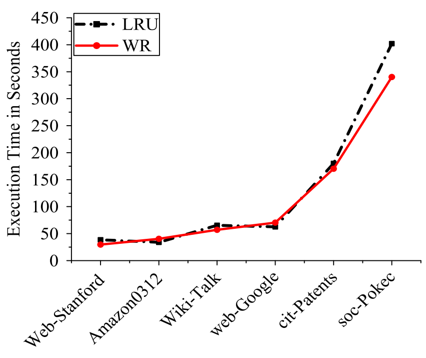
(c) Memory occupancy Rate in 2G memory



(d) Execution Time in 2G memory



(e) Memory occupancy Rate in 4G memory



(f) Execution Time in 4G memory

Figure 8. WR algorithm compared with least recently used (LRU) algorithm for different memory. WR, weight replacement.

7. CONCLUSION

Spark is an effective distributed computation framework for processing amounts of data-intensive applications. It has a better advantage over Hadoop on iterative computations because all computations in Spark are based on memory. However, Spark does not make a good mechanism to choose reasonable RDDs to cache their partitions in memory, which does not make full use of in-memory storage mechanism. So we use our proposed selection algorithm to choose value RDDs according to DAG before tasks are performed. The results show that iterative computation with selection algorithm handles faster than without that.

When the memory which is used to cache partitions is full of partitions, Spark adopts the LRU algorithm to select the LRU partitions to be replaced. It does not consider other factors, while our proposed WR algorithm considers the partitions' computation cost, the number of use for partitions, and the sizes of the partitions. Experiments show that Spark with the WR algorithm shows better performance than LRU policy in insufficient memory.

ACKNOWLEDGEMENTS

The authors are grateful to the anonymous reviewers for their constructive comments to improve the manuscript. The research was partially funded by the Key Program of National Natural Science Foundation of China (grant nos. 61133005 and 61432005), and the National Natural Science Foundation of China (grant nos. 61370095, 61472124, 61202109, and 61472126).

REFERENCES

1. Zhang X, Liu C, Nepal S, Yang C, Dou W, Chen J. SaC-FRAPP: a scalable and cost-effective framework for privacy preservation over big data on cloud. *Concurrency and Computation: Practice and Experience* 2013; **25**(18):2561–2576.
2. Howe D, Costanzo M, Fey P, Gojobori T, Hannick L, Hide W, Hill DP, Kania R, Schaeffer M, St Pierre S, Twigger S, White O, Yon Rhee S. Big data: the future of biocuration. *Nature* 2008; **455**(7209):47–50.
3. Jacobs A. The pathologies of big data. *Communications of the ACM* 2009; **52**(8):36–44.
4. Lohr S. The age of big data. *New York Times* 2012; **11**.
5. Wikipedia, Apache Spark. (Available from: http://en.wikipedia.org/wiki/Apache_Spark) [Accessed on 4 December 2014].
6. Zaharia M, Chowdhury M, Das T, Dave A, Ma J, McCauley M, Franklin MJ, Shenker S, Stoica I. Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation USENIX Association* 2012:2–2.
7. Lin X, Wang P, Wu B. Log analysis in cloud computing environment with Hadoop and Spark. *Broadband Network & Multimedia Technology (IC-BNMT), 2013 5th IEEE International Conference on,* IEEE, 2013:273–276.
8. Kim J C, Lee D, Lee CG, Kim K. RT-PLRU: A new paging scheme for real–time execution of program codes on nand flash memory for portable media players. *Computers, IEEE Transactions on* 2011; **60**(8):1126–1141.
9. Robinson JT, Devarakonda MV. *Data Cache Management Using Frequency-based Replacement*, vol. 18, ACM, New York, NY, USA, 1990; 134–142.
10. Lee D, Choi J, Kim J H, Noh S H, Min S L, Cho Y, Kim C S. On the existence of a spectrum of policies that subsumes the least recently used (LRU) and least frequently used (LFU) policies. In *ACM SIGMETRICS Performance Evaluation Review*. vol. 27, ACM: New York, NY, USA, 1999; 134–143.
11. Lee D, Choi J, Kim J H, Noh S H, Min S L, Cho Y, Kim C S. LRFU: a spectrum of policies that subsumes the least recently used and least frequently used policies. *IEEE transactions on Computers* 2001; **50**(12):1352–1361.
12. Young NE. On-line file caching. In *Proceedings of the Ninth Annual ACM-SIAM Symposium on Discrete Algorithms, Balitmore, MD*. ACM: New York, NY, USA, 1999; 82–86.
13. Forney B C, Arpaci-Dusseau A C, Arpaci-Dusseau R H. Storage-aware caching: Revisiting caching for heterogeneous storage systems. *FAST*, vol. 2, Citeseer, 2002; 5–5.
14. Swain D, Paikaray B, Swain D. AWRP: adaptive weight ranking policy for improving cache performance. *CoRR* 2011; **1107**(4851). (Available from: http://arxiv.org/abs/1107.4851).
15. Apache spark. (Available from: http://spark.apache.org/) [Accessed on 4 December 2014].
16. Zaharia M, Chowdhury M, Das T, Dave A, Ma J, Mccauley M, Franklin M, Shenker S, Stoica I. Fast and interactive analytics over Hadoop data with Spark. *USENIX; login* 2012; **37**(4):45–51.
17. Rodriguez Sossa M, Buyya R. Deadline based resource provisioning and scheduling algorithmfor scientific workflows on clouds. *Cloud Computing, IEEE Transactions on* 2014; **2**(2):222–235.
18. Stanford SNAP .online. (Available from: http://snap.stanford.edu/) [Accessed on 4 December 2014].