Piranha: Optimizing Short Jobs in Hadoop

Khaled Elmeleegy
Turn Inc.
Khaled.Elmeleegy@Turn.com

ABSTRACT

Cluster computing has emerged as a key parallel processing platform for large scale data. All major internet companies use it as their major central processing platform. One of cluster computing's most popular examples is MapReduce and its open source implementation Hadoop. These systems were originally designed for batch and massive-scale computations. Interestingly, over time their production workloads have evolved into a mix of a small fraction of large and long-running jobs and a much bigger fraction of short jobs. This came about because these systems end up being used as data warehouses, which store most of the data sets and attract ad hoc, short, data-mining queries. Moreover, the availability of higher level query languages that operate on top of these cluster systems proliferated these ad hoc queries. Since existing systems were not designed for short, latency-sensistive jobs, short interactive jobs suffer from poor response times.

In this paper, we present Piranha—a system for optimizing short jobs on Hadoop without affecting the larger jobs. It runs on existing unmodified Hadoop clusters facilitating its adoption. Piranha exploits characteristics of short jobs learned from production workloads at Yahoo![1] clusters to reduce the latency of such jobs. To demonstrate Piranha's effectiveness, we evaluated its performance using three realistic short queries. Piranha was able to reduce the queries' response times by up to 71%.

1. INTRODUCTION

Nowadays, virtually all major internet companies run clusters of commodity servers to process and store web-scale data—often measured in petabytes. Examples of these data sets include web crawls, click streams, and user profiles.

MapReduce [10] and its open source implementation, Hadoop, have popularized this approach. This is because it is very powerful and cost effective yet fairly easy to use. Currently, tens of companies including Facebook, Google, and Yahoo! run large-scale private MapReduce/Hadoop clusters (to the tune of tens of thousands of nodes) [23]. Moreover, higher level query languages from systems like Pig [21], Hive [24], CQL [8], Jaql [3], and Sawzall [22] running on top of these cluster computing infrastructures have made these systems even easier to use and hence more accessible. All this made most of these internet companies use their clusters both as data warehouses and central processing units.

Although these systems were originally built for large-scale jobs, currently their workloads are dominated by short jobs. For example, in Yahoo!'s Hadoop production clusters, over 80% of the jobs complete in under 10 minutes (Section 3). Similar behavior was reported by Facebook [27] and Microsoft [15]. This came about because over time more people started running ad hoc queries at these clusters instead of just large scale production jobs. Then, the emergence of higher level query languages that can run over these clusters made their usage for ad hoc queries very popular.

Each of these small ad hoc queries could run efficiently on a classical DBMS and does not need to use MapReduce or a cluster of machines. However, the aggregate size of the corpus of data all these queries operate on is usually too large to fit in a single DBMS. Hence, the data is usually left in these clusters as they become the de facto data warehouse. Moreover, users want to run their ad hoc queries in place, on the clusters hosting the data.

Unfortunately, these cluster computing frameworks were built without having short jobs in mind. For example, their key design goals were (1) scalability with respect to the job and cluster sizes, (2) reliability: so that jobs are fault tolerant, and (3) throughput. Consequently and according to our experience in production Hadoop clusters, short jobs do not have good response times, although for short interactive ad-hoc jobs, latency is key.

In this paper, we first studied and characterized short Hadoop jobs in a multi-thousand-node Yahoo! production cluster. We learned that these short jobs typically have a small number of tasks. Moreover, these jobs are very unlikely to experience failures. We also learned that a big fraction of these production jobs originate from higher level query languages like Pig, where many queries are compiled into a chain of MapReduce jobs.

Using this information, we developed a new system, Piranha, to optimize Hadoop's short jobs and reduce their response times without affecting the larger jobs. Piranha runs on existing unmodified Hadoop clusters, to facilitate its adoption. Although built for Hadoop, the techniques used in developing Piranha and the lessons learned can be generalized to other cluster computing platforms.

The key optimizations Piranha employs to reduce the latency of short Hadoop jobs are the following:

- Given that short jobs are small and unlikely to suffer from failures, Piranha optimizes for the common case and avoids checkpointing intermediate results to disk. Instead, it reruns the whole job in the very unlikely event of failure. This is acceptable be-

[1]Majority of this work was done, while the author was at Yahoo!Research.

Proceedings of the VLDB Endowment, Vol. 6, No. 11

cause the job is small and short so rerunning it is inexpensive, especially if it is unlikely to happen.

- Piranha's simple fault-tolerance mechanism facilitated extending Hadoop beyond MapReduce to support short jobs having a general directed acyclic data flow graph. This is because the main hurdle against supporting these kinds of jobs is their complex fault tolerance, where support for rollback recovery is needed. Supporting jobs with general directed acyclic data flow graph allows Piranha to run complex queries efficiently instead of having to inefficiently implement them in chains of MapReduce jobs as we will see in Section 3.1.3.
- In Piranha jobs, tasks are self coordinating instead of relying on Hadoop's master for coordination. This is because the master relies on a high-latency polling protocol for coordination as it trades off higher latency for better scalability and reliability as we will see in details in Section 2.2. Consequently, relying on Hadoop's master for coordination introduces significant latency for short jobs.

To demonstrate Piranha's effectiveness, we evaluated it using three realistic short queries. Piranha was able to reduce the queries' response times by 71%, 59%, and 46%.

The remainder of this paper is organized as follows. Section 2 presents the needed background about MapReduce and Hadoop. The motivation for this work is laid out in Section 3. Section 4 presents the design and implementation of Piranha. Then, the effectiveness of Piranha is evaluated in Section 5. Related work is presented in Section 6. Finally, Section 7 concludes this paper.

2. BACKGROUND

In this section, we first provide a brief overview of the MapReduce programming model. We then describe some of the key aspects of the Hadoop implementation of this model as it relates to this paper.

2.1 MapReduce Programming Model

The MapReduce programming model defines a computation step using two functions: `map()` and `reduce()`. The `map` function maps its input set of key-value records to another set of keys and values. Its output is then shuffled to the reduces, where each key and its corresponding values arrive at one reduce sorted by the key. The `reduce` function is then applied to each key group it receives.

While simple, this model is powerful and scalable. It naturally exploits parallelism: First, the input data is split such that the `map` function can be invoked on the various parts in parallel. Similarly, the `reduce` function is invoked on partitions in parallel.

2.2 Hadoop Architecture

Hadoop is an open-source Java-based implementation of the MapReduce programming model. It is designed for clusters of commodity nodes that contain both CPU and local storage and is comprised of two components: (1) a MapReduce implementation framework, and (2) the Hadoop Distributed File System (HDFS). HDFS is designed primarily to store large files and it is modeled after the Google File System [12]. In HDFS, data is striped across nodes with large block sizes (for example, 64/128 MB blocks are common) and replicated for fault-tolerance. The HDFS master process is called the NameNode. It is responsible for maintaining the file system metadata. The slaves are called DataNodes. They run on every node in the cluster storing the data blocks. In typical Hadoop deployments, HDFS is used to store the input to a MapReduce job as well as store the output from a MapReduce job. Any intermediate data generated by a MapReduce job (such as the output from a map-phase) is *not* saved in HDFS; instead, the file system on the local storage of individual nodes in the cluster is used.

In Hadoop's MapReduce implementation, a process that invokes the user's `map` function is called a *map* task; analogously, a process that reads the map output and invokes the user's `reduce` function is called a *reduce* task. The framework addresses issues such as scheduling map/reduce tasks for a job, transporting the intermediate output from map tasks to reduce tasks, and fault-tolerance.

Hadoop's MapReduce includes two key components:

- *JobTracker* process: a single master process to which users submit jobs. For each job, it creates the appropriate number of map/reduce tasks and schedules them for execution on worker nodes. It also keeps track of different jobs' progress to detect and restart failed tasks and also sends progress reports to the user. Moreover, it keeps track of the liveness of all the nodes in the cluster.
- *TaskTracker* processes: slave processes that run on individual nodes in the cluster to keep track of the execution of individual tasks. Each TaskTracker process exposes some number of slots for task execution. A slot is used to spawn and execute a map/reduce task as assigned by the JobTracker. Also, the TaskTracker monitors the liveness of individual tasks running on its node.

A heartbeat protocol is used between these two processes to communicate information and requests. It has different purposes, including (1) detection of dead nodes, (2) assignment of a task for execution by the JobTracker whenever a slot becomes available at any TaskTracker, and (3) tracking the progress of the tasks spawned by the TaskTracker. Progress tracking is used, for example, for the JobTracker to detect the completion of a map task. This information is then communicated to the corresponding reduces so that they fetch the map's output. Also, if a task fails, the JobTracker learns about the failure through this heartbeat protocol. The JobTracker then schedules the failed task for re-execution.

These heartbeats are exchanged periodically, where a lot of information and requests are multiplexed over a single heartbeat. Hence, Hadoop's JobTracker adopts the polling mode opposite to the interrupt mode. This design choice was made to allow this centralized process to scale to a cluster with many thousand nodes, each potentially running tens of tasks. Moreover, it reduces the risk of having the JobTracker overwhelmed due to a spike in load leading to a storm of heartbeats. This is critical as the JobTracker is a main controller of the Hadoop cluster and its dependability is key for the the dependability of the whole cluster. Nevertheless, polling can potentially introduce artificial timing delays, which can hurt response time and overall utilization. But since Hadoop was mainly designed for large jobs, these timing delays would be inconsequential relative to the overall large jobs runtimes. Finally, it is worth mentioning that polling for interrupt mitigation in the context of high performance computing is not new. For example, it is widely used in operating systems [18].

2.3 Map Task Execution

The JobTracker determines the number of map tasks for a MapReduce job by dividing the job input into a set of *input splits*, and then assigns each split to a separate map task. Typically, an input split consists of a byte-range within an input file (for example, a 128MB block from a HDFS file). For optimal performance, the JobTracker implements a locality optimization. It tries to schedule map tasks close to the nodes where the input split data is located. To allow the JobTracker to utilize this optimization, an input split can also optionally specify the set of nodes where the data is located. This

locality optimization helps in avoiding a network data transfer for reading the map input. The Hadoop framework provides users with the flexibility in how the input splits are generated for each of their MapReduce jobs.

When a map task starts, it parses its input split to generate key/value pairs. The user-provided `map` function is invoked for each key/value pair. The framework handles the management of the intermediate key/value pairs generated by a map task: The data is hash-partitioned by the intermediate key, sorted, and appropriately spilled to disk. Once an input split is fully processed, i.e., the user-provided `map` function has been invoked on all the key/value pairs from the split, execution is complete. Before the map task exits, the framework merges the various spill files on disk and writes out a single file comprising of sorted runs for each partition. As noted in Section 2.2, this merged file is stored in the file system on the local storage of the node where the task executed. Finally, a task completion notification is passed to the JobTracker, which in turn passes it to the job's reduce tasks.

Note that the map outputs are saved to disk and not pushed directly to the reduces for two reasons: (1) Fault tolerance: this is to handle the case of a reduce task failing. When this reduce is restarted, it can directly fetch the map outputs from disk instead of having to rerun all the map tasks. (2) Asynchrony of execution of maps and reduces: In general, a map cannot assume that all reduces will be available when it finishes. This is because a reduce task can get scheduled to run after the completion of some of the map tasks, depending on the slot availability in the cluster. This is especially true for a large job in a busy cluster. Hence, the disk is used as a *drop box* to support this asynchrony of execution.

2.4 Reduce Task Execution

When a reduce task starts, it waits for the map tasks to produce its inputs. Since the map output is stored on the local disk where the map executed, a *rendezvous* mechanism is needed for the reduce to locate the map tasks. The rendezvous between a reduce task and its input is facilitated by a combination of JobTracker and TaskTracker processes, such that the information about a map's completion along with its location is relayed to the reduce. Consequently, the reduce task fetches its inputs—the maps' outputs.

In Hadoop parlance, a reduce task execution comprises of three phases. The first phase, which is this data transfer, is known as *shuffle*. Once a reduce task has pulled all of its input data, it moves to the second phase known as *merge*, where all the inputs are merged. In the third phase, known as the *reduce* phase, the user-provided `reduce` function is invoked on each key group. Once all the groups have been processed, the reduce task execution is complete.

In Hadoop, the output from the reduce task is then materialized by saving to HDFS. When all reduce tasks complete execution, the MapReduce job execution is complete and its output is available for consumption.

3. MOTIVATION

In this section, we motivate the need for the Piranha system to optimize short jobs using information collected from the Yahoo! production clusters.

3.1 Workload Characterization

In this section, we study the job mix in the Yahoo! Hadoop clusters. We show that the majority of jobs are short, having a small set of tasks. These jobs are mostly online ad hoc queries, which are latency sensitive. Moreover, we show that failure rates are very low. Consequently, these short jobs are very unlikely to experience a failure. Furthermore, we point out that these short jobs are usually chained together. These job chains often have redundant stages, which add unnecessary extra latency to the job's execution time.

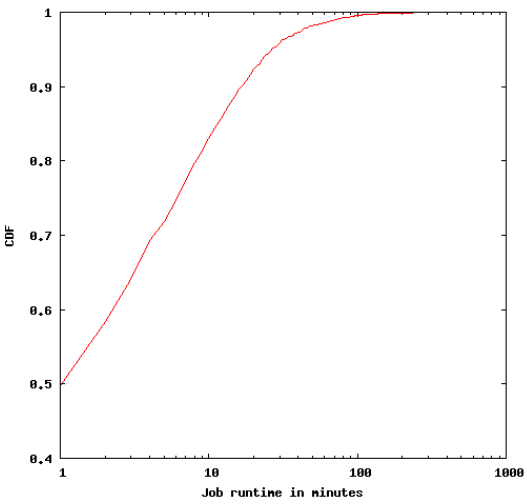

Figure 1: Distribution of overall jobs' runtimes in a production Hadoop cluster.

3.1.1 Job Sizes

To this end, we studied 2 weeks worth of job logs in one of Yahoo!'s few-thousand-nodes busy production Hadoop clusters. Figure 1 shows the CDF of individual jobs' runtimes. These runtimes are execution times and do not include the time a job may incur queued waiting to be executed. Over 80% of the jobs complete execution in under 10 minutes. Given that the vast majority of the jobs execute for a short duration, this paper is mainly concerned with optimizing Hadoop's performance for such jobs. Hence, we restrict the rest of our analysis to these short jobs.

Since these jobs run on a large cluster, runtime by itself does not tell us the full story about the job size. Thus, we analyzed the number of map and reduce tasks per job. Figure 2 shows the distribution in the number of map tasks in short jobs. The median number of maps is 15. Figure 3 shows the distribution in the number of reduce tasks in short jobs. The vast majority of the jobs have a single reduce. This typically arises from ad hoc queries.

The reason people run these tiny jobs on multi-thousand-nodes clusters is because such clusters are used as data warehouses. Consequently, these ad hoc queries, irrespective of their size, run where the data is placed.

3.1.2 Failure Rates

In Yahoo! Hadoop clusters, failure rates are about 1% per month. Assuming inter-failure times follow a poisson process, with a mean time to failure $MTTF$, then the probability of failure of one or more tasks in a job with N tasks, running for a given time, is given by the following equation.

$$P = 1 - (e^{-N \cdot t/MTTF})$$

Now, from Figure 1, the median job has a median runtime of 1 minute and from Figures 2 and 3 it has 15 maps and 1 reduce tasks, a total of 16 tasks. Plugging these values into the above equation, we find that the probability of failures in small jobs is virtually

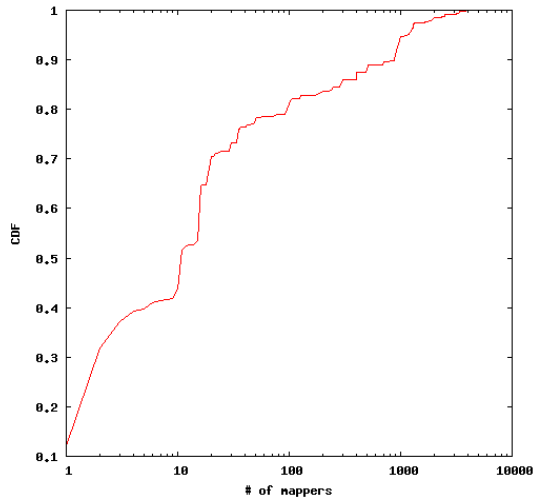

Figure 2: Distribution of number of map tasks per job for short jobs in a production Hadoop cluster.

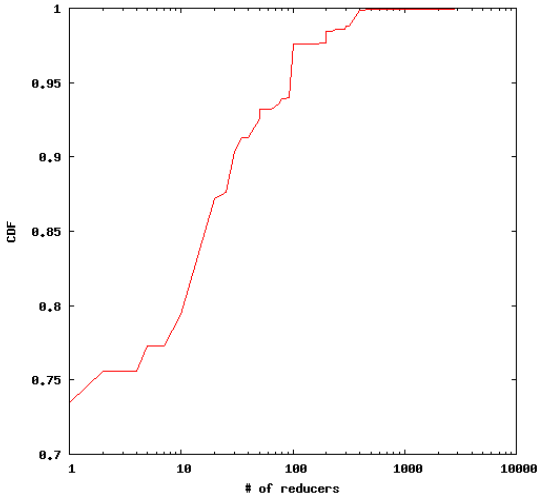

Figure 3: Distribution of number of reduce tasks per job for short jobs in a production Hadoop cluster.

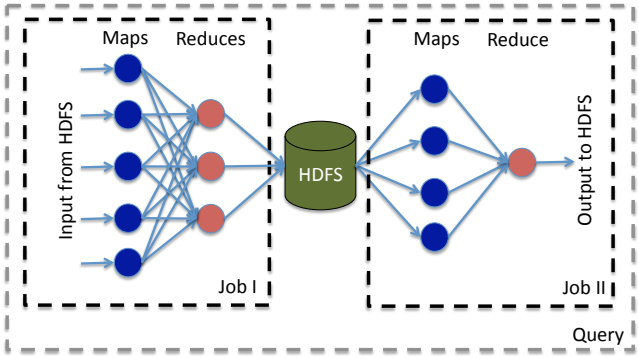

Figure 4: TopK query as a chain of two MapReduce jobs.

zero. This is consistent with our experience from Yahoo! production clusters, where small bug-free jobs virtually never experience failures.

3.1.3 Chained Jobs

In the studied workload, we also observed that over 70% of the Hadoop jobs are generated from translations of higher level, SQL-like, queries by systems such as Pig and Hive. These queries are usually translated into a directed graph of MapReduce jobs chained together. To illustrate this, consider the simple high level query: "What are the top 10 web search queries for today?". This query is translated into two back-to-back MapReduce jobs. The first job computes the frequencies of different search queries by grouping search queries. The second job sorts these frequencies to obtain the K most popular queries. Figure 4 shows a schematic diagram for the execution of the above query broken down into two MapReduce jobs. In practice, these chains are small for short jobs, i.e. they typically have two or three chained MapReduce jobs.

As described in Section 2, Hadoop only supports MapReduce jobs and does not provide native support for job chains. Instead, it treats each step in the chain as an isolated Map-Reduce job. This has two implications for job chains. First, the output of a job step has to be materialized to HDFS before it can be consumed by a subsequent job step. Second, in each step, the map phase has to be invoked to enable data to flow through the Hadoop pipeline. Both of these increase the latency for job chains. Furthermore, the latter may introduce a redundant phase to a job step. For instance, the map phase of the second job in the top K query is an identity function.

3.2 Opportunity

We can leverage the fact that the majority of jobs are short, latency-sensitive ad hoc queries. First, since such jobs rarely fail, checkpointing can be avoided and data can be streamed through tasks without touching the disk. Not only does this reduce the overhead of short jobs, but also it reduces disk contention, helping the overall job population in the cluster. In the very unlikely event of failure of a task, the whole job can be restarted to maintain fault tolerance. This is inexpensive because the job is short. Moreover, avoiding checkpointing substantially simplifies the execution of general computations, whose tasks' execution follow a general Directed Acyclic Graph (DAG) data flow. The main complexity in executing these computations is check-pointing and roll-back recovery. With the support of DAG computations, redundant tasks

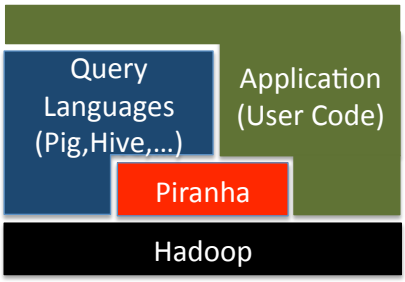

Figure 5: Where Piranha fits in the Hadoop stack

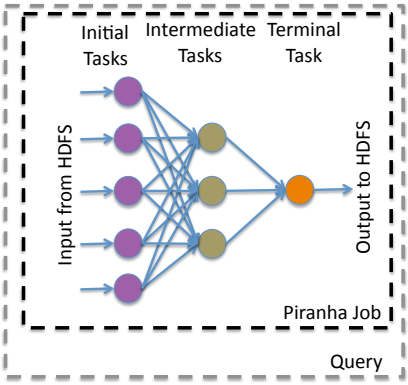

Figure 6: Execution of a top K query as a Piranha job.

can be eliminated from computation pipelines, further reducing the computation's latency.

Finally, since short jobs do not need Hadoop's elaborate fault tolerance infrastructure, for such jobs, relying on the JobTracker with its expensive heartbeats protocol for coordination between job tasks can be avoided. Conversely, tasks can self coordinate to avoid the latency introduced from relying on the JobTracker's polling heartbeats protocol.

4. PIRANHA

In this section we present the design and implementation of a novel system, Piranha, that is designed for short jobs in the cloud. Piranha exploits the special properties of short jobs to optimize their performance. Our implementation is built on top of Hadoop. However, the same concepts can be applied to build Piranha on top of other cloud infrastructures like Mesos [13] and YARN [4].

As shown in Figure 5, Piranha runs as a middleware between user defined jobs and the Hadoop framework. Also, systems like Pig and Hive can compile their high level queries into Piranha jobs to make their execution more efficient.

In the design section, we present Piranha's design concepts independent of Hadoop. In the implementation section, we show how these concepts are adapted to Hadoop.

4.1 Design

A Piranha job has a directed acyclic data flow graph, where the vertices of the DAG represent computations, i.e., tasks and edges in the DAG represent communication between tasks. Figure 6, shows the flow graph of an example query, the TopK query presented in Section 3, implemented using Piranha.

A Piranha task could be one of three types:

- **Initial**: These tasks have no parents in the DAG. Hence, they do not get their inputs from previous tasks in the job. Instead, they typically read it from the underlying file system. They pass their output to their children tasks.
- **Intermediate**: They read their inputs from their parent tasks in the DAG and pass their outputs to their children.
- **Terminal**: These tasks read their inputs from their parent tasks in the DAG. They do not have children. They write their output to the underlying file system.

A Piranha job is self organizing. This means that its tasks coordinate among themselves to drive the job to completion with minimal reliance on the underlying infrastructure. Piranha relies on the underlying infrastructure just for launching and scheduling its tasks. It also relies on the underlying infrastructure to schedule initial tasks close to their input data if the infrastructure supports this feature.

Piranha runtime handles the following key functionalities:

- **Rendezvous between tasks**: Since the underlying scheduling infrastructure launches tasks in arbitrary locations in the cluster, tasks need to locate each other. This is needed so that parents can push their outputs to their children in the DAG. In Piranha, each child task publishes its location to a job-specific location accessible by other tasks in the job. For example, this publishing medium could be the underlying distributed file system or a coordination service like ZooKeeper [14]. Parent tasks read this published information to locate their children. Piranha offloads this rendezvous process from the underlying cluster's scheduler, as it is done in the Hadoop JobTracker. This accomplishes two things. First, it reduces the load on the cluster's scheduler allowing it to scale more. Second, it does not rely on the heartbeats protocol with all the latency it introduces, hence, reducing the job's overall runtime. Finally, offloading the rendezvous to an external system is scalable as more instances of this external system can be added if the load increases, e.g., add more Zookeeper services.
- **Data transfer between tasks**: Piranha assumes that the task's children will be available once the task completes and has its outputs ready. This is achieved by requesting from the underlying cluster scheduling infrastructure to co-schedule tasks in subsequent phases. For example, in Figure 6, the Initial and the Intermediate tasks should be co-scheduled. When the Initial tasks finish, the Terminal task should be scheduled, leaving the Intermediate and the Terminal tasks running. If a task does not have all of its children running, it waits for them to come up before it exits. Hence, to make forward progress, a Piranha job needs the underlying infrastructure to give it as many slots as needed to accommodate the tasks of the biggest two consecutive phases in its DAG. For example, the job in Figure 6 would at least need 8 task slots to accommodate its first two stages so that it can make forward progress. The above assumption is reasonable for small jobs as a small set of slots will be required. The Piranha runtime can check for the minimum number of slots required for a particular job. If the user's slots share on the cluster permits, it can submit the job, else it aborts. This guarantees that no job will be submitted without being able to get the resources it needs.

  Given that Piranha guarantees that a parent and its children always coexist, parents always push their output to their chil-

dren. This is more efficient than the *dropbox* approach, used by Hadoop, for two reasons. First, it does not rely on the expensive heartbeat protocol to learn about the availability of the parent's output. Second, the output is streamed directly from the parent's memory over the network to its children without hitting the disk. Conversely, in Hadoop's MapReduce, maps leave their outputs on disk and later reduces fetch these outputs. Multiple reduces fetch different parts of the outputs at the same time introducing disk seeks and contention among the reduces and the other jobs that access the same disk.

- **Fault tolerance**: Since Piranha jobs are very small, the probability of them failing is very low as shown in Section 3. Consequently, Piranha optimizes for the most common case and adopts a very simple fault-tolerance policy, i.e., it does not do any checkpointing within the job and makes the whole job as its unit of fault tolerance. In other words, if any task fails in a Piranha job, the runtime simply aborts this run, deletes any partial output files, and resubmits the job. This approach has two advantages. First, it avoids the checkpointing performance overhead, where intermediate outputs are materialized to disk. Then, these outputs are deleted when the job finishes. This overhead can be significant as we will see in Section 5 if these intermediate outputs are large. Second, it substantially simplifies the implementation of computations having their data flow following a general DAG as it avoids supporting the complex rollback-recovery for a general DAG.

Using Piranha

For using Piranha to run a particular computation, the computation has to be represented as a Piranha DAG. The common use case is for higher level languages like Pig and Hive to use Piranha to execute their queries. The query language would produce a query plan for the query in question. Then, a plan along with its physical operators is translated into a Piranha DAG, and executed by Piranha.

Discussion

There are three points that we would like to point out:

- First, the design choices that we made for Piranha are not suitable for large jobs, e.g., the lack of fault tolerance. Thus, it is the responsibility of higher level applications, e.g., Pig or Hive, to not compile their queries into Piranha if the query is large. The job size can be checked before submission by inspecting the input sizes and the depth of the processing pipeline. This information is available to the higher level application, before submission time. Hence, it can make an informed decision on whether to compile its query to Piranha or, for large jobs, fall back to MapReduce, where fault tolerance is automatically supported and latency is not critical. This is analogous to query optimizers in classical database systems choosing query plans according to input sizes.
- Second, Piranha is only concerned with optimizing the *execution time* of short jobs. If a Piranha job is submitted to an overloaded cluster, it may incur extended queuing delays, which would inflate the job's turn around time. However, different cluster management infrastructures provide mechanisms to handle this problem—most notably priorities, e.g., Hadoop allows for assigning different priorities to different jobs. Higher priority jobs are executed first. Piranha can give its jobs higher priority to ensure that they are not queued behind larger jobs in busy clusters. This is analogous to operating systems giving higher priorities to realtime and interactive processes.
- Third, Piranha has the limitation of requiring parents and children tasks in the execution DAG to be co-scheduled. In theory, in a busy cluster, a Piranha job may have to wait for a while until free slots in the cluster are secured to accommodate the parents and the children in the DAG. In practice, this is not a problem. Other than the fact that Piranha jobs only have small number of tasks, production clusters typically have thousands of task slots that experience a lot of churn. For example, Yahoo!'s Hadoop clusters have tens of thousands of slots with average task runtime of 20 seconds. Hence, securing a handful of free slots for a high priority job is virtually instantaneous.

4.2 Implementation

Our implementation of Piranha extends Hadoop beyond the MapReduce model to support short jobs having general directed acyclic data flow graph as explained in Section 4.1. We choose Hadoop as our underlying infrastructure as it is widely used in industry (e.g. at Yahoo!, Facebook, and Turn), which would expedite the adoption of Piranha.

A Piranha job is encoded as a set of tasks defined in a configuration file. Each task has multiple attributes including: (1) an ID, (2) the class name representing the computation, and (3) the input splitIDs of children tasks, if this is a non-terminal task. This implicitly also defines the job's intertask data flow graph. Given the configuration file defining the job's execution graph, along with the code for classes to be executed by different tasks, the Piranha middleware executes the Piranha job transparently on top of unmodified Hadoop.

Piranha Tasks

To run a Piranha job on top of stock Hadoop, we need to adapt Piranha's task types with those available in Hadoop. As we observed in Section 2.1, unlike for reduce tasks, the MapReduce programming model does not impose any constraints on map tasks. To illustrate, the input to a map task is configurable (such as, read data from HDFS or over a network socket); similarly, the destination of the output from a map task is also configurable. In particular, it is not necessary that the output of a map task to be passed down the Hadoop sort pipeline and passed to reduce tasks. A Hadoop map task therefore presents the abstraction of a general-purpose task. We build on this observation and adapt the three Piranha's task types to be run *solely* using Hadoop's map tasks. In Hadoop parlance, a Piranha job is a "map-only" job.

For optimal performance, the JobTracker should run the Piranha Initial tasks close to their inputs which resides in HDFS. On the other hand, for the Intermediate/Terminal Piranha tasks, the JobTracker has the flexibility to run them on any available node in the cluster. This is because these latter tasks do not read their input from HDFS.

We accomplish these two objectives by leveraging Hadoop's flexibility in defining input splits on a per task basis. First, Piranha middleware uses the underlying job's input split generator to generate a set of splits for the initial tasks. The job's input split generator provides the locations hosting the data for the input split. The JobTracker uses this information to implement the locality optimization for the Initial tasks. Second, Piranha generates a set of *empty* input splits that correspond to each of the Intermediate- and Terminal- Piranha tasks. An empty input split is a zero-length input split. For the empty input splits, Piranha does not provide any location information. This gives the JobTracker the flexibility to schedule these tasks on any available slot in the cluster.

Rendezvous

For the rendezvous to work, the parent and the child tasks have to be co-scheduled. Hadoop does not give jobs control over which tasks to schedule. Instead, the job is submitted to the Hadoop scheduler and it schedules as many tasks as it can for this job depending on free slot availability. Consequently, in our implementation, we use Hadoop to launch all the job's tasks simultaneously. This is sub-optimal, but it satisfies the Piranha requirements and complies with the underlying Hadoop infrastructure. Hence it allows us to run on unmodified Hadoop clusters, facilitating the adoption of Piranha. If a task starts before its children, it waits until they come up and publish their location. If implemented on top of other frameworks like YARN or Mesos, the Piranha runtime can incrementally request tasks from the framework's scheduler to only run tasks in subsequent phases simultaneously. This will result in less slot usage. Also, it allows jobs make forward progress when slots are scarce in the cluster.

In our implementation, Piranha provides the rendezvous service in two ways, each using a different system.

First, it relies on HDFS. In this case, each child writes its address (IP address + port number) to a file at a job-specific location on HDFS as soon as it starts. The parent polls for these location files written by all its children. As soon as all these files exist, the parent learns the locations of its children, passes its output to them, then it can exit freely. The advantage of using HDFS for this purpose is that it is part of the Hadoop stack, so no external systems are needed. On the downside though, HDFS does not provide asynchronous notification when a file is created. Thus, parents have to resort to polling, which if done at a high rate can overload the HDFS NameNode. Conversely, if polling is done at a low rate, it can add a noticeable latency to the job runtime.

Alternatively, the Piranha middleware uses ZooKeeper for the rendezvous. Briefly, ZooKeeper is a high-performance coordination service that allows distributed processes to coordinate with each other through a shared hierarchal namespace which is organized similar to a standard file system. Unlike a typical file system, which is designed for classical storage, ZooKeeper is designed to store small, hot data, and for low latency. Finally, Zookeeper also includes event notification mechanisms whenever data stored in Zookeeper is modified and thereby lowering latency in detecting changes as well as eliminating the need for application polling. Similar to the HDFS case, each child task writes its address to a well-defined node (for example, a ZooKeeper node that corresponds to the task's id) in the ZooKeeper tree. Each parent task can determine the location of their children by retrieving the information from ZooKeeper. The only downside to the ZooKeeper approach is that the Hadoop cluster needs an extra external component, ZooKeeper, which may or may not be available. In our current implementation, we provide the choice of the rendezvous implementation as a configurable parameter.

Putting It All Together

The Piranha middleware specifies to Hadoop the number of map tasks that are required for execution. This number is the sum of the various task types required for an individual Piranha job. Hadoop by default distributes the job configuration file to all the job's tasks. When a Piranha task starts up, it uses its task ID to index into the configuration file to determine its attributes. It then performs rendezvous actions: A child task publishes its location (IP address + port number) to the rendezvous implementation; a parent task locates its children using the rendezvous implementation and establishes a TCP connection to each child so that the parent can push data. At this point, the rendezvous is complete. All the inital tasks can begin execution. The remaining tasks have to wait until their parents have completed execution. A task execution is complete once the task has consumed all its input and either it has pushed output to its children if any or written the output to HDFS. A child task can begin execution *only* after it receives all its inputs. The job execution is complete when all tasks have completed execution.

5. EVALUATION

In this section we study the effectiveness of Piranha using realistic workloads.

5.1 Experimental Setup

To evaluate the effectiveness of Piranha, we examine the performance of its jobs in isolation of other workloads. Hence, we run our experiments on a private 19-machine test cluster. Although this cluster is small in size relative to clusters used in production, it is adequate for the workload in question as we are primarily focusing on small jobs. The cluster ran stock Hadoop 0.20. HDFS was configured to have a maximum block size of 128 MB. Each machine has an Intel dual core processor, 4 GB of memory and a single hard drive. Machines are connected via a 1 Gb/s network connections. All the machines were running Linux Red Hat 5. One machine ran the Hadoop cluster masters, the JobTracker and the NameNode, while the other 18 machines were workers. Each worker was configured to have 2 map slots and a single reduce slot. Also, workers were using Hadoop's default inter-heartbeat interval of 3 seconds for the polling heartbeat protocol[2]. Finally, for optimum performance, Hadoop tasks were configured to use extra memory to avoid any extra disk writes, e.g., map tasks doing external disk sorts of their outputs in case of shortage of memory.

For coordination between Piranha tasks, i.e., rendezvous, we used Zookeeper in our cluster.

5.2 Workload

For our workload, we used a total of four different jobs. The first job is a dummy job with empty input and no computation. It acts as microbenchmark characterizing the inherent overhead of the underlying Hadoop framework. The other three jobs are different flavors of the top K query from Section 3, which are prevalent in the Yahoo! workload. These queries follow a common idiom of grouping operations followed by sorting. Moreover, they were chosen to have runtimes around one minute, which is the median job runtime in the Yahoo! cluster.

The first query runs over a small subset of Yahoo!'s internal webmap table. Each record in the table is of an internet web page recording many of its attributes, e.g., URL, domain, spam score, anchor text, etc.This working set has 1,000,000 records and is 965 MB in size and spans 8 HDFS blocks. The query computes the top 5 domains having the maximum number of web pages in the data set. We call this query *TopKDomains* query. This query was chosen because it has a relatively small number of groups, i.e., domains. This results in the first grouping sub-query to have low selectivity. Consequently, the intermediate results between tasks in the query execution DAG would be small. This diminishes the costs of materializing these intermediate results to disk. It also diminishes the cost of redundant tasks in the query's DAG as they do little work. On the other hand, this emphasizes the scheduling overheads due to polling in the heartbeat protocol described in Section 2.2.

The second query used was finding the 5 most frequent words in a corpus of wikipedia documents. The data set was 737 MB

[2]This is Hadoop's default value for clusters having 300 workers or less

spanning 6 HDFS blocks. These documents were XML encoded, which introduced significant noise and randomness in the documents. Consequently, the first grouping sub-query will have a large number of groups, i.e., tokens or words. This makes the first subquery have high selectivity. Consequently, this workload is used to emphasize the effects of materializing intermediate results to disk and also the effects of redundant tasks in the query's DAG. We call this query *TopKWords* query.

The third query also ran over the same web-map data set. Conversely, this query performs two consecutive grouping operations followed by a sort. It returns the top 5 anchor words in URLs from domains having average spam score less than or equal to 10. The query first groups URLs by domain filtering out URLs in domains with average spam score ¿ 10. Then, it counts the frequency of each anchor word and finally sorts the frequencies to get the top 5 most frequent words. We call this *NonSpamTopKAnchor* query.

In summary, these last three queries were chosen to exercise the different optimizations Piranha introduces. For example, they have redundant stages that Piranha can eliminate. Also, they have different intermediate output sizes, which helps exploring the benefits of eliminating checkpointing for different kinds of jobs. Moreover, the third query has more stages, leading to longer execution graph, which helps studying Piranha's benefits for this kind of jobs. Finally, all these queries are short. Hence, Hadoop's latency overhead will be pronounced in their runtime.

5.3 Experiments

In our experiments, we first use microbenchmarks to evaluate the basic latency overhead any job experience. Then, we use macrobenchmarks to evaluate the impact of Piranha's optimizations on realistic workloads.

5.3.1 Microbenchmarks

In this section, we study runtime of a dummy job, having empty inputs and no computation. This is to quantify the inherent scheduling and coordination overhead in Hadoop any Piranha job must incur when running on top of Hadoop. The job is run on an idle cluster to guarantee that the measured runtimes do not include any queueing delays. For this dummy job, we use three configurations: a MapReduce job, a map-only MapReduce job and a Piranha job. The MapReduce job shuffles its empty map outputs to its reduce tasks to produce empty outputs. Conversely, the map-only job skips the shuffle and the reduce phases. On the other hand, the Piranha job includes both shuffle and reduce phases. However, it is implemented as a map-only job. So, it avoids the overhead of scheduling reduces and the coordination between maps and reduces.

Table 1 shows the runtimes measured for each configuration. For all configurations, we notice high latency which is mostly due to spawning new tasks, scheduling them, and coordinating among them using Hadoop's expensive heartbeats protocol. The map-only configuration avoids the shuffling and the reduce phase. Hence, it avoids the associated scheduling and coordination overheads leading to the reduction of the runtime from 24 seconds to 17 seconds. The Piranha job, although having both shuffle and reduce phases, it is implemented as a Hadoop map-only job. It uses its light-weight coordination protocol and it does not have its reduce tasks scheduling depend on the execution of the map tasks. Hence, it achieves a runtime similar to the map-only MapReduce job. Note that this microbenchmark establishes a lower bound on the performance gains from using Piranha and a lower bound on the runtime of a Piranha job, when running on top of Hadoop. This is because, a Piranha job is a Hadoop job and hence it is subject to the scheduling overhead any Hadoop job must experience to get slots allocated to it to run its tasks. However, performance gains from using Piranha become more pronounced for more complex queries with longer chains of tasks as shown in the following sections. Finally, this limitation is not fundamental to Piranha. For example, if Piranha runs on top of other frameworks like Mesos [13] and YARN [4] to run its tasks, this overhead could be virtually eliminated and Piranha's relative performance gains would be much more pronounced. However, we choose to run on top of Hadoop for the practical reason of maintaining backward compatibility with the dominant framework in practice.

Configuration:	*MapReduce*	*Map-only MapReduce*	*Piranha*
Runtime (s)	24	17	17

Table 1: Runtimes of a dummy job under different configurations.

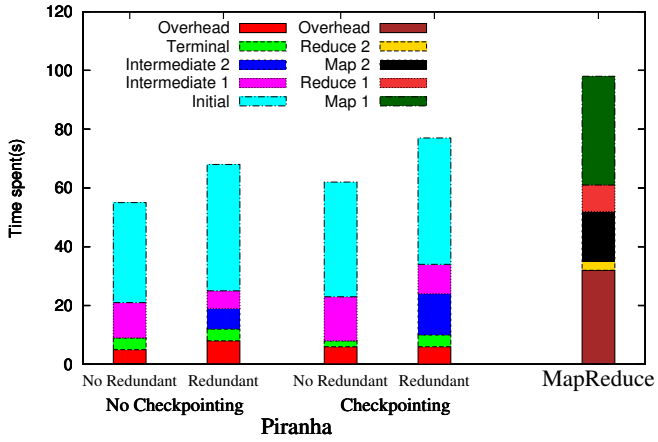

Figure 7: Breakdown of the time spent in the TopKWords query running over a corpus of wikipedia documents, using both Piranha and MapReduce. In this experiment we use four Piranha configurations by varying two variables—checkpointing and the elimination of redundant tasks that the MapReduce execution relies on.

5.3.2 Macrobenchmarks

This section studies the impact of three optimizations Piranha provides to reduce different overheads for short jobs. Specifically, we evaluate the individual contribution of each of the following optimizations on queries' runtimes. In addition, we study the performance of computations having longer chains in their execution graphs using regular MapReduce and Piranha.

Coordination Overhead

In this section, we focus on evaluating the performance gains due to avoiding the reliance on the JobTracker and its expensive heartbeats protocol for coordination. To this end, we run each of the the three queries described above using two setups: (1) Chained MapReduce jobs and (2) A Piranha job having a query execution DAG resembling that of the chained MapReduce configuration with both the redundant tasks and the materialization of intermediate results to disk. Hence, the main difference between the two setups is the inter-task coordination—one relies on the underlying Hadoop

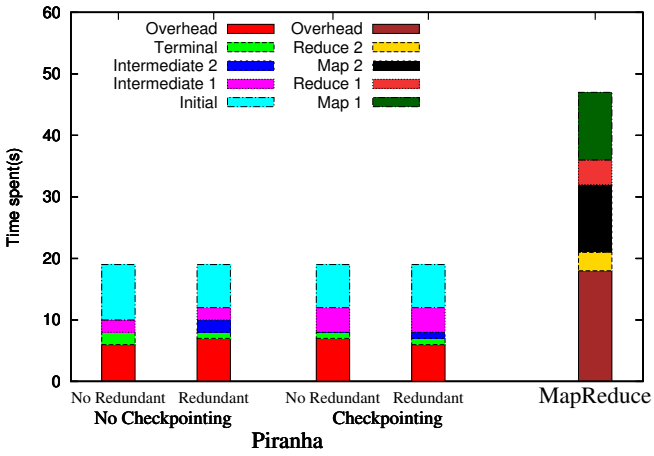

Figure 8: Breakdown of the time spent in the TopKDomains query running over Yahoo!'s web-map table, using both Piranha and MapReduce. In this experiment we use four Piranha configurations by varying two variables—checkpointing and the elimination of redundant tasks that the MapReduce execution relies on.

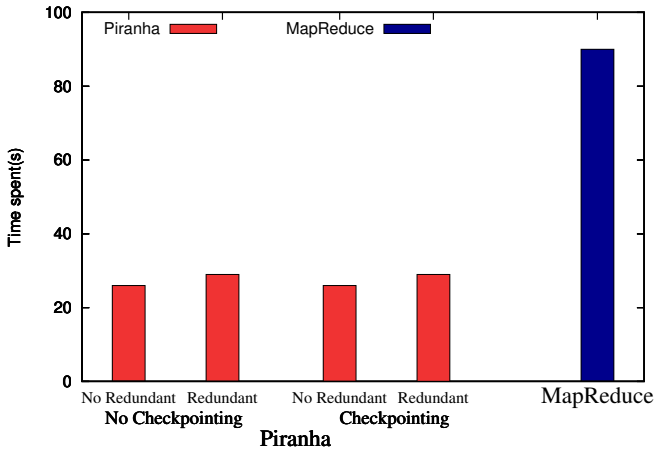

Figure 9: Runtime of the NonSpamTopKAnchor query running over Yahoo!'s web-map table, using both Piranha and MapReduce. In this experiment we use four Piranha configurations by varying two variables—checkpointing and the elimination of redundant tasks that the MapReduce execution relies on.

JobTracker, while the other relies on out-of-band, light-weight coordination using Zookeeper.

Figures 7, 8, and 9 show the runtimes, using both setups, for the *TopKWords* query, the *TopKDomains* query, and the *NonSpamTopKAnchor* query respectively. To be able to compare the effectiveness of different optimizations against each other, each figure includes measurements for different experiments (from this section as well as from next ones) using different configurations. In this experiment we focus on the Piranha configuration having both redundant tasks and checkpointing, shown in the figure, as it resembles the MapReduce execution, with the only difference of eliminating Hadoop's heavy weight coordination overhead. Other Piranha configurations are covered in the following sections.

For the *TopKDomains* query, both setups have a total of 17 tasks structured in 4 stages. In the chained MapReduce setup, the execution DAG is similar to that in Figure 4. The first job has 8 map tasks, a map per HDFS block of the input, plus 4 reduce tasks. The second job has 4 map tasks, operating on the previous reduce outputs, plus a single reduce to do the final ranking. Conversely, in the Piranha configuration there are 8 initial tasks, plus two intermediate stages, each having four tasks, plus a single terminal task. For the *TopKWords* query, there are only six tasks at the first stage reading 6 HDFS input blocks. The following stages are similar to these of the previous query. Finally, the *NonSpamTopKAnchor* has six stages, with the first stage having 8 tasks.

In the figures, each query runtime is broken down by the individual runtimes of each of its stages[3]. Since stages of the query overlap in their execution, we only report execution times of stages on the critical path to the overall query execution. For example, in Piranha configurations, a child task overlaps execution with its parents. We only report the time the child takes after it has received all its inputs from its parents. For the Piranha configurations, we see that the runtime is broken-down into 5 components: (1) the runtime of the Initial phase, (2) the runtime of the Intermediate phase, (3) the runtime of the second Intermediate phase, (4) the runtime of the Terminal phase, (5) and finally other latency overheads. These other latency overheads come about from the underlying Hadoop framework. A big fraction of these overheads come from the polling heartbeats protocol. For the chained MapReduce configuration, we also see that the runtime is broken-down into 5 components: the runtime of the map phase of the first job, the runtime of the reduce phase of the first job, the runtime of the map phase of the second job, the runtime of the reduce phase of the second job, and other latency overheads.

We note that Piranha cuts down the runtimes of the *NonSpamTopKAnchor*, the *TopKDomains* and the *TopKWords* queries by 68%, 59%, and 20% respectively. These savings are chiefly due to lower coordination overhead as parents and children tasks are co-scheduled and communicate directly and not via the JobTracker with its expensive polling heartbeats protocol. For example, in MapReduce, a reduce has to wait for a heartbeat to learn about the completion of a map task before it can fetch its output. Also, tasks are scheduled on heartbeats. So, for chained MapReduce, the last task of the first job has to finish before the second job is submitted. Then, tasks of the submitted job have to wait for heartbeats to be scheduled. Piranha avoids all that. However, it still experiences some latency overhead because it still runs over the Hadoop framework that uses heartbeats. For example, Piranha jobs still have to wait for heartbeats for their tasks to be scheduled.

Finally, there are three more things to note. First, in absolute terms, the savings from cutting the coordination overheads are comparable for both the *TopKWords* and the *TopKDomains* queries as the overheads are proportional to the number of stages in the job. However, since the overall runtime of the *TopKWords* query is significantly longer than that of the *TopKDomains*, the relative gains for the *TopKDomains* query are significantly bigger. Second, there is little variance between the runtimes of similar stages in different configurations. This variance is due to timers' variances as workers do not have synchronized clocks. Hence, different runs can experience different delays because of unsynchronized heartbeats. That

[3]For simplicity of presentation the breakdown is omitted for the *NonSpamTopKAnchor* query as it has several stages.

said, we repeated the experiments three times. These small variances aside, the results are qualitatively the same. Third, we see that the *NonSpamTopKAnchor* query enjoys the maximum performance gains as it has more stages. Piranha eliminates coordinates overheads for all its stages resulting in more gains than the other two jobs.

Checkpointing Overhead

In this section, we study the overhead introduced by materializing the intermediate outputs of tasks for fault tolerance purposes. To accomplish this, we use a different Piranha configuration for the queries we studied before. In this configuration, parent tasks stream their outputs directly to their children in the query execution graph without touching the disk. We refer to this configuration as "Redundant / Checkpointing" in Figures 7, 8, and 9. In this experiment, we compare the runtimes of the new Piranha configuration to both the Piranha configuration having disk materialization and to the MapReduce configuration used in the previous experiment. Interestingly, we notice that only the *TopKWords* query experience a noticeable gain from eliminating disk materialization as this optimization cuts 11% of its runtime in comparison to the earlier Piranha configuration that uses disk materialization. This is because this query has much higher selectivity than the *TopKDomains* and the *NonSpamTopKAnchor* queries leading to much more disk I/O due to disk materialization. More specifically, it has intermediate outputs ranging from 100 MB to 180 MB per task compared to only about 2 MB of intermediate outputs per task for the other two queries. Hence, disk materialization is far more expensive for the *TopKWords* query.

Redundant Tasks' Overhead

In this section, we study the overhead due to redundant tasks, when queries are executed as chained MapReduce jobs. These redundant tasks are imposed on the query execution graph to comply with MapReduce programming model. To this end, we construct two more Piranha configurations for each of the above queries that avoid these redundant tasks, specifically, the second intermediate stage for the *TopKDomains* and the *TopKWords* queries, and the second and the fourth intermediate stages for the *NonSpamTopKAnchor* query. In Figures 7, 8, and 9, we refer to these two new configurations as "No Redundant", while varying doing checkpointing.

In this experiment, we compare the runtimes of the new Piranha configurations to the two previous Piranha configurations along with the MapReduce configuration for the three queries. In Figure 7, we note that eliminating the redundant phase leads to 20% reduction in *TopKWords* query runtime over the corresponding Piranha job having the redundant phase. This is true for both Piranha configurations with and without disk materialization. Conversely, in Figure 8, we note that eliminating the redundant tasks virtually did not result in any noticeable gains. Again, this is because the *TopKDomains* query has very little selectivity. Hence, these redundant, pass-through tasks do not result in noticeable latency. Finally in Figure 9, while the *NonSpamTopKAnchor* query has low selectivity, it also has two eliminated redundant stages, which explains the 10% performance gain.

6. RELATED WORK

Piranha runs on top of Hadoop [1], which is an open source implementation of Google's MapReduce [10]. Both systems run parallel computations on a cluster of machines. They constrain the communication pattern between tasks such that, each map task partitions its output and distributes these partitions across all the reduce tasks. Piranha extends Hadoop to run short computations with general acyclic directed flow graphs. Moreover it is optimized for short jobs to reduce their latency.

Short and interactive jobs have received a lot of attention lately from both industry and academia.

The most relevant system is Tenzing [7], which was developed to reduce jobs' turn-around time on MapReduce clusters. It uses a pool of ever-running worker tasks, where new work is submitted to free existing workers. While using a workers pool avoids the overhead of spawning new tasks, it has two shortcomings. First, reserving more workers than needed wastes the cluster's resources. Conversely, having less workers introduces latency due to waiting for workers to become free. Consequently, a dynamic scheme may be required to set the pool size, which is challenging as the workload can be highly dynamic. Second, constraining the MapReduce scheduler to use specific workers for a particular computation may significantly limit its ability to optimize for data locality (scheduling the job at nodes where the inputs reside locally). This can increase the network load as inputs may be read off the network instead of being read locally, which may also increase the job runtime. Moreover, Tenzing is constrained by the MapReduce model. Hence, it may introduce redundant stages to execution pipelines as it cannot directly execute computations with general DAG data flow. Finally, Tenzing runs on top of MapReduce, which materializes intra-job intermediate results. Piranha's optimizations are complementary to Tenzing's. Hence, the two sets of optimizations can be applied simultaneously.

Shark [25] running on top of Spark [27] promises to make short queries run faster. To accomplish this, Spark tasks load data off disk and cache it in memory and reuses it across different computations/queries. For the memory to be effective, the working set needs to fit in memory. Moreover, queries touching new data will still have to be read off disk. Unlike Piranha, they do not run on top of Hadoop as Spark is a stand-alone system.

Dremel [19] and its open source implementation Drill [2] also attempts to make interactive queries faster. To this end, it reduces the amount of data read by queries using column-store techniques.

Dryad [15] and Hyracks [6], like Piranha, can run a parallel computation with DAG data flow on a cluster of machines. However, both Dryad and Hyracks are standalone systems that do not run on Hadoop or MapReduce clusters. Also, unlike Piranha, neither Dryad nor Hyracks are optimized for short jobs.

CIEL [20] takes things a step further to support jobs with dynamic DAG data flow on a cluster of machines. It is also not optimized for small jobs and it is a standalone system.

DryadLINQ [16,26] compiles higher level languages into Dryad. Similarly, Pig [21] and Hive [24] can compile higher level query languages into Piranha.

Frameworks like Mesos [13] and YARN [4] are introduced as resource managers of a cluster's resources, allowing different applications to share the cluster, e.g. Hadoop and MPI. Piranha can run directly on top of these systems. In this case, Piranha can get a set of slots in the cluster from the underlying resource manager to run its tasks. Thus, it can avoid Hadoop's high scheduling overhead increasing its effectiveness. Moreover, controlling the set of tasks it uses at a given time allows it to use the minimal set of tasks to complete the job hence making its execution more efficient.

Like MapReduce Online [9], Piranha pipelines data between tasks. MapReduce Online also can pipeline data between jobs. However, MapReduce Online jobs comply with the MapReduce programming model and follow its prescribed communication graph.

Hence, it cannot eliminate redundant tasks in a chain of jobs. Moreover, unlike MapReduce Online, Piranha does not modify Hadoop so it can run on unmodified Hadoop clusters.

Improving Hadoop's performance has attracted much attention from the research community [5, 11, 15, 17, 27, 28]. For example, Quincy [15] and Delay Scheduling [27] proposed fair share schedulers to address resource starvation that arises whenever small jobs are executed concurrently with large jobs. Others have focussed on improving Hadoop performance for analytical queries such as join [11, 17].

7. CONCLUSIONS

We have studied Hadoop job execution history logs in production environments. A key lesson learned is that most of the jobs are short and small. These short jobs are interactive ad hoc queries, where response time is critical. Moreover, we found that these short jobs rarely experience failures. Cluster computing platforms in general and Hadoop in particular are not optimized for low latency for these short jobs as they were originally designed for large jobs. Consequently, these short interactive jobs do not experience good response times on these platforms.

We introduced a novel system, Piranha, that is optimized for short jobs without affecting larger jobs. It avoids many of Hadoop's shortcomings for short jobs. More specifically, it extends Hadoop beyond the MapReduce model to provide light-weight execution DAGs for short jobs. This helps in significantly reducing the runtimes of short jobs. Finally, Piranha is transparent to Hadoop and runs on existing unmodified Hadoop clusters. We have experimentally demonstrated the effectiveness of Piranha. Specifically, we have shown reduction of query runtimes by up to 71% when using Piranha.

8. ACKNOWLEDGMENTS

We would like to thank the reviewers for their valuable feedback. We would also like to thank Sriram Rao for his help and support, Russel Sears and Christopher Douglas for the insightful discussions.

9. REFERENCES

[1] Apache Hadoop Project. `http://hadoop.apache.org/`.

[2] Apache Hadoop Project. `http://incubator.apache.org/drill/`.

[3] Jaql. `http://www.almaden.ibm.com/cs/projects/jaql/`.

[4] The Next Generation of Apache Hadoop MapReduce (YARN). `http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/`.

[5] A. Abouzied, K. Bajda-Pawlikowski, D. J. Abadi, A. Silberschatz, and A. Rasin. Hadoopdb: An architectural hybrid of mapreduce and dbms technologies for analytical workloads. In *VLDB*, Lyon, France, 2009.

[6] V. Borkar, M. J. Carey, R. Grover, N. Onose, and R. Vernica. Hyracks: A flexible and extensible foundation for data-intensive computing. In *ICDE 2011 : IEEE International Conference on Data Engineering*, 2011.

[7] B. Chattopadhyay, L. Lin, W. Liu, S. Mittal, P. Aragonda, V. Lychagina, Y. Kwon, and M. Wong. Tenzing a sql implementation on the mapreduce framework. *Proc. VLDB Endow.*, 4:1318–1327, 2011.

[8] S. Chen. Cheetah: a high performance, custom data warehouse on top of mapreduce. *Proc. VLDB Endow.*, 3(1-2):1459–1468, Sept. 2010.

[9] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, and R. Sears. Mapreduce online. In *NSDI'10: Proceedings of the 7th USENIX conference on Networked systems design and implementation*, pages 21–21, 2010.

[10] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. In *OSDI'04: Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation*, 2004.

[11] J. Dittrich, J.-A. Quiané-Ruiz, A. Jindal, Y. Kargin, V. Setty, and J. Schad. Hadoop++: Making a yellow elephant run like a cheetah (without it even noticing). *Proc. VLDB Endow.*, 3(1), 2010.

[12] S. Ghemawat, H. Gobioff, and S.-T. 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.

[13] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *NSDI'11: Proceedings of the 8th USENIX conference on Networked systems design and implementation*, 2011.

[14] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. Zookeeper: wait-free coordination for internet-scale systems. In *USENIXATC'10: Proceedings of the 2010 USENIX conference on USENIX annual technical conference*, pages 11–11, 2010.

[15] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg. Quincy: fair scheduling for distributed computing clusters. In *SOSP*, pages 261–276, 2009.

[16] M. Isard and Y. Yu. Distributed data-parallel computing using a high-level programming language. In *Proceedings of the 35th SIGMOD international conference on Management of data*, SIGMOD '09, pages 987–994, New York, NY, USA, 2009. ACM.

[17] D. Jiang, B. Ooi, L. Shi, and S. Wu. The performance of mapreduce: An in-depth study. *Proc. VLDB Endow.*, 3(1), 2010.

[18] I. Kim, J. Moon, and H. Y. Yeom. Timer-based interrupt mitigation for high performance packet processing. In *In Proc. 5th International Conference on HighPerformance Computing in the Asia-Pacific Region, Gold*, 2001.

[19] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis. Dremel: Interactive analysis of web-scale datasets. In *Proc. of the 36th Int'l Conf on Very Large Data Bases*, pages 330–339, 2010.

[20] D. G. Murray, M. Schwarzkopf, C. Smowton, S. Smith, A. Madhavapeddy, and S. Hand. Ciel: A universal execution engine for distributed data-flow computing. In *NSDI'11: Proceedings of the 8th USENIX conference on Networked systems design and implementation*, 2011.

[21] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: a not-so-foreign language for data processing. In *SIGMOD '08: Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1099–1110, 2008.

[22] R. Pike, S. Dorward, R. Griesemer, and S. Quinlan. Interpreting the data: Parallel analysis with sawzall. *Sci. Program.*, 13:277–298, October 2005.

[23] The Apache Software Foundation. Apache Hadoop – PoweredBy:. At http://wiki.apache.org/hadoop/PoweredBy.

[24] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive: a warehousing solution over a map-reduce framework. *Proc. VLDB Endow.*, 2(2):1626–1629, 2009.

[25] R. S. Xin, J. Rosen, M. Zaharia, M. J. Franklin, S. Shenker, and I. Stoica. Shark: Sql and rich analytics at scale. In *SIGMOD Conference*, pages 13–24, 2013.

[26] Y. Yu, M. Isard, D. Fetterly, M. Budiu, U. Erlingsson, P. K. Gunda, and J. Currey. Dryadlinq: a system for general-purpose distributed data-parallel computing using a high-level language. In *OSDI'08: Proceedings of the 8th USENIX conference on Operating systems design and implementation*, pages 1–14, 2008.

[27] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica. Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling. In *EuroSys '10: Proceedings of the 5th European conference on Computer systems*, pages 265–278, 2010.

[28] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica. Improving mapreduce performance in heterogeneous environments. In *OSDI'08: Proceedings of the 8th USENIX conference on Operating systems design and implementation*, pages 29–42, 2008.