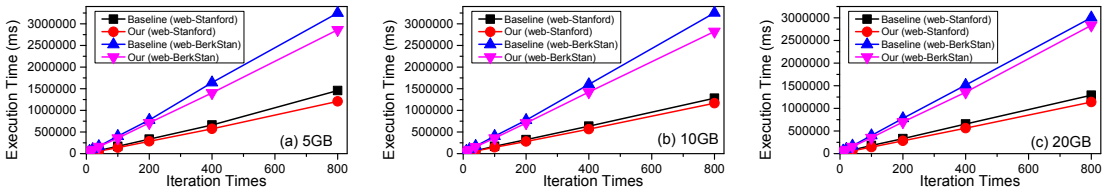

Figure 1: Comparison of the baseline and our proposed schemes with different memory size configurations.

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.