Opportunistic scheduling in cluster computing

by

Francis Deslauriers

A thesis submitted in conformity with the requirements
for the degree of Master's of Applied Science
Graduate Department of Electrical & Computer Engineering
University of Toronto

Abstract

Opportunistic scheduling in cluster computing

Francis Deslauriers

Master's of Applied Science

Graduate Department of Electrical & Computer Engineering

University of Toronto

2016

Abstract

Acknowledgements

thanks

Contents

List of Figures

Chapter 1

Introduction

Today, modern data centers provide high-availability computing and large storage capacities at relatively low cost. These facilities are used by cloud computing providers to host the software applications of various types of organizations for handling their core business and managing their operational data. The cloud infrastructure enables collecting and storing large amounts of data. Organizations use this data to perform data analysis, which helps with discovering useful business information, decision making and predicting trends [**?**].

An important challenge in this cloud environment is efficient analysis of large data sets. Today, this analysis is performed using various cluster computing frameworks that are inspired by MapReduce [20]. These frameworks are designed to enable distributed processing on large data sets that are stored across large numbers of nodes built from commodity hardware. The frameworks are designed with the fundamental assumption that hardware failures are common, and are thus designed to handle these failures using techniques such as data replication. This approach simplifies the design of applications, which can focus on their logic.

These frameworks expose functional programming style interfaces so that applica-

tions can be designed to scale to hundreds or thousands of machines. A computation or a job is split into small units of work called tasks that can be run on any machine in the cluster. The different steps of the computation are expressed in stages, where the output of a stage is the input of the next stage, similar to Unix command line pipeline. In a MapReduce system, there are generally two stages, the Map and the Reduce stages. The Map stage is an highly parallel phase, where each Map task reads a part of the input data set from disk or the network and executes the first step of the computation, independently of the other Map tasks. The output of this stage is then sent to the Reduce tasks, which produce the final result for a job.

In a typical MapReduce configuration, each node of the cluster has a number of disks and stores a part of the application data. The partitioning of the data is managed by a distributed file system that splits the data in blocks and stores replicas of the blocks across multiple nodes, ensuring fault tolerance and data availability in the presence of certain node and network failures.

Even though the Map tasks can be scheduled on any node of the cluster without affecting the correctness of the computation, accessing data from a node that stores the data locally is much more efficient than accessing it through the network. This avoids network traffic and delay caused when streaming data from one node to another. Cluster computing frameworks are designed to try to schedule computations on a node storing the input data, and several research projects have focused on improving data locality of tasks [**?**, **?**].

Apache Hadoop, a commonly-used cluster computing framework that implements the MapReduce paradigm, considers three levels of data-locality. The first option is to the run the computation on a node that stores the data. The data is read from disk to memory is accessed from there. This option is called disk-locality and avoids network traffic altogether. If another task consumes the same input and is scheduled

on this node, it can access the data from memory and avoid disk traffic. The second alternative, when disk-locality is not possible, is rack-locality. Nodes in a cluster are physically located in racks, with each rack containing 20 to 42 nodes that are connected together by a single top-of-the-rack switch. Any network traffic between two nodes in a given rack traverses this switch and does not affect nodes on other racks. With rack-locality, a computation is placed in the same rack in which the data is located. Finally, the last option is to place the computation at any node. In this case, data must be sent across racks using an inter-rack switch, which can become a bottleneck in larger clusters.

Recent workload studies have shown that there is significant data reuse in cluster computing environments. We typically find that a small set of files are accessed a large number of times. In this case, we expect that the buffer cache in the operating system will be effective, and minimize the cost of accessing these files after they are read the first time. We found that this assumption does not hold in cluster computing frameworks. Two jobs consuming the same data see limited benefits of caching even when they are scheduled close in time. This occurs because data is available at multiple locations (replicas), and since jobs run independently of each other, they are unaware of which replica has been accessed recently. A task may therefore read data from disk even when another node has this data in memory. Furthermore, jobs do not have knowledge of what is cached on the various cluster nodes and thus cannot optimize their operation based on the cached data.

In this thesis, we argue that jobs in a cluster computing environment should be aware of data that is memory resident in the cluster. This residency information can be used in two ways. First, a replica can be preferentially accessed when its data is memory resident. Second, the tasks of a job can be reordered to take advantage of data that is currently memory resident. This reordering is possible because there

is generally no correctness requirement on the order in which tasks are scheduled. We have designed a system called Quartet that embodies this approach. Quartet gathers, aggregates and selectively distributes information about the data cached in the cluster to jobs. Jobs take this information into account when scheduling their tasks on a given node by prioritizing tasks that access cached data. Jobs schedule as many such memory-local tasks as possible before scheduling disk-local tasks and tasks at other locality levels.

We have implemented Quartent for the Apache Hadoop and the Apache Spark frameworks. Our evaluation shows that Quartet significantly improves the page cache hit ratio for common file access patterns in both systems. This improvement translates to a reduction in the run time of the jobs. Programmers and data analysts that expect significant data reuse in their workloads can use Quartet to more efficiently utilize their hardware resources.

Our main contribution is the design of a system that leverages memory residency information to improve the performance of cluster computing workloads by reducing disk accesses. Based on the observation that modern cluster computing workloads show significant amount of data reuse across jobs, we have designed Quartet to reorder the work inside jobs based on memory residency of their input data. Quartet does not assume any knowledge of the files being accessed or the ordering of jobs. Quartet does not delay the execution of jobs and thus ensures their forward progress. We have implemented the Quartet approach for two different cluster computing frameworks, and expect that it can be easily extended to other cluster computing frameworks.

The rest of the thesis describes our approach in more detail. Chapter 2 describes the related work and current research in this area. Chapter 3 provides motivation for this work by describing cluster computing workloads and particularly how data is reused across jobs. Then, Chapter 4 describes the Quartet architecture. Chapter 5

evaluates the performance of Quartet using various workloads. We then describe our conclusions in Chapter 6.

Chapter 2

Related Work

Using multiple processing units to improve the efficiency and reduce the run time of a computation has long been an important research topic. Traditional solutions include multicore CPU architectures and networked machines. The complexities of using multiple processing units is abstracted via frameworks.

In the past, these frameworks typically relied on low level programming languages like FORTRAN and C/C++. They offer flexibility at the expense of user-friendliness. In recent years, cluster computing systems have been making the opposite trade-off: they make it simpler to take advantage of the processing units by reducing the size of the API, which in turn limits flexibility. These solutions have become popular due to the backing of major corporations within the web industry. They are easy to adopt because they rely on high-level, garbage-collected languages like Java and Scala [8, 11].

It is often assumed that it is better to move computation close to the data rather than the other way around because the executable files are often significantly smaller than the data that they consume [28, 41, 44]. A lot of work has been put into placing units of work onto the best possible location [13, 41, 4, 23, 29]. This preference is often

referred to as data-locality and there is typically a hierarchy of preferred locations. As the ability of frameworks to achieve disk-locality, the focus shifted toward improving memory locality. Because accessing data from memory is orders of magnitude faster than accessing it from disk, several studies explored the idea of improving memory locality of tasks [**?**, 13, 6, 1].

2.1 Traditional Distributed Computing

Saturating all cores of a modern, multi-core CPU is challenging. The key challenges to maximizing one's usage of available resources are data sharing, communication, and synchronization. Multiple libraries have been developed to simplify the task of writing concurrent programs. OpenMP [19] and Intel's Thread Building Blocks [35] are good examples of libraries abstracting some of the complexity of shared-memory parallel programming. OpenMP makes it easy to convert an existing single threaded application to run in parallel on multiple cores. For example, a developer can parallelize a loop by adding a single pre-processor directive.

Using a large number of machines to work on a particular problem has always been a challenge of the high performance computing community. Network overhead, clock skew, and fault tolerance are only a few of the challenges to be considered when designing such systems. Message Passing Interface (MPI) [33, 21] is a great example of a framework allowing programmers to design a computation that can be scaled to a large number of processors. As the name suggests, MPI defines a communication interface for processes to coordinate and exchange data across one or more machines. The MPI programming model offers a lot of flexibility to the programmer. MPI was [40] and is still [31] used for high performance scientific applications.

Despite the existence of systems that abstract away some of the complexity of

parallel computing, their programming interfaces remain hard to master. An experienced programmer can take advantage of the computing power of an entire cluster but novice developers have a steep learning curve to climb.

2.2 Modern Distributed Computing

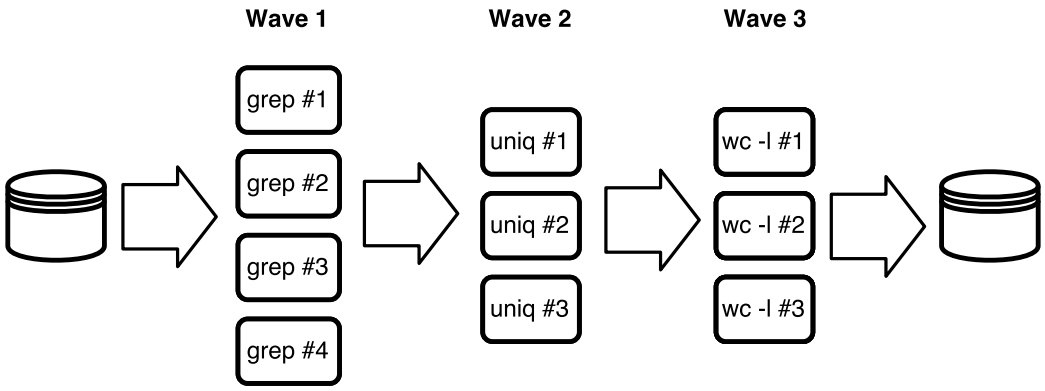
With the steady increase in data set size and the economic value of this data, the need for scalable and easy to use computing paradigms became obvious. Cluster computing frameworks like Apache Hadoop and Apache Spark have helped to democratise the use of high performance computing installations. Software developers can now scale their computations to hundreds and thousands of machines without having to master every aspect of parallel processing and distributed systems.

To scale to a large number of machines, these systems often split their computations in smaller highly parallelizable units of work called *tasks*. The different steps of a distributed algorithm are represented in waves of identical tasks. When an application is made of multiple steps, the data is processed in a pipeline of waves where the output of some step will be the input of another step. Tasks within the same wave do not have any dependency within themselves and can be run completely in parallel on a large number of machines. Figure 2.1 shows an example of such a multi-waves distributed application. The data is read from stable storage by the first wave and sent down the pipeline until the final wave saves the result back to disk.

2.2.1 Apache Hadoop

The Apache Hadoop project[8] is a free and open source cluster computing umbrella project. It was first designed at Yahoo! and is now maintained by a wide variety of organisations including Cloudera [18] and Hortonworks[26]. The Hadoop project

Figure 2.1: Pipelined distributed application

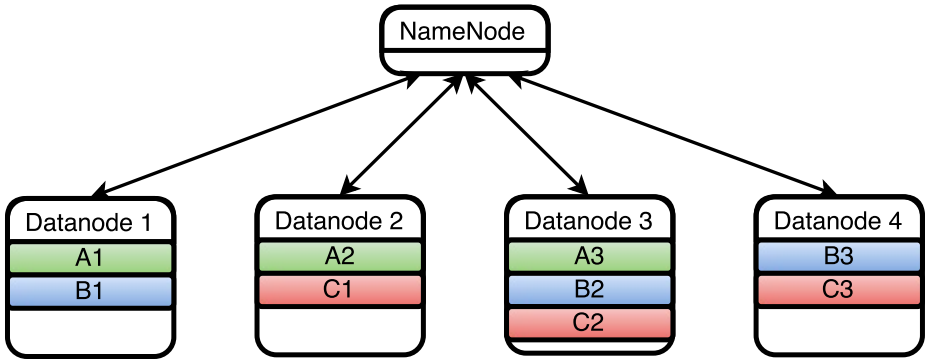


contains three main components: a distributed file system, a resource manager, and application interface that is implemented by users.

2.2.2 Hadoop Distributed File System

The Hadoop Distibuted File System (HDFS) is the storage component of Hadoop. The HDFS design is inspired by the Google File System [24], which aims to ensure data availability, fault tolerance, and the ability to distribute data to thousands of nodes. In HDFS, a large set of nodes in the cluster are running a Datanode agent that handles the on-disk storage and a master NameNode that handles file system operations like file creation, access and deletion. On HDFS, files are split in unit of a fixed size, typically 128MB or 256MB, called HDFS blocks. HDFS blocks are replicated on three different nodes to ensure availability and fault-tolerance. HDFS distributes those replicas on different racks to avoid the impact of correlated failure of multiple nodes on the same rack or network partitions within the cluster. For example, Figure 2.2 represents a simple HDFS cluster consisting of four datanodes and one namenode storing a single large file. The file is split in three blocks (A, B, and C) of 128 MB each, and each of these blocks is replicated on three different

Figure 2.2: Hadoop Distributed File System Architecture



locations (e.g. A1, A2, and A3 for block A).

2.2.3 Hadoop YARN

Hadoop YARN (short for Yet Another Resource Negotiator) is the resource management system on Hadoop. It defines the protocol in which YARN applications are assigned resources. YARN can enforce multiple resource scheduling policies such as Fair scheduling [15] or Capacity scheduling [14] to ensure that the resources of the cluster are used efficiently and fairly among its users. Computing resources are encapsulated as one or more containers on each node and YARN scheduling decisions are at the container granularity.

YARN Resource Manager

The Resource Manager(RM) is the central point of a YARN system [34]. It is running on a node of the cluster and controls access to the computing resource of the cluster. It interacts with the application to ensure that they get their fair shares of the resources.

YARN Application Master

YARN Application Masters (AM) are responsible for the execution of the entire computation, including requesting resources and scheduling work. YARN applications are initialized by the user by requesting a first container from the resource manager in order to run the AM. The AM then takes over the control of the execution and then requests the number of containers specified by the user from the Resource Manager. As containers are assigned to it, the AM will schedule work units on them according to its needs. A popular YARN application is Hadoop MapReduce, which will be discussed in detail in Section 2.2.4.

Resource Request

On YARN, AM need to request containers from the Resource Manager according to their needs in computing power, memory and locations.

YARN offers the possibility for applications to ask for containers with a specific number of virtual CPU or a minimum amount of memory dedicated to it.

Because of the large amount of data involved, data locality is an important factor of performance [**?**] in cluster computing framework. Running computation on a node that stores its input data on one of its disks is preferable to reading it from a remote node. Most distributed file systems including HDFS, ensure the availability of data by replicating it on multiple machines, thus making it possible to achieve this data locality on multiple hosts. Most applications want to express this preference for certain locations to increase the data-locality of their computation. YARN enables applications to express this desire when they ask for containers. They can ask for a container on a specific node or rack. Alternatively, if a piece of computation would not see any benefits due to being run a specific node, application can also express

that this computation can be run on any node of the cluster.

Allocate API

The *allocate* call is the main endpoint of the resource management interface. Through this endpoint, applications request resources on the cluster to run their computation. These resources are allocated in the form of YARN containers, which have a specific number of CPU cores, available memory and a specific host machine. An allocation grants temporary ownership of these specific resources on a given machine. The reply to an *allocate* call may consist of zero or more allocations of containers. When an application is done using a container, it notifies the RM through the next *allocate* call and loses the ownership of it.

YARN's resource allocation model is of the Late binding type. After receiving an allocation, the application can decide what task to schedule on that machine and is not obliged to run what it had initially intended when it first asked for the resources. The application can also return the container, without using it, if it is not needed anymore.

2.2.4 Hadoop MapReduce

Hadoop MapReduce is a popular YARN application. Its design is inspired by Google's MapReduce[20] system where users can express their complex computations by implementing two methods, namely the Map and Reduce methods. Figure 2.3 shows a dataflow in a MapReduce application. Over the course of the execution of a job, the data is read from HDFS by the Map tasks for the first computation stage and then sent to the Reduce tasks which finalize the computation and write the final result back to HDFS. Throughout this section, we will use the common example of the

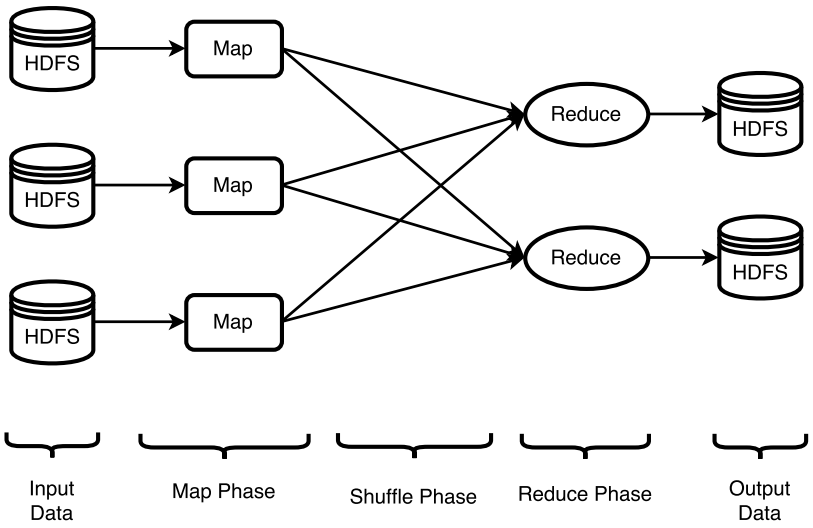


Figure 2.3: MapReduce Paradigm

word counting application that is distributed with the Hadoop source code.

Map phase

The Map method expresses a transformation that is applied to the entire dataset in parallel. The method resembles the Map function in functional programming languages as it applies a function to all the elements of a collection and returns the output. The output of the Map task is a list of key-value pairs. The Map method of WordCount consists of reading the input data, splitting each line on white spaces, and outputting a key-value pair where the key is the word and the value is the integer one. A task running the Map function is often referred to as a Mapper.

Reduce phase

The Reduce phase receives the output of all the Map tasks and produces the final result of the job. The number of reducers can be configured by the user and highly depends on the nature of the job. Each reduce task receives a partition of the keys

produced by the Map tasks and applies a reduction on them. All identical keys will be sent to the same Reduce task. In WordCount, since the keys are words, identical words will be sent to the same Reduce task. The reducer then sums all the values of a given key and outputs the number of occurrences for that word in the form of the final key-value pair. A task running the Reduce function is often referred to as a Reducer.

Shuffle phase

The shuffle phase takes place between the map and reduce phase. It is typically a network intensive phase because all mappers might need to send parts of their outputs to all the reducers. The user does not need to customize this phase but sometimes prefers to do so to limit the impact of this network activity on the performance. There are two main compoments that can be used in the shuffle stage to improve performance: a combiner and a partitioner.

By implementing the combiner method, the user can lessen the work of the reducer by pre-reducing its output. The combiner is often called a 'Map-side reducer' because it applies the same transformation that the reducer needs to do. Its advantage is to reduce the burden on the reducers and also limits the amount of network traffic in the shuffle phase.

The partitionner is used to define how the keys should be split amongst all the reducers. By default, the HashPartitioner [12] class is used and simply hashes the keys and uses the modulo operator with the number of reducers to partition the keys. Users can customize this feature by implementing their own partitioner that is relevant to their applications. For example, if the keys are timestamps partitionning them by time intervals could prove useful to find trends related to time.

Figure 2.4 shows a detailed example of a custom application that is designed to

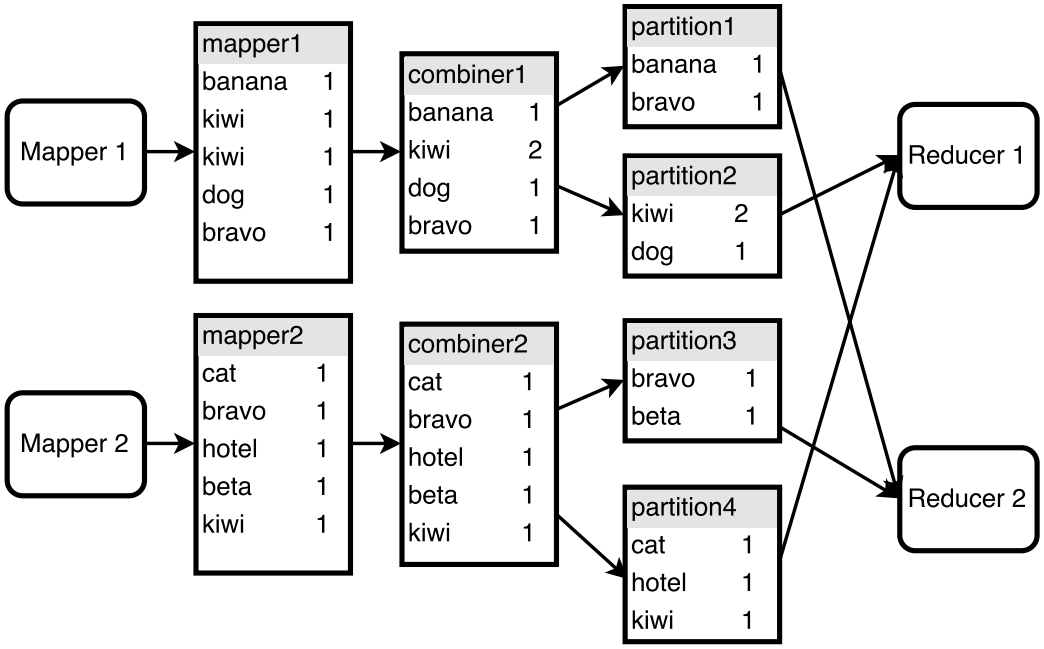


Figure 2.4: Wordcount with custom combiner and partitioner

process the words starting with the letter 'b' in a different way than the ones starting with the other letters. It is implemented using a custom combiner and partitioner. We see that the output of mapper1 can be reduced by the combiner because two of the keys are identical and it is semantically coherent in the context of this word counting application. The custom partitioner splits the keys by placing all the keys starting with the letter 'b' in the first partition and all the other keys in the second. Our job is configured with two reducers, meaning that one reducer will get all the keys starting with 'b' and the other reducer will receive the rest of the keys.

2.2.5 Dryad

Dryad [27] is a cluster computing system designed at Microsoft that aims to tackle the limitations of the MapReduce paradigm by giving more flexibility in the definition of jobs. In Dryad, the user defines a computation as a directed acyclic graph

where vertices are computation steps and edges are communication channels between those steps. Different from the pipeline model of MapReduce, it is possible to use data inputs at any levels of the computation. For example, as the last vertex of a computation DAG, an outside table could be joined with the result of the previous computation.

2.2.6 Quincy

[28] Add section

2.2.7 Apache Spark

Apache Spark [11] was introduced as a solution to speed up cluster computing applications using in-memory data structures. Like Dryad, it allows the user to use a multitude of operations organized in a computation graph. Spark aims to do as much in-memory computation as possible while expressing complex multi-stage applications. An important Spark use-case is running multiple iterations on the same data like the logistic regression classification algorithm. Implemented on Spark, this algorithm can reuse the same input from memory at every iterations. This is possible because using Spark's semantic the user can specify that a dataset should be cached in memory for the entire execution.

Spark is based on the concept of Resilient Distributed Datasets (RDD) [42] which is a data abstraction that allows for fault-tolerant large scale in-memory computation. RDD use the concept of data lineage to provide fault tolerance of the intermediate results. Lineage is a log that contains the input data and each individual computation steps applied to this RDD since its creation. If part of a RDD is lost due to node failure, Spark uses the lineage to recompute it. This technique is useful because it

allows Spark to keep as much of its working set as possible in memory, at the cost of recomputing it in the case of failure. RDDs can also be check-pointed to disk if the user believes that it would be faster to access the disk than to recompute using the lineage.

An important difference with MapReduce is that Spark reuses the JVMs for multiple tasks over the execution of a job as opposed to creating and tearing down a JVM for each individual task. Spark applications will run an Executor on each node it is given access to, and schedule tasks on them.

Like Hadoop MapReduce, Spark can be run as a YARN application but it can also be run in Standalone mode. In Standalone mode, a Spark-specific resource scheduler is managing the nodes of the cluster.

2.2.8 Apache Hive

Apache Hive [9] is a project that was first developed at Facebook as a data warehouse system to facilitate the storage and the usage of structured data [39]. Being built on top of Hadoop MapReduce, it uses a SQL-like language that the framework then compiles to a series of MapReduce jobs. It abstracts the complexity of designing MapReduces jobs and allows users to really quickly design ad-hoc queries on huge amounts of data.

2.2.9 Apache Mahout

Apache Mahout [10] is a cluster computing framework that makes use of the Hadoop ecosystem to tackle machine learning algorithms. Since many machine learning algorithms, such as K-means, are implemented in multiple iterations on the same data, Mahout implements a graph of computation made of MapReduce jobs. The technique

is similar to what Apache Hive uses for SQL query processing. The initial goal of the project was to implement the ten MapReduce versions of common ML algorithms presented in a 2007 paper from Chu et al. [17]. The MapReduce implementations of those algorithm are now being deprecated in favour of frameworks with more flexible interface, like Spark.

2.2.10 Mesos

The Mesos project [25] aims to simplify the management of multiple frameworks running on the same cluster. They claim that no one framework will emerge as the solution to all computing problems so clusters will tend to be shared by multiple frameworks possibly accessing the same data. The goal of Mesos is efficiently share a cluster across this variety of frameworks.

2.3 Data Locality

When computing on large amounts of data in parallel, it is often beneficial to run the code as close as possible to its data rather than move the data to the code. This has been an important topic in cluster computing research. Multiple strategies were developed to improve the data locality of tasks in cluster computing, some required tuning the replication of data blocks and others modified the scheduling itself.

2.3.1 Replication

Scarlett [6] is a system designed to avoid network bottlenecks caused by a highly popular block. Each block being already replicated multiple times on different locations, Scarlett keeps track of the number of accesses to a block and dynamically creates

new replicas for this block on new locations. The applications can then offload tasks consuming this block to the newly created replica and obtain disk-locality. By setting quotas on the amount of disk space used by Scarlett on each node, they make sure that disks are not filled with Scarlett-replicas.

Aurora [43] also tackles the challenges of replicating popular data to improve data locality of jobs. They argue that replicating data based on popularity does not solve the problem of nodes becoming hot spots. An extra replica is much more beneficial if it is placed on a node that is mostly idle because it stores only unpopular files. Aurora is a block placement system built on HDFS that uses a local search algorithm to find the best location for a new replica of popular data in order to enforce a notion of evenness of load.

2.3.2 Delay Scheduling

The Delay scheduling technique presented by Zaharia et al. [41] is a simple and effective strategy to improve data locality of tasks in cluster computing frameworks. Typically, cluster applications do not have correctness requirements on the location where tasks are executed. When an application is launched it shares with the resource manager its preferred locations for every task. If the preferred location, typically disk-local, is not available the scheduler would assign the second best option, a node on the same rack. The Delay Scheduling idea is to wait a limited amount of time in the hope that this preferred location becomes available. It plays on the trade-off between the advantages of disk-local execution and the cost of delaying the progress of the job by waiting.

Delay scheduling is implemented in the Apache YARN resource management system so any applications running on YARN will take advantage of it. Apache Spark

standalone also uses that method to improve data-locality.

2.4 Caching

Disk-locality offers significant advantages over network reads both in term of access latency and in avoiding unnecessary traffic. Similarly, there is a 1-2 orders of magnitude speedup when accessing data frommemory rather than from disk [**?**]. For that reason, there has been work to improve the ability of cluster computing frameworks to take advantage of this speed and capacity of modern memory [30, 22, 38]. Some techniques tried to control or change what is cached on the nodes, and others looked at how we can build more memory-oriented systems.

not more?

2.4.1 PacMan

The PACMan [7] technique can be used to improve the completion time of jobs in cases where entire waves of tasks can be run simultaneously on a cluster. The key idea is what they call the 'all-or-nothing' property of jobs. Tasks of a wave need to all have their data cached to see an improvement in the execution time of the job. If only a subset of the tasks of a wave are sped up by cached data, the rest of the tasks will become the limiting factor of the performance. This property is less relevant when the total number of tasks of a jobs is larger than what can be run in parallel. In this case, a task finishing earlier frees a container that can be used to run another task and thus is not a limiting factor.

To enforce this 'all-or-nothing' property, they implemented a caching coordination mechanism to keep track of the data being read and actively evict files according to certain eviction policies. For example, one policy would first evict a block of a file that is not entirely resident in the caches of the cluster.

2.4.2 HDFS CacheManager

The HDFS CacheManager [13] is a feature of HDFS that was added in XXXX to date allow users to specify popular files to be cached at all times in the cluster. Users can specify the number of replicas that should be in memory. This can be beneficial in cases when those popular files are smaller than the total memory of the cluster. The locations of those cached blocks are then considered when deciding the placement of tasks during the execution. This technique assumes that the administrator is aware of the popularity of the files and can quickly change configuration when the popularity of files shifts.

2.4.3 Alluxio

Alluxio [2], formerly knows as Tachyon[32], is a project that aims to improve distributed applications' performance by interposing a caching layer above the storage. They argue that replication in distributed files systems is limiting the performance of applications by relying on the network and disks to store the data permanently. An application saving its progress on the distributed file system needs to wait for all three replicas to be finished before continuing its execution. They suggest a technique based on Spark's data lineage where instead of writing the output to disk at each step, a distributed cache is used to store the data along with a description of the input and computation. In the case where this data is evicted from memory or lost because of the failure of a node, the system can easily recompute it.

Chapter 3

Motivation

The motivation this work is based on the observation that there is an important amount of data reuse in cluster computing workloads and that these systems are not designed to take advantage of this reuse. In this chapter we will go over previous workload studies and new data to characterize the data reuse in cluster computing.

3.1 Workload studies

Driven by the marketing teams and the will to understand users' habits online, private companies are collecting important amounts of data. Analysts are taking advantage of the ease of use and the power of cluster computing frameworks to extract useful information out of these immense datasets. Several workload studies looked at how this data is used by different users[36, 16, 37]. Those studies helps practitioners understand how the computing resources are used and can guide hardware upgrades and software optimizations. While some focused on corporate and other looked at academic environments we were able to see common characteristics in these various studies. Some of these commonalities related the popularity of input files and time

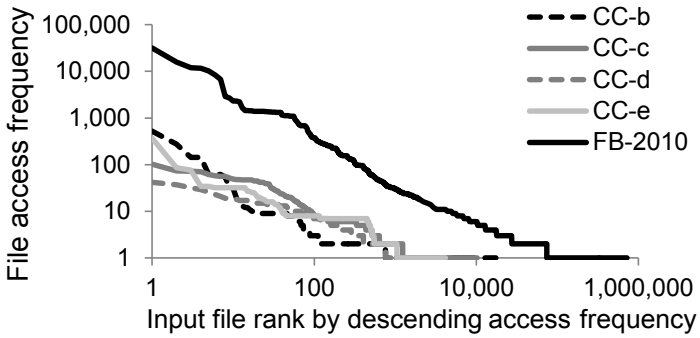


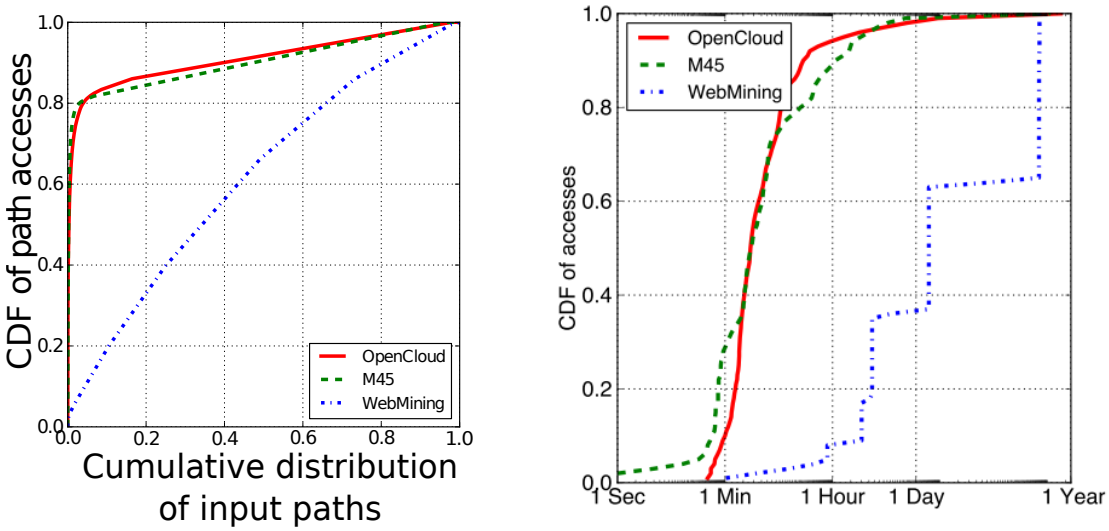
Figure 3.1: File access frequency on five corporate clusters

between accesses. We were also given access to a trace of a large cluster computing system from a large software company that we will call the *Alpha* trace. An analysis of this trace confirmed that those observations are commons in cluster computing systems.

3.1.1 Files popularity

On personal computers, we typically see that a small set of files is getting a large portion of the accesses.[] This tendency of reusing the same data over and over again is also seen in distributed computing.

A study by Chen et al. [16] analysed the traces of five Hadoop clusters and looked at the popularity of files in those different environments. Figure **??** shows a log-log plot where the X-axis shows the files ranked by number of accesses and the Y-axis shows the number of accesses. The linearity of these distributions shows that input popularity follows ZipF a distribution. This clearly shows that in those workloads, a small set of files is getting a large portion of the accesses. This phenomena was also reported on by a study by Ren et al. [36] where researchers analysed the traces of three academic clusters. As shown on figure 3.2, they saw that on two of the studied clusters 10% of the files were responsible of 80% of the accesses.



(a) File popularity

(b) Inter arrival time to the same file

Figure 3.2: Cumulative distribution of file accessed and inter arrival time for three academic clusters

We analysed the Alpha trace and we were able to see this same pattern. Figure 3.3 show a log-log plot where the X-axis represents the rank of files in terms of popularity and the Y-axis shows the number of accesses. On this plot, we see that the distribution also follows a Zipf function.

Diverse workloads and environments are showing similar distribution of the popularity of files. Typically, a small set of files is getting an important number of accesses as the vast majority of files are rarely accessed.

3.1.2 Interarrival time

Ren et al. also measured time between reaccesses of the same file. This metric is really interesting when comes to time to design caching schemes because it gives and estimate of the lifetime of a file in memory. Figure **??** shows the inter arrival time between jobs accessing the same file. We see that for the OpenCloud and M45 clusters

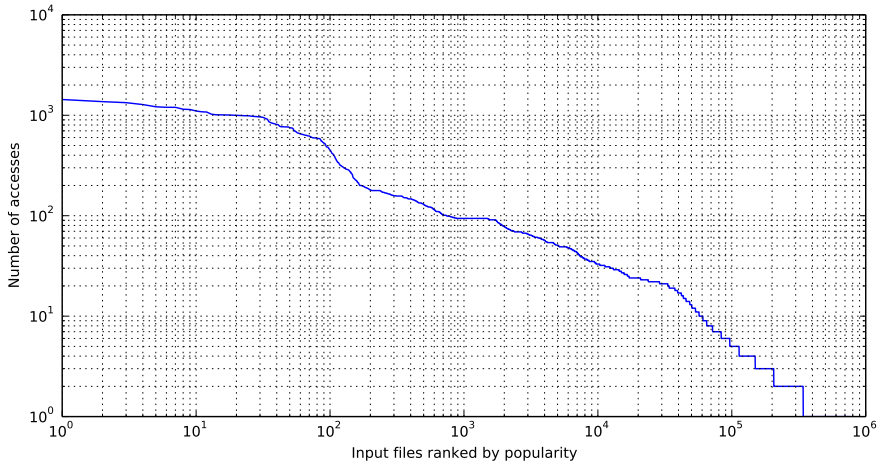


Figure 3.3: Alpha trace file popularity

close to 90% of the reaccessed happen within one hour of the last access.

We witnessed a similar pattern in the Alpha trace, where 70% of the reaccesses to files were occurring within one hour from the previous jobs consuming that file. Figure 3.4 shows a cummulative distribution function of the inter-arrival time of jobs accessing a given file. We can see in inflection point at the one hour mark. We explain this element by the presence of periodic jobs that are scheduled to consume the same file every hours. This observation matches the features of the plot shown in figure 3.5 that shows the number of jobs launched over the span of the day. On this figure, we can see that at the beginning of every hours we see a spike in the number of launched jobs.

3.2 Job scheduling

We saw that cluster computing workloads are showing that data is being reaccessed multiple times and the accesses are happening close in time. Even in this kind of

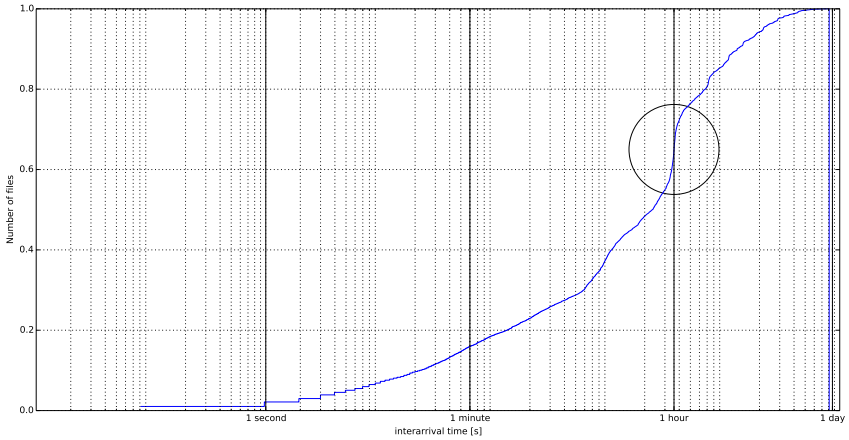


Figure 3.4: Alpha trace interarrival time of jobs

situations, we find that jobs do not have the capability to take advantage of this sharing has we come to expect on shared-memory systems. This is due to the fact that jobs are consuming data in a specific order with no consideration of what the other jobs are doing, or more generally, what is the state of the cluster when they are scheduled.

Even when its entire input data is resident in the cluster's memory, a given job has to choose all the right replicas to fully utilize this opportunity. Without any knowledge of the state of the replicas, the application master has no way to evaluate the value of its placement decisions. We can expect that a job in this situation will get about 30-40% of the place on hot replicas. This number is not exactly one third because of the data structures used in the application master to track the tasks. The matter is even amplify in cases where a file larger than memory is being popular. After a job has read the file, only the portion at the end of the file will be resident in memory. When a second job is scheduled on that file, it will start consuming the file from beginning which will evict the end section of the file. That second job is

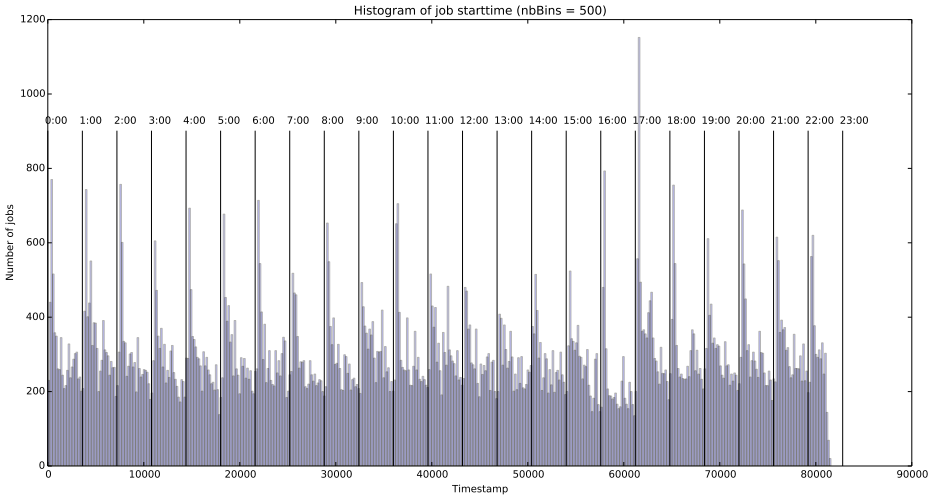


Figure 3.5: Alpha trace Job launched density

effectively evicting its own data out of memory because of the fixed order in which the blocks are consumed. We witness both of these problematic cases on our test environments and we are convinced that given the large amount of data reuse we see in workloads that those cases are widespread.

In this chapter, we saw that workloads from different environments show similar characteristics in terms of data-reuse. Users tend to focus their distributed jobs on a small set of files thus making some files much more popular than others. This popularity should be taken advantage of by caching mechanism. Since files are being accessed with a relatively short amount of time, we should expect memory to play an important role in speeding up these systems.

Chapter 4

Quartet

In cluster computing frameworks, jobs often consume the same files. Ideally, we would like to increase the efficiency of the cluster by leverage this sharing. There are two main challenges to leveraging file sharing. First, a running application needs to be aware of the state of the caches in the cluster. This means knowing what piece of its input data is cached on what nodes. Second, an application needs to use this information to schedule tasks on nodes where their data is cached.

Quartet was designed to tackle these two challenges. Quartet tracks what and where blocks are cached. Quartet does the monitoring the content of the nodes' kernel page caches and distribution of this information out to applications. Applications only receive information about their files of interest. Application masters were enhanced to leverage this cache visibility by prioritizing the assignment of tasks to nodes, where those nodes contain cached data needed by the task. We designed Quartet for a HDFS cluster.

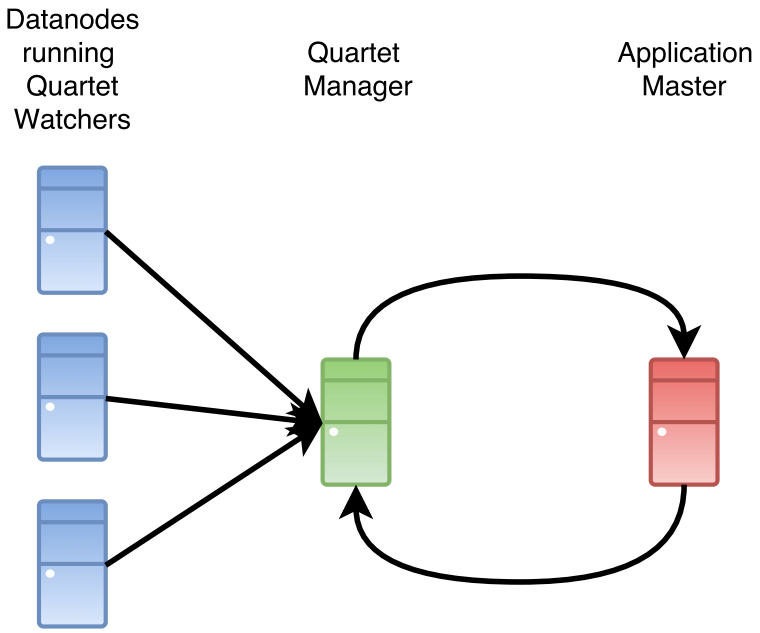


Figure 4.1: Quartet high level overview

4.1 Quartet core

The Quartet core is designed to efficiently collect and distribute a description of the contents of the memory of all the nodes to the running applications. The cache information flows from the Quartet Watchers to the Quartet Manager before being distributed on-demand to the applications. Figure 4.1 shows a high-level overview of the information flow between the different components of the Quartet system.

4.1.1 Duet

Duet[5] is a framework that gives kernel- and user-space applications visibility into the kernel's page cache. Duet is implemented as a Linux kernel module that exposes an API to these applications. Using this API, applications can register for updates on the state of the pages backing a set of files of interest. Using the cache residency information, applications can change the order in which they consume files to avoid

accessing the disk unnecessarily. Duet is designed for cases where multiple applications running on a machine are consuming an overlapping set of files. An interesting example presented in the paper is one where an anti-virus and backup software are running on the same machine. These programs have no strict requirement on the order in which files are analyzed or backed up, so using Duet they can get notified when a file of interest is being read from disk to memory and opportunistically schedule their work on that file. Using this technique, two applications consuming the same set of file can reduce their overall disk traffic by half without explicitly collaborating together.

4.1.2 Quartet Watcher

The Quartet Watcher is an agent running all the HDFS datanodes of the cluster. This agent gathers cache content information using the Duet framework. The Watcher registers a Duet application, which subscribes to events about the pages of HDFS blocks being added and removed from the page cache. The Watcher periodically receives, aggregates, and forwards per-block changes to the central Quarted Manager. Quartet intelligently avoids reporting certain events that cancel out one-another. For example, a page eviction event cancels out the page addition event for that same page.

4.1.3 Quartet Manager

The Quartet Manager is the central entity of the Quartet system. It has complete knowledge of the state of the page caches of the cluster. Upon Quartet start-up, the Manager will reach out and subscribe to a specific rate of updates from all Watcher agents. This rate can be configured to adapt to the timeliness requirements of the

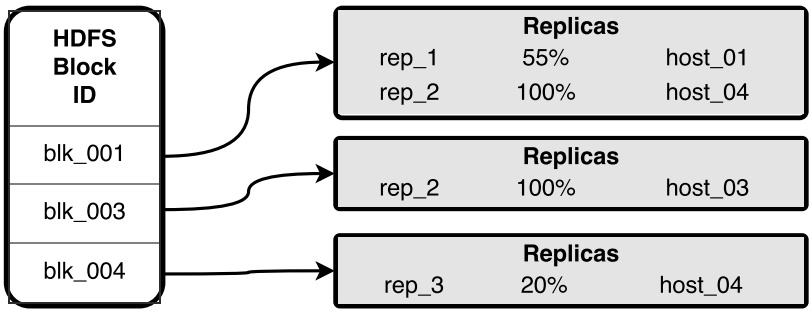


Figure 4.2: Block Residency hash table

applications. The Manager then aggregates these updates per HDFS blocks. The manager keeps track of the number of pages of all the individual replicas of each HDFS blocks. Figure 4.2 shows the hashtable used to store the page residency information of HDFS blocks. The blocks presented here can be of different files as HDFS gives identifiers independently of the file. In this example, we see that three blocks have a portion of their data in memory. Indexed by block identifier, the Manager used the hashtable to maintain the state of the cluster by keeping track of number of pages of the entire block that is resident for each hot replica. It is important that the Manager keeps track of the location of each replica since this information is key for the applications to make the right placement decisions. If all replicas of a block reach 0% residency, the entry for that block is removed for the table.

The Quartet Manager also receives the registrations from the applications. The application Masters register their interest in consuming a certain list of HDFS blocks and periodically query the manager about the status of these blocks. The messages sent to the application masters only reports on the state of replicas that saw a change in their cache residency since the last update. These updates contain the block identifier, the location and the current number of pages for every changed blocks. When a block is no longer of interest because it has been consumed already, the

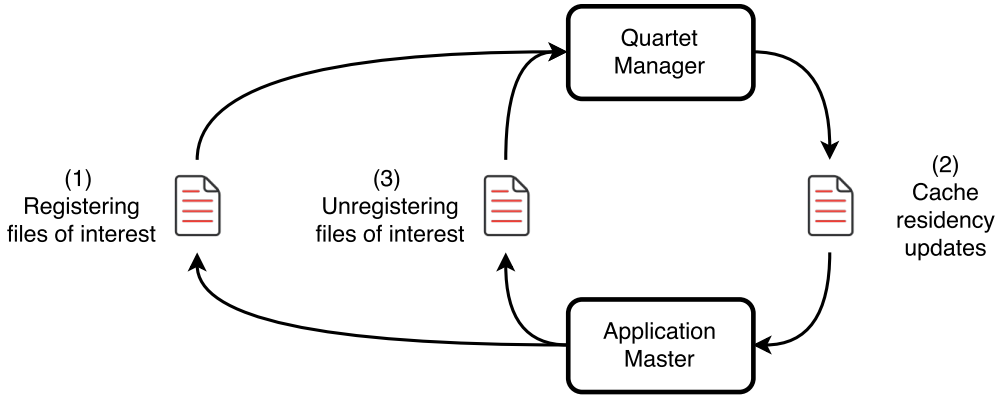


Figure 4.3: Application Master to Quartet Manager communication channel

application cancels its registration for this block to limit the network traffic. Figure 4.3 shows the way the application masters stay up to date on the state of the cluster by communicating with the Quartet Manager.

4.2 Quartet Application Frameworks

While Hadoop MapReduce and Spark standalone being different frameworks their resource management systems are quite similar. Quartet was enable on both of these frameworks with limited amount of changes to the source code. User applications do not need to be changed to work with a Quartet-enabled system.

4.2.1 Apache Hadoop MapReduce

When running Hadoop Mapreduce on Yarn, the application requests containers to the Resource Manager at the beginning of its execution based on the HDFS blocks it plans to consume. During that process, the application master also registers itself to the Quartet Manager and registers the list of blocks of interest.

During its execution, the application periodically queries the manager for the

current status of those blocks. We set period to five seconds but it could be change to adapt to the application. This information is used to choose between container allocations from the resource manager. As explained in section 2.2.3, the resource manager will offer containers to each application depending on the available resources as well as the sharing configurations in relation to the owner of the job. When the application receives a container allocation on a given node, it has to decide what tasks to run on it. When offered a container on a given node, the application master will follow what we call the two steps algorithm which is presented in algorithm 1. We use two qualifier for the tasks in relation to the state of its input a node N. We define a *hot task* as a task whose input is resident in the memory of node N. We define a task *Everywhere cold* if none of the replicas on the block it needs to consume is currently in memory on the cluster but it is on disk on node N.

Algorithm 1 Two-steps algorithm

```
1: Allocatedhost N
2: for all t in hotTasksList do
3:    if t on N then
4:       return t
5:    end if
6: end for
7:
8: for all t in everywhereColdTasksList do
9:    if t on N then
10:      return t
11:   end if
12: end for
13:
14: return delaySchedChoice
```

The 2-steps algorithm is design to ensure that every data reuse opportunities is taken advantage of and the forward progress is ensure. The first loop (line 2) checks if there exist a task left to execute that has its input data in memory on the node

that was offered by the resource manager. If this is the case, we schedule that task on the node. This is the best case scenario because the execution of this task will incur no disk read to the host. Next, at line 8, we look for tasks that would have node locality on the offered node but that none of the replicas are in memory. At this point we know that the currently offered node has nothing of interest to the application, so we choose a task that, to the best of our knowledge, has node-local as its best placement option. Finally, if no task is hot or everewhere cold to this node we fallback the normal delay scheduling policy. It is important to note that in the case of applications running on Yarn, the delay scheduling has already been considered by the resource manager. A non-optimal allocation will be offered to an application only after the configured delay is expired. In the case of our Hadoop implementation, the fallback to delay scheduling is to pick the first node-local task available.

Once a given task is finished, the application master notifies the Quartet Manager that the block consumed by this task is no longer of interest and updates on its status are no longer needed.

4.2.2 Apache Spark

The changes presented in section 4.2.1 were also implemented in Spark Standalone mode. Spark running in this mode is following a really similar resource management scheme that of YARN. We implemented the same two-steps algorithm on this platform as well.

4.2.3 Other frameworks

Our implementation is based on HDFS, any application relying on HDFS for storage could be easily modified to become Quartet-enabled. We can easily imagine Hive,

Hadoop and Spark jobs running simultaneously on a cluster.

In conclusion, the Quartet system was designed to enable applications to make informed scheduling decisions about cached data. We designed Quartet Core which is used to gather and distribute the information about the page caches of the different nodes of the cluster. We modified the Hadoop Mapreduce and Spark Standalone application master consider the data residency information during their scheduling decisions. Since the modifications are contained in the Application master, existing MapReduce and Spark jobs can be used with Quartet without any modifications. As previously mentioned, there is a trend going toward clusters that are shared between multiple frameworks.

Chapter 5

Evaluation

In this chapter, we analyze to what degree re-ordering work based on cache residency information is beneficial for distributed applications. We used a cluster to test simple and more complex workloads in order to measure the benefits of Quartet in term of resource utilization and runtime of workloads.

5.1 Experimental setup

We ran our experiments on a 25 nodes clusters running a custom Linux Kernel with Duet kernel module. Each node had 16 GB of memory, 8 physical cores and are networked together via top of rack switch. Out of these, 24 nodes are HDFS datanodes and one is running the HDFS Namenode and the Yarn Resource Manager. The aggregate resources of the cluster were 386GB of memory and 192 cores. The nodes were configured to run eight tasks simultaneously. We implemented Quartet in two distributed systems: Hadoop MapReduce and Spark Standalone.

In the following sections, we use the term 'vanilla' to characterize the unmodified upstream version of the framework and the term 'quartet' for the quartet-enabled

version.

5.2 Experiments

To evaluate the Quartet system, we ran simple benchmarks and measured the speedup and cache hit rate improvement. Those experiments helped us confirm that our implementation was working properly. We then went on to test more complex workloads, where multiple jobs are running concurrently and consuming a overlapping subsets of data.

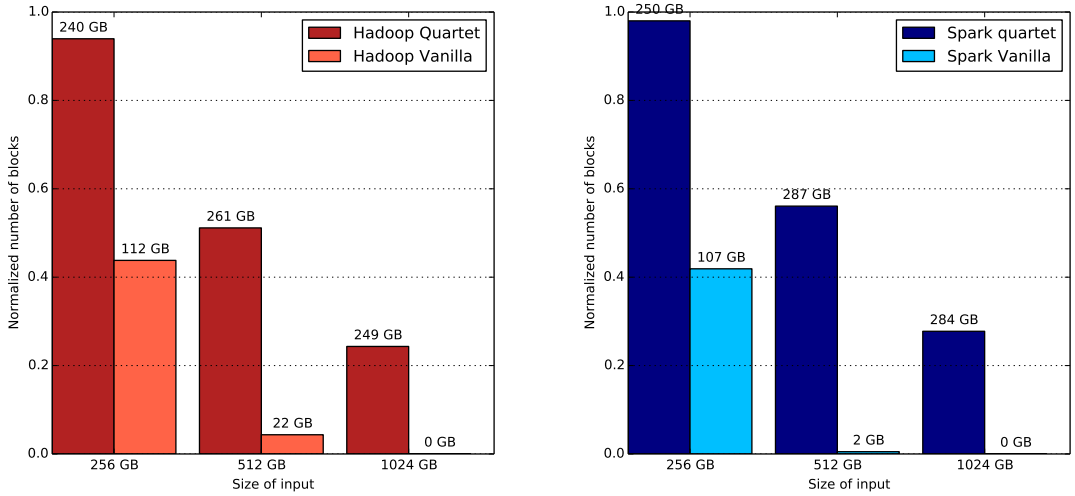
5.2.1 Sequential jobs

This workload consisted of two jobs consuming the same input ran one after the other. We used a simple line counting application that we implemented in both frameworks to simulate an O/I bound application. We varied the size of the input files in relation to the size aggregate memory of the cluster. Out of the three files that we used, one was smaller than the aggregate memory of the cluster(256 GB), one was slightly larger(512 GB) and one about three times as large(1024 GB). Those different sizes allowed us to evaluate impact of the size of the dataset compared to the memory on the reuse opportunities. We measured the runtime and number of blocks accessed from memory for the second job since it is the one that see the benefits of sharing.

For this experiment, all jobs were allocated eight tasks per node.

Block reuse

Figure 5.1 shows the number of blocks that were accessed directly from memory during the execution of the second job for both frameworks. The Y-axis is the number of



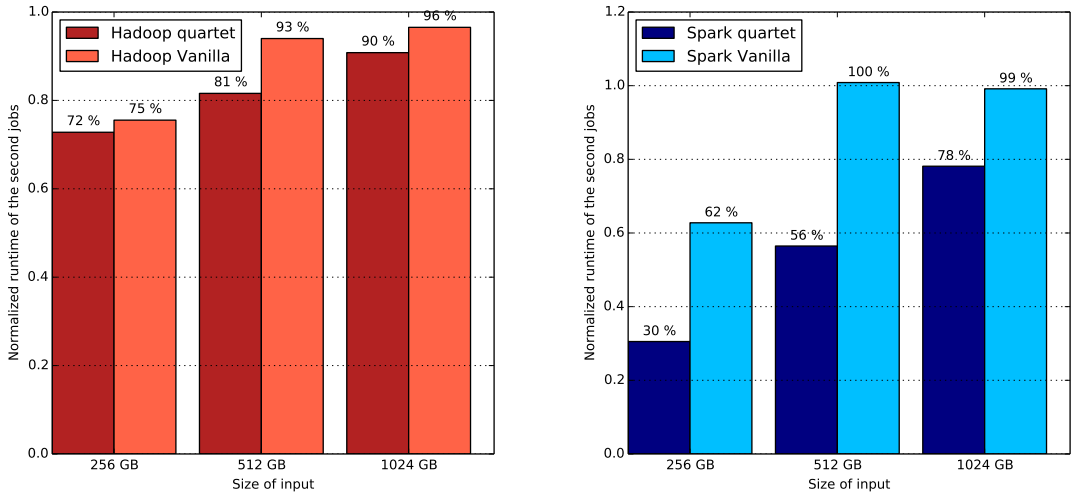
(a) Hadoop (b) Spark

Figure 5.1: Normalized cache hits of HDFS blocks of the second job over the total number of blocks on Quartet and Vanilla implementations of Hadoop and Spark

block reads saved normalized on the total number of block of the input file. The amount of disk reads saved is also displayed in GB on the top of each bar.

Looking the number of blocks reused in the vanilla cases, we see that both frameworks are only able to reuse 40-45% of the dataset even when it fits in the page caches of the nodes. Using Quartet, we are able to access close to the entire dataset from memory.

As we increased the size of the dataset, a negligible portion of the dataset is reused in the vanilla frameworks.This is due to the problem presented in section 3.2 where a job evicts its own input out of memory because jobs in Spark and Hadoop are consuming blocks always in the same order. Both Quartet frameworks are showing similar reuse numbers. As the input size increases, a smaller fraction of the HDFS can be resident at any given time. It is interesting to see that the absolute amount



(a) Hadoop (b) Spark

Figure 5.2: Normalized runtime on the second job over the first on Quartet and Vanilla implementations of Hadoop and Spark

of data used from memory is relatively stable across all runs of the same framework, around 250 GB for Hadoop and 285 GB for Spark. These numbers represent the amount of memory that is remaining for the kernel page caches for this particular workload and configuration. The rest of the memory is used by the operating system kernel, the Quartet Watcher and an handful of Hadoop or Spark JVMs.

Runtime

When looking at the runtime effect on the reuse we saw in the previous section, we see on figure 5.2 that Spark seems to be seeing significantly more benefits than Hadoop. This figure presents the normalized runtime of the second job over the first one on the Y-axis.

The Hadoop Quartet implementation shows only modest improvements from 3%

to 12% additional reduction of runtime compared to the vanilla version. The Spark Quartet implementation on the other hand is doing much better. We can see 32%, 44%, and 21% additional reduction over the vanilla runs for 256 GB, 512 GB, and 1024 GB respectively. We explain this difference in performance between the two implementations with the fact that Spark reuses JVM over the execution of the entire job. Hadoop launches a new JVM for each tasks. This process takes time and the improvements due to caching may be dwarfed by it.

These simple experiments proved that re-ordering work in cluster computing applications is feasible and can have a positive impact by reducing the disk contention and reducing the runtime of jobs. Both of these benefits can have an important impact of the overall efficiency of the cluster by allowing more jobs and more I/O to be scheduled on the same hardware.

5.2.2 Delayed launch and reversed order access

This experiment is to characterize how time between the launch of jobs and the order in which they consume their input is affecting the performance of applications using Quartet.

We also examined how the propagation delay is affecting the performance of frameworks using Quartet. This delay is caused by the fact that the page cache information needs to be transferred from the watchers to the applications. In the worst case scenario, the propagation delay can cause a job to be completely unaware of accesses on its blocks of interest by other jobs. This case can be triggered when two jobs consuming the same file are launched on the same cluster close in time. Due to the propagation delay, jobs will make uninformed scheduling decisions because the information on hot blocks is in transit in the different Quartet components.

In these experiments, we scheduled two jobs consuming the same file while varying the time offset at which the second job is started. We also varied the order in which the blocks are consumed by the second job to see how this propagation delay was affecting the performance of each implementations. In one case, both jobs were consuming the file in the same order and in the other case the second job was working its way from the end of the file towards the beginning. The propagation delay is harmless when the order is reversed because both applications would not be consuming the blocks in the same order. To compare, we measured the time between the launch of the first job until the end of the second job. In this experiment, both jobs are running concurrently on the cluster. Because of differences in the resource managers of YARN and Spark Standalone the number of concurrent tasks could not be the same for both frameworks. In the Hadoop experiment, 192 tasks were running at all time, independently to the number of jobs scheduled but in the Spark experiment jobs were each assigned 96 of the task slots leaving the half of the cluster idle when the only one job was running.

Hadoop

On our Hadoop implementation of Quartet, we see on figure 5.3 that the improvement produced by Quartet are greatly reduced when the two jobs are started at the same time. This is showed by the first data point of the *Quartet-Same* line on the plot. We explain this effect by the presence of a propagation delay inherent to Quartet. From the point of view of a application, very few of its input block are ever hot in the cluster because of the time it takes to communicate a page cache event triggered by the other job from the node to the application master.

As we increase the offset we see that this effect disappears and Quartet shows similar performance on both ordering. We see that with only a 40 seconds delay

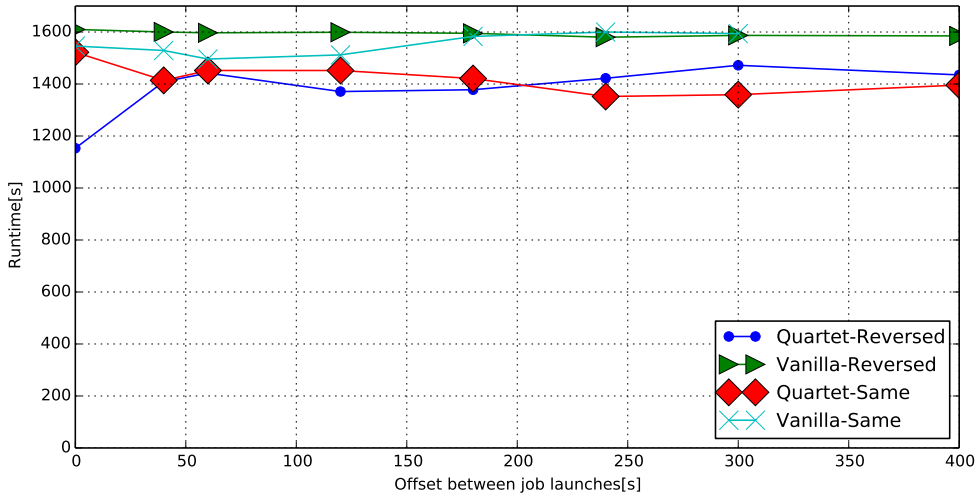


Figure 5.3: Total runtime of two jobs run with diversed input configuration with varying launched offsets on Quartet Hadoop and Vanilla Hadoop

both order show similar performance. The first job that is scheduled takes the lead and consumes its input directly from disk. The events related to those blocks are relayed to the applications interested in these blocks. The second application which is scheduled several seconds later will start by consuming those blocks and will follow the order of which the previous job is consuming blocks.

It is also interesting to compare the performance of both vanilla workloads. We see that when jobs are consuming data in the same order the runtime is reduced. This is due to the natural sharing of the applications. Jobs are sometimes making similar scheduling decisions and thus are sped up if the input is still resident. We see that this effect attenuates as we increase the delay between the jobs. This is due to the normal memory pressure on the node that evicts the blocks following the Least-Recently-Used policy. When the two jobs are consuming data in a different order close to no natural sharing is possible.

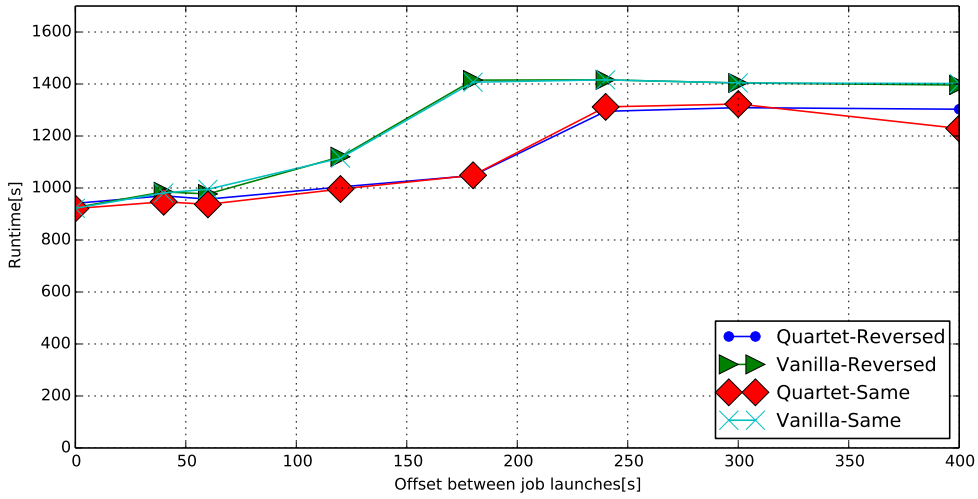


Figure 5.4: Total runtime of two jobs run with diversed input configuration with varying launched offsets on Quartet Spark and Vanilla Spark

Spark

The behaviour of Spark on this workload is different as shown on figure 5.4. Compared to our Hadoop results shown earlier, we see no difference between the two block processing orders across both Vanilla and Quartet implementations because Spark ignores the order in which the input files are supplied to the application.

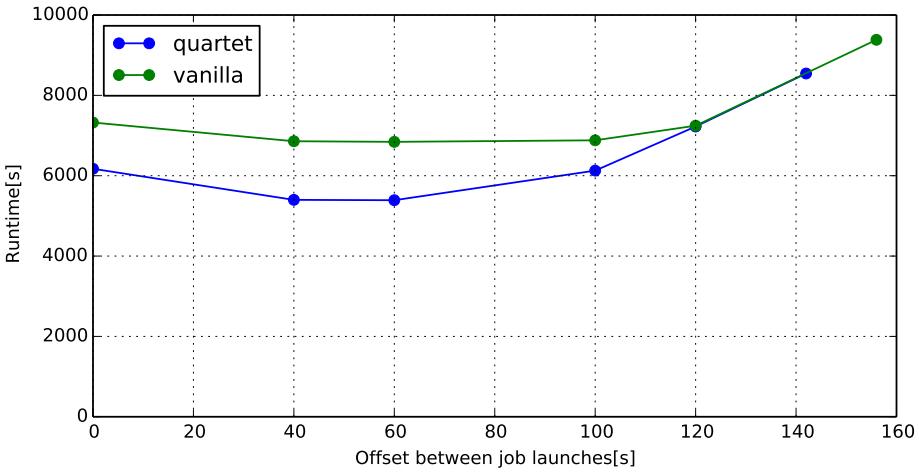
Vanilla Spark is very efficient at taking advantage of natural sharing that occurs within two minutes. On the other hand, using Quartet we can ensure good all reuse opportunities are taken advantage of even after three minutes.

This section is weak

5.2.3 Popular dataset

We evaluated how Quartet would impact the performance of a workload of a large number of jobs each consuming a random subset of the same large dataset. This can simulate a dataset that is shared between several teams each looking at the data from

Figure 5.5: Runtime of 60 Hadoop jobs launched at varying delays consuming a 15% subset of a 1 TB dataset



a different angle.

We scheduled 60 Hadoop jobs each consuming a random 15% subset of blocks forming a 1 TB file. We added the jobs to the queues at varying offsets but only 6 jobs were running simultaneously at any given time. We see on figure 5.5 the total runtime of this workload on our Hadoop cluster on both Quartet and Vanilla. The last data point of each line shows the runtime of where all the jobs were run sequentially.

We see that Quartet can significantly reduce the runtime of such workload with close to a 1300 seconds difference when jobs are scheduled 40 seconds apart which represents a 21% reduction. We measured similar speedup with 30 and 45 jobs with 20% and 21% reduction respectively.

Chapter 6

Conclusion

Based on the observations that datasets are shared between many users and is often reaccessed quickly, we studied how cluster computing frameworks are taking advantage of this sharing. We found that mainstream frameworks like Hadoop and Spark have a hard time seeing the benefits of cached data. This is due to the fact that they consumed data in a specific order with no consideration of the content of the caches of the nodes in the cluster. This rigid scheduling forces applications to miss sharing opportunities. Taking advantage of the data sharing inherent to cluster computing workloads would mean reducing the number of disk reads, speeding up tasks execution as well cutting on disk contention.

We designed Quartet to give visibility on the content of the page caches to the distributed applications. Application masters of Apache Hadoop MapReduce and Apache Spark were modified to leverage this new knowledge. Applications can now reorder their work to schedule tasks on nodes that have their input data resident in memory.

Using Quartet, we were able to show that distributed applications can take advantage of cached data given that they have the knowledge and the capability. We

showed that Quartet can reduce the run time of common reuse use cases and fully take advantage of blocks resident in memory. When some portion of data set is already resident in memory, Quartet enabled jobs are able to fully utilize this data from memory and thus avoiding disk access latency and disk contention.

This approach was showed to work particularly well on Spark but less so on Hadoop MapReduce. We attributed this difference to the fact that the I/O element of the runtime of tasks in MapReduce is relatively shorter than of Spark. So the reduction of this element is less beneficial because of Amdahl's law [3]. This difference is due to the fact that on MapReduce, a new JVM is launched for every task as opposed to being reused for multiple tasks on Spark. Quartet thus seems more beneficial for short running tasks where the disk access takes a significant portion of the run time.

Further work is needed to fully characterize how distributed applications can be improved with the added capability that is offered by Quartet. Traces of real world workloads would help to quantify the benefits that practitioners would witness in their environments. The currently available traces [**?**, **?**] do not characterize file accesses with enough detail to simulate those workloads on a Quartet-enable cluster.

Bibliography

[1] Parag Agrawal, Daniel Kifer, and Christopher Olston. Scheduling shared scans of large data files. *Proc. VLDB Endow.*, 1(1):958–969, August 2008.

[2] Alluxio Open Foundation. Alluxio. `http://www.alluxio.org/`.

[3] Gene M Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, pages 483–485. ACM, 1967.

[4] Khalil Amiri, David Petrou, Gregory R Ganger, and Garth A Gibson. Dynamic function placement for data-intensive cluster computing. In *USENIX Annual Technical Conference, General Track*, pages 307–322, 2000.

[5] George Amvrosiadis, Angela Demke Brown, and Ashvin Goel. Opportunistic storage maintenance. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 457–473, 2015.

[6] Ganesh Ananthanarayanan, Sameer Agarwal, Srikanth Kandula, Albert Greenberg, Ion Stoica, Duke Harlan, and Ed Harris. Scarlett: Coping with skewed content popularity in mapreduce clusters. In *Proceedings of the Sixth Conference on Computer Systems*, EuroSys '11, pages 287–300, 2011.

[7] Ganesh Ananthanarayanan, Ali Ghodsi, Andrew Wang, Dhruba Borthakur, Srikanth Kandula, Scott Shenker, and Ion Stoica. PACMan: Coordinated Memory Caching for Parallel Jobs. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, 2012.

[8] Apache Software Foundation. Apache Hadoop. `https://hadoop.apache.org`.

[9] Apache Software Foundation. Apache Hive. `https://hive.apache.org/`.

[10] Apache Software Foundation. Apache Mahout. `https://mahout.apache.org/`.

[11] Apache Software Foundation. Apache Spark. `http://spark.apache.org`.

[12] Apache Software Foundation. Hash Partitioner. `https://hadoop.apache.org/docs/current/api/org/apache/hadoop/mapreduce/lib/partition/HashPartitioner.html`.

[13] Apache Software Foundation. HDFS Centralized Cache Management. `http://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/CentralizedCacheManagement.html`.

[14] Apache Software Foundation. Yarn Capacity Scheduler. `http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/CapacityScheduler.html`.

[15] Apache Software Foundation. Yarn Fair Scheduler. `http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/FairScheduler.html`.

[16] Yanpei Chen, Sara Alspaugh, and Randy Katz. Interactive analytical processing in big data systems: A cross-industry study of mapreduce workloads. *Proceedings of the VLDB Endowment*, 5(12):1802–1813, 2012.

[17] Cheng Chu, Sang Kyun Kim, Yi-An Lin, YuanYuan Yu, Gary Bradski, Andrew Y Ng, and Kunle Olukotun. Map-reduce for machine learning on multicore. *Advances in neural information processing systems*, 19:281, 2007.

[18] Cloudera inc. Cloudera inc. `http://www.cloudera.com/`.

[19] Leonardo Dagum and Ramesh Menon. Openmp: an industry standard api for shared-memory programming. *IEEE computational science and engineering*, 5(1):46–55, 1998.

[20] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association.

[21] Jack J Dongarra, Rolf Hempel, Anthony JG Hey, and David W Walker. A proposal for a user-level, message passing interface in a distributed memory environment. Technical report, Oak Ridge National Lab., TN (United States), 1993.

[22] Jaliya Ekanayake, Hui Li, Bingjing Zhang, Thilina Gunarathne, Seung-Hee Bae, Judy Qiu, and Geoffrey Fox. Twister: a runtime for iterative mapreduce. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, pages 810–818. ACM, 2010.

[23] Michael J Fischer, Xueyuan Su, and Yitong Yin. Assigning tasks for efficiency in hadoop. In *Proceedings of the twenty-second annual ACM symposium on Parallelism in algorithms and architectures*, pages 30–39. ACM, 2010.

[24] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 29–43, New York, NY, USA, 2003. ACM.

[25] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D Joseph, Randy H Katz, Scott Shenker, and Ion Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *Proceedings of the 8th NSDI*, pages 295–308, 2011.

[26] Hortonworks. Hortonworks. `http://hortonworks.com/`.

[27] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *Proceedings of the 2Nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07, pages 59–72, New York, NY, USA, 2007. ACM.

[28] Michael Isard, Vijayan Prabhakaran, Jon Currey, Udi Wieder, Kunal Talwar, and Andrew Goldberg. Quincy: fair scheduling for distributed computing clusters. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 261–276. ACM, 2009.

[29] Jiahui Jin, Junzhou Luo, Aibo Song, Fang Dong, and Runqun Xiong. Bar: an efficient data locality driven task scheduling algorithm for cloud computing. In *Proceedings of the 2011 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pages 295–304. IEEE Computer Society, 2011.

[30] J. Kwak, E. Hwang, T. K. Yoo, B. Nam, and Y. R. Choi. In-memory caching orchestration for hadoop. In *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, pages 94–97, May 2016.

[31] E Larour, H Seroussi, M Morlighem, and E Rignot. Continental scale, high order, high spatial resolution, ice sheet modeling using the ice sheet system model (issm). *Journal of Geophysical Research: Earth Surface*, 117(F1), 2012.

[32] Haoyuan Li, Ali Ghodsi, Matei Zaharia, Scott Shenker, and Ion Stoica. Tachyon: Reliable, memory speed storage for cluster computing frameworks. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 1–15. ACM, 2014.

[33] Message Passing Interface Forum. Message Passing Interface. `https://www.mpi-forum.org/`.

[34] Arun C. Murthy, Vinod Kumar Vavilapalli, Doug Eadline, Joseph Niemiec, and Jeff Markham. *Apache Hadoop YARN: Moving Beyond MapReduce and Batch Processing with Apache Hadoop 2.* Addison-Wesley Professional, 1st edition, 2014.

[35] Chuck Pheatt. Intel® threading building blocks. *Journal of Computing Sciences in Colleges*, 23(4):298–298, 2008.

[36] Kai Ren, YongChul Kwon, Magdalena Balazinska, and Bill Howe. Hadoop's adolescence: An analysis of hadoop usage in scientific workloads. *Proc. VLDB Endow.*, 6(10):853–864, August 2013.

[37] Zujie Ren, Jian Wan, Weisong Shi, Xianghua Xu, and Min Zhou. Workload analysis, implications, and optimization on a production hadoop cluster: A case study on taobao. *IEEE Transactions on Services Computing*, 7(2):307–321, 2014.

[38] Avraham Shinnar, David Cunningham, Vijay Saraswat, and Benjamin Herta. M3r: Increased performance for in-memory hadoop jobs. *Proc. VLDB Endow.*, 5(12):1736–1747, August 2012.

[39] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. Hive: a warehousing solution over a map-reduce framework. *Proceedings of the VLDB Endowment*, 2(2):1626–1629, 2009.

[40] David Van Der Spoel, Erik Lindahl, Berk Hess, Gerrit Groenhof, Alan E Mark, and Herman JC Berendsen. Gromacs: fast, flexible, and free. *Journal of computational chemistry*, 26(16):1701–1718, 2005.

[41] Matei Zaharia, Dhruba Borthakur, Joydeep Sen Sarma, Khaled Elmeleegy, Scott Shenker, and Ion Stoica. Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling. In *Proceedings of the 5th European Conference on Computer Systems*, EuroSys '10, pages 265–278, 2010.

[42] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 2–2. USENIX Association, 2012.

[43] Qi Zhang, Sai Qian Zhang, Alberto Leon-Garcia, and Raouf Boutaba. Aurora: Adaptive block replication in distributed file systems. In *Distributed Computing Systems (ICDCS), 2015 IEEE 35th International Conference on*, pages 442–451. IEEE, 2015.

[44] Xiaohong Zhang, Zhiyong Zhong, Shengzhong Feng, Bibo Tu, and Jianping Fan. Improving data locality of mapreduce by scheduling in homogeneous computing environments. In *2011 IEEE Ninth International Symposium on Parallel and Distributed Processing with Applications*, pages 120–126. IEEE, 2011.