Proceedings of

, , ,

OVERCOMING THE MEMORY BOUND OF BIG DATA ANALYTICS TO IMPROVE SERVER THROUGHPUT USING FAST STORAGE

Emmanouil Anagnostakis
Graduate Research Assistant
Computer Architecture and VLSI Systems Laboratory, ICS-FORTH
Heraklion, Greece
manosanag@ics.forth.gr

Abstract

Managing big data analytics i.e. Apache Spark poses challenges due to limited memory resources in data centers. The memory pressure that arises during data processing can result in low server throughput, causing delays and inefficiencies. Memory is wasted in long GC (Garbage Collection) cycles leaving no room for useful work. In this paper, we propose a novel approach to improve server throughput for managed big data analytics using smart heap offloading to fast storage devices and reducing memory pressure. Our approach involves offloading data from heap memory to fast storage devices in a smart and efficient manner, thereby freeing up heap memory and reducing memory pressure without suffering from storage latencies. By overcoming the memory bound we leave space to the applications for more CPU utilization. We present a detailed methodology for running Apache Spark using our proposed mechanism of smart heap offloading, which significantly improves server throughput for managed big data analytics. Our proposed approach is implemented in Oracle's OpenJDK8 and in this paper we evaluate its performance against Native Spark using various workloads of the Spark Bench suite on a real-world cluster, focusing on examining the performance under the colocation of multiple instances. Our experimental results show that our approach significantly improves server throughput while reducing memory usage against native Spark, making it a promising solution for managed big data analytics in data centers. We also include results to show that our implementation can reduce money cost if deployed in a world cluster like Amazon's EC2 or Google Cloud Platform or Microsoft Azure Cloud which are available to everyone.

1. Introduction

With the exponential growth of data in various fields such as finance, healthcare, social media, and e-commerce, there is a significant need for scalable and efficient big data processing frameworks. Apache Spark [?] is one such framework that has gained popularity due to its ability to handle large-scale data processing and analytics. Spark provides a distributed computing platform that can process data in parallel across multiple nodes in a cluster. However, with the increasing size and complexity of big data workloads, Spark clusters are facing significant challenges in meeting the performance and throughput requirements.

One of the main challenges in Spark clusters is the high computational and memory requirements of big data analytics workloads. These requirements can result in excessive CPU and memory usage on Spark workers, leading to performance bottlenecks and slow job completion times. To address these challenges, researchers have proposed various techniques to optimize the performance of Spark clusters, including data partitioning, caching, and resource allocation.

In this paper, we focus on the memory limit problem of servers becoming an obstacle for further throughput increase and we propose a new technique for improving the performance and job throughput of Spark clusters by moving long-lived objects from the main managed Java Heap to a fast storage device such as NVMe, thereby saving memory for other more useful tasks and leaving space for more CPU utilization. Our approach leverages the capabilities of the underlying machine to create less memory-consuming computation tasks, thereby reducing the workload on the Spark workers, while maintaining effective per-executor performance under the colocation of multiple executors required to achieve max throughput.

Specifically, in order to achieve higher throughput and better performance for Spark, we use TeraHeap, a secondary managed memory-mapped heap over an NVMe storage device, which is used to hold the Resilient Distributed Datasets (Spark RDDs) instead of the main managed Java Heap and completely remove any Serialization/Deserialization and Garbage Collection (GC) cost over them.

TeraHeap 1) eliminates Serialization/Deserialization overheads posed by this kind of frameworks when moving data off-heap to/from fast storage devices 2) eliminates GC pauses over the secondary heap, therefore significantly minimizing overall GC overhead. By offloading the managed Java Heap and relaxing computation-intensive tasks, we aim to reduce the workload on Spark workers, thereby improving their performance and job throughput. We also explore the trade-offs between the cost of offloading and the performance gains achieved.

We demonstrate the effectiveness of our approach using various big data analytics workloads on a real-world Spark cluster. We also compare our approach with the native Spark distribution and show that our approach can be used instead of this distribution to improve performance and server throughput.

The paper makes the following contributions 1) A comprehensive evaluation of the performance and cost trade-offs of a proposed heap offloading technique that utilizes fast storage devices to improve server throughput. 2) A detailed methodology for running Apache Spark and understanding how to use it for conducting off-heap experiments using the Native or our proposed approach 3) Proves that heap offloading to fast storage devices is a significant research direction.

The rest of the paper is organized as follows. In section 2 we discuss related work on Spark optimization techniques and offloading techniques. In section 3, we describe our experimental methodology in order for someone to achieve the desired performance using TeraHeap. In section 4, we present our experimental results and evaluate the performance and cost trade-offs of our approach. In section 5, we discuss future research directions. Finally we conclude the paper in section 6 with an outline of our work.

2. Related Work

Several studies have been conducted to improve the performance of big data processing systems. One approach is to utilize memory-aware task co-location to improve Spark application throughput, which has been investigated by Marco et al. in [3]. Meanwhile, in [4], Kirisame et al. proposed optimal heap limits to reduce browser memory use. Another research direction is to leverage far memory to improve job throughput, as studied by Amaro et al. in [5]. To facilitate memory offloading in datacenters, Weiner et al. presented TMO, a transparent memory offloading system in [6]. In cloud computing platforms, Sharma et al. proposed per-VM page cache partitioning to improve performance in [7]. Chen and Wang introduced Spark on Entropy, a reliable and efficient scheduler for low-latency parallel jobs in heterogeneous clouds, in [8]. Thamsen et al. developed Mary, Hugo, and Hugo*, three learning-based schedulers for distributed data-parallel processing jobs on shared clusters in [9]. Additionally, Bhimani et al. proposed a lightweight virtualization framework for accelerating big data applications on enterprise cloud in [10], while Zhang et al. focused on understanding and improving disk-based intermediate data caching in Spark in [11]. Finally, Intasorn et al. investigated using compression tables to improve HiveQL performance with Spark in a case study on NVMe storage devices in [12].

These studies demonstrate a variety of approaches for optimizing big data processing systems, ranging from memory-aware task co-location and memory offloading to scheduler design and virtualization frameworks. The findings from these studies can provide insights and guidance for future research in the field of big data processing.

3. Experimental Methodology

To evaluate the effectiveness of our proposed approach, we conduct a set of experiments using various workloads from the Spark Bench suite on a real-world cluster. Our experimental methodology consists of several steps. First, we set up our research server, using various configurations of datacenter machines, including CPU, memory, and storage. Next, we install and configure Spark and OpenJDK8 on the cluster. We use Spark's default configuration settings for our experiments, except for the garbage collector settings, which we tune according to our proposed approach.

We select four workloads from the Spark Bench suite that represent different types of data processing, such as machine learning and graph processing, (and SQL queries ???). We run each workload using our proposed approach and compare it with the performance of the same workload using the default configuration, garbage collector tuning, and heap offloading approaches.

We utilize all the available DRAM provided by our server leaving 8-10 GB for the Operating System, while increasing the number of Spark instances that are executed. We show that by using TeraHeap, each individual instance requires less memory therefore memory becomes available for more instances to be deployed, achieving more total throughput than Native Spark in similar time windows.

To measure the performance of our approach, we use several metrics, including server throughput, memory usage, and execution time. We measure server throughput as the megabytes of the dataset processed per second, memory usage as the amount of memory used during data processing, and execution time as the time taken to complete the workload.

We repeat each experiment several times to ensure statistical significance and calculate the mean and standard deviation of the metrics for each approach.

In summary, our experimental methodology involves setting up a real-world cluster, selecting appropriate workloads, measuring performance using several metrics, and repeating each experiment several times to ensure statistical significance. By following this methodology, we can evaluate the effectiveness of our proposed approach for improving server throughput for managed big data analytics.

3.1 Server Characteristics

The server used in our experiments is a high-performance machine with hardware specifications found in real-world clusters like Amazon EC2. It is equipped with 8x DDR4 32-GB 2.4 GHz 64-bit DIMMs, providing a total of 256 GB of memory. The DDR4 memory technology is known for its high bandwidth and low power consumption, making it ideal for data-intensive applications like big data analytics. The server also features 32x Intel Xeon E5-2630 2.4 GHz 64-bit CPUs, divided into 2 NUMA islands, each with 512 KB L1, 2 MB L2, and 20 MB L3 (LLC) cache. The Xeon E5-2630 CPU is a high-performance processor designed for data centers, offering a high core count, high clock speed, and advanced features like hyper-threading and Turbo Boost. The large L3 cache helps reduce memory latency, enabling faster data access for CPU-bound workloads. In addition to the powerful CPUs and memory, the server also has 2x KVS NVMe storage devices. NVMe is a high-performance storage technology that uses PCIe to connect directly to the CPU, providing low latency and high throughput. The KVS (Key-Value Store) storage devices are designed for fast, random access to data, making them ideal for storing and retrieving large amounts of data in big data applications. Overall, the server's hardware specifications make it a powerful platform for conducting experiments on managed big data analytics and evaluating the performance of our proposed approach.

3.2 Native Spark Configuration

We use Spark v3.3.0 with Kryo Serializer, a state-of-the-art highly optimized S/D Library for Java that Spark recommends. We run Spark with Native OpenJDK8 as a baseline. We use the Parallel Scavenge garbage collector which is the one TeraHeap is implemented for. Parallel Scavenge is also the go-to collector for applications that need high throughput like Spark. We use an executor with eight mutator threads for each instance of Spark we deploy on our server. For Parallel Scavenge, we use 8 GC Threads for minor GC and the default single-threaded old generation GC. Table XXX summarizes the Spark configuration we use as baseline. Spark uses the MEMORY-AND-DISK storage level to place executor memory (heap) in DRAM and cache RDDs in the on-heap cache, up to 50% of the total heap size. Any remaining RDDs are serialized in the off-heap cache over an NVMe SSD. This device is also used by Spark for shuffling. We run each instance of Spark in a cgroup containing two JVM instances, one for Spark driver and one for Spark executor and all the processes needed to measure performance for this instance. Each cgroup has a limited DRAM Budget. A part of the budget is the capacity of the Java Heap which, for the rest of the paper, we call H1. We do this in order to be sure that every instance of Spark running on our server has a fair amount of DRAM available for it to use. We choose to try two different amounts for H1, 40% and 80% of total DRAM budget. 80% is the go-to percentage of total DRAM RedHat uses in its datacenters [?]. What remains is used by JVM for Native memory (i.e. CodeCache) and for the operating system's Page Cache. In order to avoid inter-NUMA island interference we shut down the 16 cores belonging to the second island thus leaving 16 active cores. We also turn off the swapper, because it adds significant overhead and makes it difficult to understand the results of the experiments conducted.

3.3 TeraHeap

3.3.1 What is TeraHeap?

TeraHeap is a high-capacity managed heap that is memory-mapped over a fast storage device (preferrably block-addresable NVMe or byte-addresable NVM). The high speeds these kind of devices operate in, erase any overhead caused by the use of MMIO. TeraHeap is designed as an extension of the main Java Heap. It holds specific long-lived objects that have the same lifetime span. This enables TeraHeap to operate as a GC-free heap that can delete entire regions of objects at once without a need to scan the heap over and over again for dead objects, which would be a performance kill as it would require scans over the storage device. The two main contributions of TeraHeap are the following: 1) MMIO keeps the objects that reside in the storage device deserialized, thus eliminating the need for Serialization/Deserialization, which is the no 1 overhead when running MEMORY-AND-DISK Spark 2) As discussed, TeraHeap reduces GC overheads without wasting DRAM by avoiding scans on long-lived objects, specificaly Spark RDDs.

3.3.2 Spark Configuration

The configuration for TeraHeap is pretty much the same as for Native Spark, with some necessary differences to achieve our goal. TeraHeap is mapped to a different storage device (NVMe) than that Spark is using for shuffling. We do this in order for TeraHeap to utilize its device to its fullest. MMIO allows TeraHeap Spark to run in MEMORY-ONLY storage level as Spark is unaware of using any device and the OS takes control of the I/O. We also make the same decisions for the DRAM budget trying different amounts for H1, 40% and 80% of total DRAM budget. By doing that, we have different configurations where H1 dominates PageCache and the reverse showing what the needs of Spark applications are.

3.4 What workloads did we choose to use for our experiments and why?

For our experiments, we selected four specific workloads from two different categories of the Spark Bench suite: Page Rank and Connected Component from GraphX and Linear Regression and Logistic Regression from MLLib. The primary reason for selecting these workloads is that they represent different types of big data analytics tasks: PageRank and Connected Component are graph-based workloads, while LinearRegression and LogisticRegression are machine learning workloads. By selecting workloads from both categories, we can investigate the performance of our proposed approach across a range of big data analytics tasks. Furthermore, all of them are well-established workloads that are commonly used for benchmarking big data analytics systems, making them a suitable choice for our experiments. Overall, the selection of these workloads allows us to evaluate the performance of our approach in a variety of contexts and provide insights into the effectiveness of our approach for improving server throughput in managed big data analytics systems.

3.4.1 PageRank PageRank is a widely used graph-based algorithm that measures the importance of nodes in a network. It has become a popular benchmark for evaluating the performance of distributed systems, including big data analytics systems like Apache Spark. PageRank is computationally intensive and requires significant memory and I/O resources, making it a suitable workload for evaluating the performance of our proposed approach for improving server throughput. Additionally, PageRank is a common algorithm in real-world applications, such as search engines and social networks, making it relevant for practical use cases.

3.4.2 LinearRegression LinearRegression is a machine learning algorithm that is used to predict numerical values based on input data. It is a well-known and widely used algorithm in machine learning, and is commonly used for regression analysis in fields such as economics, finance, and engineering. LinearRegression is computationally intensive and requires significant memory and I/O resources, making it a suitable workload for evaluating the performance of our proposed approach for improving server throughput. Furthermore, the inclusion of a machine learning workload like LinearRegression allows us to investigate the performance of our approach across different types of big data analytics tasks and gain insights into the effectiveness of our approach for improving server throughput in a range of contexts.

3.4.3 Logistic Regression LogisticRegression is a machine learning algorithm that is used to model the probability of a binary or categorical outcome based on one or more independent variables. It is commonly used in predictive analytics to classify data based on historical data. In Spark-bench, LogisticRegression is implemented as a machine learning workload, where the dataset is represented as an RDD of feature vectors and labels. The LogisticRegression workload involves training a logistic regression model on the dataset, using an iterative optimization algorithm such as gradient descent. The workload is computationally intensive and requires a significant amount of memory to store the dataset and model parameters.

3.4.4 Connected Component ConnectedComponent is a graph algorithm that is used to identify the connected components of a graph. It is commonly used in social network analysis to identify clusters of users with similar interests or relationships. In Spark-bench, ConnectedComponent is implemented as a graph processing workload, where the graph is represented as an RDD of edges and vertices. The ConnectedComponent workload involves iterating over the graph, identifying the connected components of each node, and merging the components as necessary. The workload is computationally intensive and requires a significant amount of memory to store the graph.

3.5 Is Spark in need of Java Heap or more cache for I/O?

Spark's memory management is critical for the performance of big data analytics applications. In Spark, memory is divided into three regions: heap memory, execution memory, and storage memory. The heap memory is used for JVM objects and the Spark driver. Execution memory is used for storing data during shuffle and join operations and for caching frequently accessed data. Storage memory is used for storing data that is too large to fit in the execution memory. Spark's memory management system automatically manages the memory allocation and deallocation process based on the workload and available resources. This dynamic allocation helps to avoid memory fragmentation and optimize resource utilization. Additionally, Spark uses a combination of in-memory and disk-based storage to provide efficient data access. Spark provides various storage levels, including MEMORY-ONLY, MEMORY-AND-DISK, and DISK-ONLY, to allow users to balance between memory usage and data availability. We choose MEMORY-AND-DISK to cache 50% of the RDDs in memory and 50%s off-heap, in the storage device, to balance memory and storage usage. Spark needs significant amounts of memory even with the use of an off-heap compute cache. Using off-heap mechanisms denotes I/O and I/O is in need of an I/O cache i.e. the Linux PageCache. Since our spark applications run within a memory-limited cgroup in order to assure fair performance in-between instances, that means that we have to investigate how the different Spark workloads that we are going to use for our evaluation are going to perform with different

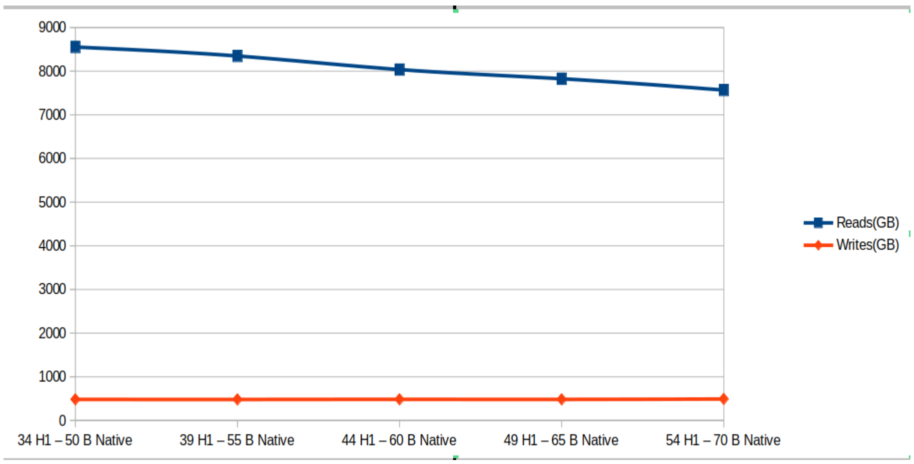

FIGURE 1. Number of GCs over time for H1 Linear Regression Native Spark investigation.

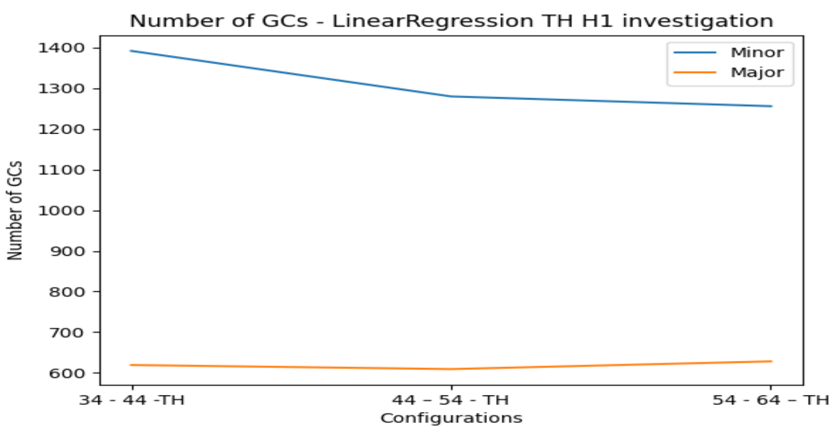

FIGURE 2. Number of GCs over time for H1 Linear Regression TeraHeap Spark investigation.

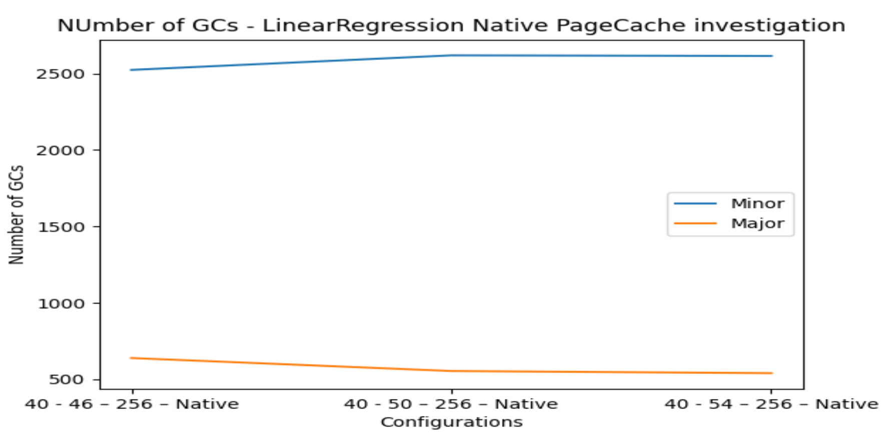

FIGURE 3. Number of GCs over time for Page Cache Linear Regression Native Spark investigation.

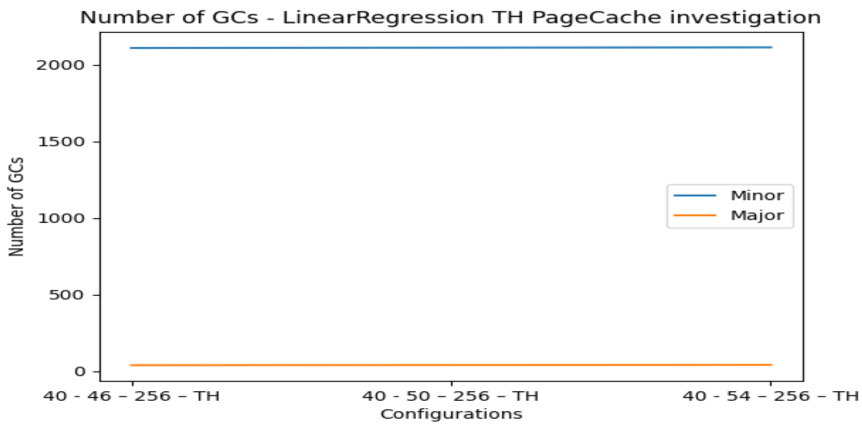

FIGURE 4. Number of GCs over time for Page Cache Linear Regression TeraHeap Spark investigation.

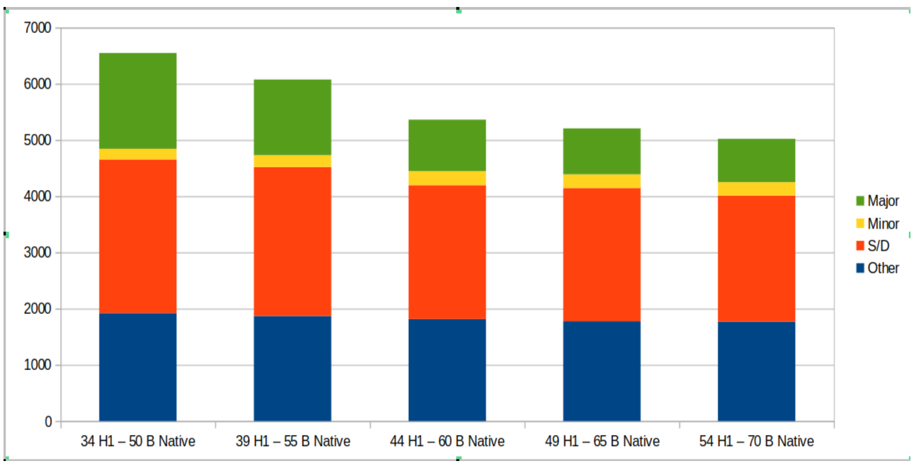

FIGURE 5. Execution time breakdown for H1 Linear Regression Native Spark investigation.

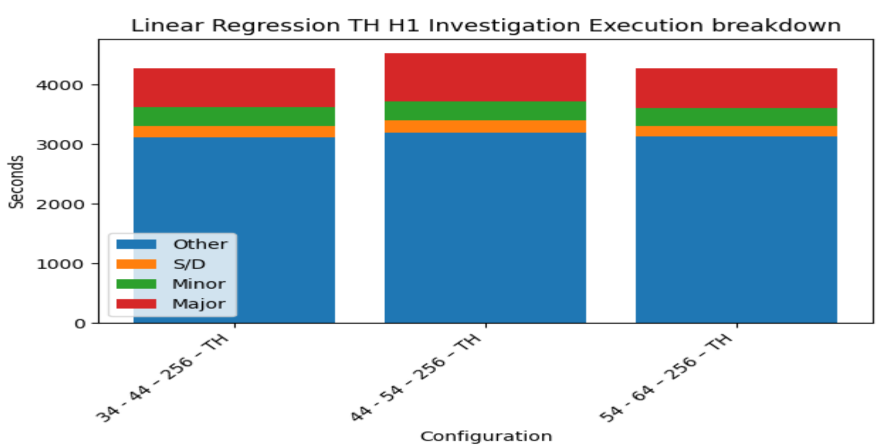

FIGURE 6. Execution time breakdown for H1 Linear Regression TeraHeap Spark investigation.

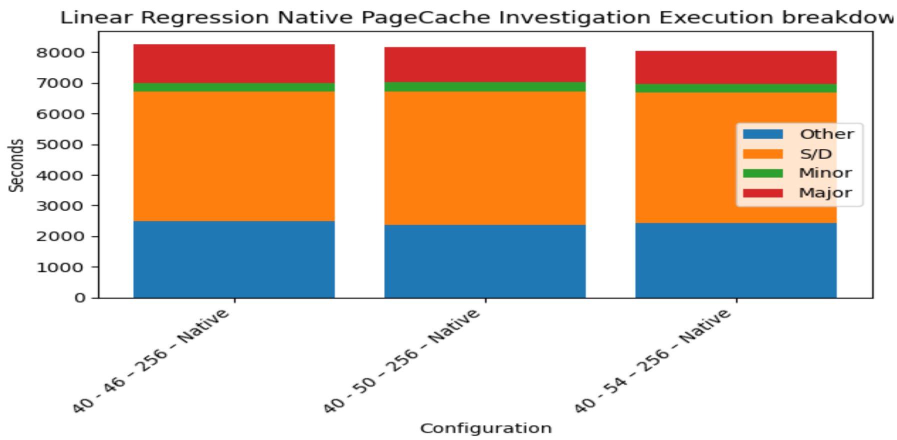

FIGURE 7. Execution time breakdown for Page Cache Linear Regression Native Spark investigation.

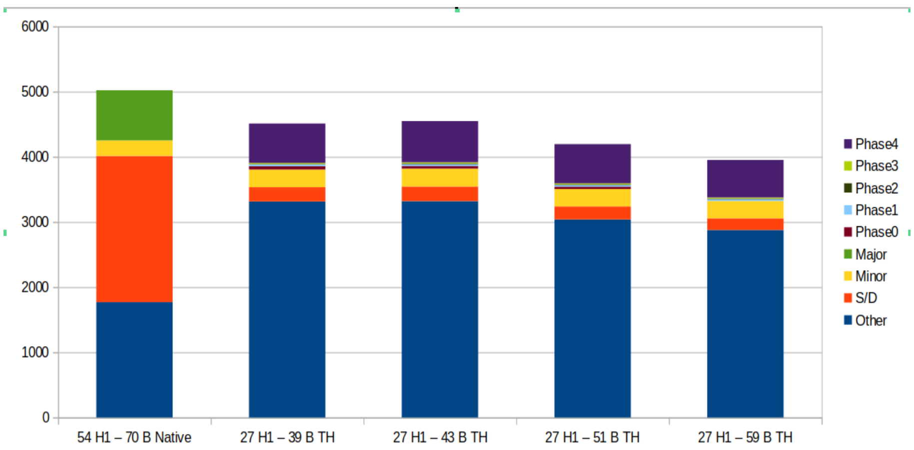

FIGURE 8. Execution time breakdown for Page Cache Linear Regression TeraHeap Spark investigation.

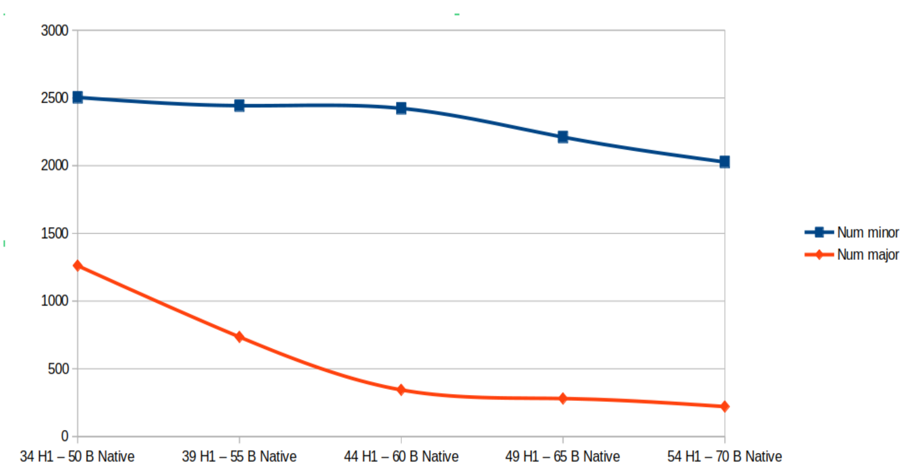

FIGURE 9. Read-Write traffic over time for H1 Linear Regression Native Spark investigation.

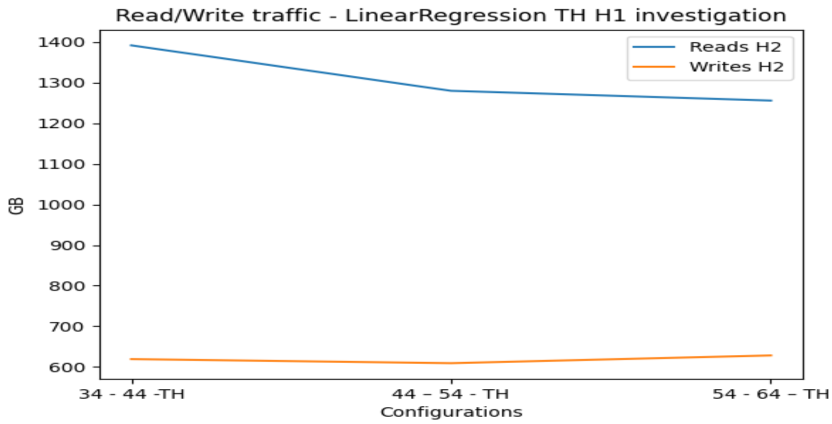

FIGURE 10. Read-Write traffic over time for H1 Linear Regression TeraHeap Spark investigation.

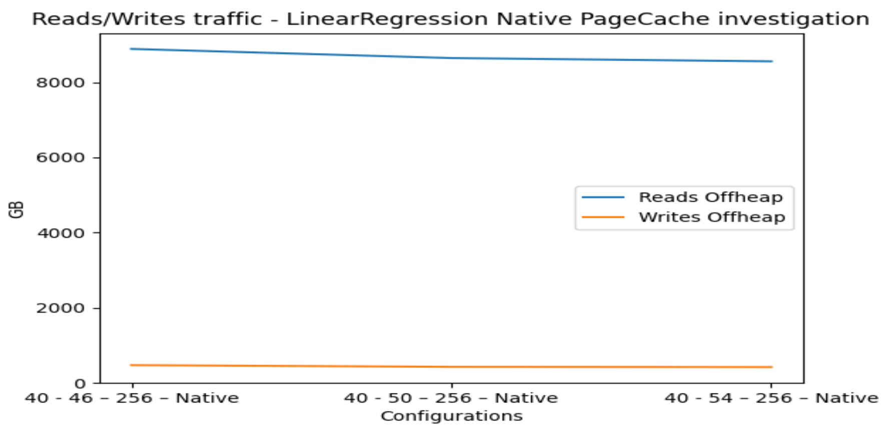

FIGURE 11. Read-Write traffic over time for PageCache Linear Regression Native Spark investigation.

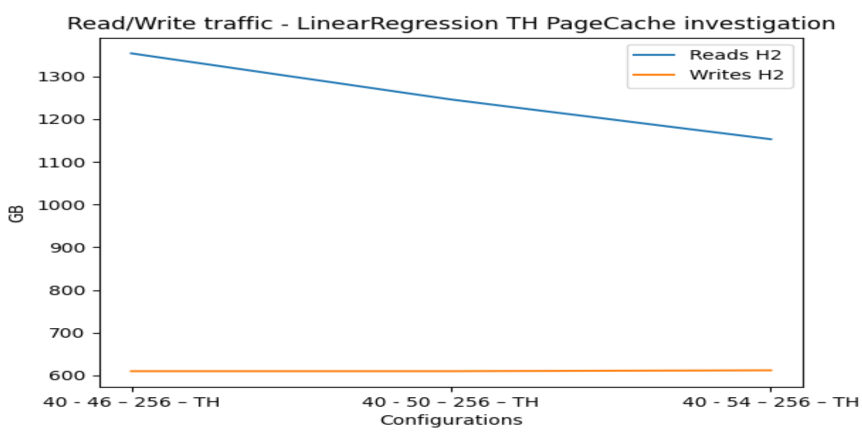

FIGURE 12. Read-Write traffic over time for PageCache Linear Regression TeraHeap Spark investigation.

amounts of H1 (Java Heap) and I/O cache. Increasing/decreasing H1 automaticaly does the opposite to the I/O cache so the mechanism is simple: we manually tune these parameters and investigate how the workloads perform. At first we investigate the performance of TeraHeap Spark regarding these two parameters. Our machine uses 256 GB total DRAM and 16 single-NUMA island CPUs. Figure 5 shows performance of single-instance Native Spark running LinearRegression with adjustable size for H1 while PageCache is kept steady at 10 GB. This graph shows that decreasing the size of H1 indicates a significant increase to Major GC and a slight increase to S/D. Other time remains the same. Figure 1 and 9 justify these numbers by showing the number of minor-major gcs to decrease while H1 increases (gc time) and the read-write traffic to remain steady (other). Figure 8 shows performance of single-instance TeraHeap Spark running LinearRegression with adjustable size for PageCache while H1 is kept steady at 40 GB. This graph shows that decreasing the size of PageCache indicates significant changes to Other time. GC time should remain the same as H1 remains steady. Figure 4 and 12 justify these numbers by showing the number of minor-major gcs to remain the same while read traffic decreases as PageCache increases. Figure 6 shows performance of single-instance TeraHeap Spark running LinearRegression with adjustable size for H1 while PageCache is kept steady at 10 GB. This graph shows that decreasing the size of H1 indicates no increase to any stat. Figure 2 and 10 justify these numbers by showing the number of major gcs to stay the same while H1 increases (gc time) and the read-write traffic to remain steady (other). Figure 8 shows performance of single-instance TeraHeap Spark running LinearRegression with adjustable size for PageCache while H1 is kept steady at 40 GB. This graph shows that decreasing the size of PageCache indicates no changes to Other time. So changes to PageCache do not affect this workload. Figure 4 and 12 justify these numbers by showing the number of minor-major gcs as well as read-write traffic to remain the same.

3.6 What kind of metrics should someone use to be accurate when measuring performance?

When measuring performance, it's important to choose metrics that provide a comprehensive view of the system's behavior. In the case of measuring the performance of Spark instances, there are several key metrics that one should consider. These include heap capacity, which is the amount of memory allocated to the Java Virtual Machine (JVM) running Spark, and total memory used by the instance, which is the actual amount of memory consumed by the Spark instance, as measured by the cgroup budget. GC time is also an important metric, as it measures the amount of time spent by the JVM garbage collector in freeing up memory. Serialization/deserialization time, measured using a Java async profiler, is important for understanding how much time is spent in this operation, which can be a bottleneck for some workloads. Other time, which is simply the difference between total time and GC and serialization/deserialization time, can provide insight into other factors that may be affecting performance, but mainly includes the time spent in I/O and also the time spent by mutator threads to run the application code. Device traffic, measured using iostat, is important for understanding how much data is being read from and written to storage devices. CPU idle and IO wait, measured using mpstat, can help identify how much of the CPU and I/O resources are being utilized. Finally, average throughput, measured using Spark Bench, is a good indicator of the overall performance of the system. Other metrics, such as the total amount of data processed and the number of minor and major garbage collections, as measured using jstat, can also provide valuable insights into system behavior. By considering a range of metrics, one can get a more accurate and comprehensive view of the performance of Spark instances.

3.7 Why would someone choose to run the workloads concurrently and not one by one?

The efficient execution of Spark workloads is essential for maximizing performance and resource utilization in big data analytics. To address this, the concurrent execution of multiple Spark workloads on a single server has emerged as a strategy to improve efficiency and productivity. This section aims to explore the reasons why running multiple Spark workloads concurrently offers significant advantages over sequential execution.

Concurrent execution of Spark workloads provides several benefits. Firstly, it enables optimal resource utilization by effectively leveraging the available hardware resources, including CPUs, memory, and storage. Rather than leaving server resources idle between workloads, concurrent execution ensures their efficient utilization, leading to improved throughput and enhanced server efficiency. Additionally, the consolidation of multiple workloads onto a single server reduces hardware footprint, simplifies management, and minimizes operational costs associated with managing multiple servers.

Another advantage is the potential for increased throughput. By executing multiple workloads concurrently, tasks progress simultaneously, resulting in faster completion and higher overall throughput. This approach is particularly valuable when workloads exhibit varying levels of computational intensity or have different resource requirements. Concurrent execution allows for efficient resource allocation, enabling each workload to access the necessary resources and perform optimally.

Concurrent execution also facilitates workload prioritization, allowing organizations to allocate resources based on workload importance or urgency. By running multiple workloads concurrently, critical or high-priority tasks can be assigned the required resources and processed in a timely manner. This flexibility in resource allocation and workload prioritization ensures efficient utilization of available resources and improves overall performance.

Furthermore, the concurrent execution of Spark workloads supports experimentation and testing. By running workloads concurrently on the same server, comparisons, performance benchmarking, and optimization can be performed in a controlled environment. This concurrent execution environment enables organizations to evaluate and fine-tune Spark applications effectively.

Scalability is another advantage of concurrent execution. As data volumes and processing demands increase, running multiple workloads concurrently allows for horizontal scalability. Additional Spark worker nodes can be added to accommodate larger workloads or handle additional workloads without the need for significant infrastructure changes. This scalability ensures that the system can handle growing demands while maintaining high performance.

In conclusion, the concurrent execution of multiple Spark workloads on a single server offers significant advantages for performance optimization. It enables optimal resource utilization, workload consolidation, improved throughput, workload prioritization, experimentation, and scalability. By carefully managing resources, workload scheduling, and monitoring, organizations can achieve higher performance, reduce infrastructure costs, and simplify management. This approach provides an effective strategy to maximize the benefits of Spark for big data analytics.

3.8 Is cost a contributing factor to pursuing higher throughput for a server?

Renting servers is a common practice for organizations requiring computational resources, and the question arises as to whether reducing the monetary cost is possible by achieving higher throughput and faster workload completion. The relationship between cost reduction and achieving higher throughput on rented servers is indeed significant. By optimizing server performance, efficiently utilizing resources, implementing workload scheduling, and improving productivity, organizations can realize cost savings. Achieving higher throughput and faster workload completion can lead to a reduced rental duration, minimizing the time and associated costs of server usage. Efficient resource utilization and workload scheduling contribute to cost reduction by minimizing the number of servers required and maximizing their utilization. Rental pricing models that take into account resource utilization or data processed can further reduce costs for organizations achieving higher throughput. Additionally, improved productivity resulting from higher throughput and faster workload completion enhances overall efficiency, allowing organizations to accomplish more work within the same rental period and reducing rental expenses. Therefore, pursuing higher throughput and faster workload completion offers tangible benefits in terms of monetary cost reduction for organizations renting servers.

4. Evaluation

4.1 Are GC and S/D still the main overheads of Native Spark when running multiple spark instances and can TeraHeap increase throughput/number of instances running by reducing these costs? What happens with Java Heap and I/O Cache?

Figure **??** shows the performance of multiple Native-TeraHeap Spark instances running LinearRegression with 64 GB dataset per instance in our 64 GB DRAM machine. Each instance of Spark uses one executor with 8 cores per executor. Available DRAM is 56 GB and 8 GB are left to the Operating system, resulting in 64 GB total DRAM. Starting from the left of the graph, the first 2 bars show the performance of Native Spark with each instance using 22 GB DRAM for H1 (Java Heap) and 6 GB for JVM Native memory and I/O cache resulting in a total of 28 GB DRAM per instance and utilizing 56 of 64 GB total DRAM. The next 2 bars show the performance of Native Spark with each instance using 11 GB DRAM for H1 (Java Heap) and 17 GB for JVM Native memory and I/O cache resulting in a total of 28 GB DRAM per instance and utilizing 56 of 56 GB total available DRAM. The next 4 bars show the performance of Native Spark with each instance using 11 GB DRAM for H1 (Java Heap) and 3 GB for JVM Native memory and I/O cache resuting in a total of 14 GB DRAM per instance and utilizing 56 of 56 GB total available DRAM. The next 2 bars show the performance of TeraHeap Spark with each instance using 22 GB DRAM for H1 (Java Heap) and 6 GB for JVM Native memory and I/O cache resulting in a total of 28 GB DRAM per instance and utilizing 56 of 56 GB total available DRAM. The next 2 bars show the performance of TeraHep Spark with each instance using 11 GB DRAM for H1 (Java Heap) and 17 GB for JVM Native memory and I/O cache resulting in a total of 28 GB DRAM per instance and utilizing 56 of 56 GB total available DRAM. The next 4 bars show the performance of TeraHeap Spark with each instance using 11 GB DRAM for H1 (Java Heap) and 3 GB for JVM Native memory and I/O cache resuting in a total of 14 GB DRAM per instance and utilizing 56 of 56 GB total available DRAM. The next 4 bars show the performance of TeraHeap Spark with increased PageCache and reduced H1 with each instance using 6 GB DRAM for H1 (Java Heap) and 8 GB for JVM Native memory and I/O cache resuting in a total of 14 GB DRAM per instance and utilizing 56 of 56 GB total available DRAM. The next 4 bars show the performance of TeraHeap Spark with increased PageCache and reduced H1 with each instance using 6 GB DRAM for H1 (Java Heap) and 8 GB for JVM Native memory and I/O cache resuting in a total of 14 GB DRAM per instance and utilizing 56 of 56 GB total available DRAM. The last 4 bars show the performance of TeraHeap Spark with increased PageCache and reduced H1 with each instance using 6 GB DRAM for H1 (Java

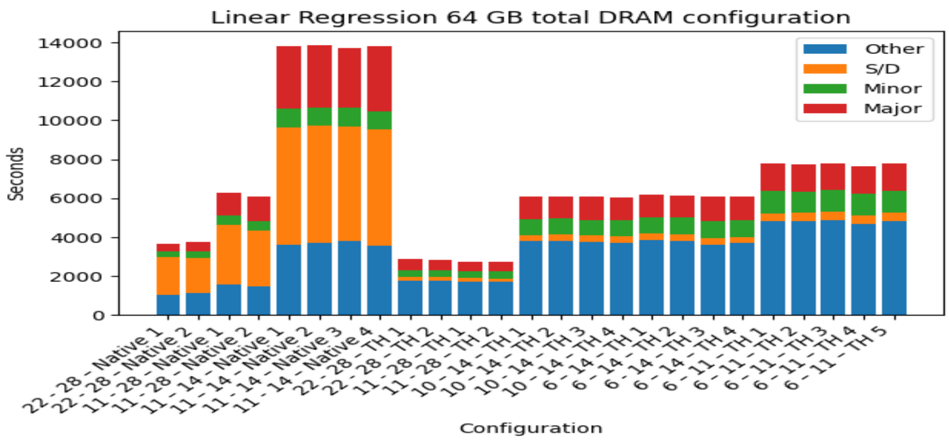

FIGURE 13. Execution time breakdown for multiple instances of LinearRegression using the 64 GB total DRAM setup. E.g. 22-28-64-Native 1 indicates the first of the 2-4-8 instances that was run in parallel and uses 22 GB H1 - 28 GB total cgroup DRAM and 64 GB total DRAM for the machine.

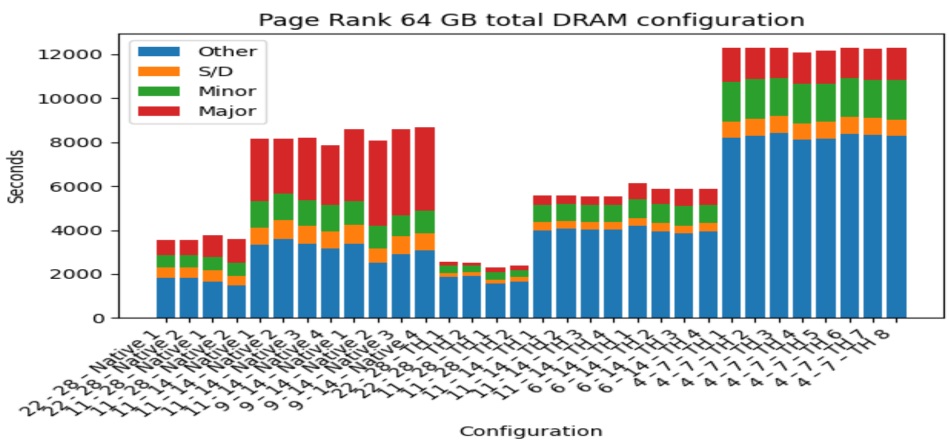

FIGURE 14. Execution time breakdown for multiple instances of PageRank using the 64 GB total DRAM setup. E.g. 22-28-64-Native 1 indicates the first of the 2-4-8 instances that was run in parallel and uses 22 GB H1 - 28 GB total cgroup DRAM and 64 GB total DRAM for the machine.

Heap) and 5 GB for JVM Native memory and I/O cache resulting in a total of 14 GB DRAM per instance and utilizing 56 of 56 GB total available DRAM. By looking at this graph it is clear that Spark suffers by both S/D and GC, but more specifically when it comes to LinerRegression S/D is higher than PageRank. Furthermore we witness Major GC to increase dramaticaly as we increase the number of Spark instances running, same as with PageRank. These overheads are absorbed by TeraHeap. Finally someone would prefer to have a bigger heap than I/O cache when running LinearRegression as well.

Figure 14 shows the performance of multiple Native-TeraHeap Spark instances running PageRank with 8 GB dataset per instance in our 64 GB DRAM machine. Each instance of Spark uses one executor with 8 cores per executor. Available DRAM is 56 GB and 8 GB are left to the Operating system, resulting in 64 GB total DRAM. Starting from the left of the graph, the first 2 bars show the performance of Native Spark with each instance using 22 GB DRAM for H1 (Java Heap) and 6 GB for JVM Native memory and I/O cache resulting in a total of 28 GB DRAM per instance and utilizing 56 of 64 GB total DRAM. The next 2 bars show the performance of Native Spark with each instance using 11 GB DRAM for H1 (Java Heap) and 17 GB for JVM Native memory and I/O cache resulting in a total of 28 GB DRAM per instance and utilizing 56 of 56 GB total available DRAM. The next 4 bars show the performance of Native Spark with each instance using 11 GB DRAM for H1 (Java Heap) and 3 GB for JVM Native memory and I/O cache resuting in a total of 14 GB DRAM per instance and utilizing 56 of 56 GB

FIGURE 15. Execution time breakdown for multiple instances of Logistic Regression using the 64 GB total DRAM setup. E.g. 22-28-64-Native 1 indicates the first of the 2-4-8 instances that was run in parallel and uses 22 GB H1 - 28 GB total cgroup DRAM and 64 GB total DRAM for the machine.

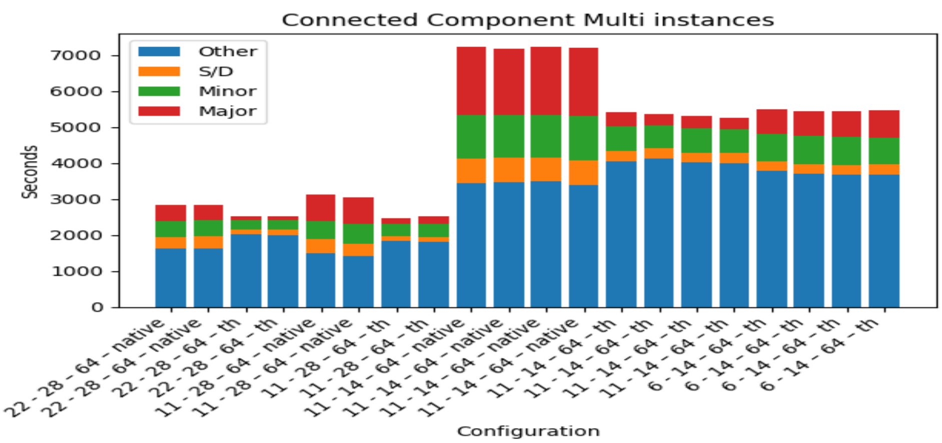

FIGURE 16. Execution time breakdown for multiple instances of Connected Component using the 64 GB total DRAM setup. E.g. 22-28-64-Native 1 indicates the first of the 2-4-8 instances that was run in parallel and uses 22 GB H1 - 28 GB total cgroup DRAM and 64 GB total DRAM for the machine.

total available DRAM. The next 4 bars show the performance of TeraHeap Spark with increased PageCache and reduced H1 with each instance using 9 GB DRAM for H1 (Java Heap) and 5 GB for JVM Native memory and I/O cache resuting in a total of 14 GB DRAM per instance and utilizing 56 of 56 GB total available DRAM. The next 2 bars show the performance of TeraHeap Spark with each instance using 22 GB DRAM for H1 (Java Heap) and 6 GB for JVM Native memory and I/O cache resulting in a total of 28 GB DRAM per instance and utilizing 56 of 56 GB total available DRAM. The next 2 bars show the performance of TeraHep Spark with each instance using 11 GB DRAM for H1 (Java Heap) and 17 GB for JVM Native memory and I/O cache resulting in a total of 28 GB DRAM per instance and utilizing 56 of 56 GB total available DRAM. The next 4 bars show the performance of TeraHeap Spark with each instance using 11 GB DRAM for H1 (Java Heap) and 3 GB for JVM Native memory and I/O cache resuting in a total of 14 GB DRAM per instance and utilizing 56 of 56 GB total available DRAM. The next 4 bars show the performance of TeraHeap Spark with increased PageCache and reduced H1 with each instance using 6 GB DRAM for H1 (Java Heap) and 8 GB for JVM Native memory and I/O cache resuting in a total of 14 GB DRAM per instance and utilizing 56 of 56 GB total available DRAM. The last 8 bars show the performance of TeraHeap Spark with each instance using 4 GB DRAM for H1 (Java Heap) and 3 GB for JVM Native memory and I/O cache resuting in a total of 7 GB DRAM per instance

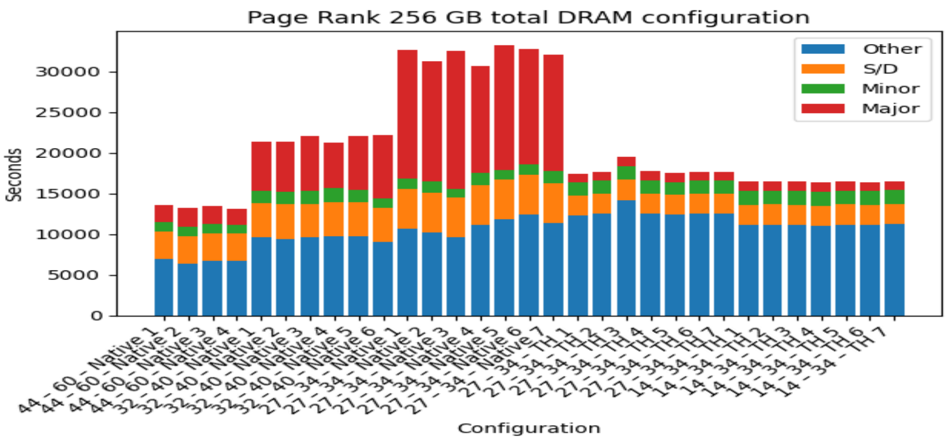

FIGURE 17. Execution time breakdown for multiple instances of PageRank using the 256 GB total DRAM setup. E.g. 44-60-256-Native 1 indicates the first of the 2-4-8 instances that was run in parallel and uses 44 GB H1 - 60 GB total cgroup DRAM and 256 GB total DRAM for the machine.

and utilizing 56 of 56 GB total available DRAM. By looking at this graph it is clear that Spark suffers by both S/D and GC, but more specifically when it comes to PageRank we witness Major GC to increase dramaticaly as we increase the number of Spark instances running. These overheads are absorbed by TeraHeap. TeraHeap also scales to 8 instances while Native can't because it runs out of memory. Finally someone would prefer to have a bigger heap than I/O cache when running PageRank.

Figure **??** shows the performance of multiple Native-TeraHeap Spark instances running LogisticRegression with 8 GB dataset per instance in our 64 GB DRAM machine. Each instance of Spark uses one executor with 8 cores per executor. Available DRAM is 56 GB and 8 GB are left to the Operating system, resulting in 64 GB total DRAM. Starting from the left of the graph, the first 2 bars show the performance of Native Spark with each instance using 22 GB DRAM for H1 (Java Heap) and 6 GB for JVM Native memory and I/O cache resulting in a total of 28 GB DRAM per instance and utilizing 56 of 64 GB total DRAM. The next 2 bars show the performance of Native Spark with each instance using 11 GB DRAM for H1 (Java Heap) and 17 GB for JVM Native memory and I/O cache resulting in a total of 28 GB DRAM per instance and utilizing 56 of 56 GB total available DRAM. The next 4 bars show the performance of Native Spark with each instance using 11 GB DRAM for H1 (Java Heap) and 3 GB for JVM Native memory and I/O cache resuting in a total of 14 GB DRAM per instance and utilizing 56 of 56 GB total available DRAM. The next 4 bars show the performance of TeraHeap Spark with increased PageCache and reduced H1 with each instance using 9 GB DRAM for H1 (Java Heap) and 5 GB for JVM Native memory and I/O cache resuting in a total of 14 GB DRAM per instance and utilizing 56 of 56 GB total available DRAM. The next 2 bars show the performance of TeraHeap Spark with each instance using 22 GB DRAM for H1 (Java Heap) and 6 GB for JVM Native memory and I/O cache resulting in a total of 28 GB DRAM per instance and utilizing 56 of 56 GB total available DRAM. The next 2 bars show the performance of TeraHep Spark with each instance using 11 GB DRAM for H1 (Java Heap) and 17 GB for JVM Native memory and I/O cache resulting in a total of 28 GB DRAM per instance and utilizing 56 of 56 GB total available DRAM. The next 4 bars show the performance of TeraHeap Spark with each instance using 11 GB DRAM for H1 (Java Heap) and 3 GB for JVM Native memory and I/O cache resuting in a total of 14 GB DRAM per instance and utilizing 56 of 56 GB total available DRAM. The next 4 bars show the performance of TeraHeap Spark with increased PageCache and reduced H1 with each instance using 6 GB DRAM for H1 (Java Heap) and 8 GB for JVM Native memory and I/O cache resuting in a total of 14 GB DRAM per instance and utilizing 56 of 56 GB total available DRAM. By looking at this graph it is clear that Spark suffers by both S/D and GC, but more specifically when it comes to LogisticRegression we witness S/D to increase dramaticaly as we increase the number of Spark instances running. These overheads are absorbed by TeraHeap. Finally someone would prefer to have a bigger heap than I/O cache when running LogisticRegression.

Figure **??** shows the performance of multiple Native-TeraHeap Spark instances running ConnectedComponent with 8 GB dataset per instance in our 64 GB DRAM machine. Each instance of Spark uses one executor with 8 cores per executor. Available DRAM is 56 GB and 8 GB are left to the Operating system, resulting in 64 GB total DRAM. Starting from the left of the graph, the first 2 bars show the performance of Native Spark with each instance using 22 GB DRAM for H1 (Java Heap) and 6 GB for JVM Native memory and I/O cache resulting in a total of 28 GB DRAM per instance and utilizing 56 of 64 GB total DRAM. The next 2 bars show the performance of Native Spark with each instance using 11 GB DRAM for H1 (Java Heap) and

17 GB for JVM Native memory and I/O cache resulting in a total of 28 GB DRAM per instance and utilizing 56 of 56 GB total available DRAM. The next 4 bars show the performance of Native Spark with each instance using 11 GB DRAM for H1 (Java Heap) and 3 GB for JVM Native memory and I/O cache resuting in a total of 14 GB DRAM per instance and utilizing 56 of 56 GB total available DRAM. The next 2 bars show the performance of TeraHeap Spark with each instance using 22 GB DRAM for H1 (Java Heap) and 6 GB for JVM Native memory and I/O cache resulting in a total of 28 GB DRAM per instance and utilizing 56 of 56 GB total available DRAM. The next 2 bars show the performance of TeraHep Spark with each instance using 11 GB DRAM for H1 (Java Heap) and 17 GB for JVM Native memory and I/O cache resulting in a total of 28 GB DRAM per instance and utilizing 56 of 56 GB total available DRAM. The next 4 bars show the performance of TeraHeap Spark with each instance using 11 GB DRAM for H1 (Java Heap) and 3 GB for JVM Native memory and I/O cache resuting in a total of 14 GB DRAM per instance and utilizing 56 of 56 GB total available DRAM. The next 4 bars show the performance of TeraHeap Spark with increased PageCache and reduced H1 with each instance using 6 GB DRAM for H1 (Java Heap) and 8 GB for JVM Native memory and I/O cache resuting in a total of 14 GB DRAM per instance and utilizing 56 of 56 GB total available DRAM. By looking at this graph it is clear that Spark suffers by both S/D and GC, but more specifically when it comes to ConnectedComponent we witness Major GC to increase dramaticaly as we increase the number of Spark instances running. These overheads are absorbed by TeraHeap. Finally someone would prefer to have a bigger heap than I/O cache when running ConnectedComponent.

Figure **??** shows the performance of multiple Native-TeraHeap Spark instances running PageRank with 32 GB dataset per instance in our 256 GB DRAM machine. Each instance of Spark uses one executor with 8 cores per executor. Available DRAM is 240 GB and 16 GB are left to the Operating system, resulting in 256 GB total DRAM. Starting from the left of the graph, the first 4 bars show the performance of Native Spark with each instance using 44 GB DRAM for H1 (Java Heap) and 16 GB for JVM Native memory and I/O cache resulting in a total of 60 GB DRAM per instance and utilizing 240 of 256 GB total DRAM. The next 6 bars show the performance of Native Spark with each instance using 32 GB DRAM for H1 (Java Heap) and 8 GB for JVM Native memory and I/O cache resulting in a total of 40 GB DRAM per instance and utilizing 240 of 240 GB total available DRAM. The next 7 bars show the performance of Native Spark with each instance using 27 GB DRAM for H1 (Java Heap) and 7 GB for JVM Native memory and I/O cache resuting in a total of 34 GB DRAM per instance and utilizing 240 of 240 GB total available DRAM. The next 7 bars show the performance of TeraHeap Spark with each instance using 27 GB DRAM for H1 (Java Heap) and 7 GB for JVM Native memory and I/O cache resuting in a total of 34 GB DRAM per instance and utilizing 240 of 240 GB total available DRAM. The last 7 bars show the performance of Native Spark with each instance using 14 GB DRAM for H1 (Java Heap) and 20 GB for JVM Native memory and I/O cache resuting in a total of 34 GB DRAM per instance and utilizing 240 of 240 total DRAM. By looking at this graph it is clear that Spark suffers by both S/D and GC, but more specifically when it comes to PageRank we witness Major GC to increase dramaticaly as we increase the number of Spark instances running. These overheads are absorbed by TeraHeap. Finally someone would prefer to have a bigger heap than I/O cache when running PageRank.

By looking at figures 18, 19, 20 and 21 we see that Native Spark's throughput decreases as the number of colocated instances-executors increase in the server. The main goal for colocating tasks is to increase the CPU utilization and achieve better throughput as effectively as possible as if running each instance isolated. While Native Spark fails to provide scalability, TH Spark achieves nearly double the throughput of Native Spark and is also able to scale beyond 4 instances.

Figures 22, 23, 24 and 25 show the percentage of idle CPU while the number of colocated instances grows. CPU idle should decrease in this particular concept and that is something proven in the figure. In PageRank graph we notice TeraHeap with 2 instances achieving less idle CPU percentage than Native Spark with 4 instances.

Tables 26, 27, 28 and 29 show a detailed synopsis of the performance and AWS, GCP and Azure costs of deploying multiple instances of Spark using both techniques. Each setup is run using 2 different amounts for H1 (80 and 40%). While the number of instances increases TeraHeap appears to maintain throughput while Native Spark has significant throughput decrease. In several occassions, if we compare Native's to TeraHeap's throughput we notice TeraHeap having more than 100% increase compared to Native. In PageRank's 64 DB DRAM setup TeraHeap manages to scale to 8 instances while Native Spark runs out of memory. Furthermore when looking at the PageRank table it becomes clear that TeraHeap achieves a higher CPU utilization, because of the S/D elimination and less I/O. Despite TeraHeap's better performance, the total time taken for all instances to complete surpasses the time it would take to run the same number of single isolated instances one after the other. This means that in order to achieve better performance we would need a scheduling policy to decide H1 and PageCache amount for each workload running. This is something discussed in the next section.

5. Future Work

While our proposed offloading technique shows promising results in improving job throughput for big data analytics workloads on Spark clusters, there are several avenues for future work to further improve the performance and scalability of Spark clusters.

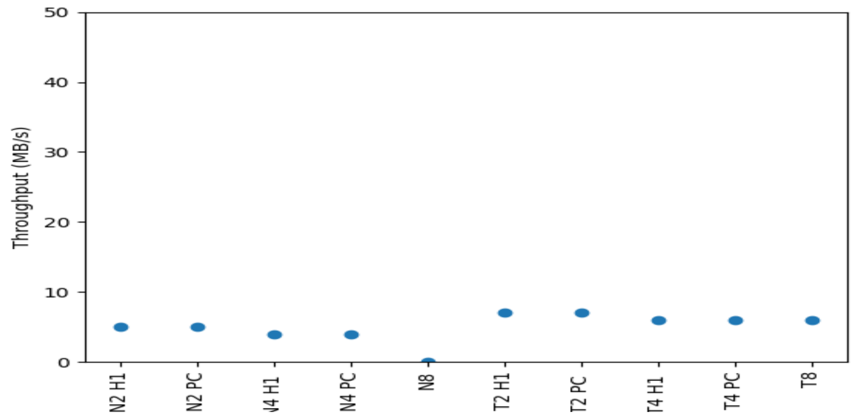

FIGURE 18. Page Rank 64 GB DRAM setup Native and TeraHeap throughput as the number of instances increases.

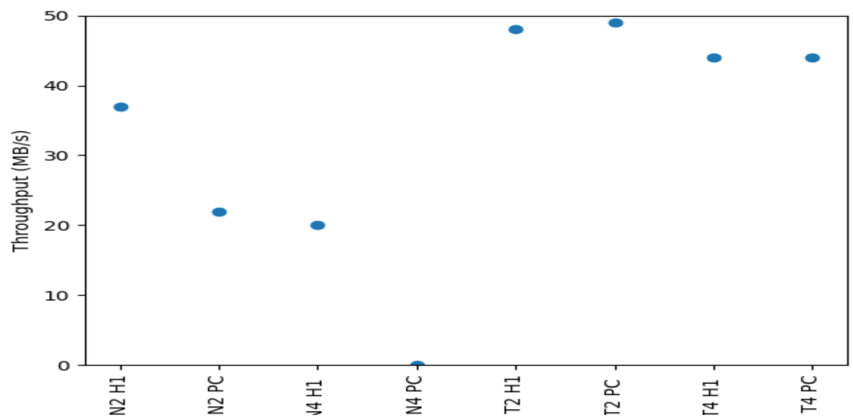

FIGURE 19. Linear Regression 64 GB DRAM setup Native and TeraHeap throughput as the number of instances increases.

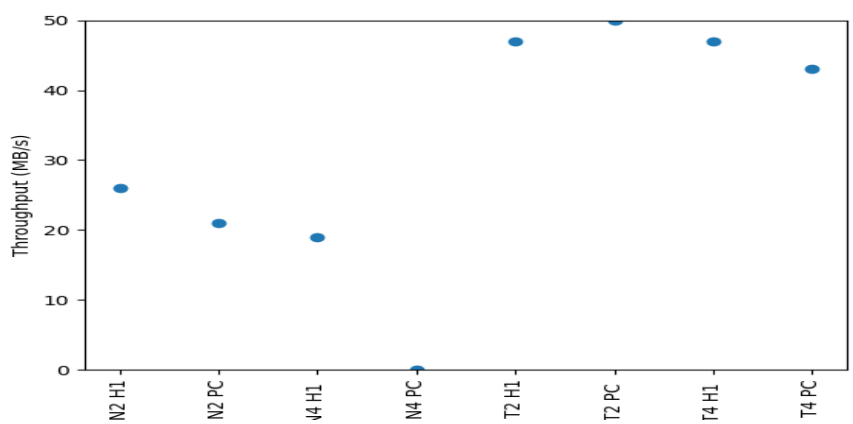

FIGURE 20. Logistic Regression 64 GB DRAM setup Native and TeraHeap throughput as the number of instances increases.

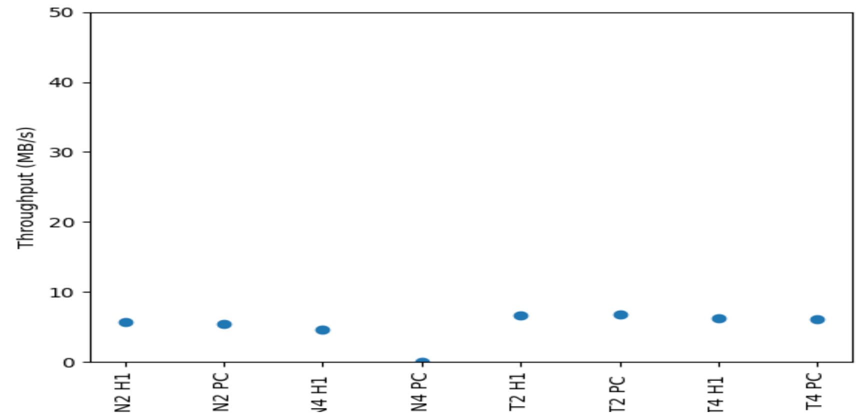

FIGURE 21. Connected Component 64 GB DRAM setup Native and TeraHeap throughput as the number of instances increases.

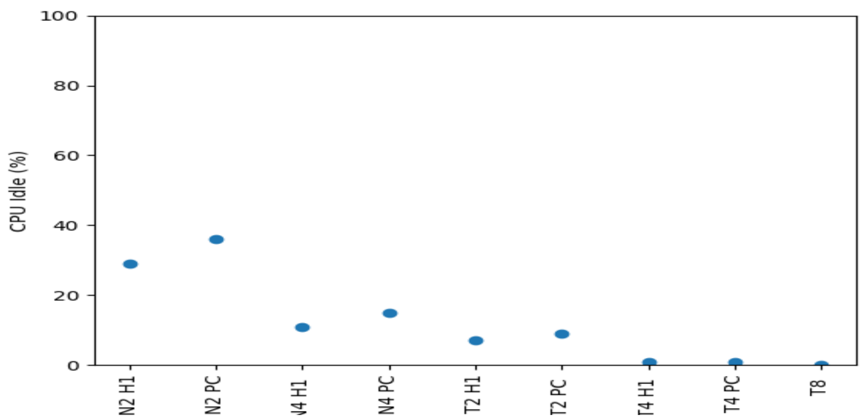

FIGURE 22. Page Rank 64 GB DRAM setup Native and TeraHeap idle CPU as the number of instances increases.

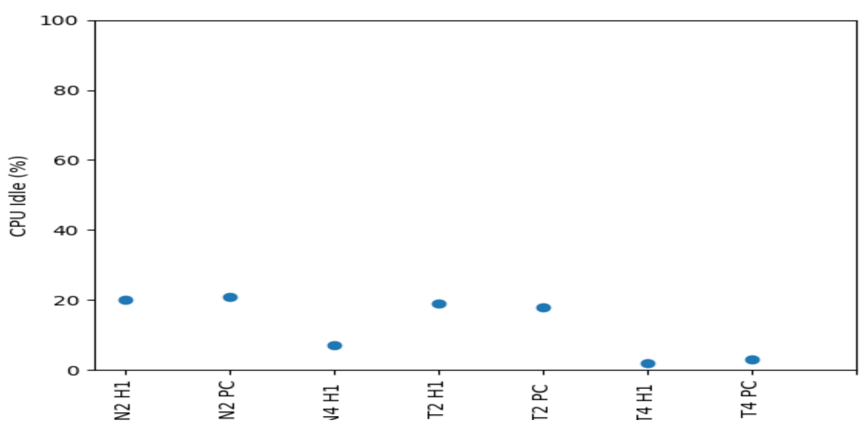

FIGURE 23. Linear Regression 64 GB DRAM setup Native and TeraHeap idle CPU as the number of instances increases.

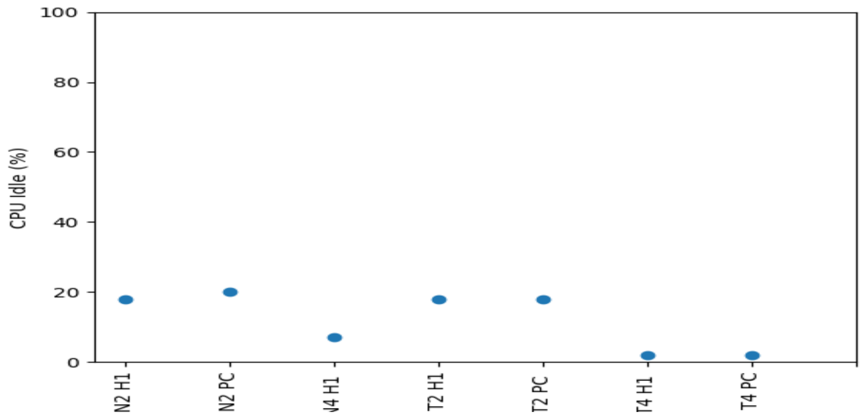

FIGURE 24. Logistic Regression 64 GB DRAM setup Native and TeraHeap idle CPU as the number of instances increases.

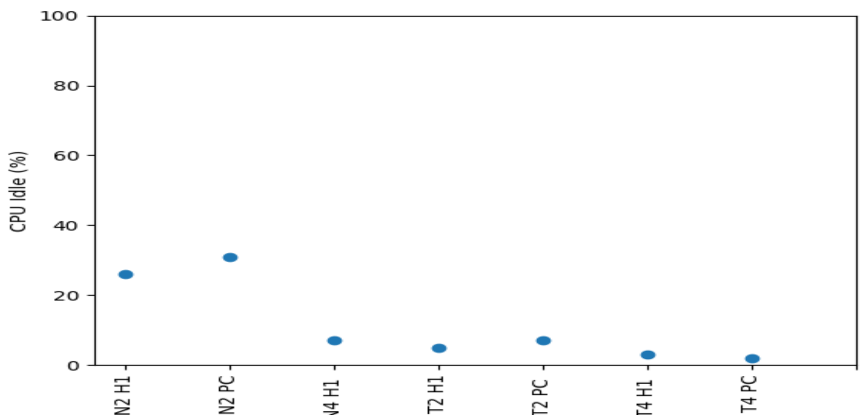

FIGURE 25. Connected Component 64 GB DRAM setup Native and TeraHeap idle CPU as the number of instances increases.

Synopsis PR	H1 Size / I	Memory / I	Total memory	#I	Exec. Time	CPU Idle	Total MB Processed	MB/s	MB/s/I	Cost $
N2 H1 - small	22	28	64	2	3563	29	16980	5	2	0.6
N2 PC - small	11	28	64	2	3759	36	16980	5	2	1.2
N4 H1 - small	11	14	64	4	8195	11	33960	4	1	1.8
N4 PC – small	9	14	64	4	8657	15	33960	4	1	1.8
T2 H1 – small	22	28	64	2	2545	7	16980	7	3	0.6
T2 PC – small	11	28	64	2	2385	9	16980	7	4	0.6
T4 H1 – small	11	14	64	4	5554	1	33960	6	2	1.2
T4 PC – small	6	14	64	4	5880	1	33960	6	2	1.2
N8 – small	4	7	64	8	OOM					
T8 – small	4	7	64	8	12305	0	67920	6	1	2.4
N4 - big	44	60	256	4	13542	15	135872	10	3	6.4
N6 – big	32	40	256	6	21245	1	203808	10	2	9.6
N7 – big	27	34	256	7	32763	0	237776	7	1	14.4
T7 H1 – big	27	34	256	7	19443	0	237776	12	2	9.6
T7 PC – big	27	34	256	7	16485	0	237776	14	2	8

FIGURE 26. Page Rank synopsis table.

Firstly, one potential direction for future work is to investigate the use of other types of storage mediums such as the hybrid NVM. This medium could improve the performance of Big data analytics further by combining the advantages of memory and storage.

Secondly, another area for future work is to develop tech-

Synopsis LinR	H1 size / I	Memory / I	Total memory	#I	Exec. Time	CPU Idle	MB Processed	MB/s	MB/s/I	Cost AWS $	Cost GCP $	Cost Azure $
N2 H1	22.0	28.0	64.0	2.0	3745.0	20.0	134896.0	37.0	18.0	0.6	0.58	0.67
N2 PC	11.0	28.0	64.0	2.0	6288.0	21.0	134896.0	22.0	11.0	1.2	1.16	1.34
N4 H1	11.0	14.0	64.0	4.0	13874.0	7.0	269792.0	20.0	5.0	2.4	2.32	2.01
N4 PC	6.0	14.0	64.0	4.0	OOM	**	0.0	0.0	0.0	**	**	**
T2 H1	22.0	28.0	64.0	2.0	2891.0	19.0	134896.0	48.0	24.0	0.6	0.58	0.67
T2 PC	11.0	28.0	64.0	2.0	2747.0	18.0	134896.0	49.0	25.0	0.6	0.58	0.67
T4 H1	11.0	14.0	64.0	4.0	6075.0	2.0	269792.0	44.0	11.0	1.2	1.16	1.34
T4 PC	6.0	14.0	64.0	4.0	6176.0	3.0	269792.0	44.0	11.0	1.2	1.16	1.34

FIGURE 27. Linear Regression synopsis table.

niques for dynamically adjusting the heap offloading decisions based on workload characteristics and resource availability. For example, the offloading decision can be based on the size of the input data or the availability of DRAM capacity in the cluster. Such techniques can help maximize the performance gains achieved by offloading while minimizing the cost of offloading.

Thirdly, an interesting direction for future work is to explore the use of heap offloading in environments where Spark clusters are deployed across multiple machines using RDMA to achieve communication between the different machines. This can help utilize the DRAM, CPU and storage availability in more than one machine and provide a more cost-effective solution for big data processing.

Finally, another potential area for future work is to investigate the use of heap offloading for other big data processing frameworks beyond Spark. Many other big data processing frameworks such as Apache Giraph can potentially benefit from offloading techniques to improve their performance and scalability.

Overall, there are many exciting avenues for future work in improving the performance and scalability of big data processing frameworks such as Spark. Our proposed offloading technique provides a solid foundation for future work and offers a promising approach for addressing the challenges of big data processing.

6. Conclusion

In this paper, we proposed a new technique for improving the performance and job throughput of Spark clusters by moving parts of the managed Java Heap to a secondary memory-mapped heap over fast storage devices such as NVMe. Our approach leverages the capabilities of the underlying running machine to free computation-intensive tasks running on the Spark workers from memory pressure, thereby reducing the workload on the workers and improving their performance and job throughput.

Our experimental results demonstrate the effectiveness of our approach using various big data analytics workloads on a Spark cluster. We also compare our approach with the native Spark distribution and showed that our approach can be used instead of this distribution to further improve performance.

Our work contributes to the growing body of research on improving the performance and scalability of Spark clusters for big data analytics workloads. Our approach offers a scalable solution for processing increasingly large and complex big data workloads and can be easily integrated into existing Spark clusters.

Overall, our offloading technique offers a promising approach to improving job throughput for big data analytics work-

Synopsis LogR	H1 Size / I	Memory / I	Total memory	#I	Exec. Time	CPU Idle	Total MB Processed	MB/s	MB/s/I	Cost AWS $	Cost GCP $	Cost Azure $
N2 H1	22.0	28.0	64.0	2.0	5127.0	18.0	133348.0	26.0	13.0	1.2	1.16	0.67
N2 PC	11.0	28.0	64.0	2.0	6302.0	20.0	133348.0	21.0	11.0	1.2	1.16	1.34
N4 H1	11.0	14.0	64.0	4.0	13730.0	7.0	266696.0	19.0	5.0	2.4	2.32	2.68
N4 PC	6.0	14.0	64.0	4.0	OOM	**	0.0	0.0	0.0	***	***	***
T2 H1	22.0	28.0	64.0	2.0	2861.0	18.0	133348.0	47.0	24.0	0.6	0.58	0.67
T2 PC	11.0	28.0	64.0	2.0	2683.0	18.0	133348.0	50.0	25.0	0.6	0.58	0.67
T4 H1	10.0	14.0	64.0	4.0	5712.0	2.0	266696.0	47.0	12.0	1.2	1.16	1.34
T4 PC	6.0	14.0	64.0	4.0	6138.0	2.0	266696.0	43.0	10.0	1.2	1.16	1.34

FIGURE 28. Logistic Regression synopsis table.

loads on Spark clusters, particularly for computation-intensive tasks. With the increasing demand for efficient and scalable big data processing frameworks, our approach provides a valuable contribution to the field of big data analytics and memory management.

ACKNOWLEDGMENT

Still an empty section.

REFERENCES

[1] Zaharia, M., Xin, R. S., Wendell, P., Das, T., Armbrust, M., Dave, A., Meng, X., Rosen, J., Venkataraman, S., Franklin, M. J., Ghodsi, A., Gonzalez, J., Shenker, S., and Stoica, I., 2016. "Apache spark: A unified engine for big data processing". In Communications of the ACM, Association for Computing Machinery.

[2] Kolokasis, I. G., Papagiannis, A., Pratikakis, P., Bilas, A., Zakkak, F., Evdorou, G., Akram, S., and Kozanitis, C., 2023. "Teraheap: Reducing memory pressure in managed big data frameworks". In ASPLOS '23, March 25–29, 2023, Vancouver, BC, Canada, Association for Computing Machinery.

[3] Marco, V. S., Taylor, B., Porter, B., and Wang, Z., 2017. "Improving spark application throughput via memory aware task co-location: A mixture of experts approach". In Proceedings of Middleware '17, Las Vegas, NV, USA, Association for Computing Machinery.

[4] Kirisame, M., Shenoy, P., and Panchekha, P., 2022. "Optimal heap limits for reducing browser memory use". In OOPSLA, Association for Computing Machinery.

[5] Amaro, E., Branner-Augmon, C., Luo, Z., Ousterhout, A., Aguilera, M. K., Panda, A., Ratnasamy, S., and Shenker, S., 2020. "Can far memory improve job throughput?". In EuroSys '20, April 27–30, 2020, Heraklion, Greece, Association for Computing Machinery.

[6] Weiner, J., Agarwal, N., Schatzberg, D., Yang, L., Wang, H., Sanouillet, B., Sharma, B., Heo, T., Jain, M., Tang, C., and Skarlatos, D., 2022. "Tmo: Transparent memory offloading in datacenters". In ASPLOS '22, February 28 – March 4, 2022, Lausanne, Switzerland, Association for Computing Machinery.

[7] Sharma, P., Kulkarni, P., and Shenoy, P. "Per-vm page cache partitioning for cloud computing platforms". Association for Computing Machinery.

[8] Chen, H., and Wang, F. Z., 2015. "Spark on entropy: A reliable and efficient scheduler for low-latency parallel jobs in heterogeneous cloud". In LCN 2015, Clearwater Beach,

Synopsis CC	H1 Size / I	Memory / I	Total memory	#I	Exec. Time	CPU Idle	Total MB Processed	MB/s	MB/s/I	Cost AWS $	Cost GCP $	Cost Azure $
N2 H1	22.0	28.0	64.0	2.0	2958.0	26.0	16980.0	5.74	3.0	0.6	0.58	0.67
N2 PC	11.0	28.0	64.0	2.0	3125.0	31.0	16980.0	5.43	3.0	0.6	0.58	0.67
N4 H1	11.0	14.0	64.0	4.0	7231.0	7.0	33960.0	4.69	3.0	1.8	1.74	2.01
N4 PC	6.0	14.0	64.0	4.0	OOM	**	0.0	0.0	0.0	***	***	***
T2 H1	22.0	28.0	64.0	2.0	2526.0	5.0	16980.0	6.72	4.0	0.6	0.58	0.67
T2 PC	11.0	28.0	64.0	2.0	2519.0	7.0	16980.0	6.74	4.0	0.6	0.58	0.67
T4 H1	11.0	14.0	64.0	4.0	5439.0	3.0	33960.0	6.24	3.0	1.2	0.58	0.67
T4 PC	6.0	14.0	64.0	4.0	5487.0	2.0	33960.0	6.18	3.0	1.2	1.16	1.34

FIGURE 29. Connected Component synopsis table.

Florida, USA, Association for Computing Machinery.

[9] Thamsen, L., Beilharz, J., Tran, V. T., Nedelkoski, S., and Kao, O., 2019. "Mary, hugo, and hugo*: Learning to schedule distributed data-parallel processing jobs on shared clusters". In Euro-Par 2019, Association for Computing Machinery.

[10] Bhimani, J., Yang, Z., Leeser, M., and Mi, N., 2017. "Accelerating big data applications using lightweight virtualization framework on enterprise cloud". In 2017 IEEE High Performance Extreme Computing Conference (HPEC), IEEE.

[11] Zhang, K., Tanimura, Y., Nakada, H., and Ogawa, H., 2017. "Understanding and improving disk-based intermediate data caching in spark". In 2017 IEEE International Conference on Big Data (Big Data), IEEE.

[12] Intasorn, Y., Rattanaopas, K., and Chuchuen, Y., 2022. "Using compression tables to improve hiveql performance with spark a case study on nvme storage devices". In 2022 26th International Computer Science and Engineering Conference (ICSEC), IEEE.

[13] Apache, 2023. "Rdd programming guide (spark 3.4.0 - 2023 update) - https://spark.apache.org/docs/latest/rdd-programming-guide.html".

[14] Apache, 2023. "Building spark (spark 3.4.0 - 2023 update) - https://spark.apache.org/docs/latest/building-spark.html".

[15] Apache, 2023. "Tuning spark (spark 3.4.0 - 2023 update) - https://spark.apache.org/docs/latest/tuning.html".

[16] Apache, 2023. "Spark configuration (spark 3.4.0 - 2023 update) - https://spark.apache.org/docs/latest/configuration.html".

[17] Apache, 2023. "Monitoring and instrumentation (spark 3.4.0 - 2023 update) - https://spark.apache.org/docs/latest/monitoring.html".

[18] chriswhocodes, 2023. "Vm options explorer - openjdk8 hotspot - https://chriswhocodes.com".