Lawrence Berkeley National Laboratory
LBL Publications

Title
Scaling Spark on HPC Systems

Permalink
https://escholarship.org/uc/item/84s551t7

ISBN
9781450343145

Authors
Chaimov, Nicholas
Malony, Allen
Canon, Shane
et al.

Publication Date
2016-05-31

DOI
10.1145/2907294.2907310

Peer reviewed

eScholarship.org Powered by the California Digital Library
University of California

Scaling Spark on HPC Systems

Nicholas Chaimov Allen Malony
University of Oregon
{nchaimov,malony}@cs.uoregon.edu

Shane Canon Costin Iancu
Khaled Z. Ibrahim Jay Srinivasan
Lawrence Berkeley National Laboratory
{scanon,cciancu,kzibrahim,jsrinivasan}@lbl.gov

ABSTRACT

We report our experiences porting Spark to large production HPC systems. While Spark performance in a data center installation (with local disks) is dominated by the network, our results show that file system metadata access latency can dominate in a HPC installation using Lustre: it determines single node performance up to 4× slower than a typical workstation. We evaluate a combination of software techniques and hardware configurations designed to address this problem. For example, on the software side we develop a file pooling layer able to improve per node performance up to 2.8×. On the hardware side we evaluate a system with a large NVRAM buffer between compute nodes and the backend Lustre file system: this improves scaling at the expense of per-node performance. Overall, our results indicate that scalability is currently limited to $O(10^2)$ cores in a HPC installation with Lustre and default Spark. After careful configuration combined with our pooling we can scale up to $O(10^4)$. As our analysis indicates, it is feasible to observe much higher scalability in the near future.

CCS Concepts

•**Software and its engineering → Ultra-large-scale systems; Cloud computing;**

Keywords

Spark, HPC, Data Analytics

1. INTRODUCTION

Frameworks such as Hadoop [30] and Spark [32] provide a productive high level programming interface for large scale data processing and analytics. Through specialized runtimes they attain good performance and resilience on data center systems for a robust ecosystem of application specific libraries [14, 22, 5]. This combination resulted in widespread adoption that continues to open new problem domains.

ACM acknowledges that this contribution was authored or co-authored by an employee, or contractor of the national government. As such, the Government retains a nonexclusive, royalty-free right to publish or reproduce this article, or to allow others to do so, for Government purposes only. Permission to make digital or hard copies for personal or classroom use is granted. Copies must 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. To copy otherwise, distribute, republish, or post, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

HPDC'16, May 31-June 04, 2016, Kyoto, Japan

© 2016 ACM. ISBN 978-1-4503-4314-5/16/05…$15.00

DOI: http://dx.doi.org/10.1145/2907294.2907310

As multiple science fields have started to use analytics for filtering results between coupled simulations (e.g. materials science or climate) or extracting interesting features from high throughput observations (e.g. telescopes, particle accelerators), there exists plenty incentive for the deployment of the existing large scale data analytics tools on High Performance Computing systems. Yet, most solutions are ad-hoc and data center frameworks have not gained traction in our community. In this paper we report our experiences porting and scaling Spark on two current very large scale Cray XC systems (Edison and Cori), deployed in production at National Energy Research Scientific Computing Center (NERSC) [2].

In a distributed data center environment disk I/O is optimized for latency by using local disks and the network between nodes nodes is optimized primarily for bandwidth. In contrast, HPC systems use a global parallel file system, with no local storage: disk I/O is optimized primarily for bandwidth, while the network is optimized for latency. Our initial expectation, was that after porting Spark to Cray, we can then couple large scale simulations using $O(10^4)$ cores, benchmark and start optimizing it to exploit the strengths of HPC hardware: low latency networking and tightly coupled global name spaces on disk and in memory.

We ported Spark to run on the Cray XC family in Extreme Scalability Mode (ESM) and started by calibrating single node performance when using the Lustre [7] global file system against that of an workstation with local SSDs: in this configuration a Cray node performed up to 4× slower than the workstation. Unlike clouds, where due to the presence of local disks Spark shuffle performance is dominated by the network [25], file system metadata performance initially dominates on HPC systems. Perhaps expected by parallel I/O experts [21], the determining performance factor is the file system metadata latency (e.g. occurring in `fopen`), rather than the latency or bandwidth of read or write operations. We found the magnitude of this problem surprising, even at small scale. Scalability of Spark when using the back-end Lustre file system is limited to $O(10^2)$ cores.

After instrumenting Spark and the domain libraries evaluated (Spark SQL, GraphX), the conclusion was that a solution has to handle *both* high level domain libraries (e.g. Parquet data readers or application input stage) *and* the Spark internals. We calibrated single node performance, then we performed strong and weak scaling studies on both systems. We evaluate software techniques to alleviate the single node performance gap in the presence of a parallel file system:

- First and most obvious configuration is to use a local file system, in main memory or mounted to a single Lustre file, to handle the intermediate results generated during the computation. While this configuration does not handle the application level I/O, it improves performance during the Map and Reduce phases and a single Cray node can match the workstation performance. This configuration enables scaling up to 10,000 cores and beyond, for more details see Section 5.3. We have extended and released the Shifter [18] container framework for Cray XC with this functionality. Deploying Spark on Shifter has unexpected benefits for the JVM performance and we observe 16% performance improvements when running in memory on $\approx$ 10,000 cores.
- As the execution during both application initialization and inside Spark opens the same file multiple times, we explore "caching" solutions to eliminate file metadata operations. In Spark, the number of files used grows linearly with the number of cores, while the number of file opens grows quadratically with cores. We developed a layer to intercept and cache file metadata operations at both levels. A single Cray node with pooling also matches workstation performance and overall we see scalability up to 10,000 cores. Combining pooling with local file systems also improves performance (up to 17%) by eliminating system calls during execution.

On Cori we also evaluate a layer of non-volatile storage (`BurstBuffer`) that sits between the processors' memory and the parallel file system, specifically designed to accelerate I/O performance. Performance when using it is better than Lustre (by 3.5× on 16 nodes), but slower than RAM-backed file systems (by 1.2×), for *GroupBy*, a metadata-heavy benchmark. With `BurstBuffer` we can scale Spark only up to 1,200 cores. The improvements come from better `fopen` scalability, rather than read/write latency and illustrate the principle that optimizing for the tail is important at scale: the `BurstBuffer` median open latency is higher than Lustre's, but its variance is much smaller than on Lustre.

Besides metadata latency, file system access latency in `read` and `write` operations may limit scalability. In our study, this became apparent when examining iterative algorithms. As described in Section 6, the Spark implementation of PageRank did not scale when solving problems that did not fit inside the node's main memory. The problem was the interplay between resilience mechanisms and block management inside the shuffle stage in Spark, that generated a number of I/O requests that increased exponentially with iterations. This overwhelmed the centralized storage system. We fixed this particular case at the algorithmic level, but a more generic approach is desirable to cover the space of iterative methods.

Overall, our study indicates that scaling data analytics frameworks on HPC systems is likely to become feasible in the near future: a single HPC style architecture can serve both scientific and data intensive workloads. The solution requires a combination of hardware support, systems software configuration and (simple) engineering changes to Spark and application libraries. Metadata performance is already a concern for scientific workloads and HPC center operators are happily throwing more hardware at the problem. Hardware to increase the node local storage with large NVRAM will decrease both metadata and file access overhead through better caching close to the processors. Orthogonal software techniques, such the ones evaluated in this paper, can further reduce metadata impact. In fact, at the time of the publication, our colleagues at NERSC have demonstrated Spark runs at $\approx 50,000$ cores using Shifter with our Lustre mounted local file system configuration. An engineering audit of the application libraries and the Spark internals will also eliminate many root causes of performance bottlenecks.

2. SPARK ARCHITECTURE

Apache Spark [32] and Hadoop [30] are open-source data analytics frameworks, designed to operate on datasets larger than can be processed on a single node while automatically providing for scheduling and load-balancing. They implement the Map-Reduce model [12] using an abstraction in which programs are expressed as data flow graphs. The nodes in the graph are of two types: *map operations*, which are purely local, and *reduce operations*, which can involve communication between nodes.

The traditional MapReduce framework [12] is limited to acyclic graphs, preventing efficient representation of iterative methods, and it uses data redundancy to provide resiliency. Spark can handle cyclic and acyclic graphs, and provides resiliency through *resilient distributed datasets* [31] (RDD), which carry sufficient information (lineage) to recompute their contents. In particular, the ability to express iterative algorithms accelerated Spark's adoption.

Programs are expressed in terms of RDDs derived from transformations of other RDDs (e.g. Map) and actions (e.g. Reduce). The application developer can choose to request that certain RDDs be cached in memory or saved to disk. The developer therefore has to make decisions based on tradeoffs between the costs of storage (in memory and time) and recomputation (in time). RDDs are lazily evaluated, which creates challenges [6] in attributing performance to particular lines or regions of code, as they do not execute until they are needed.

In Spark, the *Master* node executes a driver program, which creates the data flow graph by applying transformations and actions to RDDs, and partitions ownership of data to worker nodes within the cluster. When the result of an uncomputed RDD partition is needed, a *job* is created, consisting of multiple *stages*. Within a stage, only intra-partition communication can occur. All inter-partition communication happens at stage boundaries, through a process called *shuffling*, as shown in Figure 1. By deferring any computation until a result is needed, the scheduler can schedule work to compute only what is necessary to produce the result. In the event of the loss of a partition, only that partition needs to be recomputed.

2.1 Data Movement in Spark

Data movement is one of the performance determining factors in any large scale system. In Spark, data is logically split into *partitions*, which have an associated worker task. A partition is subdivided into *blocks*: a block is the unit of data movement and execution. Figure 2 shows the interaction of the Spark compute engine with the block and shuffle managers, which control data movement. The BlockManager handles application level input and output data, as well as intermediate data within the Map stages. The Shuf-

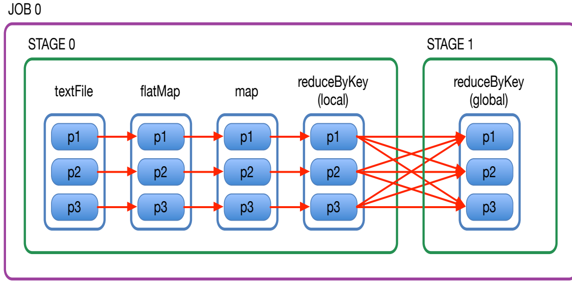


Figure 1: *Decomposition of a job into stages and tasks on partitions, with inter-partition communication limited to stage boundaries.*

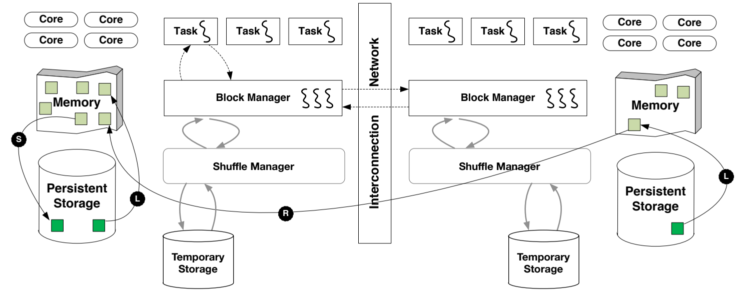


Figure 2: *Data movement in Spark and the interaction with the memory hierarchy.*

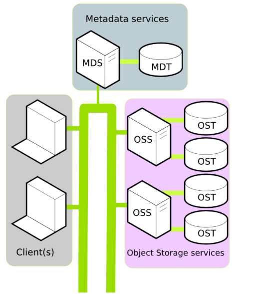


Figure 3: *Lustre architecture. (Courtesy of Intel Wiki.)*

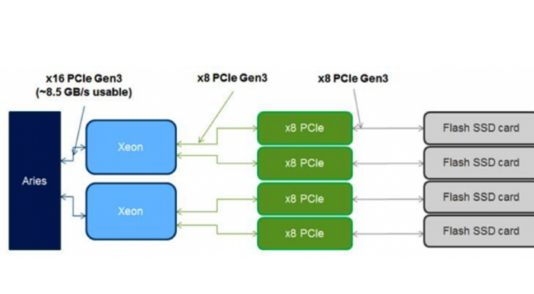


Figure 4: `BurstBuffer` *node architecture.(Courtesy of NERSC.)*

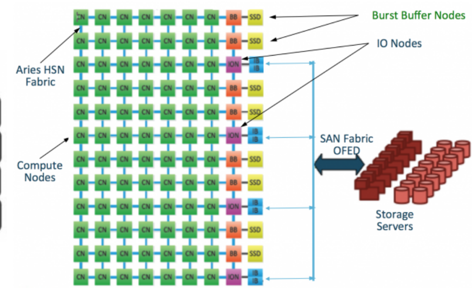


Figure 5: `BurstBuffer` *topology. (Courtesy of NERSC.)*

fleManager handles runtime intermediate results during the shuffle stage.

Data Objects and Naming: Spark manipulates data with global scope, as well as local scope. Application level data (RDDs) are using a global naming space, intermediate data blocks generated throughout execution have a local scope and naming scheme. Objects may exceed the capacity of the physical memory and need to be efficiently moved through the storage hierarchy; the typical challenge when managing naming schemes is mismatch with underlying system architecture. For instance, when global object is distributed (partitioned) across multiple storage spaces a long latency naming service may be needed to locate its physical location. Conversely, any locally named object stored in a physically shared storage may experience undue contention while servicing requests. A current research direction in the Spark community is providing an efficient global naming service, which can reduce network traffic. Note that the global file system in HPC installations provides global naming.

Vertical Data Movement: Vertical data movement refers to the movement through the entire memory hierarchy, including persistent storage. It is needed to move input data blocks into the memory for processing and for storing output data to the persistent storage. To minimize vertical movement for RDDs, Spark allows persisting data in the fast level of memory. As fast memory is capacity constrained, the Spark runtime assigns the task of moving objects across the memory hierarchy to a block manager. Whenever the working set size (input data or intermediate results) exceeds memory capacity, the block manager may trigger vertical data movement. The block manager may also decide to drop a block, in which case its later access may trigger additional vertical data movement for recomputation. Research efforts such as Tachyon [19] aim to reduce expensive (to storage) vertical data movement by replacing it with horizontal (inter-node) data movement. In network-based storage systems, a critical [4, 8] component to the performance of vertical data movement is the file setup stage (communication with the metadata servers).

Horizontal Data Movement - Block Shuffling: The horizontal data movement refers to the shuffle communication phase between compute nodes. Spark assigns the horizontal data movement to the shuffle manager and the block manager. A horizontal data movement request of a block could trigger a vertical data movement because a block may not be resident in memory. Optimizing the performance of horizontal data movement has been the subject of multiple studies [29, 17, 20], in which hardware acceleration such as RDMA is used to reduce the communication cost. The benefit of these techniques is less profound on HPC systems with network-based storage [26] because the performance is dominated by vertical data movement.

2.2 System Architecture and Data Movement

Data centers have local storage attached to compute nodes. This enables fast vertical data movement and the number of storage disks scales linearly with the number nodes involved in the computation. Their bandwidth also scale with the number of compute nodes. The archetypal file system for data analytics is the Hadoop Distributed File System (HDFS) which aims to provide both fault tolerance and high throughput access to data. HDFS implements a simple coherency for write-once-read-many file access, which fits

well the Spark and Hadoop processing models. In Spark with HDFS, global naming services are implemented in a client-server paradigm. A request is generated for the object owner, subject to the network latency. The owner services it, maybe subject to disk latency (or bandwidth) and the reply is subject to network latency (or bandwidth). Vertical data transfers access the local disk. Horizontal data are subject to network latency/bandwidth, as well as disk latency/bandwidth.

HPC systems use dedicated I/O subsystems, where storage is attached to a "centralized" file system controller. Each and all nodes can see the same amount of storage, and bandwidth to storage is carefully provisioned for the system as a whole. Given that these network file servers are shared between many concurrently scheduled applications, the servers typically optimize for overall system throughput. As such individual applications may observe increase in latency and higher variability. The Lustre [7] architecture, presented in Figure 3 is carefully optimized for throughput and implements a generic many-write-many-read coherency protocol. The installation consists of clients, a Metadata service (MDS) and Object Storage service. The Metadata service contains Metadata Servers, which handle global naming and persistence and the Metadata Targets which provide the actual metadata storage (HDD/SSD). In Spark with Lustre, global naming services access the metadata servers and are subject to network latency and MDS latency. Most existing Lustre installations in production (prior to Lustre 2.6) use a single MDS, only very recent installations [1, 10] use multiple MDSes for improved scalability. Vertical data transfers are served by the Object Storage service, which contains the object Storage Server (OSS) and the Object Storage Target (OST), the HDD/SSD that stores the data. Bandwidth is provisioned in large scale installations by adding additional OSSes.

In our quest to introduce Spark into the HPC community there are two main questions to answer.

1. *How does the differences in architecture between data centers and HPC influence performance?* Previous performance studies of Spark in data center environments [25] indicate that its performance is dominated by the network, through careful optimizations to minimize vertical data movement and maximize the memory resident working set. Ousterhout et al. [23] analyzed the performance of the Big Data Benchmark [28] on 5 Amazon EC2 nodes, for a total of 40 cores, and the TPC-DS benchmark [24] on 20 nodes (160 cores) on EC2. These benchmarks both use Spark SQL [5], which allows SQL queries to be performed over RDDs. By instrumenting the Spark runtime, they were able to attribute time spent in tasks to several factors, including network and disk I/O and computation. They found that, contrary to popular wisdom about data analytics workflow, that disk I/O is not particularly important: when all work is done on disk, the median speedup from eliminating disk I/O entirely was only 19%, and, more importantly, when all RDDs are persisted to memory, only a 2-5% improvement was achieved from eliminating disk I/O. Upon introduction to HPC systems, we similarly need to understand whether access to storage or network performance dominates within Spark.

2. *What HPC specific features can we exploit to boost Spark performance?* Previous work optimizing data analytics frameworks on HPC systems [20, 17] proposes moving away from the client-server distributed paradigm and exploiting the global file name space already available or Remote Direct Memory Access (RDMA) functionality. Upon introduction to HPC systems, we are interesting in evaluating the potential for performance improvement of adopting such techniques into Spark. Besides providing an initial guide to system researchers, we are also interested in providing configuration guidelines to users and system operators.

We explore these questions using three benchmarks selected to cover the performance space: 1) *BigData Benchmark* uses SparkSQL [5] and stresses vertical data movement; 2) *GroupBy* is a core Spark benchmark designed to capture the worst case scenario for shuffle performance, it stresses both horizontal and vertical data movement; and 3) *PageRank* is an iterative algorithm from GraphX [14] and stresses vertical data movement.

3. EXPERIMENTAL SETUP

We conducted our experiments on the Edison and Cori Cray XC supercomputers at NERSC [2]. Edison contains 5,576 compute nodes, each with two 2.4 GHz 12-core Intel "Ivy Bridge" processors. Cori contains 1,630 compute nodes, each with two 2.3 GHz 16-core Intel "Haswell" processors. Both systems use a Cray Aries interconnect based on the Dragonfly topology.

Cray provides a Cluster Compatibility Mode (CCM) for compute jobs requiring specialized services, such as secure connection, etc. CCM runs Linux and allows an easy path to configure Spark, but imposes limits on the number of nodes per job. More importantly, it disables network transfer mechanisms accelerated by the Aries hardware.

In this study, we ported Spark 1.5.0 to run on the Cray Extreme Scalability Mode (ESM) to allow better scaling of resources. In ESM, a lightweight kernel runs on the compute nodes and the application has full access to Aries. Spark 1.6 has been subsequently released: as file I/O patterns did not change the optimizations we describe in this paper remain applicable to it. We use one manager per compute node, based on YARN 2.4.1. This required additional porting efforts to allow TCP-based services. Compared to Spark's standalone scheduler, YARN allows better control of the resources allocated in each node. The Mesos [16] resource manager provides similar control as YARN, but requires administrative privilege. Job admission is done through a resource manager on the front-end node where Spark runs as a YARN client with exclusive access to all resources.

Both Edison and Cori use the Lustre file system. On Edison, the Lustre file system is backed by a single metadata server (MDS) and a single metadata target (MDT) per file system. On Cori, a master MDS is assisted by a 4 additional Distributed Namespace (DNE) MDSes. The DNEs do not yet support full functionality, and for all Spark concerns Cori performs as a single MDS system.

On Cori we also evaluate a layer of non-volatile storage (`BurstBuffer`) that sits between the processors' memory and the parallel file system, specifically designed to accelerate I/O performance. The NERSC hardware is based on Cray DataWarp and presented in Figures 4 and 5. The flash memory for Cray DataWarp is attached to Burst Buffer nodes that are packaged two nodes to a blade. Each Burst Buffer node contains a Xeon processor 64 GB of DDR3 memory, and two 3.2 TB NAND flash SSD modules attached over two PCIe gen3 x8 interfaces. Each Burst Buffer node is attached to a Cray Aries network interconnect over a PCIe

gen3 x16 interface. Each Burst Buffer node provides approximately 6.4 TB of usable capacity and a peak of approximately 5.7 GB/sec of sequential read and write bandwidth. The `BurstBuffer` nodes can be accessed from the compute nodes in *private* mode and in *striped* mode. Ours is the first evaluation on such technology at scale. However, since the hardware is new and not tuned yet for production, the `BurstBuffer` results are only indications of its potential and it features; we expect them to evolve and improve.

We evaluate *BigData Benchmark*, *GroupBy* and *PageRank* in both weak and strong scaling experiments. Together they provide good coverage of the important performance factors in Spark. *BigData Benchmark* has inputs up to five nodes and we'll concentrate the node level performance discussion around it. *GroupBy* scales and we evaluate it up to 10,240 cores. For *PageRank* we have only small inputs available and evaluate it only up to 8 nodes. Each benchmark has been executed at least five times and we report mean performance. Some `BurstBuffer` experiments were very noisy and we report only the best performance.

4. SINGLE NODE PERFORMANCE

To calibrate initial performance, we evaluated a single node of Cori and Edison against a local workstation with fast SSDs: eight 3.5GHz Xeon i7-3770K cores with 1TB fast SSD. Figure 6 shows the performance of queries 1-3 of the Big Data Benchmark [28] using both on-disk and in-memory modes. The results are quite similar on Edison and Cori. As shown, a single node of Edison when running with eight cores and accessing the file system is roughly twice as slow than the workstation. When data is preloaded in memory, eight cores of Edison match the workstation performance; this is expected as the workstation contains server grade CPUs. When scaling up the Edison node and using all 24 cores, performance is still 50% slower than the workstation. This slowdown is entirely attributed to the file system; performance scales with cores when running with data preloaded in memory, as illustrated when comparing eight cores with the full node performance.

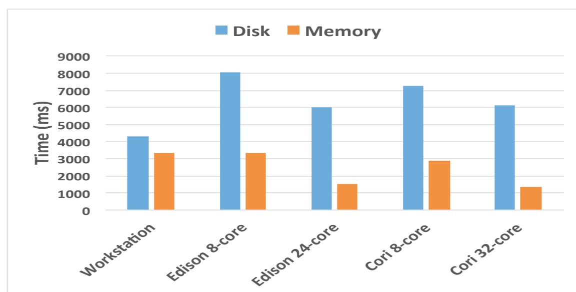


Figure 6: *BigData Benchmark performance on workstation and a single node of Edison and Cori. Input data is precached in memory or read from disk.*

To quantify the difference in I/O performance, we instrumented the Hadoop LocalFileSystem interface used by Spark to record the number of calls and the time spent in `open`, `read`, `write`, and `close` file operations. The time spent in `read`, `write`, and `close` operations did not significantly differ between the systems, while file `open` operations were *much* slower, as shown in Figure 7. On the workstation the mean file open time was 23 μs; on Edison it was 542 μs, almost 24 times greater. Some file open operations on Edison took an extreme amount of time to complete: in the worst case observed, a single file open operation took 324 ms.

The Big Data Benchmark illustrates the application level I/O bottlenecks. At this stage, the number of open operations is linear in the number of partitions. The dataset for Query 1 consists of a single directory containing one data file per partition in Parquet format: there are 3,977 partitions/files. Each file is accompanied by a checksum file used to verify data integrity. These all must be opened, so a minimum of 7,954 file opens must occur to run Query 1. The data format readers are designed to operate in series in a state-free manner. In the first step, the data and checksum files are opened and read, the checksums are calculated are compared, and the data and checksum files are closed, completing the first task. Then, each partition file is opened and the footer, containing column metadata, is read, and the partition file is closed, completing the second task. Finally, the partition file is opened again, the column values are read, and the partition file is closed again, for a total for four file opens per partition, or 15,908 file opens.

5. SCALING CONCERNS

On a data center system architecture with local disks, one does not expect file open (or create) time to have a large effect on the overall time to job completion. Thus, Spark and the associated domain libraries implement stateless operation for resilience and elastic parallelism purposes by opening and closing the files involved in each individual data access: *file metadata operations are a scalability bottleneck on our HPC systems.* Any effort scaling Spark up and out on an HPC installation has first to address this concern.

There are several Spark configuration alternatives that affect file I/O behavior. We were first interested to determine if the larger number of cores in a HPC node allows for a degree of oversubscription (partitions per core) high enough to hide the MDS latency. We have systematically explored consolidation, speculation, varying the number of partitions and data block sizes to no avail.

In the time honed HPC tradition, one solution is to throw bigger and better hardware at the problem. The first aspect is to exploit the higher core concurrency present in HPC systems. As the previous Section shows, increasing[1] the number of cores per node does improve performance, but not enough to mitigate the effects of the file system.

For the Lustre installations evaluated, metadata performance is determined by the MDS hardware configuration. Although Cori contains multiple MDSes, the current Lustre 2.6 version does not exploit them well[2] and performance for the Spark workload is identical to that of a single MDS. When comparing Cori with Edison, the former contains newer hardware and exhibits lower metadata access latency (median 270μs on Cori vs 338μs on Edison), still when using the full node (32 and 24 cores) both are at best 50% slower than a eight core workstation. Enabling multiple MDSes will improve scalability but not the latency of an operation [10],

[1]Cori Phase II will contain Intel Xeon Phi nodes with up to 256 cores per node. This will become available circa Oct 2016 to early users.

[2]Supports a restricted set of operations that are not frequent in Spark.

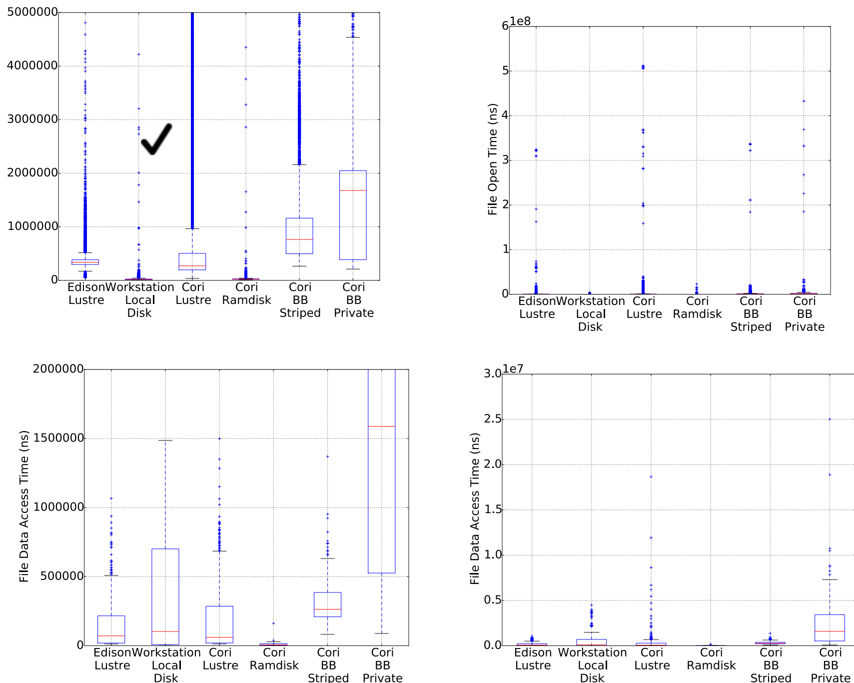


Figure 7: *Distribution of file I/O on the Lustre filesystem vs. a workstation with ext4 local disk, during the execution of Big Data. Left, median file open time is 24× higher on Lustre. Second, range of file open time, ≈ 14,000× larger on Lustre. Third, median of file read time for all BigData reads - latency similar between workstation and Lustre. Right, range of file open time - Lustre exhibits much larger variability than workstation.*

thus over-provisioning the Lustre metadata service is unlikely to provide satisfactory per node performance.

A third hardware solution is provided by the `BurstBuffer` I/O subsystem installed in Cori. This large NVRAM array situated close to the CPU is designed to improve throughput for small I/O operations and for pre-staging of data. The question still remains if it is well suited for the access patterns performed by Spark.

Besides hardware, software techniques can alleviate some of the metadata performance bottlenecks. The first and most obvious solution is to use a memory mapped file system (e.g. `/dev/shm`) as the secondary storage target for Spark. Subject to physical memory constraints, this will eliminate a large fraction of the traffic to the back-end storage system. In the rest of this paper, we will refer to this configuration as `ramdisk`. Note that this is a user level technique and there are several limitations: 1) the job crashes when memory is exhausted; and 2) since data is not written to disk it does not provide any resilience and persistence guarantees.

HPC applications run in-memory so it may seem that `ramdisk` provides a solution. For medium to large problems and long running iterative algorithms Spark will fail during execution when using `ramdisk`, due to lax garbage collection in the block and shuffle managers. To accommodate large problems we evaluate a configuration where a local file system is mounted and backed by a Lustre file, referred to as `lustremount`. This requires administrative privilege on the systems and due to operational concerns we were initially granted access to only one node. Based on the results of this study, this capability was added to Shifter [18], which is NERSC developed software that enables Docker containers to be run on shared HPC systems.

To understand scaling with large problems we develop a software caching layer for the file system metadata, described in Section 5.2. In the rest of this paper we refer to this configuration as `filepool`. This is a user level approach orthogonal to the solutions that mount a local file system. Since data is stored on Lustre, `filepool` provides resilience and persistence guarantees.

5.1 I/O Scaling in Spark

I/O overhead occurs due to metadata operations, as well as proper data access read/write operations. All these operations occur in both the application level I/O, as well as inside Spark for memory constrained problems or during the shuffle stage.

In Section 4 we have illustrated the impact of `fopen` metadata operations on the performance of *BigData Benchmark*.. There, the benchmark performed during the application input stage a number of open operations linear in the number of partitions $O(partitions)$. Big Data Benchmark did not involve a large amount of shuffle data.

Because Spark allows partitions to be cached in memory, slow reading of the initial data is not necessarily problematic, particularly in an interactive session in which multiple queries are being performed against the same data. Assuming that the working set fits in memory, disk access for input data can be avoided except for the first query. In this case, the `BurstBuffer` can be also used for data pre-staging.

In Figure 8 we show the scalability of the GroupBy benchmark up to 16 nodes (384 cores) on Edison for a weak scaling

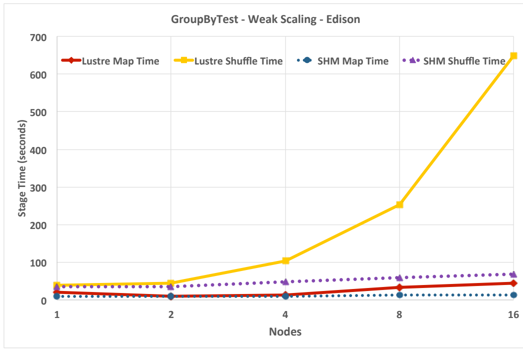


Figure 8: *Time for the map and reduce phases of GroupBy on Edison for Lustre and* `ramdisk` *as we use additional nodes to process larger datasets (weak scaling).*

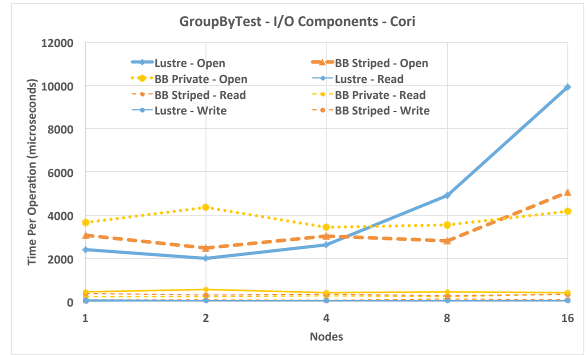


Figure 9: *Average time for a* `open`*,* `read`*, and* `write` *operation performed during the GroupBy execution with weak scaling on Cori.*

experiment where the problem is chosen small enough to fit entirely in memory.

GroupBy measures worst-case shuffle performance: a wide shuffle in which every partition must exchange data with every other partition. The benchmark generates key-value pairs locally within each partition and then performs a shuffle to consolidate the values for each key. The shuffle process has two parts: in the first (map) part, each node sorts the data by key and writes the data for each partition to a partition-specific file. This is the *local* task prior to the stage boundary in Figure 1. In the second (reduce) part, each node reads locally-available data from the locally-written shuffle files and issues network requests for non-local data. This is the *global* task after the stage boundary in Figure 1.

When running entirely in memory (`ramdisk`) performance scales with nodes, while scalability is poor when using Lustre. As illustrated, the Map phase scales on Lustre, while the Shuffle phase does not. For reference, on the workstation, mean task duration is 1,618 ms for `ramdisk` and 1,636 ms for local disk. On the Edison node, mean task duration was 1,540 ms for `ramdisk` and 3,228 ms for Lustre.

We instrumented Spark's Shuffle Manager component to track file I/O operations. During the write phase of the shuffle, a shuffle file is created for each partition, and each shuffle file is written to as many times as there are partitions. An index file is also written, which contains a map from keys to a shuffle file and offset. During the read phase, for each local partition to read and each remote request received, the index file is opened, data is read to locate the appropriate shuffle data file, which is then opened, read, and closed. The number of file open operations during the shuffle is quadratic in the number of partitions $O(partitions^2)$.

To enable load balancing, the Spark documentation suggests a default number of partitions as 4x the number of cores. On 16 nodes of Edison, with a total of 384 cores, then, we have 1,536 partitions, giving us 1,536 shuffle data files, each of which is opened 1,536 times during the write phase and another 1,536 times during the read phase, resulting in 4,718,592 file open. Not only is the number of file opens is quadratic in partitions, but the cost *per* file open also grows as we add nodes, as shown in Figure 9.

As the number of file I/O operations is linear with the number of partitions/cores during the application I/O and quadratic during the shuffle stage, in the rest of paper we concentrate the evaluation on the shuffle stage.

As for each read/write operation Spark will perform a file open, the performance ratio of these operations is an indicator of scalability. Figure 10 shows the performance penalty incurred by repeatedly opening a file, performing one read of the indicated size, and closing the file, versus opening the file once, performing many reads of the indicated size, and closing the file. Using many open-read-close cycles on a workstation with a local disk is 6× slower for 1 KB reads than opening once and performing many reads, while on Edison with Lustre, many open-read-close cycles is 56× slower than opening once and reading many times. Lustre on Cori is similar, while the Burst Buffers in striped mode reduce the penalty to as low as 16×. All of the filesystems available on our HPC systems incur a substantial penalty from open-per-read.

The scalability problems caused by the large number of file opens are exacerbated by the potentially small size of each read. Many data analytics applications have a structure in which many keys are associated with a small number of values. For example in PageRank, most write operations are smaller than 1KB. This reflects the structure of the data, as most websites have few incoming links. The data is structures as key-value pairs with a site's URL as the key and a list of incoming links as the value, so most values are short.

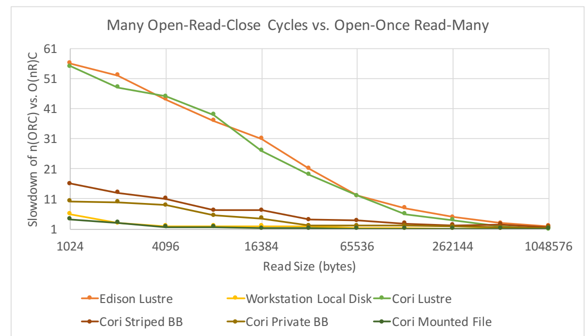


Figure 10: *Performance improvements from amortizing the cost of file opens. We compare one read per open with 100,000 reads per open.*

5.2 Improving Metadata Performance With File Pooling

For problems small enough to fit in the main memory, the `ramdisk` Spark configuration scales. However, in our exper-

iments many large problems ran out of memory at runtime, particularly iterative algorithms where the block garbage collection inside the shuffle manager is not aggressive.

In order to accommodate large problems at scale we have simply chosen to add a layer for pooling and caching open file descriptors within Spark. All tasks within an Executor (node) share a descriptor pool. We redefine `FileInputStream` and `FileOutputStream` to access the pool for open and close operations. Once a file is opened, subsequent close operations are ignored and the descriptor is cached in the pool. For any subsequent opens, if the descriptor is available we simply pass it to the application. To facilitate multiple readers, if a file is requested while being used by another task, we simply reopen it and insert it into the pool.

This descriptor cache is subject to capacity constraints as there are limits on the number of `Inodes` within the node OS image, as well as site-wide Lustre limits on the number of files open for a given job. In the current implementation, each Executor is assigned its proportional number of entries subject to these constraints.

We evaluated a statically sized file pool using two eviction policies to solve capacity conflicts: LIFO and FIFO. For brevity we omit detailed results and note that LIFO provides best performance for the shuffle stage. As results indicate, this simple implementation enables Spark to scale.

Further refinements are certainly possible. Application I/O files can be easily distinguished from intermediate shuffle files and can be allocated from a smaller pool, using FIFO. Within the shuffle, we can tailor the eviction policy based on the shuffle manager behavior, e.g. when a block is dropped from memory the files included in its lineage are likely to be accessed together in time during recomputation.

Running out of `Inodes` aborts execution so in our implementation a task blocks when trying to open a file and the pool descriptor is filled at capacity. As this can lead to livelock, we have audited the Spark implementation and confirmed with traces that the implementation paradigm is to open a single file at a time, so livelock cannot occur.

5.3 Impact of Metadata Access Latency on Scalability

In Figure 11 we show the single node performance on Cori in all configurations. As shown, using the back-end Lustre file system is the slowest, by as much as 7× when compared to the best configuration. Both file system configurations improve performance significantly by reducing the overhead of calls to open files: `ramdisk` is up to $\approx 7.7\times$ faster and `lustremount` is $\approx 6.6\times$ faster than Lustre.

`filepool` also improves performance in all cases. It is $\approx 2.2\times$ faster than Lustre, and interestingly enough is speeds up the other two configurations. For example, for *GroupBy* where each task performs $O(partitions^2)$ file opens, adding pooling to the "local" file system (e.g. `ramdisk`+`filepool`) improves performance by $\approx 15\%$. The performance improvements are attributed to the lower number of `open` system calls. For *PageRank* and *BigData Benchmark* the improvements are a more modest 1% and 2% respectively. As it never degraded performance, this argues for running in configurations where our `filepool` implementation itself or a user level file system is interposed between Spark and any other "local" file systems used for shuffle data management.

For all configurations the performance improvements are proportional to the number of file opens during the shuffle stage: *GroupBy* is quadratic in partitions while in *PageRank* it is a function of the graph structure.

In Figure 12 we show the scalability of *GroupBy* up to eight nodes (256 cores). We present the average task time and within it, distinguish between time spent in serialization (Serialization), disk access together with network access (Fetch) and application level computation (App). `ramdisk` is fastest, up to 6× when compared to Lustre. `filepool` is slower than `ramdisk`, but still significantly faster than Lustre, up to 4×. The performance differences between `ramdisk` and `filepool` increase with the scale: while system call overhead is constant, metadata latency performance degrades. When combining `filepool` with `lustremount` we observe performance improvements ranging from 17% on one node to 2% on 16 nodes.

In Figure 13 we present scalability for *PageRank* (left) and *BigData Benchmark* (right). As mentioned, the inputs for these benchmarks are not very large and we scale up to 8 nodes. The trends for *PageRank* are similar to *GroupBy* and we observe very good performance improvements from `filepool` and `ramdisk`. The improvements from combining pooling with `ramdisk` are up to 3%. In addition, when strong scaling *PageRank* the performance of `ramdisk` improves only slightly with scale (up to 25%), while configurations that touch the file system (Lustre and `BurstBuffer`) improve by as much as 3.5×. The gains are explained by better parallelism in the read/write operations during shuffle.

The performance of *BigData Benchmark* is least affected by any of our optimizations. This is because behavior is dominated by the initial application level I/O stage, which we did not optimize. This is the case where `ramdisk` helps the least and further performance improvements can be had only by applying the file pooling optimization or `lustremount`. *BigData Benchmark* illustrates the fact that any optimizations have to address in shuffle in conjunction with the application level I/O.

When using the Yarn resource manager we could not effectively scale Spark up to more than 16 nodes on either Edison or Cori. The application runs but executors are very late in joining the job and repeatedly disappear during execution. Thus the execution while reserving the initially requested number of nodes, proceeds on far fewer. After exhausting timeout configuration parameters, we are still investigating the cause.

For larger scale experiments we had to use the Spark standalone scheduler, results presented in Figure 12 right. While Yarn runs one executor (process) per node, the Spark manager runs one executor per core. The Lustre configuration stops scaling at 512 cores. The standalone scheduler limits the the performance impact of our file pooling technique: with Yarn we provide a per node cache while with the standalone scheduler we provide a per core cache. This is reflected in the results: while with YARN `filepool` scales similarly to `ramdisk`, it now scales similarly to Lustre and we observe speedup only as high as 30%. Note that `filepool` can be reimplemented for the standalone scheduler, in which case we expect it to behave again like `ramdisk`.

As illustrated in Figure 14 we successfully (weak) scaled `ramdisk` up to 10,240 cores. Lustre does not scale past 20 nodes, where we start observing failures and job timeouts. When running on the `BurstBuffer` we observe scalability up 80 nodes (2,560 cores), after which jobs abort. Note that

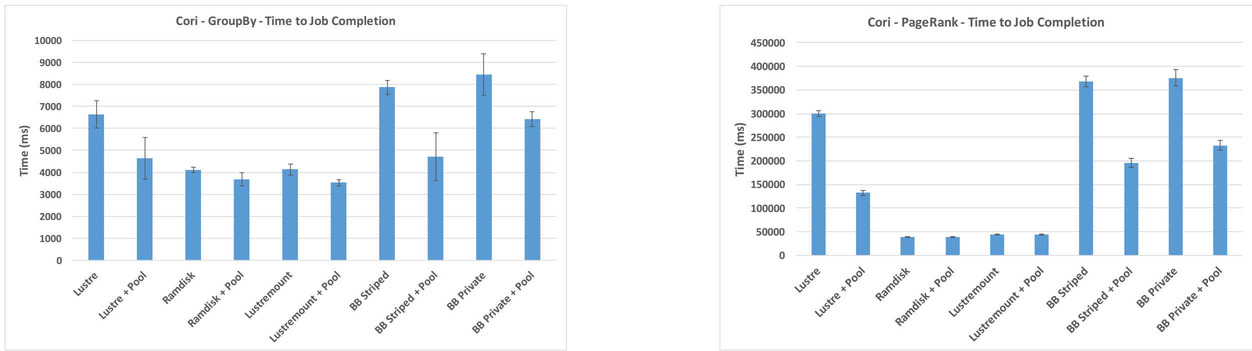


Figure 11: *GroupBy and PageRank performance on a single node of Cori.*

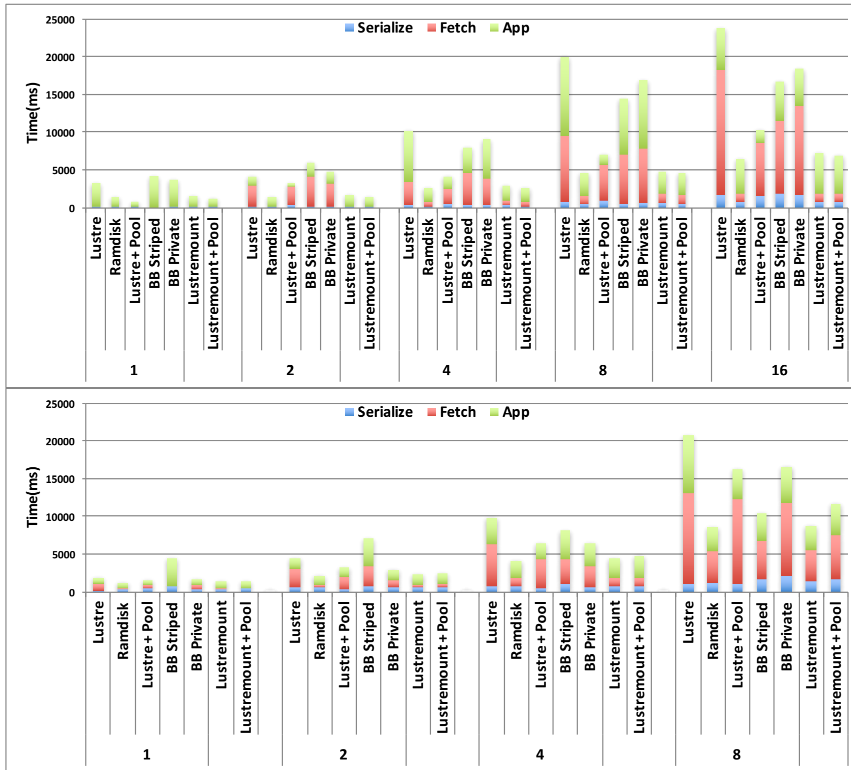


Figure 12: *GroupBy weak scaling on Cori up to 8 nodes (256 cores). Top: with YARN. Bottom: with the Spark standalone scheduler.*

`BurstBuffer` performance is highly variable at scale and we report the best performance observed across all experiments.

Figure 15 compares Lustre, `ramdisk` and `lustremount`. To use `lustremount` on more than one node, we run Spark inside a Shifter user-defined image. With Shifter, each node mounts a single image containing JVM and Spark installations in read-only mode and a per-node read/write loopback file system. Because the JVM and Spark are stored on a file-backed filesystem in Shifter, file opens required to load shared libraries, Java class files, and Spark configuration files are also offloaded from the metadata server, improving performance over configurations where Spark is installed on the Lustre filesystem. Identically configured GroupBy benchmarks running on `ramdisk` with Spark running in Shifter is up to 16% faster than than with Spark itself installed on Lustre. In addition, since the mount is private to a single node, the kernel buffer cache and directory entry cache can safely cache metadata blocks and directory entries. This can significantly reduce the number of metadata operations and improves performance for small I/O operations. For the `lustremount` implementation in Shifter initializes a sparse file in the Lustre file system for each node in the Spark cluster. These files are then formatted as XFS file systems and mounted as a loop back mount during job launch. Unlike using `ramdisk`, the `lustremount` approach is not limited to the memory size of the node and it doesn't take away memory resources from the application. Using `lustremount` we can scale up to 10,240 cores, with time to completion only 13% slower than `ramdisk` at 10,240 cores.

5.4 Impact of `BurstBuffer` on Scalability

The `BurstBuffer` hardware provides two operating modes,

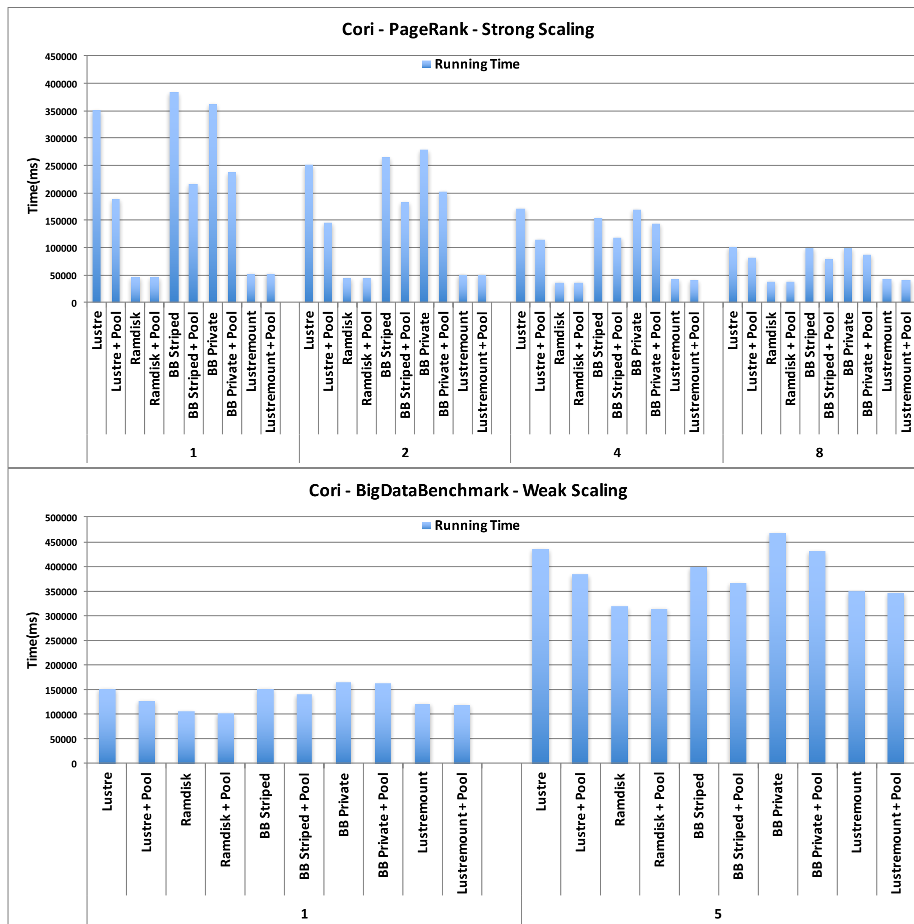


Figure 13: *PageRank and BigData Benchmark scaling on Cori, up to 8 nodes (256 cores).*

private where files are stored on a single blade (device) and striped where files are stored across multiple blades.

In Figure 7 we present the metadata latency and read operations latency for a single node run of *BigData Benchmark*. As illustrated, the mean time per operation when using the `BurstBuffer` is higher than the back-end Lustre in both striped and private mode. This is expected as interposing the `BurstBuffer` layer between processors and Lustre can only increase latency. On the other hand the variance is reduced $5\times$ compared to Lustre. When comparing striped mode with the private mode for *BigData Benchmark* striped exhibits 15% lower variance than private.

Higher latency per operation affects performance at small scale and Spark single node performance with `BurstBuffer` is slightly worse than going directly to Lustre. On the other hand, lower variability translates directly in better scaling as illustrated in Figures 9 and 12. Up to 40 nodes (1,280 cores) `BurstBuffer` provides performance comparable to running in memory with `ramdisk`. As expected, the configuration with lower variability (striped) exhibits better scalability than private mode. This is a direct illustration of the need to optimize for the tail latency at scale.

6. IMPROVING SHUFFLE SCALABILITY WITH BETTER BLOCK MANAGEMENT

Even when running using a good configuration available, e.g. `filepool+ramdisk`, some algorithms may not scale due to the memory management within the shuffle manager, which introduces excessive vertical data movement. The behavior of the PageRank algorithm illustrates this.

In Figure 16 left we show the evolution of the algorithm for a problem that fits entirely in main memory on one node of Edison. We plot both memory usage and the duration of an iteration over the execution. As shown, execution proceeds at a steady rate in both memory and time. On the right hand side of the figure, we plot the evolution of the algorithm when the working set does not fit in the main memory. As illustrated, each iteration becomes progressively slower and each iteration takes double the amount of its predecessor. The same behavior is observed on the workstation, albeit less severe.

After investigation using the instrumentation framework already developed, we observed that during constrained execution the amount of data read from disk grows at a rate two orders of magnitude higher than during unconstrained execution. After further investigation, we attributed the root cause of the problem to the shuffle block manager. Whenever running out of memory, the block manager evicts the least recently used block. The first subsequent access to the evicted block triggers recomputation, which evicts another

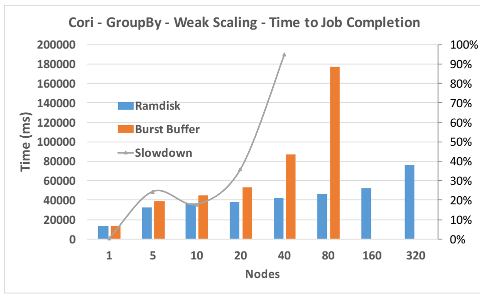


Figure 14: *GroupBy at large scale on Cori, up to 320 nodes (10,240 cores). Standalone scheduler. Series "Slowdown" shows the slowdown of* `BurstBuffer` *against* `ramdisk`*, plotted using the secondary right axis.*

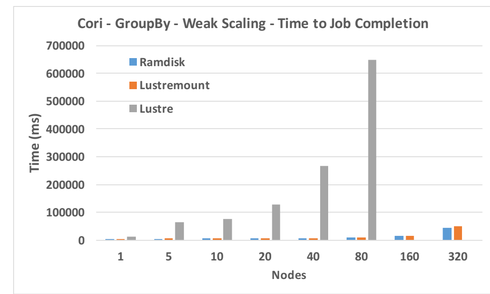
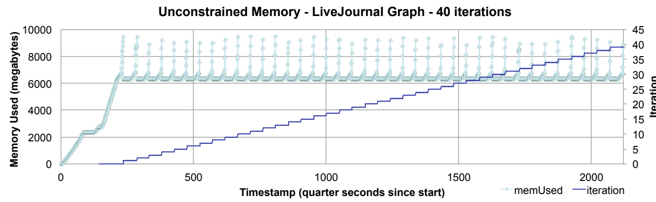


Figure 15: *GroupBy at large scale on Cori, up to 320 nodes (10,240 cores). Standalone scheduler. Lustre,* `ramdisk`*, and* `lustremount`*.*



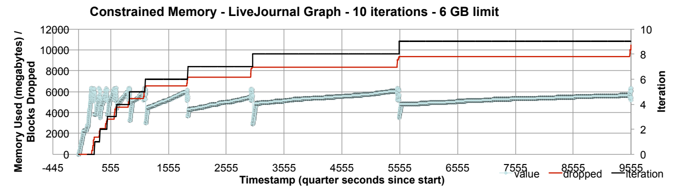
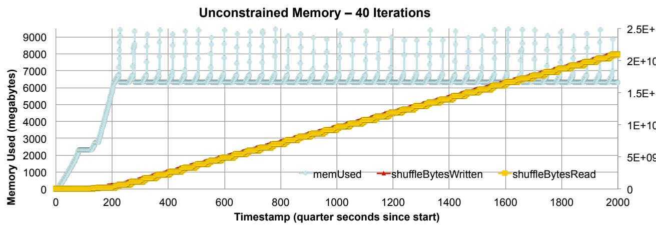


Figure 16: *PageRank performance on a single node of Edison. The amount of memory used during execution is plotted against the right hand side axis. The time taken by each iteration is plotted against the left hand side axis. Execution under constrained memory resources slows down with the number of iterations.*



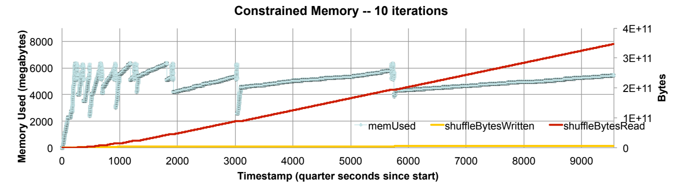


Figure 17: *PageRank IO behavior on a single node of Edison. The amount of memory used during execution is plotted against the right hand side axis. The amount of bytes read and written from disk is plotted against the left hand side axis. While memory usage stays constant, the amount of bytes read explodes under constrained memory resources.*

block needed for the partial solution which in turn triggers recomputation and eviction of blocks needed. This results in orders of magnitude increases in vertical data movement, as illustrated in Figure 17.

This behavior affects the scaling of iterative algorithms on all systems and it should be fixed. In the data center it is less pronounced as local disks are better at latency. As shown, it is very pronounced on our HPC systems. One lesson here is that because storage behaves differently, in particular for small requests, there exists incentive to specifically tune the shuffle block manager for HPC.

For the PageRank algorithm we have actually an algorithmic fix which involves marking as persistent the intermediate result RDDs from each iteration. This causes Spark to write them to the back-end storage. Upon eviction, a persistent block is read from storage instead of being recomputed. Figure 18 shows the performance of the fixed PageRank algorithm and we observe performance improvements as high as 11×. Note that all the performance findings in this paper are reported on this fixed algorithm. The original GraphX implementation does not scale beyond a single node on our systems.

There are two possible generic solutions to this problem. First, we could implement a system which tracks how often shuffle data must be reread from disk and automatically persist partitions that depend on that data when a threshold is exceeded. Second, we could track the cost of recomputing and rereading the lineage of an RDD and, rather than evicting on a least-recently-used basis, instead evict the block which will have the lowest recompute or reread cost.

Note that we were initially interested in evaluating the `spark-perf` machine learning benchmark suite [3] for this study. Due to this problem with iterative algorithms, we postponed the evaluation to the time when we can consider the aforementioned fix in the shuffle manager.

7. DISCUSSION

Metadata latency and its relative lack of scalability is a problem common to other [4, 8] parallel file systems used in HPC installations. The shuffle stage is at worst quadratic

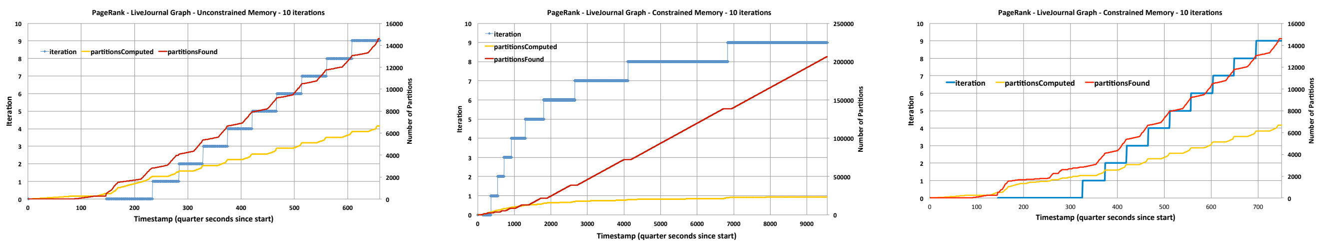


Figure 18: *Number of partitions read during the shuffle stage for PageRank˙ Left: execution with unconstrained memory. Right: when memory is constrained the number of partitions read from disk is one order of magnitude larger. Right: persisting intermediate results fixes the performance problems and we see a reduction by a order of magnitude in partitions read from disk.*

with cores in file open operations, thus metadata latency can dominate Spark performance. We believe our findings to be of interest to more than Cray with Lustre HPC users and operators. While Spark requires file I/O only for the shuffle phase, Hadoop requires file I/O for both map and reduce phases and also suffers from poor performance when run without local storage [26]. Our techniques may therefore also be applicable to Hadoop on HPC systems.

The hardware roadmap points towards improved performance and scalability. Better MDS hardware improves baseline performance (per operation latency), as illustrated by the differences between Edison and Cori. Multiple MDSes will improve scalability. The current usage of `BurstBuffer` I/O acceleration on Cori, while it degrades baseline node performance, it improves scalability up to thousands of cores. Better performance from it can be expected shortly, as the next stage on the Cori software roadmap provides a caching mode for `BurstBuffer` which may alleviate some of the current latency problems. It may be the case that the `BurstBuffer` is too far from the main memory, or that it is shared by too many nodes for scales beyond $O(10^3)$. The HPC node hardware evolution points towards large NVRAM deployed inside the nodes, which should provide scalability with no capacity constraints.

As our evaluation has shown, software approaches can definitely improve performance and scalability. Besides ours, there are several other efforts with direct bearing. Deploying Spark on Tachyon [19] with support for hierarchical storage will eliminate metadata operations. In fact, we have considered this option ourselves but at the time of the writing the current release of Tachyon, 0.8.2, does not fully support hierarchical storage (missing append). We expect its performance to fall in between that of our configuration with a local file system backed by Lustre and `ramdisk+filepool`. Note also that our findings in Section 6 about the necessity of improving block management during the shuffle stage for iterative algorithms are directly applicable to Tachyon.

The Lustre roadmap also contains a shift to object based storage with local metadata. Meanwhile, developers [13, 26] have already started writing and tuning HDFS emulators for Lustre. The initial results are not encouraging and Lustre is faster than the HDFS emulator. We believe that the `lustremount` is the proper configuration for scalability.

The performance improvements due to `filepool` when using “local” file systems surprised us. This may come from the different kernel on the Cray compute nodes, or it may be a common trait when running in data center settings. As HPC workloads are not system call intensive, the compute node kernels such as Cray CNL may not be fully optimized for them. Running commercial data analytics workloads on HPC hardware may force the community to revisit this decision. It is definitely worth investigating system calls overhead and plugging in user level services (e.g. file systems) on commercial clouds.

Luu et al [21] discuss the performance of HPC applications based on six years of logs obtained from three supercomputing centers, including on Edison. Their evaluation indicates that there is commonality with the Spark behavior: HPC applications tend to spend 40% of their I/O time in metadata operations than in data access and they tend to use small data blocks. The magnitude of these operations in data analytics workloads should provide even more incentive to system developers to mitigate this overhead.

We are interested in extending this study with a comparison with Amazon EC2 to gain more quantitative insights into the performance differences between systems with node attached storage and network attached storage. Without the optimizations suggested in this paper, the comparison would have favored data center architectures: low disk latency provides better node performance and masks the deficiencies in support for iterative algorithms. With our optimizations(`filepool+lustremount`), single node HPC performance becomes comparable and we can set to answer the question of the influence of system design and software configuration on scalability. We believe that we may have reached close to the point where horizontal data movement dominates in the HPC installations as well. Such a comparison can guide both system and software designers whether throughput optimizations in large installations need to be supplemented with latency optimization in order to support data analytics frameworks.

8. RELATED WORK

Optimizing data movement in Map-Reduce frameworks has been the subject of numerous recent studies [29, 17, 20, 11]. Hadoop introduced an interface for pluggable custom shuffle [15, 29] for system specific optimizations. InfiniBand has been the target of most studies, due to its prevalence in both data centers and HPC systems. HDFS emulation layers have been developed for parallel filesystems such as PLFS [9] and PVFS [27]. These translate HDFS calls into corresponding parallel filesystem operations, managing read-ahead buffering and the distribution (striping) of data across servers. In Spark, only input and output data is handled through the HDFS interface, while the intermediate shuffle

data is handled through the ordinary Java file API. Our work primarily optimizes intermediate shuffle data storage.

Optimizing the communication between compute nodes (horizontal data movement) has been tackled through RDMA-based mechanisms [29, 17, 20]. In these studies, optimized RDMA shows its best benefit when the data is resident in memory. Therefore, only the last stage of the transfer is carried out using accelerated hardware support. The client-server programming model is still employed to service requests because data are not guaranteed to be in memory. Performance is optimized through the use of bounded thread pool SEDA-based mechanism (to avoid overloading compute resources) [17], or through the use of one server thread per connection [29] when enough cores are available.

As we use network-attached storage, the bottleneck shifts to the vertical data movement. A recent study by Cray on its XC30 system shows that an improved inter-node communication support for horizontal movement may not yield significant performance improvement [26]. Note that this study for Hadoop also recommends using memory based file systems for temporary storage.

Optimizing vertical movement, which is one of the main motivation for the introduction of Spark, has been addressed by the file consolidation optimization [11] and by optimizations to persist objects in memory whenever possible. Our experiments have been performed with consolidation. We have analyzed the benefits of extending the optimization from per-core consolidation to per-node consolidation. As this will reduce only the number of file creates and not the number of file opens, we have decided against it.

9. CONCLUSION

We ported and evaluated Spark on Cray XC systems developed in production at a large supercomputing center. Unlike data centers, where network performance dominates, the global file system metadata overhead in `fopen` dominates in the default configuration and limits scalability to O(100) cores. Configuring Spark to use "local" file systems for the shuffle stage eliminates this problem and improves scalability to O(10,000) cores. As local file systems pose restrictions, we develop a user level file pooling layer that caches open files. This layer improves scalability in a similar manner to the local file systems. When combined with the local file system, the layer improves performance up to 15% by eliminating open system calls.

We also evaluate a configuration with SSDs attached closer to compute nodes for I/O acceleration. This degrades single node performance but improves out-of-the-box scalability from O(100) to O(1,000) cores. Since this is the first appearance of such system and its software is still evolving, it remains to be seen if orthogonal optimizations still need to be deployed with it.

Throughout our evaluation we have uncovered several problems that affect scaling on HPC systems. Fixing the YARN resource manager and improving the block management in the shuffle block manager will benefit performance.

Overall, we feel optimistic about the performance of data analytics frameworks in HPC environments. Our results are directly translatable to others, e.g. Hadoop. We scaled Spark up to O(10,000) cores and since, our NERSC colleagues have adopted the Shifter `lustremount` implementation and demonstrated runs up to 50,000 cores. Engineering work to address the problems we identified can only improve its performance. All that remains to be seen is if the initial performance and productivity advantages of Spark are enough to overcome the psychological HPC barrier of expecting bare-metal performance from any software library whatsoever.

Acknowledgements

We would like to thank Douglas M. Jacobsen at NERSC for implementing the support for file mounts inside Shifter. This work has been partially supported by the US Department of Defense and by Intel through an Intel Parallel Computing Center grant to LBNL.

10. REFERENCES

[1] Cori Phase 1. https://www.nersc.gov/users/computational-systems/cori/.

[2] National Energy Research Scientific Computing Center. https://www.nersc.gov.

[3] spark-perf benchmark. https://github.com/databricks/spark-perf.

[4] S. R. Alam, H. N. El-Harake, K. Howard, N. Stringfellow, and F. Verzelloni. Parallel i/o and the metadata wall. In *Proceedings of the sixth workshop on Parallel Data Storage*, pages 13–18. ACM.

[5] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, and M. Zaharia. Spark SQL: Relational data processing in spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 1383–1394. ACM.

[6] S. Babu and L. Co Ting Keh. Better visibility into spark execution for faster application development. In *Spark Summit*, 2015.

[7] P. J. Braam and others. *The Lustre storage architecture*.

[8] P. Carns, S. Lang, R. Ross, M. Vilayannur, J. Kunkel, and T. Ludwig. Small-file access in parallel file systems. In *IEEE International Symposium on Parallel Distributed Processing, 2009. IPDPS 2009*, pages 1–11.

[9] C. Cranor, M. Polte, and G. Gibson. HPC computation on Hadoop storage with PLFS. Technical Report CMU-PDL-12-115, Carnegie Mellon University, 2012.

[10] T. Crowe, N. Lavender, and S. Simms. Scalability testing of dne2 in lustre 2.7. In *Lustre Users Group*, 2015.

[11] A. Davidson and A. Or. Optimizing Shuffle Performance in Spark. UC Berkeley Tech. Report.

[12] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. 51(1):107–113.

[13] J. M. Gallegos, Z. Tao, and Q. Ta-Dell. Deploying hadoop on lustre storage: Lessons learned and best practices. Lustre User Group Meeting., 2015.

[14] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica. Graphx: Graph processing in a distributed dataflow framework. In *Proceedings of OSDI*, pages 599–613.

[15] A. Hadoop. Pluggable Shuffle and Pluggable Sort. https://hadoop.apache.org/docs/current/hadoop-mapreduce-client/

hadoop-mapreduce-client-core/PluggableShuffleAndPluggableSort.html.

[16] 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 the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, pages 295–308, Berkeley, CA, USA, 2011. USENIX Association.

[17] N. S. Islam, M. W. Rahman, J. Jose, R. Rajachandrasekar, H. Wang, H. Subramoni, C. Murthy, and D. K. Panda. High performance rdma-based design of hdfs over infiniband. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 35:1–35:35, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[18] D. M. Jacobsen and R. S. Canon. Contain this, unleashing docker for hpc. *Proceedings of the Cray User Group*, 2015.

[19] H. Li, A. Ghodsi, M. Zaharia, S. Shenker, and I. Stoica. Tachyon: Reliable, memory speed storage for cluster computing frameworks. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 1–15. ACM.

[20] X. Lu, M. Rahman, N. Islam, D. Shankar, and D. Panda. Accelerating spark with RDMA for big data processing: Early experiences. In *2014 IEEE 22nd Annual Symposium on High-Performance Interconnects (HOTI)*, pages 9–16.

[21] H. Luu, M. Winslett, W. Gropp, R. Ross, P. Carns, K. Harms, M. Prabhat, S. Byna, and Y. Yao. A multiplatform study of I/O behavior on petascale supercomputers. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '15, 2015.

[22] X. Meng, J. Bradley, B. Yavuz, E. Sparks, S. Venkataraman, D. Liu, J. Freeman, D. B. Tsai, M. Amde, S. Owen, D. Xin, R. Xin, M. J. Franklin, R. Zadeh, M. Zaharia, and A. Talwalkar. MLlib: Machine learning in apache spark.

[23] K. Ousterhout, R. Rasti, S. Ratnasamy, S. Shenker, B.-G. Chun, and V. ICSI. Making sense of performance in data analytics frameworks. In *Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI)(Oakland, CA*, pages 293–307.

[24] M. Poess, B. Smith, L. Kollar, and P. Larson. Tpc-ds, taking decision support benchmarking to the next level. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 582–587. ACM.

[25] R.-I. Roman, B. Nicolae, A. Costan, and G. Antoniu. Understanding spark performance in hybrid and multi-site clouds. In *6th International Workshop on Big Data Analytics: Challenges and Opportunities (BDAC-15)*, 2015.

[26] J. Sparks, H. Pritchard, and M. Dumler. The cray framework for hadoop for the cray XC30.

[27] W. Tantisiriroj, S. W. Son, S. Patil, S. J. Lang, G. Gibson, and R. B. Ross. On the duality of data-intensive file system design: reconciling hdfs and pvfs. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, page 67. ACM, 2011.

[28] UC Berkeley AmpLab. Big data benchmark.

[29] Y. Wang, X. Que, W. Yu, D. Goldenberg, and D. Sehgal. Hadoop acceleration through network levitated merge. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 57:1–57:10. ACM.

[30] T. White. *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., 1st edition, 2009.

[31] 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 the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, pages 2–2. USENIX Association.

[32] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: cluster computing with working sets. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, volume 10, page 10.