

# LCS: An Efficient Data Eviction Strategy for Spark

Yuanzhen Geng<sup>1</sup> · Xuanhua Shi<sup>1</sup>  · Cheng Pei<sup>1</sup> ·  
Hai Jin<sup>1</sup> · Wenbin Jiang<sup>1</sup>

Received: 18 October 2016 / Accepted: 26 October 2016 / Published online: 2 November 2016  
© Springer Science+Business Media New York 2016

**Abstract** As an in-memory distributed computing system, Spark is often used to speed up iterative applications. It caches intermediate data generated by previous iterations into memory, so there is no need to repeat the generation when reusing these data later. This sharing mechanism of caching data in memory makes Spark much faster than other systems. When memory used for caching data reaches the capacity limits, data eviction will be performed to supply space for new data, and the evicted data need to be recovered when they are used again. However, classical strategies do not aware of recovery cost, which could cause system performance degradation. This paper shows that the recovery costs have significant difference in Spark, thus a cost aware eviction strategy can obviously reduces the total recovery cost. To this end, a strategy named LCS is proposed, which gets dependencies information between cache data via analyzing application, and calculates the recovery cost during running. By predicting how many times cache data will be reused and using it to weight the recovery cost, LCS always evicts the data which lead to minimum recovery cost in

---

✉ Xuanhua Shi  
xhshi@hust.edu.cn

Yuanzhen Geng  
yzgeng@hust.edu.cn

Cheng Pei  
peicheng@hust.edu.cn

Hai Jin  
hjjin@hust.edu.cn

Wenbin Jiang  
wenbinjiang@hust.edu.cn

<sup>1</sup> Services Computing Technology and System Lab & Cluster and Grid Computing Lab & Big Data Technology and System Lab, School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan 430074, China

future. Experimental results show that this approach can achieve better performance when memory space is not sufficient, and reduce 30–50% of the total execution time .

**Keywords** Spark · Eviction strategy · Cache management

## 1 Introduction

Many big data analysis techniques used by high-tech enterprises have iterative characteristics, including using graph computation to do PageRank or social network analysis, using machine learning to do clustering or regression analysis. Such applications have a common characteristic that data needs to be processed iteratively until meet the convergence or end condition, meanwhile large amount of data needs to be reused between iterations [7].

Data reusing mechanism in traditional MapReduce systems [1,8] is based on disk and introduces large amount of disk I/O that affects performance badly. As an in-memory system, Spark [20] accelerates execution speed by fully utilizing memory. It provides an data abstraction called *Resilient Distributed Dataset* (RDD) [19], by caching RDDs into memory and sharing them between iterations, Spark achieves much better performance than MapReduce.

Spark is runs on *Java Virtual Machine* (JVM), and may suffer from severe memory bloat problem [6]. When dealing with iterative applications, multiple kinds of RDD need to be cached, and their memory occupation is much larger than the input data size [11]. For the sake of not affecting normal running, eviction strategy is needed when memory space is not sufficient.

Recovery evicted partitions of cache RDDs will experience different series of operations, and variations exist in recovery costs. It is desirable to retain cache partitions with higher rather than lower recovery costs during eviction. But among current eviction strategies, *First In First Out* (FIFO) focuses on the create time, *Least Recently Used* (LRU) focuses on access history for better hit ratio, none of them takes cost into consideration, so a cost-aware eviction strategy is needed. As a possible solution, the GreedyDual [18] or GD-Wheel [13] algorithm can make eviction decision for items with non-uniform cost. Both of them have taken access history and costs of cache items into consideration. But, as the execution logic of upcoming phase is known in Spark, access history has no help to eviction strategy. In other words, we need an eviction strategy integrating recovery cost of cache partition and execution logic of application in Spark.

In this paper, we analyze disadvantages of existing eviction strategy and state the necessity of taking recovery cost into consideration at first. Then we propose an eviction strategy named *Least Cost Strategy* (LCS).<sup>1</sup> LCS involves three steps. First, it gets the dependencies of RDD by analyzing application, and predicts how many times cache partitions will be reused. Second, it collects information during partition cre-

---

<sup>1</sup> We have made LCS open source at Github, and also added the patch at Apache Software Foundation. The web links are <https://github.com/SCTS/Spark-LCS> and <https://issues.apache.org/jira/browse/SPARK-14289>, respectively.

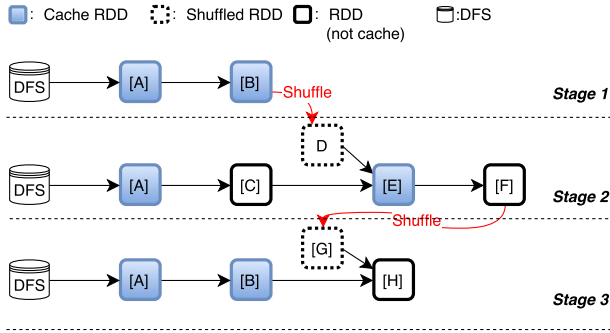


Fig. 1 DAG representation of an application. Stages are divided by Shuffle operation

ation, and predicts the recovery cost. Third, it maintains the eviction order using the above two information, and evicts the partition that incurs the least cost when memory is not sufficient. This paper describes the design and implementation of LCS in detail, and experiments have been carried out in typical iterative applications. Extensive experiments have been taken under LCS, LRU, FIFO. Results show that our strategy achieves a better performance when eviction happens, and the total execution time of application can be reduced by 30–50%.

The rest of this paper is organized as follows. In Sect. 2, we provide a overview of Spark and discuss the motivation. The overall implementations and details of LCS are presented in Sect. 3. Extensive experimental results are reported in Sect. 4. Related work is discussed in Sect. 5. Section 6 conclude this paper.

## 2 Background and Motivation

### 2.1 Background

As a representative in-memory distributed computing system, Spark introduces RDD as data abstraction which has multiple partitions and can be cached in memory.

RDD is immutable and can only be transformed from other RDD or disk file by using coarse-grained operations implemented inside Spark. These operations are divided into *transformations* and *actions*. Transformations are lazy operations which create a series of new RDDs while actions will launch a computation job to return a result or write data to external storage. Each job is split into several stages based on *Shuffle dependencies* among RDDs, which denotes that each partition of the child RDD depends on all partitions of the parent RDD(s).

Execution logic of stages in upcoming job are represented by *Directed Acyclic Graph* (DAG), as shown in Fig. 1. DAG is considered to determine the work flow of RDD and the persistence of RDD in memory. Spark splits the stages and determines the execution orders of stages all by DAG structures, it is not necessary to get the work flow by logs or program analysis [17].

Memory space of executor node can be roughly divided into two fractions: one is used for caching RDD, the other is used for computing. By default 60% of the memory

**Table 1** Details about common transformations

Transformation	Execution time (s)	Input size (GB)	Result size (GB)	R/W disk	Shuffle size
<i>textFile</i>	65	24.3	140.8	✓ ×	×
<i>map</i>	7	140.8	65.6	× ×	×
<i>reduceByKey</i>	8	65.6	3.3	✓ ✓	405 MB
<i>groupByKey</i>	25	65.6	10.8	✓ ✓	11.3 GB
<i>sortByKey</i>	162	65.6	120.3	✓ ✓	7.7 GB
<i>filter</i>	2	65.6	34	× ×	×
<i>distinct</i>	341	65.6	65.6	✓ ✓	12.1 GB

space is for cache RDD storage. Spark allows user to determine whether and when to cache RDD in memory by using *persist* API, partitions of cache RDD can be accessed and operated at memory speed. Spark only allows users to use *unpersist* API to remove all the partitions of a cache RDD, the *unpersist* operation happens after a job finished.

When remaining space for cache storage is not sufficient during job running, some partitions of cache RDD will be evicted from memory. Those partitions must be recovered before reusing them again, and the recovery process has two situations as follows. Unless user allows using deserialization for some RDDs, only recomputation can be used for recovery.

## 2.2 Existing Problems and Motivation

While running iterative applications in Spark, generally more than one RDDs need to be cached for reusing. When carrying out PageRank (3 iterations) and ConnectedComponents provided by GraphX on twitter-rv [5] dataset, the peak size of RDDs need to be cached is 155.8 and 357.1 GB, respectively, which is much larger than input (24.3 GB) resulted by the bloat of Java reference [15]. As a result, Spark could only deal with relative small input dataset if users want all cache RDDs in memory without eviction, this will restrict the usage of Spark, so the need for eviction strategy is urgent.

We first perform several common transformations on the twitter-rv dataset (experimental environment is shown in Sect. 4) to show the cost of transformations in Table 1. But, as the default eviction strategy, LRU does not consider recovery cost. When the evicted partitions of RDDs are used later, it will incur recovery cost. For example, when memory for cache storage is not sufficient during the creation of RDD E in Fig. 1, the eviction of RDD A may result in this problem. As a conclusion, for better performance, the eviction strategy should assure that recovering evicted partition brings the minimize cost in the future.

LRU may also lead to the problem of straggler. For fully utilizing cluster's resources, the number of tasks in a stage is usually several times of CPU core number. Taking Fig. 1 as an example again, and use RDD<sub>n</sub> to represent the partition set accessed by *n*th wave of tasks. After the first stage finishing, if some cache partitions (A<sub>1</sub>, B<sub>1</sub> or other similar group) need to be evicted in stage 2, it may lead to serious straggler in stage 3, as situation shows in Fig. 2a. But if we take recovery cost into account for

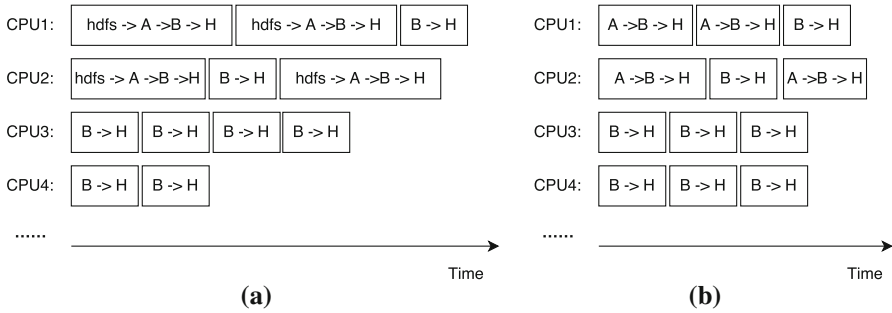
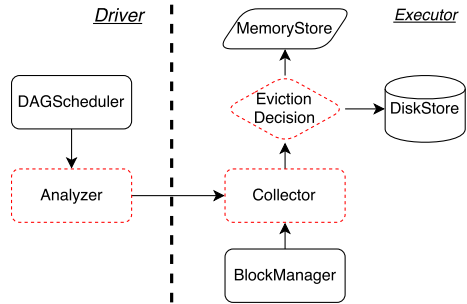


Fig. 2 a LRU may lead to serious straggler, b no serious straggler

Fig. 3 Overall architecture of our approach. Modules with dash outlines are components introduced in our approach



eviction, because most of the evicted partitions belong to RDD B, no serious straggler happens as Fig. 2b illustrates.

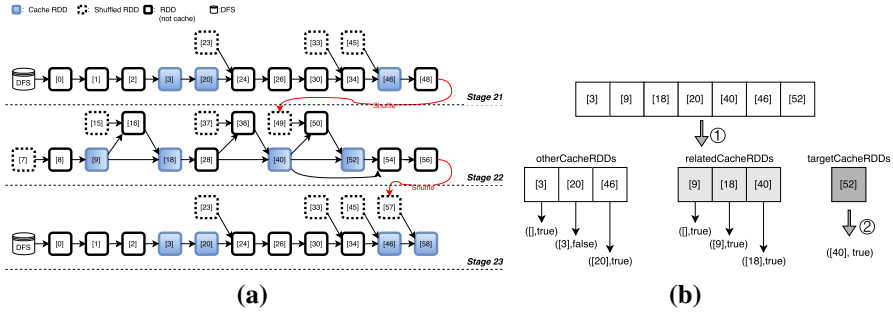
### 3 Design and Implementation

#### 3.1 Overall Architecture

LCS aims to control the eviction of cache partitions by analyzing the execution flow of application. The overall architecture is shown in Fig. 3, three necessary steps are carried out: (i) *Analyzer* in driver node analyzes the application by the DAG structures provided by *DAGScheduler*; (ii) *Collector* in each executor node records information about each cache partition during its creation; (iii) *Eviction Decision* provides an efficient eviction strategy to evict the optimal cache partition set when remaining memory space for cache storage is not sufficient, and decide whether remove it from *MemoryStore* directly or serialize it to *DiskStore*.

#### 3.2 Analyzer

*Analyzer* will use DAG provided by *DAGScheduler*. Figure 4a shows an example DAG, it can be found that the starting points have two situations: DFS, files on it can be read from local or remote disk directly; Shuffled RDD, which can be generated by fetching



**Fig. 4** **a** DAG of job 4 in GraphX PageRank, **b** classify RDDs of PageRank and calculate information of each target RDD in *Analyzer*. Cache ancestor RDDs will be recorded in *AncestorsList*; if DFS or Shuffled RDD exists, *flag* will be set to *true*

remote shuffle data. DAG indicates the longest running path of task: when all the cache RDDs are missing, task needs to run from the starting points. But when cache RDD exists, task only needs to run part of the path from cache RDD by referring dependencies between RDDs. The aim of *Analyzer* is classifying cache RDDs and analyzing the dependency information between them before each stage runs. *Analyzer* only runs in driver node and will transfer result to executors when driver schedules tasks to them.

LCS has some modification to *unpersist*, which API only happens after job finished. However, as *DAGScheduler* already knows all the stages information in a job, the *unpersist* can be promoted to the last stage that uses those RDDs. By pre-registering RDD that needs to be *unpersist*, and checking whether it is used in each stage, we put it to the *RemovableRDDs* list of the last stage to use it. The purpose of design *RemovableRDDs* list is that after using partition of this kind of RDD, the partition can be evicted directly, and will not waste the memory resources.

Cache RDDs of a stage will be classified to three kinds: the first includes the current running cache RDDs (*targetCacheRDDs*); the second includes those participate in current stage (*relatedCacheRDDs*); the third includes other cache RDDs (*otherCacheRDDs*). For Fig. 4a, the classification is shown in Fig. 4b as step ①. After classification, LCS only needs to calculate the dependencies of *targetCacheRDDs* of current stage because other two kinds of RDDs are the target RDDs of the previous completed stages. The dependencies calculation is a *Depth First Search* algorithm in fact, and the search stops at the following three situations:

- Reaching ancestor RDD that has already been cached.
- Reaching shuffled RDD which can be recomputed by pulling shuffle data from other executor nodes.
- Reaching DFS where data can be read directly.

After the analysis, ancestor cache RDDs will be recorded in *AncestorsList*, a *flag* is used to indicate whether this cache RDD needs to shuffle or read data from disk during creation. The analyzed information indicates the creating path of each cache RDD, which will be used in *Collector*. The dependency analyzes is shown in Fig. 4b as step ②. It is important to note that if *RemovableRDDs* exist in a stage,

dependency information of RDDs that dependent on it needs to be updated in the next stage, as partitions of *RemovableRDDs* will all be removed after this stage finishes.

### 3.3 Collector

With the help of *Analyzer*, *Collector* will collect information about each cache partition during task running. Information that needs to be observed are listed below:

- Create cost: Time spent on computing cache partitions after all ancestor cache partitions are found in memory, denoted as  $C_{create}$ . If data need to be read from DFS or transferred by shuffle operations, those costs are also added to  $C_{create}$ .
- Eviction cost: Time costs when evicting a partition from memory, called  $C_{eviction}$ . If partition is serialized to disk, the eviction cost is the time spent on serializing operation and writing into disk, denoted as  $C_{ser}$ . Otherwise, partition will be removed directly, the eviction cost is accounted as zero here.
- Recovery cost: Time costs when partition data are not found in memory, named  $C_{recovery}$ . If partition is serialized to disk, the recovery cost is the time spent in reading from disk and deserialization, denoted as  $C_{deser}$ . Otherwise, partition needs to be recomputed by lineage information, represented as  $C_{recompute}$ .

*Recovery Cost* can only be obtained after finishing recovering lost RDD partition data. So a new way is required to estimate it. As serialize and deserialize speeds are relevant to RDD type, we serialize and deserialize several partitions of each RDD at first and get the time spent per MB data, denoted as  $SPM$  and  $DPM$ . As RDD's recomputation will repeat the computing flow which is the same with its creation, we take  $C_{create}$  as an approximate value of  $C_{recompute}$ . We use the following Eq. 1 to calculate  $C_{eviction}$  and  $C_{recovery}$ .

$$\begin{aligned} C_{eviction} + C_{recovery} &= \min (0 + C_{recompute}, C_{ser} + C_{deser}) \\ &\approx \min (C_{create}, (SPM + DPM) * size) \end{aligned} \quad (1)$$

Then the question is transformed to calculate the  $C_{create}$  value, and the calculation is by the following steps:

1. Referring to the corresponding *AncestorsList* of RDDs, the ancestor partitions list  $L$  can also be known.
2. If ancestor partitions are not in memory, they will be recovered and their *Recovery Cost* will be updated. The time spent on recovering the missing ancestor partitions which are also needed to be cached is not counted to avoid double counting.
3. Record the start time *startTime* when all ancestor partitions could be found in memory.
4. Read DFS or transfer shuffle data if necessary, and record the end time *endTime* and *size* of target partition after putting it into memory.

As partitions data have different size,  $CPM = (endTime - startTime) / size$  is used to represent the cost per MB file. So each partition will has a (partitionID, ( $CPM$ , *size*)) pair.

Besides recovery cost, the LCS strategy also uses *reusability* of partition as the weight. As illustrated before, the execution logic of upcoming stages in a job is already known, so does the upcoming access or generating logic of cache RDDs. For each partition, we use *reusability* to denote how many times it will be reused in the future. This weight can be used to avoid evicting partitions with relative small *CPM* but will be frequently used. By referring to dependencies, we calculate how many times a cache partition will be used during creating target cache RDD of remaining stages in a job, and set this number to *reus* value of the partition. After using a cache partition, the *reus* value will reduce by one. After using partition of *RemovableRDDs*, the *reus* will become 0, and needs Eq. 2 to update *CPM* of partition dependent on it. A *RemovablePartitions* list is used to recode partitions with *reus* value of 0.

$$CPM_{renew} = \frac{CPM_{ancestor} * size_{ancestor} + CPM * size}{size}. \quad (2)$$

### 3.4 Eviction Decision

Through using information provided by *Collector*, each cache partition has a *WCPM* value calculated by Eq. 3. When memory is not sufficient, *Eviction Decision* will evict partitions in *RemovablePartitions* at first, then choose the partition with the least *WCPM* value. For a certain partition that needs to be evicted, if  $CPM * reus$  is smaller than  $SPM + DPM * reus$  when calculating *WCPM*, *Eviction Decision* will remove it directly from memory, otherwise will serialize it to disk as recovery by deserialization cost fewer time than recomputation.

$$WCPM = \min (CPM * reus, SPM + DPM * reus) \quad (3)$$

By this strategy, LCS always evicts cache partitions that bring in the fewest recovery cost in future, and guarantees that total recovery cost is fewer in entire lifetime of application than other traditional strategies.

## 4 Evaluation

### 4.1 Evaluation Environment and Method

The experiment platform includes a cluster with eight nodes, one as the master and the remaining act as the executors. Each node is equipped with two 8-core 2.6GHz Intel Xeon E5-2670 CPUs, 64GB memory and a 10,000-RPM-SAS disk, and we adopt HDFS for storage, each block has three replications. Data blocks are balanced before applications running. The JVM adopted is HotSpot 64-Bit Server VM, and the version is 1.7.0\_67. *Swappiness* is configured to 0 to prevent the OS from swapping memory page to swap space (located on disk). Version of Spark used is 1.5.2 excepts subsection 4.5, which uses the newest 2.0-snapshot version to test the newly introduced memory mode.



**Table 2** Overall performance result

Application	LRU (s)	LRU-SER (s)	FIFO (s)	FIFO-SER (s)	LCS (s)	LCS-SER (s)
PR	1285	854	1411	886	698	632
CC	1346	1158	1444	1091	855	829
KMeans	2681	2352	2204	2297	1500	1405

As LRU is the default strategy and FIFO is the most common strategy, we compare LCS with these two classical strategies. To prevent the serialization affecting performance and leading to unfair comparison, LRU, FIFO, LCS denote strategies that only use recomputation for recovery, and those with “-SER” suffix denote use both recomputation and deserialization for recovery.

The datasets are generated by HiBench [2]. In the rest of this section, PR and CC represents of PageRank and ConnectedComponents implemented by GraphX to process a 32.76 GB dataset in 3 iterations, respectively. KMeans represents of KMeans implemented by MLlib to process a 125.44 GB dataset in 5 iterations.

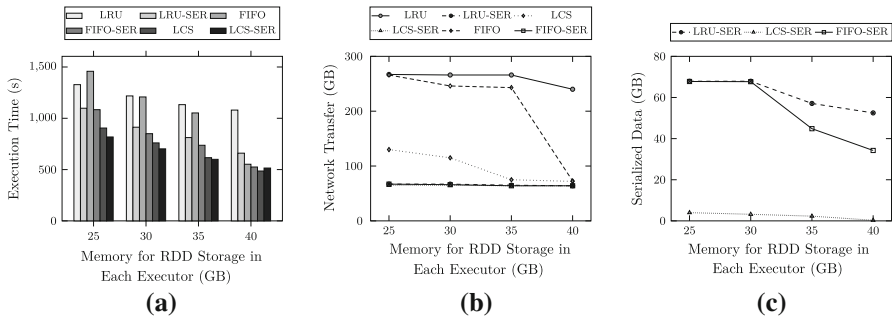
## 4.2 Overall Performance Result

By default, Spark allows to use 60% of executor memory space for cache storage. By setting memory that each executor occupies at 50 GB, Table 2 shows the performance of PR, CC, KMeans under six strategies. It shows that for PR and CC, performance of LCS is much better than LRU or FIFO (reducing 30–50% of the total running time). This is because in the default configuration, eviction happens frequently in both of these three applications, LRU and FIFO lead to lots of time cost in recomputation. There is no distinct difference between performances of LRU and FIFO, which indicates that the accession order used by LRU or creation order used by FIFO have no effect in Spark. LRU-SER and FIFO-SER show relatively better performance than LRU and FIFO, because they use deserialization to avoid recomputing partitions with high *CPM*, but LCS-SER can still reduce 20–40% time compared to them.

## 4.3 Influence of RDD Storage Size

Experienced users will configure the proportion of memory for RDD storage by application characteristics. For the I/O intensive application, its performance will not be promoted by increasing execution memory (which denotes the computing ability) if it is already sufficient. But performance can still be promoted by increasing memory for storage, because the increased space can be used for decreasing eviction.

When configuring memory of each executor to 50 GB, PR gets the best performance when execution memory is 10 GB (because PR is I/O intensive). So this subsection sets the execution memory as a constant value (10 GB) and changes the space for RDD storage to test its influence. Due to multiple RDDs need to be cached, eviction happens frequently in each experiment below, and the result is shown in Fig. 5a. It shows that



**Fig. 5** a Performance of PR, b network transfer size, c serialized data size

LRU only reduces 19% time when memory increases from 25 to 40 GB, FIFO reduces 28% time from 25 to 35 GB. This is because LRU or FIFO has no consideration of retaining partitions with the high recomputation cost when storage memory increases, therefore there is no effect on decreasing total running time. Experiments about LCS show that the completion time is reduced by 47% when memory for storage increases from 25 to 40 GB. LCS only needs 45–68% time of LRU. The reason is that more partitions with higher recomputation cost are stored in the additional memory space. As LCS-SER can automatically decide whether to serialize data, it shows 22–26% gain in performance than LRU-SER. Execution time between LCS and LCS-SER is close when memory space for RDD storage is configured as 35 and 40 GB, because with these configurations, a few partitions need to be serialized in LCS-SER.

#### 4.4 Network Transfer, Serialized Data and Straggler Results

Figure 5b shows that LCS also has ability to reduce the data size of network transmission as much as 70% compared with LRU. The reason is that LCS always chooses the partition with the least  $CPM$  value, which usually does not need shuffle operations to fetch remote data or just fetch a small amount during recomputation. LRU-SER, FIFO-SER, LCS-SER transfer fewer amount of data than both LRU and LCS, because they can use deserialization instead of recomputation to recovery the evicted partitions. As distributed computing systems are often constrained by network [12], reducing network traffic will have a great influence on performance improvement. LRU-SER, FIFO-SER, LCS-SER need to serialize some evicted data to disk, Fig. 5c compares the serialized data size of these three strategies. The result shows that LCS-SER serializes fewer data than others. One reason is that LRU-SER and FIFO-SER only allow users to serialize all evicted partitions of a RDD or not, but LCS-SER will not serialize some partitions if they have relatively small recomputation cost. The other reason is that LRU-SER and FIFO-SER will still serialize partitions of the removable RDD though they won't be used in later stages. While LCS-SER will not serialize this kind of partitions as their *reus* value is 0 (as illustrate in Sect. 3).

LCS won't lead to serious straggler situation compare to LRU. Table 3 shows execution time distribution of tasks in stage 22 of PR, it is clear that times of most

**Table 3** Time distribution results of tasks in stage 22

Strategy	Min (s)	25% (s)	Median (s)	75% (s)	Max (s)
LCS	11	13	18	22	32
LRU	13	30	34	39	60

tasks are obviously too long under LRU. In the ideal situation, tasks only need to generate the target RDDs from its direct ancestor cached RDDs. But PR under LRU shows that most of the long running tasks need to generate target RDD from accessing DFS file, in other words their creation path is much longer. Lots of execution time are wasted in waiting for straggler finish in LRU compared to LCS.

#### 4.5 Unified Memory Management

Spark has proposed *Unified Memory Management* (UMM) [3] in the latest version. UMM manages the memory much more intelligently, and users do not need to manually manage memory space. During different running phases, space of storage can be borrowed for faster execution. This means the execution ability during RDD recomputation may be different with its creation. UMM aims to provide better performance without user optimization. PR running with LRU under UMM (LRU-UMM) and original mode (LRU) take 1018 and 1285 s, respectively. While running with LCS under UMM (LCS-UMM) and LCS take 449 and 689 s, respectively. LRU-UMM can reduce 20% time than LRU, which denotes the performance gain of UMM. But LCS-UMM still reduces 36% time than LCS, and 56% time than LRU-UMM which indicates LCS can still work in UMM.

### 5 Related Work

Many eviction strategies have been proposed in in-memory systems. Memcached uses the classic LRU as the default strategy. While MemC3 [9], adopts the CLOCK algorithm to approximate LRU. When elements with relatively small size are stored, it is reasonable to adopt those classic algorithms because rebuilding those elements through reading disk costs little. As an eviction algorithm for key-value store, CAMP [10] has taken the cost into account. But there are significant differences about the size or cost between in-memory data of big data system and key-value pairs. As an in-memory distributed file system, Tachyon [14] aims to improve data sharing between different systems. Though Tachyon has provided Edge Algorithm to bound the recomputation cost when checkpointing data among multiple jobs, it is not suitable for a single job because it has no help to bound the recomputation cost to less than job's completion time. PACMan [4] provides sufficient properties about input data and caches them in memory for performance. Two eviction strategies are proposed inside for completion time and cluster efficiency respectively. However, they do not consider the properties of intermediate data and can not be applied to iterative application.

FACADE [16] goes deep into compiler to manage memory for big data systems. But optimizable space is still left for eviction strategy to manage memory in high-level.

## 6 Conclusion

In this paper, we propose an eviction strategy for Spark named LCS. During application running, LCS analyzes execution logic and collects information about cache data. LCS guarantees that the evicted data always have the minimum total recovery costs in the future. Therefore the memory pressure will be mitigated and the amount of swap-out data reduces a lot. Experiments show that LCS is efficient to deal with the situation when memory space is insufficient, and the execution time can be reduced by up to 50% compared to the classic eviction strategies.

**Acknowledgements** This paper is partly supported by the NSFC under Grant Nos. 61433019 and 61370104, International Science and Technology Cooperation Program of China under Grant No. 2015DFE12860, National 863 Hi-Tech Research and Development Program under Grant No. 2014AA01A301.

## References

1. Hadoop, A. <http://hadoop.apache.org>
2. HiBench. <https://github.com/intel-hadoop/HiBench>
3. Unified Memory Management. <https://issues.apache.org/jira/browse/SPARK-10000>
4. Ananthanarayanan, G., Ghodsi, A., Wang, A., Borthakur, D., Kandula, S., Shenker, S., Stoica, I.: Pacman: coordinated memory caching for parallel jobs. In: Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation (NSDI), pp. 267–280 (2012)
5. Boldi, P., Vigna, S.: The webgraph framework I: Compression techniques. In: Proceedings of the 13th International Conference on World Wide Web (WWW), pp. 595–602 (2004)
6. Bu, Y., Borkar, V., Xu, G., Carey, M.J.: A bloat-aware design for big data applications. In: Proceedings of the 2013 International Symposium on Memory Management (ISMM), pp. 119–130 (2013)
7. Bu, Y., Howe, B., Balazinska, M., Ernst, M.D.: Haloop: efficient iterative data processing on large clusters. *Proc. VLDB Endow.* **3**(1–2), 285–296 (2010)
8. Dean, J., Ghemawat, S.: Mapreduce: simplified data processing on large clusters. In: Proceedings of the 6th Conference on Symposium on Operating Systems Design and Implementation (OSDI), pp. 137–150 (2004)
9. Fan, B., Andersen, D.G., Kaminsky, M.: Memc3: compact and concurrent memcache with dumber caching and smarter hashing. In: Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI), pp. 371–384 (2013)
10. Ghandeharizadeh, S., Irani, S., Lam, J., Yap, J.: Camp: a cost adaptive multiqueue eviction policy for key-value stores. In: Proceedings of the 15th International Middleware Conference (Middleware), pp. 289–300 (2014)
11. Gonzalez, J.E., Xin, R.S., Dave, A., Crankshaw, D., Franklin, M.J., Stoica, I.: Graphx: graph processing in a distributed dataflow framework. In: Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI), pp. 599–613 (2014)
12. Jalaparti, V., Bodik, P., Menache, I., Rao, S., Makarychev, K., Caesar, M.: Network-aware scheduling for data-parallel jobs: plan when you can. In: Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication (SIGCOMM), pp. 407–420 (2015)
13. Li, C., Cox, A.L.: Gd-wheel: a cost-aware replacement policy for key-value stores. In: Proceedings of the Tenth European Conference on Computer Systems (EuroSys), pp. 1–15 (2015)
14. Li, H., Ghodsi, A., Zaharia, M., Shenker, S., Stoica, I.: Tachyon: reliable, memory speed storage for cluster computing frameworks. In: Proceedings of the ACM Symposium on Cloud Computing (SoCC), pp. 1–15 (2014)

15. Mitchell, N., Sevitsky, G.: Building memory-efficient java applications: practices and challenges. PLDI Tutorial (2009)
16. Nguyen, K., Wang, K., Bu, Y., Fang, L., Hu, J., Xu, G.: Facade: a compiler and runtime for (almost) object-bounded big data applications. In: Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), pp. 675–690 (2015)
17. Lu, L., Shi, X., Zhou, Y., Zhang, X., Jin, H., Pei, C., He, L., Geng, Y.: Lifetime-based memory management for distributed data processing systems. In: Proceedings of the VLDB Endowment (PVLDB), pp. 936–947 (2016)
18. Young, N.E.: The k-server dual and loose competitiveness for paging. *Algorithmica* **11**(6), 525–541 (1994)
19. Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., Franklin, M.J., Shenker, S., Stoica, I.: Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. In: Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation (NSDI), pp. 15–28 (2012)
20. Zaharia, M., Chowdhury, M., Franklin, M.J., Shenker, S., Stoica, I.: Spark: cluster computing with working sets. In: Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing (HotCloud), p. 10 (2010)