

ASC: Improving Spark Driver Performance with Automatic Spark Checkpoint

Wei Zhu*, Haopeng Chen*, Fei Hu*

*School of Electronic Information and Electrical Engineering
Shanghai Jiao Tong University, 800 Dongchuan Road, Shanghai, China
zw19880717@sjtu.edu.cn, chen-hp@sjtu.edu.cn, hufei@sjtu.edu.cn

Abstract—Many great big data processing platforms, for example Hadoop Map Reduce, are keeping improving large-scale data processing performance which make big data processing focus of IT industry. Among them Spark has become increasingly popular big data processing framework since it was presented in 2010 first time. Spark use RDD for its data abstraction, targeting at the multiple iteration large-scale data processing with reuse of data, the in-memory feature of RDD make Spark faster than many other non-in-memory big data processing platform. However in-memory feature also bring the volatile problem, a failure or a missing RDD will cause Spark to recompute all the missing RDD on the lineage. And a long lineage will also increasing the time cost and memory usage of Driver analysing the lineage. A checkpoint will cut off the lineage and save the data which is required in the coming computing, the frequency to make a checkpoint and the RDDs which are selected to save will significantly influence the performance. In this paper, we are presenting an automatic checkpoint algorithm on Spark to help solve the long lineage problem with less influence on the performance. The automatic checkpoint will select the necessary RDD to save and bring an acceptable overhead and improve the time performance for multiple iteration.

Key words—Spark, automatic checkpoint, lineage, distributed computing, big data.

I. INTRODUCTION

The abstraction of Spark[1] data set is RDD[3], which is implemented as an in-memory data structure for high speed accessing. However, the in-memory feature make in-memory RDD volatile. Lineage[5] is used to keep the RDD transformation information to recompute a RDD which Spark find it missing when it is to be accessed. In multiple iteration Spark application with data reuse, if there is no checkpoint, a long and complex lineage will be cost an unacceptable time to analyze in each iteration.

We present an automatic checkpoint algorithm on Spark, cutting off the long lineage, reducing the DAGScheduler analysis overhead. Major contribution of this paper is:

Transparent checkpoint data selection: No matter what the lineage is, the scheduler will choose the right RDDs to save, do not require application developer to assign it.

Automatically do the checkpoint: The scheduler will automatically make tradeoff between the checkpoint overhead and lineage cutting off.

II. SPARK LINEAGE AND CHECKPOINT

A. Lineage implementation in Spark

Spark uses Dependency and Stage class to storage the dependencies between different RDDs. And shuffle dependencies divide the lineage into many stages while narrow dependencies do not

We can define that narrow dependencies can get the parent RDD directly while shuffle dependencies couldn't, because there will be more than one parent RDD.

In the scheduler implementation store the shuffle information in shuffle id in the memory, but compute the entire stage information when a job was submitted, and clean them after job finishing. For multiple iteration application, the lineage will be too long, stage object will increase linearly. Since Spark use scala implementation, the stage objects will stay in the old generation heaps in JVM unless there is a JVM full GC. After a certain numbers of iteration, the old generation heap will not have enough space and take a JVM full GC. Since the lineage keeps increasing, JVM will take a full GC more frequently, this will cost an unacceptable overhead. We take a simple experiment to show this issue: we use a 1KB graph data to run the PageRank algorithm without checkpoint which provided by the GraphX Lib [11]. And in this case data computing nearly takes cost no time, driver's scheduling consume almost the overhead time. Fig.1 shows the time cost per iteration, at first is less than 1s and increased to 11s after 720 iteration. The peaks in Fig.1 are the extra jvm full GC overhead and iteration time cost keeps increasing and after 723 jvm threw stackoverflow exception which ended the application.

B. Checkpoint implementation in Spark

Spark has its own checkpoint implementation, and the checkpoint will replace the parent RDD with a checkpointRDD and the cut off the lineage. When RDD accessing miss or failure occurs Spark will recompute the lineage from begin which now is the checkpointRDD instead of the input data source or the

ancestor RDD. Application developers have to set the checkpoint path and call the `RDD.checkpoint()` method to make a checkpoint, which means they must determine which RDDs should be checkpointed and when they need to be checkpointed. This will require them know the details about the system.

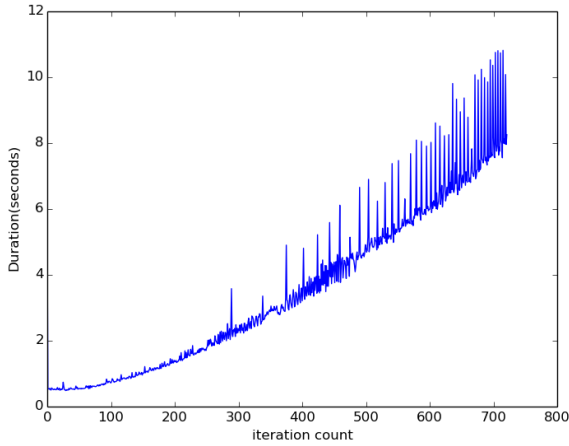


Figure 1. Duration time for each iteration on origin spark with no checkpoint for 723 iterations

III. DESIGN AND IMPLEMENTATION

We present an automatically checkpoint for Spark, which can choose the appropriate RDDs to save and reduce the lineage reanalyze overhead for each job with slight overhead.

A. Selection of the checkpoint data

Spark will produce several RDDs in a single API function, and an iteration will have a lot of mid result RDDs. For naïve approach, we may just save the result of a job. But we found that there will be some other RDDs which are not the final RDD of a job, still required by the next iteration computing. Like the RDDs shows in the Fig.2. In Fig.2, RDDs' dependencies are presented by the arrows (e.g. `VertexRDDn` depends on `VertexRDDn-1` and `updatesn`). We could figure out that `VertexRDD` is the result of each iteration, but `EdgeRDD` is also needed for next iteration computing. Then we came out with the solution that we just trace back the lineage, find and keep all the RDDs which is created in the job with direct parents in the previous job so that we could recompute from these RDDs to get all the RDDs in this job. The tracing back lineage method abstraction is described below:

```

WHILE (QUEUE.NOTEMPTY)
  FOR RDD r IN STACK
    FOR PARENT_RDD p OF RDD r
      IF p IS CREATED BEFORE THIS JOB
        RESULT.ADD r
      ELSE
        QUEUE.PUSH p
RETURN RESULT

```

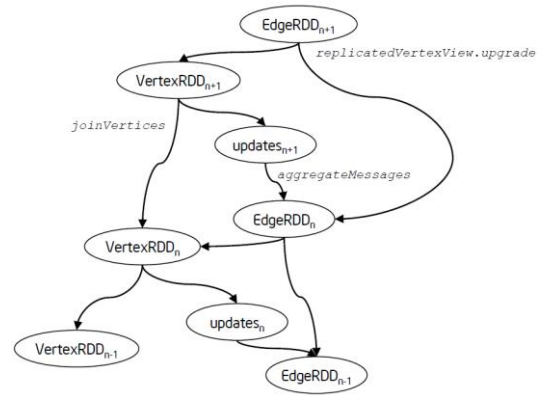


Figure 2. RDD Lineage in the Graphx PageRank

B. Timing of Checkpoint

In section II A, we illustrated with Fig 1 that JVM full GC overhead per iteration grows rapidly with iterations count increasing and no checkpoint. Therefore, we take the utilization rate of JVM old generation heap space as one threshold for timing of checkpoint. We noticed that before first full GC the memory usage rate increased slowly, and if we do the checkpoint the lineage will be cut off, and the Stage objects produced within each iteration will reduce to the same as first iteration. So, we set the threshold of utilization rate of old head space to a value K.

The abstraction of the checkpoint algorithm is below:

```

IF (USAGE_RATE_OLD < K)
  SET CHECKPOINTED = FALSE
ELSE IF USAGE_RATE_OLD > K AND CHECKPOINTED
  = FALSE
  CPRDD = FIND_CHECKPOINT_RDD
  FOR RDD r IN CPRDD
    r.checkpoint()
  CHECKPOINTED = TRUE

```

IV. EVALUATION

We analyze the performance and behavior of the Spark automatic checkpoint in this sections. We measure the automatic checkpoint with following aspects:

The application total time overhead and time cost in single iteration.

The scalability of checkpoint with different size of input

ALL experiments are performed on Spark 1.4.0, 5 physical machines cluster with 1 master and 4 slaves and with K = 0.8.

A. Time performance in single iteration

As it mentioned in section II, the long lineage will significantly increase the time cost after several iterations. We use the Spark Graphx library PageRank algorithm as bench mark to show the performance. We design the experiment with a 1000 iterations PageRank application with 100MB input file on 4GB memory driver. Fig.3 shows the time cost per iteration with ASC, the peaks in Fig.3 are the checkpoint overhead. We

can find that time overhead per iteration is increasing and then drop to around 0.3 second which equals to the first iteration's cost.

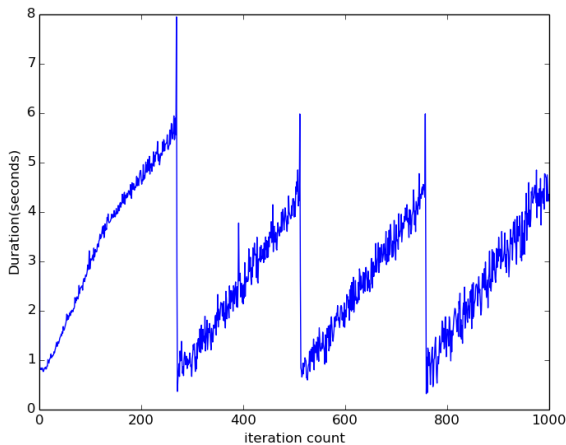


Figure 3. Duration time for each iteration with input file of 1MB on ASC

B. Scalability of Checkpoint

We use input file with different size to show the scalability of the implementation. We set the input file size to 1MB 100MB, 500MB and Fig 4, 5, 6 show the total time cost of application with ASC compared to origin Spark without checkpoint for the three different scale of input file size. We can see that in the first 400 iterations ASC will cost a little extra overhead, but after about 400 iterations, ASC has less total time cost. The increase rate of ASC also reduce after each checkpoint.

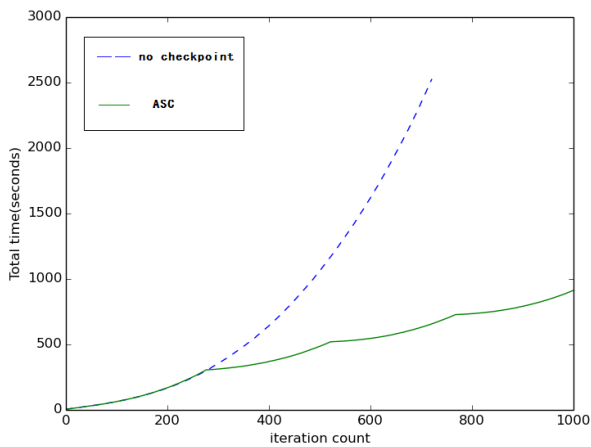


Figure 4. Total time for 1000 iteration with input file 1MB both on ASC and no checkpoint condition

In Table 1 and Table 2 we show the total time overhead of jvm full GC overhead in previous experiments and the total iterations without checkpoint in the experiments are at most 772 due to the stack overflow error of jvm . We can see that jvm full GC take a significant percentage in the total time without checkpoint and ASC reduces both the jvm full GC overhead and total GC overhead more than 90% and improves the

performance greatly. Minor GC and the long lineage analyzing take the other percentages of the extra overhead.

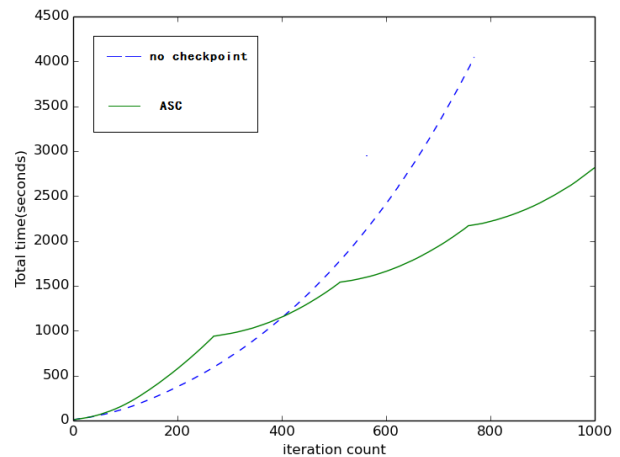


Figure 5. Total time for 1000 iteration with input file 100MB both on ASC and no checkpoint condition

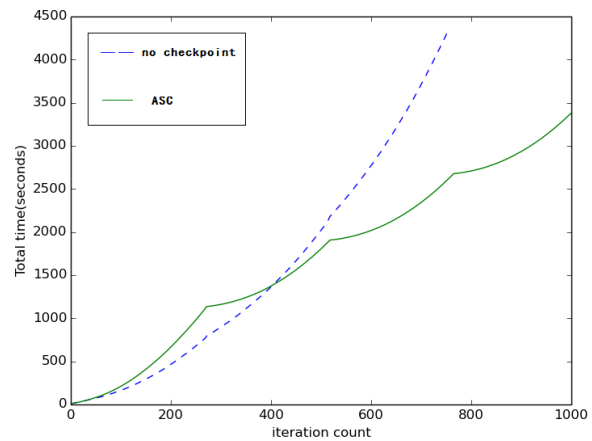


Figure 6. Total time for 1000 iteration with input file 500MB both on ASC and no checkpoint condition

TABLE I. TIME COST IN EACH EXPERIMENT WITHOUT CHECKPOINT

Input file size	Time cost for 770 iterations without checkpoint		
	Total time	JVM full /total GC	GC time percentage in total time
1MB	2527.08s	442.51s/1616.03s	17.5% / 64%
100MB	4048.29s	600.2s/2025.5s	14.8% / 50%
500MB	4336.72s	613.6s/2643.5s	14.1% / 61%

TABLE II. TIME COST IN EACH EXPERIMENT WITH ASC

Input file size	Time cost for 1000 iterations with ASC		
	Total time	JVM full /total GC	GC time percentage in total time
1MB	911.57s	10.77s/ 174.87s	1.2% / 19.1%
100MB	2812.59s	11.7s/ 184.9s	0.4% / 6.5%
500MB	3375.86	11.34s/ 181.08s	0.3% / 5.4%

V. RELATED WORK

Spark is designed as a fast and generic using distributed computing system with the advantage of easy to use and adaptive to various data source (e.g. HDFS, Cassandra, HBase [9]). The in-memory implementation of RDD makes Spark run faster than Hadoop Map Reduce [2] but it need the long lineage to store the step to get the RDD. Checkpoint in Spark help both on cutting off the lineage and fault tolerance.

Fault tolerance is the main design purpose in other circumstances and there already many researches on this aspect. Early researches like [4], [10] shows us optimum solution of the interval for the checkpoint. In [6] presents an incremental checkpoint with transparent feature for parallel computers, which uses multiple step to overcome the dirty pages issues.

There are also research on checkpoint for other specific platform or condition as in [7], [8].

VI. CONCLUSIONS

Spark shows great performance in big data analysis, in-memory data abstraction which helps speeding up the data fetch in the computation but also required to store a lineage to help rebuild when data miss or failures.

We observed and analysed the long lineage issues which occurred the multiple iteration computation in Spark then designed the ASC which allows Spark automatically to do the checkpoint, helps cutting off the lineage and reduce the jvm GC time overhead with little extra overhead.

We implemented ASC on Spark 1.4.0 and evaluated ASC to show the overhead and performance of it. The time of a single iteration reduced periodically with ASC instead of keeping increasing with no checkpoint. And total execution time performance also reduce by more 50% with ASC compared to no checkpoint.

VII.Reference

- [1] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, Ion Stoica(2010). Spark: Cluster Computing with Working Sets. HotCloud 2010. June 2010.
- [2] Dean, J., & Ghemawat, S. (2008). MapReduce: simplified data processing on large clusters. Communications of the ACM, 51(1), 107-113.

- [3] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, Ion Stoica(2012).Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. NSDI 2012. April 2012.
- [4] J. W. Young. A first order approximation to the optimum checkpoint interval. Commun. ACM, 17:530–531, Sept 1974
- [5] R. Bose and J. Frew. Lineage retrieval for scientific data processing: a survey. ACM Computing Surveys, 37:1–28, 2005.
- [6] Roberto Gioiosa, Jose Carlos Sancho, Song Jiang, Fabrizio Petrini, Kei Davis (2005). Transparent, Incremental Checkpointing at Kernel Level: a Foundation for Fault Tolerance for Parallel Computers. SC '05 Proceedings of the 2005 ACM/IEEE conference on Supercomputing.
- [7] Yuval Tamir , Carlo H. Séquin(1984), Error Recovery in Multicomputers Using Global Checkpoints. 1984 International Conference on Parallel Processing
- [8] Bronevetsky, Greg, et al. "Application-level checkpointing for shared memory programs." ACM SIGOPS Operating Systems Review 38.5 (2004): 235-247.
- [9] Vora, Mehul Nalin. "Hadoop-HBase for large-scale data." Computer Science and Network Technology (ICCSNT), 2011 International Conference on. Vol. 1. IEEE, 2011.
- [10] Daly, J. (2003). A model for predicting the optimum checkpoint interval for restart dumps. In Computational Science—ICCS 2003 (pp. 3-12). Springer Berlin Heidelberg.
- [11] Xin, Reynold S., et al. "Graphx: A resilient distributed graph system on spark." First International Workshop on Graph Data Management Experiences and Systems. ACM, 2013.



Wei Zhu. He received his Bachelor degree of Computer science and technology, Chongqing University in 2011 Chongqing China. And now his is working for his Master degree of software engineering in Shanghai Jiao Tong University, Shanghai, China. He is interested in fields of distributed system and big data processing.



Haopeng Chen. He received his Ph.D degree from Department of Computer Science and Engineering, Northwestern Polytechnical University, Xi'an, Shanxi Province, China in 2001. He has worked in School of Software, Shanghai Jiao Tong University since 2004 after he finished his two-year postdoctoral research job in Department of Computer Science and

Engineering, Shanghai Jiao Tong University, Shanghai, China. He got the position of Associate Professor in 2008. In 2010, he studied and researched in Georgia Institute of Technology as a visiting scholar. His research group focuses on Distributed Computing and Software Engineering. They have kept researching on Web Services, Web 2.0, Java EE, .NET, and SOA for several years. Recently, they are also interested In cloud computing and researching on the relevant areas, such as cloud federation, resource management, dynamic scaling up and down, and so on.



Fei Hu. He recieved his Bachelor degree from Department of computer software, Northwest University, Xi'an, Shanxi Province, China in 1990 and received his Master degree of computer science and engineering and Ph.D of Precision Guidance and Control both from Northwest Polytechnical University, Xi'an, Shanxi Province, China in 1993 and 1998. He

has worked in Department of Computer Science and Engineering , Northwestern Polytechnical University lecturer , from 1993 to 2006. From 2006/ 9 to now he has worked in School of Software, Shanghai Jiao Tong University. Prof Hu's Publications are as follows: Zhiyang Zhang, Fei Hu and Jian Li, "Autonomous Flight Control System Designed for Small-Scale Helicopter Based on Approximate Dynamic Inversion," The 3rd IEEE International Conference on Advanced Computer Control (ICACC 2011), 18th to 20th January 2011, Harbin, China.