Quartet: Harmonizing task scheduling and caching for cluster computing

Francis Deslauriers, Peter McCormick,
George Amvrosiadis, Ashvin Goel, Angela Demke Brown
University of Toronto

Abstract

Cluster computing frameworks such as Apache Hadoop and Apache Spark are commonly used to analyze large data sets. The analysis often involves running multiple, similar queries on the same data sets. This data reuse should improve query performance, but we find that these frameworks schedule query tasks independently of each other and are thus unable to exploit the data sharing across these tasks. We present Quartet, a system that leverages information on cached data to schedule together tasks that share data. Our preliminary results are promising, showing that Quartet can increase the cache hit rate of Hadoop and Spark jobs by up to 54%. Our results suggest a shift in the way we think about job and task scheduling today, as Quartet is expected to perform better as more jobs are dispatched on the same data.

1 Introduction

Cluster computing frameworks such as Apache Hadoop [4] and Apache Spark [5] are commonly used to run a variety of data analytics applications, helping uncover correlations and trends in large data sets. These frameworks allow users to focus on their particular data analysis problem, while handling the complexities of distribution, including data placement and replication, computation placement, fault tolerance, and resource negotiation in a shared cluster.

Unfortunately, the exponential growth in the volume of data collected and used for analysis [13] has not been matched by corresponding increases in hard disk performance, where this data is primarily stored [7]. As a result, performing analysis on even the most recent part of a dataset may be limited by disk access speeds. Furthermore, a common analysis pattern involves running multiple, similar queries over the same data. For example, during typical exploratory analysis of a dataset, a data scientist may dispatch multiple queries that differ marginally from each other. Evidence of this work pattern is seen in a recent study of three academic Hadoop clusters by Ren et al., which found that less than 10% of input files are responsible for more than 80% of all accesses, and that only 1% of datasets are shared across users [12]. They further found that 90% of data re-accesses happen within 1 hour, and that 35-60% of job pipelines[1] are submitted within 10 seconds after an earlier pipeline from the same user finishes executing. Similar results are reported by Chen et al. [8] for production workloads of Cloudera customers and Facebook. In other words, users commonly submit a series of jobs that access the same input datasets in a relatively short period of time. Similarly, in production settings, multiple data scientists may run independent analyses that access a large portion of a common dataset concurrently.

Users of cluster computing frameworks enjoy limited benefit from this data sharing today. MapReduce, for example, only considers disk and rack locality when making task placement decisions. It does not take memory locality into account, possibly because it assumes that massive data sets will be displaced from memory before they can be reused. The more recent caching support added to the Hadoop Distributed File System (HDFS) requires users to explicitly identify files that should be pinned in memory; these pinned files are then also considered during task placement. This explicit cache management, however, both increases the burden on users and is ineffective when the reuse occurs across multiple jobs operating on datasets that are larger than available memory. Users similarly expect caching to matter little and tend to run jobs serially even when they are not dependent on the results of the previous jobs [12].

Another challenge for exploiting data reuse in current frameworks is that the order in which jobs process data is agnostic to the data availability in the cache, which can lead to unnecessary cache evictions and disk accesses. As a result, even when the inputs of two jobs overlap completely, if they are scheduled even slightly apart in time they can miss opportunities for sharing data accesses. In our cluster, scheduling two identical Spark jobs 4 minutes apart increases their runtime by 145% compared to starting them together.

We argue that jobs on frameworks such as Hadoop and Spark should be able to benefit from data reuse when jobs share any portion of their input. We leverage the fact that tasks within a job are independent of each other, which is a core requirement in these frameworks for scalability and for task re-execution to mask failures and stragglers.

[1] A pipeline is defined as a sequence of jobs where the output of one job forms the input of the next job in the pipeline.

Our insight is that we can reorder tasks within a job to prioritize the processing of data brought into memory by other jobs, without affecting correctness. To inform tasks of data available in memory we use Duet, an in-kernel framework that exposes page cache information to applications by notifying them about changes in the page cache state, such as a page being added to, or removed from, the cache [1]. As a result, we can harmonize task scheduling with caching in these frameworks.

This paper makes the following contributions. First, we propose changes to existing cluster computing frameworks so that users can benefit from inter-job data sharing. We describe the design of our system, Quartet, and apply it to both Hadoop MapReduce and Spark. Second, we present promising preliminary results that show Quartet can effectively exploit data reuse with low overheads, reducing job runtimes and disk contention.

In the next section, we provide necessary background on the architecture of cluster computing frameworks and Duet. Section 3 reviews related work. We describe the design of Quartet in Section 4, and evaluate its performance in Section 5. Finally, we conclude in Section 6.

2 Background

This section provides background on Hadoop and Spark, as well as an introduction to the Duet framework.

Hadoop. Hadoop is a distributed processing system that aims to take advantage of the computing resources of a cluster while being easy to understand and use [4]. Its main components include: (i) the Hadoop Distributed File System (HDFS), (ii) the YARN resource management platform, and (iii) the MapReduce application.

HDFS is a distributed file system where each file is split into blocks, typically 128 or 256 MB in size, with each block replicated to at least 3 different nodes in the cluster to ensure availability in case of node failure or network partitions. An HDFS installation consists of at least one centralized Name Node service that manages all file metadata and block location information, while each storage node runs a Data Node service to manage the blocks stored in the local file system.

The YARN resource management platform dictates how resources are allocated and shared across user jobs. At the core of YARN lies the Resource Manager, which keeps track of all currently running applications and ensures that each receives their fair share of the cluster. Each worker node runs a Node Manager, which is responsible for managing local computation slots, called Containers, and for reporting usage statistics to the Resource Manager. Applications on YARN are run by submitting a request to the Resource Manager to allocate a container in order to launch a designated Application Master, which can in turn request additional containers and resources on behalf of a job. Typically a worker node will take on both the HDFS Data Node and YARN Node Manager roles, enabling computations to be performed on the same physical machine as the input data.

Hadoop MapReduce is an implementation of MapReduce, a popular model of distributed computation [9], as a YARN application. A MapReduce job consists of many smaller tasks that are either mappers or reducers. Mappers read their input data from storage (typically limited to one HDFS block) and run the first stage of the computation, while reducers take the output from multiple mappers to create the final result. The MapReduce Application Master requests many containers from the Resource Manager, and attempts to schedule each mapper task to a node that is nearest to the input data. Ideally, the selected node will be one of the 3 replicas that contains that block on its local disk.

Spark. Apache Spark is another popular framework for distributed data processing, designed to support applications with cyclic data flow and in-memory computing, in addition to on-disk computing. A Spark application can be run as a YARN Application Master, on top of Mesos [10], or in standalone mode, which is a simplified cluster management system designed specifically to run Spark jobs. This mode is widely used, especially when provisioning Spark-only clusters on cloud computing platforms. We adopt this version for our work.

While similar in operation to YARN, the standalone mode uses different terminology. Roughly speaking, the Master corresponds to the Resource Manager, the Worker corresponds to the Node Manager, the Driver corresponds to the Application Master, and an Executor corresponds to a YARN container.

In Spark, the application flow is as follows. A Driver connects to a Spark Master, receives Executor allocations on Worker nodes, and schedules tasks to those Executors over the duration of an application. The Driver is responsible for all of the task scheduling and placement logic. One distinguishing feature of Spark is that an Executor runs many tasks for a given Driver within one long-lived JVM process, which greatly reduces the overhead of starting up a new container for each task.

Duet. To enable a given framework to schedule tasks on the basis of cached data, we need to expose cache information to the framework's scheduler. To do so, we exploit Duet, a framework that provides notifications to applications about events such as a page being added to, modified in, or evicted from, a node's memory [1]. Duet is implemented as a Linux kernel module, with hooks in the operating system's page cache. Applications register with Duet to receive notifications on events of interest. We build upon Duet to provide aggregated information about HDFS blocks resident in memory across the cluster, which the Application Master (or Driver) scheduler can use to opportunistically select tasks with cached data for a particular Container (or Executor), allowing them to incur less I/O and complete faster.

3 Related Work

A significant amount of work has focused on improving memory locality in cluster computing frameworks. We restrict our discussion to work most relevant to Quartet.

Ananthanarayanan et al. [2] argue that in modern clusters, accessing data from a remote node incurs marginal overhead compared to accessing it directly on the node storing the data. They predict that scheduling tasks for disk locality will become less important, with focus being shifted to memory locality oriented scheduling.

PACMan [3] is a cache management system for cluster environments, which aims to improve job completion. To achieve this goal, PACMan introduces two cache replacement policies that aim to reduce the impact of stragglers, by scheduling together tasks of a given job that are expected to exhibit high memory locality. Quartet's architecture is inspired from PACMan, but our work focuses on jobs who's input data does not fit in the aggregated memory of the cluster. We aim to take advantage of cached data to reduce disk reads as well reducing pressure on the OS page cache.

The Alluxio project [11] aims to reduce the disk write latency caused by the replication scheme used for fault tolerance in large scale storage systems. The authors argue that replication limits job throughput. Instead, they provide fault tolerance by lazily checkpointing output files to stable storage, keeping track of input and computation information needed to recreate each file if it is lost. Our approach focuses on changing the order in which jobs process their inputs to take advantage of memory contents and reduce read latency.

The HDFS Cache Manager [6] was added so that users can provision a specific amount of the cluster's memory to a cache pool that can serve popular files directly from memory. HDFS ensures that those blocks are resident in memory on a specified portion of the hosts at any time. Our technique does not require the user to specifically provision memory for caching, or to know what files are being shared by a large number of jobs. Quartet opportunistically takes advantage of synergistic workloads without any interactions with the users.

4 Design and Implementation

This section describes the Quartet system architecture, and our modifications to the Hadoop and Spark scheduling components to take advantage of memory locality.

4.1 Architecture

Quartet consists of four components: (i) the Duet kernel module, (ii) the per-node Quartet Watcher service, (iii) the centralized Quartet Manager, and (iv) changes to the task schedulers in Hadoop MapReduce and Spark.

Quartet Watcher and Duet. The Watcher service runs on each worker node in the cluster. Using the Duet API, it tracks all kernel page cache changes related to HDFS blocks. These observations are aggregated and periodically reported to the Quartet Manager as a list of tuples of the form *(hdfsBlockId, totalCachedPages, deltaSinceLastReport)*.

Quartet Manager. The Manager receives periodic reports from the Watchers, and maintains a centralized view of the location and number of memory-resident pages of all cached HDFS blocks across the cluster. Applications register blocks of interest with the Manager that correspond to the input files that they will access. Using the centralized view, the Manager periodically notifies applications about whether the relevant blocks are currently cached on a given node. As applications make progress, they unregister interest in completed blocks, minimizing the size of these updates.

Application Master / Driver. We modified the Hadoop MapReduce Application Master and Spark Driver to take advantage of the memory locality information offered by the Quartet Manager. These systems currently schedule tasks such that the input blocks are processed from the start of the file. We changed them so that at job submission time, the HDFS blocks that will be read by the tasks of a job are registered with the Quartet Manager. We then use updates from the Manager to reorder the scheduling of these tasks so that tasks with in-memory blocks are executed before tasks whose blocks are not cached currently (see Section 4.2). The information from the Manager is periodically refreshed so that jobs can have an up-to-date view of the cluster.

While Hadoop and Spark are different projects, written in different languages, their internal structures were similar enough to allow these changes to be implemented in less than 500 lines of code for each. We believe that applying the idea of Quartet task reordering to other systems should require similar effort.

4.2 Task Scheduling

MapReduce and Spark refer to a task as being *node-local* if it can read its input block from a local disk of the worker node on which it is run. We refer to a task as *memory-local* if it is both node-local and its input block is memory-resident on that same node. The goal of Quartet is to exploit all opportunities to run memory-local tasks. It does this by preferring to schedule memory-local tasks ahead of merely node-local ones whenever possible.

In both Hadoop and Spark, once their task schedulers have been granted an allocation to launch a task on a node N, they search for a suitable candidate to launch. Under Quartet, task T will be scheduled to N if 1) T is memory-local to N, or 2) T is node-local to N but not currently memory-local to any other node in the cluster. The memory-local status of all candidate tasks will be checked in Step 1 before proceeding to Step 2. Finally, if neither of these conditions are met for all T, we fall back to the default delay scheduling policy [14].

Step 1 ensures that the application will take any memory-local opportunities available to it, thus avoiding unnecessary reads, disk contention and page cache evictions. Step 2 helps to ensure that redundant disk reads are avoided, but that forward progress is still made if a task is currently not memory-local anywhere.

5 Evaluation

This section evaluates the benefits of Quartet by measuring the improvement in the cache hit rate and runtime of jobs that are scheduled after jobs working on the same dataset. We pick this scenario as representative of real-world workloads reported in the literature [12].

5.1 Setup

Our cluster consists of 24 worker nodes, each configured with an 8-core Intel Xeon L5420 CPU, 16 GB RAM and a 1 TB 7200 RPM hard drive. In order to use Duet, these nodes ran a modified Linux 3.13.6 kernel. A separate, identical node was dedicated to running the HDFS Name Node and Quartet Manager services, in addition to the YARN Resource Manager for Hadoop (resp. the Spark Master for Spark). We configured HDFS with three replicas and 128 MB blocks. Each worker node was allocated 8 concurrent YARN containers (resp. Spark Executors).

The vanilla Hadoop version was 2.7.1, and vanilla Spark was 1.6.0. The Quartet modifications were made to each of these versions.

5.2 Experiments

We measure the effectiveness of Quartet through an experiment based on a real-world workload. Ren et al. report that 90% of data re-accesses occur within one hour of the last access on two different Hadoop clusters [12]. To approximate this scenario, we run two jobs that access

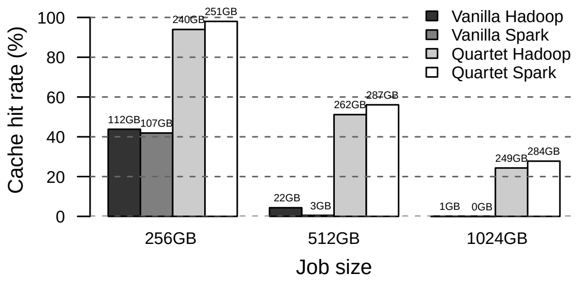

Figure 1: Percentage of HDFS blocks read from cache (hit rate) for the second of two identical jobs.

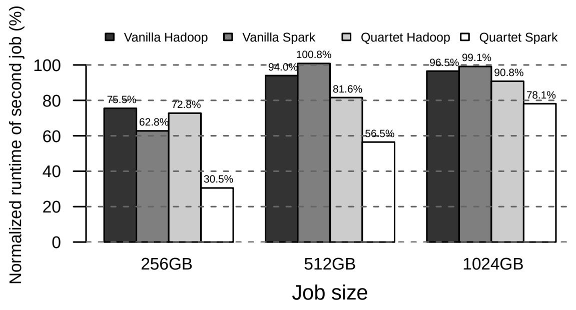

Figure 2: Runtime of the second job, normalized by the runtime of the first, identical job.

the same dataset sequentially. We repeated our experiment using three files of different sizes: 256 GB, 512 GB, and 1 TB. The first of these fits completely within the physical memory of the cluster (384 GB) while the latter two exceed it. For our experiments, we used a custom line counting application for both Spark and Hadoop, to simulate an I/O-bound workload.

Disk accesses. Figure 1 shows the percentage of data accesses that were satisfied from the cache, for the second of two identical jobs. We show results for both Hadoop and Spark, with and without Quartet. As expected, for 256 GB jobs most of the blocks are still in the page cache when the first job finishes. Quartet can take advantage of that fully, demonstrating cache hit rates of 92-98%. In the case of vanilla Hadoop and Spark, however, cache hit rates are as low as 42-44%, because task scheduling is influenced by the timing of resource availability reports from the worker nodes. For job inputs larger than the cluster memory, such as 512 GB and 1 TB, part of the input data was already evicted from the cache by the end of the first job. This, coupled with replica selection in HDFS results in less than 4% cache hit rates. When Quartet is enabled, however, resident blocks are prioritized, and cache hits rates of 25-56% are possible.

Runtime. Reducing the amount of I/O for I/O-bound jobs is also expected to reduce their runtime. Figure 2 shows the runtime achieved by a job, once an identical job has finished running. Overall, we find that Quartet improves most on Spark job runtimes, because it reuses Executors, while Hadoop launches new JVM containers for each task. The cost of container setup and teardown is significant (2-6 seconds), dwarfing the runtime improvements made by Quartet and putting more pressure on the page cache. More specifically, when job data fits entirely in memory the runtime of the second job can be reduced by an additional 45% for Spark and 2.7% for Hadoop using Quartet, compared to the vanilla versions of both frameworks. When the job data exceed our memory capacity, Quartet on Spark improves on job runtime by an additional 21-43%, while Quartet on Hadoop improves runtimes by 6-13%.

Overhead. On each of the worker nodes, our prototype Watcher implementation adds less than 20% CPU overhead on a single core, while the Manager itself consumed less than 5% CPU usage on a single core. The network traffic between the watcher, the manager, and the applications is proportional to the number of HDFS blocks with page cache updates, and the update rate requested by the application. In our experiments updates are requested once per second, and this traffic is in the order of 10-100 KB/s per application.

6 Conclusion

We presented Quartet, a system based on enhancements applicable across cluster computing frameworks. Quartet leverages information on cached data to schedule together tasks that operate on the same data. We have implemented Quartet on both Hadoop and Spark, and our preliminary results show that when jobs overlap, Quartet can almost eliminate I/O of subsequent jobs depending on memory capacity. We believe that our results suggest a shift in the way we think about job and task scheduling today, as Quartet performs better with more jobs being dispatched on the same data concurrently. We are also investigating scenarios where Quartet is used on different frameworks accessing the same data concurrently.

The performance improvement achieved by using Quartet depends on the amount of data reuse, and the timing of re-accesses in a given workload. Analyses of production workloads have shown that re-accesses tend to occur close in time, so we are optimistic that our design will be applicable in those cases. On that end, we are currently evaluating Quartet for more complex workloads, building on earlier work on workload characterization for cluster computing frameworks. So far, however, we have met a shortage of real-world traces that can be used to evaluate our prototype. Due to the nature of our approach, we require an understanding of the data sharing that occurs across jobs in the field. To achieve this, we need to characterize the timing and amount of the sharing of data across jobs. Workloads and traces that are currently available publicly either only capture the start time of jobs without a description of the input data, or contain HDFS events without information that would allow us to link them to job scheduling. We hope that this work encourages a discussion on workload tracing for cluster computing frameworks.

References

[1] Amvrosiadis, G., Brown, A. D., and Goel, A. Opportunistic storage maintenance. In *Proceedings of the 25th Symposium on Operating Systems Principles* (2015), pp. 457–473.

[2] Ananthanarayanan, G., Ghodsi, A., Shenker, S., and Stoica, I. Disk-locality in datacenter computing considered irrelevant. In *Proceedings of the 13th USENIX Conference on Hot Topics in Operating Systems* (2011), HotOS'13.

[3] Ananthanarayanan, G., Ghodsi, A., Wang, A., Borthakur, D., Kandula, S., Shenker, S., and Stoica, I. PACMan: Coordinated Memory Caching for Parallel Jobs. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation* (2012), NSDI'12.

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

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

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

[7] Brewer, E., Ying, L., Greenfield, L., Cypher, R., and T'so, T. Disks for data centers. Tech. rep., Google, 2016.

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

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

[10] Hindman, B., Konwinski, A., Zaharia, M., Ghodsi, A., Joseph, A. D., Katz, R. H., Shenker, S., and Stoica, I. Mesos: A platform for fine-grained resource sharing in the data center. In *Proceedings of the 8th NSDI* (2011), pp. 295–308.

[11] Li, H., Ghodsi, A., Zaharia, M., Shenker, S., and Stoica, I. Tachyon: Reliable, memory speed storage for cluster computing frameworks. In *Proceedings of the ACM Symposium on Cloud Computing* (2014), SOCC '14, pp. 6:1–6:15.

[12] Ren, K., Kwon, Y., Balazinska, M., and Howe, B. Hadoop's adolescence: An analysis of hadoop usage in scientific workloads. *Proc. VLDB Endow. 6*, 10 (Aug. 2013), 853–864.

[13] Shaw, J. Why big data is a big deal. *Harvard Magazine 116*, 4, 30–35.

[14] Zaharia, M., Borthakur, D., Sen Sarma, J., Elmeleegy, K., Shenker, S., and Stoica, I. Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling. In *Proceedings of the 5th European Conference on Computer Systems* (2010), EuroSys '10, pp. 265–278.