Exploring Benefits of NVMe SSDs for BigData Processing in Enterprise Data Centers

Mahsa Bayati
Northeastern University
Boston, USA
bayati.m@husky.neu.edu

Janki Bhimani
Northeastern University
Boston, USA
bhimani@ece.neu.edu

Ronald Lee
Samsung Semiconductor
San Jose, CA, USA
r2.lee@samsung.com

Ningfang Mi
Northeastern University
Boston, USA
ningfang@ece.neu.edu

***Abstract*—Big data processing environments such as Apache Spark are prominently deployed for applications with large scale workloads. New storage technologies such as Non-Volatile Memory Express Solid State Drives (NVMe SSDs) provide higher throughput comparing to the traditional Hard Disk Drives (HDDs). Therefore, NVMe SSDs are rapidly substituting HDDs in modern data centers. In this paper, we explore whether it is critically necessary to use NVMe SSD for a large workload running on the Spark big data framework. Specifically, we investigate what are the influential factors of application design and Spark data processing framework to exploit the benefits of NVMe SSDs. Our real experimental results reveal that some applications even with large workloads cannot fully utilize NVMe SSDs to obtain high I/O throughput. Interestingly, we find out that characteristics of Spark data processing framework such as *shuffling* (i.e., the volume of transition data generated by an application), and *parallelism* (i.e., the number of concurrently running tasks) has very crucial impacts on the performance of big data applications running on NVMe SSDs.**

***Keywords*-NVMe SSD; Throughput; Spark; I/O Access Pattern; Data Processing.**

I. Introduction

Big data processing environments, such as MapReduce [1] and its implementations like Hadoop [2] and Spark [3], provide a productive high-level programming interface for large scale data processing and analytics. Apache Hadoop [2], a widely adopted cloud framework, has been criticized in recent years for its inefficiency of handling iterative and interactive applications [4]. Apache Spark [3] overcomes Hadoop's shortcomings by caching all intermediate data represented as Resilient Distributed Datasets (RDDs) into memory. Thus, the Spark data processing framework is designed to limit the large amounts of disk I/O operations for storing and accessing intermediate data. A Spark architecture is generally composed of a master (or driver) that manages the cluster and distributes jobs and tasks and multiple workers that process assigned jobs and tasks. Besides, Hadoop Distributed File System (HDFS), a distributed file system, is used as an important component that is supported by Spark for data sharing and fault tolerance in distributed systems. However, with the explosion of big data, such frameworks need to execute hundreds or thousands of parallel tasks to process massive datasets, that generate a huge amount of the disk I/O operations which are still the biggest bottleneck in the Spark cluster.

Traditional Hard disk drive (HDD) has been the predominant storage device due to its low price. However, HDD performance suffers a lot due to the seeking time and mechanical movement of the disk head. Recent advancement in storage devices like Non-Volatile Memory Express (NVMe) [5] Solid State Drives (SSDs), boosts the expected throughput in comparison to traditional HDDs. Thus, one popular solution to solve the I/O bottleneck issue is to replace HDDs with SSDs. In particular, NVMe SSD utilizes a large pool of NVMe queues to provide high I/O parallelism and further improve I/O access performance. Therefore, as the big data processing with large workloads becomes prominent, more cluster systems and data centers reconstruct their storage system by moving data to NVMe SSDs in order to obtain high I/O performance for big data processing applications. However, previous study [6] has shown that the existing large data processing frameworks (such as Hadoop) do not fully exploit SSD benefits. We also found that these data processing frameworks impose different aspects, such as the level of parallelism and scheduling of data flow, which can affect I/O access pattern and performance. These findings motive us to construct a real distributed system and carry out a comprehensive study of understanding the performance characteristics of applications with large workloads while running on big data processing framework on the system supporting high-speed storage devices.

In this paper, we strive to investigate whether it is significantly necessary for all large scale workloads to deploy NVMe SSD and further identify the effects of application design characteristics and Spark platform elements on SSD utilization and I/O performance. In particular, we build a real Spark cluster across multiple physical servers. We use Apache Hadoop YARN for resource management and job scheduling and HDFS for distributed data management. We then run various data processing applications (ranging from CPU-intensive, I/O-intensive applications to iterative and non-iterative applications) on high-performance NVMe SSDs to explore different application I/O behavior and understand the impacts of data shuffling and task parallelism on I/O performance. Our experiment results include a detailed study of average I/O performance (e.g., throughput), I/O access patterns (e.g., I/O size and I/O offset distribution), and the shuffling data volume. The main contributions of this paper are as follows.

978-1-7281-4024-7/19/$31.00 ©2019 IEEE
DOI 10.1109/BIGCOM.2019.00024

Authorized licensed use limited to: FLORIDA INTERNATIONAL UNIVERSITY. Downloaded on October 24,2022 at 21:37:45 UTC from IEEE Xplore. Restrictions apply.

- Not all data processing applications (even with large data sets) can take advantage of NVMe SSD's high bandwidth. Thus, it is not critically necessary to deploy costly NVMe SSDs for processing large workloads of those data processing applications (e.g., CPU-intensive ones).
- Various Spark data processing framework factors, such as *shuffling* (i.e., the volume of shuffling data) and *parallelism* (i.e., the number of concurrently running tasks), have the essential impacts on the performance of the data processing applications running on NVMe SSDs.
- System designers should identify the characteristics of data processing applications as well as the Spark platform factors to choose the proper storage devices (e.g., HDDs or NVMe SSDs) for achieving high I/O performance and meanwhile maintaining a good profit.

The rest of this paper is organized as follows. Sec. II introduces the system architecture and application layout constructed in our real system deployment. Sec. III presents the experimental results and our observations. Sec. IV presents the related work. Finally, we conclude the paper and discuss our future work in Sec. V.

II. Hardware Architecture and Application Layout

In this section, we describe our infrastructure to systematically study the impact of SSDs for large-scale data processing frameworks such as Apache Spark. More specifically, we first explain different components of our application processing infrastructure and then describe the architectural layout of various operating layers in our real system deployment. Primarily, we set up our infrastructure using large-scale data processing framework of Apache Spark [3] over virtual machine hypervisor. We build Yarn [7] containers for resource management while running multiple Spark executors in virtual machines. Lastly, we perform data management using Hadoop Distributed File System (HDFS) [8] for data availability and reliability of virtual machine disk (VMDK) running over distributed physical servers.

A. Application Processing Infrastructure

Components of Our Data Processing Infrastructure: Fig. 1 illustrates an example of data flow when running a data processing application in our infrastructure. Specifically, we deploy Apache Spark as a large-scale data processing framework. Apache Spark is a fast and reliable open-source distributed data processing engine. Spark performs application execution using multiple virtual machines running over distributed physical server nodes. As shown in Fig. 1, Spark cluster comprises of a master and multiple workers, each running on different virtual machines. The Spark Application Manager on the master node consists of a *Spark driver* which includes the modules such as *RDDGraph*, *DAGScheduler*, *TaskScheduler*, and *SchedulerBackend*. The *ClusterManager* module creates Resilient Distributed Datasets (RDDs) and performs a series of transformations to achieve its final result. Here, RDDs store

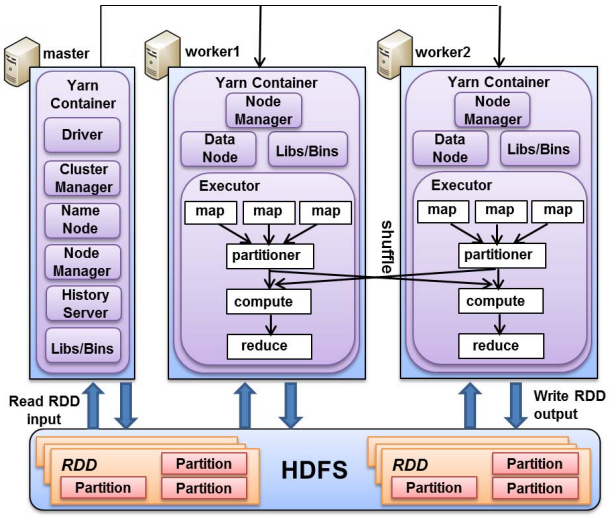

Fig. 1. An example of data flow while running application using Spark data processing framework with Yarn resource manager and HDFS distributed file system.

intermediate results into memory to accelerate data processing. These transformations of RDDs are translated into a Directed Acyclic Graph (DAG) and submitted to the scheduler to be distributed and executed on a set of worker nodes. There are multiple worker nodes in Spark and each of them is a Java Virtual Machine (JVM) process that runs multiple executors. These executors run tasks scheduled by the driver, store computation results in memory, and conduct on-disk or off-heap interaction with the storage system. Each worker node consists of one or more executors depending on the number of cores used by a worker.

The Yarn resource manager instantiates Yarn containers on the master and all worker nodes. The Yarn resource manager on master works with the Spark Application Manager to drive the application execution and accordingly allocate appropriate resources to each worker. Each worker node contains a component of Yarn, called *Node Manager*, which handles resource allocation among executors running on each node. Node Manager provides memory and CPU resources as resource containers, launches the assigned tasks, and tracks the processes executing on the node. We perform the data management using HDFS that is designed to provide scalable and reliable data storage over a large cluster of servers. HDFS is a scalable, fault-tolerant, distributed storage system that works closely with a wide variety of concurrent data access applications, coordinated by Yarn. The HDFS consists of the two main operational components, namely *NameNode* and *DataNode*. The NameNode is located on the master node, which keeps the metadata to record where each file is located across the cluster. Besides, each of worker nodes has a DataNode to manage the storage of the data with which it is working. In HDFS, a file is partitioned into one or more blocks, which are separately stored on a set of DataNodes. Multiple replicas of each data block are maintained for better

Authorized licensed use limited to: FLORIDA INTERNATIONAL UNIVERSITY. Downloaded on October 24,2022 at 21:37:45 UTC from IEEE Xplore. Restrictions apply.

performance and fault tolerance.

Execution of Data Processing Application: Upon launching a Spark application on our infrastructure, the master node creates RDDs and constructs an execution plan by making DAG. The corresponding required data is then identified by HDFS. HDFS also pre-processes and partitions the data in small data blocks that are replicated and distributed over multiple worker nodes. Simultaneously, the master node *maps* different data partitions and a set of transformation functions to each worker. The worker then shuffles the data from other workers to access the data block using HDFS which corresponds to the partition assigned to it. Each worker processes the *reduce* phase by computing the assigned transformation functions on respective data partitions.

Generally, a Spark data processing application may use multiple iterations to complete data processing. In each iteration, multiple independent Spark jobs run with multiple stages. Each stage consists of a different number of tasks that are often executed in parallel. The number of tasks for each stage depends on how input files are partitioned and distributed among nodes. Each worker node has one or more executors to perform a series of transformations on RDDs. Finally, through an action task, the master node accumulates the results which are then written to HDFS.

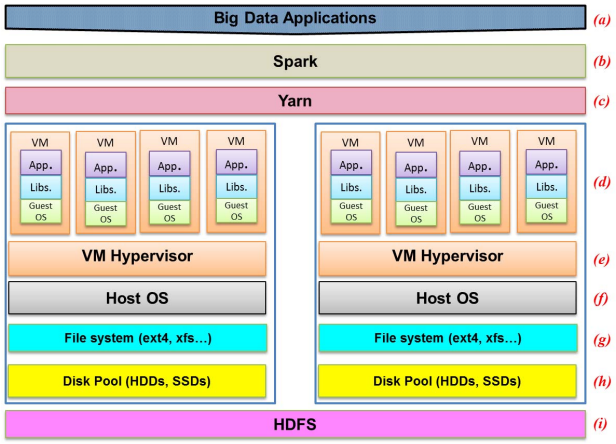

Fig. 2. Overview of the application stack in our cluster.

B. System Architecture

We build our architecture for efficient and reliable execution of the large-scale data processing applications. Fig. 2 illustrates the layout of various operating layers in our real system deployment. The workloads of big data processing applications, such as Spark-bench [9], TPC-H benchmarks [10], and K-means [11], are run using operating techniques of Spark data-processing framework (see Fig. 2(a) and (b)). The Spark data processing framework uses the yarn resource manager to schedule jobs and manage resource distribution over the cluster (see Fig. 2(c)). Specifically, Yarn decides how much resource (i.e., memory and cpu) should be dedicated to each Spark job, stage, and task. The particular configurations of Spark and Yarn that we use in our setup are specified in Table I.

On each physical server, a set of virtual machines (VMs) are created and managed by VM hypervisor, see Fig. 2(d) and (e). Specifically, in our infrastructure, we have multiple VMs with identical OS image, memory limit, and VMDK. We configure one of the VMs to serve as the master node and the remaining VMs as worker nodes for our Spark data processing framework. The VM hypervisor runs over the host OS and the file system (see Fig. 2(f) and (g)) of a physical server. The file system data is mapped on the pool of storage disks which can be either HDDs or SSDs, see Fig. 2(h). In this work, we consider deploying two different setups that use either traditional hard disks or NVMe SSDs for data storage. Lastly, we maintain the HDFS cluster over the physical storage pools (see Fig. 2(i)) for efficient data sharing and fault-tolerance while processing big data using multiple physical nodes.

TABLE I
SPARK AND YARN CONFIGURATION

Spark Configuration		Yarn Configuration	
		Version	2.9.1
Version	2.3.1	Max Container Mem.	3GB
Executor Mem.	1.8GB	Min Container Mem.	2.2GB
Overhead Mem.	1GB	Virtual Core/ Container	2
No. of Req. Executors	8	No. of Container/Node	1
		Node Manger Mem.	3GB
		HDFS block size	128MB

III. EXPERIMENTS AND RESULTS

In this section, we present our evaluation of the forementioned architectural layout with NVMe SSD devices by running different data processing applications (including defacto standard benchmarks like generic Spark applications and database I/O intensive applications) on our real Spark cluster.

A. Testbed Configuration

Platform - We run our experiments on two Intel(R) Xeon physical servers, each of which has 12 cores and 16GB RAM and consists of two different storage devices including NVMe SSD SAMSUNG PM963 model with 2TB capacity and HDD Dell Exos 15E900 model with 600GB capacity. Four virtual machines (VMs) are running on each physical server, and there are totally eight VMs in the cluster. Each VM is configured with 2 CPU cores, 3GB memory size, and 100GB disk size with 10GB of the swap memory.

Table II shows the hardware configuration for physical servers and VMs in our experiments. Hadoop-Yarn 2.9.1 and Spark 2.3.1 are installed on this platform with one master and seven worker nodes. We initiate one Yarn container with two virtual cores per node such that Spark can request for eight containers at the same time. All input and output results are stored in HDFS that has the default block size of 128MB. We ran our experiments with different HDFS block sizes, but our results remain similar. The evaluation results are obtained by using tools and commands, like `blktrace`, `dstat` and `iostat`.

Authorized licensed use limited to: FLORIDA INTERNATIONAL UNIVERSITY. Downloaded on October 24,2022 at 21:37:45 UTC from IEEE Xplore. Restrictions apply.

TABLE II
HARDWARE CONFIGURATION

Physical Servers	
CPU Type	Intel(R) Xeon
CPU Speed	3.7GHz
No. of Cores	12
OS	Ubuntu 16.04.5
No. of VMs	4
Mem. Size	16GB
Swap Size	128GB
SSD Storage Disk	NVMe SAMSUNG PM963 (2 TB)
HDD Storage Disk	Dell Exos 15E900 (600 GB)
VMs	
No. of Cores	2
Mem. Size	3GB
Disk Size	100GB
Swap Size	10GB
Java Version	1.8.0_181
Java Heap Size	2GB

Benchmarks - To evaluate and analyze the architectural layout, we consider benchmarking using applications with different characteristics, including CPU, memory and I/O intensive, iterative and non-interactive. Based on our experimental results, we group the benchmarks into generic Spark applications and database applications. We choose K-means clustering [11] and TPC-H-Spark [10] as examples to represent generic Spark applications and database applications, respectively.

- K-means clustering [11] is a popular unsupervised machine learning algorithm, which aims to partition and group "n" data points into "k" clusters. Each data point is allocated to one of k clusters with the nearest distance. The K-means algorithm iteratively performs the calculation until convergence of the data clusters. In our experiments, we set $k = 10$ clusters and two iterations for K-means clustering. Additionally, each data point has twenty-four features (or dimensions). We also run the K-means application with different workload sizes (e.g., 12GB and 20GB).
- TPC-H-Spark [10] is the implementation of the TPC-H decision support benchmark for running on Spark. The benchmark has twenty-two different queries executed on eight different input tables of various sizes. Different queries are concurrently performed on a large volume of data in parallel. TPC-H-Spark consists of a wide variety of database queries, including simple data selection, aggregation and multiple types of join operations. Thus, we use TPC-H-Spark as the representative benchmark for investigating query-based data processing on Spark. We run the TPC-H-Spark application on different datasets, ranging from 10GB to 50GB. To confirm the reproducibility of our results, we execute each workload multiple times.

B. Results and Analysis

*1) **Benefits of SSD**:* In this section, we investigate how to achieve the best performance using NVMe SSDs for Spark applications.

Generic Spark Applications - First, we run widely used generic Spark applications such as K-means, TeraSort, and FFT to assess NVMe SSD's performance by measuring I/O throughput using the blktrace tool. We find that the performance of these Spark applications improves by 1.2x-1.5x while using NVMe SSDs when compared to that of HDDs. Interestingly, the best throughput (i.e., 50MB/s - 60MB/s) that we obtain using NVMe SSDs in our experiments is significantly smaller than the achievable bandwidth (i.e., 160MB/s - 1.7GB/s) of these NVMe SSDs[1].

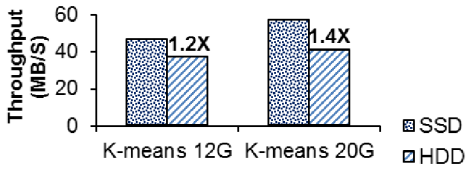

Fig. 3. K-means throughput with different workload sizes on NVMe SSD and HDD.

Fig. 3 shows the average throughput (MB/s) of the K-means clustering Spark application as a representative. We compare the throughput using HDD and SSD for different workload sizes (i.e., 12G, and 20G). We observe that even by increasing the workload size, the best throughput saturates below 60 MB/s. As we know, SSD is reported to obtain the throughput of more than hundreds of megabytes usually. However, we still cannot obtain such high throughput for the K-means application even after experimenting with different Spark application settings such as the number of executors, partition size and application workload size. Later in this section, we explain the potential root cause of such low SSD utilization while running generic Spark applications.

Database Applications - We next study online transaction processing (OLTP) applications such as TPC-C, TPC-E, and TPC-H. In particular, we present our evaluation results by using the TPC-H benchmark that provides a set of business-oriented queries and concurrent data modifications.

Fig. 4 shows the average throughput (MB/s) of TPC-H-Spark on NVMe SSD, compared to HDD (base case) with the workload size ranging from 10GB to 50GB. We can observe more throughput improvement (up to 2.4x) obtained by TPC-H-Spark than by K-means. We also observe that the TPC-H-Spark throughput using NVMe SSD now can reach around 115-140MB/s, which is almost in the range of the achievable throughput as mentioned in the NVMe SAMSUNG PM963's datasheet [12].

More importantly, the throughput by using NVMe SSD keeps increasing as the workload size increases. However, the

[1]Based on NVMe SAMSUNG PM963's datasheet [12], the bandwidth for randomized read is about 1.7GB/s and for write is 160MB/s.

Authorized licensed use limited to: FLORIDA INTERNATIONAL UNIVERSITY. Downloaded on October 24,2022 at 21:37:45 UTC from IEEE Xplore. Restrictions apply.

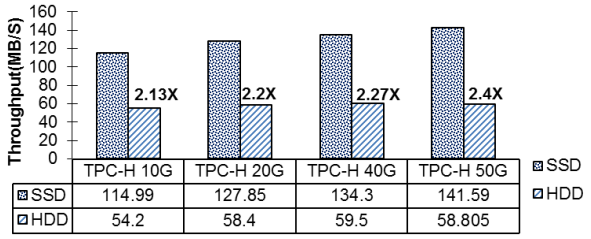

Fig. 4. TPC-H-Spark throughput with different workload sizes on NVMe SSD and HDD.

throughput of HDD has already reached saturation point (e.g., around 58MB/s in Fig. 4) and constantly remains the same while the workload size increases. These results indicate that for such database applications, HDD cannot help to obtain performance enhancement when workload size is increased (i.e., more I/O requests are submitted). On the other hand, NVMe SSD is more efficient to handle large-scale database applications and has the potential to improve throughput further as the workload gets larger. *We thus conclude that database applications like TPC-H-Spark can attain more benefits from NVMe SSD's high bandwidth, while generic Spark applications such as K-means cannot fully utilize SSD for performance improvement.* Next, we investigate the characteristics of these data processing applications and study the Spark platform related factors to understand performance better while using NVMe SSDs.

*2) **Shuffling Impacts on SSD Performance**:* To study the performance difference between K-means and TPC-H-Spark, we look closely into the Spark platform procedure that runs these applications. As mentioned in Sec. II, Spark consists of multiple stages, and each stage includes several tasks. The DAG scheduler is responsible for determining the execution of stages and tasks in Spark. In the remaining of this section, we discuss the shuffle and run time information across different classes of applications. The experiments in this section are measured using the Spark and Yarn Web User Interface (UI) information and the application execution log.

Shuffle Volume - When investigating the actual execution of K-means's stages and tasks, we notice that the majority of K-means's execution time is composed of computation time for allocating each data point to a cluster. We thus classify the K-means application as CPU intensive. Whereas, we find that the TPC-H-Spark application spends more time on processing I/Os for data queries and modifications, and thus consists of a large volume of I/Os across time. We then classify the TPC-H-Spark application as data intensive.

Specifically, TPC-H-Spark needs to transfer intermediate data (i.e., RDDs) from one stage to another, which is called the shuffling process. Shuffling is an essential component in Spark, which re-partitions and redistributes intermediate data across multiple tasks in the format of partitions. Obviously, such shuffling might cause moving data across different executors and thus launch intensive I/O requests [3]. For example, Table III shows the total volume of shuffling data under K-means and TPC-H-Spark with different workload sizes. We can see that even though the input data size is as large as 20GB, the total shuffle data volume of K-means is only around 1-2MB, which does not raise many I/O requests to the back-end storage. In contrast, TPC-H-Spark has a significant amount of shuffling data with the order of GB, and the total shuffle data volume grows fast as the size of the workload increases.

Based on these observations, *we conclude that the shuffle volume in Spark is one of the crucial factors in taking advantage of SSD high I/O bandwidth.* If there is a small amount of intermediate data for shuffling, then using SSD instead of HDD may not be helpful to improve I/O performance for such a Spark application. Whereas, deploying NVMe SSD becomes the most beneficial for those Spark applications (such as TPC-H for database queries) that contain a large shuffle data volume during their execution.

TABLE III
K-MEANS AND TPC-H-SPARK SHUFFLE VOLUME

Workload	Shuffle Volume
10 GB-TPC-H-Spark	14 GB
50 GB-TPC-H-Spark	70.6 GB
12 GB-K-means	0.92 MB
20 GB-K-means	1.5 MB

Run-time Measurements - Although the average throughput of TPC-H-Spark running on NVMe SSD is considerable, we still notice that the throughput is not consistently high but instead varying significantly during the execution. For example, Fig. 5 (a) and (b) shows the runtime TPC-H-Spark throughput during two monitoring windows. We observe that the throughput is fluctuating markedly, from 10 MB/s (e.g., in Fig. 5 (b) around 720 sec) to 80 MB/s (e.g., in Fig. 5 (b) around 650 sec).

To better understand the cause of such throughput inconsistency, we further measure the I/O volume rate occurred during Spark stages runtime by computing the I/O volume (i.e., stage input/output and shuffle read/write) divided by each stage execution time. The I/O volume measured here refers to the total size of I/Os observed by a Spark application, which is different from I/O throughput that presents the I/O volume observed at the disk level.

The corresponding I/O volume rate occurred during the two monitoring windows (shown in Fig. 5 (a) and (b)) are illustrated in Fig. 5 (c) and (d), respectively. We can observe similar patterns in these two sets of measurements (i.e., throughput and I/O volume rate during Spark stages). We also find that the low end of throughput or I/O volume rate happens when a limited number of tasks are running. The corresponding stages are actually the transfer stages that only pass a small amount of dependent data between two stages, without any computation or data processing. Thus, the amount of I/O volume rate and consequent throughput are decreased dramatically. When the following stages start to execute, the amount of I/O volume rate increases, consequently increasing the throughput because these stages usually have more concurrent tasks to perform

Authorized licensed use limited to: FLORIDA INTERNATIONAL UNIVERSITY. Downloaded on October 24,2022 at 21:37:45 UTC from IEEE Xplore. Restrictions apply.

computation or data processing. *We thus conclude that the number of tasks during each stage for CPU or data processing is another important factor that can affect the performance of Spark applications running on NVMe SSDs.*

*3) **I/O Access Behaviour of Database applications**:* In this section, we study the I/O characteristics of TPC-H-Spark, as an example of database applications. We analyze the distribution of I/O sizes and I/O access when using HDDs or NVMe SSDs.

I/O Size: HDD VS SSD - I/O size is another crucial measurement that needs to be analyzed for understanding the behavior of an application. In our evaluation, we observe that TPC-H-Spark's I/O sizes running on NVMe SSD are completely different from those running on HDD. As shown in Fig. 6, the size of both read and write I/Os exhibits a periodic pattern when using NVMe SSD. Large read/write I/Os are periodically clustered together, with some idle intervals between I/O size spikes. Such a periodic pattern in I/O size is more visible in reads than in writes. We also notice that more read requests are submitted than writes during the execution of TPC-H-Spark on NVMe SSD.

On the other hand, I/O sizes are uniformly distributed across time for both reads and writes of TPC-H-Spark on HDD, see Fig. 7. No periodic patterns are found in I/O sizes on HDD. Additionally, we observe that I/Os on HDD are much smaller than those on NVMe SSD, where the maximum I/O size on HDD is 60KB while the I/O size on NVMe SSD reaches to 250KB. These differences indeed arise from the NVMe SSD structure that has 64K queues in parallel. The internal controller manages asynchronous concurrent I/Os, which results in higher I/O throughput. When some workers submit I/O requests that need to access data in adjacent blocks, the driver layer of NVMe SSD can identify these I/Os and merge them together to process efficiently. As a result, the I/O size becomes larger, and the number of occurring I/Os decreases as well on NVMe SSD.

Distribution of I/O Access on SSD - Another main factor that can affect the SSD performance is the distribution of I/O access, especially the distribution of write I/O access. In our experiments, we use `blktrace` to collect the information of block layer I/Os and parse the disk offset of these I/Os in both HDD and NVMe SSD. Fig. 8 shows the results (i.e., disk offset) of the write I/Os on NVMe SSD when we run the TPC-H-Spark application with the 50G workload. We observe that most of the write I/Os are spatially located within the disk offset from 839GB to 1258GB and the rest of SSD space (totally 2TB) has very few or even no I/O access. Similar disk offset range can also be found for write I/Os on HDD.

The spatial locality is good for HDD to obtain high throughput by sequentially accessing data blocks. However, it is not beneficial to NVMe SSD because clustered data blocks might hinder parallel I/O access and degrade SSD endurance. As mentioned in Sec. II, HDFS is responsible for managing the distribution of data on the storage device. Our results indicate that the current HDFS mapping scheme is designed to cluster data blocks in the storage without the consideration of SSD parallelism and endurance. To address this issue, a new HDFS mapping scheme should be developed to map data across the entire SSD such that the throughput can be improved by accessing data in parallel and the endurance can be enhanced by evenly distributing writing on all blocks of an SSD.

C. Discussions

*1) **SSD as Extension of Physical Memory**:* SSD could be used as the main storage and at the same time be configured as an extension of the main memory through "swap". Swap is a space on the main storage that is used when the amount of physical RAM is full. When the system runs out of RAM, inactive pages are moved from the RAM to the swap space. By default, in computer systems the swap space is not present, but it can be configured as swap partition or swap file in the operating system. We initially believed that while using SSD as main storage, swap utilization will allow us to attain better performance. This is because the bandwidth gap between the physical memory and the HDD is much more significant than that between the physical memory and the SSD.

However, during our Spark platform setup, we realized that taking advantage of SSD as a swap was not possible when Spark is running on multiple VM nodes. The VMware server setup requires to set the maximum allowable memory size of each VM and the amount of the host's memory that can be used for VMs. The total memory of all currently running VMs is limited and cannot exceed the capacity of physical memory on the host. Interestingly, from our experiments, we explore that VMware only allows the VM's swap space to be mapped to the physical host's memory instead of the physical swap or disk space. Thus, in-spite of configuring our VM image with 10GB VM swap space and physical servers with 128GB GB swap, we observe that when VM reaches the allocated memory limit it does not utilize physical swap space. Consequently, it becomes impossible to use the storage (e.g., SSD) as the swap space when VMs run out of memory. Also, such inherent properties of VMware server restrict the number of VMs on the same physical machine because the physical RAM needs to be partitioned among all the VMs.

*2) **Processing Parallelism in Spark**:* Exploiting available parallelism is important to achieve optimal performance. Spark performs parallel data processing by managing the number of simultaneous executors. Executors are processes on each worker node to run individual tasks in a given Spark job. Executors are launched at the beginning of a Spark application and typically run for the entire lifetime of that Spark application. In our evaluation, we further analyze SSD performance with a different number of executors. Fig. 9 shows the execution time of K-means running on HDD and NVMe SSD when we set the different number of executors in the Spark platform. We observe that upon increasing the number of executors, the performance improves until the specific number of executors is reached. Since then, the performance remains the same while further increasing the number of executors. We investigate that in this case, the optimal number of executors is eight because

Authorized licensed use limited to: FLORIDA INTERNATIONAL UNIVERSITY. Downloaded on October 24,2022 at 21:37:45 UTC from IEEE Xplore. Restrictions apply.

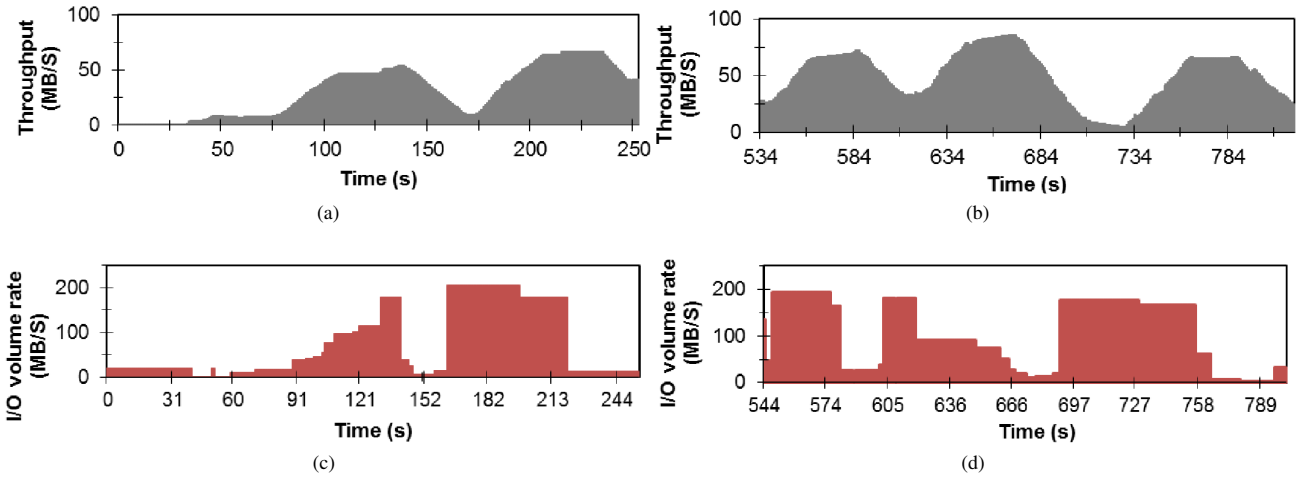

Fig. 5. Runtime throughput and I/O volume rate issued by Spark stages under TPC-H-Spark of 10GB workload during two monitoring windows.

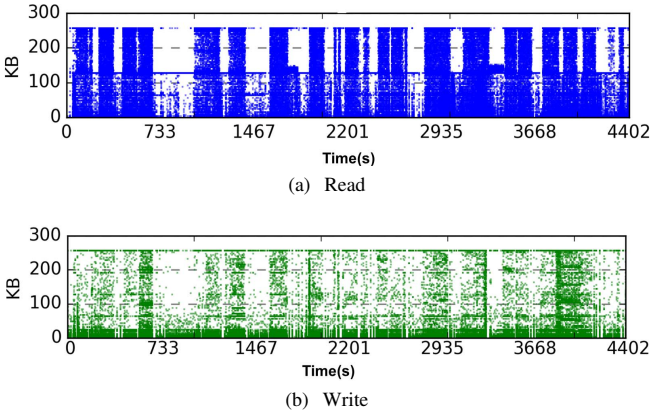

Fig. 6. (a) Read and (b) write I/O size of TPC-H-Spark with 50G workload on NVMe SSD.

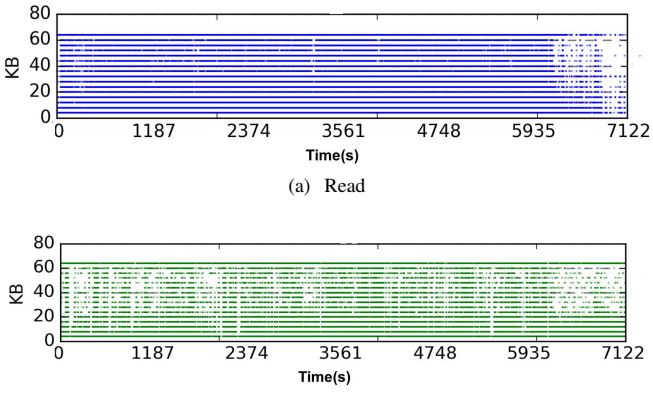

Fig. 7. (a) Read and (b) write I/O size of TPC-H-Spark with 50G workload on HDD.

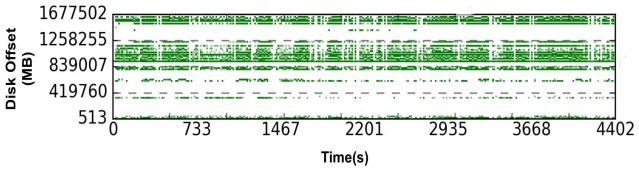

Fig. 8. Disk offset of write I/O access on NVMe SSD under TPC-H-Spark with 50G workload.

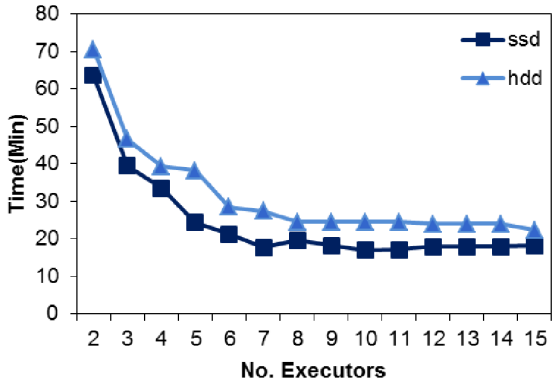

Fig. 9. Optimum level of parallelism in SSD and HDD (1 core per VM).

the platform is configured with one master and seven workers and each worker is set with one core.

IV. RELATED WORK

Deploying high throughput storage system is essential for processing data-intensive applications at the application layer. Originally, since $/GB of flash drives was considerable, SSDs were deployed as bufferpool. Bufferpool caches data between RAM and hard disk [13], [14], [15]. Whereas, as the SSD's cost of ownership decreases, SSD has been used to substitute

Authorized licensed use limited to: FLORIDA INTERNATIONAL UNIVERSITY. Downloaded on October 24,2022 at 21:37:45 UTC from IEEE Xplore. Restrictions apply.

HDD as the primary storage device [16]. In addition, [17] deployed NVMe to meet the storage massive need for a high-performance SSD by communicating over PCIe. Previous studies [18], [19] investigated the SSD and NVMe resource management and discussed the issues related to the reduction of the total cost of ownership and the utilization of Flash device. Different techniques were proposed to take advantage of SSD throughput. For example, [20] used hardware support to expand parallelism inside the SSD device.

In recent years, SSDs have been widely adopted in data centers and big data platforms. In [21], [22], authors explored the performance of SSDs as main storage for database applications. Studies [23], [24] focused on how to fully utilize SSD resources in big data and cloud computing platforms. According to [25], there are many sources of inefficiency in Spark especially during the shuffle phase that can affect Spark application performance. [26] investigated how the modification of block manager during the shuffle time and Yarn resource management can benefit Spark performance. In this paper, we focus on investigating the deployment of NVMe SSD in the Spark platform when different Spark data processing applications are executing. We aim to understand the underlying causes and factors that can help to utilize the benefits of NVMe SSD for improving the performance of data processing applications in Spark.

V. Conclusions and Future Work

In this paper, we explore the NVMe SSD benefits on big data environments during the execution of data processing applications. Based on our experimental results, we found that data processing applications even with large workloads cannot always ensure to make the best use of NVMe SSD devices to achieve high performance. We categorized data processing applications into two classes - generic Spark applications and database applications and observed that database applications such as TPC-H-Spark can attain more benefit from NVMe SSDs comparing to generic Spark application like K-means. We further observed that TPC-H-Spark's high I/O throughput is essentially the outcome of handling the large volume of shuffling data through NVMe SSDs. Our analysis also revealed that the Spark runtime parallelism plays another important role in obtaining the full benefit of NVMe SSDs. We finally conclude that deploying NVMe SSD is not critically necessary for all data processing applications with large scaled workloads. The characteristics of application design and Spark platform should be taken into account for deploying and utilizing NVMe SSDs. In our future work, we plan to adjust the mapping of HDFS to meet SSD structural need for obtaining better performance and also plan to investigate big data processing behaviors in a containerized environment.

Acknowledgements

This work was partially supported by National Science Foundation Career Award CNS-1452751, and Samsung Semiconductor Inc. Research Grant.

References

[1] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," vol. 51, no. 1. ACM, 2008, pp. 107–113.

[2] "Hadoop installation guide," https://www.alexjf.net/blog/distributed-systems/hadoop-yarn-installation-definitive-guide/.

[3] A. G. Shoro and T. R. Soomro, "Big data analysis: Apache spark perspective," *Global Journal of Computer Science and Technology*, 2015.

[4] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets." *HotCloud*, vol. 10, no. 10-10, p. 95, 2010.

[5] "NVM Express," https://nvmexpress.org/.

[6] S. Moon, J. Lee, X. Sun, and Y.-s. Kee, "Optimizing the hadoop mapreduce framework with high-performance storage devices," *The Journal of Supercomputing*, vol. 71, no. 9, pp. 3525–3548, Sep 2015. [Online]. Available: https://doi.org/10.1007/s11227-015-1447-3

[7] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth *et al.*, "Apache hadoop yarn: Yet another resource negotiator," in *Proceedings of the 4th annual Symposium on Cloud Computing*. ACM, 2013, p. 5.

[8] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system," in *Mass storage systems and technologies (MSST), 2010 IEEE 26th symposium on*. Ieee, 2010, pp. 1–10.

[9] "SparkBench," https://sparktc.github.io/spark-bench/.

[10] "TPC-H-SPARK Benchmark," https://github.com/ssavvides/tpch-spark.

[11] U. R. Raval and C. Jani, "Implementing and improvisation of k-means clustering," *Int. J. Comput. Sci. Mob. Comput*, vol. 5, no. 5, pp. 72–76, 2016.

[12] "Samsung Datasheet NVMe-PM963," https://www.hammer-drive.com/assets/uploads/downloads/drive-selector/_OLD/PM963.pdf.

[13] M. Canim, G. A. Mihaila, B. Bhattacharjee, K. A. Ross, and C. A. Lang, "Ssd bufferpool extensions for database systems," *Proceedings of the VLDB Endowment*, vol. 3, no. 1-2, pp. 1435–1446, 2010.

[14] L.-P. Chang, "Hybrid solid-state disks: combining heterogeneous nand flash in large ssds," in *Proceedings of the 2008 Asia and South Pacific Design Automation Conference*. IEEE Computer Society Press, 2008, pp. 428–433.

[15] H. Jo, Y. Kwon, H. Kim, E. Seo, J. Lee, and S. Maeng, "Ssd-hdd-hybrid virtual disk in consolidated environments," in *European Conference on Parallel Processing*. Springer, 2009, pp. 375–384.

[16] A. Leventhal, "Flash storage memory," *Communications of the ACM*, vol. 51, no. 7, pp. 47–51, 2008.

Authorized licensed use limited to: FLORIDA INTERNATIONAL UNIVERSITY. Downloaded on October 24,2022 at 21:37:45 UTC from IEEE Xplore. Restrictions apply.

[17] T. Y. Kim, D. H. Kang, D. Lee, and Y. I. Eom, "Improving performance by bridging the semantic gap between multi-queue ssd and i/o virtualization framework," in *2015 31st Symposium on Mass Storage Systems and Technologies (MSST)*. IEEE, 2015, pp. 1–11.

[18] J. Bhimani, J. Yang, Z. Yang, N. Mi, Q. Xu, M. Awasthi, R. Pandurangan, and V. Balakrishnan, "Understanding performance of i/o intensive containerized applications for nvme ssds," in *2016 IEEE 35th International Performance Computing and Communications Conference (IPCCC)*, Dec 2016, pp. 1–8.

[19] Z. Yang, M. Awasthi, M. Ghosh, and N. Mi, "A fresh perspective on total cost of ownership models for flash storage in datacenters," in *2016 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, Dec 2016, pp. 245–252.

[20] Y. J. Yu, D. I. Shin, W. Shin, N. Y. Song, J. W. Choi, H. S. Kim, H. Eom, and H. Y. Yeom, "Optimizing the block i/o subsystem for fast storage devices," *ACM Transactions on Computer Systems (TOCS)*, vol. 32, no. 2, p. 6, 2014.

[21] Y. Wang, K. Goda, M. Nakano, and M. Kitsuregawa, "Early experience and evaluation of file systems on ssd with database applications," in *2010 IEEE Fifth International Conference on Networking, Architecture, and Storage*. IEEE, 2010, pp. 467–476.

[22] D. Narayanan, E. Thereska, A. Donnelly, S. Elnikety, and A. Rowstron, "Migrating server storage to ssds: analysis of tradeoffs," in *Proceedings of the 4th ACM European conference on Computer systems*. ACM, 2009, pp. 145–158.

[23] Z. Yang, M. Hoseinzadeh, P. Wong, J. Artoux, and D. Evans, "A hybrid framework design of nvme-based storage system in cloud computing storage system," *Patent US*, vol. 62, p. 540555, 2017.

[24] Z. Yang, T. D. Evans, and J. Wang, "Adaptive caching replacement manager with dynamic updating granulates and partitions for shared flash-based storage system," Mar. 8 2018, uS Patent App. 15/400,835.

[25] A. Davidson and A. Or, "Optimizing shuffle performance in spark," *University of California, Berkeley-Department of Electrical Engineering and Computer Sciences, Tech. Rep*, 2013.

[26] N. Chaimov, A. Malony, S. Canon, C. Iancu, K. Z. Ibrahim, and J. Srinivasan, "Scaling spark on hpc systems," in *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*, ser. HPDC '16. New York, NY, USA: ACM, 2016, pp. 97–110. [Online]. Available: http://doi.acm.org/10.1145/2907294.2907310

Authorized licensed use limited to: FLORIDA INTERNATIONAL UNIVERSITY. Downloaded on October 24,2022 at 21:37:45 UTC from IEEE Xplore. Restrictions apply.