Optimizing Checkpointing Performance in Spark
Ya-Meng Zhang\textsuperscript{1,a}, Yu Luo\textsuperscript{2,b} and Yan-Chen Li\textsuperscript{3,c}

\textsuperscript{1}College of Computer National University of Defense Technology, Changsha, China.
\textsuperscript{2}College of Computer National University of Defense Technology, Changsha, China.
\textsuperscript{3}College of Computer National University of Defense Technology, Changsha, China.
\textsuperscript{a}zhangyameng1993@163.com, \textsuperscript{b}yuluo@nudt.edu.cn, \textsuperscript{c}liyanchen15@nudt.edu.cn

Keywords:Apache spark, Fault tolerant, Checkpointing.

Abstract. Spark \cite{1} is a cluster framework that performs in-memory computing. As with other distributed data processing platforms, fault tolerant plays an important role in the whole architecture. The Fault Tolerant of Spark contains Lineage and Checkpointing. The latter is expensive because doing checkpoint always causes RDD recomputation. In this paper, we analyze the workflow in the execution of the current design and propose alternatives to improve the performance of checkpointing, one of which is based on an existing approach. We evaluate our results in terms of application level throughput.

Introduction
Apache Spark is an open-source analytics cluster computing framework developed in AMP Lab at UC Berkeley. Apache Spark is a general purpose cluster computing system with the goal of outperforming disk-based engines like Hadoop \cite{2}. Spark is an implementation of Resilient Distributed Datasets (RDD); it provides high level APIs in Java, Scala and Python. Mostly Scala is used in Spark programming. Spark enables applications in Hadoop clusters to run up to 100x faster in memory and 10x faster running on disk. It comes with a built-in set of over 80 high-level operators. Apache Spark has been used by many companies including Amazon, Facebook, Yahoo and Group On.

\textbf{Figure 1.} Spark runtime. A user’s driver program launches multiple workers, which read data blocks from a distributed file system and can cache computed RDD partitions in memory.

Fig. 1 shows the architecture of Spark. Spark applications run as independent sets of processes on a cluster, coordinated by the Spark Context object in your main program (called driver program).
which defines one or more RDDS and invokes actions on them. Specifically, to run on a cluster, the Spark Context can connect to several types of cluster managers (either Spark's own standalone cluster manager, Mesos or YARN), which allocate resources across applications. Once connected, Spark acquires executors on nodes in the cluster, which are processes that run computations and store data for your application. Next, it sends your application code (defined by JAR or Python files passed to Spark Context) to the executors. Finally, Spark Context sends tasks to the executors to run.

**Fault Tolerance**

The fault tolerance of Spark is supported efficiently. In general, there are two options to make a distributed dataset fault-tolerant: checkpointing the data or logging the updates made to it. In the target environment (large-scale data analytics), checkpointing the data is expensive: it would require replicating big datasets across machines over the datacenter network, which typically has much lower bandwidth than the memory bandwidth within a machine [3], and it would also consume additional storage. Consequently, logging updates was chosen called Lineage. However, the computation to RDDS is also expensive if lineages are long or dependencies between RDDS are wide. Consequently, checkpointing is also available to make up the lack of lineage. Users have option for them.

RDD Lineage is a graph of the entire parent RDDS of an RDD. It is built as a result of applying transformations to the RDD and creates a logical execution plan. All RDD lineages form lineage graph. Spark can recomputed loss partitions in the event of node failures based on RDD graph. Fig. 2 shows an example of lineage graph. Checkpointing is a process of truncating lineage graph and saving it to a reliable distributed (HDFS) or local file system. There are two types of checkpointing: Reliable in Spark-core and local in Spark Streaming or Graph X. Here we do not investigate the latter. Thus, Checkpointing an RDD means saving the actual intermediate RDD data to a reliable distributed file system, e.g. HDFS. Checkpointing allows Spark to get loss partitions from disk instead of recomputations and it is a supplement to lineage. As mentioned above, we can choose to persist an RDD to disk when the computation is expensive (long lineages or wide dependencies). If some partitions of an RDD are lost, recompilation will begin at the check pointed RDD, which significantly reduces the overhead of Spark.

![Figure 2. An example of lineage. Each box is an RDD, with partitions shown as shaded rectangles.](image)

However, rather than persisting them to disk during the first job, there is a second submit of job for each check pointed RDD. That means doing check pointing will cause RDD reacquisition (recomputation or getting from local storage), which increases system overhead. Thus, in optimizing the performance of check pointing, it is crucial to improve the performance of RDD reacquisition.

**Existing Solutions**

A solution has been proposed to reduce the overhead of RDD reacquisition. Because recomputation is expensive, it is strongly recommended that a check pointed RDD is persisted in memory. However, the effectiveness of this method is highly dependent on applications [4]. It is very common that customers strange to Spark forget to cache check pointed RDDs.

The rest of the paper is organized as follows. Section 2 describes prior work relevant to our project. Section 3 describes our approach in improving the performance of check pointing. Finally, in section 4, we evaluate the changes we propose.

**Related Work**

We have found an attempt at a solution to this problem in Stack Overflow, whose explicit goal is to improve the stability of the solution mentioned in Subsection 1.2 by adding useful warnings about caching check pointing RDDs [5]. However, stability problem still exists.

**Our Approach**

In this section, we discuss the approach to improving the performance of checkpointing. In our analysis in Section 1.1, we demonstrated to achieve high performance, it is important to reduce the overhead of RDD reacquisition. Thus, we pursue the following two alternatives: avoiding reacquisition or making it faster.

![Lineage graph for our example](image)

**Avert Reacquisition**

We attempted to eliminate the reacquisition of checkpointed RDDs which means we should persist checkpointed RDDs to disk in the first computation. Before exploring the approach for that, it is necessary to analyze the process of the RDD computation. As shown in Fig. 3, we illustrate that through an example:
val rdd = sc.parallelize(1 to 5, 1)
val map_rdd = rdd.map(_ + 1)
val filter_rdd = map_rdd.filter(_ > 3)
val res = filter_rdd.reduce(_ + _)

Line 1 defines an RDD containing 1-5, while line 2 derives a MappedRDD from it. Line 3 applies a filter transformation to the MappedRDD and exports a FilterRDD. Line 4 adds all elements of FilterRDD and it leads to a submit of job. Through analyzing the source code of Spark-core, we learned that the elements of these RDDs is generated in order of (RDD1,1), (RDD2,2), (RDD1,2), (RDD2,3), (RDD1,3), (RDD2,4), (RDD3,4), (RDD1,4), (RDD2,5), (RDD3,5), (RDD1,5), (RDD2,6), (RDD3,6). It is obvious that in most cases, the elements belonging to an RDD cannot be generated successively. That means the files which check pointed RDDs are persisted to are opened and closed many times in the jobs of check pointing. This places significant burden on the disk I/O. Additionally, RDDs are first computed in the first job. Thus, too many I/Os also slow down that. It is possible that the approach could be used to mitigate the overhead of check pointing, but we did not investigate this.

**Cache Check pointed RDDs Automatically**

As shown in Subsection 1.2, we introduced that caching check pointed RDDs in memory could improve the performance of RDD reacquisition, because reading memory is faster than recomputation. In this subsection, we propose a method: rather than caching check pointed RDDs by application, our solution is to let RDDs be cached automatically by revising the source code of Spark-core. It is more stable than the method mentioned in Subsection 1.2 and Section 2, and it makes up for the deficiency of the latter.

The way to achieve the goal of caching check pointed RDDs automatically is not complicated. The key lies in the modify of the RDD. Storage Level field in function RDD. checkpoint. So we do not explain this in detail.

**Evaluation**

For the main text, please use 9.5-point type and Times New Roman. Italic type may be used to emphasize words in running text. Bold type and underlining should be avoided.

The first paragraph after a section or subsection should not be indented; subsequent paragraphs should be indented by 6 mm. The use of sections to divide the text of the paper is optional and left as a decision for the author.

**Experimental Environment**

We ran our tests on a 3-virtual-node Spark cluster. Each node was details shown in Table 1. Maximum observed disk throughput was around 50 MB/s and maximum inter-node network throughput was 80 MB/s. We used CentOS-7.0-Minimal as the operating system. We deployed Hadoop-2.2.0 and Spark-1.1.0 on the cluster. One node was configured as Name Node and Master, respectively. The rest two nodes were configured as slave nodes for HDFS storage and Spark task execution.

<table>
<thead>
<tr>
<th>Virtual cores</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAM</td>
<td>1G</td>
</tr>
<tr>
<td>Disks</td>
<td>10G</td>
</tr>
<tr>
<td>Network</td>
<td>1Gbps</td>
</tr>
</tbody>
</table>

Table 1. Experiment Configurations.
### Table 2. Experiment Workloads.

<table>
<thead>
<tr>
<th></th>
<th>Workload 1</th>
<th>Workload 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input data size</td>
<td>100M</td>
<td>1G</td>
</tr>
<tr>
<td>#Partitions</td>
<td>50</td>
<td>100</td>
</tr>
</tbody>
</table>

**Experiment**

We compared the performance of checkpointing implementation using Auto-Caching with that without Auto-Caching for the PageRank application. We examined two workloads (Table 2). Both workloads were based on real data and use-cases from a Spark-user. We ran 10 iterations of the PageRank algorithm and marked the Rank RDD checkpointed in the 9th iteration. The results from this experiment are shown in Fig. 4, for Workload 1 and Workload 2 respectively.

As we can see in Fig. 4, Auto-Caching led to around a 3x improvement on Workload 1 and around a 7x improvement on Workload 2.

![Figure 4. Performance of checkpointing, with and without Auto-Caching.](image)

**Conclusion**

By identifying the lack of Spark checkpointing, we have discussed two alternatives to mitigate the job overheads of checkpointing. In the analysis in Subsection 3.1, we demonstrated that averting RDD reacquisition makes no sense for performance improvement. The more fruitful one is Auto-Caching, which can cache checkpointed RDDs automatically. This simple solution led to a significant improvement in checkpointing job completion time.

**References**


