J. Parallel Distrib. Comput. 120 (2018) 369–382

Contents lists available at ScienceDirect

J. Parallel Distrib. Comput.

journal homepage: www.elsevier.com/locate/jpdc

Making sense of performance in in-memory computing frameworks for scientific data analysis: A case study of the spark system

Xuechen Zhang [a,*], Ujjwal Khanal [a], Xinghui Zhao [a], Stephen Ficklin [b]

[a] Washington State University Vancouver, Vancouver, WA 98685, USA
[b] Washington State University, Pullman, WA 99164, USA

HIGHLIGHTS

- We develop a benchmark, ArrayBench, for benchmarking scientific data analytics that process gene expression matrices using Spark and SciDB.
- We study the correlations between the performance of Spark data analytics and various OS components, e.g., memory, storage, and file systems.
- Our findings shed light on the improvement of Spark and SciDB and the future development of data-intensive data analytics using the in-memory computing frameworks.

ARTICLE INFO

Article history:
Received 28 July 2017
Received in revised form 11 October 2017
Accepted 27 October 2017
Available online 22 November 2017

Keywords:
Spark
SciDB
In-memory computing
Scientific data analytics

ABSTRACT

Over the last five years, Apache Spark has become a major software platform for in-memory data analysis. Acknowledging its widespread use, we present a comprehensive study of system characteristics of Spark targeting scientific data analytics performing large-scale matrix operations. We compare its performance to SciDB, a disk-based platform for array data analysis. A benchmark, ArrayBench, is developed to evaluate the performance of four analytics processing gene expression matrices using basic data operators of Spark and SciDB. It is applied to data from a real biological workflow whose data inputs are in matrix form. Herein, we report the findings, which shed light on the improvement of Spark and SciDB and the future development of data-intensive scientific data analytics using the in-memory computing frameworks.

1. Introduction

Data warehouses using a multidimensional view of data have become very popular in science in recent years. Two approaches are typically used for building a scientific data warehouse. One is traditional SQL parallel database management systems (e.g., MySQL [23]), while the other is NoSQL database systems (e.g., SciDB [32], and software which uses MapReduce framework in Hadoop MapReduce [15], and Apache Spark [45,46]). Among the existing systems, Spark is the only platform designed for in-memory data analysis. It becomes increasingly adopted as other managed, disk-based systems are proving increasingly inadequate in meeting the demands of data analytics.

There has been much research devoted to understanding the causes of specific performance bottlenecks that are correlated to the CPU [5], storage [29], and network systems [3] in various data analytics platforms. Most recently, Ousterhout et al. [25] found that CPU is the major bottleneck of the end-to-end performance of Spark analytics. Much of this work studied the performance of transactional workloads (e.g., TPC-DS [38]). There has been little effort directed towards understanding Spark's performance in the context of scientific data analytics, especially compared to traditional disk-based software systems (e.g., SciDB [32]), which are heavily optimized for processing scientific data in the form of arrays. Therefore, we want to answer three key questions. (1) Do previously reported results hold true for matrix-based scientific applications, especially when matrix size is significantly larger than memory size and out-of-core computation is required? (2) For these same matrix-based applications, do in-memory systems outperform modern disk-based distributed systems? (3) Given the unique characteristics of in-memory data analytics, how can developers tune the memory and storage systems with high resource efficiency for matrix analytics?

For this purpose, we quantitatively analyze performance of matrix-based scientific data analytics developed using both Spark and SciDB. Spark was designed for coarse-grained in-memory computation using a key–value data model. SciDB was designed for processing scientific matrices using an array data model. The

* Corresponding author.
E-mail addresses: xuechen.zhang@wsu.edu (X. Zhang), ujjwal.khanal@wsu.edu (U. Khanal), x.zhao@wsu.edu (X. Zhao), stephen.ficklin@wsu.edu (S. Ficklin).

https://doi.org/10.1016/j.jpdc.2017.10.016
0743-7315/

analytics discussed in the paper are used in real scientific workflows from biological research. Moreover, we study the impact of multiple OS components on the performance of the analytics. Specifically, this paper makes two contributions towards a more comprehensive understanding of the system characteristics of in-memory data analytics platforms.

The first contribution is a benchmark, named ArrayBench, used for benchmarking both Spark and SciDB. Currently, it implements 4 data analytics, *matrix-scan* which scans an input dataset for filtering and thresholding; *matrix-join* which compares multiple versions of matrices, each generated at distant time steps or records various physical attributes; *degree-distribution* which analyzes graph attributes; and *community-detection* which is designed for identification of highly connected sub clusters in biological graphs. More details of the analytics can be found in Section 2.2.

The second contribution of the paper is the application of ArrayBench to comprehensively study the in-memory data analytics platforms (e.g., Spark), compared to the disk-oriented platforms (e.g., SciDB). In addition, ArrayBench is also used to study how performance of scientific data analytics is correlated to various OS components, e.g., virtual memory, page cache, and distributed file systems, among others. Our key findings from the study are summarized as follows:

- Spark does not consistently outperform SciDB when in-situ execution can be applied. Caching intermediate results may increase memory pressure, resulting in poor shuffle performance 3.1.
- Spark with cache can help achieve a super-linear speedup because computational results stored in cached resilient distributed datasets (RDDs) can be reused. Without caching, the speedup of the analytics with both SciDB and Spark is far less than a linear speedup 3.2.
- There is a strong correlation between the configurations (e.g. block size) of distributed file systems and the execution time of analytics 5.1, whereas the impact of local file systems is only noticeable for *community-detection*.
- Memory compression may have a negative impact on the performance of the analytics. Among the compression algorithms, *lzf* works best for the scientific data used for this study 4.3.
- Caching data in process memory space as Java objects can be 4X more efficient than recomputing those data. Choosing RDDs to cache is critical to performance 4.4.
- The effectiveness of SSDs is strongly correlated to the size of Java objects (e.g. RDD) serialized on storage devices. Using SSDs can reduce the execution time but by only 8% for graph analytics 5.3.

The remainder of this paper is organized as follows. The methodologies used in our study are described in Section 2. We present the overall performance of ArrayBench and its scalability with Spark, compared to that with SciDB in Section 3. To understand the interaction between Spark and the operating systems, we study how the performance of ArrayBench is affected by the system components of Linux, such as virtual memory management in Section 4, and file and block management on storage devices in Section 5. We summarize the key observations and discuss their implications in Section 6. Related work is presented in Section 7 and the final section concludes the paper.

2. Methodology

This section outlines the reason for selecting Spark [45] and SciDB [32] as target open-source software platforms for scientific data analysis. Then we describe the components of ArrayBench and the format of input datasets used in a biological scientific workflow from the field of systems genetics. Finally, we describe our experimental setup and the approaches for execution time analysis.

Table 1
A summary of major differences between Spark and SciDB. Note: The MapReduce programming model in Spark is implemented based on RDD transformation.

	Spark	SciDB
Data model	Key–value pairs	Arrays
Programming model	MapReduce	Matrix computation
Language	Java/Scala/Python	C++
Interface	Spark library	AQL/AFL
Result caching	User-defined	System-assisted
I/O optimization	In-memory	Disk-oriented
Data format on disk	Objects	Files

2.1. Selected software platforms for scientific data analysis

We select Spark [45] and SciDB [32] because they represent two of the most popular distributed systems for large-scale data processing. Spark was selected because it is designed for in-memory data analysis using a key–value data model. When the dataset size is larger than memory size, intermediate results can be serialized as Java objects and stored on secondary storage devices (e.g., disks and SSDs) or recomputed when needed at runtime. Therefore, minimization of I/O and compute overhead for object serialization and deserialization can be critical to the performance of Spark. SciDB was selected because it is implemented using an array data model. For I/O optimization, arrays are chunked and striped over multiple disks. Therefore, developers need to minimize the overhead of accessing files on disks. Major differences between Spark and SciDB are summarized in Table 1.

2.2. Driving scientific data analytics

Here we study the characteristics of Spark and SciDB in the context of large-scale scientific data analysis. For this purpose, we developed a benchmark called ArrayBench. It is composed of 4 analytics, which cover a wide range of characteristics from unary to binary operations, from array computation with regular memory access to graph computation with irregular memory access, and simple analytics to complex analytics comprising multiple execution stages. These analytics are implemented using basic data operators, which are realized using the APIs of Spark/GraphX or the AQL/AFL query interface of SciDB. The implementation of ArrayBench follows the Jim Gray's 4 principles of domain-specific benchmarks, including *relevant*, *portable*, *scalable*, and *simple* [14]. The implementation of the analytics are described below.

- **Matrix-scan**: this analytic selects a subset of data from an input file. For example, for a gene expression matrix [35] (as described in the following section), *matrix-scan* can be executed to find a subset of genes that match a criteria to reduce outliers in the data. We use the *filter*() functions provided by both Spark and SciDB in the implementation of the analytic.
- **Matrix-join**: this is a binary array analytic, which can be used to study similarity of two arrays. A use case of the analytic is studying the similarity of two gene expression matrices. For implementation of this analytic, we use the *join*() functions provided by both Spark and SciDB.
- **Degree-distribution**: this is an analytic widely used for observing attributes of graphs whose vertices are connected by edges that indicate, in the biological context, interactions between genes. Spark natively supports graph computation using the MapReduce programming framework. We use the Scala method *graph.inDegrees* to calculate the degree distribution of nodes in a graph, which is created using its edge list. In contrast, an implementation of *degree-distribution* operation using the array data model of SciDB requires two steps of array re-dimension and degree reduction for each vertex in the graph.

Table 2
The compute and I/O characteristics of the 4 steps of the biological workflow.

	Compute-intensive	I/O-intensive
$Step_1$	No	No
$Step_2$	Yes	Yes
$Step_3$	No	Yes
$Step_4$	Yes	Yes

- **Community-detection**: this analytic is used to detect communities in a graph. For example, researchers [35] used connected-component algorithms to detect genes that are co-functional in the biological networks. Spark has built-in connected-component algorithms in its graph library GraphX. For SciDB, we implement *community-detection* using adjacent matrices. Specifically, we initialize all vertices as unmarked. Then for each unmarked vertex, we run a depth-first search algorithm [37] to identify all vertices discovered as part of the same component. It requires multiple steps of array reduction operations until the results converge. SparkX uses label propagation algorithm in the implementation of connected-component algorithm [28] while for SciDB components are detected by initiating multiple breadth-first searches. Assume there are N vertices and M edges in an input graph. The computational and I/O complexity per iteration are $O(N + M)$ for SciDB and $O(M)$ for Spark.

2.3. Input datasets

We use data from a real biological workflow as input. The workflow includes 4 major steps: (1) preprocessing, normalization, and outliers removal given a compendium of gene expression measurement sample files ($step_1$); (2) pair-wise correlation analysis for construction of a gene expression matrix ($step_2$); (3) significance thresholding using random matrix theory ($step_3$); and (4) analysis of biological networks ($step_4$). Table 2 shows the characteristics of I/O and computation for the programs at every step. For this study we only consider $step_3$ and $step_4$ because they are I/O-intensive and time-consuming in the workflow and conducive for distributed data processing. $step_2$ can be computation and I/O intensive but high-performance computing is sufficient to perform this step and therefore these steps were not a part of this study.

ArrayBench is designed to process two types of data files in an array format. $Type_A$ arrays are one-dimensional (N) arrays recording gene expression data measurements. These arrays are provided in a single input file referred to as the gene expression matrix. $Type_B$ arrays are two-dimensional ($N * N$) matrices recording pairwise correlation values of all gene pairs. This matrix is referred to as a gene co-expression similarity matrix. In the experiments, N is set to 52,489, which is the total number of genes in the expression matrix. The gene co-expression similarity matrix is represented in a tab-delimited text format with each pairwise correlation represented by a separate line. Each line contains "columns" of annotations specific to the comparison. An illustration of the data format of $Type_B$ arrays is presented in Fig. 1. Both $Type_A$ and $Type_B$ matrices are stored in a row-major manner in the input files.

2.4. Cluster setup

To benchmark Spark and SciDB, our experiments are run in a cluster of 12 servers. HDFS is configured for hosting input datasets, which are then loaded and distributed across all 12 servers of the cluster if the number of Spark slave servers is set to 12. With this setup, all intermediate RDDs are stored locally for shuffling. Each server is configured with Intel Core2 Duo 3.16 GHz CPU, 4 GB memory, one 1 TB disk (Seagate Barracuda 7200.12), and one 128 GB SSD (OCZ-VERTEX 3). The servers are intentionally set up with relatively small amount of memory in the experiments to study the performance of different Java serializers and the impact of heterogeneous storage devices (e.g., SSDs) for storing serialized Java objects. These servers run Ubuntu Linux of kernel-3.16.0. All nodes are interconnected through a switched Gigabit Ethernet. Our experiments use Apache Spark version 1.6.0, Scala version 2.11.7, Hadoop version 2.6.0, and SciDB 15.7.0 community version.

We carefully select the software settings of Spark and SciDB using well-documented approaches [40,41]. Then we measure performance fluctuations as we change a single parameter, such as RDD compression algorithms. By default, we use 128 MB block size for the HDFS and ext4 for the local file system. The Spark workers use disks to store shuffle files. And by default anonymous huge page(THP) [39] is enabled in the operating system.

To make fair comparisons we flush OS buffer caches to ensure that all requested data are served at storage nodes of the HDFS. For testing the performance of Spark with objects persisted in memory or on disks/SSDs, we run analytics to warm up the JVM with specified objects. We present average results of three runs for all the experiments discussed.

2.5. Execution time analysis

We use three approaches to study performance of the analytics, including software logging, source code instrumentation, and system profiling. Spark provides a web-based GUI [42] for performance debugging (e.g., looking for task stragglers) using execution times of various instances (e.g., stages, jobs, and tasks) provided by Spark. However, they cannot be efficiently used to identify a system bottleneck. For this purpose, Ganglia [21] is used to monitor CPU, memory, and network utilization on every node. Blktrace [1] is installed to study I/O characteristics (e.g., request size and sequentiality) of workloads at the block level for disk efficiency. Finally, source code instrumentation is adopted for profiling Java functions. As shown in the following sections, using these approaches effectively help us understand the correlations of analytics' performance with the characteristics of variable system components (e.g., virtual memory and file systems).

3. Does Spark consistently outperform SciDB?

Spark is designed to perform in-memory data analysis leveraging distributed memory of a cluster. Its intermediate results can be cached as RDDs by analytics in memory or persisted on disks for reuse. In contrast, SciDB does not allow programmers to explicitly cache temporary arrays in memory. Its storage manager maintains a cache for array data, which are recently accessed. If no space is available in the cache, the temporary results are flushed to disks. In this section, we will study which platform performs better given different dataset size and number of compute nodes.

3.1. Impact of the dataset size

We study the impact of the dataset size on data loading time and analytics execution time. In the experiments, 4 compute nodes are set up as a cluster to execute the analytics of ArrayBench. We increase the dataset size from 5 to 28 GB.

3.1.1. Data loading time

Before running the analytics the input data of different sizes is loaded to HDFS for Spark and the data store of SciDB, respectively.

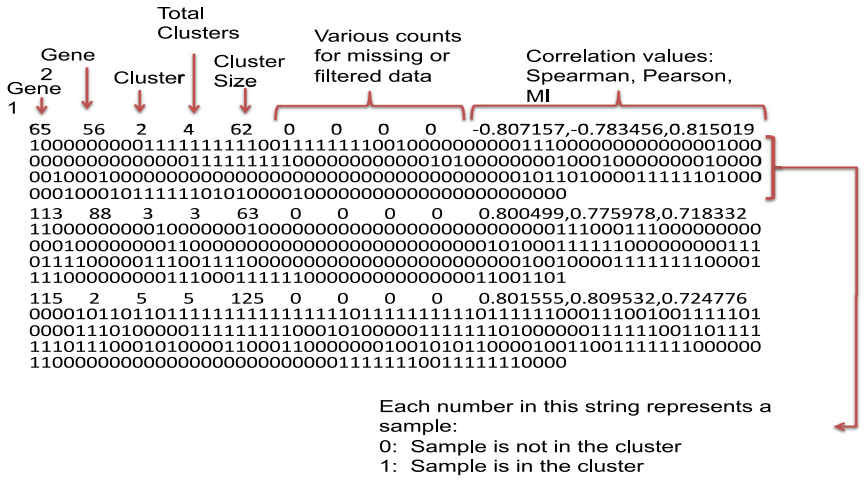

Fig. 1. Data format of input files of the $Type_B$ matrices. Each row of the matrix describes correlations of two genes. The correlations are recorded using multiple metrics, e.g., clusters of correlation genes, correlation values, and samples in the cluster. Correlation values are calculated using three methods including Spearman, Pearson, and MI [12].

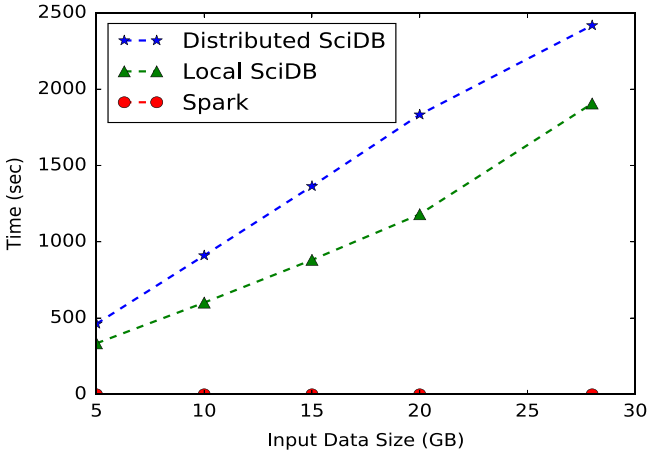

Fig. 2. Data loading time. 4 servers are used in the experiment.

We increase the dataset size from 5 to 28 GB. Two versions of SciDB were set up: one only runs as a single-thread daemon on a single compute node (*Local SciDB*), while the other one employs 4 SciDB threads, each of which runs on one compute node (*Distributed SciDB*). As shown in Fig. 2, we observed that both *Local SciDB* and *Distributed SciDB* can achieve linear scalability in data loading. However, *Local SciDB* outperforms *Distributed SciDB* by 28% on average because the latter has communication overhead for distributing data in the cluster. In contrast, Spark spent very little time on loading data because it can directly process the input data in situ without the need of converting input data into its internal data format, e.g., two-level chucks [33] used by SciDB.

3.1.2. Analytics execution time

For Spark, input data is read from HDFS if cache is not used. Otherwise, it is read from objects stored in memory or on disks if memory space is limited. The data retrieved from disks may require deserialization. For SciDB, we clear both SciDB cache and OS page cache for the runs without cache. We then run the same analytics repeatedly for three times without cleaning the cache to study the impact of the cache. Fig. 3 shows the execution time of the analytics with and without cache, respectively. We have the following observations.

First, *Spark does not consistently outperform SciDB*. For example, for *matrix-scan* they have similar execution times given different input dataset size. The execution time of *degree-distribution* with Spark can be 59X longer than SciDB if analytics do not reuse intermediate results in cache. We found the reason is because of data format mismatch. ArrayBench stores gene expression matrices in input files. SciDB can directly perform matrix reductions to calculate degree distribution of vertices. However, Spark requires an additional step to convert the original input data to pairs of vertex and edge and store them in RDDs before calling the GraphX methods for degree calculation. Consequently, we found that 96.4% of the execution time of the analytic was spent on data conversion with Spark.

Second, *caching intermediate results would increase memory pressure and result in poor shuffle performance*. Analytics of Spark may be executed in multiple steps of map-reduce operations. As an example, *community-detection* requires 6 reduce steps. For each step, data shuffle time may dominate its execution time. Spark uses page cache of operating systems to hide latency of storing shuffle files on disks. When dataset size is 5 GB, the average memory demand is 364 MB on a node, therefore, the cluster has adequate memory to serve the I/O requests of shuffle operations using page cache. It significantly reduced the shuffle time and the execution time. However, when the dataset size is 28 GB, the average memory demand is 1.4 GB on a node. We observed only 2%–5% of system memory is used as page cache. As a result, most I/O requests of data shuffle were served on the disks with high I/O latency. This explains why the execution time of Spark can be 1.45X longer than that of SciDB for *community-detection*. Also because of the poor shuffle performance its execution time of the reduce operations is increased by up to 57X as we increased the dataset size from 5 to 28 GB for the analytic.

Third, *Spark improves I/O efficiency for binary-input operations*. We use *matrix-join* as an example. Without cache, Spark uses 91% less time than SciDB. We observed that maximum CPU wait time on I/Os is 61% for SciDB and 33% for Spark during the execution of the analytic. It indicates that Spark spent less time on I/Os for two reasons. (1) It used 256 MB file block size to amortize disk seek time, leading to less small and random I/Os on disks. Fig. 4 shows the order of accessed disk addresses, or roughly the path of disk head movement at compute server 2 in a 10-second execution sample. When SciDB is used, the disk head rapidly alternates between two disk regions, each storing a matrix, which is the input to the join operation. When Spark is used, the large block size

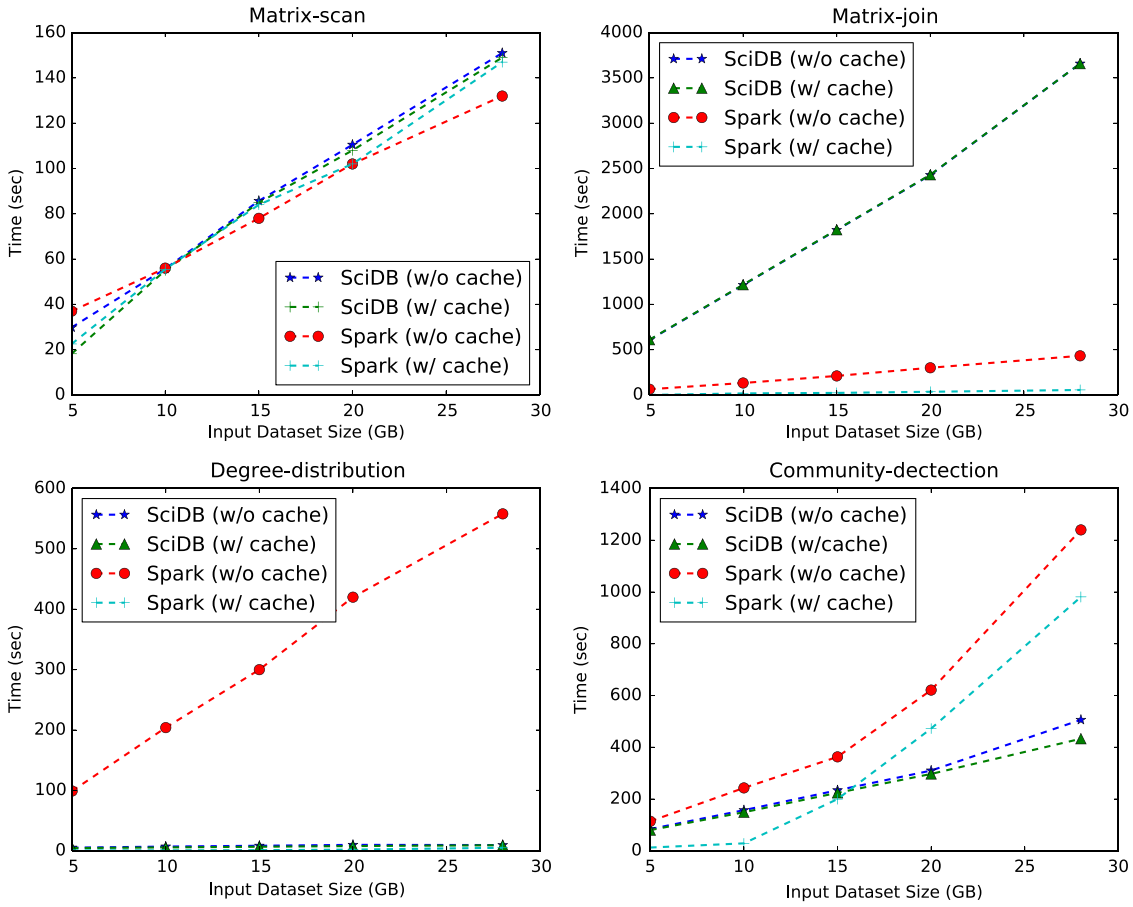

Fig. 3. The execution times of the analytics. We increase the input dataset size from 5 to 28 GB. 4 servers are used in the experiment. SciDB (w/o cache): execution time of analytics using SciDB. System cache and OS page cache are empty before execution. SciDB (w/cache): execution time of analytics using SciDB with caches being warmed up before execution. Spark (w/o cache): execution time of analytics using Spark with input data read from HDFS. Spark (w/cache): execution time of analytics reusing intermediate RDDs being cached before execution.

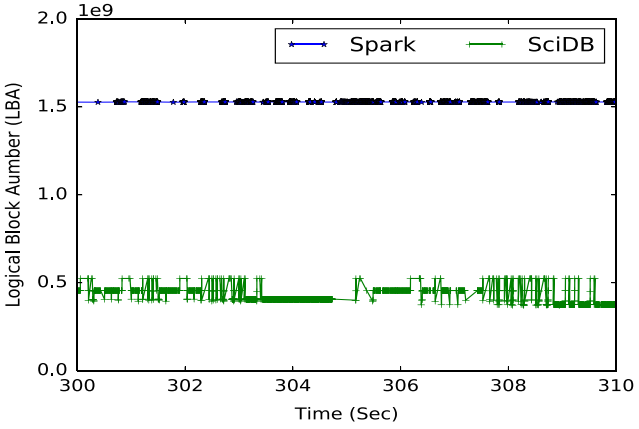

Fig. 4. Disk addresses (LBAs) of data access on the disks of compute server 2 in a sampled 10-second execution period.

helps exploit the strong locality into efficient disk access. (2) Spark transferred up to 84.7% less data than SciDB. We believe this is because the job scheduler of Spark is more aware of data location and can further explore data locality given multiple input datasets, compared to SciDB. Another interesting observation is the OS page cache does not help improve the performance of *matrix-join* for SciDB. In contrast, reusing data in Spark cache can further reduce the execution time of the analytic by 82% on average. We found this is caused by weak temporal locality of the analytic. Consequently, OS cannot effectively keep the pages in memory cache for reuse using the existing cache replacement algorithms (e.g., clock [20]) in the Linux kernels.

3.2. System scalability

We then study the scalability of Spark and SciDB by increasing the number of compute nodes from 4 to 12 with a fixed input dataset size 28 GB. We show the speedup of execution time in Fig. 5. Because Spark is not recommended to run in a single-node cluster, we consider the execution times of the analytics with 4 compute nodes as the baseline in the calculation of the speedup. We present the results of Spark and SciDB without and with cache, respectively, in Fig. 5. For the convenience of our illustration, a curve of an ideal scalability is also shown in the figure with the assumption that an analytic scales linearly and is 100% parallelizable. We have the following observations from the figure.

First, *without using cache the speedup of the analytics with both SciDB and Spark is far less than the ideal scalability*. It is because these analytics are data-intensive and the I/O bottleneck makes the analytics not scalable. For example, Fig. 6 shows the percentage of CPU time spent on I/O and computation for each analytic of ArrayBench with 12 compute nodes in the Spark cluster. On average, Spark spent 49.5% on networking and disk I/Os, with a majority of which being for reading input data from the HDFS to compute nodes. Spark created multiple jobs for *community-detection*. I/O wait time of one job of the analytic may account for up to 74.7%

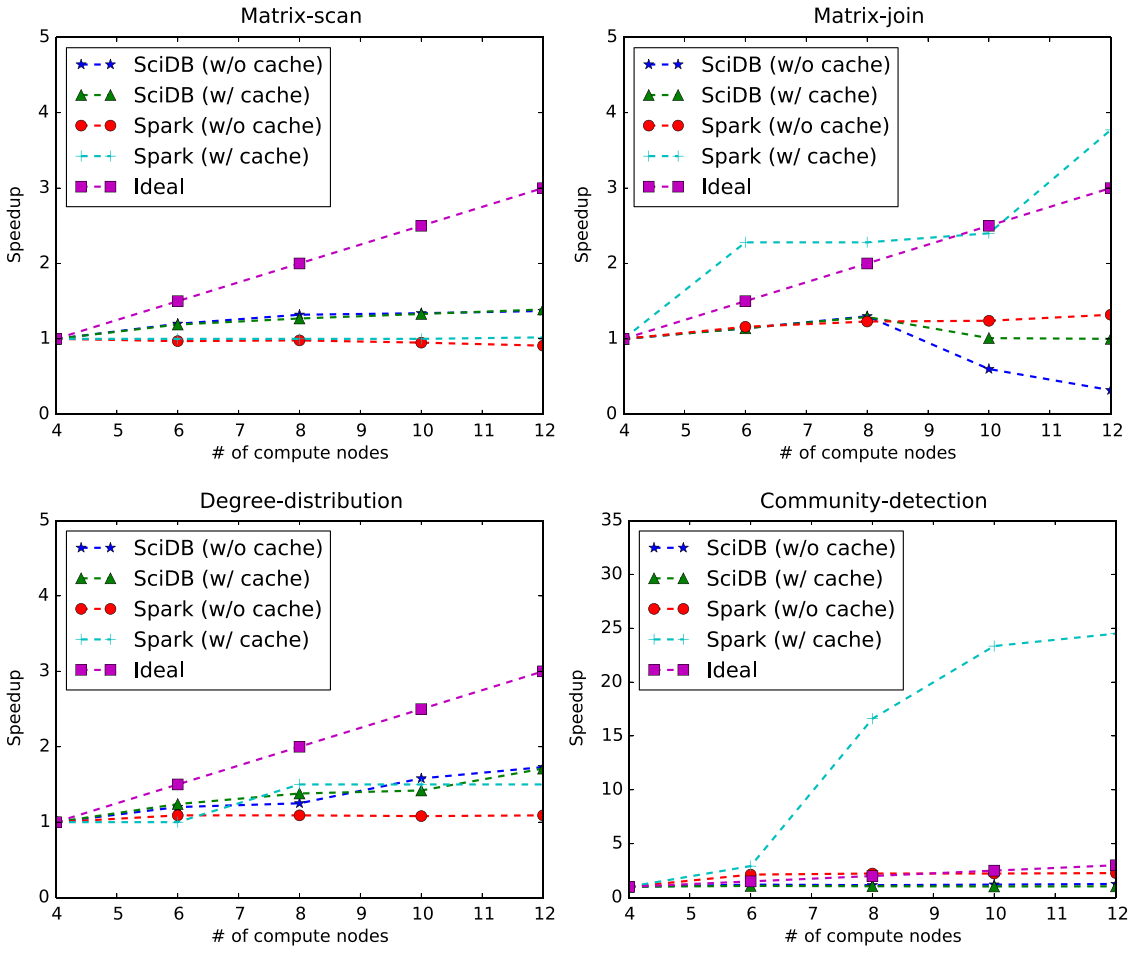

Fig. 5. The execution times of the analytics as we increase the number of compute nodes from 4 to 12. Note: *Ideal* denotes an ideal scalability.

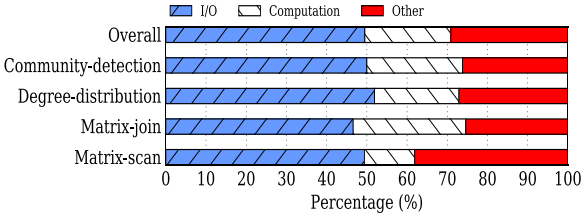

Fig. 6. A CPU time breakdown for I/O, computation, and other (e.g., idle). Note: if an analytic is composed of multiple Spark jobs, the average value is shown. 12 servers are used in the experiment.

of the CPU time. Another observation is that CPUs are not the bottleneck because 20.5% of the CPU time was, in fact, idle for the analytics.

Second, *Spark with cache can help achieve a super-linear speedup because of reusing computation results in cached RDDs*. For example, the execution of *community-detection* is composed of 191 stages. Spark automatically selected and cached 9 RDDs across the stages in the memory space of the Java processes. By reusing the computation results in the RDDs, Spark can reduce the execution time of *community-detection* by up to 13.7X, compared to that without using cache. However, for the analytics (e.g., *matrix-scan*) which have only one reduce stage, Spark cache did not have the opportunities of reusing computation results. For such analytics, Spark's scalability is even worse than SciDB for the following two reasons. (1) Each cached RDD is internally stored as multiple small RDD partitions on disks. (2) They are stored as the Java objects, which consume 8.3% more disks space than the raw data used by SciDB.

Third, *SciDB is not as scalable as Spark for the majority of the analytics in ArrayBench.* As an example, for *matrix-join*, when we increased the number of the compute nodes from 8 to 10, its speedup is reduced from 1.3 to 0.6 without cache and from 1.29 to 1.01 with cache. SciDB cannot effectively use page cache when programs have weak temporal locality. This means the data cached in memory can be replaced before being accessed again. Spark exploits data locality by managing the memory space itself. Furthermore, SciDB suffered from lower I/O efficiency because of two reasons. (1) Array chunk size of SciDB can be much smaller than file block size of Spark. (2) It stores intermediate results as temporary arrays on the same disks that host input and output files, leading to more random access. In contrast, Spark usually uses dedicated distributed file systems for management of these files. The only analytic that achieved better scalability with SciDB is *matrix-scan*. They both sequentially access data on the disks and none of its intermediate results can be effectively cached. For the analytic, the performance advantage of SciDB is largely because of its implementation using C++, which is more efficient than the Java language.

Summary: Spark does not consistently outperform SciDB in the situation that the latter can process matrices in situ without data transformation. Without caching, Spark may perform worse than SciDB. However, with reusing the cached RDDs, Spark can outperform SciDB for most of the analytics. I/O efficiency of shuffle operations can be compromised because of memory competition between object cache of Spark and page cache of an OS.

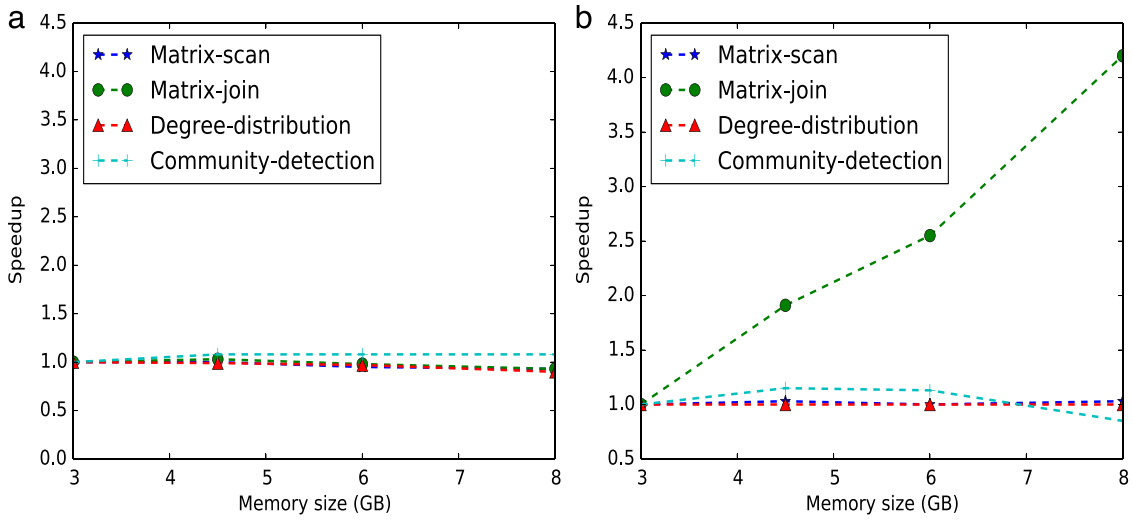

Fig. 7. Execution speedup of the analytics as the total amount of memory is increased from 3 to 8 GB. Input data are read from HDFS (a) and cached RDDs (b). 12 servers are used in the experiment.

4. How does memory management affect results?

For in-memory execution, Spark analytics have high memory demands for both computation and storage. For computation, Spark needs to store shuffle files in page cache to hide disk I/O latency. For storage, RDDs are stored in memory for reuse between operations. Therefore, memory management can affect the performance of Spark analytics in many ways. In this section, we will study how existing approaches (e.g., transparent huge pages and memory compression) for virtual memory management of Linux affect the execution of in-memory data analytics.

4.1. Does increasing system memory improve performance?

We first evaluate the analytics of ArrayBench with the total amount of memory being increased from 3 to 8 GB. We show the execution speedup of the analytics in Fig. 7. In the experiments we consider the execution time with 3 GB memory as a baseline in the calculation of the speedup. Fig. 7(a) shows the results when input data of the analytics are read from HDFS. No speedup is observed as memory size is increased. This is because disk I/O for accessing the HDFS is the bottleneck. For *community-detection*, the ratio of CPU time on I/O wait can be up to 53.7%. Fig. 7(b) shows the results when input data are read using cached RDDs if they are available in memory. In this case, the performance of *matrix-scan*, *degree-distribution*, and *community-detection* is still not sensitive to memory size because the size of their RDDs are significantly larger than the memory size. Spark only cached part of the RDDs which were fit in the memory. The rest of the RDDs were still created from the input files in the HDFS, causing the disk bottleneck. Interestingly, *matrix-join* achieved a linear speedup because of improved disk efficiency with larger memory. The analytic has two input files. Therefore, two RDDs are created to store data from each file. When Spark read data from the two cached RDDs, they were concurrently accessed on disks as only part of the RDDs can be found in memory. Therefore, larger memory size for caching the RDDs can significantly reduce the disk seeking time for the analytic.

4.2. Does memory page size affect performance?

In-memory data analytics platforms need high-speed memory paging systems. And memory page size is a critical factor because it may impact performance of Translation Lookaside Buffer (TLB) lookups. Linux transparent huge pages (THP) [39] is one of the optimizations implemented in an OS kernel to improve the performance of paging. We study the impact of THP in the context of scientific data analysis with Spark. THP is enabled by default supporting anonymous page size of 2 MB. In the experiments we disabled THP and only used virtual memory of 4 kB page size to serve the analytics. The results with and without THP are shown in Fig. 8.

Contrary to our expectation, THP does not improve the performance of the analytics. In fact, the execution time of *community-detection* without THP is 31% shorter on average. We found that Spark workloads performed poorly with THP because they can have sparse and random memory access patterns during shuffling. As a result, the operating system may need to do memory compaction to move page around to defragment the free space. And the overhead of the compaction operation can dramatically offset the benefit of THP.

4.3. Does memory compression help?

Data compression can reduce the amount of shuffle data and the size of RDDs, therefore, reducing memory pressure during the execution of Spark analytics. Both Spark and Linux kernels support data compression. Spark implemented three compression algorithms, lz4, lzf, and snappy [34]. Linux kernel uses zram [48] as a special swap device. All pages to be swapped out are compressed in memory before writing to a swap partition on disks. In the experiments 1 MB memory is allocated to the zram device. We compare the performance of the Spark compression to Linux kernel compression. The results are shown in Fig. 9.

We have following observations from the figure. (1) For Spark analytics with scientific data, lzf performs better than lz4 and snappy. (2) Zram achieves similar performance to other compression approaches implemented in the application level. (3) Running compression algorithms incurs up to 16% overhead, including both compression and decompression time, for analytics frequently accessing cached RDDs, e.g., *community-detection*, resulting in a longer execution time.

4.4. Which objects to cache?

Spark provides the *persist()* and *cache()* methods to cache a dataset as a RDD object in memory for reuse across operations. Caching is the key to the performance of analytics. We study impact

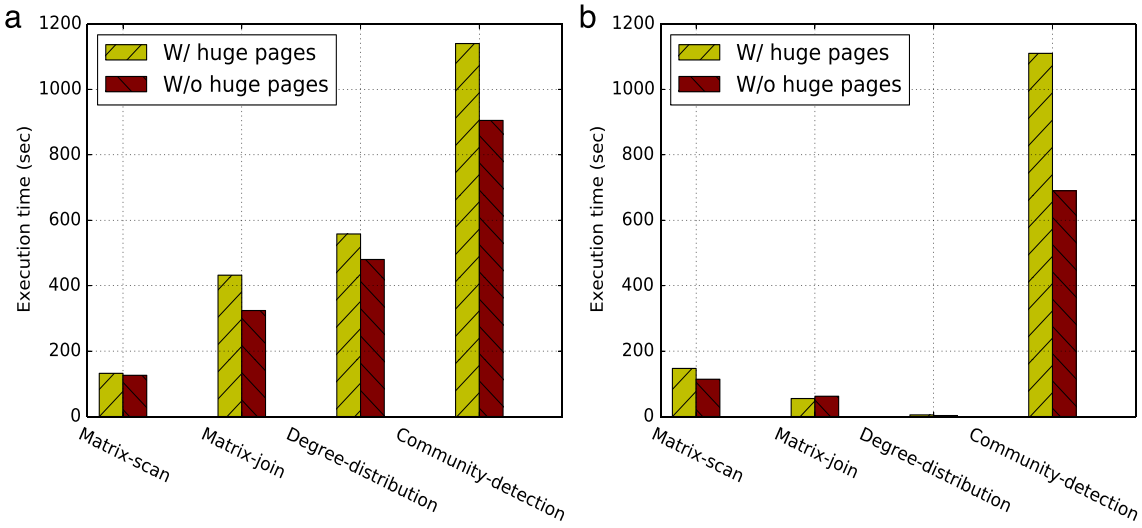

Fig. 8. Execution times of the analytics without and with anonymous huge pages. Input datasets are read from HDFS (a) and cached RDDs (b). 12 servers are used in the experiment.

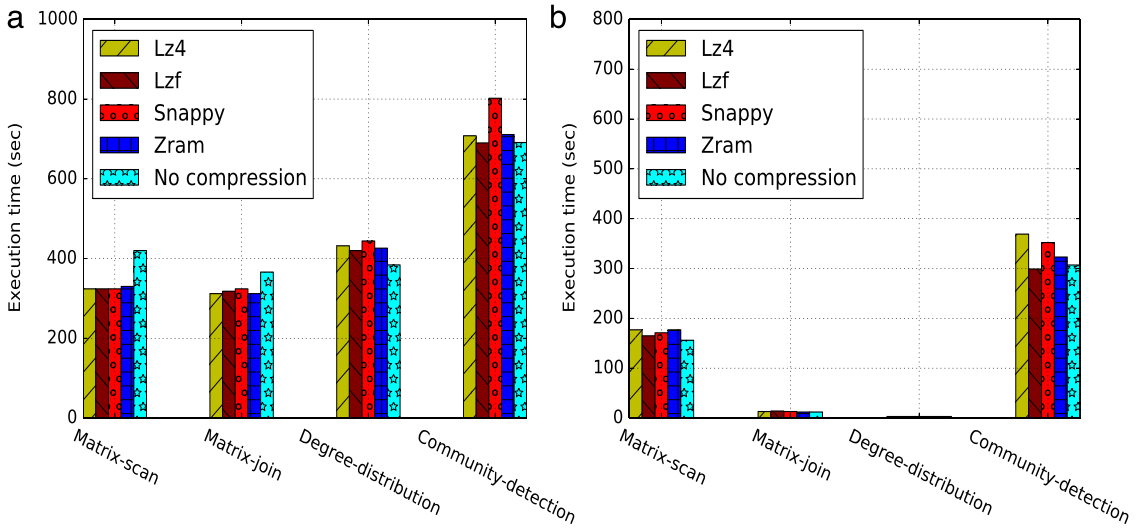

Fig. 9. Execution times of the analytics with different compression approaches. Input data are read from HDFS (a). and cached RDDs (b). 12 servers are used in the experiment.

of caching policies with different RDDs being cached in the experiments. Specifically, we run *matrix-join*. A fragment of its source code is shown in Fig. 10. It uses three RDDs, including *kvrdd1* and *kvrdd2* for storing input matrices, and *joinrdd* for storing results of the operation.

We cache different RDDs to study their impact on the execution time of *matrix-join*. For *Policy1*, only input RDDs (e.g., kvrdd1 and kvrdd2) are cached. For *Policy2*, only output RDDs (e.g., joinrdd) are cached. For *Policy3*, both input and output RDDs are cached. We measure the execution time of the analytic with Spark caching RDDs using only memory, only disks, and both (*Memory&disk*), respectively. The results are shown in Fig. 11.

Very interesting results can be observed. (1) Caching all the RDDs using *Policy 3* does not guarantee best performance in execution time. For example, the execution time of *matrix-join* using *Policy3* is 14% longer than that using *Policy2* on average. (2) When RDDs are only cached in memory, caching using *Policy1* may dramatically degrade performance of analytics by 60% and 78% compared to other two polices. The reason is that Spark had to recompute *joinrdd* given only *kvrdd1* and *kvrdd2* were available in the cache. However, recomputing the *joinrdd* is much slower than reusing the cached output RDDs. (3) Analytics performance can be significantly improved with enough memory being allocated for caching shuffle files. For example, when disks are used to store RDDs, execution time of *matrix-join* can be reduced by up to 78%, compared to using memory only. It is because 4 GB memory, which was originally used for storing the RDDs, was freed for caching the shuffle files.

4.5. Which serializer to use?

Two serializers are available in Spark, default Java and Kryo [2]. The effectiveness of serialization is the key to the performance of Spark, especially under high memory pressure and when a large amount of data needs to be deserialized to objects in memory. We study the performance of the existing serializers in the context of serialization and deserialization of scientific data. In the experiments we run the analytics with Java and Kryo serializers. Fig. 12(a) shows the execution times of the analytics with input datasets read from HDFS. And Fig. 12(b) show the times with input read from the RDDs cached in the previous operation by Spark. In the figures, we can find Kyro outperforms Java by 16% on average. The

```
import org.apache.spark._

object JoinQuery{
  def main(args: Array[String]){
    ...
    val filerdd1=sc.textFile(sourcefile1).map(_.split("\t")).
                                          filter(_.size==11)
    val filerdd2=sc.textFile(sourcefile2).map(_.split("\t")).
                                          filter(_.size==11)

    val kvrdd1=filerdd1.map(line=>((line(0),line(1)), (line(2))))
    val kvrdd2=filerdd2.map(line=>((line(0),line(1)), (line(2))))

    val joinrdd=kvrdd1.join(kvrdd2)
    joinrdd.repartition(5).saveAsTextFile(joinoutput)
    ...

  }
}
```

Fig. 10. A fragment of source code of *matrix-join* with Spark.

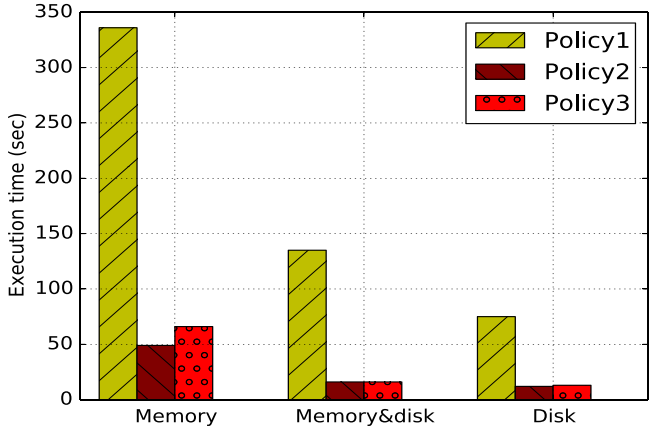

Fig. 11. Analytic execution times with different Spark RDDs being cached. *Policy1*: cache both kvrdd1 and kvrdd2; *Policy2*: cache joinrdd only; *Policy3*: cache all the three RDDs, kvrdd1, kvrdd2, and joinrdd. 12 servers are used in the experiment.

performance of *community-detection* is more sensitive than other analytics because it executes more serialization operations on 9 RDDs across 191 stages.

Summary: Memory management of Spark plays a key role in the execution of data-intensive analytics. However, installing larger memory does not guarantee a reduced execution time of an analytic for disk bottlenecks. Both compression and huge page may compromise the performance of analytics. The compression approaches supported in the Linux kernel can achieve similar performance as those provided by Spark. Finally, memory efficiency can be compromised if Java objects are not selectively cached in memory.

5. How do storage systems affect results?

Storage systems are heavily involved in the execution of scientific analytics with Spark. (1) Spark reads input data and writes output data to an HDFS distributed file system. (2) Map tasks of Spark write shuffle files directly to disks and then exchange the data with reduce tasks. (3) RDDs can be persisted on the disks for reuse of computational results across operations. In this section, we will study how the configurations of storage systems affect the performance of scientific analytics. Furthermore, as SSDs are increasingly adopted in HPC systems, we will also evaluate how SSDs would improve the performance of Spark analytics.

5.1. What is the impact of distributed file systems?

HDFS [31] is used by Spark for scalable and reliable storage services. In the experiments, we measure the execution times of the analytics as the block size of HDFS is increased from 64 to 128, 256, 512 and 1024 MB. No Spark cache was used. Fig. 13 shows the results.

An interesting observation is that only the performance of *community-detection* is very sensitive to the block size. For example, its execution time is reduced by 78% with 1024 MB block size, compared to that with 64 MB. We found this is because of reduced shuffle overhead. Multiple factors can impact the shuffling performance, such as the number of reduce operations, the number of mapper and reducer tasks involved in shuffling, and the amount of page cache available for caching shuffle files in memory. We found the first two factors are the major ones affecting the performance of *community-detection*. Specifically, it requires 6 reduce stages while other three analytics have one to three stages. In addition, Spark determines the number of tasks according to the number blocks of an input file in HDFS. When the block size is 64 MB, Spark can

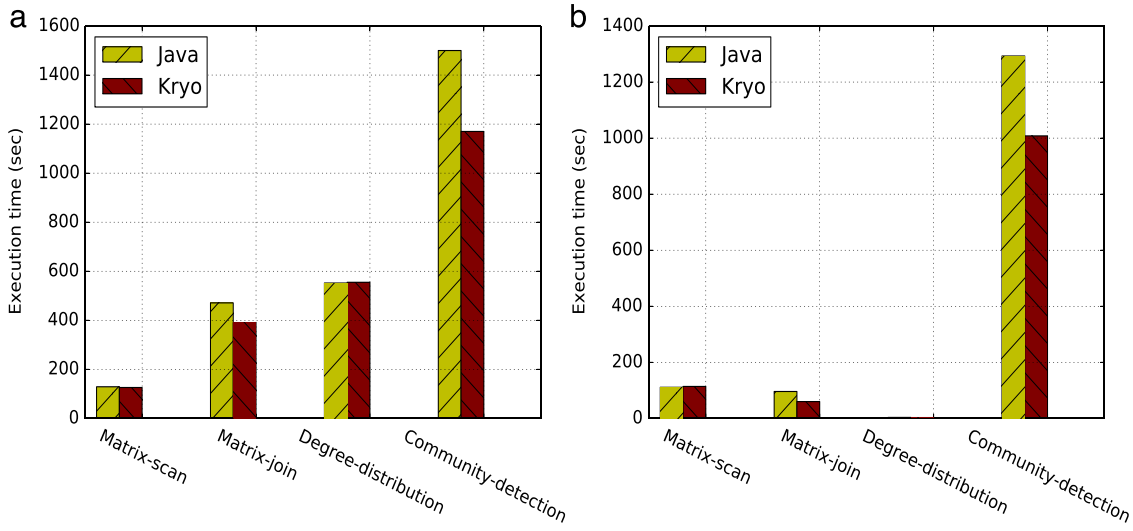

Fig. 12. Execution times of the analytics with default Java and Kryo serializers. Input datasets are read from HDFS (a) and cached RDDs (b). 12 servers are used in the experiment.

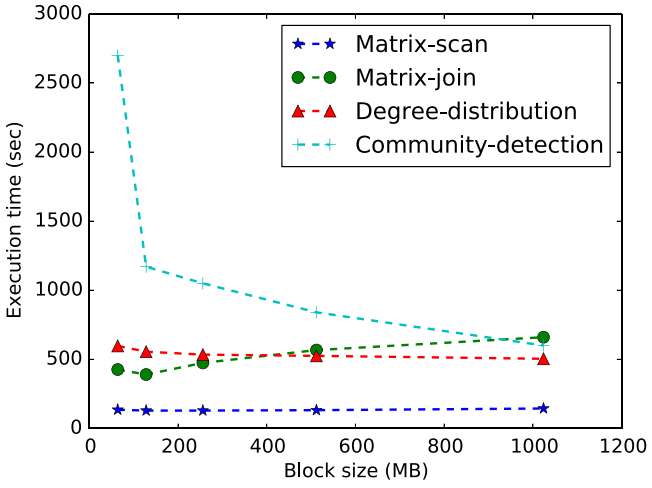

Fig. 13. Execution times of the analytics when the block size of HDFS is increased from 64 to 1024 MB. 12 servers are used in the experiment.

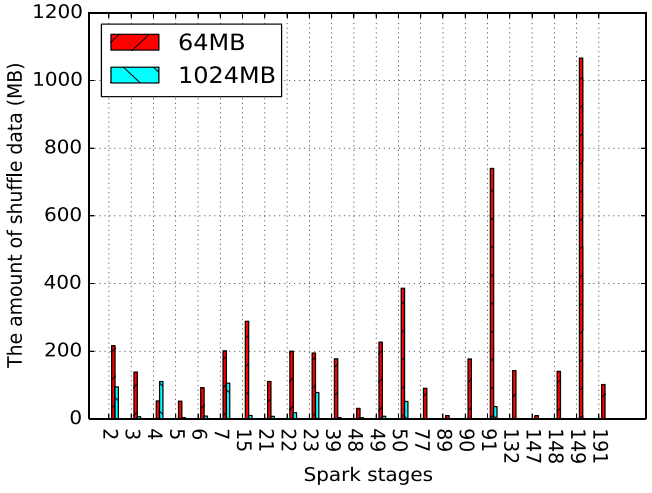

Fig. 14. The amount of shuffle data at each execution stage of *community-detection* with Spark.

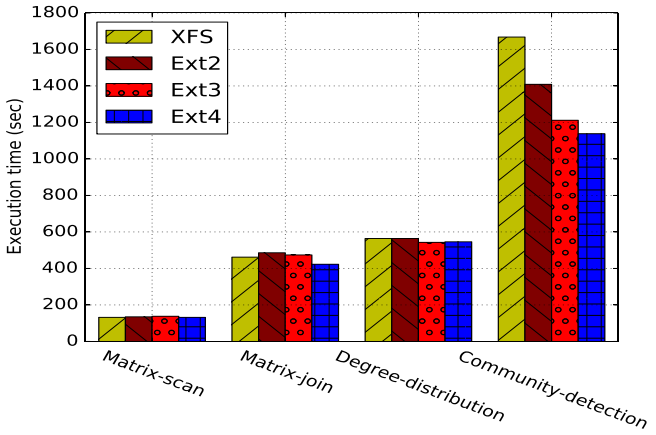

Fig. 15. Execution times of the analytics with ext2, ext3, ext4, and XFS file systems. 12 servers are used in the experiment.

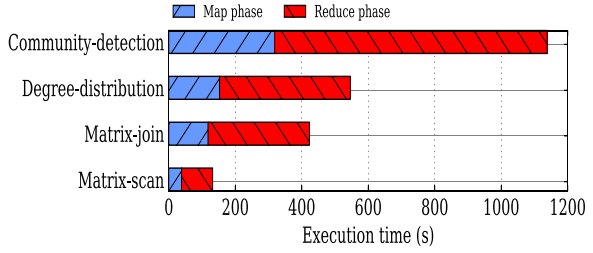

Fig. 16. Execution time of the map and reduce phases when ext4 is used. 12 servers are used in the experiment.

Table 3
A comparison of I/O throughput of the SSDs and disks with sequential and random workloads. 4 kB requests are used in the benchmarking.

	SSD	Hard disk
Capacity	120 GB	1 TB
Sequential read	170 MB/s	114 MB/s
Random read	27 MB/s	0.71 MB/s
Sequential write	189 MB/s	112 MB/s
Random write	52 MB/s	0.63 MB/s

create 828 map tasks and 424 reduce tasks. Each map task can write 424 shuffle files. Therefore, 351,072 shuffle files need to be accessed during shuffling at a reduce stage. In comparison, when the block size is 1024 MB, only 1458 files were accessed. Finally, Spark creates index files for improving the I/O performance of data shuffle. Having more shuffle files requires more small index files being maintained in the system. As shown in Fig. 14, the total amount of shuffle data can be 8.9X larger with 64 MB block size than that with 1024 MB block size. Because of the above reasons, the shuffle performance is dramatically improved with large block size for *community-detection*. Other analytics have much fewer reduce stages leading to less shuffling overhead.

5.2. What is the impact of local file systems?

We compare the execution times of the analytics with 4 local file systems including ext2, ext3, ext4, and XFS. Spark cache is not used. The results are shown in Fig. 15. Our results are in line with the previous published results [10]. For *community-detection*, ext4 can reduce their execution time by 32% compared to XFS. This is because ext4 uses *extents*, range of contiguous physical blocks to improve data locality and reduce fragmentation when serving large files. Each ext4 inode can store up to 4 extents, each of which can index up to 128 MB contiguous disk space. Therefore, accessing large file using extents can reduce the number of reads to read inode, compared to other file systems without using the extents. We found no significant improvement for other analytics when ext4 is used.

Another observation is that Spark spent 2X more time on the reduce phase than that on the map phase when executing the 4 analytics as shown in Fig. 16. The reason is that the reduce phase needs to access data shuffle file on disks which can incur small and random I/Os.

5.3. What is the impact of SSDs?

SSDs have been increasingly adopted by HPC systems for its superior random performance. We evaluate whether SSDs can significantly improve performance of scientific data analytics with Spark. Specifically, each compute node is configured with an OCZ Vertex 3 SSD and a Seagate Barracuda disk. The performance characteristics of the two devices are compared in Table 3 using the fio benchmark [18].

In the experiments, we measure the execution times of the analytics when disks and SSDs are respectively used to store shuffle

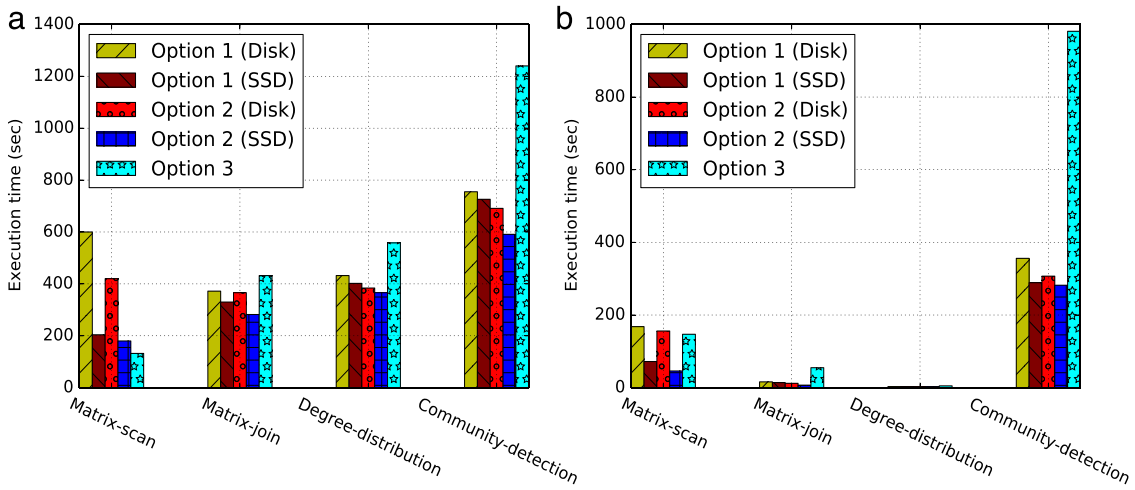

Fig. 17. (a) The analytics are executed the first time with their input data being read from HDFS. Then the input RDDs are cached using different storage options. (b) The analytics are executed using the cached RDDs. 12 servers are used in the experiment.

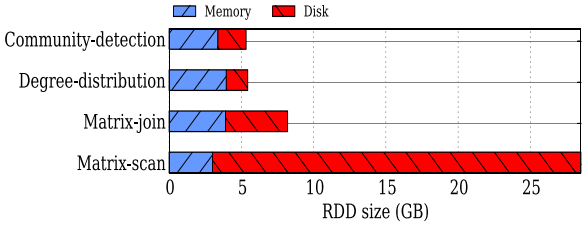

Fig. 18. A distribution of RDDs in memory and on disks during the execution of the analytics.

files and RDDs. In addition, we compare three different storage options of RDD management. *Option 1*: RDDs are cached in memory and on disks; *Option 2*: RDDs are only cached on disks; *Option 3*: RDDs are only cached in memory. No RDDs are stored on disks or SSDs. For *Option 3*, if the size of RDDs is larger than memory size, only RDD partitions, which fit in memory, are cached. The rest of partitions will be recomputed or read from HDFS when being accessed again. The results are shown in Fig. 17. We have three interesting observations from the figure.

First, *the performance benefit of SSDs can be insignificant.* SSDs have superior I/O performance compared to disks. As shown in Table 3, I/O throughput of SSDs is 38X and 83X higher than disks for random reads and writes, respectively. However, in Fig. 17, we observed that using SSDs reduced the execution time of the analytics by 34% on average. For graph analytics, it improved their performance by as little as 8%. Using Linux profiling tools (e.g., blktrace [1]), we found it is because Spark workloads tend to issue large block-level I/O requests (e.g., 512 kB) and have strong spatial locality. These optimizations in the system can significantly improve I/O efficiency, therefore, reducing the benefit of using SSDs.

Second, *the effectiveness of SSDs is determined by the amount of Java objects (e.g., RDDs) serialized on the storage devices.* For example, the performance of *matrix-scan* is more sensitive to SSDs than other analytics. Specifically, using SSDs can reduce its execution time by up to 57% for two reasons. (1) The analytic has much larger amount of data being serialized on disks. We compare the amount of objects stored in memory and on disks in Fig. 18. *Matrix-scan* stored 25.5 GB on disks, which is 5.9X, 17.2X, and 13.1X larger than *matrix-join*, *degree-distribution*, and *community-detection*, respectively. (2) 94% of I/O requests were small (e.g., 4 kB).

Third, *caching RDDs in memory may not achieve optimal performance.* The performance of the analytics (e.g. *community-detection*) with memory only using *Option 3* is much worse than those with other options. This is because the cost of recomputing intermediate results is typically higher than the cost of storing them as RDDs on storage devices, especially for complex analytics consisting of a series of computation stages.

Summary: The performance of *community-detection* is sensitive to the block size of HDFS. Widely used local file systems have similar performance. Using storage devices for storing RDDs at runtime can significantly reduce execution time of analytics. An SSD can significantly improve performance of analytics when accessing disk for Java objects becomes a bottleneck.

6. Discussion

Having comprehensively evaluated Spark and SciDB, we found an I/O bottleneck is still the major concern for in-memory data analytics especially when out-of-core computation is required to process large matrices. This is because they need to frequently access disks for creating RDDs using input files, caching RDDs, and shuffling data between thousands of mappers and reducers. Strong correlations can be found between the performance of scientific analytics and the system characteristics, which are insightful to the design and implementation of in-memory analytics platforms for processing scientific data.

Programming: Spark provides users multiple knobs to control memory allocations for memory stack/heap of Java processes and page cache of operating systems. Programmers should carefully calculate memory demands of analytics and use the knobs to help reduce execution time of the analytics. Caching compute results is the only way to achieve good scalability for data-intensive analytics. However, choosing which RDDs to cache can be critical to their performance because caching more data can reduce the effectiveness of page cache. Based on our results, caching output of a reduce operation can be important. To achieve an optimal performance, an auxiliary tool should be designed to automatically detect the RDDs that have the most impact, while taking into account memory demands of various running instances.

Virtual memory: One of the results of our study is that using transparent huge pages (THP) may not help analytics with Spark because of random memory access pattern. Therefore, we need to rethink the design of virtual memory systems supporting huge pages. For example, THP should be disabled when kernel memory defragmentation is frequently operated. We also found that page

cache size can be critical to performance of binary array operations. Lzf is the best algorithm for compression of the scientific data used for this study.

File and storage systems: For analytics having multiple stages of reduce operations, we should use large block size to store input data files in HDFS. When the size of page cache is small, we should decrease the number of mappers and reducers to decrease the amount of shuffle data as well as the overhead of shuffle file management. When the page cache is adequate, it can significantly hide shuffle overhead by reducing random I/Os on disks. We did observe that local file systems played a vital role for the analytics having frequent shuffle operations.

Configuration: Many configuration parameters of Spark are relevant to performance tuning, as misconfiguration can easily lead to suboptimal performance of various system components, e.g., memory and disks. We need to use existing auto-tuning software or configuration checkers to learn how parameters can affect system performance.

Testing: Many distributed data analytics frameworks, e.g., SciSpark [26], SciDB [32], and SciHadoop [7], have been proposed to process scientific data generated in different areas. An open and shared large-scale testing platform supporting all the frameworks is needed to determine which one of them can best serve domain-specific scientific data analytics. Further, we need to develop a set of tools that can help test programs written using these frameworks and free domain scientists and engineers from this error-prone task.

In-situ execution: Another interesting result of our study is that SciDB can achieve better performance for very large datasets. This result suggests that array data model can be helpful for less disk I/Os and data transfer with in-situ execution for data-intensive analytics. However, based on our experience, it is not straightforward to directly program graph algorithms using array data model of SciDB. We need to help users easily switch between the two data models. Furthermore, a job scheduler of a distributed system should decide which data model to use given various configurations of software platforms and characteristics of scientific data analytics.

7. Related work

We review the research literature in three areas: scientific database benchmarks, performance understanding of data analytics platforms, and scientific data analysis using Spark and SciDB.

7.1. Scientific database benchmarks

Scientific data warehouse is being widely used for interactive and low-cost exploration over the daily produced data. Two approaches are typically used for building the data warehouse. One is adopting traditional SQL parallel database management systems (e.g., MySQL [23]), while the other one is using NoSQL database systems (e.g., SciDB [32], Hadoop [15], and Spark [45]).

There has been much research effort on a comparison of SQL and NoSQL databases using transactional workloads [27] and scientific workloads [9]. Most of the studies were conducted for disk-based software systems. In contrast, ArrayBench is proposed to evaluate in-memory data analytics. As shown in Section 3, they have different system characteristics, compared to previous disk-based database systems (e.g., SciDB), which are mostly optimized for I/O efficiency.

Three scientific database benchmarks were implemented based on the requirements of domain-specific scientific applications. For example, SS-DB [9] and SDSS [36] were modeled on astronomy workloads. Sequoia 2000 [11] was designed for geoscience research. ArrayBench is the only one designed using real biological workflows processing matrices. We believe our findings are applicable to other domain-specific analytics because the requirements for matrix processing are common across disciplines.

7.2. Performance understanding of data analytics platforms

Much of the existing work examined the system characteristics of data analytics platforms. They studied how performance bottlenecks are correlated to various system components. Camdoop [8] proposed to reduce networking traffic caused by all-to-all communication for shuffle. Themis [29] and Pacman [6] were proposed to solve disk bottlenecks by reducing swapping I/Os and improving cache efficiency. In the most recent study [25] on the performance of in-memory data analytics platforms, the authors used the block time analysis to evaluate the impact of disk I/O, network, and straggler tasks on the execution times of analytics. However, these studies used transactional workloads of a relational-data model. The novelty of this work is the use of scientific workloads of an array-data model to reveal interesting results which seem counter to previous findings. Instead of using benchmarks, Jian et al. [17] studied bug correlation using patch databases. Because Spark has a relatively short history, the number of its patches is not sufficient for a comprehensive evaluation.

7.3. Scientific data analysis using Spark

Spark has been adopted in a number of scientific areas for developing data-intensive analytics running on large-scale HPC clusters or in cloud environments. SciSpark was designed to process earth science data and climate data for weather event detection [26,44]. It extends Spark for processing structured scientific data in netCDF [30] and HDF [24] formats using the Scientific Resilient Distributed Dataset as the key programming abstraction. Hail was proposed to process genetic data in VCF, BGEN, or PLINK format [13,16] using Spark. It supports a variety of operations, e.g., annotations, for genetic data analysis. Toil enables biomedical data analysis targeting RNA sequencing [43]. It is integrated with Spark and is compatible with common workflow language. It has been tested in multiple cloud environments, e.g., Amazon Web Services [4] and Microsoft Azure [22]. Most recently, Zhang et al. developed Kira for processing astronomy imagery using Spark [47] and showed that Spark Streaming's capability in supporting near real-time data analysis for surveys, e.g., LSST [19]. Different from these existing work, our benchmark can be used as the building blocks of solutions to processing data for research of systems genetics, especially on detecting genes that are co-functional in the biological networks. Even though the difference, our findings are general to other applications using the Spark in-memory data analytics framework.

8. Conclusion

We have designed ArrayBench for a comprehensive and in-depth study of the in-memory data analytics platform Spark versus the disk-based system SciDB. Our experimental results confirmed many well understood features of Spark, such that caching intermediate results in applications is a very effective approach for data-intensive analytics with weak temporal locality. At the same time, we also observed many unexpected performance issues. For example, Spark does not consistently outperform SciDB. Most of the existing problems in the design of Spark are related to Java object management on disks and data shuffling. In general, the performance of data shuffling is determined by the availability of page cache of operating systems. Users should understand the memory footprint of Java objects (e.g., RDDs) and its performance impact in a computing environment with a deep and heterogeneous memory hierarchy. These findings should help the development of new scientific data analytics with Spark as well as the new development of large-scale analytics platforms for in-memory matrix data analysis.

As future work, we plan to (1) further evaluate the performance of Spark and SciDB in cloud environments, e.g., Amazon EC2 and Microsoft Azure, using ArrayBench benchmark and study query performance using Spark DataFrame and Dataset and/or GPUs; (2) design an auto-tuning framework to automate the setting of Spark's parameters at runtime considering their performance impact on memory, storage, and networks; (3) develop a set of tools that help developers understand the performance characteristics of RDD operations used in scientific data analytics by exploiting code analysis using compilers.

Acknowledgments

We would like to thank the anonymous reviewers for their helpful comments and feedback. This research was supported in part by NSF ACI-1565338 and WSU Vancouver Research Grant.

References

[1] A block layer I/O tracing tool, http://linux.die.net/man/8/blktrace.
[2] A fast and efficient object graph serialization framework for Java, https://github.com/EsotericSoftware/kryo.
[3] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, A. Vahdat, Hedera: Dynamic Flow Scheduling for Data Center Networks, in: 7th USENIX Symposium on Networked Systems Design and Implementation, NSDI'10, 2010.
[4] Amazon elastic compute cloud, https://aws.amazon.com/ec2/.
[5] G. Ananthanarayanan, A. Ghodsi, S. Shenker, I. Stoica, Effective straggler mitigation: attack of the clones, in: 10th USENIX Symposium on Networked Systems Design and Implementation, NSDI'13, 2013.
[6] G. Ananthanarayanan, A. Ghodsi, A. Wang, D. Borthakur, S. Kandula, S. Shenker, I. Stoica, PACMan: coordinated memory caching for parallel jobs, in: 9th USENIX Symposium on Networked Systems Design and Implementation, NSDI'12, 2012.
[7] J.B. Buck, N. Watkins, J. LeFevre, K. Ioannidou, C. Maltzahn, N. Polyzotis, S. Brandt, SciHadoop: array-based query processing in hadoop, in: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis SC '11, 2011.
[8] P. Costa, A. Donnelly, A. Rowstron, G. O'Shea, Camdoop: exploiting in-network aggregation for big data applications, in: 9th USENIX Symposium on Networked Systems Design and Implementation, NSDI'12, 2012.
[9] P. Cudre-Mauroux, H. Kimura, K.-T. Lim, J. Rogers, S. Madden, M. Stonebraker, S.B. Zdonik, P.G. Brown, SS-DB: A standard science DBMS benchmark, in: 4th Extremely Large Databases Conference, XLDB'10, 2010.
[10] A. Davidson, A. Or, Optimizing shuffle performance in Spark.
[11] J. Dozier, M. Stonebraker, J. Frew, Sequoia 2000: a next-generation information system for the study of global change, in: IEEE Symposium on Mass Storage Systems, MSS'94, 1994.
[12] E.C. Fieller, H.O. Hartley, E.S. Pearson, Tests for rank correlation coefficients. I, Biometrika 44 (3–4) (1957) 470–481.
[13] A. Ganna, G. Genovese, D.P. Howrigan, A. Byrnes, M.I. Kurki, S.M. Zekavat, C.W. Whelan, M. Kals, M.G. Nivard, A. Bloemendal, J.M. Bloom, J.I. Goldstein, T. Poterba, C. Seed, R.E. Handsaker, P. Natarajan, R. Magi, D. Gage, E.B. Robinson, A. Metspalu, V. Salomaa, J. Suvisaari, S.M. Purcell, P. Sklar, S. Kathiresan, M.J. Daly, S.A. McCarroll, P.F. Sullivan, A. Palotie, T. Esko, C.M. Hultman, B.M. Neale, Ultra-rare disruptive and damaging mutations influence educational attainment in the general population, Nature Neurosci. 19 (12) (2016) 1563–1565.
[14] J. Gray, Benchmark Handbook: For Database and Transaction Processing Systems, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1992.
[15] Hadoop, https://hadoop.apache.org.
[16] Hail: An open-source framework for scalable genetic data analysis, https://github.com/hail-is/hail.
[17] J. Huang, X. Zhang, K. Schwan, Understanding issue correlations: a case study of the hadoop system, in: Proceedings of the Sixth ACM Symposium on Cloud Computing, SoCC '15, 2015.
[18] Inspecting disk IO performance with fio, https://www.linux.com/learn/tutorials/442451-inspecting-disk-io-performance-with-fio/.
[19] Z. Ivezic, J.A. Tyson, T. Axelrod, D. Burke, C.F. Claver, K.H. Cook, S.M. Kahn, R.H. Lupton, D.G. Monet, P.A. Pinto, M.A. Strauss, C.W. Stubbs, L. Jones, A. Saha, R. Scranton, C. Smith, LSST Collaboration, LSST: From science drivers to reference design and anticipated data products, in: American Astronomical Society Meeting Abstracts #213, Vol. 41, 2009.
[20] S. Jiang, F. Chen, X. Zhang, CLOCK-Pro: An effective improvement of the clock replacement, in: Annual Technical Conference, USENIX ATC'05, 2005.
[21] M.L. Massie, B.N. Chun, D.E. Culler, The ganglia distributed monitoring system: Design, implementation and experience, in: Parallel Computing, 2003.
[22] Microsoft azure, https://azure.microsoft.com.
[23] MySQL database, http://www.mysql.com.
[24] NCSA HDF Calling Interfaces and Utilities, National Center for Supercomputing Applications, 1989.
[25] K. Ousterhout, R. Rasti, S. Ratnasamy, S. Shenker, B.-G. Chun, Making sense of performance in data analytics frameworks in: 12th USENIX Symposium on Networked Systems Design and Implementation, NSDI'15, 2015.
[26] R. Palamuttam, R.M. Mogrovejo, C. Mattmann, B. Wilson, K. Whitehall, R. Verma, L. McGibbney, P. Ramirez, SciSpark: Applying in-memory distributed computing to weather event detection and tracking in: 2015 IEEE International Conference on Big Data (Big Data), 2015 pp. 2020–2026.
[27] A. Pavlo, E. Paulson, A. Rasin, D.J. Abadi, D.J. DeWitt, S. Madden, M. Stonebraker, A comparison of approaches to large-scale data analysis, in: Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data, SIGMOD'09, 2009.
[28] U.N. Raghavan, R. Albert, S. Kumara, Near linear time algorithm to detect community structures in large-scale networks, Phys. Rev. 76 (2007).
[29] A. Rasmussen, V.T. Lam, M. Conley, G. Porter, R. Kapoor, A. Vahdat, Themis: An I/O-efficient MapReduce, in: Proceedings of the ACM Symposium on Cloud Computing, SoCC'12, 2012.
[30] R. Rew, G. Davis, NetCDF: an interface for scientific data access, IEEE Comput. Graph. Appl. 10 (4) (1990) 76–82.
[31] K. Shvachko, H. Kuang, S. Radia, R. Chansler, The hadoop distributed file system in: 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies, MSST'10, 2010.
[32] E. Soroush, M. Balazinska, Time travel in a scientific array database in: IEEE International Conference on Data Engineering, ICDE'13, 2013.
[33] E. Soroush, M. Balazinska, D. Wang, ArrayStore: a storage manager for complex parallel array processing, in: Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data, SIGMOD'11, 2011.
[34] Spark compression and serialization, http://spark.apache.org/docs/latest/configuration.html#compression-and-serialization.
[35] F.A.F. Stephen, P. Ficklin, Feng Luo, The association of multiple interacting genes with specific phenotypes in rice using gene coexpression networks, Plant Physiol. 154 (1) (2010) 13–24.
[36] A.S. Szalay, P.Z. Kunszt, A. Thakar, J. Gray, D. Slutz, R.J. Brunner, Designing and mining multi-terabyte astronomy archives: the sloan digital sky survey, in: Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, SIGMOD'00, 2010.
[37] R. Tarjan, Depth-first search and linear graph algorithms, SIAM J. Comput. 1 (2) (1972) 146–160.
[38] Transaction processing performance council (TPC). TPC benchmark DS standard specifi- cation. http://www.tpc.org/tpcds/spec/tpcds_1.1.0.pdf.
[39] Transparent hugepage support, https://www.kernel.org/doc/Documentation/vm/transhuge.txt.
[40] Tuning spark, http://spark.apache.org/docs/1.6.0/tuning.html.
[41] Tuning your SciDB Installation, http://www.paradigm4.com/HTMLmanual/14.8/scidb_ug/apc.html.
[42] Understanding your Spark application through visualization, https://databricks.com/blog/2015/06/22/understanding-your-spark-application-through-visualization.html.
[43] J. Vivian, A.A. Rao, F.A. Nothaft, C. Ketchum, J. Armstrong, A. Novak, J. Pfeil, J. Narkizian, A.D. Deran, A. Musselman-Brown, H. Schmidt, P. Amstutz, B. Craft, M. Goldman, K. Rosenbloom, M. Cline, B. O'Connor, M. Hanna, C. Birger, W.J. Kent, D.A. Patterson, A.D. Joseph, J. Zhu, S. Zaranek, G. Getz, D. Haussler, B. Paten, Toil enables reproducible, open source, big biomedical data analyses, Nature Biotechnol. 35 (4) (2017) 314–316.
[44] B. Wilson, R. Palamuttam, K. Whitehall, C. Mattmann, A. Goodman, M. Boustani, S. Shah, P. Zimdars, P. Ramirez, SciSpark: Highly interactive in-memory science data analytics, in: 2016 IEEE International Conference on Big Data (Big Data), 2016, pp. 2964–2973.
[45] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M.J. Franklin, S. Shenker, I. Stoica, Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing, in: 9th USENIX Symposium on Networked Systems Design and Implementation, NSDI'12, 2012.
[46] M. Zaharia, M. Chowdhury, M.J. Franklin, S. Shenker, I. Stoica, Spark: cluster computing with working sets, in: Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing, HotCloud'10, 2010.
[47] Z. Zhang, K. Barbary, F.A. Nothaft, E.R. Sparks, O. Zahn, M.J. Franklin, D.A. Patterson, S. Perlmutter, Kira: Processing Astronomy Imagery Using Big Data Technology, IEEE Trans. Big Data PP (99) (2017) 1–14.
[48] zram: Compressed RAM based block devices, https://www.kernel.org/doc/Documentation/blockdev/zram.txt.

Xuechen Zhang received the M.S. and the Ph.D. degrees in Computer Engineering from Wayne State University. He is currently an assistant professor in the School of Engineering and Computer Science at Washington State University Vancouver. His research interests include the areas of file and storage systems, parallel and distributed data management systems, and high-performance computing. He is a member of the IEEE.

Ujjwal Khanal is a CS graduate student at Washington State University Vancouver.

Xinghui Zhao received the Ph.D. degree in Computer Science from University of Saskatchewan. She is an assistant professor in the School of Engineering and Computer Science at Washington State University Vancouver. She is interested in developing models, software abstractions, and high-level programming constructs which enable fine-grained resource coordination in large-scale distributed systems.

Stephen Ficklin received the Ph.D. degree in Plant and Environmental Sciences from Clemson University. He is an assistant professor affiliated with the Department of Horticulture, Washington State University. He is interested in the improvement of the sensitivity and specificity of the models and creation of cyberinfrastructure to improve access and use of systems-genetics datasets.