A Memory-Aware Spark Cache Replacement Strategy

Jingyu Zhang[1,2], Ruihan Zhang[1], Osama Alfarraj[3], Amr Tolba[3], Gwang-Jun Kim[4*]

[1] School of Computer & Communication Engineering, Changsha University of Science & Technology, China
[2] Science and Technology on Information Systems Engineering Laboratory, School of Systems Engineering, National University of Defense Technology, China
[3] Computer Science Department, Community College, King Saud University, Saudi Arabia
[4] Department of Computer Engineering, Chonnam National University, South Korea
zhangzhang@csust.edu.cn, 19108041070@stu.csust.edu.cn, oalfarraj@ksu.edu.sa, atolba@ksu.edu.sa, kgj@chonnam.ac.kr

Abstract

Spark is currently the most widely used distributed computing framework, and its key data abstraction concept, Resilient Distributed Dataset (RDD), brings significant performance improvements in big data computing. In application scenarios, Spark jobs often need to replace RDDs due to insufficient memory. Spark uses the Least Recently Used (LRU) algorithm by default as the cache replacement strategy. This algorithm only considers the most recent use time of RDDs as the replacement basis. This characteristic may cause the RDDs that need to be reused to be evicted when performing cache replacement, resulting in a decrease in Spark performance. In response to the above problems, this paper proposes a memory-aware Spark cache replacement strategy, which comprehensively considers the cluster memory usage, RDD size, RDD dependencies, usage times and other information when performing cache replacement and selects the RDDs to be evicted. Furthermore, this paper designs extensive corresponding experiments to test and analyze the performance of the memory-aware Spark cache replacement strategy. The experimental data show that the proposed strategy can improve the performance by up to 13% compared with the LRU algorithm in different scenarios.

Keywords: Big data, Spark, Cache replacement, Memory resource utilization

1 Introduction

After the advent of Hadoop [1], distributed systems gradually replaced single-machine systems as a new processing platform for large data sets in various fields [2-4]. MapReduce specializes the split-apply-combine strategy [5], but the drop operation after each computation greatly affects the performance efficiency [6]. In order to solve the performance impact of frequent shuffle operations, Spark distributed computing framework based on Directed Acyclic Graph (DAG) optimization has gradually become the mainstream platform for big data processing. However, when all data can not be cached into storage memory, Spark's performance becomes poor. Spark's basic data structure is based on a distributed data abstraction called Resilient Distributed Dataset (RDD) [7]. When the memory storage is insufficient, if a new RDD needs to be cached, Spark will replace the RDD cache [8]. Currently, Spark's default cache replacement algorithm is the Least Recently Used (LRU) algorithm. This algorithm maintains a Linked Hash Map to evict the longest unused data from storage memory when RDDs need to be evicted. However, the cache replacement strategy based on LRU is not always effective. When the memory of the cluster is insufficient, the cache hit rate of the LRU algorithm will drop rapidly.

In order to improve the performance efficiency, scholars have conducted related research for Spark. The traditional cache replacement algorithms include First In First Out [9], Least Frequently Used [10] and so on [11-13], and they are the focus for performance improvement. In [14], Cao et al. proposed a novel LRU-K algorithm, which sets a data cache threshold K for a performance improvement. Some other researchers focus on the weight model of data block for replacement strategies [15-17]. [18] proposed an AWRP weight replacement algorithm, which obtains the weight of each data block by calculating the occurrence frequency of data blocks, the last time of use and the total number of accesses. Spark cache replacement strategy optimization based on DAG is another research direction [19-21]. Yu et al. proposed an LRC cache replacement scheme in [22], which obtained a DAG graph according to the RDD dependencies, and then used the number of dependencies of each RDD as the selection basis for the cache replacement strategy. There are also some studies that try to improve the performance of big data clusters from other directions [23-25]. However, most of the solutions may cause RDDs that are not used for a long time to consume the system's storage memory prematurely. And, most of the proposed algorithms do not analyze the future usage of RDDs, which may easily lead high-weighted but not reused RDDs to occupy memory.

In order to further optimize the memory management and performance for data analysis and processing, this paper conducts research on the RDD cache replacement algorithm in Spark systems, and proposes a novel memory-aware cache replacement method. The main contributions of this paper are mainly as follows:

1. This paper proposes a memory-aware Spark cache replacement strategy for performance improvement. We build the new RDD weight models for the proposed method, and analyze the key parameters.

*Corresponding Author: Gwang-Jun Kim; E-mail: kgj@chonnam.ac.kr
DOI: 10.53106/160792642022112306002

2. We design the extensive experiments for the proposed method. The experimental results show the better performance of our method compared with traditional algorithm.

The rest of this paper is organized as follows: Section 2 details the memory-aware Spark cache replacement strategy; Section 3 verifies the performance of the new cache replacement strategy through experiments; Section 4 concludes this work.

2 Model Design and Implementation

2.1 Cache Replacement Scheme Design

We present a new architecture for Spark cache replacement as shown in Figure 1. In this architecture, we design the Memory-Aware Cache Manager (MACM) component to communicate with other components in the cluster and guide the cache replacement process.

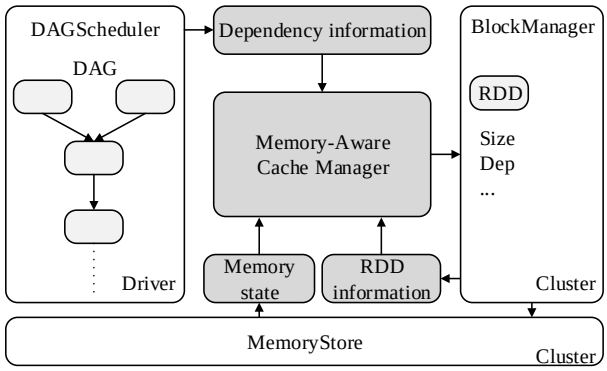

Figure 1. The architecture of the memory-aware Spark cache replacement strategy

When a job is submitted, MACM will extract the DAG information of the job from DAG Scheduler, and analyze it to obtain the number of dependencies and dependencies of each RDD. For the job, Whenever an RDD is cached, MACM will obtain the RDD size to be cached from the Block Manager. When a new RDD needs to be cached, MACM will judge the current system storage memory usage, and adaptively select the memory-aware Spark cache replacement algorithm or LRU.

2.2 Analysis of Weight Model

In order to optimize the existing cache replacement algorithm, it is necessary to analyze the weighting factors that may affect Spark cache performance. The notation table of all symbols used in this section is shown in Table 1.

1) RDD size. Spark's RDD computation is performed on a basic partition. Tasks are divided into Executors on each machine in the cluster for execution. The RDD size weight is the RDD partition size obtained from the acquireMemory() function. We can get the following.

$$S_{R_i} = M_{R_i} \tag{1}$$

Among them, S_{R_i} represents the size factor of RDD, M_{R_i} represents the RDD size obtained by the acquireMemory() function.

Table 1. Notations

Symbol	Description
R_i	The i-th RDD
fR_i	Parent RDD of the i-th RDD
M_{R_i}	RDD partition size obtained by the acquireMemory function
D_W	Wide dependency Function
D_N	Narrow dependency function
pN_{R_i}	The number of partitions of the i-th RDD
D_{R_i}	Recomputation cost function for the i-th RDD
OD_{R_i}	Out-degree of the i-th RDD
N_{R_i}	The number of times function of the i-th RDD
W_{R_i}	Weight function for the i-th RDD
λ_i	Weight function parameters
Er	Spark's efficiency drop rate
T_S	Execution time with sufficient memory
T_{inS}	Execution time with insufficent memory

2) RDD recomputation cost. RDDs with different dependencies have different costs to recompute after been evicted. Recomputing RDDs incurs varying amounts of communication overhead according to the dependencies. When there is a narrow dependency between the current RDD and its parent RDD, each Executor only needs to communicate with one corresponding Executor to complete the computation, and it is allowed to be performed in the form of a pipeline. When the current RDD and its parent RDD have a wide dependency, each Executor in the cluster needs communicate with all other Executors, resulting in huge network overhead within the cluster.

Therefore, this paper defines the RDD recomputation cost as the number of communications between Executors. Under wide dependencies, each partition in the parent RDD is used by multiple partitions of the child RDDs. Therefore, the recomputation cost of wide dependency is defined as the multiplication of the number of RDD_i partitions and the number of parent RDD partitions. The recomputation cost of wide dependencies is calculated as shown in Equation 2, where D_W represents the recomputation cost of wide dependencies, pN_{fR_i} represents the number of partitions for the parent RDD of the RDD_i, and pN_{R_i} represents the number of partitions of the current RDD.

$$D_W = \sum_{i=0}^{n} (pN_{fR_i} \times pN_{R_i}) \tag{2}$$

When the dependency is narrow, the Executor corresponding to each parent RDD partition only needs to communicate with the Executor corresponding to the partition of one child RDD, and the recomputation cost is defined as the number of parent RDD partitions of the RDD_i. The recomputation cost of narrow dependencies is shown in Equation 3, where D_N represents the computational cost of

narrow dependencies, and pN_{fR_i} represents the number of partitions for the parent RDD of the RDD_i.

$$D_N = \sum_{i=0}^{n} pN_{fR_i} \quad (3)$$

The total dependency weight can be defined as the sum of the recomputation cost of wide dependencies and the recomputation cost of narrow dependencies on the RDD_i dependency chain. The recomputation cost is denoted as shown in Equation 4. Among them, D_{R_i} represents the recomputation cost weight factor of RDD.

$$D_{R_i} = D_W + D_N \quad (4)$$

3) The RDD usage times. Before the job submitted by the Spark driver is called by the Spark Context, the processing of the specific data set will be put on hold, and the DAG Scheduler will first build it into a DAG according to the dependencies passed by the RDD. Therefore, the weight factor of the RDD usage times can be directly defined as the out-degree of RDD_i in the DAG.

$$N_{R_i} = OD_{R_i} \quad (5)$$

Among them, N_{R_i} represents the number of times the RDD_i is used, and OD_{R_i} represents the out-degree of the RDD_i obtained from the DAG Scheduler.

2.3 Algorithm Design

When the Spark memory is insufficient, the cache replacement method will be called more frequently, and hit rate will become lower than when the memory is sufficient. Under this situation, if the algorithm (e.g., LRU) predicts the possibility of future reuse based on historical conditions, it is easy to cause data blocks that need to be reused to be evicted and lead to a drop in cache hit rate and efficiency. When the Spark job has sufficient memory, the memory can accommodate the RDDs in more iterations. In this case, the accuracy of the LRU algorithm based on historical conditions will be greatly improved. LRU has the advantages of low time complexity and short algorithm execution time.

This paper designs a memory-aware cache replacement strategy. This strategy will collect the DAG information of the job from the DAG Scheduler when the Spark job starts. It also records the number of times that the RDD is been referenced. The algorithm divides the RDD into two queues. One queue called RCs_1 is for the RDD that needs to be reused, and another called RCs_0 is for the RDD that will not be used again according to the number of times the RDD is been referenced. For RDDs that will be reused, evicting them from the cache will inevitably result in overhead of retransmission or recomputation; for RDDs that will not be reused, they should be preferentially evicted from storage memory for better performance.

Algorithm 1. Memory-aware cache replacement algorithm

Input: Initiate RDD_new, Memory size S, RDD weight set w, RCs_0, RCs_1
Output:

```
Begin
  if(RCs_0 + RCs_1 >= 10) then
    Use LRU algorithm for cache replacement
  else
    var evictedRDDsize = 0
    Calculate RDD weight
    while(S+evictedRDDsize<RDD_new) do
      if (!isEmpty(RCs_0)) then
        EvictRDDs in RCs_0 by weight
      else
        EvictRDDs in RCs_1 by weight
      endif
      evictedRDDsize+=evictedRDD.size()
  endif
end
```

The pseudo-code of the memory-aware Spark cache replacement strategy is shown as Algorithm 1. In the strategy, when a new RDD needs to be cached, the algorithm traverses the maintained RDD queues and calculates the average RDD size. When the average size of RDD is less than the threshold K of the available memory space, it means that the cache is relatively sufficient. In this case, the memory contains RDDs used for iterations, and LRU has good performance. When the average RDD size is greater than the threshold K, it means that the cache hit rate of the LRU algorithm may decrease rapidly with the increase of RDD, and the memory-aware (MA) RDD cache replacement model will be used. In this study, the value of K defaults to 10%.

3 Experimental Results and Analysis

In order to verify the effect of the memory-aware Spark cache replacement algorithm, we conduct the extensive experiments on PageRank jobs on the Spark platform. As a comparison, we chose Spark's default cache replacement algorithm, LRU, to compare with our method. Fours typical PageRank datasets are Amazon0601, web-Google, web-Google, and com-DBLP for evaluation.

The experiments are based on Spark 2.1.0. Utilizing the memory-aware RDD weight model, the data in the storage memory will is replaced based on the RDD weight value. This section aims to compare and evaluate the memory-aware Spark cache replacement strategy and the existing LRU through comparative experiments.

The experimental platform configures Intel@ core i5-10400f 2.9GHz, and the RAM is 32GB. The operating system used by the cluster is Ubuntu 16.04, the Java development kit version is 1.8, the Scala version is 2.11.9, the Hadoop version is 2.7.1, and the Spark version is 2.1.0. In the experiments, the job execution time is obtained through the Spark console, and it is recorded by averaging the results of three runs.

3.1 Single-Job Evaluation

For each test, we use single-job workload for performance evaluation. We conduct two test groups with the same data set. We prepared four data sets: Amazon0601, com-dblp.all.cmty, facebook-combined, and web-Google. In order to better observe how the effect of the replacement strategy varies with the cached RDDs, the number of iterations is set to 2, 4, 6, 8, and 10.

Figure 2 shows the experimental results when the Spark memory size is 1G. Under the same Executor memory, the MA strategy has obvious performance optimization compared with the LRU. As the number of iterations increases, the Spark execution time using different cache replacement strategies increases, but the MA strategy has a relatively significant reduction in execution time compared with LRU. The MA strategy performs better in the Amazon0601 dataset and the web-Google dataset, but the performance difference of com-dblp.all.cmty dataset and facebook-combined dataset is not significant. The reason is that the size of dblp dataset or facebook dataset is small, and the memory is sufficient during the entire Spark job process. When memory replacement is rarely triggered, there is almost no performance difference between the two replacement strategies.

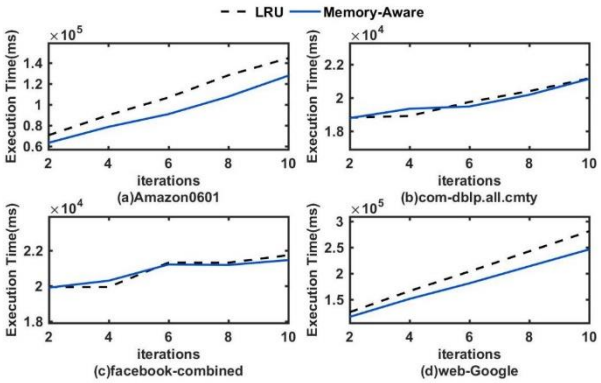

Figure 2. Execution time under different datasets

Also, this paper verifies the performance comparison with different Executor memory under the web-Google dataset. The experimental results shown in Figure 3 indicate that under the same data set, both 2GB Executor memory and 1GB Executor memory have an optimization effect of 5%-13% relative to the LRU. Because the LRU algorithm only considers the time when the data block cached in memory was last accessed, it starts to evict the data block that has not been used for the longest time. It ignores the cost caused by repeated reads and writes of data blocks in memory and the future use of data blocks. Conversely, the MA cache replacement algorithm comprehensively considers the future use of data blocks, the data block size, dependencies and times of use, to get a more accurate assessment of the data block. It is possible to ensure that data blocks that are more valuable for application execution remain in storage memory.

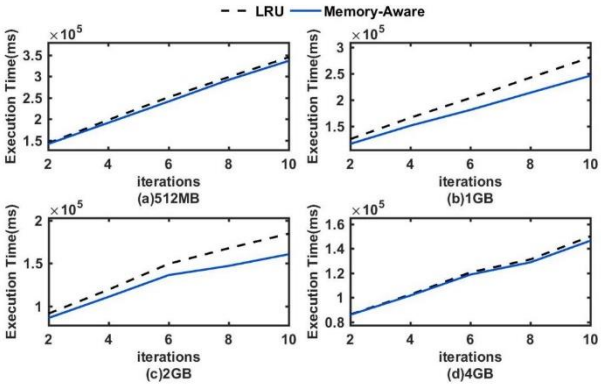

Figure 3. Execution time under different Executor memory

The PageRank experiments on the web-Google dataset performed are somewhat optimized at 512MB and 4GB. According to the analysis of the job logs, in the experimental environment of 512MB, the PageRank algorithm using the Google dataset experienced too frequent cache replacement, and the storage memory can only accommodate 1-2 data blocks. When there is a large data block for cache operation, the more important RDD block will be evicted from memory, resulting in frequent read and write operations. It is difficult to achieve accurate selection of high-value RDD, and the optimization space is small. In the 4GB experimental environment, the memory is sufficient, and there are few cache replacement operations. In this case, different replacement strategies have little impact on the execution time.

3.2 Multi-Job Evaluation

Previous experiments have compared the LRU with the memory-aware Spark cache replacement algorithm on a single-job workload. Here, we conduct experiments on the optimization of the Spark running performance using different multi-job data collections. Each data collection contains more than one single-job data sets (including Amazon0601, etc.).

According to the results shown in Figure 4 and Figure 5, we can find that in multi-job environment, the memory-aware Spark cache replacement strategy has different degrees of optimization effects compared with LRU. With the increase of iterations, MA always outperforms LRU in terms of execution time under different memory size settings. This shows that the MA strategy has more accurate data block replacement rules when the cache competition happens. Our strategy can make more full use of Spark's memory resources and maximize the performance of Spark's distributed computing.

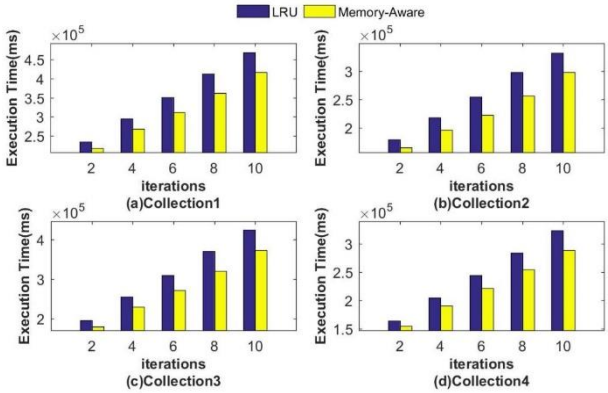

Figure 4. Execution time under different collections

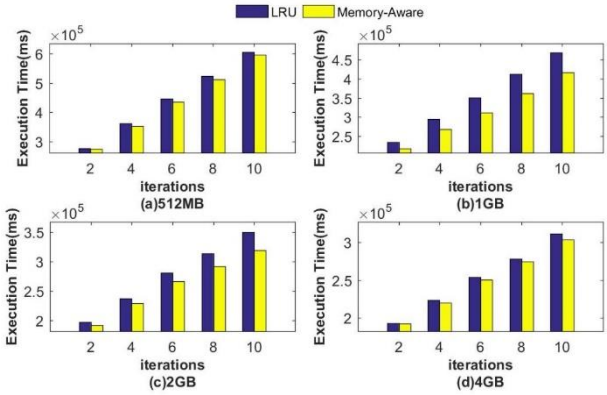

Figure 5. Execution time comparison of Collection1

4 Conclusion

With the development of Spark systems in different application scenarios, the traditional LRU cache replacement algorithm can not always achieve high performance under different environments. Especially, when the cache capacity is insufficient, LRU does not deal with the cache replacement operations wisely and efficiently. To solve the performance issues for Spark, in this paper we design a novel memory-aware cache replacement mechanism. The detailed working framework of our proposed method is introduced, and each component is designed with specific functions. We further present the memory-aware cache replacement algorithm, including the RDD weight models and symbol definitions. Finally, we conduct designed extensive experiments to verify the performance for our proposed method under different settings. Evaluation shows the MA strategy outperforms LRU in terms of Spark execution time. In the future, we plan to integrate our approach with other systems [26-28] to provide the efficiency and security for data storage and processing.

Acknowledgments

This work is supported by the National Natural Science Foundation of China (No. 62172058, 61802031), the Natural Science Foundation of Hunan Province, China (No. 2020JJ5605, 2020JJ2029, 2022SK2107, 2022GK2019, 2021JJ30735), and the Foundation of State Key Laboratory of Public Big Data (No. PBD2021-15), Guizhou University. This work is also funded by the Researchers Supporting Project No. (RSP-2021/102) King Saud University, Riyadh, Saudi Arabia.

References

[1] J. Wang, Y. Yang, T. Wang, R. S. Sherratt, J. Zhang, Big Data Service Architecture: A Survey, *Journal of Internet Technology*, Vol. 21, No. 2, pp. 393-405, March, 2020.

[2] Y. Chen, Y. Lin, Z. Zheng, P. Yu, J. Shen, M. Guo, Preference-aware edge server placement in the internet of things, *IEEE Internet of Things Journal*, Vol. 9, No. 2, pp. 1289-1299, January, 2022.

[3] J. Wang, W. Wu, Z. Liao, Y. Jung, J. Kim, An enhanced PROMOT algorithm with D2D and robust for mobile edge computing, *Journal of Internet Technology*, Vol. 21, No. 5, pp. 1437-1445, September, 2020.

[4] Y. Chen, P. Yu, W. Chen, Z. Zheng, M. Guo, Embedding-based Similarity Computation for Massive Vehicle Trajectory Data, *IEEE Internet of Things Journal*, Vol. 9, No. 6, pp. 4650-4660, March, 2022.

[5] H. Wickham, The Split-Apply-Combine Strategy for Data Analysis, *Journal of Statistical Software*, Vol. 40, No. 1, pp. 1-29, April, 2011.

[6] H. Ke, P. Li, S. Guo, M. Guo, On Traffic-Aware Partition and Aggregation in MapReduce for Big Data Applications, *IEEE Transactions on Parallel and Distributed Systems*, Vol. 27, No. 3, pp. 818-828, March, 2016.

[7] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, I. Stoica, Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing, *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, San Jose, CA, America, 2012, pp. 15-28.

[8] J. Zhang, C. Wu, D. Yang, Y. Chen, X. Meng, L. Xu, M. Guo, HSCS: a hybrid shared cache scheduling scheme for multiprogrammed workloads, *Frontiers of Computer Science*, Vol. 12, No. 6, pp. 1090-1104, December, 2018.

[9] O. Eytan, D. Harnik, E. Ofer, R. Friedman, It's Time to Revisit LRU vs. FIFO, *12th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 20)*, Virtual Event, USA, 2020, pp. 1-7.

[10] G. Hasslinger, J. Heikkinen, K. Ntougias, F. Hasslinger, O. Hohlfeld, Optimum caching versus LRU and LFU: Comparison and combined limited look-ahead strategies, *2018 16th International Symposium on Modeling and Optimization in Mobile, Ad Hoc, and Wireless Networks (WiOpt)*, Shanghai, China, 2018, pp. 1-6.

[11] D. Lee, J. Choi, J. H. Kim, S. H. Noh, S. L. Min, Y. Cho, C. S. Kim, LRFU: a spectrum of policies that subsumes the least recently used and least frequently used policies, *IEEE Transactions on Computers*, Vol. 50, No. 12, pp. 1352-1361, December, 2001.

[12] G. Quan, J. Tan, A. Eryilmaz, N. B. Shroff, A New Flexible Multi-flow LRU Cache Management Paradigm for Minimizing Misses, *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, Vol. 3, No. 2, pp. 1-30, June, 2019.

[13] J. Zhang, S. Zhong, J. Wang, X. Yu, O. Alfarraj, A Storage Optimization Scheme for Blockchain Transaction Databases, *Computer Systems Science and Engineering*, Vol. 36, No. 3, pp. 521-535, January, 2021.

[14] P. Cao, E. Felten, A. R. Karlin, K. Li, Implementation and performance of integrated application-controlled file caching, prefetching, and disk scheduling, *ACM Transactions on Computer Systems (TOCS)*, Vol. 14, No. 4, pp. 311-343, November, 1996.

[15] C. Bian, J. Yu, C. Ying, W. Xiu, Self-Adaptive Strategy for Cache Management in Spark, *Acta Electronica Sinica*, Vol. 45, No. 2, pp. 278-284, February, 2017.

[16] T. Chen, L. Zhang, K. Li, L. Zhou, Optimization of RDD Cache Replacement Strategy Optimization in Spark Framework, *Journal of Chinese Computer Systems*, Vol. 40, No. 6, pp. 1248-1253, June, 2019.

[17] W. Huang, L. Meng, D. Zhang, W. Zhang, In-memory parallel processing of massive remotely sensed data using an apache spark on hadoop yarn model, *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, Vol. 10, No. 1, pp. 3-

19, January, 2017.
[18] D. Swain, B. Paikaray, D. Swain, AWRP: Adaptive weight ranking policy for improving cache performance, *Journal of Computing*, Vol. 3, No. 2, pp. 209-214, February, 2011.
[19] Y. Zhao, J. Dong, H. Liu, J. Wu, Y. Liu, Improving Cache Management with Redundant RDDs Eviction in Spark, *Computers, Materials & Continua*, Vol. 68, No. 1, pp. 727-741, March, 2021.
[20] H. Inagaki, R. Kawashima, H. Matsuo, Improving Apache Spark's Cache Mechanism with LRC-Based Method Using Bloom Filter, *2018 Sixth International Symposium on Computing and Networking Workshops (CANDARW)*, Takayama, Japan, 2018, pp. 496-500.
[21] J. Chen , K. Li, Z. Tang, K. Bilal, S. Yu, C. Weng, K. Li, A Parallel Random Forest Algorithm for Big Data in a Spark Cloud Computing Environment, *IEEE Transactions on Parallel and Distributed Systems*, Vol. 28, No. 4, pp. 919-933, April, 2017.
[22] Y. Yu, W. Wang, J. Zhang, K. B. Letaief, LRC: Dependency-aware cache management for data analytics clusters, *IEEE INFOCOM 2017-IEEE Conference on Computer Communications*, Atlanta, USA, 2017, pp. 1-9.
[23] H. Li, M. Dong, K. Ota, M. Guo, Pricing and Repurchasing for Big Data Processing in Multi-Clouds, *IEEE Transactions on Emerging Topics in Computing*, Vol. 4, No. 2, pp. 266-277, April-June, 2016.
[24] J. Wang, Y. Yang, J. Zhang, X. Yu, O. Alfarraj, A. Tolba, A Data-Aware Remote Procedure Call Method for Big Data Systems, *Computer Systems Science and Engineering*, Vol. 35, No. 6, pp. 523-532, November, 2020.
[25] M. Duan, K. Li, X. Liao, K. Li, A Parallel Multiclassification Algorithm for Big Data Using an Extreme Learning Machine, *IEEE Transactions on Neural Networks and Learning Systems*, Vol. 29, No. 6, pp. 2337-2351, June, 2018.
[26] J. Zhang, S. Zhong, T. Wang, H. Chao, J. Wang, Blockchain-based Systems and Applications: A Survey, *Journal of Internet Technology*, Vol. 21, No. 1, pp. 1-14, January, 2020.
[27] Z. Xu, W. Liang, K. C. Li, J. Xu, H. Jin, A blockchain-based Roadside Unit-assisted authentication and key agreement protocol for Internet of Vehicles. *Journal of Parallel and Distributed Computing*, Vol. 149, pp. 29-39, March, 2021.
[28] Y. Luo, K. Yang, Q. Tang, J. Zhang, P. Li, S. Qiu, An optimal data service providing framework in cloud radio access network, *EURASIP Journal on Wireless Communications and Networking*, Vol. 2016, p 23, January, 2016.

Biographies

Jingyu Zhang received the Ph.D. degree in Computer Science and Technology from Shanghai Jiao Tong University in 2017. He is currently a Distinguish Associate Professor at the School of Computer & Communication Engineering, Changsha University of Science and Technology, China. His research interests include computer architecture, mobile computing and blockchain.

Ruihan Zhang received the B.E. degree in Electronic Science and Engineering from Xiamen University in 2018. He is currently a graduate student at the School of Computer & Communication Engineering, Changsha University of Science & Technology, China. His research interests include big data and storage systems.

Osama Alfarraj received the master's and Ph.D. degrees in information and communication technology from Griffith University, in 2008 and 2013, respectively. He is currently an Associate Professor of computer sciences with King Saud University. His current research interests include eSystems, cloud computing, and big data.

Amr Tolba received the M.Sc. and Ph.D. degrees from Mathematics and Computer Science Department, Menoufia University, Egypt, in 2002 and 2006, respectively. He is currently on leave from Menoufia University to the Computer Science Department, Community College, King Saud University. His current research interests include AI, IoT, and data science.

Gwang-Jun Kim received the B.E., M.E., and Ph.D. degrees in computer engineering from Chosun University, in 1993, 1995, and 2000, respectively. He is currently a Professor in computer engineering at Chonnam National University, South Korea. His research interests include the area sensor networks, the IoT and real-time communication.