Intelligent RDD Management for High Performance In-Memory Computing in Spark

Mingyue Zhang[1,3], Renhai Chen[1,3]*, Xiaowang Zhang[1,3], Zhiyong Feng[2,3], Guozheng Rao[1,3], and Xin Wang[1,3]
[1]School of Computer Science and Technology, Tianjin University, Tianjin, China
[2]School of Computer Software, Tianjin University, Tianjin, China
[3]Tianjin Key Laboratory of Cognitive Computing and Application, Tianjin, China
{zhangmingyue, renhai.chen, xiaowangzhang, zyfeng, rgz, and wangx}@tju.edu.cn

ABSTRACT

Spark is a pervasively used in-memory computing framework in the era of big data, and can greatly accelerate the computation speed by wrapping the accessed data as resilient distribution datasets (RDDs) and storing these datasets in the fast accessed main memory. However, the space of main memory is limited, and Spark does not provide an intelligent mechanism to store reasonable RDDs in the limited memory. In this paper, we propose a fine-grained RDD checkpointing and kick-out selection strategy, by which Spark can intelligently select the reasonable RDDs to maximize the memory usage. The experiment is conducted on a server with four nodes. Experimental results demonstrate that the proposed techniques can effectively accelerate the execution speed.

1. INTRODUCTION

Spark is a commonly used in-memory computing engine for big data processing. The key component in Spark is Resilient Distributed Dataset (RDD), which is a distributed memory abstraction that lets programmers perform in-memory computations on large clusters in a fault-tolerant manner[4]. RDD leverages the main memory to cache the intermediate results, with which, Spark has a huge advantage over other large-scale data-intensive frameworks, e.g. Hadoop[1].

Although the in-memory feature of RDD makes Spark faster than many other non-in-memory big data processing platform, in-memory feature also brings the volatile problem. That is, the loss of RDD will cause Spark to recompute all the missing RDDs on the lineage, and recomputing of a long lineage chain will introduce the considerable computation overhead. To address this issue, checkpointing mechanism is integrated in Spark. Checkpointing mechanism can cut off the lineage and save the data in the external storage for the coming computing. Spark currently provides an interface for checkpointing, while leaves the decision of which data to be checkpointed to the user. Users can control storage strategies of RDDs (i.e., main memory or external disk) and based on that, Spark can cache many multiple RDDs, so that users can reuse intermediate results across multiple computations.

*Renhai Chen is the corresponding author.

WWW'17 Companion, April 3–7, 2017, Perth, Australia.
ACM 978-1-4503-4914-7/17/04.
http://dx.doi.org/10.1145/3041021.3054246

However, frequently checkpointing the RDDs (slow disk I/O operations) will significantly influence the performance of Spark. Whether to checkpoint the RDDs completely depends on the experience of programmers. Since applications grow very complex, and it is very difficult for programmers to provide an optimal checkpointing solution. When useless intermediate results are kept in distributed memory according to the storage strategy, it is highly possible to waste memory space and degrade the performance.

Another important issue in Spark is the memory space replacement. When limited memory is used up, Spark needs to select victim RDDs to reclaim the memory space. Spark selects the least recently used (LRU[2]) RDD to be replaced. However, the LRU algorithm just considers whether the partitions are used recently, and does not make sure the chosen one is valueless. For example, if the choice of the victim RDD is far from its ancestor, we should spend considerable computing cost in the future.

In this paper, we present an intelligent RDD management scheme in Spark to help solve the long lineage problem with less influence on the performance and address the limited memory space issue. We first present the fine-grained checkpointing technique to avoid discarding of RDDs with high using frequency. Then, we discuss how to smartly checkpoint some RDDs to eliminate the computation overhead. To address the limited memory space issue, a novel kick-out selection scheme is proposed to first cache RDDs with long lineage graphs containing wide dependencies. Experiments based on the Spark-1.5.2 platform and big data workloads demonstrate that the proposed design achieves up to 28.01% performance improvement over the baseline schemes.

2. INTELLIGENT RDD MANAGEMENT

2.1 Fine-Grained Checkpointing

Spark provides a coarse-grained checkpointing strategy that discards all the ancestral RDDs of the checkpointed RDD. This mechanism can avoid the recomputing for the checkpointed RDD. However, this coarse-grained method may serious degrade the performance of Spark. Since RDDs are usually frequently used in the large-scale data-intensive applications, it is very likely that the discarded ancestral RDDs are used in the near future. At that time, the spark should recollect the information about how the RDD was derived from other RDDs to recompute that RDD, which introduces considerable computation overhead. To address this issue, we propose a fine-grained checkpointing strategy. Different from prior art, fine-grained checkpointing scheme does not directly discard the ancestral RDDs and globally

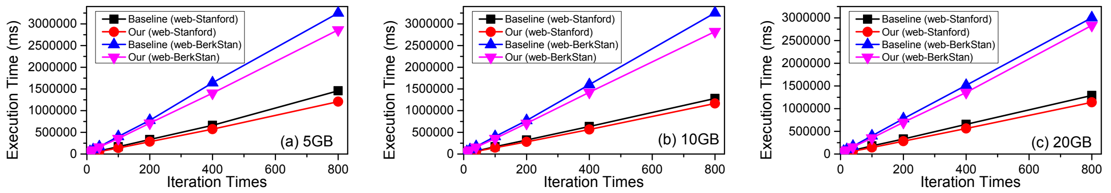

Figure 1: Comparison of the baseline and our proposed schemes with different memory size configurations.

determines the discarded RDDs, which is discussed in Section 2.3.

2.2 Lineage vs. Checkpointing

Although lineage can always be used to recover RDDs after a failure, such recovery may be time-consuming for RDDs with long lineage chains [4]. Thus, we consider to checkpoint some RDDs to eliminate this computation overhead.

In general, checkpointing is useful for RDDs with long lineage graphs containing and frequent recovery. We take these two factors to guide the selection of RDDs for checkpointing, and use the following equations to model this event.

$$L \geq C \times (1 - \omega) \tag{1}$$

In Equation (1), L and C denote the recovery time via recomputing and checkpointing, respectively. ω $(0 \leq \omega \leq 1)$ presents the frequency of failures. If the condition described in Equation (1) is satisfied, we will perform the checkpointing.

2.3 Kick-Out Selection

To reclaim the limited memory space, Spark selects the least recently used (LRU) RDD to be replaced. However, the LRU algorithm just considers the temporal locality, and it does not exploit the unique features of RDD, which results in low memory utilization. In general, RDDs with long lineage graphs containing wide dependencies should have the first priority to use the memory space. This is because the long lineage graph will introduce noticeable computation overhead, and wide dependency implies the high-probability accessing in the future. So, caching this kind of RDDs can minimize the computation overhead introduced by memory replacement, and correspondingly maximize the memory utilization.

$$P = \begin{cases} \frac{L}{L_{max}} + \frac{D}{D_{max}} & \text{if } RDD \text{ is not checkpointed} \\ \frac{C}{C_{max}} + \frac{D}{D_{max}} & \text{if } RDD \text{ is checkpointed} \end{cases} \tag{2}$$

We assign a priority P to each RDD, and the RDD with high P has the high priority to use the memory space. The priority P is calculated according to Equation (2). In this equation, L and C are discussed in Section 2.2, and D represents the degree of dependency. We normalized the recovery time and dependency to the range between 0 and 1 by dividing the maximum number (i.e., L_{max}, C_{max}, and D_{max}).

3. EVALUATION

3.1 Experimental Setup

We deploy Spark to manage four server nodes. Each server node is equipped with 64GB memory, 13TB hard disk drive, and 2.2GHz Intel CPU. The operating system is Ubuntu 14.04. The version for Scala is 2.10.4. We use Hadoop-2.6.0, Scala-2.10.4, and Spark-1.5.2 for all experiments. The memory assigned to Spark is variable, and we set it as 5G, 10G, or 20G under different conditions.

Table 1: Characteristics of datasets

Name	Nodes	Edges	Description
web-Stanford	281,903	2,312,497	Web graph of Stanford.edu
web-BerkStan	685,230	7,600,595	Web graph of Berkeley and Stanford

We choose 15 graph datasets from SNAP [3] to do comprehensive experiments. Because of the limited space, we select two representative datasets as examples to show the effectiveness of the proposed scheme. The characteristics of datasets are shown in Table 1. The numbers of nodes and edges have a great influence on the execution time and memory usage. The whole iterative process finishes until the processing is convergence.

3.2 Experimental Results

Figure 1 illustrates the performance improvement by comparing of the baseline and our proposed schemes with different memory size configurations. The baseline scheme is the original Spark design. The performance is improved by up to 28.01% (13.63% on average) compared with the baseline schemes.

4. CONCLUSIONS

In this paper, we present an intelligent RDD management method to enhance the performance of Spark. We analysed the long computation lineage and low memory space management issues. Based on these issues, we propose three techniques, namely, fine-grained checkpointing, lineage vs. checkpointing, and kick-out selection. Experimental results demonstrate the proposed scheme can effectively enhance the performance of Spark.

5. ACKNOWLEDGMENTS

This work is supported by the National Natural Science Foundation of China (61672377 and 61373165), the Key Technology Research and Development Program of Tianjin (16YFZCGX00210), the National Key Research and Development Program of China (2016YFB1000603), and the Tianjin Science and Technology Commissioners Project (15JCTPJC56400).

6. REFERENCES

[1] Apache. *Hadoop*, 2017. `http://hadoop.apache.org/`.

[2] D. Lee, J. Choi, J. H. Kim, S. H. Noh, S. L. Min, Y. Cho, and C. S. Kim. LRFU: A spectrum of policies that subsumes the least recently used and least frequently used policies. *IEEE Transactions on Computers*, 50(12):1352–1361, 2001.

[3] Stanford. *Stanford Network Analysis Project*, 2017. `http://snap.stanford.edu/`.

[4] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI'12*, pages 1–14, 2012.