R3S: RDMA-based RDD Remote Storage for Spark

Xinan Yan, Bernard Wong, and Sharon Choy
David R. Cheriton School of Computer Science
University of Waterloo
{xinan.yan, bernard, s2choy}@uwaterloo.ca

ABSTRACT

In the past few years, Spark has seen rapid adoption by organizations that require high-performance data processing. This growth stems from its significant performance improvement over past frameworks such as Hadoop and MapReduce. Spark achieves its performance improvement by aggressively storing intermediate computation in memory in order to avoid slow disk accesses. However, Spark's performance advantage disappears when Spark nodes do not have enough local memory to store the intermediate computations, requiring results to be written to disk or recomputed.

In this paper, we introduce R3S, an RDMA-based in-memory RDD storage layer for Spark. R3S leverages high-bandwidth networks and low-latency one-sided RDMA operations to allow Spark nodes to efficiently access intermediate output from a remote node. R3S can use this flexibility to treat the memory available on the different machines in the cluster as a single memory pool. This enables more efficient use of memory and can reduce job completion time. R3S can also work together with the cluster manager to add storage-only nodes to the Spark cluster. This can benefit workloads where adding memory has a far larger impact on performance than adding processing units. Our prototype reduces job completion time by 29% compared to Spark using Tachyon as the RDD storage layer on a machine learning benchmark.

CCS Concepts

•**Software and its engineering → Reflective middleware; Distributed memory; Cloud computing;**

Keywords

Spark; RDMA; Adaptive Storage

1. INTRODUCTION

Data-processing on Big Data sources is typically organized into multiple computational stages, where each stage corresponds to a single iteration through a distributed computation framework such as Hadoop [1] and, more recently, Spark [15]. The increasing popularity of Spark is primarily due to its performance advantages over other frameworks. In contrast to Hadoop, where computation stages can only communicate by reading and writing files in a distributed file system, intermediate computations in Spark can be persisted in memory. This subtle but powerful change allows stages in Spark to reuse computations without requiring any slow accesses to persistent storage.

However, Spark's dependence on persisting intermediate computations in memory increases the memory requirements necessary to achieve good performance. Insufficient memory can cause significant slowdowns as intermediate computations will have to be either re-fetched from disk or completely recomputed. Even with sufficient memory, the increased memory requirements can affect performance by increasing JVM garbage collection (GC) latency [9].

Efforts have been made to use external storage systems, such as Tachyon [9], to store Spark's resilient distributed datasets (RDDs). This approach persists RDDs off-heap in local memory, which reduces Spark's GC time. It also allows Spark executors on the same machine to share a single pool of memory. However, when local memory is full, Tachyon will store RDD partitions to secondary storage systems (e.g. disks and SSDs). Workloads with non-uniform memory requirements can cause some nodes in the cluster to exhaust their local memory, requiring writes to disks and SSDs, while leaving other nodes with memory available.

In this paper, we introduce *R3S*, an RDMA-based RDD remote storage system for Spark. Similar to Tachyon, R3S is an external RDD storage system that reduces garbage collection overhead by storing RDDs off the Java heap. However, unlike Tachyon, R3S provides significantly more flexibility in RDD partition placement. R3S manages all of the memory available on the Spark executors as a single memory pool. When local memory is exhausted, RDD partitions can be saved on a remote node's memory. R3S aims to choose a partition's storage location that minimizes partition fetching time in subsequent computation stages. Our prototype actively monitors the amount of memory available on each Spark executor, and selects a storage location for a partition based on this information. We plan to also use other sources of feedback from the system, such as dependency information from the Spark scheduler and the rate of memory consumption on each Spark executor, to further improve our partition placement algorithm.

One of the enabling technologies for this approach is *one-*

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ARM 2016, December 12-16, 2016, Trento, Italy

© 2016 ACM. ISBN 978-1-4503-4662-7/16/12…$15.00

DOI: http://dx.doi.org/10.1145/3008167.3008171

sided RDMA, which allows low-latency remote access to a remote node's memory without requiring any interactions with the node's CPU. Using one-sided RDMA significantly reduces the overhead of storing and retrieving RDD partitions from a remote node. Furthermore, by eliminating CPU involvement, it opens up the possibility of allocating memory-only virtual machine instances from cloud providers. These instances can be allocated and added to the global memory pool when memory is exhausted, and may be more cost-effective than allocating normal virtual machine instances for workloads where adding CPUs to an existing cluster will only minimally impact the completion time. R3S determines which instance type to add based on the estimated improvement in completion time per dollar from adding an instance, which can be calculated offline based on past executions of the workload or online using runtime system statistics.

We have implemented a prototype of *R3S* on top of an RDMA-based key-value store [13]. Our benchmark results show Spark jobs that retrieve RDD partitions from remote memory using RDMA complete in 41% and 71% of the time compared to retrieving the partitions from a hard disk and an SSD respectively.

2. BACKGROUND AND RELATED WORK

Big Data Infrastructure: As data set sizes continue to grow, large-scale data processing frameworks become increasingly more popular. MapReduce [4] and Hadoop [1] are currently the most commonly used scalable distributed programming frameworks. In these frameworks, computation is divided into a series of stages, and each stage consists of a pair of function calls (`map` and `reduce`). The output of one stage is used as the input of later stages, and communication between stages rely on reading and writing to shared files in a distributed file system. Although using shared files to communicate between stages is simple and ensures persistence of the intermediate output, it requires reading and writing to a slow persistent storage medium.

Spark [15, 2] addresses the performance problem of using files to communicate between stages by introducing an immutable data abstraction called Resilient Distributed Dataset (RDD) for storing intermediate data. By tracking the lineage of its RDDs, Spark can ensure that, in the event of a node failure, missing RDD partitions can be regenerated without restarting the computation from the beginning. This property reduces the need to store intermediate output persistently, allowing Spark to store RDDs in memory when space allows.

Storing Spark RDDs: As illustrated in Figure 1, a Spark executor, which takes RDD partitions as its input and generates RDD partitions as its output, has a *BlockManager* to determine where to retrieve or store a given RDD partition. When storing an RDD partition, the BlockManager can choose to store it either in local memory, on disk, or in an external storage system. If it decides to store the partition in memory, it must then choose to either store it as a native Java object, or as a serialized object that is more space efficient but requires deserialization before it can be used.

Although storing RDD partitions in local memory provides better performance than storing them on disk, performance can significantly degrade when storing a large number of partitions in memory due to long Java GC delays required

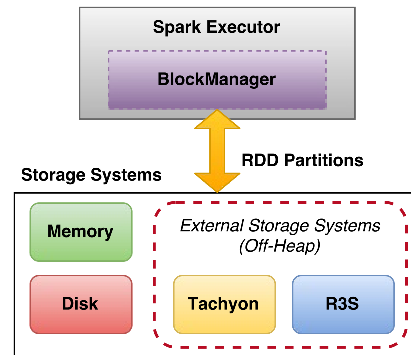


Figure 1: RDD Storage in Spark

to reclaim unused memory. Efforts have been made to use external storage systems to store Spark's RDDs off the Java heap in order to reduce GC overhead. Tachyon [9] is currently the most commonly used external RDD storage system. Although a typical Spark+Tachyon deployment has Tachyon running on every Spark node, Tachyon will store partitions to a lower tier storage system when local memory is exhausted instead of storing the partitions at another Tachyon node. R3S is an external RDD storage system for Spark that uses RDMA to retrieve and store RDD partitions from remote nodes in order to more efficiently utilize the available memory across the entire deployment.

Spark Dynamic Resource Allocation: Spark supports dynamic resource allocation using coarse-grained cluster managers such as YARN [14] and Mesos [6]. In response to changes to its workload, Spark can add more executors or remove idle ones by allocating or deallocating resources from the cluster manager. Each executor is given the same amount of memory and the same number of cores in order to avoid creating stragglers due to computational imbalances.

However, many workloads that generate a significant amount of intermediate data benefit far more from an increase in memory than an increase in the number of CPU cores. Therefore, allocating additional executors to satisfy memory requirements would result in an inefficient allocation of CPU resources. In a cloud computing environment where multiple applications and tenants are sharing the same resources, the underutilized CPU cores allocated to an executor would increase cost without providing a good return on investment. Our approach supports allocating storage-only R3S nodes with minimal CPU allocations but large memory allotments. A storage-only R3S node would provide a cost effective way for memory hungry workloads to allocate more memory without incurring the cost of allocating unnecessary CPU cores. The existing Spark executors would be able to transparently access the storage-only R3S nodes through the external storage system interface; the R3S module in Spark redirects fetch and store requests for RDD partitions to the appropriate node as selected by the R3S storage scheduler.

Remote Direct Memory Access (RDMA): With falling memory prices and the prevalence of RDMA-capable networks, there is an emerging class of computation platforms and in-memory storage systems [5, 12, 8, 13] that leverage RDMA to increase request throughput and reduce latency. RDMA has also been used to replace TCP in order

to improve Spark's network I/O operations [11]. R3S builds on Nessie [13], an RDMA-based in-memory key-value store (RKVS), to retrieve and store RDD partitions from remote nodes using RDMA. Unlike other RKVSes, Nessie uses one-sided RDMA for both read and write operations. This approach significantly reduces the CPU requirements of the server, which is a key-enabler for creating storage-only R3S nodes. R3S extends Nessie by supplying a partition scheduler that determines where partitions should be stored, and which partitions should be evicted to disk when there is inefficient memory to store all partitions in memory.

3. PARTITION PLACEMENT

In this section, we first model the relationship between job completion time and RDD partition location. Based on this model, we outline our RDD placement approach with the goal of minimizing job completion time for a given workload and a fixed amount of resources.

3.1 Cost Model

A Spark deployment consists of a collection of Spark executors that work concurrently on different parts of the problem to complete a computation stage. In our simplified cost model, a Spark job consists of one or more computation stages that execute sequentially. Therefore, the completion time of a job is the sum of the completion time of its stages. Given n Spark executors for a stage, a stage's completion time T_{stage} is defined as follows:

$$T_{stage} = max\{t_i | i = 1, 2, ..., n\} \quad (1)$$

where t_i is the running time of Spark executor i for this stage. Given this definition, reducing a stage's completion time requires reducing the running time of the slowest Spark executor.

A Spark executor's running time is in part determined by the amount of time required to fetch RDD partitions generated from previous stages. Fetching RDD partitions from disk or recomputing RDD partitions can significantly increase a Spark executor's running time. We define FT_i as the time that executor i needs to fetch the RDD partitions that it requires to perform its computation for a given stage. Therefore, t_i can be defined as a function of FT_i:

$$t_i = f(FT_i) \quad (2)$$

In our model, RDD partitions can be located either in local memory (LM), remote memory (RM), or on local disk (LD). We define the time required to fetch an RDD partition from each as FT_{LM}, FT_{RM}, and FT_{LD} where $FT_{LM} \ll FT_{RM} < FT_{LD}$. Similarly, we define the number of RDD partitions that executor i needs to fetch from each storage location as LM_i, RM_i, and LD_i. From these definitions, we can compute FT_i using the following equation:

$$FT_i = LM_i \cdot FT_{LM} + RM_i \cdot FT_{RM} + LD_i \cdot FT_{LD} \quad (3)$$

In general, minimizing FT_i for Spark executor i requires fetching RDD partitions from local memory whenever possible. This is limited by the amount of memory available at each executor. When local memory is exhausted, an RDD placement algorithm should aim to increase the probability of allowing executors to fetch RDD partitions from remote memory instead of from local disk.

Reducing stage completion time also requires placing RDD partitions in a manner that avoid stragglers. Ideally, all executors use the same amount of time to fetch RDD partitions and have approximately the same running time. To achieve this, an RDD placement algorithm should aim to have executors fetch partitions from remote memory and local disks approximately the same number of times by allocating memory fairly across all executors that require more memory than is available locally.

3.2 Partition Placement Approach

Given the cost model in Section 3.1, we propose an RDD partition placement scheme that reduces the partition fetching time (FT) for Spark executors. Because fetching a partition from remote memory or from a local disk results in a significant increase in fetching time, our approach always attempts to store partitions in local memory. If a partition needs to be placed in remote memory, we select the executor that has the most available memory. Without knowing more information about the workload, this choice minimizes the likelihood that the partition will be replaced later.

In the event that memory from all executors is exhausted, our placement approach replaces the partition that is owned by the executor with the largest memory footprint and occupies remote memory. R3S aims to balance the amount of remote memory allocated to the executors in order to avoid creating stragglers. Finally, given a set of candidate partitions, our approach uses LRU to determine the partition to replace.

We use the following notation to formally outline our approach: given Spark executor i, M_i is its assigned maximum amount of local memory; U_i is the amount of unused local memory; L_i is the amount of local memory that it uses; R_i is the amount of remote memory that it uses; Q_i is the total amount of memory that it uses. Q_i is the sum of L_i and R_i as shown in the following equation:

$$Q_i = L_i + R_i \quad (4)$$

When Spark executor i stores an RDD partition, our approach decides the partition's storage location as follows:

- If $L_i < M_i$, the partition is stored to local memory.
- If $L_i == M_i$ and $\sum_{i=1}^{n} Q_i < \sum_{i=1}^{n} M_i$, the partition is persisted to the memory of Spark executor r that is determined by the following equation:

  $$U_r = max\{U_j | j = 1, 2, ..., n, j \neq i\} \quad (5)$$

- If $L_i == M_i$ and $\sum_{i=1}^{n} Q_i >= \sum_{i=1}^{n} M_i$, the partition is stored to the remote memory that remotely stores the data of Spark executor k, which is determined by:

  $$Q_k = max\{Q_j | j = 1, 2, ..., n\} \quad (6)$$

  The data of executor k will be evicted and be stored to its disk.

Under the first condition of the above three listed ones, data eviction may be required on local memory. The evicted data belongs to Spark executor m that is determined by using Equation 6. In this case, if memory is available in the cluster, the evicted data is stored on a Spark executor that is determined by Equation 5. Without available memory, the evicted data is stored to the disk of Spark executor m.

A workload's runtime information can also be used to improve the effectiveness of our partition placement scheme. For example, we can estimate the fetching order of RDDs

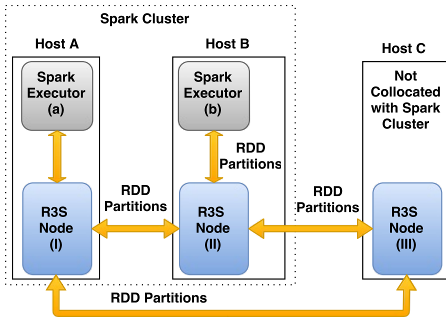


Figure 2: An Example of an R3S Deployment

by analyzing the dependency DAG between RDDs from the Spark scheduler. By knowing the fetching order, we can reduce partition fetching time by keeping partitions that will likely be requested in the near future in local memory. Furthermore, if we can use runtime information to estimate the memory requirements of a Spark executor for a stage, we can use this information to reserve local memory at each Spark executor. Without local memory reservations, a Spark executor may write to a remote location that will later experience local memory pressure, resulting in eviction of its data. Our future work includes further exploring the use of dynamic feedback in our partition placement algorithm.

4. RESOURCE ALLOCATION

Many applications repeat Spark jobs regularly with new or updated input. For example, an application that aggregates statistical data at the end of each business day performs the same task each time it runs. This provides us with the opportunity to perform offline analysis to determine the resource requirements of a job's different stages. Specifically, offline analysis can determine the amount of memory required at each stage to ensure that the vast majority of RDD partition retrievals are from local and remote memory instead of from disk.

Given a large budget to execute a job, a sufficient number of Spark executors, each of which has a fixed allocation of CPU cores and memory, can be allocated for each stage to meet the stage's memory requirements, which is pre-determined by using offline analysis. Additional Spark executors can be allocated to further reduce job completion time. We can determine which stage to add an executor by determining the marginal job completion time reduction per dollar (MJCT/$) from adding one executor for each stage, and choosing the stage with the highest MJCT/$.

In the case where there is insufficient budget to allocate only Spark executors to meet memory requirements, R3S storage-only nodes can be allocated in place of executors in order to reduce cost. However, R3S must satisfy the constraint that each stage must have at least one executor. We use the same MJCT/$ metric to determine which stage to select, and we allocate storage-only nodes in place of executors until we meet or come under our budget.

For other applications where historical information is not available, we can perform online analysis to estimate the memory and CPU requirements of each stage. The analysis takes into account online system and workload statistics, such as CPU utilization, data size, and RDD dependencies, in generating its estimate. R3S uses this estimate to calculate the MJCT/$ of both an executor and a storage-only node.

Another factor that affects MJCT/$ is the cost of an instance. A cloud provider may sell an instance with a large amount of memory but minimal CPU allocation at a significant discount compared to one of its regular instances. This would encourage the use of storage-only nodes. Instance pricing can change rapidly in spot pricing markets, in which prices change in response to changes in demand. R3S can reactively take advantage of price volatility by actively monitoring the spot pricing market to adjust its MJCT/$ estimates.

Allocating storage-only nodes also enables flexible resource allocation within a cluster. For example, in a multi-application cluster, a computation-intensive application may require a large number of CPUs but only consume a small amount of memory. Therefore, there is an excess of memory available that Spark would be unable to completely use since it cannot allocate memory independently of CPU. Storage-only nodes in R3S break that dependency, allowing Spark to take advantage of CPU and memory imbalances in multi-application clusters.

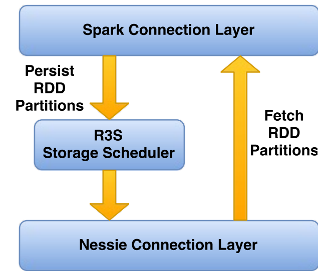


Figure 3: Architecture of an R3S Node

5. R3S ARCHITECTURE

R3S is an external RDD storage system for Spark. Similar to Tachyon [9], R3S is a Spark plugin that persists RDDs, so that they are stored off the JVM heap, which reduces the GC time. GC time can be significant when processing large workloads that require hundreds of gigabytes of memory on each Spark node. However, unlike Tachyon, R3S manages all of the allotted memory in its deployment as a single memory pool. This allows RDD partitions to be persisted to remote memory in R3S. R3S uses Nessie [13] as the backend storage system for its memory pool.

An R3S deployment consists of a group of R3S nodes, where each R3S node provides storage for RDD partitions. An R3S node may or may not be collocated with a Spark node. Figure 2 illustrates an R3S deployment in a shared cluster environment, where some R3S nodes are collocated with the Spark cluster. In this example, Hosts A and B are two machines that belong to the Spark cluster, whereas host C does not. R3S nodes I, II and III are deployed on hosts A, B and C respectively. Spark executors a and b use R3S nodes I and II to locally store RDDs. Furthermore, R3S node III is not collocated with any Spark node; nonetheless, it may be used by other Spark executors to store RDDs. An RDD partition can be persisted to any of these R3S nodes; however, a Spark executor does not need to be aware of the R3S node that its RDD partitions are persisted to.

Figure 3 shows the three main components of an R3S node:

- Spark Connection Layer (SCL)
- R3S Storage Scheduler (RSS)
- Nessie Connection Layer (NCL).

SCL implements Spark's RDD storage layer interface. As RDD partitions are generated, they are sent to the SCL for persistence. The SCL queries its local RSS instance, which is responsible for determining the storage location of an RDD partition. Furthermore, each local RSS instance tracks local memory usage, where some memory may be used by remote Spark executors. Each R3S deployment has a master RSS that is a global scheduler. This master RSS is responsible for determining where to store RDD partitions, and deciding if eviction of RDD partitions is necessary, which we have described in Section 3. In order to facilitate global scheduling of RDD partitions, the RSS master requires the memory usage of each Spark executor at each R3S node. This information is acquired by either informing the master once an RDD partition has been persisted or by having the master pull information from each local RSS instance.

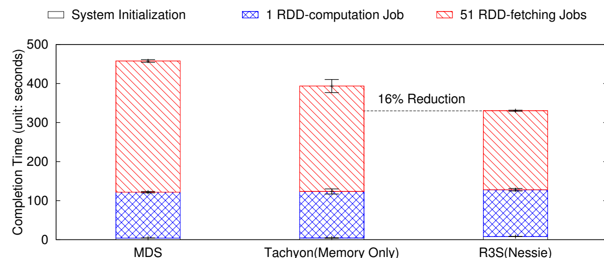
After determining the RDD partition's storage node from the RSS, the SCL forwards the RDD partition to the storage node's NCL instance. An R3S node has a single Nessie node, and it's NCL instance connects to its own Nessie node. Each RDD partition is stored as a key-value pair in Nessie, where the key is its partition identifier. Transferring data over the network to store RDD partitions does not introduce significant overhead because data persistence is a non-blocking operation in Spark. In the future, we plan to have the SCL forward the RDD partition to a local NCL instance with the desired location as a parameter. This would enable us to take advantage of the one-sided RDMA write support in Nessie.

To retrieve an RDD partition, the SCL forwards the request to its local instance of the NCL. The NCL retrieves the RDD partition from Nessie. The RSS is not involved in RDD partition retrieval as the RDD partition's location is maintained by Nessie. Furthermore, the NCL also takes advantage of Nessie's one-sided RDMA read operation to reduce the CPU requirements of fetching an RDD partition from remote memory.

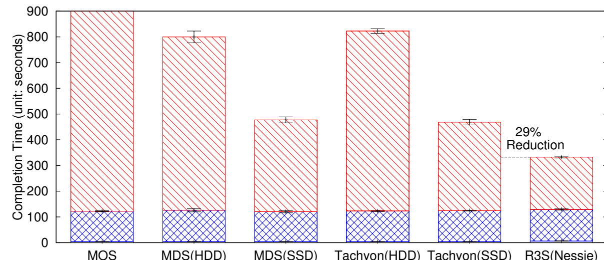
6. IMPLEMENTATION

We have implemented a prototype of R3S that can be integrated into Spark v1.5 or higher. Like Tachyon, R3S connects with Spark as a plugin by implementing Spark's *ExternalBlockManager* interface. To persist RDDs into R3S, Spark applications just need to specify R3S as the RDD external storage system in their configuration files and use the corresponding RDD storage level, *OFF_HEAP*, to persist RDDs. The RSS master is implemented in Java and runs as an independent process in our prototype. The RPC communication between the RSS master and a local RSS instance uses Apache Thrift [3].

As Nessie is implemented in C++ and a Spark executor runs in a JVM, the NCL uses Java Native Interface (JNI) to allow RDD partitions in a JVM to be stored in Nessie. To reduce the memory-copy overhead of passing a serialized RDD partition (i.e. a byte array) through JNI, the NCL directly reads or writes data in JVM memory space by passing the array's starting address as a pointer into the JNI code.



(a) 100% RDD Partitions Stored in Local Memory



(b) 50% RDD Partitions Stored in Local Memory

Figure 4: Logistic regression benchmark results. MOS and MDS denote memory-only with serialization and memory and disk with serialization, respectively.

7. PRELIMINARY EVALUATION

In this section, we present some preliminary results that demonstrate the advantages of using R3S to store Spark's intermediate computation data. In particular, our evaluation focuses on the reduction of job completion time when RDD partitions are persisted to remote memory. We first introduce our experimental setup. We then describe our logistic regression workload, which belongs to SparkBench [10]. Finally, we present the results of our experiments.

7.1 Experiment Setup

Our test environment consists of 4 machines that are connected to a 40 Gbps network. Each machine has two 6-core Intel E5-2630v2 CPUs. In order to simulate a shared cluster environment, two machines are set to form a Spark cluster. We deploy Spark 1.5.1 and HDFS 2.4 on this cluster. Spark executors run on these two machines, and they are configured to have the same amount of memory for all experiments. The remaining two machines do not belong to the aforementioned Spark cluster. Instead, these machines run non-Spark jobs that utilize most of their CPU; however, a large amount of memory on these machines remain unused.

The benchmark input data, which we describe in detail in the following section, is stored in HDFS. To maintain a reasonable execution time, the input data size is 3.2 GB. From this, our benchmark generates a 5 GB RDD. The RDD has 5000 partitions, where each partition is about 1 MB. The benchmark requires less than 40 MB of shuffle data, which is negligible in terms of memory usage. In our evaluation, we compare Tachyon (0.7.1) and R3S, where each system is deployed on all four machines.

7.2 Logistic Regression Benchmark

Logistic regression is widely used as a classifier for data prediction [7, 10] in machine learning applications. The logistic regression benchmark begins by pre-processing input data from HDFS and then storing the output in memory. Subsequent jobs use the stored data to train a machine learning model. If we do not limit the number of iterations, the benchmark finishes after executing 53 Spark jobs. The first job counts the input data, and the second job pre-processes the data and stores the output as an RDD. These two jobs are presented as one RDD-computation job. The remaining 51 jobs read the stored RDD and use it for further training, which are referred to as RDD-fetching jobs.

Our experiments compare three RDD storage approaches: R3S, Tachyon and Spark's memory and disk with serialization (MDS). Figure 4(a) shows the benchmark's completion time when all RDD partitions are stored in the Spark machines' local memory. MDS has the longest completion time due to Java GC overhead in the Spark executors. Both R3S and Tachyon persist RDDs off the JVM heap of Spark executors, which results in smaller GC overhead. Figure 4(a) also shows that R3S achieves a shorter job completion time when compared with Tachyon. This is because Tachyon uses file streaming to read an RDD partition, which is inefficient at reading a large number of small RDD partitions.

Figure 4(b) compares Spark's completion time when using different RDD storage systems. In this experiment, each Spark executor persists 50% of its RDD partitions to local memory, which is approximately 1.25GB. Memory-only with serialization (MOS) is the slowest RDD storage mechanism requiring 3371 seconds to complete. Using MOS, RDD partitions are dropped after local memory is full. Re-computing the dropped partitions from the input is slower than fetching RDD partitions from disk. The benchmark completes in approximately 40% less time when using an SSD compared to a hard disk. Storing RDD partitions in remote memory with R3S results in about 30% reduction in benchmark completion time when compared to using an SSD.

8. CONCLUSION

This paper introduces R3S, an RDMA-based remote storage system to improve memory utilization across a cluster by allowing RDD partitions to be stored at remote locations from its Spark executor and retrieved through high-performance one-sided RDMA operations. The R3S partition placement approach aims to reduce partition fetching time for Spark executors by storing partitions in local memory whenever possible, storing them at remote executors with the most memory available when local memory is exhausted, and balancing the memory footprint among the executors when memory across the cluster is exhausted. We also introduce storage-only nodes that can efficiently replace Spark executors in budget-constrained situations. Compared with Tachyon, our experimental results show that storing data to remote memory with R3S achieves a 29% reduction in completion time for a logistic regression benchmark.

9. ACKNOWLEDGMENTS

This work is supported by the Natural Sciences and Engineering Research Council of Canada. We would like to thank the anonymous reviewers for their valuable feedback. We would also like to thank Benjamin Cassell, Jonathan Ma, Zhenyu Bai for their help with Nessie and the Spark benchmark, and Jack Ng for his technical advice on the project.

10. REFERENCES

[1] Apache. Hadoop. https://hadoop.apache.org.

[2] Apache. Spark. https://spark.apache.org.

[3] Apache. Thrift. https://thrift.apache.org.

[4] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Proceedings of OSDI*. USENIX, 2004.

[5] A. Dragojević, D. Narayanan, O. Hodson, and M. Castro. Farm: Fast remote memory. In *Proceedings of NSDI*. USENIX, 2014.

[6] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *Proceedings of NSDI*. USENIX, 2011.

[7] G. James, D. Witten, T. Hastie, and R. Tibshirani. *An Introduction to Statistical Learning: With Applications in R*. Springer, 2014.

[8] A. Kalia, M. Kaminsky, and D. G. Andersen. Using rdma efficiently for key-value services. In *Proceedings of SIGCOMM*. ACM, 2014.

[9] H. Li, A. Ghodsi, M. Zaharia, S. Shenker, and I. Stoica. Tachyon: Reliable, memory speed storage for cluster computing frameworks. In *Proceedings of SOCC*. ACM, 2014.

[10] M. Li, J. Tan, Y. Wang, L. Zhang, and V. Salapura. Sparkbench: A comprehensive benchmarking suite for in memory data analytic platform spark. In *Proceedings of the 12th International Conference on Computing Frontiers*. ACM, 2015.

[11] X. Lu, M. Wasi-ur Rahman, N. Islam, D. Shankar, and D. Panda. Accelerating spark with rdma for big data processing: Early experiences. In *Proceedings of the 22nd Annual Symposium on High-Performance Interconnects*. IEEE, 2014.

[12] C. Mitchell, Y. Geng, and J. Li. Using one-sided rdma reads to build a fast, cpu-efficient key-value store. In *Proceedings of ATC*. USENIX, 2013.

[13] T. Szepesi, B. Wong, B. Cassell, and T. Brecht. Designing a low-latency cuckoo hash table for write-intensive workloads using RDMA. In *Proceedings of the International Workshop on Rack-scale Computing*, Amsterdam, The Netherlands, 2014.

[14] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O'Malley, S. Radia, B. Reed, and E. Baldeschwieler. Apache hadoop yarn: Yet another resource negotiator. In *Proceedings of SOCC*. ACM, 2013.

[15] 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 *Proceedings of NSDI*. USENIX, 2012.