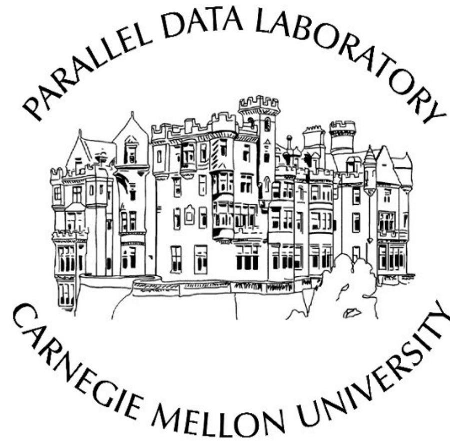

Addressing the Long-Lineage Bottleneck in Apache Spark

Haoran Wang[⋆], Jinliang Wei[⋆], Garth Gibson[⋆,†]
[⋆]*Carnegie Mellon University,* [†]*Vector Institute*

CMU-PDL-18-101
January 2018

Parallel Data Laboratory
Carnegie Mellon University
Pittsburgh, PA 15213-3890

Acknowledgements: We thank the members and companies of the PDL Consortium (including Broadcom, EMC, Facebook, Google, Hewlett-Packard, Hitachi, IBM Research, Intel, Microsoft Research, MongoDB, NetApp, Oracle, Salesforce, Samsung, Seagate, Two Sigma, Toshiba, Veritas, Western Digital) for their interest, insights, feedback, and support.

Keywords: Apache Spark, Lineage

Abstract

Apache Spark employs lazy evaluation [11, 6]; that is, in Spark, a dataset is represented as Resilient Distributed Dataset (RDD), and a single-threaded application (driver) program simply describes transformations (RDD to RDD), referred to as lineage [7, 12], without performing distributed computation until output is requested. The lineage traces computation and dependency back to external (and assumed durable) data sources, allowing Spark to opportunistically cache intermediate RDDs, because it can recompute everything from external data sources. To initiate computation on worker machines, the driver process constructs a directed acyclic graph (DAG) representing computation and dependency according to the requested RDD's lineage. Then the driver broadcasts this DAG to all involved workers requesting they execute their portion of the result RDD. When a requested RDD has a long lineage, as one would expect from iterative convergent or streaming applications [9, 15], constructing and broadcasting computational dependencies can become a significant bottleneck. For example, when solving matrix factorization using Gemulla's iterative convergent algorithm [3], and taking tens of data passes to converge, each data pass is slowed down by 30-40% relative to the prior pass, so the eighth data pass is 8.5× slower than the first.

The current practice to avoid such performance penalty is to frequently checkpoint to durable storage device which truncates lineage size. Checkpointing as a performance speedup is difficult for a programmer to anticipate and fundamentally contradicts Spark's philosophy that the working set should stay in memory and not be replicated across the network. Since Spark caches intermediate RDDs, one solution is to cache constructed DAGs and broadcast only new DAG elements. Our experiments show that with this optimization, per iteration execution time is almost independent of growing lineage size and comparable to the execution time provided by optimal checkpointing. On 10 machines using 240 cores in total, without checkpointing we observed a 3.8× speedup when solving matrix factorization and 16× speedup for a streaming application provided in the Spark distribution.

1 Introduction and Background

Lazy evaluation has been widely employed by many distributed computating systems and deep learning frameworks, such as Spark [14], TensorFlow [1] and MXNet [2]. This paper is concerned with lazy evaluation in Spark. A Spark application (driver) program may create an RDD by reading a text file or transforming other RDDs (such as using *map* or *join*); it may also perform an *action* on an RDD to generate some output, such as returning a value to the application program or saving the dataset to some external storage. The intermediate work done by many application program lines is to simply record the operation and dependency among RDDs, which is referred to as the RDD's lineage. When and only when an action is invoked, the requested RDDs are computed by recursively computing their dependencies, and thus eliminating unused computation becomes trivial. By delaying computation as late as possible, Spark accumulates a large amount of computation, potentially enabling more opportunities for parallelism. Spark's scheduler creates a *job* when an action is invoked. In order to schedule a job's compuation on to distributed workers, Spark's scheduler constructs a dependency DAG according to RDDs' lineage graph, where the nodes are *stages*, containing pipelined RDD transformations. A stage contains as many pipelined RDD transformations with *narrow dependencies* as possible, where each partition of the parent RDD is needed by at most one child RDD partition. Stage boundaries are *wide dependencies* (i.e. *shuffle dependencies*), where each parent partition is needed by more than one child partition. Wide dependencies can be caused by transformations such as *join* and *groupBy*. After a DAG is constructed, Spark's scheduler broadcasts the DAG and *task* to workers who compute their portion of the result RDD. Moreover, by keeping track of RDDs' lineage, Spark may recover a lost RDD (either due to cache eviction or failure) by recomputing it, eliminating the need to duplicate intermediate RDDs on storage.

If lineage is long and contains a large number of RDDs and stages, the DAG may be huge. Such a problem can easily happen in iterative machine learning training when repeatedly updated model parameters are stored as an RDD. Since Spark's dependency DAG can't represent branches and loops, a loop is unrolled in the DAG and the stages of an iteration are repeated tens or even hundreds of times. It may also easily happen in streaming applications, which may run for a long period of time and update an RDD whenever new data arrives.

To demonstrate this problem, we implemented in Spark 2.1 a non-checkpointing matrix factorization application, which factorizes a large data matrix into two lower-rank factor matrices. Our implementation uses Gemulla's parallel stochastic gradient descent algorithm [3], which partitions the data matrix by both row and column, resulting in totally P^2 partitions to efficiently utilize P parallel workers. Each data pass (i.e. iteration) is divided into P subepochs. Within each subepoch, workers process data partitions with no overlapping rows and columns and update rows and columns of the two factor matrices corresponding to the given data partition. Our implementation stores all three matrices as RDDs and joins the factor matrices to the data matrix with different keys each subepoch. Therefore, the lineage of the factor matrices grows by at least one stage each subepoch. Our experiment used the MovieLens dataset [5] and was performed on a PRObE cluster [4]. All our matrix factorization experiments used 10 worker machines, each with 64 cores and 120GB of memory. We launched 240 executors in total and partitioned the data matrix into 57,600 partitions to fully utilize all executors in parallel. As can be seen in Fig 1, the dependency DAG grows from 244 to 1924 stages in 8 iterations and the execution time per iteration grows by nearly $8\times$.

2 Solution and Evaluation

Based on tracing/profiling methods and tools [10, 13, 8], we found that a huge dependency DAG, such as occurs in our matrix factorization example, is a performance bottleneck because of both its construction and communication.

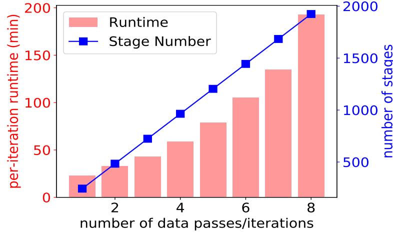

Figure 1: Overhead of growing lineage - the runtime and number of stages in the dependency DAG for each data pass

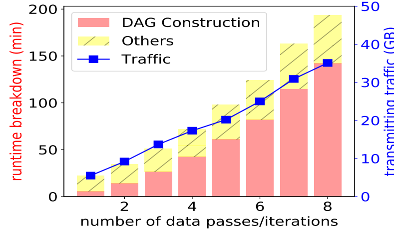

Figure 2: NonCheckpointing Matrix factoriztion - the runtime breakdown and network traffic for each data pass

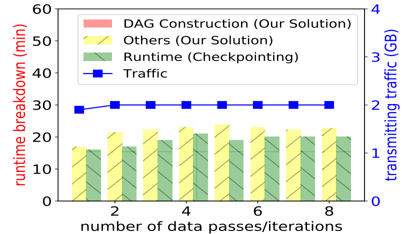

Figure 3: Matrix factoriztion with proposed optimizations and checkpointing - runtime breakdown and network traffic

When a job is submitted, Spark's scheduler creates a *result stage* that contains the result RDD, which is the RDD that an action is performed on to generate an output to the application. In order to create the result stage, Spark's scheduler recursively creates its ancestors stages according to the lineage graph. Then Spark schedules execution of stages whose results have not been cached. When the lineage is long, constructing a task description of all dependent stages along the lineage can become a significant amount of work in the single-threaded scheduler. Fig 2 shows the time spent on constructing all dependent stages for each data pass. Even in the first iteration, dependency DAG construction takes 25% of the execution time and it takes 73% of the time after only 8 iterations. After a job completes, its dependency DAG is deleted, and may be rebuilt in part or whole by later actions.

When scheduling a stage for execution, Spark's scheduler assigns a set of *tasks* to workers each computing a missing partition of this stage. Before dispatching tasks to workers, Spark's scheduler broadcasts the metadata of the RDD to be computed to all workers, which includes the full lineage of that RDD. Moreover, when dispatching a task to a worker, Spark's scheduler sends to the worker the task's dependencies (the parent RDD partitions that this task depends on) and its ancestors' dependencies. This can result in a significant communication from the single-threaded scheduler. As shown in Fig 2, the driver sends out 5.5GB traffic in the first iteration and it reaches 35GB in the eighth iteration.

This long lineage bottleneck is widely known by sophisticated Spark application programmers. A common practice for dealing with long lineage is to have the application program strategically checkpoint RDDs at code locations that truncate much of the lineage for checkpointed data and resume computation immediately from the checkpoint. Thus the future RDDs' lineage only traces back to the checkpoint. Even though this solution is effective if properly implemented as we observe in our experiments, it imposes additional burden on programmers. Since this is checkpointing as a performance optimization, not a fault-tolerant mechanism. More importantly, it fundamentally contradicts Spark's philosophy that the working set should remain in memory and not be replicated in (slow) storage.

In iterative machine learning training of large models, long lineage happens when model parameters are stored as RDDs and iteratively updated. In fact, each iteration may simply append to the previous iteration's dependency DAG. Therefore, one solution to reduce dependency DAG construction time is to cache dependency DAGs in Spark's scheduler (i.e. not delete stages after a job completes) so only the new stages need to be constructed. We refer to this optimization as *dependency DAG caching*.

Similarly, since Spark caches RDDs, executors only need to know the dependencies back to RDD partition they cached, which can effectively reduce Spark scheduler's communication volume if for example a worker node evicts data from its cache. We refer to this optimization as *delta dependency broadcast*. Because the scheduler might not have up-to-date information on which RDDs are cached, an executor may query the scheduler for full dependency if it fails to find any RDD or dependency DAG element if need.

We implemented both optimizations in Spark 2.1 and evaluated them using the above described matrix

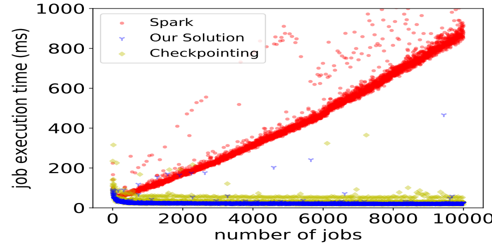

Figure 4: Stateful Network Word Count - the execution time of first 10k jobs on different solutions: Spark, Caching DAG, Checkpointing

factorization application. The runtime of aversion of this application with checkpointing at end of each data pass is also measured. The results can be seen in Fig 3. Compared to Fig 2, we can see that the driver's communication volume is reduced to 2GB per iteration and remains constant; the per-iteration execution time also becomes nearly constant and we observe an 8.5× speedup for executing the eighth iteration and 3.8× speedup for the total execution, which is comparable to checkpointing result, i.e. 9.7× speedup for the eighth iteration and 4.4× speedup for total execution.

As discussed before, streaming applications can also suffer from growing lineage size. We demonstrate this problem and evaluate the effectiveness of our solution using the Stateful Network Word Count application provided by Apache Spark. The application receives a stream of words and every 100ms it counts the words in the current mini-batch and accumulates the result in a word count RDD. Our experiments used 2 machines, each with 64 cores and 120GB of memory, and the number of partitions was set to 2. We used an empty input dataset so we are solely measuring the intrinsic system overhead and executed 10,000 jobs. Our results can be found in Fig 4 which plots the execution time for each job. The original application (plotted as checkpointing) checkpoints the word count RDD every 100ms to avoid the lineage from growing infinitely. After removing the checkpoints (plotted as Spark), the per-job execution time grows significantly overtime due to growing lineage size. For the same application (without checkpointing) running on Spark with our optimizations, the per-job execution time remains nearly constant regardless of lineage size. Overall our solution improved the total runtime of 10,000 jobs from 4305 seconds (without checkpointing) to 261 seconds, while the solution using checkpoints took 311 seconds.

3 Code Release

Our experiments and code revisions are based on Spark version v2.1.0-cr5[1]. Our implementation is released at https://github.com/HrWangChengdu/spark under the master branch. All changes were squashed into the latest commit.

4 Conclusion

In this technical report, we showed that constructing and broadcasting computational dependencies can become a significant bottleneck in Spark. For example, when solving matrix factorization using Gemulla's iterative convergent algorithm [3], and taking tens of data passes to converge, each data pass is slowed down by 30-40% relative to the prior pass. We proposed two optimizations in Spark to address the long lineage overhead. Our evaluations show that the execution time is independent of lineage size for the matrix factorization and a streaming application.

[1] url of the release: https://github.com/apache/spark/releases/tag/v2.1.0

Our performance evaluation is still preliminary, using only two applications and on small or empty datasets, which doesn‘t stress checkpointing. We have not tested our optimizations under less common cases such as RDD cache eviction, machine failures, etc. More extensive evaluation is left to the future work. There also exists alternative solutions to this problem, for example, instead of building the full dependency graph with all stages in the driver, we only need to build the graph with stages that have not been computed and cached, we plan to implement and compare alternative solutions as well.

References

[1] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, et al. Tensorflow: A system for large-scale machine learning. 2016.

[2] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. 2015.

[3] R. Gemulla, E. Nijkamp, P. J. Haas, and Y. Sismanis. Large-scale matrix factorization with distributed stochastic gradient descent. In *Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '11, pages 69–77, New York, NY, USA, 2011. ACM.

[4] G. Gibson, G. Grider, A. Jacobson, and W. Lloyd. Probe: A thousand-node experimental cluster for computer systems research. volume 38, June 2013.

[5] F. M. Harper and J. A. Konstan. The movielens datasets: History and context. In *ACM Transactions on Interactive Intelligent Systems (TiiS)*, page Article 19 (December 2015), 2015.

[6] P. Hudak. Conception, evolution, and application of functional programming languages. *ACM Computing Surveys (CSUR)*, 21(3):359–411, 1989.

[7] R. Ikeda and J. Widom. Data lineage: A survey. Technical report, Stanford InfoLab, 2009.

[8] M. Massie, B. Li, B. Nicholes, V. Vuksan, R. Alexander, J. Buchbinder, F. Costa, A. Dean, D. Josephsen, P. Phaal, et al. *Monitoring with Ganglia: Tracking Dynamic Host and Application Metrics at Scale*. " O'Reilly Media, Inc.", 2012.

[9] X. Meng, J. Bradley, B. Yavuz, E. Sparks, S. Venkataraman, D. Liu, J. Freeman, D. Tsai, M. Amde, S. Owen, et al. Mllib: Machine learning in apache spark. *The Journal of Machine Learning Research*, 17(1):1235–1241, 2016.

[10] K. Ousterhout, R. Rasti, S. Ratnasamy, S. Shenker, B.-G. Chun, and V. ICSI. Making sense of performance in data analytics frameworks. In *NSDI*, volume 15, pages 293–307, 2015.

[11] J. C. Reynolds. *Theories of programming languages*. Cambridge University Press, 2009.

[12] Y. L. Simmhan, B. Plale, and D. Gannon. A survey of data provenance techniques. *Computer Science Department, Indiana University, Bloomington IN*, 47405, 2005.

[13] L. YourKit. The yourkit java profiler. *URL http://www. yourkit. com*, 2010.

[14] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 15–28, San Jose, CA, 2012. USENIX.

[15] M. Zaharia, T. Das, H. Li, S. Shenker, and I. Stoica. Discretized streams: An efficient and fault-tolerant model for stream processing on large clusters. *HotCloud*, 12:10–10, 2012.