EFFICIENT AND HIGH-PERFORMANCE DATA ORCHESTRATION FOR LARGE SCALE CLOUD WORKLOADS

by

SHOUWEI CHEN

A dissertation submitted to the

School of Graduate Studies

Rutgers, The State University of New Jersey

In partial fulfillment of the requirements

For the degree of

Doctor of Philosophy

Graduate Program in Electrical and Computing Engineering

Written under the direction of

Ivan Rodero

and approved by

_____

_____

_____

_____

New Brunswick, New Jersey

May, 2021

ABSTRACT OF THE DISSERTATION

Efficient and High-Performance Data Orchestration for Large Scale Cloud Workloads

By Shouwei Chen

Dissertation Director:

Ivan Rodero

The computing frameworks running in the cloud environment at an extreme scale provide efficient and high-performance computing services to various domains. These cloud computing frameworks build scalable, reliable, and highly accessible data pipelines for many academia, science, and industry services. Data analytics generates a large amount of intermediate data at the back of cloud computing frameworks while processing large amounts of data from different data sources. However, enormous data addresses the challenges to these frameworks to deal with data high performance and efficiency. The data orchestration based on memory and high-performance storage devices has become a key concern to optimize these cloud computing frameworks' performance.

The increasing data scale and complexity of the cloud environment pose challenges to run applications fast and efficiently. The existing computing clusters can fetch the data from different cloud infrastructure, including common storage, high-performance storage devices, and high-speed fabric interconnection. However, it is still challenging to provide the corresponding data orchestration for the existing computing frameworks. First, computing frameworks access the underlying persistent data storage layer based

on the different storage devices and memory. Furthermore, the revolution of storage devices addresses new challenges for existing computing frameworks to utilize advanced storage devices efficiently. Second, most of the existing computing frameworks use an intermediate data layer for intermediate storage. However, providing an efficient and high-performant storage layer for large-scale computing frameworks, such as intermediate data storage and shuffle data storage, is still challenging. The imbalance and small data storage introduce new challenges, including new hardware devices and appropriate data orchestration designs. Consequently, the revolution of hardware devices requires a new paradigm for data orchestration for cloud computing frameworks.

This thesis addresses the above challenges and proposes novel mechanisms and solutions for building efficient and high-performance data orchestration for big data frameworks, and makes the following contributions: (1) Studies representative workloads for big data processing frameworks using different storage technologies and design choices and explores the I/O bottleneck of in-memory big data frameworks on high-performance computing clusters with non-volatile memory. (2) Designs and explores architectural foundations to run in-memory big data framework in the hybrid cloud environment with fast fabric interconnection between geo-distributed data centers. (3) Proposes an abstraction for disaggregated memory pool based on persistent memory and Remote Direct Memory Access (RDMA) to optimize the computing resource efficiency and performance of intermediate storage of big data frameworks. (4) Provides a novel in-transit shuffle mechanism for big data frameworks, which is lightweight and compatible with modern in-memory big data frameworks.

The proposed mechanisms and solutions have been implemented, deployed, and evaluated in high-performance clusters and real computing environments, including academic clusters at Rutgers and production systems at scale in the information technology industry.

Acknowledgements

First and foremost, I would like to thank my advisor, Dr. Ivan Rodero, for his guidance and support through my Ph.D study and research. I am truly grateful for his advice, encouragement, patience and cheerfulness.

I would like to thank to Dr. Manish Parashar, Dr. Ivan Marsic, and Dr. Wensheng Wang for serving as my Ph.D committee members and spending their precious time in reading and review my thesis.

My special thanks to my parents Gang Chen and Shuihua Shou, and other family members, for their encouragement, support and enormous love.

Dedication

To Haoyuan.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation and Background

With the growth of the scale and complexity of modern services, cloud computing platforms have become a choice of preference to deploy services because of cost-efficient, reliable, and high-available services. Cloud computing frameworks provide scalable and extensible platforms to support data analytics. For example, Hadoop [1], and Spark [2] are widely used for data warehouse analytics, machine learning analytics. Hive [3] and Spark SQL [4] provide the structure data processing in a data warehouse; Presto [5] and Impala [6] are the in-memory database engine for running interactive data analytics; Storm [7], Flink [8], and Spark streaming enable high throughput streaming processing in the cloud environment. These cloud computing frameworks provide reliable and high-performance cloud services for petascale data warehouses.

With the growth of the scale of data and complexity of data analytics, the computing frameworks deal with a large volume of raw data while dealing with the same or even larger intermediate data at run time. Therefore, cloud computing frameworks need a scalable, reliable persistent storage layer, like HDFS (Hadoop Distributed File System), Amazon Simple Storage Service (S3), and a fast, scalable intermediate storage layer between storage and computing layer.

The traditional distributed persistent file systems and object storage systems, like HDFS [9] and S3, are designed to store a large number of files/objects with a large volume of data. Thus, the design of the system is based on affordable hardware, like spinning disks. Moreover, because of master-worker architecture, most of the existing distributed persistent storage systems have limited bandwidth for metadata processing, including but not limit to list status, make directories, and open files. Therefore,

the traditional distributed system can not offer the expected performance to various cloud data analytics, which has high-performance requirements. At the same time, the advanced storage devices and fast fabric interconnection, like solid-state disks, non-volatile memory, persistent memory, and Remote Direct Memory Access (RDMA), provide more opportunities to build high-performance persistent data orchestration for cloud computing frameworks.

The large volume of data exchange between computing frameworks and the storage layer can be divided into two key categories, (1) caching data, the shared data across different stages in the same applications, and (2) shuffling data, the data exchange themselves across the computing cluster between stages within the same application. Efficient and high-performance data orchestration of the storage layer in cloud computing frameworks running in large-scale clusters faces various challenges and requires new approaches.

These large amounts of cloud data analytics are based on the distributed storage systems designed based on spinning disks and low bandwidth fabric interconnection. For example, HDFS is one of the most widely used persistent data layers in the cloud. Such a distributed storage system provides cost-efficient and reliable persistent data storage for cloud data analytics. However, HDFS has high I/O overhead even on advanced storage devices as metadata operation is expensive due to metadata operations overheads and the high latency across the network. Thus, these distributed storage systems cannot fully utilize the performance of advanced storage devices, which are becoming an affordable and available solution at the current time. Consequently, it is necessary to fill the performance gap of these distributed storage systems with advanced storage devices.

Furthermore, cloud computing's architecture trends indicate that hybrid clouds are becoming one of the most significant use cases in cloud computing. For example, a computing cluster can fetch data from different data centers, regions, and even countries. These trends introduce the challenges of dealing with remote data efficiently. A logical solution is building a caching layer between the persistent data layer and the computing clusters. For example, Alluxio [10] builds a caching layer for hybrid cloud

and multi-cloud environments, which optimize the performance for frequently accessed (or "hot") data. However, it is still necessary to efficiently run cloud applications in hybrid cloud models, regardless of solutions for improving hot data access.

Finally, the increasing complexity of data analytics makes the execution plan of computing frameworks very challenging. Complex data analytics makes computing frameworks have a higher overhead on accessing the intermediate data layer than the persistent data layer. It makes the optimization of intermediate data access more and more critical. For example, Spark implements a DAG (Directed Acyclic Graph) engine to allow the cloud analytics to reuse the same job's intermediate data. Thus, the tasks in the same job can reuse the previous data without repeated computation. However, there are still open challenges, such as shuffling in the Map-reduce model. Shuffling is an n-to-n communication operation in Map-Reduced-based computing frameworks, which profoundly slow down cloud applications processing. The stability of shuffling is highly affected by the scale of data, a critical concern to large-scale cloud data analytics.

This dissertation explores efficient and high-performance data orchestration with novel approaches and new hardware technologies to overcome the above concerns.

1.2 Research Challenges

The complexity of cloud architectures and data analytics poses challenges for building efficient and high-performance data orchestration. The key challenges addressed in this thesis are as follows:

Local Data access performance: Computing frameworks try to co-locate computing units with data, which can reduce data movement across the network. Without the network transmission between different nodes, the I/O performance for the compute units highly depends on the local data access rate, which relies on the I/O performance of the storage devices. For example, Spark stores intermediate data in disk or memory to improve iterative data analytics performance. However, the limited memory space makes it infeasible to store all intermediate data in memory. High-performance storage devices, like NVMe, are becoming reliable and cost-effective candidates to optimize the

performance for the persistent data layer and the intermediate data layer. However, there are still challenges for existing computing frameworks to utilize high-performance storage devices' performance fully. Thus, we must understand and provide solutions to the primary cause of the performance gap between computing frameworks, storage systems, and storage devices.

Data access across geo-distributed data centers: Existing data orchestration techniques in hybrid cloud environments include caching mechanism to eliminate the performance degradation due to the low bandwidth between geo-distributed data centers. However, existing caching techniques between geo-distributed data centers have significant limitations. While caching policies provide significant performance gain for hot data accesses, they do not fit for cold data accesses. An alternative solution is to build a fast fabric interconnection model between geo-distributed data centers. Furthermore, it is necessary to investigate methods for fully utilizing the computing resources between two geo-distributed data centers with such as fast fabric interconnection.

Utilization of computing and storage resources: The performance of cloud applications highly depends on the performance of the cloud infrastructure, including the computing and storage resources associated with them. For example, the performance of in-memory computing, such as Spark and Presto, highly depends on the volume of memory. Spark has to spill the data into disks without enough intermediate data storage capacity. The limitation of memory and storage resources can significantly impact the performance of in-memory computing frameworks. As a result, it is essential to provide enough and appropriate computing resources and storage resources to support the in-memory computing frameworks running at the expected state.

Intermediate data storage: The primary I/O cost in cloud data analytics is accessing intermediate data, including reusing data blocks and shuffling data. For example, Spark uses a DAG engine to reuse data blocks in the same job. This mechanism can reduce the computing burden of re-computing data repeatedly. Compared to reused data blocks, shuffling, which is an n-to-n communication in Map-Reduce frameworks, introduces unreliability and a performance bottleneck in large-scale cloud data analytics. Thus, it is necessary to optimize the reliability and accessing rate for shuffling.

1.3 Overview of Thesis Research

This thesis addresses the research challenges related to data analytics in cloud environments to provide efficient and high-performant data orchestration for big data computing frameworks. It first presents a high-level data orchestration model of cloud computing frameworks focusing on the different hardware technologies. Specifically, it describes how different storage technologies and computing resources impact performance using various data formats. It assesses the performance gap between DRAM, spinning disks, and non-volatile memory. Furthermore, it explores critical performance bottlenecks of high-performance storage devices in existing cloud-based data computing frameworks. This thesis proposes an approach to optimize cloud computing frameworks' performance with high-performance storage devices and enhanced serialization and de-serialization mechanisms.

This thesis also explores mechanisms for delivering elastic computing clusters across geo-distributed data centers to deal with scattered large volumes of data efficiently. It investigates a scalable and efficient hybrid cloud architecture for cloud-based big data computing frameworks to achieve resource efficiency across geo-distributed data centers. By offloading the computing burden from the local data center, it fully utilizes the spare computing resources across geo-distributed data centers. Also, it explores the potential of elastic computing clusters across geo-distributed data centers with fast fabric interconnection. It aims to provide ways for enabling data analytics on high latency and high bandwidth hybrid cloud-based environments, which access remote persistent data layer and local intermediate data layer, to deliver comparable performance to environments with local storage and computing resources. This thesis envisions an architecture that can be deployed in real-world cloud data analytics and hybrid cloud environments satisfying performance requirements at scale and facilitating harvesting sparse computing resources across geo-distributed data centers.

Furthermore, this thesis proposes a method based on a disaggregated persistent memory pool for delivering efficient and high-performant data orchestration for Map-Reduced oriented cloud computing frameworks. The co-design of the disaggregated

memory pool and data analytics computing frameworks avoids single node failures due to data skew and re-arrange memory utilization across different computing nodes. This enhancement is achieved by replacing local shuffling and data persistence with the proposed disaggregated persistence memory pool. The proposed method explores the potential of edge storage technology (i.e., non-volatile memory), allowing computing clusters to increase the memory capacity at a low cost and optimizing uneven computing resource utilization. This solution is deployed and evaluated using a real-world cloud environment tested with data analytics at scale. Moreover, RDMA technology is explored to improve the performance of the targeted cloud computing frameworks.

Finally, this thesis studies stateless shuffling to improve cloud computing frameworks' stability in large-scale cloud environments at low cost. This novel shuffling approach separates the storage and computing units, making computing frameworks compatible with current major cloud environments. It also provides a cost-efficient method to optimizing shuffling for Map-Reduce-based computing frameworks with HDFS, which does not require storage systems to provide a high-throughput namespace. Specifically, it proposes in-transit shuffling, which moves the shuffle data into a specific shuffle cluster instead of the local computing nodes. The stateless shuffling is compatible with most shuffle mechanisms and avoids holding the shuffle service's namespace data. This solution provides higher reliability than local shuffling in large-scale cloud environments.

1.4 Contributions

This thesis's primary research contributions are around efficient data orchestration to optimize cloud computing frameworks at scale. These contributions significantly enhance cloud computing frameworks' efficiency and performance, which impact a broad range of data analytics in multiple fields. The specific contributions are summarized as follows:

- Studies representative workloads for big data processing frameworks to formulate a performance model for cloud computing frameworks and data orchestration. This formulation is based on different storage technologies and design choices for

computing clusters. It provides a fundamental understanding of different storage devices' performance gaps, including DRAM, spinning disk, and non-volatile memory. It also explores the I/O bottleneck of in-memory big data frameworks on high-performance computing clusters with non-volatile memory.

- Designs and explores a hybrid cloud architecture to enable elastic computing clusters across geo-distributed data centers with fast fabric interconnection. The overarching approach consists of harvesting spare computing resources across geo-distributed data centers for running cloud data analytics with limited performance loss. Different potential architectures to run cloud-based data analytics are explored. The proposed architectural foundations improve the overall computing resource utilization across geo-distributed data centers without significant performance loss.

- Proposes an abstraction for disaggregated memory architecture for cloud computing frameworks to optimize the performance and computing resource management in large-scale deployments. This innovative approach increases the total cluster memory capacity by building a shared data pool for intermediate data storage with persistent memory and Remote Direct Memory Access (RDMA). This disaggregated memory architecture increases overall memory capacity with affordable costs and improves cloud data analytics's end-to-end performance.

- Formulates and provides a novel in-transit shuffle mechanism that can increase cloud computing frameworks' reliability in large-scale deployments. This mechanism improves the computing cluster's reliability by solving single-point failures and proposes a stateless shuffle service without maintaining a separate namespace. It is lightweight and compatible with modern in-memory big data frameworks.

1.5 Dissertation Overview

The rest of this document is organized as follows:

Chapter 2 provides the necessary background and presents an overview of related

techniques in achieving efficient and high-performant data orchestration for big data computing frameworks.

Chapter 3 analyzes Spark persistence technology's performance using different storage technologies and compares native Spark persistence with the Alluxio distributed in-memory file system. It shows that NVMe with columnar compression is an appropriate candidate for complementing memory in Spark clusters. It also indicates that Alluxio has better performance than Spark persistence technology for large datasets and that Alluxio has better load balancing than Spark persistence. Experimental results suggest that software-defined infrastructure can be a viable solution for provisioning bare-metal disaggregated data center resources. They also provide significant data points to illuminate in-memory systems' requirements to scale next-generation software-defined infrastructure implementations efficiently.

Chapter 4 characterizes the computing resource utilization of different geo-distributed computing clusters and introduces the use of fast fabric interconnections across geo-distributed data centers. Then, it explores the potential deployment with these data centers and evaluates the system's performance based on Spark, HDFS, and Kubernetes in a production enterprise environment. The findings motivate exploring the potential of using fast fabric interconnections to harvest spare computing resources across geo-distributed data centers. Experimentation based on simulation of a data warehouse core service shows that an elastic computing cluster across geo-distributed data centers can speed up large data warehouse enterprise services.

Chapter 5 presents the design and implementation of a disaggregated memory system with persistent memory for Map-reduce-based cloud computing frameworks. The proposed system optimizes the data center's memory efficiency with external extended persistent memory and a disaggregated memory pool. Leveraging shuffle and persistence optimization demonstrates that the system can significantly reduce the execution time for shuffle-intensive applications. The experimental evaluation of the shuffle and persistence mechanisms using an in-memory distributed file system shows that

the proposed approach can also increase the overall memory capacity with low overhead. Further, the results show persistent memory viability for implementing bare-metal software-defined infrastructures in production enterprise environments.

Chapter 6 presents Comet, an in-transit shuffle service that improves the stability and performance of large-scale data analytics. Comet enhances the stability of Spark by offloading CPU and the I/O burden to the external shuffle cluster. Comet also enhances the fault tolerance of shuffle data storing with a distributed file system instead of a local disk, which does not have single-point failures. Furthermore, with better stability and destination-based aggregation mechanisms, Comet can achieve significant performance improvement compared to vanilla Spark.

Chapter 7 presents the conclusions of the thesis and proposes possible directions for future work.

Finally, Appendix A complements the thesis's main contributions by providing an understanding of big data processing systems behavior and the tradeoffs associated with the use of different architectural designs and processing frameworks. It further motivates the thesis research focused on in-memory processing systems with deeper memory hierarchies.

Chapter 2

Background and Related Work

Big data computing frameworks running in the cloud environment at scale provide opportunities to a wide range of domains. However, delivering high-performant and effective services requires efficient computing resource management and data orchestration of cloud computing frameworks. Unlike small-scale data analytics, the data analytics at extreme scale easily suffer the single point failure from data skew and overloading of computing resource. As a result, it is essential to efficiently manage data and computing resources and working with high stable data orchestration mechanisms, which motivate this thesis research.

Typical data warehouse services in e-commerce provide core data joining and aggregation services to various businesses. As they offer core data and services to many customers with explicitly defined SLAs, delivering quality of service requirements is critical. For example, data warehouse services in the e-commerce industry are typically composed of data analytics sets dealing with TBs to PBs of data, running with thousands of computing nodes. In these data analytics, since most of the data consumer services are dependent on the processed data from the data warehouse service, data analytics require fast processing while with a large scale of data. The main research challenge for these data analytics is building a stable and high-performance persistent data orchestration to process a large amount of data efficiently.

Business intelligent data analytics usually deal with complex data from different data sources with complex computing logic. At runtime, the late stage of data analytics could reuse a large amount of intermediate data generated by early stages. For example, large e-commerce companies suffer frequent price denial-of-service (DDoS) attacks, such as coordinated product price crawling. Therefore, to efficiently support

such data analytics, better management of intermediate data inside data analytics is required. This chapter provides the foundations of big data frameworks to support these issues and explores state of the art on key issues to enable existing big data computing frameworks to deliver performant, stable, and efficient data orchestration for large-scale cloud-based workloads.

2.1 Big Data Computing Frameworks

The proliferation of digital data provides new opportunities in all areas of science, engineering, and industry. About 2.5 quintillion bytes of data [11] is generated every day through the Internet. However, the increasing volume and rate of data [12], along with the associated costs in terms of latency and energy, quickly overpower and limit data analytics applications' ability to leverage this data in an effective and timely manner. The co-design process enables scientists to reason about the rich design spaces available in software and hardware, which is fundamental for constructing the next generations of cyber-infrastructure. While system architecture trends include larger core counts, deeper memory hierarchies (e.g., larger amounts of non-volatile memory), and constrained power budgets, application formulations for data analytics are trending toward in-memory processing solutions. Nevertheless, as current solutions for data analysis pipelines require complex solutions involving different specialized platforms and configurations depending on application requirements, it is not clear how to effectively realize and optimize them in these ongoing architectures. Further, ongoing processor architectures, non-volatile technologies such as Intel Optane NVMe, and the advances in integrated silicon photonics promise systems capable for delivering off-node non-volatile memory latency and bandwidth comparable to PCIe-based in-node access [13], which is essential for realizing actual software-defined infrastructures.

Current data analysis workflows may require different types of analytics, where some are more appropriate for batch-oriented processing (e.g., Hadoop), micro-batch processing (e.g., Spark), or near real-time processing (e.g., Storm, Flink, Heron). However, there is an increasing interest from both scientific and industry communities to move to in-memory approaches for a broader range of analytics. Existing work [14] has

shown that the performance of storage devices used in Hadoop deployments impacts the execution time of data and compute-intensive applications, and that the execution of specific graph-based workloads is more energy efficient with Spark (i.e., Spark GraphX) than with Hadoop (i.e., Hadoop Giraph) [15].

We studied behaviors and tradeoffs for two of the main distributed processing systems for big data analytics: Apache Hadoop and Spark, which are currently the most widely used open-source parallel processing frameworks for big data analytics. A comprehensive study of representative workloads for Hadoop and Spark using different storage technologies and design choices and an exploration of the potential of software-defined infrastructure for big data processing frameworks, with a concentration on the non-volatile deep memory hierarchy, are provided in Appendix A. This study motivated us further to focus on in-memory processing frameworks as addressed in the following sections.

2.1.1 MapReduce Workloads

A large body of literature in this area is focused on MapReduce's workloads and runtime instead of hardware/software co-design issues. A comprehensive study of a MapReduce workload analyzed a ten-month workload trace from the Yahoo! M45 supercomputing cluster [16]. However, most of existing studies focus on benchmarks instead of real production workloads [17, 18, 19]. Other work has focused on specific issues, such as job and task run times [16, 17, 18, 19, 20], Map vs. Reduce tasks [16, 18], CPU and memory demand [21], I/O and data locality [18, 22], and cluster utilization, failures, and energy consumption [16, 22]. Models for MapReduce workloads have also been developed [16, 17, 18, 23, 24]; however, their primary focus is on job completion times. Furthermore, different MapReduce simulators have been developed [18, 24, 25, 26, 27] that mainly focus on simulating the execution of synthetic workloads.

Other recent research efforts, such as the Aloja project [28], aim to explore upcoming hardware architectures for big data processing and reduce the Total Cost of Ownership (TCO) of running Hadoop clusters. Aloja's approach is to create a comprehensive open public Hadoop benchmarking repository based on empirical executions. It allows

for comparisons between not only software configuration parameters, but also current hardware (e.g., SSDs, Infiniband networks).

2.1.2 Energy Efficiency

Existing literature has studied the optimization of energy efficiency at the cluster level for Hadoop MapReduce [29] by dividing the cluster into two zones: a "hot" zone with frequently used data on higher performance processors and a "cold" zone for low-frequency access data with a large amount of disks. Goiri et al. [30] introduced GreenHadoop, which is powered via solar array and uses the electrical grid as backup. Lang et al. [31] came up with the All-In-Strategy (AIS), which toggles nodes on or off based on the amount of Hadoop jobs in the queue. Amur et al. [32] presented the power-proportional distributed file system (Rabbit) that divides the nodes of a cluster into primary nodes (for primary replicas) and secondary nodes (for other replicas), which also provides a higher level of fault tolerance. Chen et al. [33] implemented Berkeley Energy Efficient MapReduce (BEEMR), an energy-efficient MapReduce workload manager motivated by the empirical analysis of real-life MapReduce with Interactive Analysis (MIA) traces at Facebook. BEEMR classifies jobs into either an interactive zone, a full-power-ready state and batch zone, and a low-power state in order to optimize energy efficiency.

Dynamic Voltage and Frequency Scaling (DVFS) has been used to improve energy efficiency. Tiwai et al. [34] addressed CPU frequency tuning based on application type to decrease energy consumption. Wirtz et al. [35] compared three different CPU frequency policies for Hadoop: 1) a fixed frequency for all cores during execution, 2) a maximum CPU frequency for map and reduce functions and a minimum CPU frequency otherwise, and 3) an adjustment to the CPU frequency while satisfying performance requirements. Li et al. [36] proposed temperature-aware power allocation (TAPA) to reduce energy consumption and Shadi et al. [37] recently explored DVFS usage in Hadoop clusters.

Current research also addresses hardware and data optimization to improve energy efficiency. Chen et al. [38] studied the energy consumption for Hadoop applications

in three dimensions: the number of nodes, the number of HDFS replicas and different HDFS block sizes, and data compression methods that may improve energy efficiency [22]. Yigitbas et al. [39] proposed an Intel Atom processor-based Hadoop cluster for better energy efficiency than an Intel Sandy Bridge processor-based Hadoop cluster with I/O-bound MapReduce workloads. Luo et al. [40] evaluated CPU frequency, memory mode, and different storage parameters for compute intensive, storage intensive, and I/O intensive applications.

The literature summarized above can be complemented with research on MapReduce schedulers [21, 41, 42, 43] and existing work on MapReduce frameworks for many-core systems (e.g., the Intel Xeon Phi platform) focusing on SIMD support and performance issues [44]. Although the thesis focus on performance and stability, considering this issue is an important aspect for future work as described in Chapter 7.

2.2 In-memory Processing Frameworks

The rapid growth of the sheer quantity of digital data generated every day through the internet has motivated the science, engineering, and industry communities to develop big data processing frameworks. As discussed earlier, Apache's Hadoop is one of the most popular big data frameworks and is based on the MapReduce model for distributed processing of large scale datasets using an affordable infrastructure. However, the increasing volume and rate of data [12], along with the associated costs in terms of latency, quickly limit the ability of data analytics applications to leverage this data in an effective and timely manner. For example, new requirements in different areas of science and engineering for supporting quick response analytics (e.g., machine learning algorithms) of data being produced or collected at high rates (e.g., real-time streaming data) are pushing application formulations for big data toward in-memory processing solutions. Apache Spark has become increasingly popular due to its in-memory and directed acyclic graph (DAG) execution engine. Further, Spark provides the abstraction of resilient distributed datasets (RDD), which is essential to delivering high-performance capabilities while remaining compatible with existing Hadoop

ecosystems, e.g., the Hadoop distributed file system (HDFS) and a number of high-level APIs in Scala, Java, and Python. Another example is Alluxio (formerly known as Tachyon [10]), which delivers a distributed file system for reliable in-memory data sharing. However, the increasing amount of memory and power required to run complex data-centric applications and workflows using in-memory processing systems are pushing the community to explore novel system architectures (e.g., deeper memory hierarchies) and new application formulations. Furthermore, DRAM memory represents a large portion of the investment for building datacenter IT infrastructure, especially when they target in-memory processing systems.

Understanding the limitations of current in-memory processing frameworks and the ability to trade off response time, power consumption, and infrastructure cost is an important concern; however, exploring system design choices for enabling next generation infrastructure to effectively support these platforms and, in turn, co-designing new software frameworks is paramount. Ongoing processor architectures, non-volatile technologies such as Intel Optane NVMe, and advances in integrated silicon photonics promise systems capable of delivering off-node non-volatile memory latency and bandwidth comparable to PCIe-based in-node access [13], which is essential for the implementation of software-defined infrastructures. Software-defined infrastructure is based on the concept of resource desegregation. Previous work has explored requirements for building disaggregated datacenters [45, 46, 47, 48], taking into consideration flash [49] and DRAM memory [50, 51] resources as well as RDMA technologies [52]. Although network fabric latency and bandwidth has been a major blocker for the adoption of software-defined infrastructures, recent technological advances such as NVMe over fabrics [53], Intel Omni-Path, and Mellanox Quantum 200G HDR exemplify a step forward toward this vision. Rack scale design (RSD) architecture is one of the realizations of software-defined infrastructure. Intel RSD uses a high-performance PCIe to reduce data transmission time within the same rack. However, the data transmission time across different racks is still challenging. Next-generation SDI is expected to deliver lower latency and higher bandwidth interconnects for desegregated resources (e.g., compute, memory, and storage) within datacenters. In Appendix A we explore the potential of

software-defined infrastructure for HDFS targeting different storage technologies (e.g., NVMe); however, exploring the potential of next generation software-defined infrastructure for in-memory frameworks has become a critical concern, which is addressed in chapter 3.

Existing studies in the literature have aimed at improving the performance of MapReduce and in-memory computing frameworks, such as Spark, using different network technologies and optimization strategies. Sur et al. [54] evaluated the impact of high-speed interconnects for datacenters such as Infiniband and 10Gb Ethernet for supporting Hadoop distributed file systems (HDFS). This work also revealed that Infiniband and 10Gb Ethernet is more suitable for systems based on solid-state drives than spinning hard disks. Islam et al. [55] addressed the design of HDFS using remote direct memory access (RDMA) over Infiniband. They evaluated the performance of Gigabit Ethernet and IP-over-Infiniband on the QDR platform, showing that InfiniBand has much better performance than Gigabit Ethernet. Lu et al. [56, 57] proposed a high-performance RDMA approach based on accelerating the shuffle stage for Spark and evaluated the performance of RDMA with InfiniBand for different workloads in Spark. Kamburugamuve et al. [58] accelerated Apache Heron using InfiniBand and Intel Omni-path, which can increase the speed of network throughput for real-time big data frameworks. Gupta et al. [59] used Intel Omni-Path to accelerate big data frameworks such as Spark and Hadoop.

There are also proposals in the existing literature of optimizations for Spark using GPU and FPGA. Manzi et al. [60] explored the potential of GPU to accelerate different workloads (WordCount, K-Means and Sort) for the Spark framework. Li et al. [61] developed HereroSpark, which can utilize the GPU to increase the speed of machine learning algorithms for Spark. Rathore et al. [62] designed a GPU-based Spark system for processing real-time large-size city traffic video data. Gupta et al. [59] proposed a design using Xeon and FPGA to increase the speed of Spark and Hadoop.

Current research has also addressed algorithm optimization to improve the performance of Spark. Davidson et al. [63] used RDD compression to increase the speed of Spark, which increased the CPU utilization while reducing the I/O pressure. Chaimov

et al. [64] developed a file pooling layer to improve metadata performance in Spark. The utilization of various storage technologies to improve the performance of big data frameworks has been previously evaluated. Kambatla et al. [65] and Moon et al. [66] explored the potential of using SSD in HDFS for accelerating Hadoop deployments. Moon et al. also studied how to accelerate Hadoop using multiple disks.

2.3 Geo-distributed Data Centers

To process large volumes of data, big data applications require a large number of computing resources. However, expanding the size of the data center computing clusters within the same geo-location is challenging for large-scale organizations due to several reasons, including space and power supply limitations. For example, the experiments described in chapter 4 are conducted on a computing cluster in Beijing, with about 3,700 servers to support warehouse applications, which has reached the data center's limits. However, these warehouse applications support the data supply for all departments and, as a result, the demand to reduce their execution time is increasing. Furthermore, the cost of expanding the computing capabilities by purchasing more servers can substantially increase the total cost of ownership of enterprise data centers [67, 68], as considerable infrastructure investments might be needed to support additional resources. To reduce the total cost of ownership of enterprise data centers, we harvest spare computing resources from existing computing clusters.

Current research efforts [69, 70, 71, 72, 73] mostly focus on low-bandwidth cloud environments. The data transmission time for a task across data centers can determine the execution time of applications. In most of these cases, data distribution and caching are reasonable solutions; however, with existing fast fabric interconnections, we integrate computing clusters as an elastic computing resource pool across geo-distributed data centers. Existing work has proposed mechanisms for harvesting spare computing resources within the same data center [74, 75]. As opposed to existing work, and based on the observation from the analysis above, this thesis aims at efficiently utilizing computing resources across geo-distributed data centers. Existing work has also explored

the potential and methods for utilizing geo-distributed data centers with limited network bandwidth [69, 71, 72, 73]. However, these methods cannot meet the performance requirement for large enterprise data-centric services, which can consist of hundreds or even thousands of applications.

Chapter 4 explores the potential of the elastic computing cluster abstraction across data centers, explores scheduling strategies for data warehouse applications, and provides meaningful evaluations of the proposed approach in real-world, large-scale computing cluster deployments.

2.4 Resource Management for In-memory Big Data Analytics

The increasing generation of data at high rates is creating new requirements for large-scale enterprise applications from different business areas. For example, faster big data analytics are critical for supporting speedier business intelligence and for leveraging the recent advances in machine learning. This trend is pushing big data analysis systems toward high-performance in-memory processing solutions. However, in-memory big data processing systems impose a significant memory burden on the data center enterprise, especially in terms of capital and operation costs.

To deliver high performance, the computing infrastructure must provide sufficient DRAM memory to hold large amounts of intermediate data. In addition, it is common to see a data skew in production applications as the data size increases. A common practice in production environments is to over-allocate the memory assigned to the data workers. This lowers overall memory utilization and increases the data center's memory requirements. At the same time, real large-scale production data centers are facing several challenges, such as uneven resource utilization levels between CPU and memory and the expensive cost of DRAM memory. As a result, these factors are limiting how data centers can deliver high-performance in-memory big data applications efficiently.

Both academia and industry have devoted significant efforts to design and implement data center architectures to optimize the use of novel hardware technologies.

However, advances in processor, memory, storage, and network technology show different growth and demand trends. Therefore, data center architectures that are built based on the vision of server-centric resources face significant challenges with adopting the latest hardware innovations quickly [76, 77]. Conversely, disaggregated data center designs allow different resources to be scaled independently, enabling the faster adoption of novel hardware developments at a lower cost. In turn, enterprise applications can effectively obtain full value from an advanced disaggregated infrastructure and its associated investments. This is especially beneficial for supporting in-memory big data frameworks. However, to address the challenges associated with upgrading data center infrastructures at affordable cost, co-designing applications with innovative memory system architectures is essential. To address these challenges, chapter 5 provides an in-memory big data processing system using disaggregated memory resource. The system is based on a state-of-the-art in-memory big data framework and a novel in-memory distributed file system.

Existing research has considered disaggregated resources with advanced infrastructure to increase the performance and compute resource efficiency for big data frameworks. Existing research [57, 56, 64, 78] has explored the potential of RDMA for Spark. Gao et at.[47] address the network requirements of disaggregated data centers, and compare the performance loss of different network latency and bandwidth. Rao et al. [79] compare Spark SQL's performance with different memory bandwidth. The industry has already proposed disaggregated data center architectures with PCIe and fast fabric interconnects such as the Intel Rack Scale Design (RSD) for rack-scale disaggregation [80], HP's "The Machine" concept [81], and Facebook's proposed disaggregated data center [82].

Existing related research efforts have also explored the potential of remote page caching [83, 84, 85, 86]; however, unlike existing approaches, we increase the memory capacity of the data processing cluster with large-volume DIMM-based persistent memory (PMEM) [87] and the abstraction provided by the proposed in-memory distributed file system co-designed with PMEM. With our approach described in chapter 5, we implement a disaggregated memory pool without modifying the kernel. Furthermore,

instead of remote page caching, our implementation can transparently employ disaggregated memory resources in a data processing cluster using Spark [88] and the proposed in-memory distributed file system.

2.5 Shuffling in Data Analytics

A large amount of data is stored in the companies' data warehouse, which is from different businesses and also used for different purposes. Besides, the rapidly increasing data not only add a heavy burden to the storage systems but also addresses a series of challenges to data analytics frameworks, e.g., Spark [88], Presto [89], Flink [8] and Map-Reduce [90],

In order to process large amounts of data efficiently, modern services use data analysis frameworks to process large amounts of data across hundreds, even thousands of nodes concurrently. These frameworks are not only involved in ETL services but also play an essential role in Business Intelligence services, e.g., data pre-processing, machine learning. Further, since the cloud ecosystem has become more and more important in building modern services, it is essential to make data analytics frameworks compatible with the cloud ecosystem. To achieve better parallelism, the modern data analytic frameworks divide a large task into small tasks [91, 92, 93, 94, 95]. With this mechanism, the data analytic can process a large amount of data efficiently. However, the n-to-n communicating operation can be the primary performance and stability bottleneck in these frameworks. For example, the shuffling operation can be the main drawback in Spark because of the high cost of network transmission, data skew, and stability issues.

The existing optimization for data-intensive analytics mostly focuses on data source optimization [96, 10, 97]. However, these works either assume the cost of data source can be the dominating cost of the whole job, which is not always true in the production environment. For example, one of major I/O cost in the core data warehouse applications of a large e-commerce enterprise is shuffling, like data sorting and joining. In these scenarios, the cost of the data source is not the dominating cost of jobs. The cost of n-to-n communication intermediate data can be the main cost of these jobs.

As we observed in production services [98], shuffling is one of the most expensive operations in current data analytics, which is a common scenario across different frameworks. For example, the modern distributed computing frameworks suffer performance degradation because of the substantial cost of shuffling [99, 100]. Recent research on shuffling has considered how to optimize the performance of shuffling through I/O performance optimization.

2.5.1 Shuffling Optimizations

Existing work [98, 101, 102, 103] proposes performance optimizations of Spark shuffling with wither data aggregation, or high-performance memory pool. Other performance improvements of big data analytics framework are based on improving the access efficiency of shuffle data and spill data. Themis [104] reduces the number of I/O access for shuffling to spinning disks to improve the performance of shuffling. TritonSort [105] implements the efficient and high preferment distributed sorting system based on Themis. However, since Themis is specifically designed for sorting problems, it can not cover general shuffle cases in data analytics frameworks.

Sailfish [103] proposes the concept I-file to aggregate the shuffle data, which decrease the number of I/O requests, to optimize the performance of shuffling for Map-Reduce. However, Sailfish uses customized HDFS, which has an extra cost on modifying the under-layer distributed file system. COSCO [106] build efficient shuffle service with warm storage [107], which manages and aggregates shuffle data with independent shuffle scheduler. Riffle [101] leverages the mapper side aggregation shuffle service for Spark. However, as we notice in the production environment, the overuse of I/O causes the instability of the computing node, which decreases the stability of the whole cluster. Furthermore, Riffle ensures fault tolerance with keeping the original copy of shuffle data, which is not space-efficient. In some worse cases, like severe data skew, it can occupy a large amount of space and I/O bandwidth of local disk. Spark presents the sort based shuffle service [63] instead of the original hashed based shuffle service, which aggregates the shuffle data inside the same JVM. With sort based shuffle service, Spark enhances the stability and performance of large-scale data analytics.

The optimizations proposed in existing literature involve expensive software cost or hardware upgrading, which is expensive for production environments.

2.5.2 I/O Challenges

Generally, we can divide the I/O cost of Spark into data source cost and intermediate data cost. Most of the time, data analytics access the data layer while reading/write intermediate data from the internal data structure. Dremel [96] has explored the high-performance nested format for data analytics frameworks. Different nested high-performance data formats, which are related to Dremel, are widely used in the companies' data warehouse with different frameworks, e.g., [108, 88, 89]. Besides, existing works reduce the gap between cloud storage systems and data analysis frameworks. For example, Alluxio [10] builds the unified data engine to connect storage systems with data analytics frameworks with a caching mechanism. These works solve most of the existing challenges related to data source, storage system, and data analytics frameworks. These solutions (e.g., Hive, Spark, and Presto) can be leveraged to build a stable and efficient service in a production environment; however, there are critical performance and unstability issues that are related to intermediate data.

Compared to the research related to the data source, there are still lots of remaining challenges to building an efficient intermediate data layer in a large-scale cloud environment. The modern data analytics frameworks use memory to store intermediate data for better performance. For example, one of the most significant innovations of Spark is storing intermediate data in memory. With this mechanism, Spark can have much better performance than Map Reduce in iterative jobs with the DAG engine. However, the hungry framework asks for more and more memory to meet the requirement of fast computation, but we can not always provide enough memory to this data analytics. The data analytics sometimes suffer performance degrading and failure because of the lack of memory.

Furthermore, the frequent disk spilling results in soft failures, like time out and failure of requests. The I/O overhead for shuffle spilling significantly decrease the performance of in-memory data analytics. For example, Spark uses a sort based shuffle

service. Data analytics suffers $O(nlogn)$ of usage to local disk, which is affordable with memory access but not affordable with spinning disk access. Thus, the data analytics frameworks have significant performance degradation or failure with inefficient memory [109]. Different from the data sources, it is expensive to build high-reliable middleware for intermediate data in a large-scale environment. For example, Crail [102] uses NVMe and RDMA to build high-performance intermediate storage for the current Spark shuffle service. DMO [98] build hyper-converged infrastructure with persistent memory [110] and RDMA, to optimize the shuffling and RDD caching performance of Spark. What is more, Riffle [101] built an external aggregation scheduler to optimize the performance of Spark shuffling. To do so, Riffle aggregates the shuffle data at the mapper side, which increases the overloading of disk I/O at the mapper side. Sailfish [103] proposes a new Map-Reduce framework with aggregating intermediate data. However, the Sailfish require the specific file system, which supports multi-writer and multi-reader. Some of the most popular cloud distributed file systems, e.g., HDFS [9] and S3 [111], do not support the multi-writer. As opposed to existing work, it is necessary to build a cloud-native shuffle service, which is affordable to current hardware infrastructure and software infrastructure, like hard disk drive and HDFS, with succinct architecture.

Chapter 6 focuses on the design and implementation of this in-transit shuffle service, which is stable and cost-efficient in the cloud environment.

Chapter 3

Exploring the Potential of In-memory Big Data Frameworks

This chapter focuses on understanding the behaviors and limitations of current in-memory frameworks, thus leading to insights regarding design choices toward next-generation software-defined in-memory frameworks. Our empirical experimental evaluation focuses on persistence methods for in-memory processing frameworks, i.e., Spark and the use of Alluxio. It revolves around an empirical evaluation of Spark persistence methods using different storage technologies and Alluxio, which answers questions related to behaviors and limitations of current in-memory processing systems, provides meaningful data points to better understand requirements and design choices for next-generation software-defined infrastructure and explores the use of disaggregated off-rack memory and NVMe via simulation due to the limitations of current network fabric interconnects.

3.1 Understanding In-Memory Big Data Frameworks

3.1.1 Spark Persistence

Spark RDD persistence (or caching) is one of Spark's key capabilities, allowing it to deliver low latency and high performance. It allows users to store intermediate RDD into memory, disk, or a combination of memory and disk and reuse them in other actions on that dataset. Spark persistence technology can dramatically increase the speed of Spark applications (often by more than 10x [112]) with proper configuration and using programming best practices, especially in iterative applications.

Spark uses several approaches to RDD persistence. Memory can be used to store

RDDs as de-serialized Java objects when they have enough memory space to store intermediate RDD datasets. Because RDD datasets are stored as de-serialized Java objects in memory, the required memory space has to be estimated based on an "expansion Index" and the size of the original dataset. Using memory with de-serialized Java objects is the fastest Spark persistence mechanism when sufficient memory space is available. Spark provides three different approaches to persist RDDs when insufficient memory space is available. The most straightforward way is to use block storage (e.g., disk) as an addition to memory. Spark can also store a whole RDD into block storage; however, in most cases, this approach is slow. RDDs can also be stored as serialized Java objects, which is less CPU-efficient compared to using de-serialized Java objects. Furthermore, Spark can also use off-heap memory (i.e., memory that is outside of executors) as storage space for persistent RDD.

3.1.2 Spark Persistence Memory Management

Spark memory management is based on Java Virtual Machine (JVM) memory management. Spark divides the memory into execution and storage, based on the different memory usages. The memory used for computation in shuffles, joins, sorts, and aggregations is considered execution memory and the memory used for caching (Spark RDD persistence with memory) and internal data transfers within clusters is considered storage memory. Former versions of Spark (version <1.6) implemented persistence using static memory management, which used differentiated memory spaces for execution and storage memory. With this approach, storage memory cannot use execution memory even when the execution memory is not utilized. To resolve this drawback, Spark introduced the current unified memory management, which merges execution and storage memory.

Important aspects of unified memory management in Spark include the following:

1. Spark reserves 300 MB of memory for the system as default.

2. Unified memory represents the execution and storage memory, which can be customized ($spark.memory.fraction$, default 0.6).

3. The rest of the heap memory includes user data structures, internal metadata in Spark, and safeguarding against out-of-memory errors (default 0.4).

In comparison with static memory management, unified memory management allows storage memory to use more memory when demand for execution memory is not high. More importantly, execution memory can evict storage memory if there is not enough memory in the unified space, while storage memory has minimum space protection.

3.1.3 Spark Data Locality and Delay Scheduler

Data locality plays an important role in Spark. It specifically organizes the data into three main levels based on locality:

1. Data can be fetched from the same JVM as the running executor.
2. Data can be fetched from the same node (data from other executors in the same node, e.g., HDFS and Alluxio data in the same node).
3. Data can be fetched from the same rack of the cluster.

The first option is clearly the fastest, the second a bit slower, and the last one is the slowest because datasets must be sent over the network. Spark checks whether there are any available datasets in the same JVM as the running executer. If Spark cannot find datasets in the same JVM as the running executor, then it checks for data within the same node; otherwise, it checks for data in the same rack.

3.1.4 Alluxio

Alluxio (formerly known as Tachyon) is a distributed in-memory storage system; it has an API similar to HDFS but aims at accelerating big data frameworks by using memory technology as the main substrate for implementing the distributed file system. In this thesis, we use Alluxio to accelerate Spark executions and implement persistency models that have the potential for supporting data coupling between multiple Spark applications. However, there are some important differences between Alluxio and Spark

persistence models. On the one hand, predictability (e.g., estimating execution time) is more complex in Spark as its persistence mechanisms, such as unified memory management, evict storage memory when possible. On the other hand, Spark cannot easily share intermediate data (i.e., RDDs) with other applications and platforms. In this contribution, we evaluate and compare the performance of Alluxio and Spark persistence models to understand software design issues that should be considered in future implementations targeting next-generation software-defined infrastructure.

3.2 Evaluation Methodology

This section describes three aspects of the experimental evaluation methodology. First, we evaluate Spark RDD persistence using different methods and storage technologies:

- Memory (MEMORY_ONLY).
- Memory with serialized Java objects (MEMORY_ONLY_SER).
- Disk only (DISK_ONLY), using NVMe and hard disk.

Please note that storing RDD datasets with serialized Java objects requires trading off CPU utilization and I/O speed. We explore the need for using serialized Java objects when using NVMe technology. Second, we compare the traditional disk-based distributed file system that uses HDFS with an in-memory virtual distributed storage system (Alluxio). Alluxio is not intended to replace persistent distributed storage systems; rather, it provides a faster intermediate storage layer that interfaces with other file systems (e.g., HDFS, Amazon S3) and big data frameworks (e.g., Spark) to speed up data access. However, we compare these two approaches to understand the potential tradeoffs between cost and performance. Third, we study Spark RDD persistence with Alluxio. We use MEMORY_ONLY and MEMORY_ONLY_SER as storage levels for Spark, and Alluxio uses memory as its only storage device. Finally, based on the results obtained from the empirical evaluation, we explore via simulation different scenarios using remote memory and NVMe for Spark data persistence.

3.2.1 Testbed and System Configuration

The empirical executions were conducted on the NSF-funded research instrument computational and data platform for energy-efficiency research (CAPER). CAPER is an eight-node cluster based on a SuperMicro SYS-4027GR-TRT system with a flexible configuration. The servers have two Intel Xeon Ivy Bridge E5-2650v2 (16 cores/node) and the configuration used in this work includes 128GB DRAM, 1TB Flash-based NVRAM (Fusion-io IoDrive-2), 2TB SSD, 4TB hard disks (as a RAID with multiple spindles, as recommended by best practices), and both 1GbE and 10GbE and Inbiniband FDR network connectivity. This platform mirrors key characteristics of datacenter infrastructure, which will allow us to extrapolate our models to larger systems and make projections.

We configured the big data framework and distributed storage file system as baselines using commonly used and balanced configurations without specific optimization. The characteristics of the system configuration are described as follows. Spark version 2.0 was deployed using YARN. One server was configured as Master and six servers were configured as Slaves. Hadoop version 2.7 (with HDFS) was deployed using YARN. One server was configured as the Name Node and six servers were configured as Data Nodes. Alluxio version 1.5 was deployed on seven servers, with one server configured as Master and six nodes configured as Workers. The system was configured with 24 executors, using 4 cores each. The JVM memory was set to 20GB and 16GB for Spark Driver and Executor, respectively.

3.2.2 Workloads and Data Set

The evaluation is focused on three types of benchmarks:

1. LineCounter (I/O intensive application).
2. WordCount-reduceByKey (I/O and network intensive application).
3. WordCount-groupByKey (network intensive application).

All of these three workloads are based on real data from Wikipedia in text format.

Furthermore, LineCounter is the most I/O intensive and WordCount-groupByKey is the most network intensive of these benchmarks. The executions are conducted using data sets of different sizes for different benchmarks, from 10GB to 250GB (50GB-250GB for LineCounter, 10GB-250GB for WordCount reduceByKey and 10GB-40GB for WordCount groupByKey).

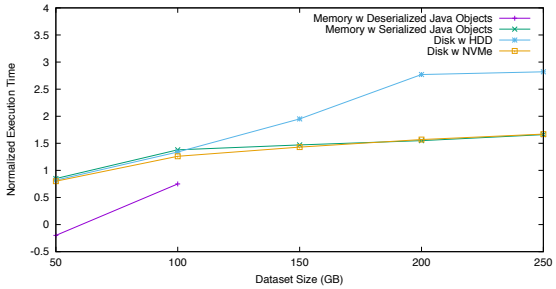
3.3 Experimental Results

3.3.1 Spark RDD Persistence

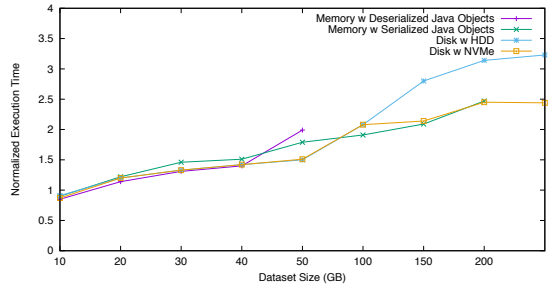
This section explores the impact on Spark RDD persistence performance of using different storage technologies. Although Spark RDD persistence can be configured using different storage levels, including memory, Spark clusters cannot always provide sufficient memory capacity for co-locating the application and persisted datasets. When insufficient memory is available in the system, Spark uses additional storage levels such as the hard disk to store RDDs as serialized Java objects, i.e., it extends the data memory space with storage devices. Because the serialization of Java objects requires significant CPU resources, there is a tradeoff between memory space and application execution performance.

There is an "expansion Index" for de-serialized Java objects, which means that the required storage space for RDDs is bigger for de-serialized Java objects than for serialized Java objects. For the particular case of LineCounter, the expansion index is 2.09. We assigned 480GB of memory to Spark executors, thus providing Spark with 288GB of available unified memory. As a result, the experiments discussed in this section using Spark persistence on disk persists up to 250GB of data; however, only 100GB of data are persisted in experiments using Spark RDD persistence with de-serialized Java objects in memory.

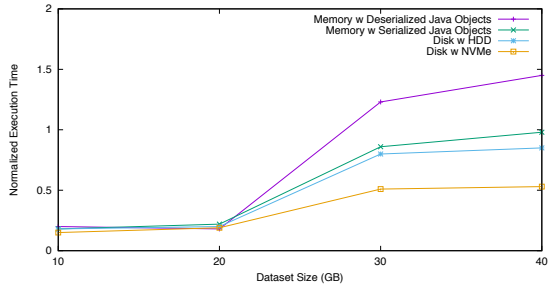
According to the results shown in Figure 3.1a, the execution time of LineCounter using RDD persistence with de-serialized Java objects is the shortest among the tested methods because Spark does not de-serialize the RDD datasets during the execution. The average task execution time for each task is about 20 ms; however, Figure 3.2



(a) LineCounter



(b) WordCount-reduceByKey



(c) WordCount-groupByKey

Figure 3.1: Normalized execution time of different benchmarks using Spark RDD persistence and different storage technologies

presents the normalized average task execution time.

Figures 3.1b and 3.1c show the execution time of WordCount-reduceByKey and WordCount-groupByKey. Both of them have same result with same input data, but WordCount-groupByKey has much more shuffle data than WordCount-reduceByKey, which is shown in Table 3.1.

Figure 3.1b shows that WordCount-reduceByKey with persisted deserialized Java Objects provides the best performance. However, as the data size increases, the cluster can not offer enough storage memory. This causes performance degradation with 50 GB and larger data sets. Furthermore, the experimental evaluation shows that NVMe provides high performance, which can match the performance of memory with serialized

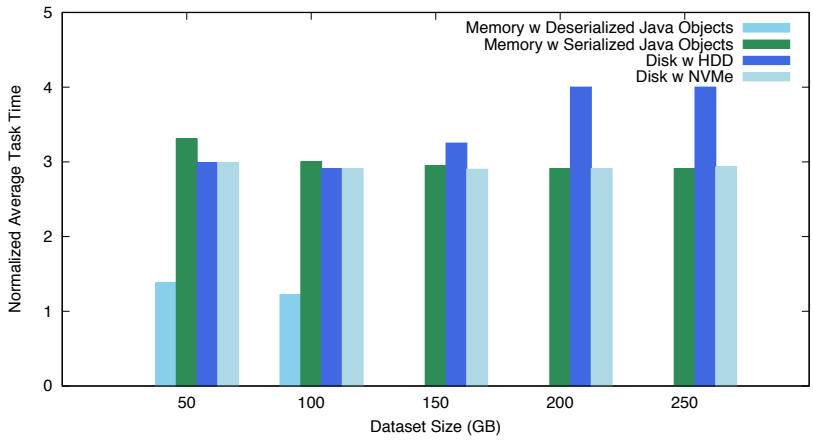


Figure 3.2: Normalized AVG LineCounter task execution time using Spark RDD persistence and different storage choices

Java Objects.

Data Size	WC-reduceByKey (GB)	WC-groupByKey (GB)
10 GB	0.76	2.4
20 GB	1.53	4.6
30 GB	2.3	6.7
40 GB	3.1	8.7

Table 3.1: Shuffle r/w size of WordCount (WC)

Compared to WordCount-reduceByKey, WordCount-groupByKey shows quite low performance. This is not only because of the shuffle size. GroupByKey is a quite expensive task because it sends data to the assigned executors, which consumes most of the memory and network resources. Once the data set increases to 30GB, Spark evicts part of the data from memory. Thus, Spark needs to calculate the required data block again. As a result, it is easy for GroupByKey to trigger the delay scheduler. Table 3.2 shows the number of off-node blocks that are needed to be fetched for WordCount-groupByKey.

While the execution of LineCounter using Spark RDD persistence with serialized Java objects in memory is slower than with de-serialized Java objects, its execution with serialized Java objects on NVMe has performance similar to that using memory for persisting the serialized Java objects. Although these results might be counterintuitive, the de-serialization of Spark RDD uses most of the CPU resources in Spark tasks. In

other words, the I/O throughput is not the bottleneck for persisting RDD datasets with serialized Java objects on NVMe (or memory).

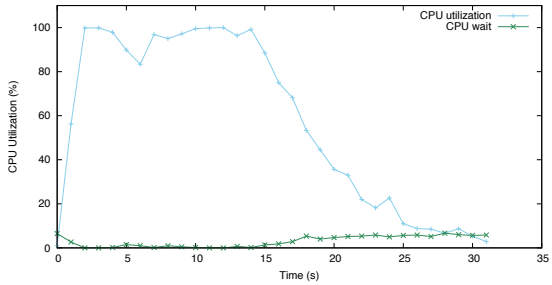
Data Size	Memory	Memory w Ser	HDD	NVMe
10 GB	0	0	0	0
20 GB	0	0.4	0.4	3
30 GB	21.2	1.4	3	3
40 GB	31	4.6	1.2	4.2

Table 3.2: Off node data fetching of WordCount-groupByKey

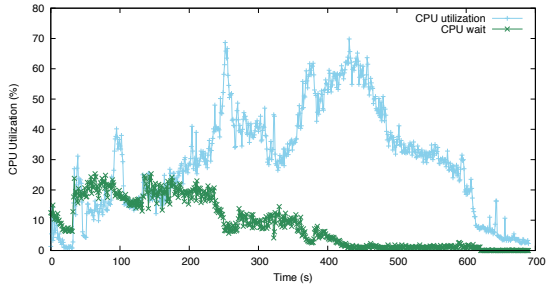
The execution of workloads using Spark RDD persistence with serialized Java objects on hard disk drives is the slowest method, as expected. However, this is not necessarily only because hard disk drives are slower than memory and NVMe. As shown in Figure 3.2, the processing time for each task increases significantly as the dataset size increases from 50GB to 250GB (specifically from under 1 second to about 10 seconds). There are two main reasons for this workload slowdown: (1) as datasets become bigger and bigger, the operating system cannot offer sufficient memory buffer to accelerate the data fetching, and (2) as tasks require longer and longer processing time, it eventually exceeds the threshold of the delay scheduler, which means that it will fetch data from a remote node instead of a local one.

We choose LineCounter here to analyze different persistence technologies. Figures 3.3a and 3.3c show that the CPU utilization of the cluster is almost 100% at the beginning and middle stages of the workload execution. This is because Spark consumes most of the CPU resources in de-serializing the RDD datasets. Thus, the bottleneck of Spark persistence is not the I/O throughput with NVMe and memory (with serialized Java objects). This indicates that NVMe is a good candidate to replace traditional storage when the amount of memory capacity is constrained. Furthermore, Spark has almost the same performance with memory and NVMe combined with compression (columnar compression in Spark) while NVMe has a lower cost than memory.

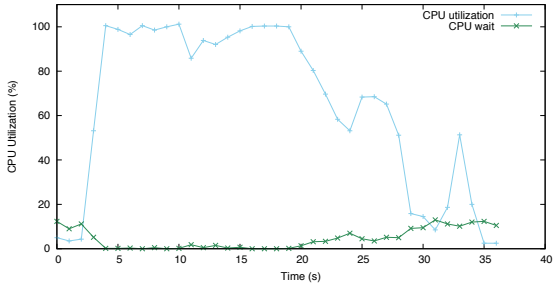
Figure 3.3b shows that the workload CPU utilization ranges from 20% to 60% using Spark RDD persistence with a hard disk drive. This indicates that the I/O throughput is the bottleneck when using the hard disk, while the bottleneck is the processor when



(a) Memory (Java serialized objects)



(b) Hard disk



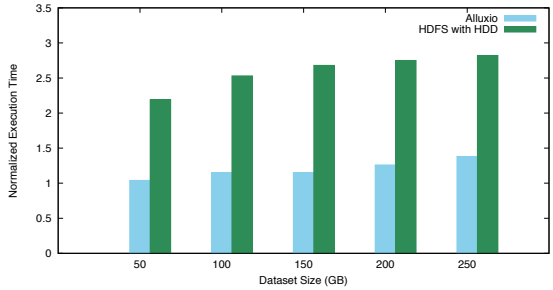
(c) NVMe

Figure 3.3: CPU utilization using Spark RDD persistence with different storage technologies and 200 GB data sets

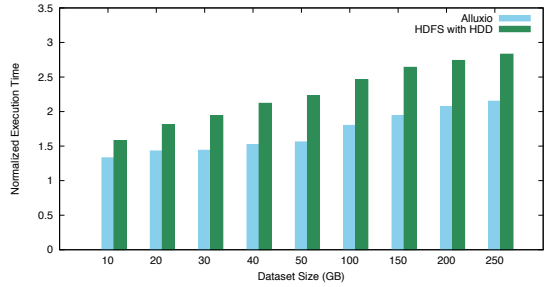
using NVMe or memory. We conclude that Spark RDD persistence using NVMe with compression as an intermediate data buffer can achieve almost the same performance as using memory and, therefore, using remote non-volatile memory has a large potential for supporting in-memory processing data persistency using NVMe over fabrics and/or current generation software-defined infrastructure.

3.3.2 Comparative Study of HDFS and Alluxio

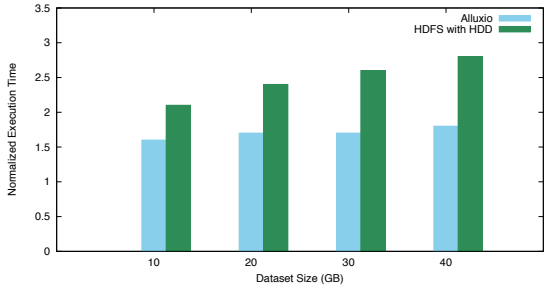
Compared to HDFS, the most important difference when using Alluxio is in the memory utilization approach. As shown in Figures 3.4a, 3.4b and 3.4c, Alluxio is much faster than HDFS for all dataset sizes. Because HDFS is based on a hard disk and Alluxio is



(a) LineCounter



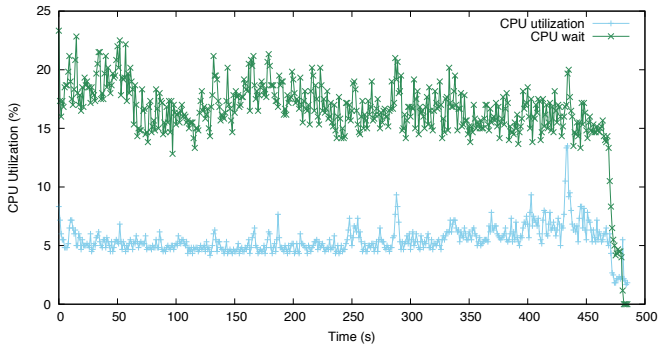
(b) WordCount-reduceByKey



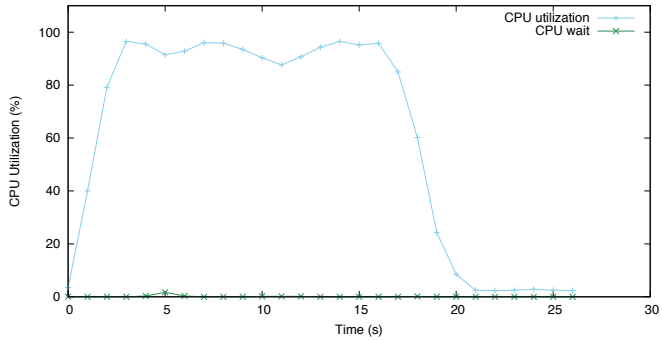
(c) WordCount-groupByKey

Figure 3.4: Execution time of different benchmarks using Alluxio and HDFS (hard disk-based)

based on an in-memory storage layer, these two distributed file systems have different bottlenecks. We also choose LineCounter to show the detail insights. Figures 3.5a and 3.5b show that the I/O throughput is the bottleneck for HDFS while the CPU performance (deserialization speed) is the bottleneck for Alluxio. However, although Alluxio is much faster than HDFS, it is not expected to replace HDFS in the short term because memory is still a very expensive resource in datacenters. Alluxio seems a very promising solution not only as an intermediate layer between HDFS and Spark but also as a means to provide mechanisms for data coupling in data-centric workflows as discussed below. Furthermore, Spark can pre-fetch required data into Alluxio, which can improve the overall performance of Spark applications. This is especially relevant



(a) HDFS



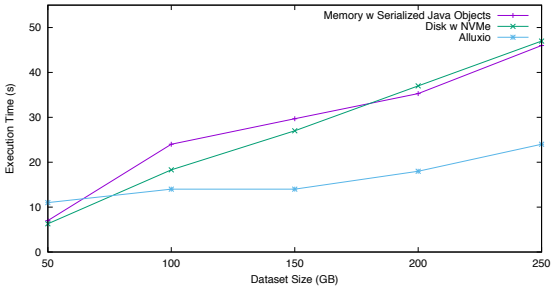
(b) Alluxio

Figure 3.5: CPU Utilization of LineCounter using HDFS and Alluxio with 200GB data sets

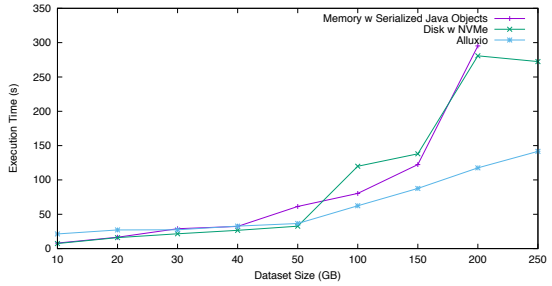
if we expect low-latency access to remote non-volatile memory, which is a current trend in architecture.

3.3.3 Spark RDD Persistence vs. Alluxio

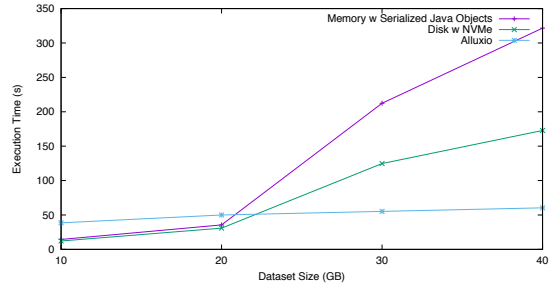
In this section, we evaluate the performance of Alluxio with two different Spark persistence technologies. Because Alluxio stores the serialized dataset in its own system, we store Spark RDDs as serialized Java objects in Alluxio using memory and NVMe. As shown in Figures 3.6a, 3.6b and 3.6c, Spark persistence technology shows better performance than Alluxio using both memory and NVMe with small datasets. However, Alluxio shows better performance with large datasets compared to Spark persistence. This is because Alluxio has better load balancing and high-throughput optimization than Spark persistence technology. As a result, in order to improve the execution of Spark applications, Spark persistence is more appropriate for small datasets and Alluxio



(a) LineCounter



(b) WordCount-reduceByKey



(c) WordCount-groupByKey

Figure 3.6: Execution time of different benchmarks using on Spark RDD Persistence and Alluxio

is more appropriate for large datasets.

We also choose LineCounter here as an example to give detailed insights. As shown in Figure 3.7, the tasks' execution time (i.e., execution time of the different executors) using Spark persistence and Alluxio are almost the same. However, Alluxio is more stable than Spark persistence. Because Spark application execution time is dominated by the longest executor, Alluxio provides better overall performance than Spark persistence. This indicates that new models for data persistency might be needed for optimizing in-memory processing systems using next-generation software-defined infrastructure.

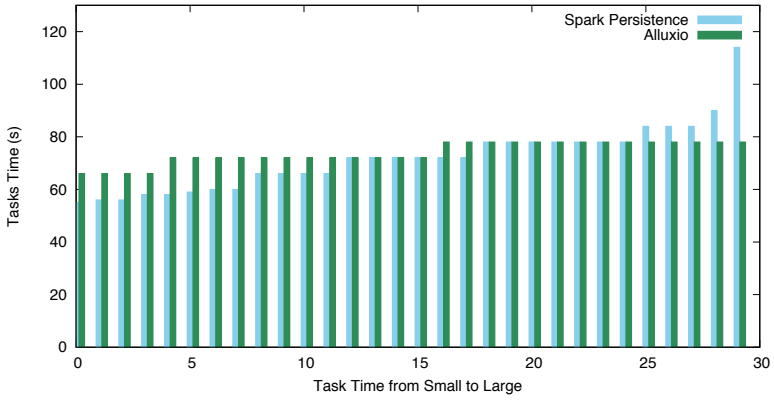


Figure 3.7: Task execution time of different executors using Spark Persistence in memory and Alluxio, 200 GB data sets

3.4 Simulation-Based Evaluation

In order to set a baseline configuration for our software-defined infrastructure model, we divide the Spark model into two parts: (1) a model for estimating Spark's application execution time, and (2) a model for estimating the datacenter interconnect transmission time. Our model also considers key aspects of current software-defined infrastructure such as Intel Rack Scale Design (RSD).

Specifically, in this architecture the storage node is connected to compute nodes using high-performance PCIe, which requires a new data locality strategy in Spark. Based on this architecture, we define data locality using two approaches, rack and off-rack. We assume that our Spark cluster is based on the computing nodes that are on the same rack. Because of the resource pool design of RSD, we assume that Spark can access remote memory and disk while there are insufficient resources in the local Spark cluster.

We build the Spark execution time model based on the experimental evaluation results discussed above, which provides a baseline performance characterization of current software-defined infrastructure[1]. Our model is built upon the performance prediction model by Wang et al. [113]. The execution time of a Spark stage is described as follows:

[1]For reproducibility, the experiments to build the model are provided in GitHub at https://github.com/HelloHorizon/SBAC-PAD-18

$$Time_{Stage} = Time_{start} + \max_{c=1}^{P} \sum_{i=1}^{K_c} Time_{Task} + Time_{cleanup}$$

where P is the total number of processor cores available in the cluster. In Spark, the executors fetch tasks from a task pool once an existing task is completed, and thus different processor cores deal with different numbers of tasks in one stage. K_c is the number of finished tasks for a given core c.

Based on the assumptions described above, a Spark cluster can fetch persisted RDD datasets from a remote storage device through a fast (i.e., low-latency and high-throughput) datacenter interconnect, which means that data can be fetched from remote memory or remote NVMe (e.g., via RDMA). In order to build the Spark execution time model targeting next-generation software-defined infrastructure, we must consider several parameters in our design choices, including network bandwidth, network latency, and the size of the RDD dataset.

Different models have been proposed in the existing literature to estimate the cost of data transmission over networks. For example, Thakur et al. [114] proposed the $\alpha + n\beta$ model to estimate the cost of one message sent between two nodes. In this model, α is the latency of each message, β is the cost of transferring one byte, and n is the number of bytes in the message. Thus, we can drive our model for estimating data transfer cost as follows:

$$Time_{network} = \alpha + \frac{S_{Block} \times N_{tasks}}{B}$$

where α is the latency of each message, S_{Block} is the size of each block in Spark, and B is the available bandwidth between two nodes. Because Spark runs multiple tasks at the same time, N_{task} represents the number of tasks running at the same time.

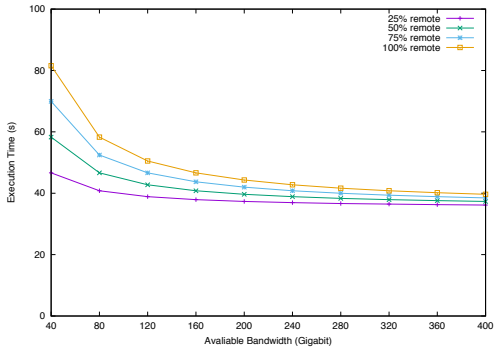
In the simulations, we assume that the Spark cluster does not have enough memory capacity to store the intermediate Spark RDD datasets. We also assume that remote resources in the same datacenter can provide additional memory and NVMe as storage for Spark persistence.

Because different Spark clusters share the datacenter's network bandwidth, we target the execution time of Spark applications for different network design choices in terms of bandwidth. We assume that next-generation software-defined infrastructure will be capable of delivering off-node bandwidth comparable to PCIe-based infrastructure but with a baseline of 400Gbps, which vendors have advertised as being available in the market in the near future [13]. We consider the datasets from section 3.3, i.e., 200GB in serialized format. The input parameters for the simulations can be summarized as follows:

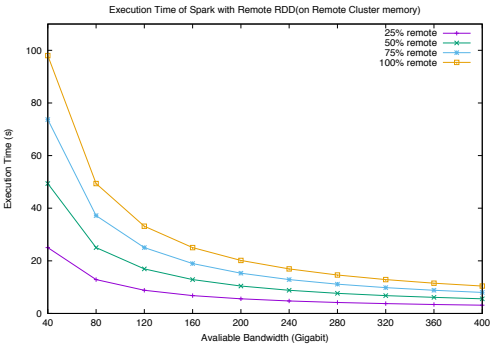
- 20-100% RDD data sets stored in remote resources
- 10-100% network bandwidth available for the cluster
- 400 Gbps network bandwidth with RDMA technology
- RDD data sets persisted in NVMe (serialized format)
- RDD data sets persisted in memory (de-serialized format)

As shown in Figure 3.8a, if a Spark cluster can monopolize the datacenter's bandwidth, then the execution time becomes shorter as available bandwidth increases. Spark applies the columnar compression to decrease I/O pressure on the storage devices. Because the network bandwidth is sufficiently provisioned in the simulated system, we simulate the Spark execution time with RDD datasets in de-serialized format. Figure 3.8a shows that Spark with de-serialized RDD datasets performs better than Spark with serialized RDD datasets when available bandwidth exceeds 280Gbps. Further, when the larger RDD datasets are persisted on remote resources, Spark with de-serialized RDD datasets has worse performance. In summary, Spark performance with de-serialized RDD datasets is better than its performance with serialized RDD datasets when there is sufficient network bandwidth available or when the dataset is small.

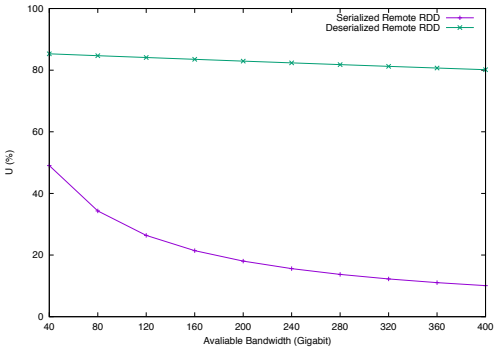
As shown in Figure 3.8c, Spark cannot fully utilize the network bandwidth using the targeted software-defined infrastructure. According to our model and Figure 3.8c, Spark with de-serialized RDDs can decrease the execution time by leveraging current



(a) Execution time, remote RDDs using NVMe



(b) Execution time, remote RDDs using RAM



(c) Network bandwidth utilization

Figure 3.8: Simulation results using RDDs in remote (i.e., off-node) resources

high-bandwidth network fabrics; however, we conclude that the current standard implementation of Spark is not expected to fully exploit future interconnects (e.g., technologies based on silicon photonics).

3.5 Discussion

In this chapter, we analyzed the performance of Spark persistence technology using different storage technologies and compared native Spark persistence with the Alluxio distributed in-memory file system. We observed that NVMe with columnar compression is a good candidate for complementing memory in Spark clusters. We also concluded that Alluxio has better performance than Spark persistence technology for large datasets and found that Alluxio has better load balancing than Spark persistence. Results indicate that software-defined infrastructure can be a viable solution for provisioning bare-metal disaggregated datacenter resources and provide meaningful data points to illuminate the

requirements of in-memory systems to efficiently scale next-generation software-defined infrastructure implementations (e.g., 400G MSA vs. 400/800G embedded optics vs. PCIe 5.0).

While this work represents the foundation or a segment in the most ambitious path towards software-defined in-memory frameworks, the insights obtained from this work are critical for our undergoing work on a caching system for combining Spark and Alluxio more efficiently.

Chapter 4

Elastic Computing in Geo-distributed Data Centers

We found the computing resource utilization of different data centers differs in the time scale and the spatial scale. For example, the computing resource utilization of online service clusters (e.g., a search and online shopping service) can be much lower than off-line service clusters during particular periods, because the system has to reserve computing resources for periods requiring peak computing capabilities (e.g., 200% computing resources compared to normal operations). Another difference is in the time scale, as the busy time of online service clusters is typically daytime, while offline service clusters can be busy at that time (e.g., running Extract, Transform, Load workloads). As the resource utilization of offline service clusters is usually high (70% to 90%), running online services simultaneously may require significant additional resources to avoid resource contention and/or workload performance degradation. With the consideration of network latency and bandwidth as key factors, we explore the use of fast fabric interconnections to overcome this problem.

This chapter focuses on a data warehouse service, which deals with a large volume of data, i.e., hundreds of GBs to tens of TBs per application at a PB-level data warehouse. To better understand the performance of different methods with fast fabric interconnection, we first characterize two different situations:

- Performance of data warehouse applications with a storage cluster (HDFS) within the same data center.
- Performance of data warehouse applications with a remote storage cluster (HDFS).

Based on the findings from our empirical performance evaluation, we investigate methods for fully utilizing the computing resources between two geo-distributed data

centers with fast fabric interconnection and the abstraction of an elastic computing cluster. The results of the experimental evaluation provide support for harvesting spare computing resources across geo-distributed data centers as this approach can accelerate large-scale big data services, such as the evaluated data warehouse service

4.1 Resource Utilization of Computing Clusters

We first discuss the main challenges faced in this work with an analysis of the computing resource utilization of a real production enterprise computing cluster. Then we evaluate the two proposed types of deployment to estimate the performance and cost of network transmission between geo-distributed data centers.

We leverage data collection with an internal monitor platform, big data Platform Eye (BDPEYE), which is used in production services. With BDPEYE, users can collect different types of metrics from different components (hardware metrics, application metrics, cluster metrics, etc.). We select metrics from schedulers (YARN and Kubernetes) to show the computing resource utilization of different data centers.

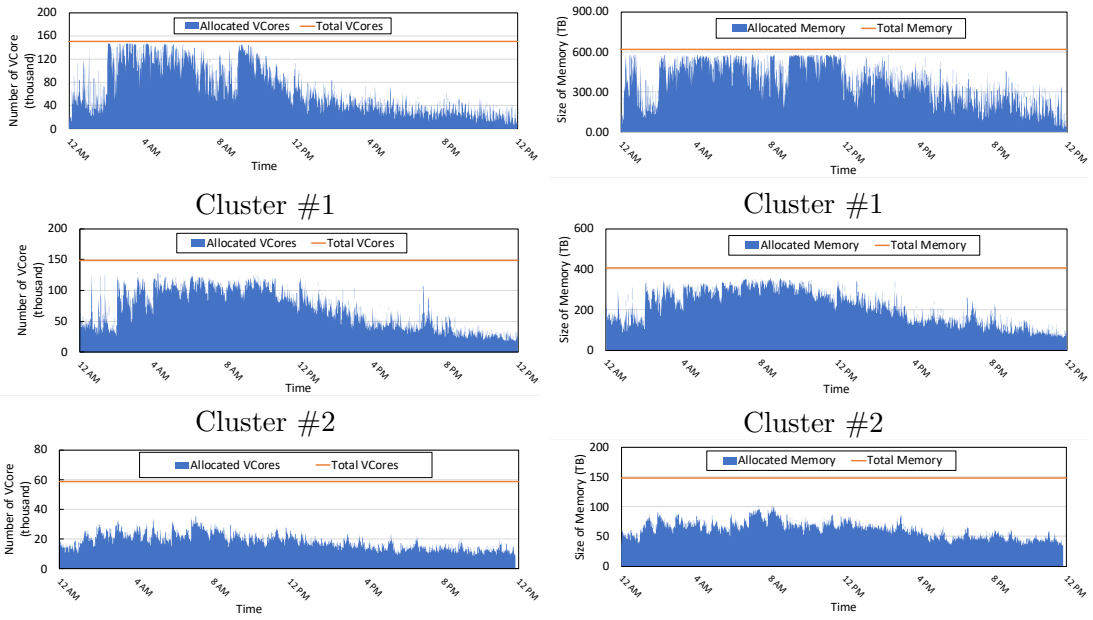


Figure 4.1: Vcores and memory utilization of three different computing clusters in geo-distributed data centers

BDPEYE collects monitoring information every 30 sec. As part of this work, we collected one month (August 2018) of monitoring information from two different production computing clusters. To show the available computing resource for Spark, we leverage CPU and memory utilization from clusters in Figure 4.1. Based on the computing resource utilization of the computing clusters, we classify the potential states of the computing clusters into three categories: overloaded (applications waiting in the queue), healthy (no applications waiting in the queue), and free (the cluster can offer computing resources to other clusters).

Resource limitations in a single data center. With the limited space and power supply in the same geo-location, it is difficult to expand the data center and provide the abstraction of unlimited servers. Because of this reality, in many cases, large production services cannot achieve good performance with limited computing resources. In practice, large Spark applications usually cannot get enough executors (consisting of CPU and memory) on time, which causes a significant performance loss in the production environment. As we can see from the analysis above, we have to find an efficient way to solve these performance challenges with existing computing resources.

4.2 Harvesting Spare Computing Resources in Geo-distributed Data Centers

Harvesting spare computing resources within the same data center has been proposed in previous work [74, 75]. As opposed to existing work, and based on the observation from the analysis above, we efficiently utilize computing resources across geo-distributed data centers. Existing literature [69, 71, 72, 73] has also explored the potential and methods for utilizing geo-distributed data centers with limited network bandwidth. However, these methods cannot meet the performance requirement for large enterprise data-centric services, which can consist of hundreds or even thousands of applications. This challenge is explained in detail and analyzed in section 4.5.

To illustrate the proposed method in building a disaggregated computing resource

Figure 4.2: Geographic locations of the four data centers and the network connection between the data centers

pool between geo-distributed data centers, we use a case scenario described in Figure 4.2. The figure shows four data centers distributed in four different regions: Beijing Daxing district (DC #1), Beijing Haidian district (DC #2), Beijing Tongzhou district (DC #3), and Langfang city (DC #4). Please note that the figure is a sketch and does not provide the exact location of the data centers.

Table 4.1: Network bandwidth (Gbps)/straight-line distance (KM) between the four geo-distributed data centers

	DC #1	DC #2	DC #3	DC #4
DC #1	- / -	- / 38	- / 33	760 / 37
DC #2	- / 38	- / -	- / 43	760 / 70
DC #3	- / 33	- / 43	- / -	960 / 38
DC #4	760 / 37	760 / 70	960 / 38	- / -

Table 4.1 shows the network bandwidth and straight-line distance between the four data centers. The real network transmission distance is longer than the straight-line distance. Considering the availability of the high bandwidth network, we formulate two design choices of big data analytics with geo-distributed data centers. As shown

in the table (the capabilities delivered by the data centers with fast and high-capacity fabric interconnections), big data analytics has the potential to benefit from additional remote spare computing resources.

4.3 Spark-based Services on Cloud Resources

Currently, cloud computing service providers such as AWS (Amazon Web Services), Google Cloud and Tencent Cloud provide various big data analytics services, including data processing, ETL (extract, transform, load) workflows, and machine learning. Cloud computing service providers usually provide elastic computing resources with a core storage cluster. Furthermore, some of them also provide auto-scaling services for dynamically increasing storage capabilities (i.e., avoiding data loss). For example, AWS, Google Cloud and Tencent Cloud [115, 116, 117] provide elastic computing clusters for Spark services and Google Cloud provides auto-scaling services for the storage layer (HDFS).

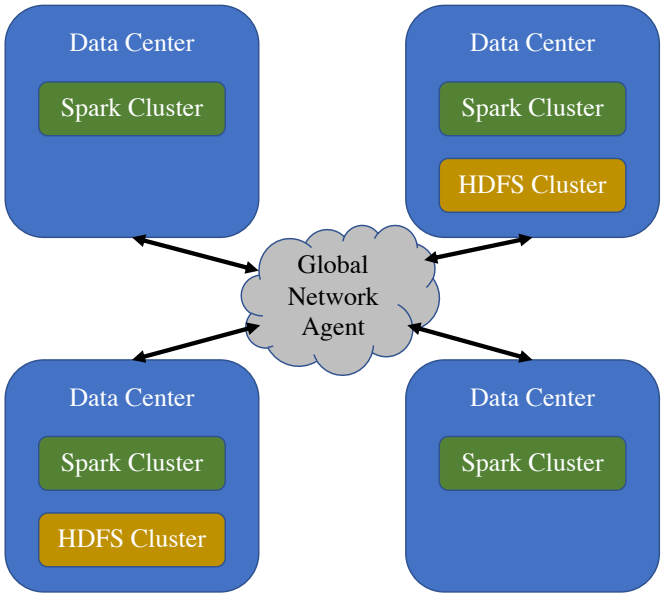


Figure 4.3: Architecture of geo-distributed data centers, which is a common architecture in cloud environments

In the use case scenario discussed in this work we have the same Spark service within several different geo-distributed data centers, as illustrated in Figure 4.3.

4.4 Modeling the Computing Cost Across Data Centers

Because the elastic computing cluster is built with geo-distributed data centers, we estimate the cost of network transmission across data centers. Based on our formulation, we can see that it is more realistic for us to use separate computing clusters than one computing cluster. Existing work [70, 118, 119, 120] proposed network models to calculate the network transmission for big data analytics (e.g., MapReduce Hadoop, Spark, Flink). We adapt the model to fit the targeted elastic cluster model to run with big data analytics.

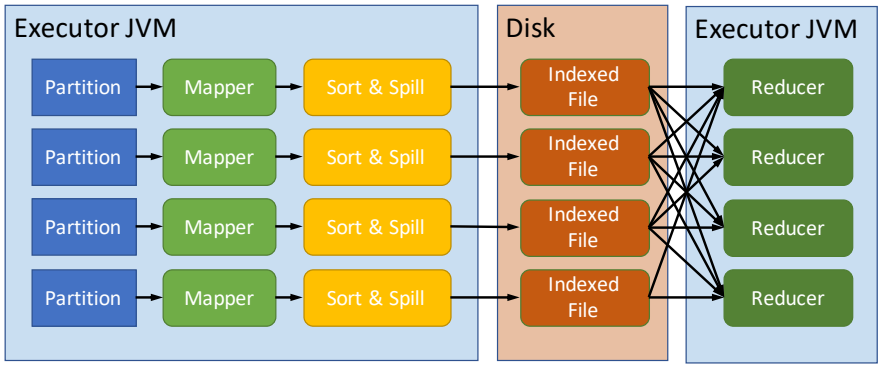


Figure 4.4: Spark data flow

To formulate the cost of network transmission, we use Spark [88] with HDFS [9] as a driving use case. The main costs of data transmission over the network in Spark are read data from HDFS, shuffle, and write the result back to HDFS. Figure 4.4 shows the basic data flow for Spark. Spark can implement efficient data locality mechanisms when Spark executors read data blocks from a local node or in a small computing cluster; however, it is harder for Spark to read data blocks from a local node in a large-scale environment. In contrast to the data locality of input data and output data, Spark has good data locality within JVM (memory persistence) and a local disk (disk persistence) for intermediate data. To simplify the formulation, we assume all input and output data is transmitted through the network while intermediate data is not. Thus, the cost of the (network) data transmission ($\mathcal{N}$) can be abstracted as follows:

$$\mathcal{N} = \mathcal{N}_{\text{input}} + \mathcal{N}_{\text{shuffle}} + \mathcal{N}_{\text{output}} \tag{4.1}$$

Spark assigns a set of operations, which process part of the data within a single process individually. Thus, the Spark performance model can be abstracted as a MapReduce problem. In a traditional MapReduce problem, the performance pattern can be abstracted as a directed acyclic graph $\mathcal{G} \in \{\mathcal{V}, \mathcal{E}\}$.

- Stage is the basic process step in Spark, which is divided by the shuffle operation. $\mathcal{V}$ is a set of stages.
- $\mathcal{E}$ is a set of edges, which represent the execution time of the stages.

Before we formulate the cost of each edge $\mathcal{E}$, we need to elaborate on the shuffling process in Spark. We consider the sort-based shuffle in Spark, which is used in the production environment. Spark shuffle can be classified into shuffle write and shuffle read. Shuffle write happens in the "map" phase, while shuffle read happens in the "reduce" phase. Different from MapReduce in Hadoop, Spark does not offer network overlapping transmission for shuffle data. The "reducers" read shuffle data after all "mappers" are finalized in Spark. We can formulate the performance of a single task in a "reducer" as $\mathcal{T}_{\text{task}} = \mathcal{T}_{\text{execution}} + \mathcal{T}_{\text{disk}} + \mathcal{T}_{\text{net}}$ and in "mapper" as $\mathcal{T}_{\text{task}} = \mathcal{T}_{\text{execution}} + \mathcal{T}_{\text{disk}}$).

- $\mathcal{T}_{\text{execution}}$ is the task computing time.
- $\mathcal{T}_{\text{disk}}$ is task disk I/O time, which includes the disk persistence time, shuffle write time in the "mapper" phase, and shuffle read time in the "reducer" phase. We can formulate the disk I/O time for Spark as $\mathcal{T}_{\text{disk}} = \frac{\mathcal{S}_{\text{disk}}}{\mathcal{B}_{\text{disk}}}$. $\mathcal{S}_{\text{disk}}$ is the shuffle data size for a single task, and $\mathcal{B}_{\text{disk}}$ is the disk bandwidth.
- $\mathcal{T}_{\text{net}}$ is the task network transmission time. Furthermore, the main network communication cost are shuffle and fetching input data, and writing output data. To further formulate the network transmission time of shuffle, we use the $\alpha - \beta$ model [114]. Although there are more models, $\alpha - \beta$ model is suitable for the problem that we address. Thus, we can formulate the network transmission time $\mathcal{T}_{\text{net}}$ as $\mathcal{T}_{\text{net}} = \alpha \frac{\mathcal{S}_{\text{net}}}{n} + \frac{\mathcal{S}_{\text{net}}}{\beta}$. α is the latency for sending a message, β is the network bandwidth between two nodes, $\mathcal{S}_{\text{net}}$ is the size of shuffle read, and n is the size of each message.

Thus, we can formulate the execution time ($\mathcal{T}_{\text{application}}$) for Spark as follows:

$$\begin{aligned}\mathcal{T}_{\text{application}} &= \sum_{i=0}^{n} \mathcal{T}_{\text{stage}} = \sum_{i=0}^{n} max(\mathcal{T}_{\text{task}}) \\ &= \sum_{i=0}^{n} max(\mathcal{T}_{\text{execution}} + \mathcal{T}_{\text{disk}} + \mathcal{T}_{\text{net}}) \\ &= \sum_{i=0}^{n} max(\mathcal{T}_{\text{execution}} + \frac{\mathcal{S}_{\text{disk}}}{\mathcal{B}_{\text{disk}}} + \alpha\frac{\mathcal{S}_{\text{net}}}{n} + \frac{\mathcal{S}_{\text{net}}}{\beta})\end{aligned} \tag{4.2}$$

To calculate the cost of network transmission intra-data centers and across data centers, we adopt the equations (4.1) and (4.2) in the targeted geo-distributed computing cluster model. The total network transmission ($\mathcal{N}$) can be divided into intra-data center transmissions ($\mathcal{N}_{\mathsf{intra}}$) and across-data centers transmissions ($\mathcal{N}_{\mathsf{across}}$). Currently, we use Kubernetes [74, 121] as the primary scheduler for Spark. In this work, we address two types of deployment to implement the elastic computing cluster for Spark, as shown in Figure 4.3.

We use the driving example to illustrate the cost of network data transmission between geo-distributed data centers for a single Spark application with a single computing pool and separate computing pools. Spark gets the available executors (the basic computing unit of Spark) from all four geo-distributed data centers when using a single computing pool. We assume Spark receives the same number of executors from n data centers in this scenario. Conversely, Spark receives all input data from the local data center or a single remote data center for separate computing pools. Thus, we can calculate the cost of network data transmission across data centers for a single computing pool as follows:

$$\mathcal{N}_{\mathsf{across}} = \frac{\text{n-1}}{\text{n}}(\mathcal{N}_{\text{input}} + \mathcal{N}_{\text{output}} + \frac{\text{n-1}}{\text{n}}\mathcal{N}_{\text{shuffle}}) \tag{4.3}$$

We calculate the cost of network data transmission across data centers for multiple separate (individual) computing pools as follows:

$$\mathcal{N}_{\mathsf{across}} = \frac{\text{n-1}}{\text{n}}(\mathcal{N}_{\text{input}} + \mathcal{N}_{\text{output}}) \tag{4.4}$$

For example, warehouse application #6 is a Spark SQL job with 2,666GB input data, 3,714 GB shuffle data, and 16 GB output data. We assume that this warehouse application runs on a 300-node computing cluster. The volume of network data transmission across the data centers is $\frac{299}{300} \times \frac{3}{4} \times 3,714 + \frac{3}{4} \times (2,666 + 16) = 4,789GB$ for single

computing pools, and $\frac{299}{300} \times 2,666 = 2,657GB$ for multiple individual computing pools. The latency between the data centers is longer than the latency within a data center, and the shuffle in Spark could be small all-to-all data communication. The IOPS for shuffle is usually very large [101], which results in high overhead for Spark shuffle data transmission. This calculation does not consider the network transmission, because of the delay scheduler in Spark [122], and metadata transmission with the master node. Note that when this is taken into account, the network transmission could be even higher for a single computing cluster environment.

We adopt the performance equation (4.2) for geo-distributed data centers. Because of the hardware performance gap between different data centers, Spark can have a significant performance gap or imbalance for different tasks in the same stage. Further, the data skew phenomena are typically found in large-scale enterprise computing clusters with real production data [123]. Scheduling a heavy task in Spark in a low-performance node could cause a significant execution imbalance between different tasks in the same stage.

Based on the analysis above, we can characterize the features of geo-distributed data centers as follows:

- The hardware performance in different data centers can have significant differences. For example, because a new data center may use a newer and better-performing CPU and memory with larger network bandwidth, the performance of Spark in the new data center (DC #4) can save in practice 30% to 50% execution time compared to the old data center (DC #3), depending on the application.

- The data source of the applications is usually in a single data center. Because the volume of the existing data source is huge, it is very challenging to make a copy in each data center.

- The total network bandwidth between data centers is smaller than the total bandwidth within the data center. With fast fabric interconnection between data centers, we can afford part of the network transmission in Spark across geo-distributed data centers.

A Spark application running in several data centers at scale can suffer the long tail problem because of hardware heterogeneity. As we can see from equation (4.2), the execution time $\mathcal{T}_{\text{execution}}$ strongly depends on the performance of the CPU. However, the variability in the CPU performance of servers in different data centers can be significant. Thus, the long tail problem can lead to a large waste of computer resources. As a result, we try to schedule the application within the same geo-location rather than across geo-distributed data centers, when possible.

4.5 Experimental Methodology

In this section, we characterize the core services of the data warehouse application running in the production environment. Then we use the data warehouse application workloads to evaluate the performance of the system, locally and with a configuration that uses remote resources.

4.5.1 Data Warehouse Applications Characterization

Data warehouse applications provide data services to various businesses, which include user information, order information, shipping information, and storage information, among others. In the time scale, the core service of the data warehouse processes all historical data and provides the data for second-day services. In large-scale enterprises such as JD.com, hundreds of PBs of data are stored in an HDFS cluster, and the data warehouse service has to process about a PB of data every day. For large-scale warehouse applications, tens and even hundreds of data requests are fetched from the distributed file system, and tens of TBs of shuffle data are generated during runtime. Thus, the computing cluster has to provide a large volume of computing resources to allow Spark to provide a reasonable performance.

The services in the enterprise generate an incredibly large volume of data every day. Therefore, the data is stored in a high-compression format to save storage space. Furthermore, there are thousands of input tables, and some of the applications are dependent, so the whole warehouse chain can be divided into seven layers. Figure

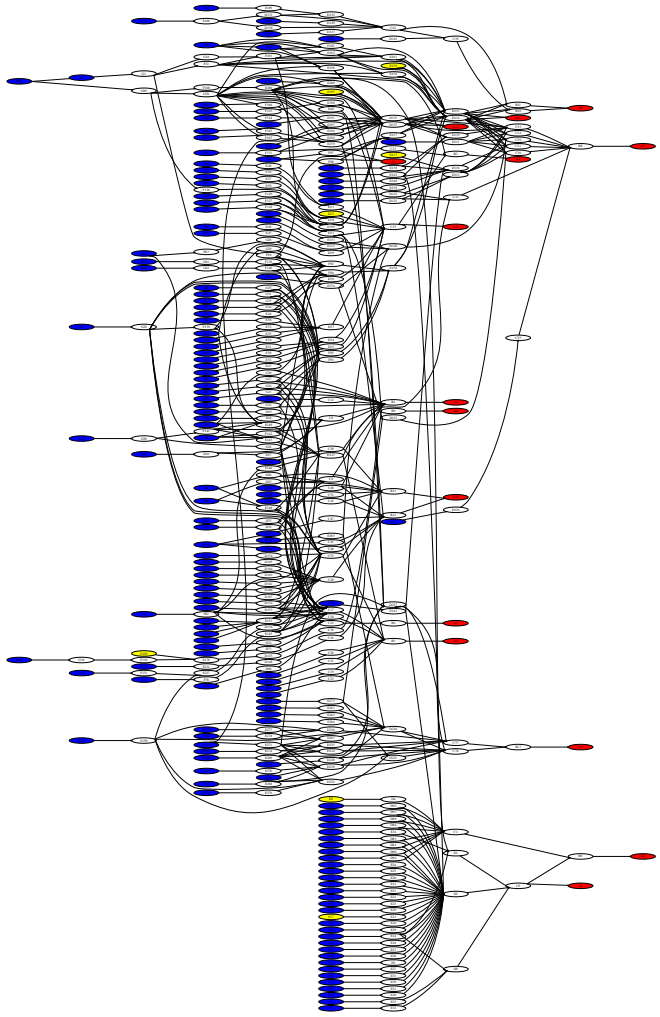


Figure 4.5: Dependency relationship between the services of the data warehouse use case

4.5 shows the dependency relationship of the warehouse applications. This dependency graph has 435 warehouse applications (143 core data warehouse applications), including basic data movement and a data checker, among others. The figure shows the relationship between all warehouse applications, including data processing, data movement, data scanning, etc.

In the experimental evaluation, we selected the core services of the data warehouse as the target workloads. The core services of the data warehouse involve a large number of applications, and the data warehouse applications involve a large volume of data that varies in size. Figure 4.6 shows the input, output, and shuffle(r/w) size of 20 (out of

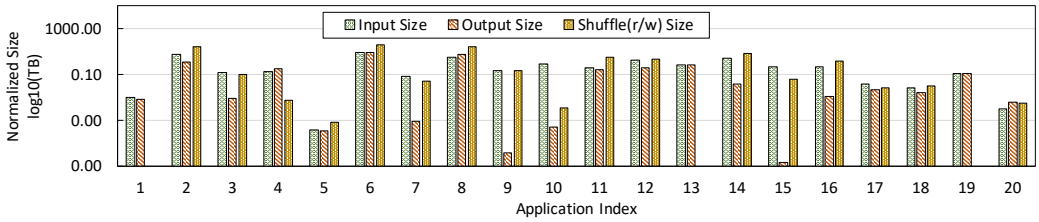


Figure 4.6: Input, output and shuffle size of core warehouse applications (20 out of 143)

143) data warehouse applications from the enterprise production environment.

We observe that the core data service of the data warehouse has the following requirements and characteristics:

- High-performance requirements. Because the data service provides data support for other services, the service must be finished as soon as possible.
- Large volume of data with a high-compression format. With the incredible increasing volume of data, high-compression formats can help distributed file systems provide sufficient storage space.
- Large number of applications with several dependent layers. The data warehouse application has pre-requested data warehouse applications in the data pipeline. Thus, the data warehouse application in the next layer cannot start before all applications from the previous layer have finished.
- The input data varies in size. The input data size of the warehouse applications can be significantly diverse, which can cause a big imbalance in the required computing resources.

The targeted core service of the data warehouse runs on a computing cluster with about 3,700 compute nodes and includes 143 Spark SQL applications. However, it is hard for each data warehouse application to obtain sufficient computing resources on time, especially applications that require a large volume of data (i.e., require more computing resources to deliver adequate performance). In practice, although we use 1,000 executors, sometimes, the application may receive only about 400 executors from the computing cluster.

The executor is the minimum computing resource unit, composed of virtual cores and memory, in Spark. Currently, each unit has the same number of virtual cores and memory; therefore, we can calculate the number of available executors for a particular application as follows:

$$Executor_{available} = Min(\frac{Memory_{available}}{Memory_{executor}}, \frac{VCore_{available}}{VCore_{executor}})$$

4.5.2 Testbed

Our testbed is composed of 294 nodes in a production computing cluster. The servers have two Intel(R) Xeon(R) CPU E5-2640 v2 @ 2.00GHz processors, and the configuration used in this work includes 256 GB DRAM, and 40 GbE network connectivity between each server. We configure the big data framework, container management system, local distributed file system as the baseline with the same configuration as with the remote distributed file system. Spark version 2.3 was deployed using Kubernetes version 1.10, CentOS version 7.5 and HDFS version 2.7. We reserve one core (3.1%) and 2 GB (0.8%) memory for Kubernetes, and three cores (9.4%) and 9 GB (3.5%) memory for the operating system.

4.5.3 Workloads

As we discussed above, our approach proposes building an elastic computing pool with individual computing clusters. However, in the geo-distributed environment, the location of data storage could be different than the computing resources. We decide to place the Spark application in different data centers without considering data location based on two reasons: (1) it is hard for engineering to predict the available computing resource in data centers, and (2) most of the existing data is located on the single data center. Thus, it is important for us to find a way to use spare geo-distributed computing resource efficiently. Based on this condition, we evaluate the performance of Spark with remote HDFS and local HDFS. Finally, we use two deployed HDFS systems on data centers #3 and #4, individually.

To evaluate the performance gap between the local data source and remote data

source, we select six (typical) warehouse applications from the core service of the data warehouse as shown in table 4.2. We use the input data from production HDFS as standard data. Further, table 4.2 shows the input and output size of warehouse applications. Since we need to meet SLA (Service Level Agreement) requirements, we use the most performant configuration for warehouse applications. Thus, we allocate enough memory resource to every executor. The number of cores in each executor is based on the best practices in production environments, which can fully use the I/O throughput for both JVM and disk in the same executor. Furthermore, to evaluate the different type of warehouse applications, we choose different types of applications from the core service of data warehouse, including:

1. Compute-intensive application: CPU bound application.
2. Shuffle intensive application: Spark write and read shuffle data into local disk [124], thus shuffle intensive application is network and disk I/O intensive application.
3. I/O intensive application: disk I/O bound application.

Table 4.2: Characterization and configuration of warehouse applications

App ID	Input Size	Output Size	Shuffle Size	#Executors	Mem/Cores
#1	795 GB	2.6 GB	1.00 GB	1000	20 GB/5
#2	165 GB	309 GB	110 GB	1000	20 GB/5
#3	1843 GB	421 GB	2031 GB	1250	24 GB/5
#4	2.85 GB	2.92 GB	2.44 GB	1	20 GB/5
#5	15.1 GB	4.6 GB	7.4 GB	300	20 GB/5
#6	2666 GB	16 GB	3714 GB	1000	20 GB/5

4.6 Experimental Results

We use Spark with local HDFS as a performance baseline. Then we evaluate the performance of Spark with remote HDFS, which has a separate storage cluster and computing cluster. For each test, we run it five times and use the average execution time as a reference for the performance of the warehouse applications using Spark with local HDFS and Spark with remote HDFS.

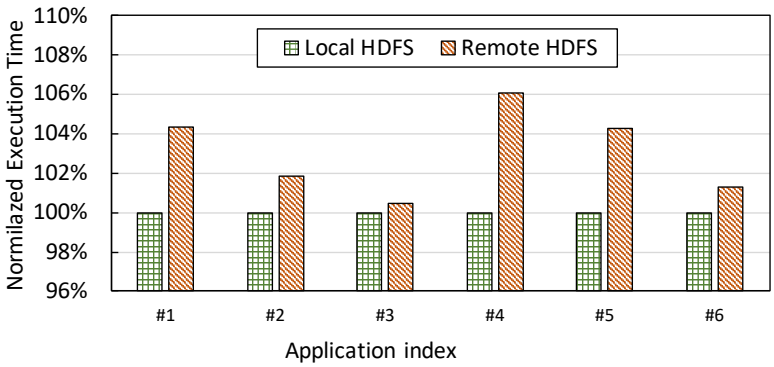


Figure 4.7: Normalized execution time of Spark with local HDFS and remote HDFS

Figure 4.7 shows the normalized execution time of the warehouse applications with respect to the execution time of Spark with local HDFS. We can observe that the execution time of Spark with the local data source and with the remote data source is almost the same.

In a real production environment, to ensure that the data warehouse core services' execution time is within the SLA constraints, all requests from other queues may need to be suspended until the core service has finished. The main reason is that the computing resources in the same geo-location are not enough for such a large service. Thus, we want to harvest the spare computing resource from other data centers. Furthermore, existing research has explored pre-data movement methods based on computing resource prediction. However, because of the growth of and change in existing workloads, the pre-data movement is challenging to implement in practice. It is also hard to transmit the required data to remote computing clusters based on the computing resource availability before runtime. Thus, transmitting runtime data across data centers is an efficient way to reduce the computing resource utilization burden in a single computing cluster.

Since we cannot experiment with the performance and resource utilization of the core service of a data warehouse with four data centers, we implement a simulator to reproduce the performance and resource utilization of this system across four geo-distributed data centers. We simulate the execution time for the core services of the data warehouse, and the CPU and memory utilization for four geo-distributed data

Algorithm 1: Task scheduler policy

1 Function Storage Device Priority ;
Input : APP_{pool}, App_{index}, $App_{prerequest}$, App_{cores}, App_{memory}, App_{time}, $Cluster_{index}$, $Cluster_{index\_cores}$, $Clusters_{index\_memory}$
Output: $T_{execution}$

```
start time;
while APP_pool is not empty do
    for i = 1; i <= App_index; i++ do
        if App_prerequest == true then
            if App_cores >= Cluster_1_cores & App_cores >= Cluster_1_memory then
                Assign Application into cluster 1;
            else if FindMax(Cluster_2-4) then
                if App_cores >= Cluster_index_cores & App_memory >= Cluster_index_memory then
                    Assign Application into cluster_index;
                end if
            else
                No cluster available;
            end if
        end if
    end for
end while
end time;
return (T_execution);
```

centers. In this simulation, based on the existing production environments, we assume that we have one storage cluster in data center 1, and an elastic computing cluster across four different geo-distributed data centers. The performance of a single node in the computing clusters of the various data centers differs. However, to simplify the model, we assume that different computing clusters in different data centers have similar performances.

To simulate the overall performance of the elastic computing cluster with the warehouse applications, we use the workloads of the core warehouse applications as shown in Figure 4.7. We profile the core utilization, memory utilization, and running time of 143 core warehouse applications in a production environment. The simulation scheduler policy is described in Algorithm 1.

We mark the co-located computing cluster, which has the same geo-location as a storage cluster, as the highest priority in the scheduling algorithm. It tries to assign applications to cluster 1. Once computing cluster 1 cannot offer enough computing resources to the application, the scheduler assigns remote computing clusters to the

Table 4.3: Execution time of core service of data warehouse

Scenario	Computing cluster	execution time (s)
#1	#1	8,408
#2	#1,#2	5,908
#3	#1,#2,#3	5,272
#4	#1,#2,#3,#4	5,272

application following a round-robin policy. The scheduler then checks the available computing resources in each remote computing cluster before assigning one. Next, the scheduler places the application in the computer cluster that has more available computing resources in different data centers. Further, we divide the core service of the data warehouse into seven layers based on the dependency relationship of their services. Thus, the scheduler cannot start a data warehouse application without finishing the prerequisite data warehouse applications first. Furthermore, we assume that each computing cluster has 10,000 CPU cores with 53 TB of memory.

Table 4.3 shows the execution time of the data warehouse core service within a single data center, and with harvesting spare computing resources across geo-distributed data centers, based on the approach described above.

4.7 Discussion

In this chapter, we characterized the computing resource utilization of four different geo-distributed computing clusters. Then we introduced the use of fast fabric interconnections across the geo-distributed data centers. We explored the potential deployment with these data centers. Then, we evaluated the performance of the system based on Spark, HDFS, and Kubenetes in a production enterprise environment. Based on the results, we explored the potential of using fast fabric interconnection to harvest spare computing resources across geo-distributed data centers. We built a simulation based on a data warehouse core service, and then we verified that we could build an elastic computing cluster across geo-distributed data centers, which can speed up large data warehouse enterprise services.

Chapter 5

Resource Management for In-memory Big Data Analytics

This chapter first characterizes the computing resource efficiency of a real production enterprise data center. It addresses the uneven utilization of memory and CPU, and then we address the bottleneck of the current in-memory big data framework within an existing data center. It also characterizes the remote read and write performances of large-volume DIMM-based persistent memory (PMEM) and DRAM, enabling us to explore the potential of PMEM as an affordable replacement for DRAM, used in a disaggregated memory pool. Next, it presents the design and implementation of the proposed in-memory big data processing system using disaggregated memory with Spark and an in-memory distributed file system called Distributed Memory Objects (DMO).

We present the design and implementation of an external shuffle service with DMO, leading to a savings of up to 72% in execution time with shuffle-intensive Spark applications having the same memory consumption. Furthermore, we present the design and implementation of extended external storage for Spark data shuffling and persistence with DMO and large-volume PMEM. We empirically evaluate our system's performance with real production workloads, which demonstrates that the system can increase memory capacity at affordable cost and with low overhead compared to using DRAM exclusively. Finally, we discuss the impacts of our approach on real production data processing systems at scale.

5.1 Analysis of the Efficiency of Data Center Computing Resources

Before discussing the details of our proposed approach, in this section, we provide a detailed utilization analysis of computing resources in a real production data center

with 3,700 servers from a large e-commerce company. This characterization will enable us to understand the bottlenecks of current production computing clusters. We first analyze the utilization of computing resources (both CPU and memory) during one month (August 2018), and then we analyze the performance pattern of real production applications.

We collected profiling data through an internal cluster monitoring tool. The data includes multiple system metrics from the data center's computing cluster running big data analytics using Spark.

As the focus of this work is on computing resource management, we use the allocation information of the YARN scheduler collected through the monitoring tool, to analyze the utilization of the CPU and memory, which are the most important computing resources in the cluster.

In order to illustrate the computing resource utilization in a data processing cluster, we define cluster memory overhead $\text{MEM}_{\text{Overhead}}$ as:

$$\text{MEM}_{\text{overhead}} = \text{CPU}_{\text{util}} - \text{MEM}_{\text{util}}$$

where CPU utilization CPU_{util} is the ratio between the number of allocated CPU cores and the number of allocable CPU in the cluster, and memory utilization MEM_{util} is the ratio between the amount of allocated memory and the amount of total allocable memory in the cluster.

Figure 5.1 shows the memory overhead of the computing cluster during the data collection period. The average memory utilization is mostly above that of the CPU utilization for the data center. Therefore, the big data processing frameworks that run on the computing cluster cannot fully utilize the available CPU cores because of limited

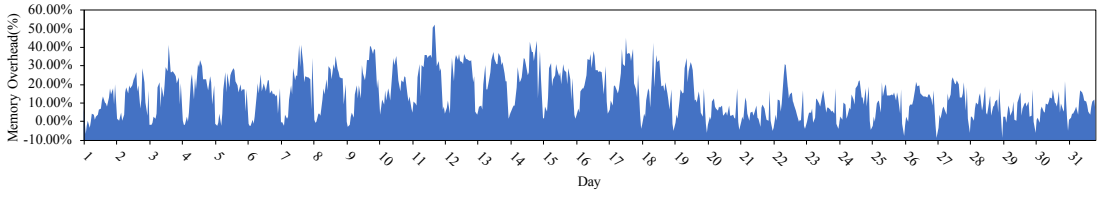


Figure 5.1: Memory overhead of a production computing cluster with 3,700 servers

memory resources. In the following section, we discuss the current memory architecture and memory management of Spark, and explain the high memory utilization of the computing cluster.

5.2 Spark Memory Management and Shuffle Service

Spark, which is one of the most powerful big data frameworks [88], uses memory to speed up large-scale data processing. In this section, we describe the latest memory management and shuffle service used in Spark (version 2.3) and provide cost analysis based on its current memory management and architecture.

5.2.1 Spark Memory Management Optimization

In Spark, data is abstracted as resilient distributed datasets (RDD) [125], which represents a collection of objects partitioned across a set of compute nodes. RDDs include the lineage information, which can help Spark applications recompute the RDDs to ensure data reliability upon task failures. Spark tries to keep all RDDs in memory to ensure fast access to the data and uses the disk when memory space is insufficient.

Spark divides memory into two main regions, execution and storage memory. The execution memory is mainly used for storing the objects required during the execution of Spark tasks. The intermediate data of shuffle is stored in execution memory. Execution memory will not be evicted for storage memory purposes. The storage memory is a region of memory used for caching and for intermediate serialized data. We are facing two key challenges regarding the optimization of the performance and computing resource utilization in Spark:

- Tuning Challenge: Although computing resource tuning can help adjusting the amount of memory to the Spark application, it is hard to set the optimal configuration for every application because (i) enterprise data centers can run tens of thousands of applications in the computing cluster every day; and (ii) the size of input data vary every day.

- Uneven utilization of computing resources: Current cluster is not typically designed for running in-memory big data frameworks and, as mentioned in previous sections, the utilization of the CPU is lower than the utilization of the memory. As a result, there is significant under-utilized CPU resource in the computer cluster while memory may not be sufficient to persist data in memory.

Spark has several optimization strategies, such as dynamic memory tuning at run time, which is called "Dynamic Resource Allocation". In this strategy, Spark applications request computing resources back to the computing cluster if they are no longer being used, and they request computing resources again when needed [126]. However, we found that dynamic resource allocation cannot solve this problem in a production system for two reasons.

On the one hand, Dynamic Resource Allocation can only kill or start an executor to release currently unused resources early but cannot address the waste of processing resources due to an imbalance between CPU and memory utilization. Figure 5.2 shows the memory and CPU utilization of the production computing cluster from 8:30 AM to 12:30 PM during a workday. The memory utilization of the entire cluster can remain close to 100% for a long period, while utilization of the CPU ranges from 30% to 70%, representing a large waste of CPU resource. On the other hand, once a resource is freed in production environments, the scheduler may take a long time to get enough executors for the same Spark application because of computing resource racing. The memory utilization of the entire computing cluster reached 100%, so it cannot offer enough executor memory to the Spark applications in the following stages on time.

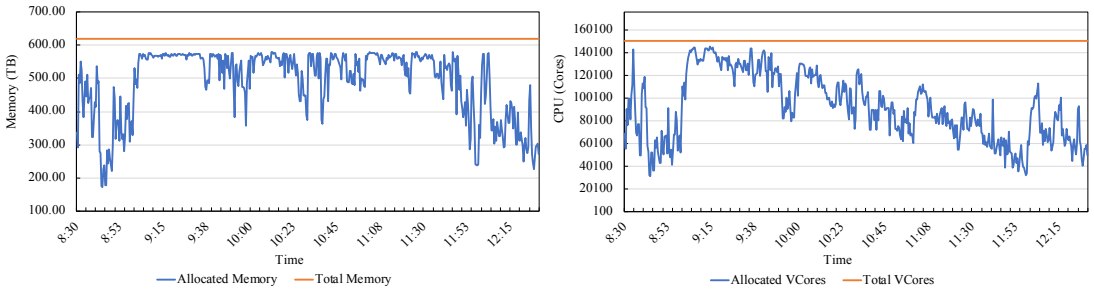


Figure 5.2: Computing resources utilization of a production cluster with 3,700 servers from 8:30 AM to 12:30 PM

5.2.2 Spark Shuffle Management

The shuffle operation can re-distribute data to certain tasks [127], and it is involved in a large number of Spark operations. Shuffling is one of the most expensive operations in Spark, as it involves disk I/O, data serialization, and network I/O. We divide the main I/O costs of Spark shuffle into two parts:

1. Data spill cost: The output data from an individual mapper is kept in the memory until there is insufficient memory in JVM. Spark will spill data onto disks once the execution memory is insufficient. Moreover, Spark sorts the shuffling results based on the target partitions. Thus, with sort-based shuffling, spill data can significantly decrease the performance of Spark.

2. Shuffle read/write cost: Spark task writes/reads shuffle data to/from disks, which can introduce high I/O cost

5.2.3 Spark Memory Requirements

A Spark application is executed in stages. Specifically, it considers all operations before the shuffle phase to be one stage and all operations after to be another. If an operation does not require shuffling, it is considered one stage. Spark can suffer significant performance loss if it does not have sufficient memory to be allocated to every executor. Furthermore, in production environments, it is possible to have large variability in the input data's size for different tasks at the same stage. As a result, it is hard to balance the trade-off between performance and memory usage efficiency.

Figure 5.3 shows the data distribution of one of our typical production Spark workloads at different stages. The input data size is organized into six layers, from 0 MB to 1 GB. The figure shows that the input data size of every task differs, while the allocated computing resources are the same in every executor. Thus, the allocated memory for every executor will be dominated by the largest partition in its tasks, which leads to memory under-utilization during data processing. An effective way to address this imbalanced memory requirement issue is to have a large shared memory pool to satisfy different executor's memory requirement.

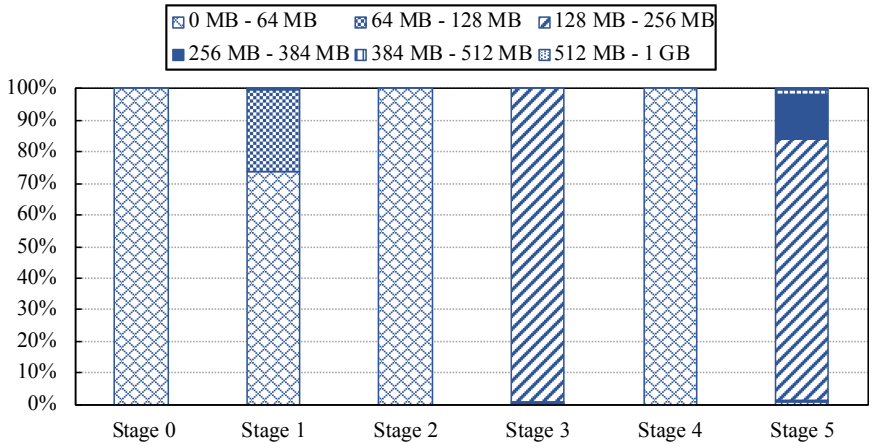


Figure 5.3: Input data size at different stages of a production Spark application

We observed in the targeted production system that most of the executor memory resource is used for two purposes: persistence and shuffling. Based on the observation, we focus on optimizing the shuffle and persistence mechanisms in Spark with in-memory distributed file system DMO and persistent memory (PMEM). DMO helps to improve the memory utilization while PMEM helps to increase the cluster's memory capacity with affordable cost. In the next section, we show that the I/O throughput of PMEM can meet the requirements of the targeted production system.

5.3 Characterizing Remote PMEM

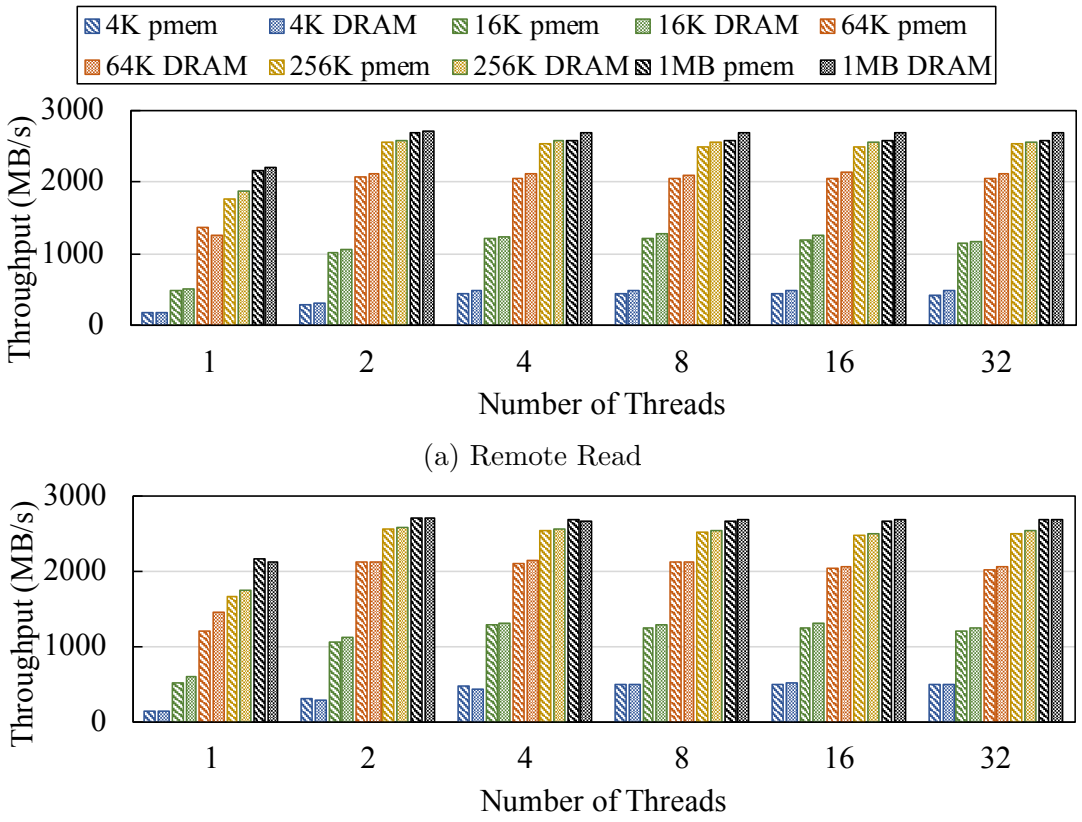
Recent architecture developments feature DIMM-based PMEM, which utilizes novel storage media to achieve a much higher data density than DRAM. Moreover, the per GB cost of PMEM has also been projected to be much lower than that of DRAM. There is no layer between memory and storage in most modern computer architectures, but there is a large performance gap between memory and disk. PMEM has been proposed recently both by industry and academia to fulfill such a performance gap. Therefore, PMEM can be considered as larger but slower memory or faster persistent storage. In this work, we empirically characterize and compare the read/write performance of PMEM and DRAM remotely. Furthermore, we explore whether the read/write throughput of PMEM is sufficient to be used as a remote disaggregated resource via RDMA technology.

5.3.1 Experimental Setup

Since our evaluation investigates the performance of using a disaggregated memory pool for Spark workloads, we mimick the scenario where Spark executors read from and write to remote PMEM: we set up a two-node cluster within a 25 GbE network (this is the current network bandwidth used across the whole data center). Among these two nodes, one node serves as the external PMEM server, and the other emulates the Spark compute node. We let the external PMEM server be equipped with two Intel Xeon Gold 6252 CPUs @ 2.10 GHz with 24 physical cores, 192 GB of DDR4 memory and 1.5 TB PMEM samples in DIMM form factor from a major vendor. The compute node uses two Intel Xeon E5-2650 v4 @ 2.2 GHz with 12 physical cores and 256 GB DDR4 memory. Both nodes use Mellanox ConnectX-4 Ethernet cards. The operating system is CentOS 7.5 with default 3.10 kernel that comes with PMEM support. PMEM in our system can be configured in different modes, allowing user applications to treat it as either volatile memory or persistent block or character device. In this work, we use the latter approach and configure all the PMEM as device DAX [128]. This further allows us to build a highly customized distributed in-memory storage system on top of the PMEM as we show in section 5.4.

5.3.2 Remote PMEM Performance

We further build an internal remote memory profiling tool that measures the throughput of single-sided RDMA read and write operations sent to the external memory node and compare the remote access performance between PMEM and DRAM. To better understand the sequential read/write performance of remote PMEM, we use different data transfer sizes, from 4 KB to 1 MB, with the number of threads ranging from 1 to 32. Figure 5.4 shows the remote read and write throughput of PMEM and DRAM, respectively. The results indicate that both remote PMEM and DRAM access can almost saturate the 25Gb network bandwidth at higher I/O sizes. Moreover, PMEM can offer the same remote read/write throughput compared to DRAM as throughput starts being bottlenecked by the network instead of the bandwidth of PMEM itself.



(a) Remote Read

(b) Remote Write

Figure 5.4: Remote RDMA read/write throughput for PMEM and DRAM using 25GbE network

5.3.3 PMEM Viability for Remote Memory Implementation

Although existing data centers deploy servers with different types of resources (e.g., CPU, DRAM, GPU, HDD, SSD) that are typically used within a server, our observations discussed above indicate that the server-centric architecture cannot fully utilize the computing resources available in the data processing clusters. For example, in current data centers, usually only a limited number of servers feature advanced storage devices and large volumes of memory while most servers are equipped with standard hardware configurations. Because of this, it is not realistic to approach a high performance and cost effective solution based on the traditional server-centric data center architecture. The advances and continuous cost reduction in communication networks (e.g., RDMA) enables computing nodes in modern data center to leverage remote resources efficiently. The analysis above clearly show that PMEM is a solid candidate for

implementing extended remote memory solutions for enterprise data centers.

In this work, we design and implement disaggregated memory pool with DRAM and PMEM. Figure 5.5 shows the basic architecture of our system, co-designed with Spark. Our system stores all spill, persistence and shuffling data in the disaggregated memory pool. To solve the problem discussed in Sections 5.1 and 5.2, we use a server featuring large volume of PMEM as supplemental memory for the disaggregated memory pool. Moreover, to ensure the high availability of remote PMEM, we connect the servers via fast fabric interconnection. In the next section, we present the detailed design and implementation of our system.

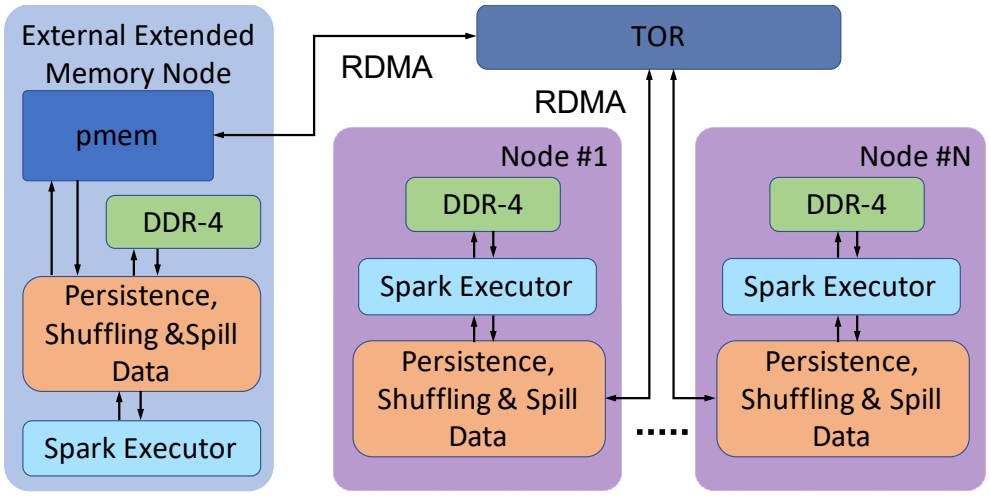


Figure 5.5: Extended memory design with remote PMEM

5.4 System Implementation and Deployment

We present the design and implementation of two key issues of our envisioned system: (1) a distributed in-memory storage system named distributed memory objects (DMO); and (2) the integration of DMO with Spark via memory-speed RDD storage and a newly designed Spark shuffle manager.

5.4.1 Distributed Memory Objects (DMO)

DMO is designed as next-generation data infrastructure software optimized for memory-centric computing and high bandwidth networks. It aims at providing memory-speed

data persistence and very fast data exchange that are highly demanded by distributed in-memory computation frameworks such as Spark. DMO offers a large PMEM pool by aggregating PMEM resources from its member nodes. For Spark, the pool becomes a disaggregated memory resource that extends the off-heap memory capacity for every Spark executor. Current Spark utilizes off-heap memory for accelerating RDD caching/storage and storing data used by shuffle tasks. With the help of PMEM as well as our RDMA-based networking framework, accessing any data in the pool has been made even faster than accessing local disks.

DMO consists of client and storage backend. The client integrates with user applications through a set of APIs that performs data and metadata operations on the storage backend. The storage backend is a remote cluster of multiple PMEM-equipped servers where each node executes one or more of the following DMO components: name server and object store. Across nodes, communications within DMO are purely through RDMA over Converged Ethernet (RoCE). In particular, we use single-sided RDMA write/read for data transfer and use double-sided RDMA send/receive for RPC messages. We briefly describe each of the DMO components in the following paragraphs.

DMO client exposes a set of APIs for data and metadata operations. Currently, Java, C, and C++ versions of object APIs have been implemented. Table 5.1 describes the major APIs used in this work. These APIs are sufficient for supporting the integration of DMO with Spark.

The name server records the locations of a DMO object's metadata and some other

Table 5.1: Major DMO client side APIs

API	Description
`connect`	Establish connections with DMO backend.
`disconnect`	Drop an existing connection with DMO backend.
`create`	Create an empty object in DMO.
`write`	Write data to some offset of an existing object.
`read`	Read some offset of an existing object.
`get_attr`	Retrieve the attributes of an existing object.
`delete`	Delete an object from DMO.
`create_dir`	Creating an empty directory for holding objects.
`remove_dir`	Remove a directory.

attributes. In DMO, metadata are stored in an object store, as described below, and records the object's data chunks' address information. The name server further maintains another map that tracks the information of directories, such as what objects are contained inside a directory. Multiple name servers can be deployed inside a DMO cluster to handle a large number of objects. In such a scenario, entries in name servers are sharded and replicated using consistent hashing [129].

A DMO object consists of metadata chunks and data chunks. All these chunks are held by the object store. Metadata chunks mainly hold each data chunk's address information, including node index, PMEM device index, and the starting offset of the data chunk in the PMEM device. As all the data are stored in memory, data operations are done through load and store using memory copy. To guarantee data persistency, we use `clflush` instruction to flush CPU cache back to PMEM region [130].

Caching is still needed to achieve close-to-DRAM performance as PMEM is still lower in performance compared to DRAM. We chose to cache an object's metadata in DRAM when the object is accessed for the first time.

Furthermore, when an object's data is accessed by a remote node, a copy of the related data chunks is cached on the remote node after the first read. Cached chunks are evicted either upon a timeout or when the available space for caching in the node is approaching a limit specified by the user when configuring DMO.

5.4.2 Integrating DMO with Spark

We made two major efforts to integrate DMO with Spark: (1) a modification of Spark to allow persisting RDD into DMO; and (2) a Spark shuffle manager based on DMO.

Persisting RDD to DMO: In Spark, RDD can be cached in memory or disk in order to avoid recomputation of lineage [125]. By default, RDD is cached using the `persist` API. The caching policy specifying the media RDD will be placed on can be passed in as an input parameter. We modified Spark to allow RDD to be persisted directly into the external PMEM pool of DMO. This is done by adding a new RDD persist policy, while augmenting the implementation of the `persist` function of Spark's block manager. To further enable RDD retrieval from DMO, we modified the

function for reading RDD data blocks in block manager. Besides enabling the use of remote PMEM for RDD storage, our modification further replaced the TCP/IP based Netty communication framework used for cross-node communication with our RDMA framework used by DMO.

Shuffle with DMO: We implement a customized shuffle manager based on DMO. Compared to default Spark shuffle manager, mappers write shuffle outputs directly to remote PMEM of DMO instead of to local DRAM and disks; reducers read shuffle output by pulling data directly from DMO instead of each Spark mapper node. All the remote data operations are efficiently carried out with our RDMA-based networking layer instead of the TCP/IP based Netty framework.

Building a pluggable shuffle manager requires us to implement the shuffle reader, the shuffle resolver, and the shuffle writer components specified by the `ShuffleManager` trait of Spark. Figure 5.6 illustrates the structure of our implementation.

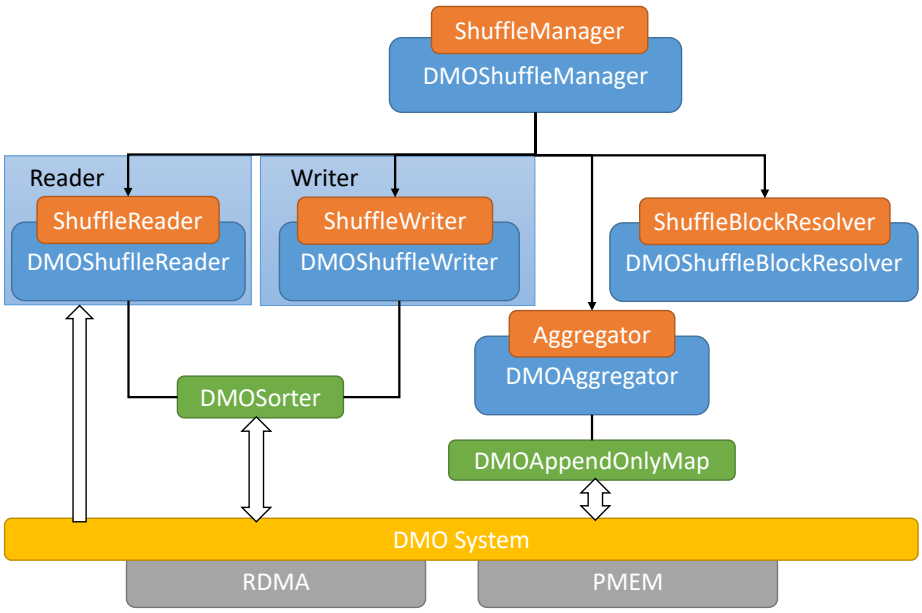


Figure 5.6: The structure of DMO-based shuffle manager

A shuffle task is performed in two consecutive stages: map and reduce. During map stage, a mapper uses `DMOShuffleWriter` to produce a shuffle output. The output is made of an index file and a data file, which are stored as separate objects by

DMO. Data file contains multiple partitions, and each partition stores the data to be read by one reducer. Offsets marking the starting positions of the partitions in data file are kept in index file. To allow reducer to be able to retrieve needed partitions correctly, mapper registers with `DMOShuffleBlockResolver`, assigning the partitions of the mapper to corresponding reducers. In reduce stage, each reducer uses `DMOShuffleReader` to retrieve all the partitions belonging to the reducer following the guidance of `DMOShuffleBlockResolver`. The reducer then merges all the partitions together.

Both `DMOShuffleWriter` and `DMOShuffleReader` utilize `DMOSorter`. A sorter receives partitions written by mapper or read by reducer, and insert them into an in-memory collection for fast computation. During the insertion, it performs aggregation and sorting if specified. However, when memory assigned to a sorter is not sufficient for holding all the data, spilling part of the data to disk is required to avoid out-of-memory (OOM) failure. In our implementation, `DMOSorter` starts spilling part of the in-memory data by serializing and writing them to DMO. Therefore, DMO objects holding spilled data are treated as part of the spilled collection of a `DMOSorter`.

An aggregator is used when shuffle is triggered by "group by" computations. Depending on the characteristics of input data, aggregator of default Spark shuffle manager may also spill in-memory data to local disk when there is not enough memory. Similar to `DMOSorter`, we implement our own `DMOAggregator` to speed up data spill by directly writing spilled data into DMO. We also implemented some custom logic to avoid out of memory issues when the result of aggregation consumes too much memory.

5.5 Experimental Evaluation

5.5.1 Workloads and Experimental Setup

We selected three different workloads to evaluate the performance, memory efficiency, and scalability of our system. Table 5.2 shows the input size and characteristics of the workloads, and Table 5.3 shows the shuffling size and persistence RDD size of the workloads.

Table 5.2: Input size and characteristic of Workloads

Workload	Input Size	Characteristic
Terasort	600 GB	I/O intensive
Warehouse application	200 GB	I/O intensive
price protection application	726.9 GB	I/O, CPU intensive

Table 5.3: Shuffling size and Persistence RDD size of Workloads

Workload	Shuffling Size	Persistence Size
Terasort	334.2 GB	N/A
Data warehouse application	234.7 GB	N/A
Price protection application	57.6 GB	349 GB

We select TeraSort from HiBench [131]. TeraSort is a standard shuffling intensive benchmark well suited to evaluate the I/O performance (especially shuffling performance) of big data frameworks, such as Spark.

Core service of data warehouse application: We selected a typical data warehousing scenario in an e-commerce company that provides core data joining and aggregation services to various businesses, including user information, order information, shipping information, and storage information. It consists of more than 150 Spark SQL-based applications. Since it provides core data to a large amount of downstream customers with explicitly defined SLAs, reducing its execution time is critical. This workload is I/O-intensive (both disk and network), as it is based on the *select*, *insert*, and $fullouterjoin$ operations.

Price protection service: This is a core service of large e-commerce companies that typically suffers frequent price DDoS attacks, such as coordinated product price crawling. The price protection application can find abnormal information, e.g., IP addresses, using several different strategies. With these information, the price protecting strategy can help data scientists make better decisions regarding product price to minimize the loss from price DDoS. Since the price protection application reuses RDDs tens of times, it is both I/O intensive and computing intensive.

The experimental setup includes one server featuring PMEM technology and 10 regular production enterprise servers. The regular server is equipped with two Intel(R) Xeon(R) CPU E5-2650 v4 @ 2.20GHz with 24 cores leveraging hardware threads, and

256 GB of DDR-4 RAM memory. Spark version 2.3 was deployed using standalone mode, CentOS 7.5 and HDFS version 2.7.

5.5.2 Experimental Results

We evaluate the performance of the three workloads with 60 Spark executors, each using five cores and a different amount of memory varying from 1 GB to 20 GB. We determine the total memory utilization from the allocated executor memory in Spark and the storage memory in DMO, i.e.,

$$\text{Mem}_{\text{total}} = \text{Mem}_{\text{Spark}} + \text{Mem}_{\text{DMO}}$$

Note that the PMEM usage is also accounted into the total memory utilization.

Based on the above experimental setup, we evaluate the performance of our system with four configurations: (1) Spark, (2) Spark with DMO Local (all DMO data chunks written/read to/from local DRAM, no remote PMEM), (3) Spark with DMO Remote (all DMO data chunks written/read to/from remote PMEM), and (4) Spark with DMO Local with caching.

Overall performance: Figure 5.7 shows the execution time of Spark with different executor memory and DMO memory using different workloads. For TeraSort, default Spark cannot successfully finish until the executor memory is more than 10 GB (total memory is 600 GB). This is because the tasks cannot obtain enough local memory which leads to significant Java GC, even OOM, and executor connections are closed because the bandwidth of local disk is too low. Consequently, as shown in Figure 5.7, the minimum required memory for default Spark is 600 GB, while our proposed system is 410 GB. The most expensive operation in TeraSort is shuffling, because Spark must write/read large amounts of shuffling data on disk. With our optimization of shuffling (i.e., external shuffling memory instead of disk), our system can reduce the execution time by up to 40% compared to default Spark with the same memory consumption. Figure 5.7 also shows that the performance is similar comparing using DMO with local DRAM and with a disaggregated remote PMEM pool.

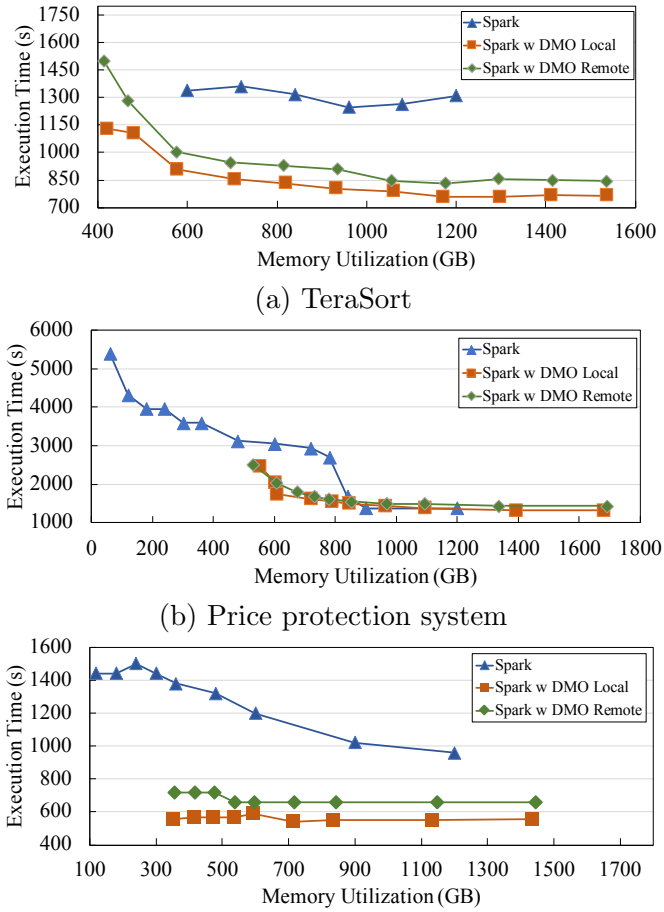


Figure 5.7: Total memory - Execution time of TeraSort, price protection system and data warehouse application

The price protection system first generates the intermediate RDDs, which are reused tens of times in the subsequent stages. We store persistent RDDs with serialized Java objects in Spark. Our system stores shuffling data, spills, and persistence data in DMO, which is backed by a distributed DRAM and PMEM memory pool, which can offer sufficient storage memory to Spark. Furthermore, the re-computation strategy enables default Spark to finish all stages with insufficient storage memory for the JVM. However, significant execution overhead was introduced because of re-computation. With a greater executor memory capacity, the default Spark can achieve the best performance with more than 14 GB of memory per executor, with a total of 840 GB of allocated memory. The best performance of our proposed system provides a 4.3% execution time reduction with the same memory utilization. Although we do not

optimize Spark's persistence strategy, the system provides performance improvement from garbage collection reduction. Note that compared to default Spark, the proposed system with DMO needs larger minimum total memory to finish the workload, as the additional memory is used to persist all intermediate RDDs with disaggregated memory pool in these experiments.

The core service of the data warehouse application is a typical Spark SQL application that involves a large volume of data. This workload profile is similar to the TeraSort application. As shown in Figure 5.7, our proposed system can reduce the execution time up to 59.5% with the same amount of memory compared with default Spark.

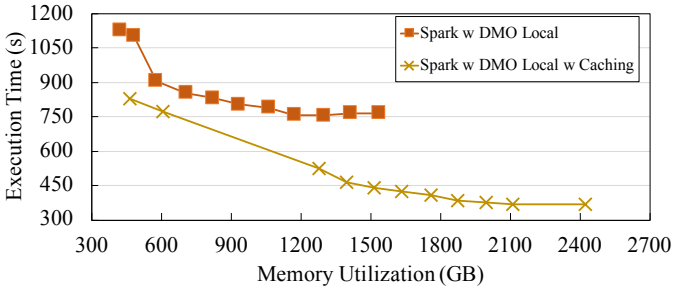
Computing resource efficiency: As discussed in Section 5.1, the data center memory utilization is usually higher than the CPU utilization in the data center. Since each regular production server used features 256 GB of DRAM already, increasing the memory size further may not be possible.

We use one server featuring large-capacity PMEM to increase the overall memory size and optimize the use of computing resources. Specifically, we increase by 66.5% of the cluster's overall memory capacity with ten regular nodes by adding one server with 192 GB of DRAM and 1.5 TB of PMEM. The performance overhead is only 10.5%, 9.1%, and 18% for TeraSort, price protection, and data warehouse application, respectively. In the worst scenario, all DMO reads and writes are from/to remote PMEM, compared to all DMO reads and writes from/to local DRAM.

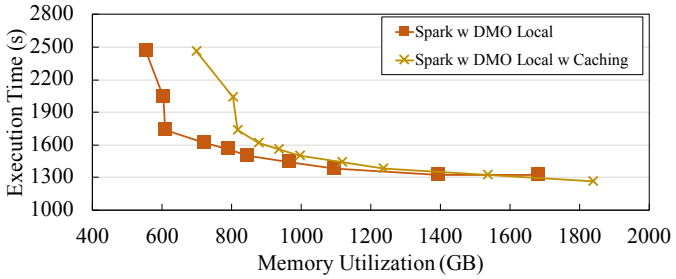
Additionally, the extended memory is shared across different executors in a Spark application for persistence and shuffling because of the disaggregated design. It eases the performance tuning in the production scenario and improves the overall memory utilization, especially for imbalanced data partition cases.

Quality of service: To meet service-level agreements (SLAs) in enterprise data centers, system engineers tend to assign large volumes of memory to Spark applications even though, in some cases, Spark can finish jobs with lower memory requirements. It usually leads to memory resource waste. For example, assuming a price protection application is required to complete in less than 2,000 seconds, an engineer could allocate 840 GB of memory to the application using default Spark. However, the engineer would

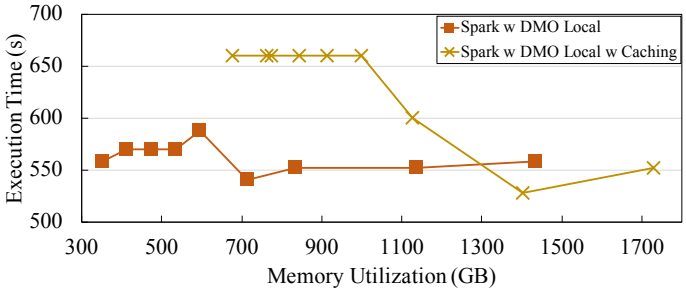
still worry about running out of memory occasionally due to input data change, which can ruin the SLA. In contrast, our proposed system's memory requirement would be only around 600 GB, and it would be more scalable to data change.



(a) TeraSort



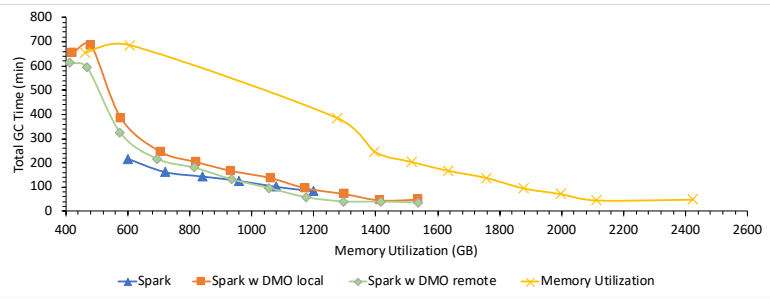
(b) Price protection system



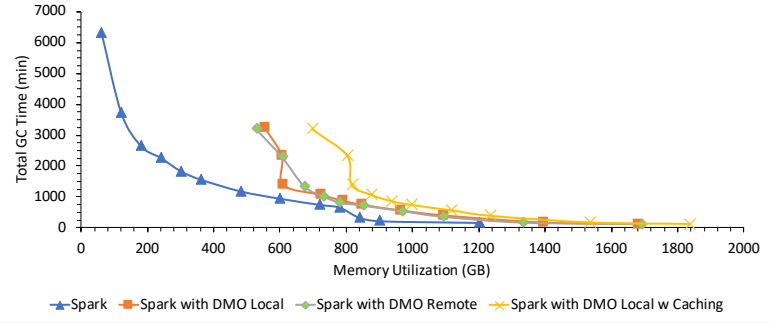
(c) Data warehouse application

Figure 5.8: Total memory - Execution time of TeraSort, price protection system and data warehouse application

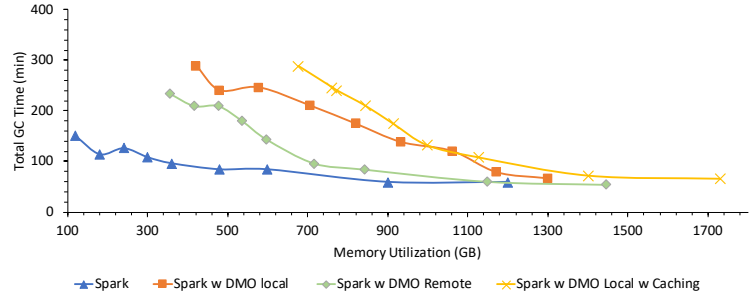
Impact of caching: Figure 5.8 shows that with caching optimization, TeraSort can further reduce its execution time by 52% with the same amount of overall memory, which is up to 3.5-fold performance improvement comparing to default Spark. However, for the price protection and data warehouse applications, caching does not show obvious benefits. This is because TeraSort does the sort operation at reduce side, which produces heavy I/O loads. Thus, TeraSort can benefit from caching, while it is not the case for the other two workloads.



(a) TeraSort



(b) price protection system

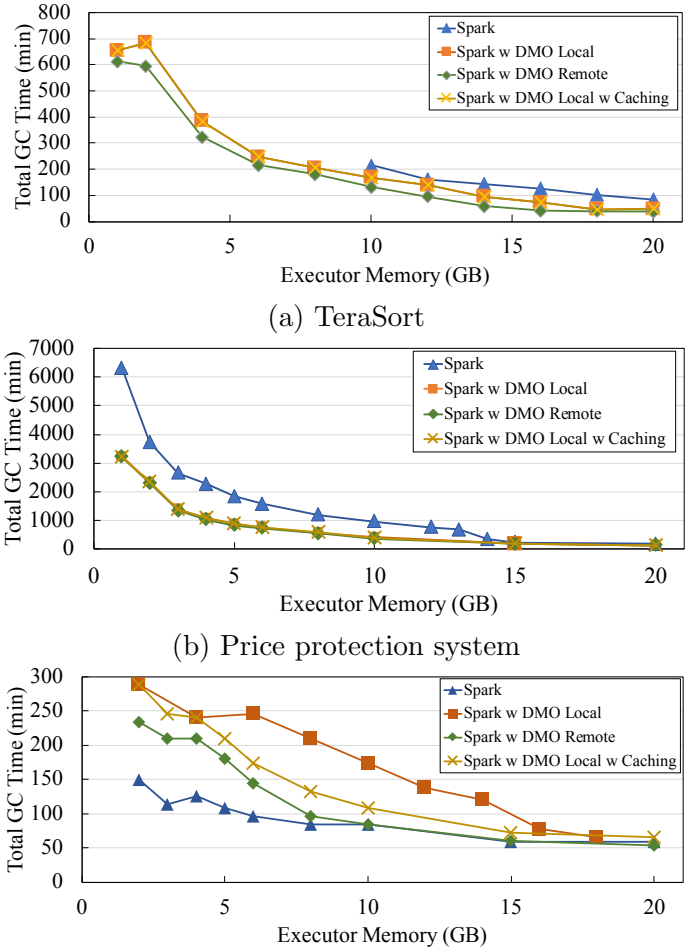


(c) Data warehouse application

Figure 5.9: Total memory - GC time of TeraSort, price protection system and data warehouse application

Impact of garbage collection (GC): We observed in production environments that the GC of a Spark application could represent a significant fraction of its execution time. The most common approach to avoid performance degradation due to GC is to increase the size of executor memory in Spark. Figures 5.9 and 5.10 show the total memory utilization and GC time of default Spark and Spark with DMO. As shown in the figures, our proposed system can save up to 42% of the total GC time for TeraSort and price protection applications with the same amount of executor memory because the persistence and shuffling data are offloaded to DMO. However, for the data warehouse application with low memory utilization, the GC time in Spark with

DMO could be larger than the default Spark. This is because we do not implement the `BypassMergeSortShuffleWriter` in DMO, which is used when the number of partitions is no more than 200 by default which is the case for the data warehouse application. With the same executor memory size, DMO (external extended memory) is expected to decrease the GC time.



(a) TeraSort

(b) Price protection system

(c) Data Warehouse application

Figure 5.10: Executor memory - GC time of TeraSort, price protection system and data warehouse application

N+1 architecture: We propose a system with a disaggregated memory pool, which can utilize the extended PMEM with N+1 architecture (N regular nodes with one PMEM node). We evaluated the network throughput of the server featuring PMEM in our system with the TeraSort application. We increase the number of executors and the size of data as the number of regular nodes used for computation increases. Figure 5.11

shows the average incoming throughput for shuffle write phase and outgoing throughput for shuffle read phase on the single remote PMEM server. We aim at finding a trade-off between the bandwidth of the computing cluster and the data center's computing resource efficiency. As shown in Figure 5.11, as the application load increases, the shuffle write throughput can reach the available network bandwidth in our system. It also provides meaningful data points for identifying efficient enterprise data center system design choices.

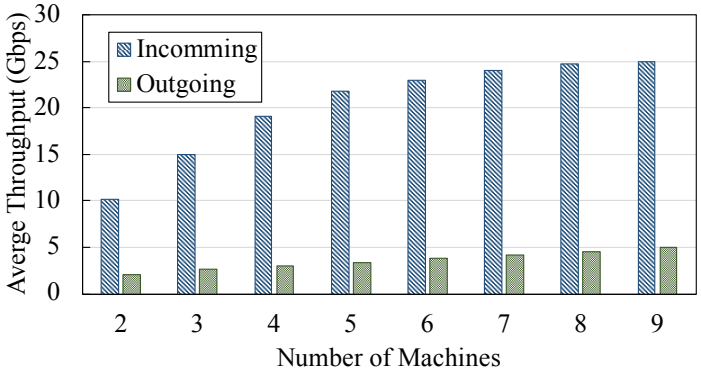


Figure 5.11: Average network throughput of PMEM-based server with TeraSort

5.6 Discussion

In this chapter, we provided a solution for persistent memory for in-memory big data processing frameworks and presented the design and implementation of a disaggregated memory system. Our proposed system optimizes the data center's memory efficiency with external extended persistent memory and a disaggregated memory pool. By leveraging shuffle and persistence optimization, we demonstrate that our system can effectively reduce the execution time by up to 72% for shuffle-intensive applications. Moreover, we incorporate shuffle and persistence mechanisms into big data frameworks using an in-memory distributed file system. The experimental evaluation results from empirical executions show that with remote large-volume persistent memory and a disaggregated memory pool, our proposed system can also increase the overall memory capacity by 66.5% with much lower overheads. While the experimental evaluation has been conducted using a 25 GbE commodity network, we expect our proposed system

to provide significantly better results using faster networks (e.g., 40/100 GbE or Intel Omnipath technology). It shows the viability of PMEM for implementing bare-metal software-defined infrastructures in production enterprise environments.

Based on this research and proof of concept, a large e-commerce company is looking at deploying the solution with persistent memory in production for shuffle/persistence intensive Spark applications with high SLA requirements. The plans include deploying Spark on a container-based Kubernetes deployment on a private cloud environment, mixed with other workloads to improve resource utilization. The solution fits nicely with the enterprise's deployment requirements as it eliminates the dependency on the local disks, which is difficult to virtualize and manage in the targeted environment. With Spark executor in the cloud, the solution provides truly elastic means to run Spark applications.

Chapter 6

In-Transit Shuffling for Large-Scale Data Analytics

Building modern big data services in large-scale enterprises such as JD.com requires addressing bottlenecks of existing shuffle services. First, the existing shuffle service is not fully compatible with cloud environments, which causes the failure of large-scale data analytics. There is a consensus I/O overloading challenge in developing a native cloud platform with modern data analytics [132]. Although modern big data frameworks handle part of failures with the DAG engine, there is an extra cost because of data re-computation. In the worst case, frameworks can not finish the work because of the continued errors. Secondly, the existing Spark shuffle service is sorted based. It adds some unexpected high I/O burden on the system caused by data skew, large data, and non-optimized configuration. We expect all shuffle data to fit into memory in the ideal scenario, which has the best performance [133, 134]. However, Spark can not always sort shuffling data in memory because of the limited memory capacity. The shuffling data sorting with a large amount of data is common in production environments for several reasons, including 1) a large amount of shuffle data with an inefficient number of executors, and 2) the imbalance data distribution because of data skew. The high I/O burden causes the failure of the current job and causes the performance degradation or failure of co-located jobs.

An effective way to overcome these challenges is building an in-transit shuffle service, which is partially isolated from existing data analytics frameworks. The critical observations of the bottleneck of shuffling are related to three issues:

1. The status of computing workers become unhealthy because of the high demand of I/O bandwidth.

2. The existing shuffle service is not native stateless to a container-based scheduler,

which makes the design and integration between data analytics and container-based scheduler more complex.

3. The external shuffle service highly relies on the customized distributed file systems, which is not a general solution for data analytics frameworks.

For example, Crail [102] implements a high-performance master to satisfy the high demand for I/O requests. Sailfish [103] customizes the distributed file system with the multi-writer. However, the spinning hard disk is much more cost-efficient in a real production environment, especially with large amounts of data [135, 136].

6.1 Comet Overview

This thesis addresses the critical challenge of shuffling for large-scale data analytics with Comet, an in-transit shuffle service, which provides solutions to the stability issues of large-scale data analytics in cloud environments. This chapter focuses on the design, implementation, and evaluation of Comet.

Comet stores the shuffle data into the distributed file system instead of the local disk, which has better fault tolerance than the local shuffle service. Compared to the existing solutions, Comet does not need a redundant scheduler and expensive hardware, which is suitable for most cloud environments with minimum cost. At the same time, as a remote shuffle service, it decreases the local disk burden compared to the existing solutions and is more friendly to co-located jobs. Furthermore, this in-transit shuffle service has better accessibility, which is designed with stateless indexing. Key technical challenges of this Comet's contributions are described as follows:

1. Comet is an affordable shuffle service, which does not rely on customized high-performance distributed file systems or high-performance hardware. The main bottleneck of accessing intermediate data in the distributed file system is the limited performance of the master. However, the most popular distributed file systems, such as HDFS, are not designed for dealing with a larger amount of metadata, which is a series of small files. Thus, the performance of the master

becomes the main bottleneck for storing intermediate data, like shuffle data. To reduce the number of I/O requests to HDFS, Comet aggregates shuffling data based on reducers' destination.

2. Comet is a lightweight shuffle service. The indexing of shuffling data in Comet relies on the file paths of HDFS, which has no extra cost of data management and scheduler. Compared to Sailfish [103] and Riffle˜ [101], Comet is a stateless shuffle service with lower deployment and scheduler costs.

3. Comet is a compatible shuffle service to existing data analytics frameworks. Since existing data analytics frameworks, e.g., Spark, have recovery mechanisms for fault tolerance capabilities, which are complex and diverse from each other. Comet is compatible with recovering mechanisms without changing the recovering logic. With stateless shuffle service, we can build a unified in-transit shuffle service, which has better extensibility.

Comet was deployed in an enterprise production environment for more than half a year, which provides validation of the approach as it supported the enterprise to build a Petabyte level data warehouse with modern data analytics frameworks.

6.2 N-to-N communication in Spark

Spark can process large amounts of data effectively with a large number of computation tasks based on the Map-Reduce model. Compared to Hadoop, Spark has better performance on iterative data processing with data reuse. One of the main bottlenecks for large-scale parallelism in Spark is shuffling, which is an n-to-n communication. As illustrated in Figure 6.1, shuffling re-distributes data into different executors, as needed. As we observed in real production Spark clusters, shuffling is one of the most time-consuming operations in data analytics at scale.

Currently, Spark is widely used in large-scale enterprises such as JD.com data infrastructure for large-scale data analytics. While building the data warehouse with Spark, it was noticed that the performance of Spark SQL is not as good as expected for some

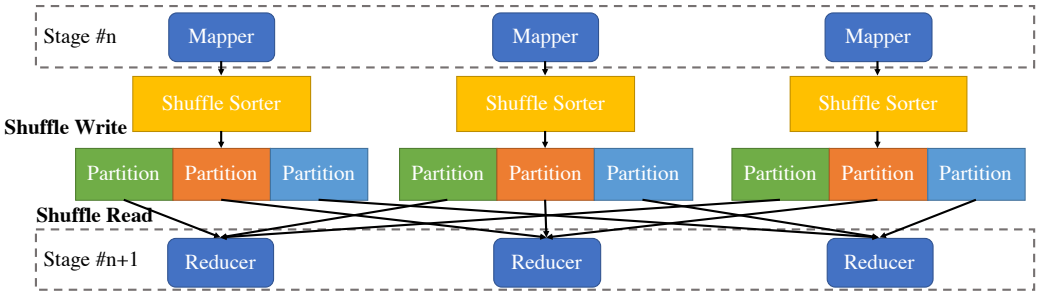


Figure 6.1: Sort Based Shuffle, which sort the data partition based on the key before send out to next tasks

large-scale ETL jobs. While it provides excellent performance in sophisticated data analytics, especially iterative data analytics, it does not provide such a performance with one-pass ETL workloads. Furthermore, Spark is unstable in large-scale data processing, which could be more than tens of Terabytes, with highly compressible columnar data format like Parquet.

The main shortcomings of the Spark shuffle service to dealing with large-scale data are around the following three aspects:

- The existing sort-based shuffle service reduces network requests' burden while increasing the bandwidth demand of local disk. The shuffling data of Spark is spilled to disk when memory is insufficient for the whole data set. However, the read and write amplification is high for sorting.

- The current Spark is not good at dealing with serious data skew jobs, which is common in a production environment. We noticed that Spark has a critical long time problem because of data skew. For example, many data analytics, which have unexpected skewed data, write more than 90% of shuffling data into a single node.

- It is hard to achieve the best performance of Spark with reasonable memory on a local computing cluster. It is not easy to achieve the best performance in the production environment since we can not afford a large amount of memory in the computing cluster. An effective solution could be removing some unnecessary computation and storage memory to external shuffle clusters.

Comet's basic idea to increase Spark's stability and higher performance is building an in-transit shuffle service and partially split the Spark shuffle service. Comet is friendly to the Hadoop ecosystem, which is compatible with the HDFS interface. Thus, Comet is a stable in-transit shuffle service with an existing distributed file system, which does not need extra effort on modifying the distributed file system.

The following sections presents the architecture overview of Comet and discusses the co-design between the Spark and HDFS interface. We also discuss the benefits of a stateless shuffle service compared to existing shuffle services.

6.2.1 Design Requirements

Compared to existing solutions, Comet does not need an additional scheduler to manage the shuffle data. To design Comet, we consider the following essential technical requirements:

Data consistent mechanism. We do not aim at developing an additional framework for the existing shuffle service. Adding an extra scheduler incurs software cost to the whole system and increases the deployment and maintenance costs. To do so, we separate the shuffling mechanism from Spark and design Comet to ensure data correctness between stages.

Affordable shuffle service. Comet should not rely on customized distributed file systems, which are not commonly used in cloud environments. We design and implement Comet based on a co-design approach with the Hadoop ecosystem, which has good compatibility with cloud-based distributed file system interfaces like HDFS [9], S3 [111] and Alluxio [10]. Furthermore, Comet does not require the distributed systems to have a multi-writer and multi-reader.

Failure handling. Since each different data analytics framework have a different failure handling mechanism, it is unnecessary and unfeasible for Comet to handle every failure by itself. To achieve better extensibility, we target a compatible shuffle service instead of an independent shuffle framework. To match Spark's failure handling, Comet partially isolates the shuffle service from the computing cluster, but Comet is still compatible with the existing DAG engine of Spark.

Data flow controller. The local shuffle service continues to write shuffle data into the local disk without data flow control. However, since Comet writes shuffle data into a remote shuffle cluster, it is necessary to have a data flow controller to avoid the overload of the shuffle cluster.

Performance requirement. Although Comet's primary goal is not for performance optimization, we still target Comet's performance to be comparable to the current local shuffle service. However, it is hard to implement a shuffle service, which is dependent on the remote shuffle cluster and current shuffle mechanism, with the same performance as the local shuffle service. Thus, we build a destination-based aggregation shuffle service, optimizing performance for block-based file systems for Comet.

The following sections describe Comet's overall architecture illustrated in Figure 6.2 and discusses implementation issues such as how Comet ensures the data consistency and stability of the computing cluster.

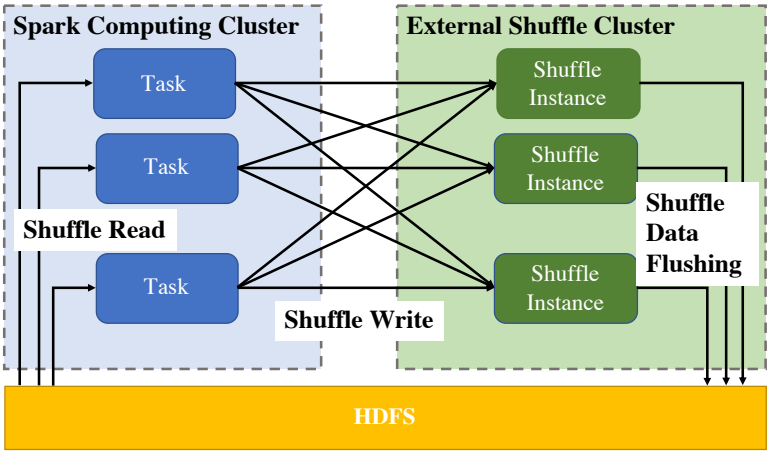


Figure 6.2: Comet's overall architecture

6.2.2 Data Pipeline with Data Consistency Mechanism

Comet is designed as an in-transit shuffle service, which is a highly reliable and high-performing intermediate layer for Spark. One of the most important issues is ensuring data consistency between two different stages in Spark. The system has to overcome two challenges: 1) handle failures across stages with the existing Spark DAG engine, and 2) handle failures between Comet and HDFS. We next describe how Comet is compatible

with Spark DAG engine and how Comet ensures the data consistency between two Spark stages.

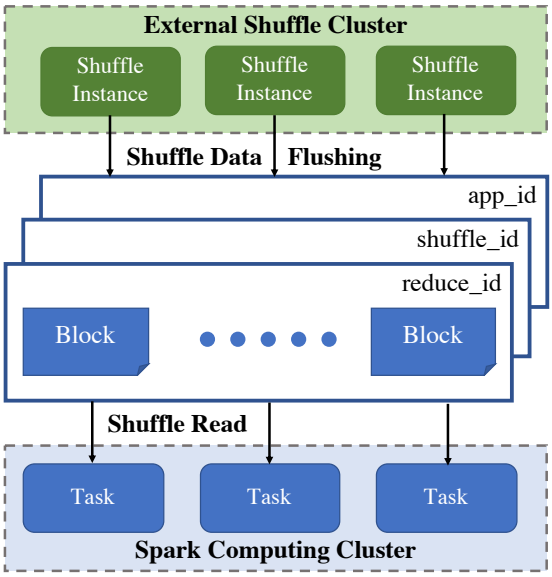


Figure 6.3: Shuffle data indexing of Comet

Stateless shuffle service. As opposed to existing solutions, Comet does not use an independent scheduler to manage the metadata for shuffle data. We implement the stateless indexing for shuffle data, which relies on HDFS files' pathname, as shown in Figure 6.3. It is worth noting that this stateless shuffle indexing without the requirement of an individual scheduler.

Since the Hadoop distributed file system is the dependency of the existing system, Comet builds the indexing based on the HDFS file's pathname instead of storing the shuffle data indexing in the individual scheduler. Comet creates the path during the shuffle write phase, which follows the rule $/appid/shuffleid/reduceid$, in HDFS. Then the task can read the corresponding shuffle data based on the same rule. In this situation, Comet does not have to have another scheduler to store shuffle data indexing.

Data consistency checker. To check the consistency of shuffle data, we keep the metadata of shuffle data inside the Spark driver, although Comet separates the shuffling mechanism from Spark. As shown in Figure 6.4, Spark driver records the unique identification for each shuffle partition. Once the reduce tasks fetch the shuffle data, they also check the correctness and completeness to ensure the shuffle data consistency

between two stages.

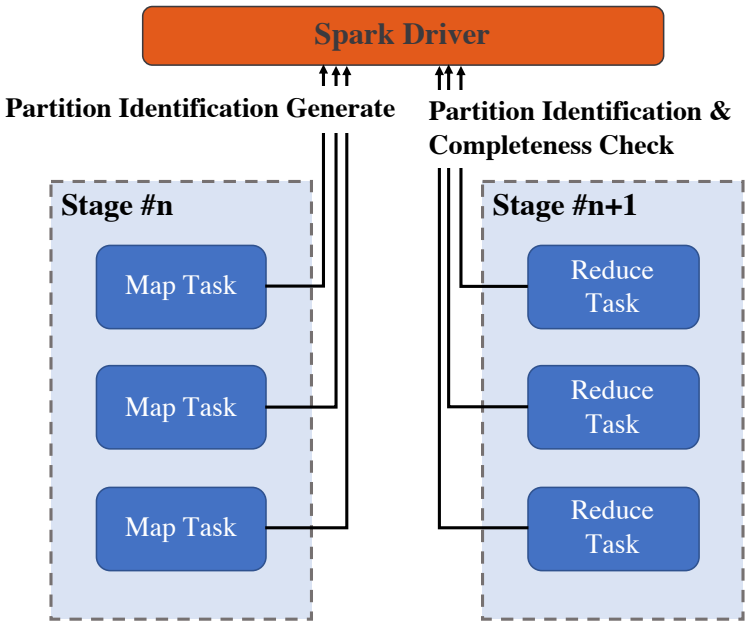


Figure 6.4: Comet's consistency checker architecture

DAG engine compatible shuffle service. One of the most sophisticated mechanisms in data analytics frameworks is the data recovering mechanism. In Spark, the lost data, which is damaged for some reason, is protected by the DAG engine. Once a Spark job loses part of the data, it recomputes a series of tasks to re-generate the lost data with the DAG engine. Since the complexity of the recovery mechanism and, considering the expansibility between in-transit shuffle service and existing data analytics frameworks, Comet is designed to be compatible with the Spark DAG engine without involving Spark's DAG engine.

Although Comet is not responsible for recomputing data because of data corruption, it is fully compatible with the Spark data recovery mechanism to ensure the fault tolerance of storing shuffle data. The shuffled data is protected with two different Comet mechanisms: 1) the Spark recover mechanism protects the shuffle data outside of the shuffle cluster, and 2) the shuffle data inside the shuffle cluster is protected by the HDFS system. In this way, Comet becomes a stateless shuffle service with good fault tolerance. The mechanism is illustrated in Figure 6.5. In the figure, the shuffle data, which in mapper, is protected by Spark DAG engine. The shuffle data, which is

successfully written into HDFS, is protected by HDFS's three copy mechanism

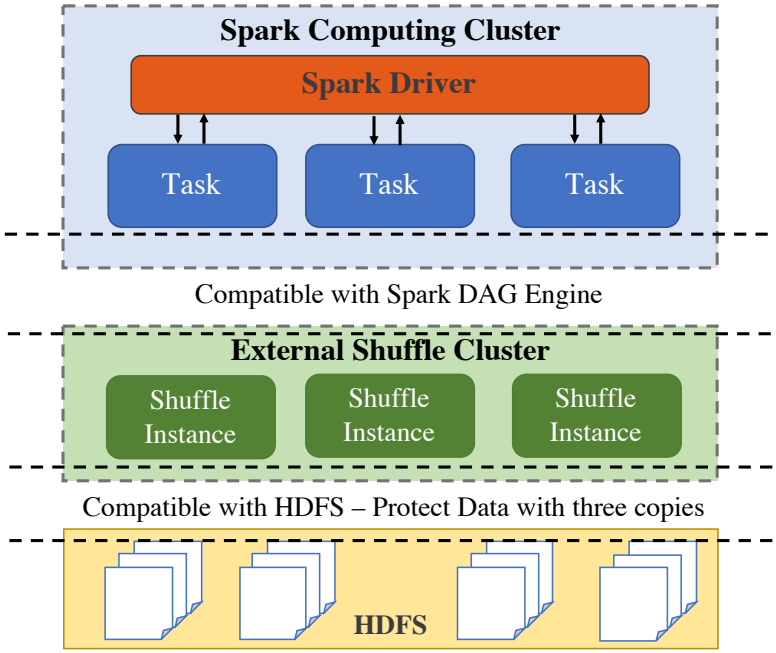


Figure 6.5: Stateless design of Comet, which is fully compatible to Spark recover mechanism.

Comet requires the mapper to keep running until all shuffle data, which is generated inside the mapper tasks, is successfully flushed into HDFS. By keeping the mapper running, Comet does not need to keep track of the data flow inside Spark applications. Comet also benefits from the fault tolerance mechanism of HDFS, which does not need further effort to maintain the mapping between the data block and Spark master.

6.2.3 Data Flow Controller with Back Pressure Mechanism

As opposed to the local shuffle service, Comet uses an external shuffle cluster to build an in-transit shuffle service. It is essential to build a data flow controller between the Spark computing cluster and the shuffle cluster.

The external shuffle cluster is composed of sets of shuffle instances. For each shuffle instance, there is a series of memory pools to receive shuffle data from map tasks, which belong to the previous stage. However, Comet flushes the shuffle data into HDFS, which may not as fast as receive speed. In some cases, it may cause the buffer overflow because of the slow flushing. Thus, Comet has to control the shuffle data generation in

the mapper side. The architecture of the data flow controller is shown in Figure 6.6. Once the associated shuffle instance is available to the mapper tasks, the tasks send the shuffle data to the shuffle instance's buffer pool. In other words, once the shuffle instance runs out of the buffer pool, the shuffle instance stops sending the data from the mapper tasks. With the data flow controller, Comet can avoid failures, which can be caused by a buffer overflow, and control the bandwidth usage of map tasks.

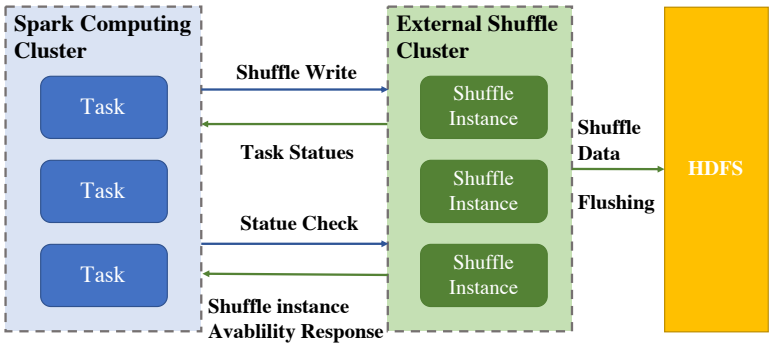


Figure 6.6: Data flow controller with back pressure mechanism

6.2.4 Destination Based Aggregation Mechanism

The existing local shuffle service is optimized for performance compared to the in-transit shuffle service. As discussed earlier, the distributed system's namespace performance is the primary performance bottleneck of storing intermediate data for data analytics. Further, since we add one more layer between Spark stages, we have extra cost compared to the existing local shuffle service.

Aggregation based shuffle service. To reduce the number of I/O requests to HDFS, one efficient way is aggregating data outside of HDFS instead of storing a large number of small files into HDFS. To do so, Comet aggregates the shuffle data based on shuffle data destinations, which is processed inside the same tasks in the next stage, as illustrated in Figure 6.7. Since HDFS, a block-based distributed file system, is not designed to store small files, Comet has to aggregate the small shuffle files into larger shuffle files before flushing them into HDFS.

External shuffle data aggregation. The main reason for Comet to use external aggregation instead of local mapper side aggregation is because of I/O overhead at the

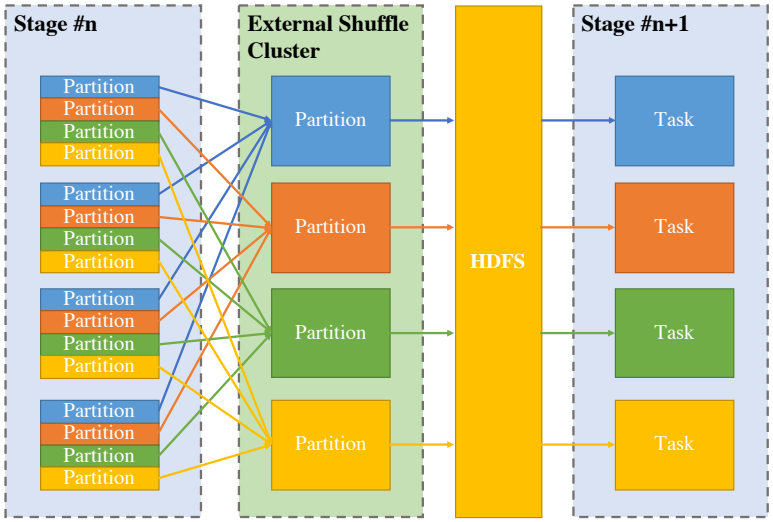


Figure 6.7: Destination based aggregation mechanism

mapper side. Since the I/O overuse could be one of the unstable points for large-scale data analytics (e.g., fetch error in Spark), I/O causes both hard and soft computing node failures. Further details are discussed in the following sections. Compared to existing solutions, e.g., Riffle [101], Comet does not add extra I/O burden to mapper side tasks, which decreases the I/O burden from the local disk and increases the stability of the computing cluster.

6.3 Evaluation Methodology

6.3.1 Testbed

We evaluate Comet in a real production environment at JD.com, which is a computing cluster with thousands of nodes. Each node is equipped with two CPUs with 24 cores leveraging hardware threads, 256 GB DDR4 DRAM, and is connected with 10 Gbps Ethernet links. Since Comet uses an external shuffle cluster to the server the shuffle service to the Spark cluster, we use a separate shuffle cluster for experiments, which has 20 nodes. Each node of the shuffle cluster features two CPUs E5-2640 with 32 cores leveraging hardware threads, 64 GB DDR3 DRAM, and are also connected with 10 Gbps Ethernet links.

We use the latest stable version Spark (i.e., version 2.4). Since the configuration

of production workloads are different from each other, we list the main Spark configurations of stability and performance evaluation in Tables 6.1 and 6.2. Compared to vanilla Spark, Comet has additional configurations as shown in Table 6.3. We use HDFS version 2.7 as the underlying distributed file system for Comet.

Benchmark	# of Executors	Tasks/Executor	Memory/Executor
1	800	3	35 GB
2	300	15	45 GB
3	600	8	32 GB
4	300	6	24 GB
5	500	5	20 GB

Table 6.1: Spark Configuration for Large-Scale Evaluation

Benchmark	# of Executors	Tasks/Executor	Memory/Executor
1	100	4	12 GB
2	100	4	12 GB
3	600	6	24 GB
4	1000	4	16 GB
5	600	6	24 GB

Table 6.2: Spark Configuration for Mid-scale Evaluation

# of Nodes	20
Threads/Node	96
Buffer/Node	12 GB
BatchSize	1 MB

Table 6.3: External shuffle cluster configuration

6.3.2 Workloads

To comprehensively evaluate Comet's stability and performance, we use TeraSort and representative workloads from the JD.com production environment. Since vanilla Spark always presents some failure with large-scale data analytics, we divide the evaluation into two parts, 1) Large-scale data analytics evaluation and 2) Medium-scale data analytics evaluation. For large-scale data analytics evaluation, we use large-scale data analytics, which is part of the production workload, to evaluate Comet's stability better. We also use TeraSort and three different production workloads, which have from tens of Gigabytes to several Terabytes of shuffle data, to evaluate Comet's performance.

Benchmark	Input data	Shuffle data	Output data
1	2.3 TB	49 TB	14.6 TB
2	47.7 GB	14.3 TB	153.2 GB
3	936.2 GB	75.9 TB	35.9 GB
4	5.5 TB	9.3 TB	1.6 TB
5	68.9 GB	23.5 TB	189 MB

Table 6.4: Large-Scale Workloads. 1) Recommendation Data Analysis #1, 2) Product Violation Detection, 3) Product Tracking System, 4) Recommendation Data Analysis #2, and 5) Business Flow Tracking

Benchmark	Input data	Shuffle data	Output data
1	50GB	27.4 GB	50GB
2	1TB	564.5GB	1 TB
3	227.3GB	5.5 TB	103.9 GB
4	193.3GB	1.4 TB	43.7 GB
5	92.5GB	2.1 TB	48 GB

Table 6.5: Medium-Scale Workloads. 1) TeraSort(50GB), 2) TeraSort(1TB), 3) Purchases and Sales Analysis #1, 4) Purchases and Sales Analysis #2, and 5) Core Data Warehouse Application

As shown in Tables 6.4 and 6.5, we select nine different workloads to evaluate the stability and performance of Comet. The tables also provide data volume and shuffle size of the workloads that are characterized as follows:

- **Recommendation data analysis #1 & #2.** Recommendation data analysis is a complex data analytics used to generate the recommendation information based on existing data. Product recommendation systems are data- and compute-intensive workloads with a large amount of shuffle data.

- **Product violation detection.** Product violation detection is also a complex data analytics used to detect the description violation of a product. The product detection service reads a small amount of input data (47.7 GB) while generating a large amount of shuffle data (14.3 TB), a typical data and compute-intensive workload.

- **Product tracking system.** The product tracking system tracks the flowing of products, a compute- and data-intensive data analytics.

- **Business flow tracking.** Business flow tracking provides the tacking service of

the original user. It reads a small amount of input data but generates a large volume of shuffle data, a compute- and data-intensive data analytics.

- **TeraSort.** TeraSort is a standard workload to evaluate the shuffle performance of data analytics. It sorts the input data to order data with the TeraSort algorithm. TeraSort is a compute- and data-intensive data analytics.

- **Purchases and sales analysis #1 & #2.** Purchases and sales analysis provides the insight analysis and visualization of purchases and sales. Purchases and sales analysis service is both compute-intensive and data-intensive data analytics.

- **Core data warehouse application.** Core data warehouse application is a typical ETL workload, which provides cleaned data to the downstream business. Core data warehouse application is a data-intensive data analytics.

6.4 Experimental Evaluation

This section analyzes the stability of Comet and evaluates its performance with large-scale and medium-scale data analytics. First, we compare the wall time of workloads between Comet and vanilla Spark. Second, we analyze in detail the execution time of stages within different workloads, which shows the details of the performance of shuffle writes and shuffle reads of Comet.

6.4.1 Comet's Stability with Large-Scale Data Analytics

We evaluate Comet's stability with large-scale data analytics, which has tens of Terabytes of shuffle data for each data analytics. As we present in Section 6.3.2, the workloads of stability are large-scale real data analytics from the production environment. To evaluate Comet's stability, we compare the task failures of Comet versus vanilla Spark. The statics of failures are provided in Figure 6.8.

The results indicate that the number of failures for Comet is much smaller than for vanilla Spark. Comet eliminates 99.9% and 92.8% for workloads #1 and #2, respectively. Moreover, Comet successfully finishes workloads #3, #4, and #5 without any

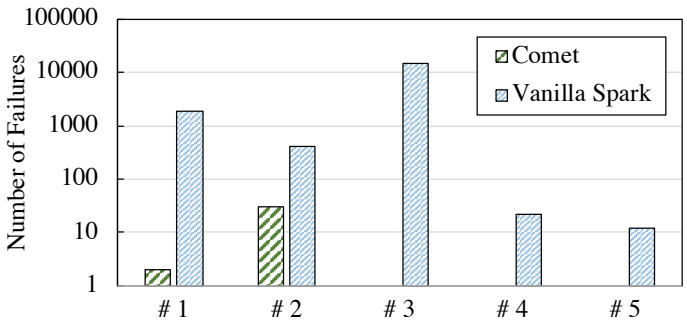


Figure 6.8: Number of failures of large-scale data analytics. 1) Recommendation Data Analysis #1, 2) Product Violation Detection, 3) Product Tracking System, 4) Recommendation Data Analysis #2, and 5) Business Flow Tracking

failures, while vanilla Spark can not complete work with more than fifteen thousand failures.

As we observed in the production environment, the main reason for failure is because of overuse of CPU and I/O bandwidth due to different reasons, including 1) executor connection closed, 2) node manager (YARN) connection close, and 3) fail to connect to node manager (YARN). The root cause of CPU and I/O bandwidth overuse [137] is caused by data transformation for intermediate data. Since Comet controls the shuffle data speed and offloads part of the I/O burden from the computing cluster, Spark can deal with a larger amount of data without many failures.

6.4.2 Comet's Performance

Figures 6.9 and˜6.10 show the performance evaluation results of large and medium-scale analytics. The results indicate that Comet speeds up large-scale workloads from 17% to 55%, while it speeds up the medium-scale workloads from 0% to 29%. Two factors affect the performance of data analytics 1) task retry with failures, and 2) I/O performance of shuffling.

The main performance factor of large-scale workloads is the overhead of tasking reruns. As we discussed in Section 6.2, Spark ensures fault tolerance with the DAG engine, which recomputes the lost data with the dependency graph. As a result, although some soft failures would not kill the job, they cause significant performance degradation because of instability.

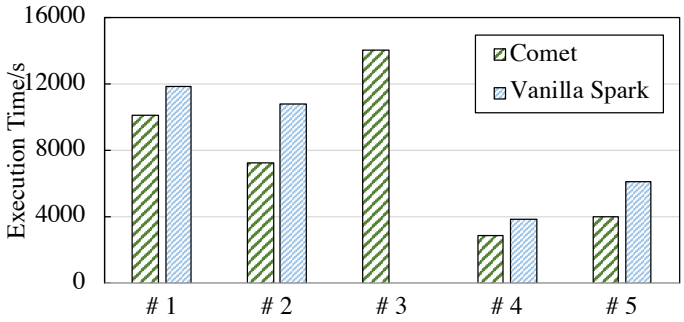


Figure 6.9: Execution time of large-scale workloads. 1) Recommendation Data Analysis #1, 2) Product Violation Detection, 3) Product Tracking System, 4) Recommendation Data Analysis #2, and 5) Business Flow Tracking

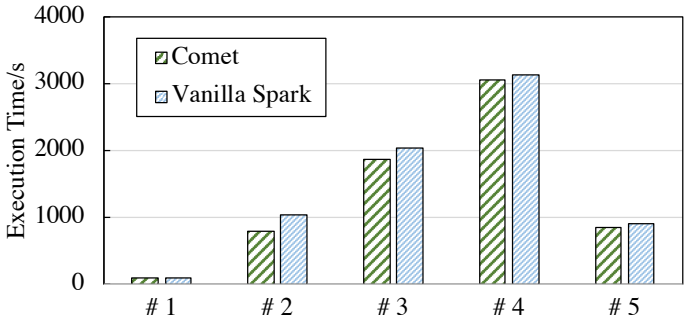
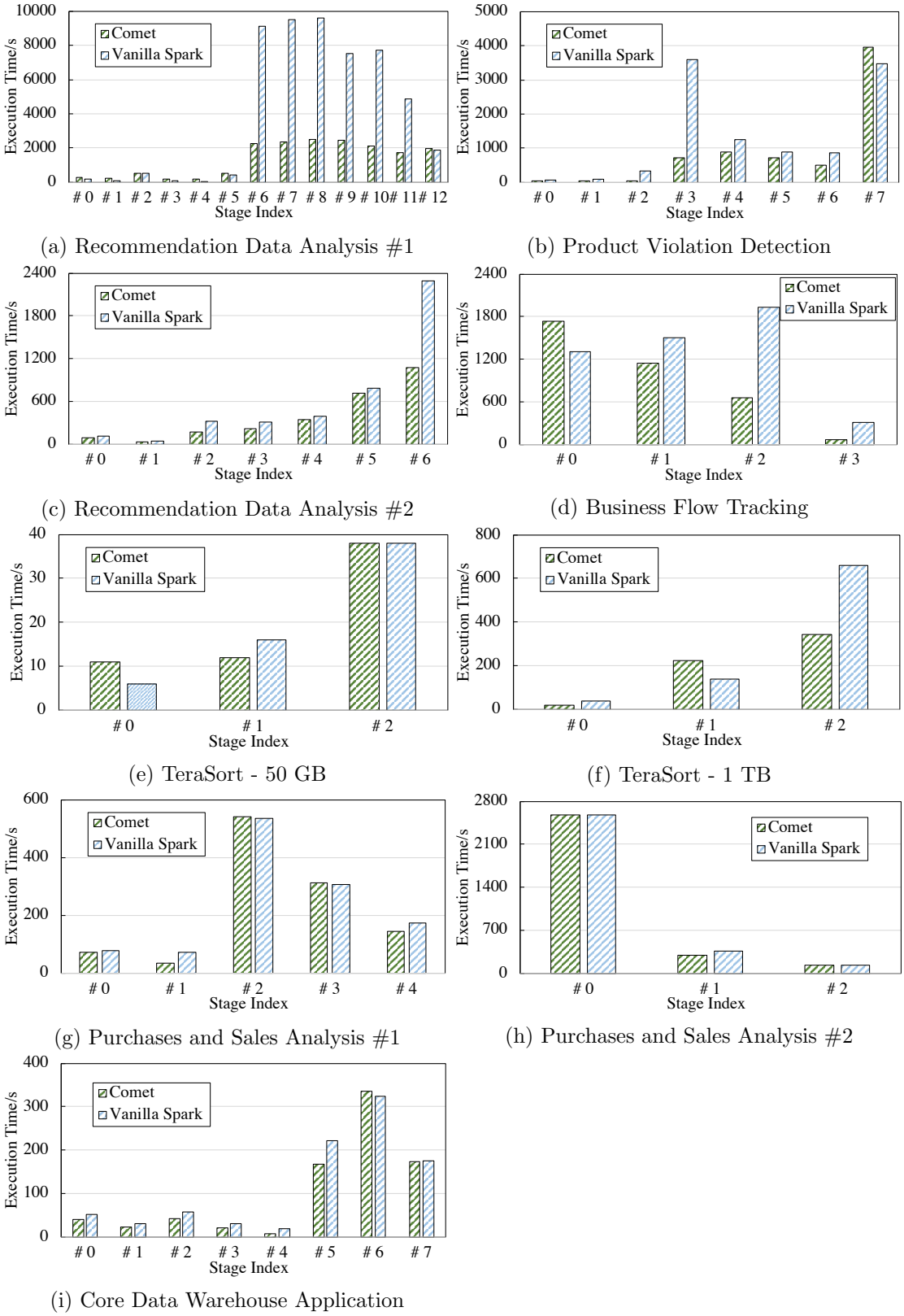


Figure 6.10: Execution time of medium-scale workloads. 1) TeraSort(50GB), 2) TeraSort(1TB), 3) Purchases and Sales Analysis #1, 4) Purchases and Sales Analysis #2, and 5) Core Data Warehouse Application

For medium-scale data analytics, Comet has comparable performance as vanilla Spark. However, we can not conclude on shuffling performance only with overall performance. Thus, we describe the shuffling performance in the next part with Spark stages analysis, individually.

Spark stages performance analysis. Since Spark divides the stages with shuffle operation, we classify the stages into four cases, 1) shuffle write-only, 2) shuffle read-only, 3) shuffle write and shuffle read, and 4) no shuffling. The evaluation of Comet focuses on the first three cases. The detail of the statistics are provided in Table 6.6.

Figure 6.11 shows the main stage performance results of different data analytics. Since the product tracking system failed with vanilla Spark, we removed it from stage analysis. As discussed earlier, Comet is an in-transit shuffle service, which connects two stages in Spark with data shuffling. Thus, Spark transmits shuffle data step by step, 1) shuffle write, and then 2) shuffle read. During the shuffle write phase, mapper



(a) Recommendation Data Analysis #1

(b) Product Violation Detection

(c) Recommendation Data Analysis #2

(d) Business Flow Tracking

(e) TeraSort - 50 GB

(f) TeraSort - 1 TB

(g) Purchases and Sales Analysis #1

(h) Purchases and Sales Analysis #2

(i) Core Data Warehouse Application

Figure 6.11: Execution time of main stages Comet versus vanilla Spark.

Benchmark	Shuffle Write	Shuffle Read
Large # 1	0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12	1, 3, 4, 5, 6, 8, 9, 10, 11
Large # 2	0, 1, 2, 3, 4, 5, 6	2, 3, 4, 5, 6, 7
Large # 4	0, 1, 2, 3, 4, 5	1, 3, 6
Large # 5	0, 1, 2	1, 2, 3
Medium # 1	1	2
Medium # 2	1	2
Medium # 3	0, 1, 2, 3	1, 2, 3, 4
Medium # 4	0, 1	1, 2
Medium # 5	0, 1, 2, 4, 5	2, 3, 5, 6, 7

Table 6.6: Shuffle write and read labels of the workloads

tasks write the shuffle data into shuffle instances, which is outside of the computing cluster. After all of the shuffle writes finish, reduce tasks read the shuffle data from remote shuffle instances.

Performance of shuffle write and shuffle read in Comet. The results show that Comet provides the same performance or slightly reduced lower performance for tasks that only have shuffle write. These results are better than expected because of the asynchronous data transmission between the computing cluster and the shuffle cluster. Although Comet has extra network transmission compared to the local shuffle service, the transmission latency is hidden by data skew tasks and several iterations of tasks running with asynchronous data transmission. Thus, the pure shuffle writes time for Comet is comparable to vanilla Spark. However, since Comet aggregates shuffle data based on reducers' destination, Comet has better performance than vanilla Spark in the shuffle read phase.

Performance degradation because of failures. The primary performance degradation of large-scale data analytics is task failures. The high overhead of Spark task failures in the production environment is not only because of recomputing overheads but also because of the overheads associated with re-scheduling overheads. To save computing resources, Spark uses a dynamic allocation mechanism, which releases the idle executors. However, Spark takes extra time to apply computing resources from the YARN scheduler when it retries to run the failed stages, which significantly decreases Spark's performance.

6.5 Discussion

In this chapter, we presented Comet, an in-transit shuffle service, to improve large-scale data analytics's stability and performance. Comet enhances the stability of Spark by offloading CPU and the I/O burden to the external shuffle cluster. Comet also enhances the fault tolerance of shuffle data storing with a distributed file system instead of a local disk, which does not have single-point failures. Furthermore, with better stability and a destination-based aggregation mechanism, Comet can achieve up to 55% of speed up compared to vanilla Spark.

Chapter 7

Conclusion and Future Work

The cloud services running at extreme scales on modern cloud infrastructure are composed of several cloud applications that need to access the data orchestration frequently with a large volume of data at run time. With the increasing scale and complexity of cloud data analytics, there is a large amount of data exchanged in different nodes, clusters, even different data centers. Because of the cloud environment's complexity, the existing data orchestration becomes the performance bottleneck of cloud data analytics based on cloud frameworks. In the meantime, the fast revolution of hardware technologies and cloud architectures proposes new opportunities to build efficient and high-performance data orchestration for cloud frameworks in large-scale cloud environments. However, there are still challenges in designing and implementing efficient and high-performance data orchestration for cloud frameworks under different cloud environments.

This thesis identified and addressed the critical problems and requirements to build efficient and high-performance data orchestration for cloud frameworks in a large-scale cloud environment. First, this thesis addresses the critical performance gap between modern cloud frameworks and advanced storage devices. It formulates the performance of cloud frameworks with different bandwidth with different storage devices. It also identifies the performance gap between existing cloud frameworks and advanced storage devices because of the cost of serialization and de-serialization of data. Second, this thesis explores the potential hybrid cloud architecture for cloud Frameworks running in geo-distributed data centers with fast fabric interconnection. It verifies it is feasible to harvest spare computing resources across geo-distributed data centers with fast fabric interconnection. Third, this thesis presents the design and implementation

of a disaggregated memory system with persistent memory for in-memory cloud frameworks. By leveraging shuffle and persistence optimization, it can effectively reduce the execution time for shuffle-intensive data analytics, incorporating shuffle and persistence mechanisms into big data frameworks using an in-memory distributed file system. The experimental evaluation results from empirical executions show that remote large-volume persistent memory and a disaggregated memory pool can also increase the overall memory capacity with much lower overheads. Finally, this thesis proposes an in-transit shuffle service to improve large-scale data analytics's stability and performance. It enhances Map-Reduce based cloud compute frameworks' stability by offloading CPU and the I/O burden to the external shuffle cluster. It also enhances the fault tolerance of shuffle data storing with a distributed file system instead of a local disk, which does not have single-point failures.

In the future, this thesis can be extended in several directions, including:

- **Explore the energy-efficient data orchestration:** The existing data orchestration in the cloud environment is mainly designed and implemented for performance and cost-efficient. One of the potential future research is to explore the data management across computing clusters for energy efficiency. This research can be extended to develop energy-efficient data orchestration in various cloud architectures.

- **Explore the fast and extensible namespace for Cloud storage and Structure data service:** One of the essential cloud service features is the almost infinite scalability of the namespace, but it also introduces a list of challenges to handle metastore operations efficiently. This research can be extended to develop scalable and high-performance data orchestration with scalable and high-performance name space policy.

- **Explore the potential of caching policy in the hybrid cloud environment:** The caching policy is one of the trends to build efficient and high-performance data orchestration in the hybrid cloud environment. It can save a large amount of network transmission across computing clusters and data centers at a lower

cost. This research can be extended to build the high-performance caching data orchestration for cloud compute frameworks in the hybrid cloud environment.

Appendix A

Understanding Behavior Trends of Big Data Frameworks

This chapter complements the thesis's main contributions by providing an understanding of big data processing systems behavior and the tradeoffs associated with the use of different architectural designs and processing frameworks for different classes of relevant applications under different constraints. It provides the foundations to develop models that can fundamentally enable Big Data analytics on ongoing cyber-infrastructure based on software-defined infrastructure (SDI). As opposed to other research efforts that investigate balanced systems for a range of analytics applications [14], it aims to understand the optimal design choices, given a multi-criteria approach and under different constraints (e.g., power budget). This work is focused on these behaviors and tradeoffs for two of the leading distributed processing systems for Big Data analytics: Apache Hadoop and Spark, both of which are currently the most widely used open-source parallel processing frameworks for Big Data analytics.

A.1 Tradeoffs in Big Data Systems

In order to construct models to understand and explore the design space of big data systems, it is required to characterize a comprehensive set of data-centric benchmark applications in terms of performance, energy, and power behaviors on an instrumented platform to understand their resource requirements best and identify possible performance/power tradeoffs.

This characterization can be useful, on the one hand, to build models and develop heuristics/meta-heuristics that will consider different criteria and constraints, including but not limited to: performance (e.g., response time or quality of service), capital costs (e.g., the infrastructure available, such as the number of servers, cores, or memory),

operational costs (e.g., energy consumption), and the power budget. Such an approach is expected to be multi-dimensional and multi-criteria. Some example parameters are (1) hardware choices (e.g., core count, memory size, I/O, and network bandwidth), (2) data processing system (e.g., batch vs. micro-batch), (3) virtualization (e.g., bare-metal vs. containers vs. VMs), (4) processing framework (e.g., Hadoop vs. Spark), and (5) programming language (e.g., Scala vs. Python). On the other hand, it can help develop resource provisioning and scheduling approaches for big data workloads in systems based on ongoing software-defined infrastructure. In this scenario, the problem becomes more challenging as it spans across different dimensions (multi-dimensional knapsack-like problem, i.e., NP-hard).

This chapter focuses on understanding the behaviors and tradeoffs of the two primary open-source processing frameworks for Big Data analytics: Hadoop and Spark. Data movement is one of the main bottlenecks for these data processing frameworks, as their applications typically require heavy read and write operations during processing. While Hadoop supports the MapReduce programming model using a storage-centric approach, Spark is based on in-memory processing (through Resilient Distributed Datasets - RDDs) and requires much less access to storage. These two processing frameworks are shown in Figure A.1, which provides an overall classification of some of the most widely used open-source distributed data processing frameworks.

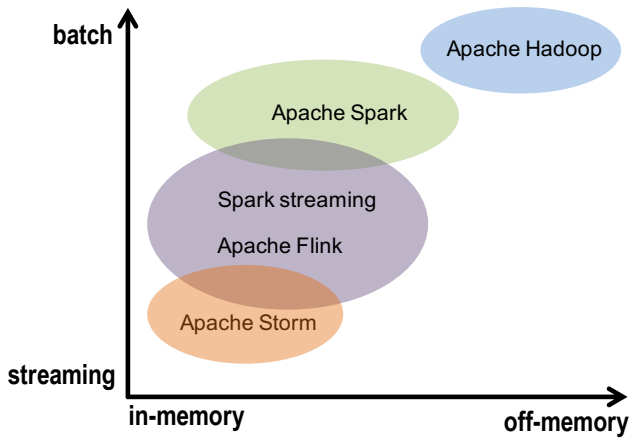


Figure A.1: Classification of the most extended (Apache-based) distributed processing back-ends for big data analytics

The characterization considers two fundamental parameters: (i) energy/power tradeoffs, and (ii) storage technology (i.e., memory hierarchy). While in-memory processing systems are expected to be faster than storage-based systems for running a Big Data processing workload, the power required to run this workload in-memory is expected to be higher if a larger pool of resources are needed to handle in-memory data. There are also clear issues related to power requirements for a more I/O-bounded approach due to lower CPU and memory utilization over time.

The tradeoff between required power and energy consumption in this context requires investigation. For example, in the scenario depicted in Figure A.2, the energy cost of running a workload using Hadoop could be higher than using Spark; however, the fastest option (i.e., Spark) might not be viable due to the power budget constraints or availability of servers needed to handle RDDs in memory. The figure also shows that power capping can be used as a mechanism to manage these possible tradeoffs. Power capping has also been considered to better understand the possible tradeoffs between Hadoop and Spark. Since the memory hierarchy/storage technology is expected to significantly impact performance and other metrics, Section A.3 investigates different storage hierarchies (ranging from hard disk to PCIe-based non-volatile memory devices) and power capping using RAPL. However, the use of power capping strategies in software-defined infrastructures remains a potential direction for future work.

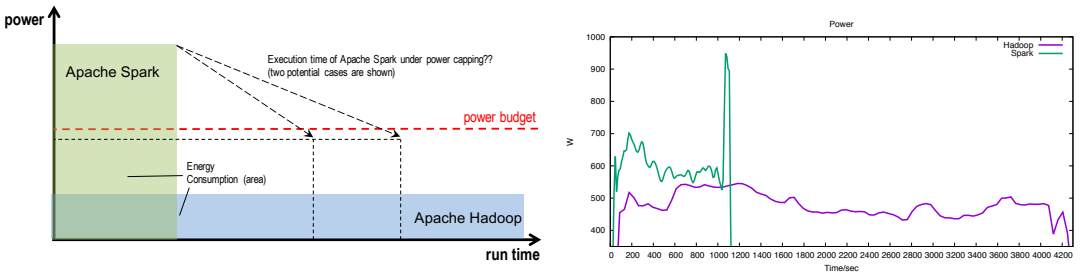


Figure A.2: Possible (top) and observed (bottom) run time and power consumption behavior of a data analytics workload run with Hadoop and Spark. The real execution of the bottom is obtained using Grep (see Section A.2 for more details)

A.2 Evaluation Methodology

While the evaluation focused on software-defined infrastructures in systems with non-volatile memory technology is based on simulations, the empirical executions were conducted on the NSF-funded research instrument "Computational and dAta Platform for Energy efficiency Research" (CAPER) described in section 3.2.1. In addition to server level power measurement mechanisms, it supports RAPL (Running Average Power Limit)-based metering to provide CPU-centric power measurements at a sampling rate at processor level up to 20Hz. RAPL also provides power capping capabilities by setting power limitations on the processor package or DRAM.

We configured the big data processing frameworks as a baseline using commonly used and balanced configurations without optimizations (e.g., it doesn't feature DC/OS layer such as Apache Mesos). The specific characteristics of the system configuration are described as follows. (1) Hadoop version 2.7.1 was deployed using YARN. One server was configured as NameNode and seven servers as DataNodes for the HDFS file system. HDFS uses 128MB blocks with 3 replicas for each block. Hadoop was configured to run 32 containers per node, and there is at least 2GB memory for each container. The memory of JVM heap size, Map Task and Reduce Task are set to 4GB, per task, and (2) Spark version 1.5.1 was deployed using YARN. Like Hadoop, one server was configured as NameNode and seven servers as DataNode for the HDFS file system. One server as Master and seven servers as Slaves were configured. For each Spark application 7 executors were configured (i.e., one per node), using 8 cores each. The JVM memory was set to 20GB and 64GB for Spark Driver and Executor, respectively.

A comprehensive set of representative workloads were selected, including Grep, K-Means, and WordCount for both Hadoop and Spark. TeraSort, PageRank, and Connected Components were used for Spark to understand and characterize Spark behaviors in more detail. Tables A.1 and A.2 show the workloads that we used with their typical characteristics and utilized data sets, respectively. Grep and WordCount are data intensive and one-pass-type workloads. K-Means is typically compute intensive and an iterative workload. In order to further investigate the impact of storage technologies

Table A.1: Hadoop and Spark Workloads

Workload	Description	Type
Grep	extracts matching strings from text files and counts how many time they occurred	IO-bound, one pass
Word Count	reads text files and counts how often words occur	IO-bound, one pass
K-Means	K-Means classifier	CPU-bound, iterative
Terasort	samples the input data and uses map/reduce to sort the data into a total order	Network-bound
PageRank	measures the importance of each vertex in a graph	CPU-bound, iterative
Connected Components	labels each connected component of the graph with the ID of its lowest-numbered vertex	CPU-bound, iterative

Table A.2: Hadoop and Spark Datasets

Input Dataset	Workload
PUMA Wikipedia	Grep, Word Count
Friendster social network	PageRank, Connected Components (65×10^6 Nodes, 1.8×10^9 Edges)
Hadoop TeraGen	TeraSort
BigDataBench K-Means	K-Means

in Spark, Terasort, PageRank, and Connected Components were selected. Different metrics were collected for each of the workloads, including energy consumption, power requirements, execution time, and resource utilization (e.g., CPU utilization, RAM memory pressure - via LLC miss rate, and I/O throughout).

A.3 Experimental Results

A.3.1 Characterizing Behavior Patterns of Big Data Frameworks

This sub-section first explores and discusses how different Big Data processing frameworks impact performance, power, energy, and resource utilization using Grep, K-Means, and WordCount workloads. Hadoop and Spark workloads are both configured to run using HDD as the storage device for the HDFS setup, which is its baseline and standard configuration.

Figure A.3 (top) presents the energy consumption of Grep, K-Means, and WordCount for both Hadoop and Spark. The results show that executions with Hadoop consume about 3.2 times, 3.1 times, and 2.2 times more energy than Spark for Grep, K-Means, and WordCount, respectively. Figure A.3 (bottom) shows that the execution time of Grep, K-Means, and WordCount using Hadoop is 5%, 2.4 times, and 23% longer than using Spark, respectively, which indicates that executions with Spark consume less

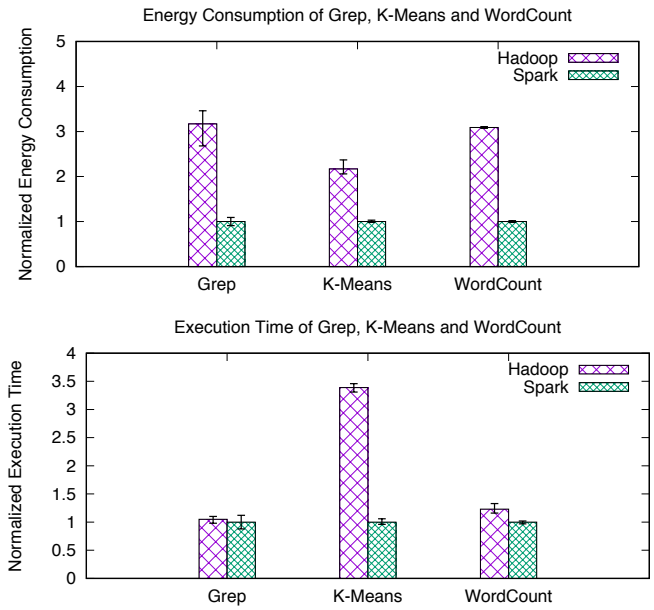


Figure A.3: Normalized energy consumption (top) and normalized execution time (bottom) of Grep, K-Means and WordCount using Hadoop and Spark

Table A.3: CPU utilization and power consumption of Grep, K-Means, and WordCount execution using Hadoop and Spark

Workload	Framework	AVG Power (W)	Max Power (W)	CPU Util AVG (%)
Grep	Hadoop	667	1,031	61
	Spark	222	979	22
K-Means	Hadoop	625	1,346	40
	Spark	687	938	37
WordCount	Hadoop	1,015	1,261	75
	Spark	573	1,221	48

energy than with Hadoop - and not only because Spark executions are shorter. The results also show that the I/O throughput with Spark is 7% and 15% higher than Hadoop for Grep and WordCount, on average. Since Grep and WordCount are one-pass-type workloads, Hadoop and Spark have similar sizes of shuffle data, where Hadoop spends more time waiting for data reading and writing when compared to Spark.

As shown in Table A.3, the CPU utilization during the execution of Grep, K-Means, and WordCount using Hadoop is longer (e.g., up to 1.8 times for Grep) than the execution of these benchmark applications using Spark. The results indicate that both execution time and CPU utilization with Hadoop are higher than with Spark; as a result, Spark is more energy efficient than Hadoop. Note that these quantitative results are with a system configuration using HDD and 1G network connectivity.

Figures A.4 and A.5 show that CPU utilization and power consumption using Spark is much lower than using Hadoop; however, the I/O throughput using Spark is higher than using Hadoop. This behavior suggests that Spark is capable of delivering higher efficiencies when running workloads than Hadoop under the same constraints and system configuration. The results provided in the following sections will show that Spark provides higher resource utilization efficiencies with other storage and network configurations, and therefore, higher overall efficiency, which is consistent with existing literature.

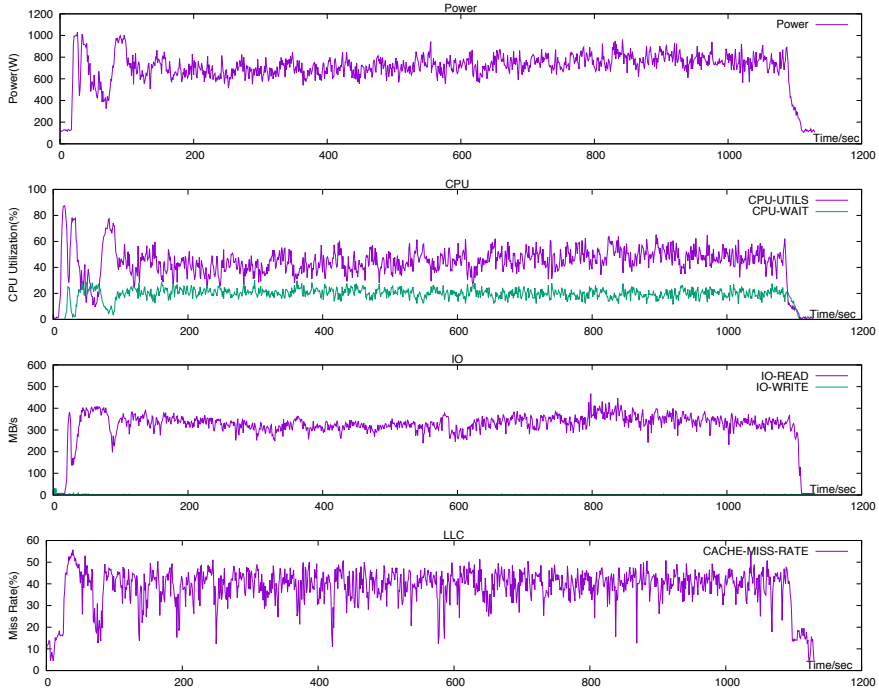


Figure A.4: Resource utilization and power consumption of Grep using Hadoop

Understanding the Impact of Storage Technology. Figures A.6 and A.7 show that energy consumption for executions using NVRAM is lower than when using HDD or SSD for all application workloads with both Hadoop and Spark, as expected. The energy consumption of Grep executions using HDD is higher compared to executions using an SSD or NVRAM, especially for Spark (i.e., 40% and 2 times higher energy consumption with Hadoop and Spark, respectively). However, the difference in execution time across storage technology configurations is much higher than the difference in energy consumption (e.g., 1.28 times longer execution time using Hadoop and HDD with

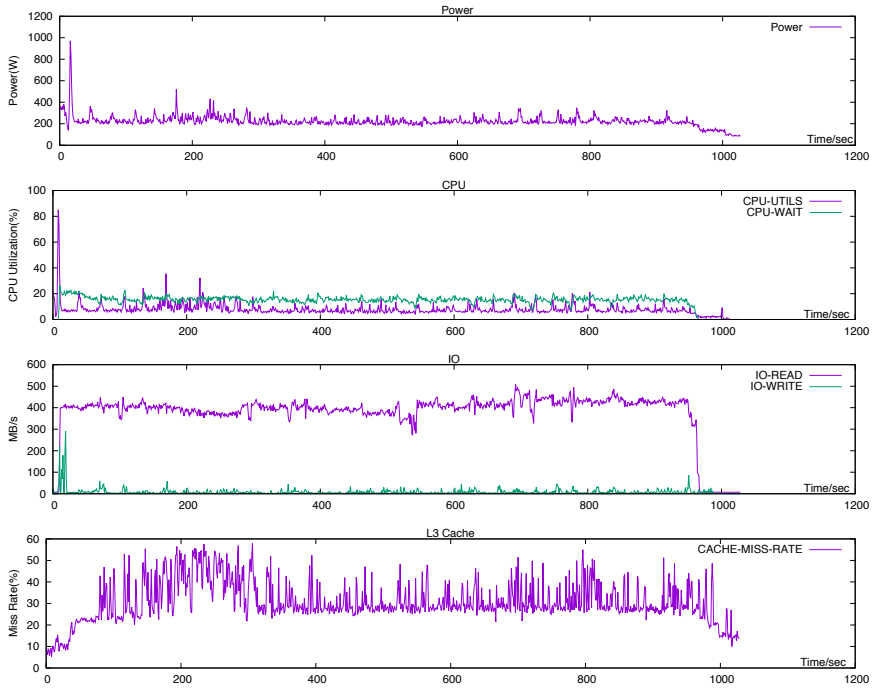


Figure A.5: Resource utilization and power consumption of Grep using Spark

respect to NVRAM vs. 40% increased energy consumption). Overall, the difference between execution time and energy increase is significantly higher with Spark. Overall, the CPU wait percentage is higher using Spark (i.e., 65.3%, 32.6%, and 6.8% for HDD, SSD, and NVRAM). These results indicate that using NVRAM reduces CPU wait time for both Hadoop and Spark; consequently, it provides higher energy efficiency.

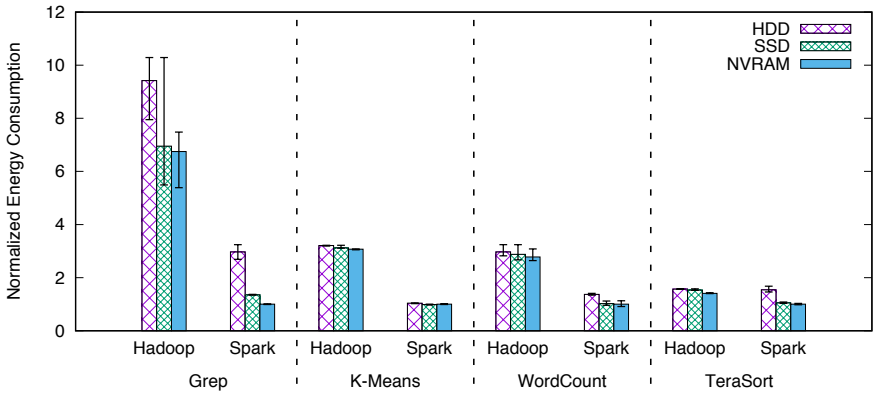


Figure A.6: Energy Consumption of Grep, Kmeans, WordCount and Terasort using HDD, SSD and NVRAM with Hadoop and Spark

Since K-Means is iterative and not an I/O-bound workload, its executions using the

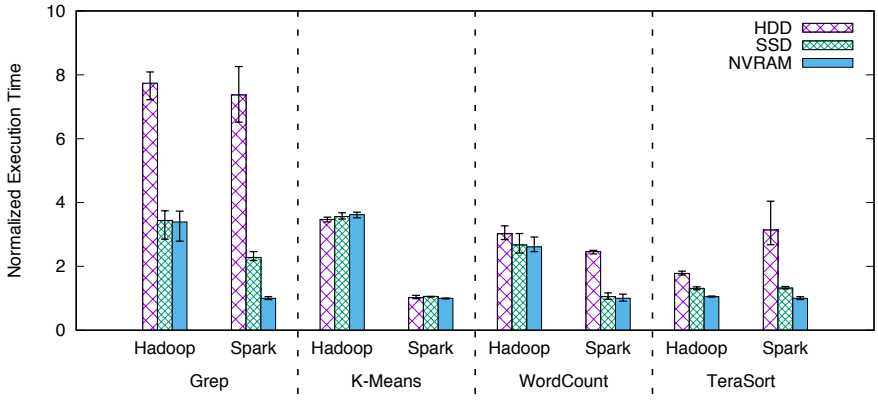


Figure A.7: Execution Time of Grep, Kmeans, WordCount and Terasort using HDD, SSD and NVRAM with Hadoop and Spark

different storage choices are similar in terms of execution time and energy. However, executions using Spark are much shorter and consume much less energy than executions using Hadoop. Another factor in Spark is RDD caching. Specifically, the first iteration of K-Means scans all data into RDD caching, which means the calculation of subsequent iterations are based on the cached RDD. Since Spark reads are from RDDs, K-Means is not constrained by I/O throughout using Spark.

Figures A.6 and A.7 show that WordCount executions with HDD are 15% longer and consume 7% more energy than executions with NVRAM using Hadoop. However, the execution time and energy consumption of WordCount executions with SSD and NVRAM are similar. In the case of Spark, WordCount executions using HDD are significantly longer (up to 1.46 times) and consume more energy than executions using an SSD and NVRAM. As a result, the power required for executions using HDD are approximately half of the power required in executions using an SSD and NVRAM.

Figures A.6 and A.7 show that TeraSort executions using HDD and SSD consume significantly more energy than executions using NVRAM, especially with Spark (up to 68.3%). Spark does not support a simultaneous read and write function, therefore, if the storage and/or network are not fast enough, the shuffling phase will consume a lot of time. As TeraSort is an I/O-bounded workload, it has a heavy shuffle phase. Consequently, Hadoop provides similar or superior performance than Spark in executions using HDD. Figures A.8 and A.9 show TeraSort behavior patterns with Hadoop and

Spark using NVRAM. The figures clearly show different CPU and I/O patterns, which result in different power consumption profiles.

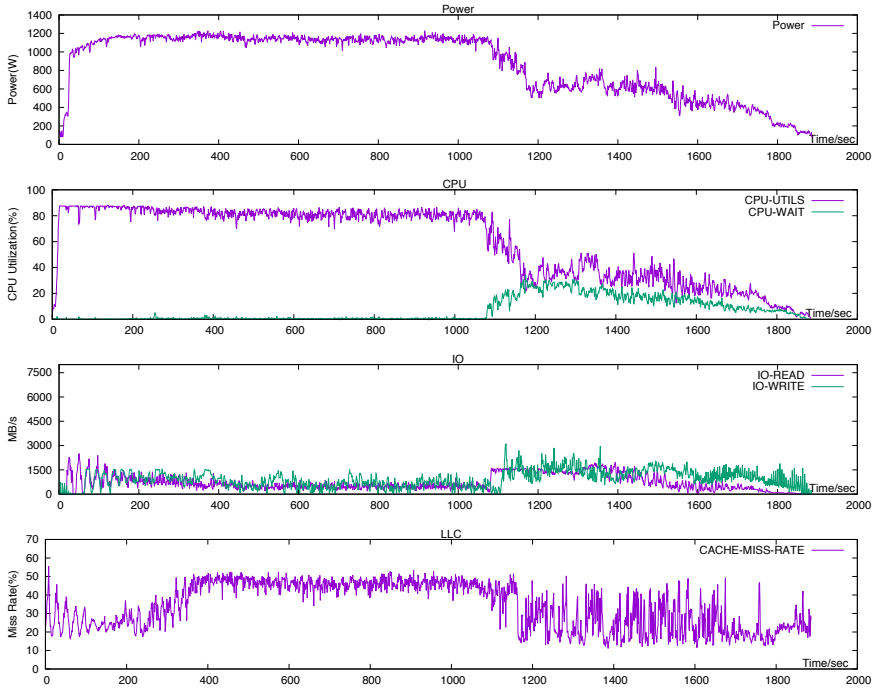


Figure A.8: Resource utilization and power consumption of TeraSort with Hadoop using NVRAM

The results discussed above show tradeoff between power budget, execution time and energy consumption and indicate that, overall, Spark provides higher performance and lower energy consumption. The rest of this sub-section concentrates on further understanding the impact of the different storage choices using the following three Spark workloads:

TeraSort is a popular sorting workload for benchmarking Big Data frameworks. TeraSort executions are heavily influenced by the storage technology. As TeraSort is both I/O- and CPU-bounded, the CPU wait percentage is lower with superior storage technologies (i.e., NVRAM).

PageRank is a graph algorithm proposed by Google to rank web pages by the number and quality of links to a page. Five iterations of the workload were used in each execution. In contrast to TeraSort, Table A.4 shows that the CPU wait percentage is similar and almost null for the different storage technologies. However, the energy

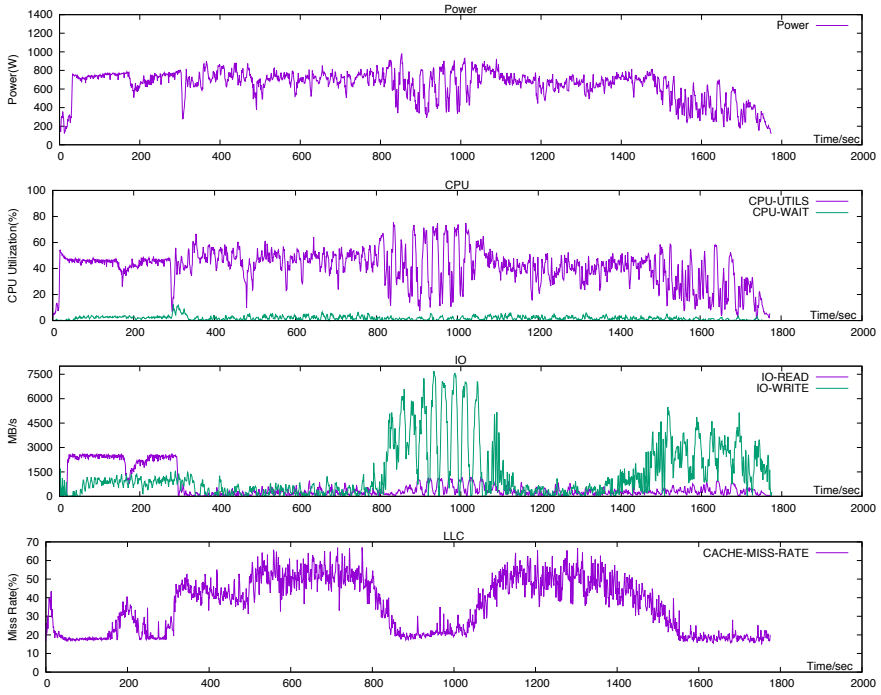


Figure A.9: Resource utilization and power consumption of TeraSort with Spark using NVRAM

Table A.4: Resource utilization of PageRank execution with Spark using HDD, SSD and NVRAM

Storage	Energy (KJ)	Time (s)	AVG Power (W)	Max Power (W)	CPU-Util (%)	CPU-Wait (%)	IO (MB/s)
HDD	715	2,921	321	759	99.5	0.5	22.0
SSD	628	2,212	284	747	99.6	0.4	25.0
NVRAM	657	2,137	308	830	99.6	0.5	31.6

consumption is most efficient using NVRAM (6.8% and 4.6% lower than HDD or SSD, respectively).

Connected Components is also an iterative graph processing workload. It computes the connected component of each vertex and returns a graph with the vertex value containing the lowest vertex ID in the connected component containing that vertex. The results shown in Table A.5 indicate that Connected Components and PageRank have very similar behavior patterns.

Understanding the Impact of the Network. This experiment focuses on how the network bandwidth impacts performance, power, energy, and resource utilization with Hadoop and Spark using Grep, WordCount, K-Means, TeraSort, PageRank, and

Table A.5: Resource utilization of Connected Components execution with Spark using HDD, SSD and NVRAM

Storage	Energy (KJ)	Time (s)	AVG Power (W)	Max Power (W)	CPU-Util (%)	CPU-Wait (%)	IO (MB/s)
HDD	893	3,587	256	760	99.6	0.4	12.8
SSD	888	3,558	249	739	99.6	0.4	17.2
NVRAM	839	2,952	287	755	99.6	0.4	24.9

Connected Components (CC). The network is configured to use either 1Gb Ethernet or 10Gb Ethernet interfaces. Figure A.10 shows that behavior patterns are workload-dependent and that in general, the energy consumption is higher (or similar) for executions using the 10G network compared to the energy consumption of executions using the 1G network with both Hadoop and Spark; however, the execution time using the 10G network is shorter (or similar) than executions using the 1G network.

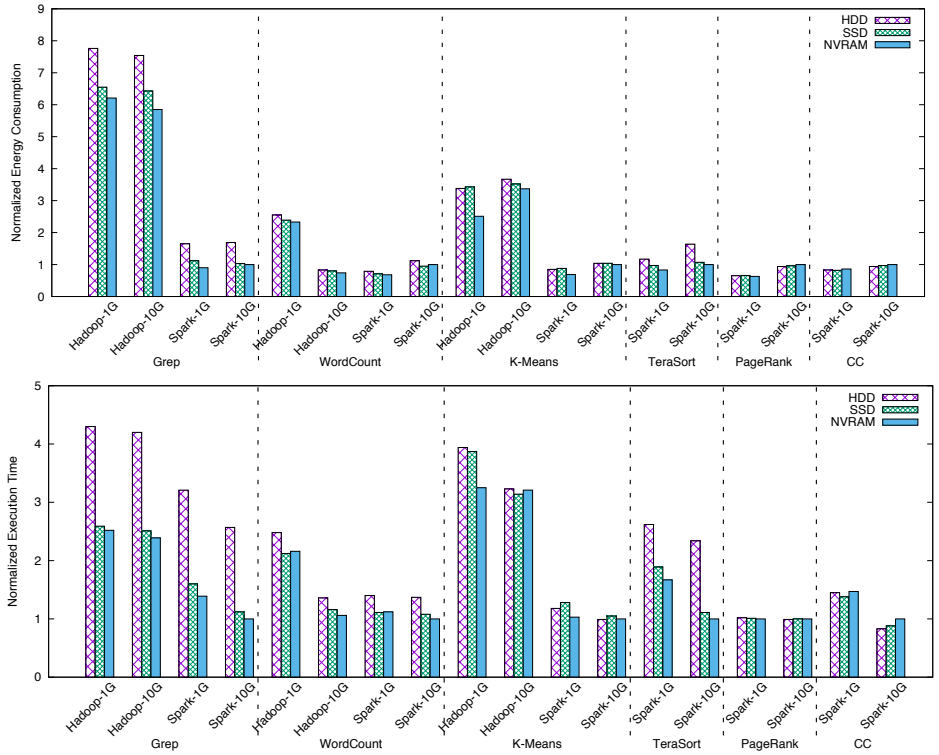


Figure A.10: Normalized energy consumption (top) and execution time (bottom) of Grep, WordCount, K-Means, TeraSort, PageRank and Connected Components using HDD, SSD and NVRAM with Hadoop and Spark

The results discussed above are consistent with the expected behaviors; however they provide an understanding of different design choices based on different workload

profiles and optimization goals (e.g., performance, power, and cost), which we use for understanding the potential of software-defined infrastructure in the context of big data processing frameworks. For example, using a 10G network is worthwhile for Spark when high performance is needed and neither power nor budget are constrained; however, when performance degradation can be tolerated and power is not heavily constrained, 10G is not worth the cost when using Hadoop. This example represents a class of data-intensive one-pass workloads that can be heavily influenced by the storage technology used.

A.3.2 Exploring the Potential of Software-Defined Infrastructure

In this sub-section we explore the potential of software-defined infrastructure for existing big data processing frameworks through simulation using the information obtained in the characterization presented above. We have developed a simulation framework focused big data workload allocation to resources. These workloads are composed of big data applications for both Hadoop (Grep, WordCount, TeraSort and K-Means) and Spark (Grep, WordCount, TeraSort, K-Means, Connected Component and PageRank) frameworks. As the storage technology used in the big data frameworks running these applications significantly impact different key metrics such as execution time and energy consumption, the workload allocation algorithm focuses on storage issues in the system design.

In order to simulate the execution of the workloads in software-define infrastructure and traditional infrastructures (i.e., with fixed amount of accessible storage resources), we: (1) assume that the datacenter is composed of multiple big data deployments (clusters) for running big data workloads, and (2) fix the total amount of storage available in the datacenter (i.e., total amount of HDD, SSD and NVRAM). Under these assumptions, we consider two scenarios:

- **Non-SDI** (traditional): The storage available in one cluster is fixed and can be used only by applications running in that cluster.
- **SDI**: All storage available in the datacenter is available to all applications running

in any cluster.

In this initial approximation, we also assume that the latency and bandwidth to off-node and in-node storage devices are similar. Our ongoing work includes the exploration of the design space (e.g., interconnect capabilities required for realizing effective software-defined infrastructure) by introducing different latency and bandwidth limitations to off-node storage device access.

Our simulation study requires key data points, such as the workloads' execution time and energy consumption using different storage device technology. The meaning of the parameters used in the simulation are described as follows:

- W: Randomly generated workload with 100 application instances
- S_{WL}: Workload required storage capacity
- $S_{HDD}, S_{SSD}, S_{NVRAM}$: Available HDD, SDD and NVRAM capacity, respectively
- $T_{HDD}, T_{SSD}, T_{NVRAM}$: Workload execution time using HDD, SSD and NVRAM, respectively
- $E_{HDD}, E_{SSD}, E_{NVRAM}$: Energy consumption using HDD, SSD and NVRAM, respectively
- $C_{HDD}, C_{SSD}, C_{NVRAM}$: Energy consumption using HDD, SSD and NVRAM devices, respectively
- $T/E/C_{SDI}, T/E/C_{non-SDI}$: Execution Time, Energy Consumption and Cost, using SDI and non-SDI configurations, respectively

In the simulations, the datacenter contains 10 clusters, each composed of 8 nodes, which is the configuration used in the characterization presented above. The total size of HDD, SSD and NVRAM for the overall datacenter are set to 37 TB, 10.8 TB and 6-48 TB, respectively. The cost (C) refers to the capital cost of different technologies (e.g., NVRAM vs. HDD), which is part of TCO (Total Cost of Ownership). Default values are based on standard pricing for enterprise storage at \$0.4882/GB, \$0.5859/GB

and $1.0417/GB for HDD, SSD and NVRAM, respectively. Algorithm 2 presents the workload allocation algorithm, which by default prioritizes NVRAM as the first choice, the second choice is SSD and last choice is HDD. We follow this approach to understand the tradeoff between response time and energy efficiency and cost, which is a key issue in datacenter design and deployment.

Algorithm 2: Workloads Allocation Algorithm.

```
1 Function Storage Device Priority ;
  Input  : W, S_WL,
           S_HDD, S_SSD, S_NVRAM, T_HDD, T_SSD, T_NVRAM,
           E_HDD, E_SSD, E_NVRAM, C_HDD, C_SSD, C_NVRAM
  Output: T_SDI, E_SDI, C_SDI, T_non-SDI, E_non-SDI, C_non-SDI
   1: start time;
   2: while W is not empty do
   3:    for i = 1; i <= 10; i++ do
   4:       if cluster i is empty then
   5:          if S_NVRAM >= S_WL then
   6:             push next workload into NVRAM;
   7:          else if S_SSD >= S_WL then
   8:             push next workload into SSD;
   9:          else
  10:             push next workload into HDD;
  11:          end if
  12:       end if
  13:    end for
  14: end while
  15: end time;
  return (T_SDI, E_SDI, C_SDI, T_non-SDI, E_non-SDI, C_non-SDI);
```

Figure A.11 shows the tradeoff between cost and execution time and energy consumption using different NVRAM size (i.e., different investment choices) using non-SDI and SDI scenarios. The results show that both execution time and energy consumption in the SDI scenario are significantly lower than the non-SDI scenario with up to 36TB of NVRAM. While larger NVRAM sizes provides better performance/energy (up to 24-30TB) in the SDI scenario, these improvements are at a significant cost increase.

As opposed to CPU-bound workloads, data-intensive workloads are highly impacted by the storage technology used. In order to understand this issue in SDI and non-SDI scenarios, we classify the simulated applications into two types: (1) Data intensive (Grep, WordCount and TeraSort) and 2) non-Data intensive (K-Means, PageRank and Connected Component) and generate workloads with different proportions of these

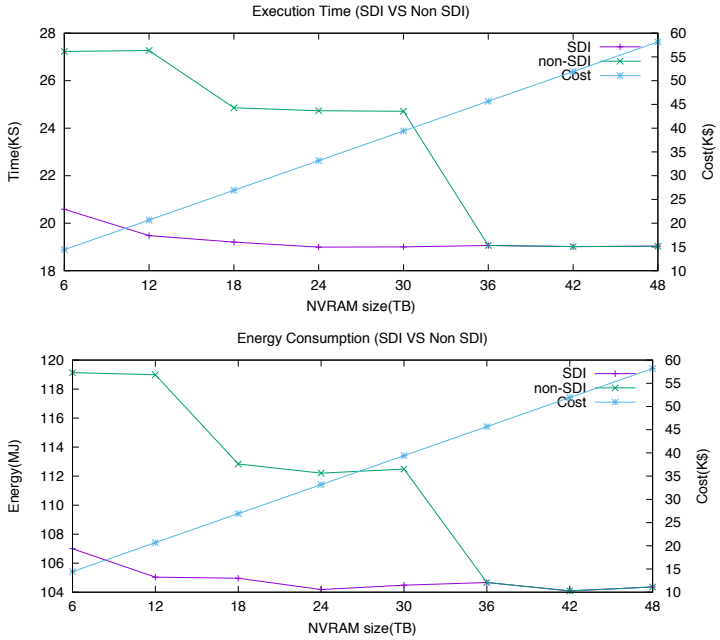


Figure A.11: Execution time (top) and energy (bottom) vs. total storage cost

types of applications (from 0% to 100%). Figure A.12 shows the execution time and energy overhead of the non-SDI scenario with respect to the SDI one. In addition to the tradeoffs shown, the results indicate that the execution time of data-intensive workloads in the SDI scenario can get up to 87.7% shorter than in the non-SDI scenario. However, with 36TB of NVRAM the execution time and energy are similar in both SDI and non-SDI scenarios, which is consistent with the results shown in Figure A.11. These results clearly show the potential of software-defined infrastructure for big data processing frameworks.

A.4 Discussion

This chapter provided a detailed evaluation of performance, power and resource utilization behaviors trends of Hadoop and Spark using a relevant set of Big Data benchmarks and different technology choices. The experimental evaluation supports the argument that NVRAM is a solid candidate for supporting in-memory analytics in ongoing architectures with deeper memory hierarchies. The experimental evaluation also showed

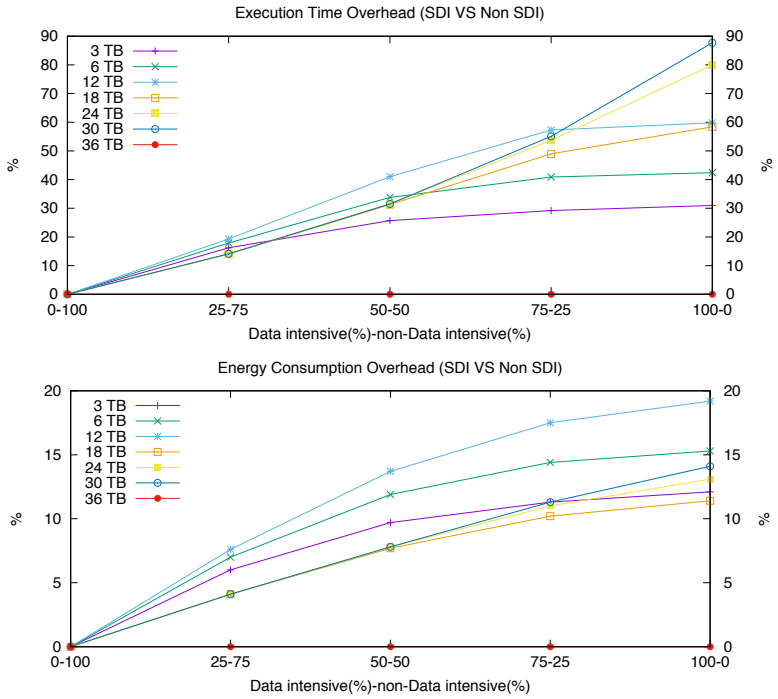


Figure A.12: Execution time (top) and energy (bottom) overheads of non-SDI scenarios with respect to SDI

that the network bandwidth impacts more significantly the performance in Spark workloads than in Hadoop's ones. Simulation-based experimentation showed the significant advantages (upper bound) of software-defined infrastructures for existing Big Data processing frameworks.

The results from this work provide meaningful data points to build multi-criteria application-centric models for Big Data co-design and motivate further research focused on in-memory processing systems with deeper memory hierarchies and different design options and constraints for software-defined infrastructures (e.g., 400G MSA vs. 400/800G embedded optics vs. PCIe 5.0).

References

[1] T. White, *Hadoop: The definitive guide.* " O'Reilly Media, Inc.", 2012.

[2] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Presented as part of the 9th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 12)*, pp. 15–28, 2012.

[3] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy, "Hive: a warehousing solution over a map-reduce framework," *Proceedings of the VLDB Endowment*, vol. 2, no. 2, pp. 1626–1629, 2009.

[4] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, *et al.*, "Spark sql: Relational data processing in spark," in *Proceedings of the 2015 ACM SIGMOD international conference on management of data*, pp. 1383–1394, 2015.

[5] R. Sethi, M. Traverso, D. Sundstrom, D. Phillips, W. Xie, Y. Sun, N. Yegitbasi, H. Jin, E. Hwang, N. Shingte, *et al.*, "Presto: Sql on everything," in *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pp. 1802–1813, IEEE, 2019.

[6] M. Kornacker, A. Behm, V. Bittorf, T. Bobrovytsky, C. Ching, A. Choi, J. Erickson, M. Grund, D. Hecht, M. Jacobs, *et al.*, "Impala: A modern, open-source sql engine for hadoop.," in *Cidr*, vol. 1, p. 9, 2015.

[7] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, *et al.*, "Storm@ twitter," in *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pp. 147–156, 2014.

[8] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, "Apache flink: Stream and batch processing in a single engine," *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, vol. 36, no. 4, 2015.

[9] K. Shvachko, H. Kuang, S. Radia, R. Chansler, *et al.*, "The hadoop distributed file system.," in *MSST*, vol. 10, pp. 1–10, 2010.

[10] H. Li, A. Ghodsi, M. Zaharia, S. Shenker, and I. Stoica, "Tachyon: Reliable, memory speed storage for cluster computing frameworks," in *Proceedings of the ACM Symposium on Cloud Computing*, pp. 1–15, ACM, 2014.

[11] "IBM:What is big data: Bring big data to the enterprise." `http://www-01.ibm.com/software/data/bigdata/what-is-big-data.html/`, 2012.

[12] C. Lynch, "Big data: How do your data grow?," *Nature*, 2008.

[13] L. Liao, "Intel silicon photonics: from research to product," *IEEE Components, Packaging and Manufacturing*, 2017.

[14] A. K. Das, S.-J. Park, J. Hong, and W. Chang, "Evaluating different distributed-cyber-infrastructure for data and compute intensive scientific application," in *Big Data (Big Data), IEEE Intl. Conf. on*, pp. 134–143, 2015.

[15] K. N. Khan, M. A. Hoque, T. Niemi, Z. Ou, and J. K. Nurminen, "Energy efficiency of large scale graph processing platforms," in *Proceedings of the 2016 ACM International Joint Conference on Pervasive and Ubiquitous Computing: Adjunct*, pp. 1287–1294, ACM, 2016.

[16] S. Kavulya, J. Tan, R. Gandhi, and P. Narasimhan, "An analysis of traces from a production mapreduce cluster," in *Cluster, Cloud and Grid Computing (CCGrid), 2010 10th IEEE/ACM International Conference on*, pp. 94–103, IEEE, 2010.

[17] A. Ganapathi, Y. Chen, A. Fox, R. Katz, and D. Patterson, "Statistics-driven workload modeling for the cloud," in *Data Engineering Workshops (ICDEW), 2010 IEEE 26th International Conference on*, pp. 87–92, IEEE, 2010.

[18] G. Wang, A. R. Butt, P. Pandey, and K. Gupta, "Using realistic simulation for performance analysis of mapreduce setups," in *Proc. of the 1st ACM workshop on Large-Scale system and application performance*, pp. 19–26, 2009.

[19] K. Kim, K. Jeon, H. Han, S.-g. Kim, H. Jung, and H. Y. Yeom, "Mrbench: A benchmark for mapreduce framework," in *Parallel and Distributed Systems, 2008. ICPADS'08. 14th IEEE Intl. Conf. on*, pp. 11–18, 2008.

[20] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. H. Katz, S. Shenker, and I. Stoica, "Mesos: A platform for fine-grained resource sharing in the data center.," in *NSDI*, vol. 11, pp. 22–22, 2011.

[21] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica, "Dominant resource fairness: Fair allocation of multiple resource types.," in *NSDI*, vol. 11, pp. 24–24, 2011.

[22] Y. Chen, A. Ganapathi, and R. H. Katz, "To compress or not to compress-compute vs. io tradeoffs for mapreduce energy efficiency," in *Proceedings of the first ACM SIGCOMM workshop on Green networking*, pp. 23–28, 2010.

[23] "Rumen: A tool to extract Job Characterization Data from Job Tracker Logs." `https://www.top500.org/lists/2016/06//`, 2009.

[24] K. Cardona, J. Secretan, M. Georgiopoulos, and G. Anagnostopoulos, "A grid based system for data mining using mapreduce," in *Seventh IEEE International Conference on Grid Computing*, p. 33, Citeseer, 2007.

[25] S. Hammoud, M. Li, Y. Liu, N. K. Alham, and Z. Liu, "Mrsim: A discrete event based mapreduce simulator," in *Fuzzy Systems and Knowledge Discovery (FSKD), 2010 Seventh Intl. Conf. on*, vol. 6, pp. 2993–2997, 2010.

[26] A. C. Murthy, "Mumak: Map-reduce simulator," *MAPREDUCE-728, Apache JIRA*, 2009.

[27] C. Douglas and H. Tang, "Gridmix3 emulating production workload for apache hadoop," 2010.

[28] "Aloja, benchmark repository and performance analysis tool," 2014.

[29] R. T. Kaushik and M. Bhandarkar, "Greenhdfs: towards an energy-conserving, storage-efficient, hybrid hadoop compute cluster," in *Proceedings of the USENIX annual technical conference*, p. 109, 2010.

[30] Í. Goiri, K. Le, T. D. Nguyen, J. Guitart, J. Torres, and R. Bianchini, "Greenhadoop: leveraging green energy in data-processing frameworks," in *Proceedings of the 7th ACM european conference on Computer Systems*, pp. 57–70, ACM, 2012.

[31] W. Lang and J. M. Patel, "Energy management for mapreduce clusters," *Proceedings of the VLDB Endowment*, 2010.

[32] H. Amur, J. Cipar, V. Gupta, G. R. Ganger, M. A. Kozuch, and K. Schwan, "Robust and flexible power-proportional storage," in *Proceedings of the 1st ACM symposium on Cloud computing*, ACM, 2010.

[33] Y. Chen, S. Alspaugh, D. Borthakur, and R. Katz, "Energy efficiency for large-scale mapreduce workloads with significant interactive analysis," in *Proc. of the 7th ACM european conference on Computer Systems*, pp. 43–56, 2012.

[34] N. Tiwari, U. Bellur, S. Sarkar, and M. Indrawan, "Identification of critical parameters for mapreduce energy efficiency using statistical design of experiments," in *Parallel and Distributed Processing Symposium Workshops, 2016 IEEE International*, pp. 1170–1179, IEEE, 2016.

[35] T. Wirtz and R. Ge, "Improving mapreduce energy efficiency for computation intensive workloads," in *Green Computing Conference and Workshops (IGCC), 2011 International*, pp. 1–8, IEEE, 2011.

[36] S. Li, T. Abdelzaher, and M. Yuan, "Tapa: Temperature aware power allocation in data center with map-reduce," in *Green Computing Conference and Workshops (IGCC), 2011 International*, pp. 1–8, IEEE, 2011.

[37] S. Ibrahim, T.-D. Phan, A. Carpen-Amarie, H.-E. Chihoub, D. Moise, and G. Antoniu, "Governing energy consumption in hadoop through cpu frequency scaling: An analysis," *Future Generation Computer Systems*, vol. 54, pp. 219–232, 2016.

[38] Y. Chen, L. Keys, and R. H. Katz, "Towards energy efficient mapreduce," *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2009-109*, 2009.

[39] N. Yigitbasi, K. Datta, N. Jain, and T. Willke, "Energy efficient scheduling of mapreduce workloads on heterogeneous clusters," in *2nd International Workshop on Green Computing Middleware*, 2011.

[40] L. Luo, W. Wu, D. Di, F. Zhang, Y. Yan, and Y. Mao, "A resource scheduling algorithm of cloud computing based on energy efficient optimization methods," in *Intl. Green Computing Conference (IGCC)*, pp. 1–6, 2012.

[41] A. Murthy, "The hadoop map-reduce capacity scheduler," *URL http://developer. yahoo. com/blogs/hadoop/posts/2011/02/capacity-scheduler*, 2011.

[42] J. Polo, D. Carrera, Y. Becerra, M. Steinder, and I. Whalley, "Performance-driven task co-scheduling for mapreduce environments," in *IEEE Network Operations and Management Symposium-NOMS 2010*, pp. 373–380, 2010.

[43] T. Sandholm and K. Lai, "Dynamic proportional share scheduling in hadoop," in *Job Scheduling Strategies for Parallel Processing*, pp. 110–131, 2010.

[44] M. Lu, L. Zhang, H. P. Huynh, Z. Ong, *et al.*, "Optimizing the mapreduce framework on intel xeon phi coprocessor," in *Big Data, IEEE International Conference on*, pp. 125–130, 2013.

[45] S. Han, N. Egi, A. Panda, S. Ratnasamy, G. Shi, and S. Shenker, "Network support for resource disaggregation in next-generation datacenters," in *Proceedings of the Twelfth ACM Workshop on Hot Topics in Networks*, p. 10, ACM, 2013.

[46] J. Lee, Y. Turner, M. Lee, L. Popa, S. Banerjee, J.-M. Kang, and P. Sharma, "Application-driven bandwidth guarantees in datacenters," in *ACM SIGCOMM Computer Communication Review*, vol. 44, pp. 467–478, ACM, 2014.

[47] P. X. Gao, A. Narayan, S. Karandikar, J. Carreira, S. Han, R. Agarwal, S. Ratnasamy, and S. Shenker, "Network requirements for resource disaggregation.," in *OSDI*, vol. 16, pp. 249–264, 2016.

[48] S. Legtchenko, H. Williams, K. Razavi, A. Donnelly, R. Black, A. Douglas, N. Cheriere, D. Fryer, K. Mast, A. D. Brown, *et al.*, "Understanding rack-scale disaggregated storage," in *9th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 17), Santa Clara, CA*, 2017.

[49] A. Klimovic, C. Kozyrakis, E. Thereska, B. John, and S. Kumar, "Flash storage disaggregation," in *Proceedings of the Eleventh European Conference on Computer Systems*, p. 29, ACM, 2016.

[50] K. Lim, J. Chang, T. Mudge, P. Ranganathan, S. K. Reinhardt, and T. F. Wenisch, "Disaggregated memory for expansion and sharing in blade servers," in *ACM SIGARCH Computer Architecture News*, vol. 37, pp. 267–278, ACM, 2009.

[51] K. Lim, Y. Turner, J. R. Santos, A. AuYoung, J. Chang, P. Ranganathan, and T. F. Wenisch, "System-level implications of disaggregated memory," in *High Performance Computer Architecture (HPCA), 2012 IEEE 18th International Symposium on*, pp. 1–12, IEEE, 2012.

[52] J. Gu, Y. Lee, Y. Zhang, M. Chowdhury, and K. G. Shin, "Efficient memory disaggregation with infiniswap.," in *NSDI*, pp. 649–667, 2017.

[53] Z. Guz, H. H. Li, A. Shayesteh, and V. Balakrishnan, "Nvme-over-fabrics performance characterization and the path to low-overhead flash disaggregation," in *Proceedings of the 10th ACM International Systems and Storage Conference*, p. 16, ACM, 2017.

[54] S. Sur, H. Wang, J. Huang, X. Ouyang, and D. K. Panda, "Can high-performance interconnects benefit hadoop distributed file system," in *Workshop on Micro Architectural Support for Virtualization, Data Center Computing, and Clouds (MASVDC). Held in Conjunction with MICRO*, Citeseer, 2010.

[55] N. S. Islam, M. W. Rahman, J. Jose, R. Rajachandrasekar, H. Wang, H. Subramoni, C. Murthy, and D. K. Panda, "High performance rdma-based design of hdfs over infiniband," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, p. 35, IEEE Computer Society Press, 2012.

[56] X. Lu, M. W. U. Rahman, N. Islam, D. Shankar, and D. K. Panda, "Accelerating spark with rdma for big data processing: Early experiences," in *High-performance interconnects (HOTI), 2014 IEEE 22nd annual symposium on*, pp. 9–16, IEEE, 2014.

[57] X. Lu, D. Shankar, S. Gugnani, and D. K. D. Panda, "High-performance design of apache spark with rdma and its benefits on various workloads," in *Big Data (Big Data), 2016 IEEE International Conference on*, pp. 253–262, IEEE, 2016.

[58] S. Kamburugamuve, K. Ramasamy, M. Swany, and G. Fox, "Low latency stream processing: Apache heron with infiniband & intel omni-path," in *Proceedings of the10th International Conference on Utility and Cloud Computing*, pp. 101–110, 2017.

[59] P. Gupta, "Accelerating datacenter workloads," in *26th International Conference on Field Programmable Logic and Applications (FPL)*, 2016.

[60] D. Manzi and D. Tompkins, "Exploring gpu acceleration of apache spark," in *Cloud Engineering (IC2E), 2016 IEEE International Conference on*, pp. 222–223, IEEE, 2016.

[61] P. Li, Y. Luo, N. Zhang, and Y. Cao, "Heterospark: A heterogeneous cpu/gpu spark platform for machine learning algorithms," in *Networking, Architecture and Storage (NAS), 2015 IEEE International Conference on*, pp. 347–348, IEEE, 2015.

[62] M. M. Rathore, H. Son, A. Ahmad, A. Paul, and G. Jeon, "Real-time big data stream processing using gpu with spark over hadoop ecosystem," *International Journal of Parallel Programming*, pp. 1–17, 2017.

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

[64] 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, pp. 97–110, ACM, 2016.

[65] K. Kambatla and Y. Chen, "The truth about mapreduce performance on ssds," in *28th Large Installation System Administration Conference (LISA14)*, pp. 118–126, 2014.

[66] S. Moon, J. Lee, and Y. S. Kee, "Introducing ssds to the hadoop mapreduce framework," in *Cloud Computing (CLOUD), 2014 IEEE 7th International Conference on*, pp. 272–279, IEEE, 2014.

[67] L. A. Barroso, J. Clidaras, and U. Hölzle, "The datacenter as a computer: An introduction to the design of warehouse-scale machines," *Synthesis lectures on computer architecture*, vol. 8, no. 3, pp. 1–154, 2013.

[68] A. Greenberg, J. Hamilton, D. A. Maltz, and P. Patel, "The cost of a cloud: research problems in data center networks," *ACM SIGCOMM computer communication review*, vol. 39, no. 1, pp. 68–73, 2008.

[69] I. Cano, M. Weimer, D. Mahajan, C. Curino, and G. M. Fumarola, "Towards geo-distributed machine learning," *arXiv preprint arXiv:1603.09035*, 2016.

[70] A. C. Zhou, Y. Gong, B. He, and J. Zhai, "Efficient process mapping in geo-distributed cloud data centers," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, p. 16, ACM, 2017.

[71] C.-C. Hung, L. Golubchik, and M. Yu, "Scheduling jobs across geo-distributed datacenters," in *Proceedings of the Sixth ACM Symposium on Cloud Computing*, pp. 111–124, ACM, 2015.

[72] A. Rajabi, H. R. Faragardi, and T. Nolte, "An efficient scheduling of hpc applications on geographically distributed cloud data centers," in *International Symposium on Computer Networks and Distributed Systems*, pp. 155–167, Springer, 2013.

[73] R. Viswanathan, G. Ananthanarayanan, and A. Akella, "Clarinet: Wan-aware optimization for analytics queries.," in *OSDI*, vol. 16, pp. 435–450, 2016.

[74] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, "Large-scale cluster management at google with borg," in *Tenth European Conference on Computer Systems*, p. 18, ACM, 2015.

[75] Y. Zhang, G. Prekas, G. M. Fumarola, M. Fontoura, Í. Goiri, and R. Bianchini, "History-based harvesting of spare cycles and storage in large-scale datacenters," in *Proceedings of the 12th USENIX conference on Operating Systems Design and Implementation*, no. EPFL-CONF-224446, pp. 755–770, 2016.

[76] Nvidia, "Gpu." `https://www.nvidia.com/en-us/about-nvidia/ai-computing/`. Accessed March, 2019.

[77] Intel, "Optane ssd." https://www.intel.com/content/www/us/en/architecture-and-technology/intel-optane-technology.html. Accessed March, 2019.

[78] N. S. Islam, D. Shankar, X. Lu, M. Wasi-Ur-Rahman, and D. K. Panda, "Accelerating i/o performance of big data analytics on hpc clusters through rdma-based key-value store," in *Parallel Processing (ICPP), 44th International Conference on*, pp. 280–289, IEEE, 2015.

[79] P. S. Rao and G. Porter, "Is memory disaggregation feasible?: A case study with spark sql," in *Proc. of the 2016 Symp. on Architectures for Networking and Communications Systems*, pp. 75–80, 2016.

[80] Intel, "Intel rack scale design." https://www.intel.com/content/www/us/en/architecture-and-technology/rack-scale-design-overview.html. Accessed March, 2019.

[81] hp, "Hp the machine." http://www.labs.hpe.com/research/themachine/. Accessed March, 2019.

[82] J. Taylor, "Facebook's data center infrastructure: Opencompute, disaggregated rack, and beyond," in *Optical Fiber Communication Conference*, pp. W1D–5, Optical Society of America, 2015.

[83] S. Dwarkadas, N. Hardavellas, L. Kontothanassis, R. Nikhil, and R. Stets, "Cashmere-vlm: Remote memory paging for software distributed shared memory," in *International Symposium on Parallel and Distributed Processing*, pp. 153–159, 1999.

[84] M. D. Flouris and E. P. Markatos, "The network ramdisk: Using remote memory on heterogeneous nows," *Cluster Computing*, vol. 2, no. 4, pp. 281–293, 1999.

[85] S. Liang, R. Noronha, and D. K. Panda, "Swapping to remote memory over infiniband: An approach using a high performance network block device," in *IEEE Cluster Computing*, pp. 1–10, 2005.

[86] E. P. Markatos and G. Dramitinos, "Implementation of a reliable remote memory pager.," in *USENIX Annual Technical Conference*, pp. 177–190, 1996.

[87] Intel, "Persistent memory." https://software.intel.com/en-us/persistent-memory. Accessed March, 2019.

[88] 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.

[89] "Presto." https://prestosql.io. Accessed: 2019-12-22.

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

[91] G. Ananthanarayanan, M. C.-C. Hung, X. Ren, I. Stoica, A. Wierman, and M. Yu, "{GRASS}: Trimming stragglers in approximation analytics," in *11th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 14)*, pp. 289–302, 2014.

[92] S. Kambhampati, J. Kelley, C. Stewart, W. C. Stewart, and R. Ramnath, "Managing tiny tasks for data-parallel, subsampling workloads," in *2014 IEEE International Conference on Cloud Engineering*, pp. 225–234, IEEE, 2014.

[93] K. Ousterhout, C. Canel, S. Ratnasamy, and S. Shenker, "Monotasks: Architecting for performance clarity in data analytics frameworks," in *Proceedings of the 26th Symposium on Operating Systems Principles*, pp. 184–200, ACM, 2017.

[94] K. Ousterhout, A. Panda, J. Rosen, S. Venkataraman, R. Xin, S. Ratnasamy, S. Shenker, and I. Stoica, "The case for tiny tasks in compute clusters," in *Presented as part of the 14th Workshop on Hot Topics in Operating Systems*, 2013.

[95] K. Ousterhout, R. Rasti, S. Ratnasamy, S. Shenker, and B.-G. Chun, "Making sense of performance in data analytics frameworks," in *12th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 15)*, pp. 293–307, 2015.

[96] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis, "Dremel: interactive analysis of web-scale datasets," *Proceedings of the VLDB Endowment*, vol. 3, no. 1-2, pp. 330–339, 2010.

[97] L. Sun, M. J. Franklin, S. Krishnan, and R. S. Xin, "Fine-grained partitioning for aggressive data skipping," in *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pp. 1115–1126, ACM, 2014.

[98] S. Chen, W. Wang, X. Wu, Z. Fan, K. Huang, P. Zhuang, Y. Li, I. Rodero, M. Parashar, and D. Weng, "Optimizing performance and computing resource management of in-memory big data analytics with disaggregated persistent memory," in *19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGrid 2019*, pp. 21–30, Institute of Electrical and Electronics Engineers Inc., 2019.

[99] "Tuning spark." `https://spark.apache.org/docs/latest/tuning.html`. Accessed: 2019-12-22.

[100] "Presto - query optimizer: pursuit of performance." `https://www.slideshare.net/Hadoop\_Summit/presto\-query\-optimizer\-pursuit\-of\-performance`. Accessed: 2019-12-22.

[101] H. Zhang, B. Cho, E. Seyfe, A. Ching, and M. J. Freedman, "Riffle: optimized shuffle service for large-scale data analytics," in *Proceedings of the Thirteenth EuroSys Conference*, p. 43, ACM, 2018.

[102] P. Stuedi, A. Trivedi, J. Pfefferle, R. Stoica, B. Metzler, N. Ioannou, and I. Koltsidas, "Crail: A high-performance i/o architecture for distributed data processing.," *IEEE Data Eng. Bull.*, vol. 40, no. 1, pp. 38–49, 2017.

[103] S. Rao, R. Ramakrishnan, A. Silberstein, M. Ovsiannikov, and D. Reeves, "Sailfish: A framework for large scale data processing," in *Proceedings of the Third ACM Symposium on Cloud Computing*, p. 4, ACM, 2012.

[104] A. Rasmussen, V. T. Lam, M. Conley, G. Porter, R. Kapoor, and A. Vahdat, "Themis: an i/o-efficient mapreduce," in *Proceedings of the Third ACM Symposium on Cloud Computing*, pp. 1–14, 2012.

[105] A. Rasmussen, G. Porter, M. Conley, H. V. Madhyastha, R. N. Mysore, A. Pucher, and A. Vahdat, "Tritonsort: A balanced large-scale sorting system.," in *NSDI*, 2011.

[106] "Cosco: An efficient facebook-scale shuffle service." `https://databricks.com/session/cosco-an-efficient-facebook-scale-shuffle-service`. Accessed: 2019-12-25.

[107] S. Muralidhar, W. Lloyd, S. Roy, C. Hill, E. Lin, W. Liu, S. Pan, S. Shankar, V. Sivakumar, L. Tang, *et al.*, "f4: Facebook's warm {BLOB} storage system," in *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*, pp. 383–398, 2014.

[108] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, *et al.*, "Spanner: Google's globally distributed database," *ACM Transactions on Computer Systems (TOCS)*, vol. 31, no. 3, p. 8, 2013.

[109] S. Chen and I. Rodero, "Understanding behavior trends of big data frameworks in ongoing software-defined cyber-infrastructure," in *Proceedings of the Fourth IEEE/ACM International Conference on Big Data Computing, Applications and Technologies*, pp. 199–208, ACM, 2017.

[110] "Intel® optane™ dc persistent memory." `https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html`. Accessed: 2019-12-24.

[111] "Amazon s3." `https://aws.amazon.com/s3/`. Accessed: 2019-12-29.

[112] Spark, "Rdd persistence." https://spark.apache.org/docs/2.0.0/programming-guide.html#rdd-persistence, 2017.

[113] K. Wang and M. M. H. Khan, "Performance prediction for apache spark platform," in *Proceedings of HPCC-CSS-ICESS'15*, pp. 166–173, IEEE, 2015.

[114] R. Thakur, R. Rabenseifner, and W. Gropp, "Optimization of collective communication operations in mpich," *The International Journal of High Performance Computing Applications*, vol. 19, no. 1, pp. 49–66, 2005.

[115] "Apache spark on amazon emr." `https://aws.amazon.com/emr/features/spark/`. Accessed: 2018-12-3.

[116] "Scaling clusters." `https://cloud.google.com/dataproc/docs/concepts/configuring-clusters/scaling-clusters`. Accessed: 2018-12-3.

[117] "Sparking on tencent cloud." `https://cloud.tencent.com/product/sparkling`. Accessed: 2018-12-3.

[118] T. Hoefler, E. Jeannot, and G. Mercier, "An overview of process mapping techniques and algorithms in high-performance computing," 2014.

[119] C.-H. Lee, M. Kim, and C.-I. Park, "An efficient k-way graph partitioning algorithm for task allocation in parallel computing systems," in *Systems Integration, 1990. Systems Integration'90., Proceedings of the First International Conference on*, pp. 748–751, IEEE, 1990.

[120] H. Chen, W. Chen, J. Huang, B. Robert, and H. Kuhn, "Mpipp: an automatic profile-guided parallel process placement toolset for smp clusters and multiclusters," in *Proceedings of the 20th annual international conference on Supercomputing*, pp. 353–360, ACM, 2006.

[121] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes, "Borg, omega, and kubernetes," *Queue*, vol. 14, no. 1, p. 10, 2016.

[122] "Spark data locality." `https://spark.apache.org/docs/latest/tuning.html\#data-locality`. Accessed: 2018-12-3.

[123] R. S. Xin, J. Rosen, M. Zaharia, M. J. Franklin, S. Shenker, and I. Stoica, "Shark: Sql and rich analytics at scale," in *ACM SIGMOD International Conference on Management of data*, pp. 13–24, 2013.

[124] "Shuffle operations." `https://spark.apache.org/docs/latest/rdd-programming-guide.html\#shuffle-operations/`. Accessed: 2018-12-3.

[125] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation*, (San Jose, CA), pp. 15–28, 2012.

[126] Spark, "Spark dynamic resource allocation." `https://spark.apache.org/docs/latest/job-scheduling.html#dynamic-resource-allocation`. Accessed March, 2019.

[127] Spark, "Shuffle operations." `https://spark.apache.org/docs/latest/rdd-programming-guide.html#shuffle-operations`. Accessed March, 2019.

[128] D. Williams, ""Device DAX" for persistent memory." `https://lwn.net/Articles/687489/`. Accessed March, 2019.

[129] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin, "Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web," in *Proceedings of the Twenty-ninth Annual ACM Symposium on Theory of Computing*, STOC '97, pp. 654–663, 1997.

[130] A. Rudoff, "Persistent memory: The value to hpc and the challenges," in *Proceedings of the Workshop on Memory Centric Programming for HPC*, (New York, NY, USA), pp. 7–10, ACM, 2017.

[131] S. Huang, J. Huang, J. Dai, T. Xie, and B. Huang, "The HiBench benchmark suite: Characterization of the mapreduce-based data analysis," in *Data Engineering Workshops (ICDEW), 2010 IEEE 26th International Conference on*, pp. 41–51, IEEE, 2010.

[132] "Apache spark on k8s best practice and performance in the cloud download slides." `https://databricks.com/session/apache-spark-on-k8s-best-practice-and-performance-in-the-cloud`. Accessed: 2019-12-22.

[133] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, and R. Sears, "Mapreduce online.," in *Nsdi*, vol. 10, p. 20, 2010.

[134] C. Engle, A. Lupher, R. Xin, M. Zaharia, M. J. Franklin, S. Shenker, and I. Stoica, "Shark: fast data analysis using coarse-grained distributed memory," in *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pp. 689–692, ACM, 2012.

[135] L. M. Grupp, J. D. Davis, and S. Swanson, "The bleak future of nand flash memory," in *Proceedings of the 10th USENIX conference on File and Storage Technologies*, pp. 2–2, USENIX Association, 2012.

[136] V. Kasavajhala, "Solid state drive vs. hard disk drive price and performance study," *Proc. Dell Tech. White Paper*, pp. 8–9, 2011.

[137] S. Chen and I. Rodero, "Exploring the potential of next generation software-defined in memory frameworks," in *2018 30th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pp. 201–208, IEEE, 2018.