SkewTune Deployment and Evaluation

Jonathon Cai, Department of Computer Science, Yale University

Nowadays, companies, researchers, and governments accumulate large amounts of data that they process using advanced analytics. Big data processing is often achieved through scalable parallel computing frameworks such as MapReduce and Spark. A typical parallel computing job can be described as a workflow consisting of successive computation stages, each stage divided into fixed-sized tasks. Proper resource allocation is crucial for optimally executing large-scale parallel processing jobs. When a machine has available resources, the scheduler allocates the resource to the pending task, matching the available CPU cycles, memory and disk space to the estimated resource requirements of the task. However, this approach has limitations; the actual amount of available resources is inherently uncertain due to multiple tasks often competing for the same resources on the same machine.

The goal of this project is to contribute to a larger one in Wenjun Hu’s group regarding progress-aware job execution in parallel computing. Conventional resource allocation for tasks in parallel programming frameworks is problematic for two reasons: the resource requirements for a task are inherently uncertain, and the amount of available resources visible to the scheduler is also uncertain. In order to address these issues, a task allocation approach, fine-tuning task sizes to fit available resources, was suggested. This strategy is similar to how TCP probes and fills up available bandwidth on a bottleneck link. In order to demonstrate that task allocation performs better than existing methods, it is necessary to test the approach against competing implementations. By evaluating competitors, one gets a sense of the average performance, the implementation complexity, and a better idea of which parts of the parallel computing architecture pertain to resource allocation techniques. I focused on evaluating one existing competitor for the Hadoop framework, a software implementation of MapReduce. The competitor, SkewTune, mitigates skew on Hadoop. I implement various work-arounds and tests to see if results match those described in the original paper. I conclude that the current implementation of SkewTune does not function properly.

1 Background

1.1 Resource Allocation for Parallel Processing

Nowadays, companies, researchers, and governments accumulate and process large amounts of data. Big data processing is often achieved through scalable parallel computing frameworks such as MapReduce [8] and Spark [12]. Conventionally, a parallel computing job is a workflow consisting of successive computation stages, with each stage divided into fixed-sized

1
Proper resource allocation is crucial for optimally executing large-scale parallel processing jobs. Existing approaches to resource mapping generally fall under two categories: request-based and deadline-based resource allocation. Request-based resource allocation fills the requests of tasks in order of their scheduling priority. In the other category, the scheduler allocates resources to jobs based on their deadlines. The performance of these systems depends heavily on the accuracy of resource estimates, which are inherently inaccurate, due to factors such as computational skew, heterogeneous execution environments, and multi-dimensional resources. In addition, once a task is launched, the resource allocated for the task cannot be changed, making adaptation difficult in case the allocation was suboptimal. Hence, in resource allocation schemes, it is hard to mitigate stragglers. It is reported that in large-scale production jobs on Yahoo’s Hadoop clusters, among the jobs with more than 1000 mappers, 5% have stragglers, whereas among those with more than 1000 reducers, as many as 50% do. [1]

A parallel programming system can simultaneously run multiple jobs competing for a node’s resources and traffic bandwidth. These conflicts cause slowdown in the execution of tasks. The overall job completion time is bounded by the time of the slowest task, or straggler, so in order to make the job run as quickly as possible, all tasks should finish around the same time.

1.2 Project Context

The goal of this project is to contribute to a larger one in Wenjun Hu’s group regarding progress-aware job execution in parallel computing. Conventional resource allocation for tasks in parallel programming frameworks is problematic for two reasons: the resource requirements for a task are inherently uncertain, and the amount of available resources visible to the scheduler is also uncertain. In order to address these issues, a task allocation approach, fine-tuning task sizes to fit available resources, was suggested. This strategy is similar to how TCP probes and fills up available bandwidth on a bottleneck link. In order to demonstrate that task allocation performs better than existing methods, it is necessary to test the approach against competing implementations. By evaluating competitors, one gets a sense of the average performance, the implementation complexity, and a better idea of which parts of the parallel computing architecture pertain to resource allocation techniques. I focused on evaluating one existing competitor for the Hadoop framework, a software implementation of MapReduce. The competitor, SkewTune, mitigates skew on Hadoop.

1.3 MapReduce

MapReduce is a framework designed for easily writing applications that process vast amounts of data on clusters of commodity hardware, with potentially thousands of nodes, in a reliable, fault-tolerant manner. A distributed file system is set up across all nodes in a cluster. The data can then be processed in parallel, using the semantics of mapping and reducing from functional programming. Mapping tasks map input key/value pairs to a set of intermediate key/value pairs. The mappers are the individual tasks that transform input records into intermediate records. The key space of the intermediate map outputs can be partitioned
using a Partitioner class; the number of partitions is equal to the number of reducer tasks. All intermediate values associated with a given output key are grouped by the framework and passed to reducer tasks, which determine the final output. The reducer task has three phases: a shuffle, sort, and reduce phase. In the shuffle phase, the reducer task fetches the relevant partition of the intermediate mapper output. In the sort phase, the intermediate mapper outputs are merged by key (the shuffle and sort are performed simultaneously). In the reduce phase, a particular operation (called the reduce operation) is performed on the <key, (list of values)> pairs for the grouped inputs.

The MapReduce framework operates exclusively on <key, value> pairs. That is, the framework views the input to a job as a set of <key, value> pairs and produces a set of <key, value> pairs as the output of the job, possibly of different types. The flow of operations can be viewed as such: (Input) <k1, v1> -> map -> <k2, v2> -> partition -> <k3, v3> -> reduce -> <k3, v3> (Output). The number of mappers and reducers is determined by the data size and user-supplied parameters for splitting the data.

It is important that parallel data processing systems like MapReduce minimize global synchronization overhead to ensure high performance and produce incremental results when possible. More specifically, in MapReduce, reducers are allowed to start copying data before previous mappers finish execution. Additionally, new MapReduce extensions strive to further facilitate pipelining during execution. [10]

1.4 Skew

MapReduce is a popular data processing tool, but it still has important limitations. In particular, skew is a significant challenge in many applications executed on this platform [10]. When skew arises, some partitions of an operation take significantly longer to process their input data than others, slowing down the entire computation. Load imbalance can occur during either the map or reduce phases. Such imbalanced situations are referred to as map-skew and reduce-skew, respectively. There are several reasons why skew can occur within a task. One common type of skew is skew caused by an uneven distribution of input data to operator partitions (or tasks). In MapReduce, this type of skew occurs during the partitioning phase. Certain partitions might have more keys than others; for example, in an inverted index application, the Partitioner class could partition keys based on the first letter of a word. Due to Zipf’s law, some partitions would have many more keys than others.

Another common type of skew occurs when some portions of the input data take longer to process than others. This could occur during both mapper and reducer tasks. For example, map-skew can arise in the PageRank algorithm. PageRank is a link analysis algorithm that assigns ranks to each vertex in a graph by iteratively aggregating the weights of its neighbors. Vertices with larger outdegree take longer to process because the mapper generates a tuple per outgoing edge. Similarly, reducer tasks take longer to process expensive key groups.

2 SkewTune

SkewTune is one approach to solving the problem of skew. The framework attempts to decrease the overall running time of a job by partitioning a straggler into several smaller
tasks, called mitigators. An ideal scenario is illustrated in Figure 1. SkewTune detects that T2 is a straggler task at time point $t_{1}$ and repartitions the remainder of task T2 into mitigators, so that the aggregate time over each slot is approximately the same. The approach SkewTune takes to repartitioning stragglers is conservative. It performs late skew detection, meaning it only repartitions a task if there is an available slot in the cluster. Stragglers are identified by checking if $t_{\text{remain}} > \omega$, where $\omega$ is the repartitioning overhead and $t_{\text{remain}}$ is the time remaining in the task. $\omega$ is hard-coded as 30 seconds, an estimate for SkewTune overhead. At most one task is repartitioned at a time, as one repartitioning could occupy the entire cluster. When a straggler is identified, SkewTune records the input record the straggler has stopped on and repartitions the remaining input data for the straggler into key intervals, assigning the key intervals via range partitioning to mitigator tasks. Range partitioning is used in order to preserve the original output order of the task. It is evident that in this approach, accurate time estimates regarding the remaining time are crucial in order to execute the scheme. However, as machine environments fluctuate, it could be that the machine environment is drastically different from the point the time estimate is assigned. Hence, the estimate is no longer applicable.

The key features of SkewTune are the following. It mitigates two common types of skew described earlier: skew due to uneven distribution of input data to partitions and skew due to expensive records. SkewTune optimizes unmodified MapReduce programs, meaning that programmers do need not change a single line of their code, Hence, SkewTune supports developer transparency. It also preserves interoperability with other user-defined operations, guaranteeing that the output of an operator consists of the same number of partitions with data sorted in the same order within each partition as an execution without SkewTune. Further,
thermore, SkewTune is compatible with pipelining optimizations proposed in the literature, and it does not require any synchronization barrier between consecutive operators.

It should be emphasized that one of the main advantages of SkewTune is developer transparency. If a developer wants to use the SkewTune framework, he does not need to modify his MapReduce program much from the original. This contrasts with other skew mitigation techniques, which require the use of special templates [7] or user-specified cost functions [9].

3 SkewTune Deployment

3.1 Deployment of Hadoop 0.21.0

Hadoop is a popular software implementation of MapReduce, written mostly in Java. SkewTune is a library built on top of Hadoop. Although the SkewTune paper [10] states that it uses Hadoop version 0.21.1, there is no publicly available version of 0.21.1 online in the historical archive of Hadoop releases [1], so instead, I deployed SkewTune on Hadoop version 0.21.0. Furthermore, 0.21.0 is emitted when compiling the SkewTune jar – the file emitted is `hadoop-mapred-0.21.0-SKEWTUNE.jar`, so I assume the authors made a typo in their paper.

I deploy a Hadoop instance on a set of four machines, with one node designated as the master, and all of the nodes designated as slave workers. The master is called `master`, while the slaves are called `slave1`, `slave2`, and `slave3`.

The master node coordinates the workers. The workers execute the tasks associated with a client program. The nodes are all connected locally via Ethernet. In order to ensure that the cluster is working, one must continuously inspect the Hadoop logs and run `jps` repeatedly to make sure all Hadoop daemons are running properly. On the master node, three Hadoop daemons must be running at all times – the NameNode, the SecondaryNameNode, and the JobTracker. On all the slave nodes, two Hadoop daemons, the TaskTracker and the DataNode must be running. Hadoop version 0.21.0 utilizes a simple architecture: user programs submit jobs to the JobTracker on the master node. The JobTracker talks to the NameNode to determine the location of the relevant data files and locates TaskTracker nodes with slots at or near the data. The JobTracker then submits work to selected TaskTracker nodes, which are monitored via heartbeat signals. A TaskTracker notifies the JobTracker when a task fails; the JobTracker then decides what to do afterwards, such as resubmitting the job elsewhere or blacklisting the TaskTracker. [3] It should be noted that Hadoop 0.21.0 is an old version of Hadoop, and the current Hadoop version, 2.7.0 as of this writing, uses an architecture (YARN) that is substantially different. Hence, many issues I troubleshooted may no longer be relevant for new versions of Hadoop.

I used an online tutorial [5] to set up an instance of Hadoop 0.21.0. Along the way, I ran into a significant bug while reformatting the file system and restarting the cluster [2]. I wrote my own script `hadoop/namespace.sh` to quickly resolve the problem of incompatible namespace ID’s, and this issue shows up frequently.

Logs are stored in the `hadoop/logs` directory. In order to debug issues, it is necessary to inspect these logs carefully. Hadoop uses the log4j
logging system [6]. It is possible to set a global logging level such as INFO, DRFA, or DEBUG (or a combination of the former settings) in the hadoop/bin/hadoop-config.sh script, by altering the line `HADOOP_OPTS="$HADOOP_OPTS -Dhadoop.root.logger=$HADOOP_ROOT_LOGGER:-INFO,DRFA"`. In particular, I found it necessary to set the logging level to DEBUG, in order to view debugging statements; however, if this setting is applied globally, the system generates debugging statements for every file, including statements for all the general classes in the Hadoop framework. This generates extraneous statements which clog the logging file; furthermore, they are not useful for debugging issues particular to SkewTune. In order to control the specificity of the logging for particular files, I added a SkewTune-specific log4j.properties file to hadoop/etc/hadoop. In that log4j.properties file, I added lines like `log4j.logger.skewtune.mapreduce.STJobTracker=DEBUG` to specifically generate debugging statements from STJobTracker while keeping other files at the less verbose INFO level.

### 3.2 Deployment of SkewTune Library

There is essentially no documentation for deploying SkewTune. The website contains a bare-bones quick-start guide that is not detailed. Source code is available on the public repository via the authors’ site [https://code.google.com/p/skewtune/wiki/SkewTune](https://code.google.com/p/skewtune/wiki/SkewTune). The implementation is a jar file that overwrites files in the original Hadoop 0.21.0 instance. The documentation states that one should overwrite the previous instance, but it was unclear about precisely which original Hadoop files needed to be overwritten, as the `hadoop-mapred-0.21.0-SKEWTUNE.jar` file produced by compiling code in the `skewtune` directory is not a complete instance. Initially, I placed the `hadoop-mapred-0.21.0-SKEWTUNE.jar` in the `HADOOP_COMMON` directory but ran into errors thereafter. I resolved the issue by modifying `bin/hadoop-config.sh` file and placing `hadoop-mapred-0.21.0-SKEWTUNE.jar` as the first jar to be loaded in the `HADOOP_CLASSPATH`. Since SkewTune modifies files such as `MapTask.java` and `ReduceTask.java`, originally in `org.apache.hadoop.mapred`, to permit skew detection and mitigator planning, if `hadoop-mapred-0.21.0-SKEWTUNE.jar` is not loaded before the old classes, the Hadoop configuration loads old files and SkewTune does not function properly.

Besides vagueness regarding overwrite files, there was vagueness in determining if the SkewTune framework was actually running properly. SkewTune’s default settings produce log files for the `STJobTracker` and `STTaskTracker`. The default level of logging is not verbose and does not specify whether or not tasks have been successfully repartitioned.

Since SkewTune emphasizes developer transparency, at first I assumed I would not need to worry about the SkewTune source code and partitioning details. Initially, I did not carefully go through the source code or inspect the actual repartitioning activity. I soon reneged on this.
4 Tools

4.1 Measurement Scripts

Hadoop considers slots to be cores. Measurements at the core level are difficult to obtain from the Hadoop logs. So I measured the total runtime of mapper and reducer tasks over each node in the cluster. By default, Hadoop does not come with a script that performs node-specific analyses. If the aggregate time on one node is greater than another, this implies the existence of skew; furthermore, the overall running time of the job is bounded by the maximum aggregate time on a node. In order to collect these node-level measurements, I needed information about the runtime of each task and the node where the task was running. The Hadoop Avro-JSON log file that is emitted for each job contains such information. I wrote Python scripts to parse the log file, which contains separate entries for each task’s initialization and completion events. By taking the difference between finish time and start time, as well as the task’s node name, one can calculate the aggregate time spent over each node.

Ideally, after running a job through SkewTune, the aggregate time spent on each node should be approximately the same. It is expected that SkewTune incurs some overhead as more messages are being generated than in the vanilla Hadoop instance – trackers are continuously monitoring the progress of tasks throughout the lifetime of the job and determining whether or not to repartition them. This overhead, however, should be negligible in the case of longer jobs. Ideally, SkewTune should reduce the overall runtime of a job, compared to the vanilla Hadoop implementation.

4.2 PageRank MapReduce Program

I wrote my own MapReduce programs, modifying an existing algorithm implementation from the Cloud9 Hadoop toolkit, [11], to test the functionality of SkewTune. It is worth giving a brief description of PageRank algorithm and its MapReduce implementation.

PageRank iteratively assigns a rank to each node in a directed graph $G = (V, E)$. The rank is given as a probability of being at the node; initially all the nodes have equal rank. Given a certain number of iterations the vertex’s rank is dependent on its structure. A node of higher degree will likely have a higher rank in the final output. In particular, the update rule for a rank of a given node at iteration $t$, starting at iteration 0, is given by the following update rule:

$$P(v_i, t) = \begin{cases} \frac{1}{|V|} & \text{for } t = 1 \\ \frac{1-d}{|V|} + d \sum_{v_j \in N^-(v_i)} \frac{P(v_j, t-1)}{|N^+(v_j)|} & \text{if } t > 0. \end{cases}$$

$d$ is a damping factor that represents a probability of randomly teleporting somewhere else in the graph (that is, not sending probability mass to a neighbor). Also, $N^-(v_i)$ represents the set $\{v_j | (v_j, v_i) \in E\}$ and $N^+(v_i)$ represents the set $\{v_j | (v_i, v_j) \in E\}$. We are interested in designing a MapReduce program to represent the probability at the next iteration, for $t > 0$. The expression $\frac{1-d}{|V|} + d \sum_{v_j \in N^-(v_i)} \frac{P(v_j, t-1)}{|N^+(v_j)|}$ can be made into a MapReduce program.
by having each node send a message to its neighbor. This represents the summation over $N^{-}(v_i)$. If every node $v_i$ sends the quantity $\frac{P(v_i, t-1)}{|N+(v_i)|}$ (probability at a time point divided by the number of neighbors) to its neighbor, the results can be summed together to yield the final sum.

This is precisely the algorithm given in Figure 2. The mapper accepts as its input a pair $<n, N>$, where $n$ represents a numerical identifier for a vertex and $N$ represents the structure of node number $n$, storing its adjacency list and rank. Messages are emitted to all the vertex’s outgoing neighbors in the for loop of the mapper. The reducer sums together all the masses for a given node $m$. Note that in this implementation, the adjacency list for a vertex is emitted as a message in each mapping task in line 4; the corresponding line 4 in the reducer ensures that the graph structure is maintained for successive iterations of the algorithm.

There are several steps in the Cloud9 implementation of PageRank. First, a graph must be converted into a set of adjacency list pairs, $<n, N>$, as previously discussed. For these experiments, I wrote a MapReduce program to generate synthetic graphs with obvious skew over four mapping tasks, namely graphs with $n$ vertices where the first $\frac{n}{4}$ vertices have degree zero, the second $\frac{n}{4}$ vertices have degree $\frac{n}{2}$, the third $\frac{n}{4}$ vertices have degree $\frac{n}{2}$ and the final $\frac{n}{4}$ vertices have degree $\frac{3n}{4}$. I refer to these graphs as “quarter-graphs”. This step is represented in the code as `BuildPageRankRecords`.

After the records are constructed, they are then partitioned into groups. I use a range partitioner to preserve the skew, partitioning the vertices into 4 groups. The partition with
the nodes of degree zero are easy to process, and the other partitions get progressively harder. This step is represented in the code as **PartitionGraph**.

After a graph has been converted to its appropriate record structure, we may then apply the PageRank MapReduce programs described earlier. I use only one iteration, as one iteration is enough to observe skew.

For a quarter-graph with 50,000 nodes, I obtained the following results over three trials when submitting to the vanilla Hadoop instance using a partition of size 4. The first member of the pair is the node name, and the second is the total amount of time spent for tasks on a node in milliseconds. Note that the picture of these tasks is quite similar to Figure 1, and that there are four distinctly different run-times for the four mapping tasks.

<table>
<thead>
<tr>
<th>Path</th>
<th>Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>/default-rack/slave1</td>
<td>192548</td>
</tr>
<tr>
<td>/default-rack/slave2</td>
<td>126604</td>
</tr>
<tr>
<td>/default-rack/slave3</td>
<td>12039</td>
</tr>
<tr>
<td>/default-rack/master</td>
<td>422747</td>
</tr>
<tr>
<td>/default-rack/master</td>
<td>161660</td>
</tr>
<tr>
<td>/default-rack/slave2</td>
<td>297012</td>
</tr>
<tr>
<td>/default-rack/slave3</td>
<td>220410</td>
</tr>
<tr>
<td>/default-rack/slave1</td>
<td>23702</td>
</tr>
<tr>
<td>/default-rack/master</td>
<td>300100</td>
</tr>
<tr>
<td>/default-rack/slave2</td>
<td>124300</td>
</tr>
<tr>
<td>/default-rack/slave3</td>
<td>15687</td>
</tr>
<tr>
<td>/default-rack/slave1</td>
<td>270435</td>
</tr>
</tbody>
</table>

I spent quite a bit of time getting the PageRank algorithm to compile properly; I rolled back to a 2012 version of the Cloud9 repository which had many defunct dependencies (the ant compilation tool automatically downloads program dependencies which are not already on the machine, and many of the links had deprecated). As a result, I needed to find some of the dependencies by hand; in addition, an error about missing a class in com.google.common.base.Preconditions appeared frequently. This required the addition of a -libjars option for the guava-10.0.jar file when running the Hadoop command.

## 5 Experiments

### 5.1 Line Numbers in Code

All line references, regarding code, are with respect to the Nov. 1, 2012 version of the Skew-Tune code provided by the authors on their website. It is also viewable by browser at [https://code.google.com/p/sketetune/source/browse/#git%2Fsrc%253Fstate%253Dclosed](https://code.google.com/p/sketetune/source/browse/#git%2Fsrc%253Fstate%253Dclosed).

### 5.2 PageRank Submission using the Old SkewTune API

I attempted to run one iteration of PageRank over 4 partitions on a quarter-graph of 50,000 nodes using the old API, submitting a job via the JobConf class. For simplicity of analysis,
the number of reducer tasks was set to 0; consequently only mapper tasks were considered.

5.3 PageRank Submission using the New SkewTune API

The same experiment in the previous section was run, expect on the new API, submitting a job via the Job class.

6 Results and Discussion

6.1 PageRank Submission using the Old SkewTune API

Recall that the SkewTune API requires that a job be submitted using the skewtune.mapreduce.SkewTuneJob.getInstance method. This is purportedly the only line necessary, besides import statements, to submit a job via the SkewTune framework. The documentation declares that the method can accept an object of the form JobConf or Job. The former represents the old Hadoop API, while the latter represents the new one.

Although the SkewTune documentation states that the old API is acceptable, through testing, it was discovered that the semantics of jobs specified through JobConf was not preserved. For a PageRank program from the Cloud9 toolbox, with classes conforming to the old MapReduce API, the use of a custom partitioner class (a RangePartitioner class) during a graph partitioning step was not preserved during the SkewTune job submission process. Reading through the source code, in particular line 55 of the file SkewTuneJob.java, namely this.job = Job.getInstance(cluster,conf), the author intended to convert an old Hadoop job of class JobConf into a new Hadoop job of class Job. However, the method Job.getInstance does not achieve this, as it only operates on the parent class of JobConf, Configuration, which does not convert mapping, reducing, and partitioning classes associated with its child JobConf class to the new API. Since the method is incapable of converting the appropriate mapping, reducing, and partitioning class classes into the new API, a default Hash Partitioner class is assigned to the new Job. This was also confirmed through a print statement directly after line 55 of SkewTuneJob.java. There is no obvious way to fix this issue using the current implementation; the only way to submit the old job properly using the current implementation is to rewrite the entire program in terms of the new API. But then, the job would no longer use the old API. Hence, in order to implement interoperability with the old API, the authors should have used a separate code path using operators solely from the old API, instead of attempting to convert JobConf’s into Job’s.

6.2 PageRank Submission using the New SkewTune API

Instead of trying to submit jobs using the old API, I submitted jobs using the new API. Using this option, however, new errors arose. One error arose for a reason similar to the previous section. It was a ClassNotFoundException involving a file that was already in the jar running the program, namely NonSplitableSequenceFileInputFormat.class. If the job is submitted via the normal Hadoop JobTracker, no such issue arises. So, this was a problem with the SkewTune framework. Investigating further, the error arose from a method called
SplitTask.call, line 2040 of STJobTracker.java. The method uses a JobInProgress created from the method createReactiveJob. The method createReactiveJob initializes a JobInProgress, namely line 323 in JobInProgress, which converts a job using the incorrect Job.getInstance method as from before. This method causes errors in job configuration, due to action on a superclass instead of a lower class. Consequently, the method is unable to access more specific information pertaining to the JobConf, causing the jar class to be inaccessible.

In order to resolve the previous issue, I added the jar running the program, cloud9.jar, to all the nodes and added the jar file to the HADOOP_CLASSPATH via bin/hadoop-config.sh. This allowed all the jar files to be accessible to all nodes a priori.

After the job configurations were proper, an important error arose involving incorrect time estimates for the tasks. The estimates for the task were nonsensical and causing errors in repartitioning. Namely, the time remaining for a task was estimated to be 9.22 E 15 seconds, while was the time per byte was estimated to be infinity. As the repartitioning relies on accurate job statistics, this is evidently problematic. In order to understand this issue, it was necessary to understand the execution flow of a job. I found the exact place in the code causing the issue but was unable to fix it.

7 Execution Flow

7.1 Tracing a Job from Submission

When a SkewTuneJob is created via the new API, it is submitted to both the Hadoop JobTracker and the SkewTune Job Tracker. Submitting it to the Hadoop JobTracker allows the job to proceed normally; the SkewTune Job Tracker monitors the job’s progress and determines if it should be repartitioned.

Inside the file STJobTracker.java, an object called SpeculativeScheduler contains an infinite loop which checks, for every five seconds, if there is an idle slot in the cluster. If there is, a method called speculateSlowest is called. This in turn calls a method named findSpeculativeTaskNew for each of the jobs in progress, located in JobInProgress.java. The return type for findSpeculativeTaskNew is a list of STTaskStatus’s. For mapper tasks, in order to determine what to return, a method called tasks[i].getAllTaskStatus is queried for each of the tasks in progress associated with a job in progress. getAllTaskStatus, within TaskInProgress.java, only returns a non-empty list if certain threshold conditions are reached regarding the taskStatus values associated with the tasks in progress. Namely, the task must be running for longer than a minute, and the remaining time must be greater than 60 seconds (this is where the hard-coded ω, mentioned earlier, is located). The taskStatus values are determined by the setStatus method in TaskInProgress. setStatus is called by a method named handleTaskHearbeat in JobInProgress. handleTaskHearbeat is called in the heartbeat method inside STJobTracker.java. STJobTracker only calls this method in response to heartbeats from task trackers. The task tracker emits a TaskTrackerStatus, which STJobTracker then inspect and dexamines a particular jobReport from the status, which in turn contains a taskReport. The TaskTrackerStatus is generated by passing
it to the `heartbeat` method in line 545 of `STTaskTracker.java`. The status is generated by calling the method `cloneTaskInProgressStatuses` inside `STTaskTracker.java`. `cloneTaskInProgressStatuses` does this by cloning tasks in progress from existing running jobs. A task gets added to a running job in the `init` method of `STTaskTracker`. The `STTaskTracker` implements an interface named `SkewTuneTaskUmbilicalProtocol`.

Classes that contain the `SkewTuneTaskUmbilicalProtocol` objects are located in a different part of the source tree, namely the part that contains modifications of Hadoop tasks, like `MapTask.java`, `ReduceTask.java`, and `Task.java`. The purpose of `SkewTuneTaskUmbilicalProtocol` is to update statistics for tasks in progress. Inside `MapTask.java`, there is a method called `startReporter`, which begins reporting the progress of tasks to `STTaskTracker`, namely line 389 of `MapTask.java`. The `startReporter` method is fed an object called `myProgress` which is of type `TaskProgress.MapProgress`. The file `TaskProgress.java` maintains details about the progress of a task. Through some logging statements, I discovered that there were two unreasonable initialized values, which in turn caused the nonsensical values for the remaining time and time per byte. Both remaining time and time per byte are related to `sortStartAt` and `computeStartAt`. I discovered in the logs that these values were set to 0, instead of being set to something on the order of `System.currentTimeMillis`, a large integer. There were two methods, namely `beginCompute()` and `beginSort()`, which initialize `sortStartAt` and `computeStartAt` to `System.currentTimeMillis`. I then examined which parts of the code called `beginCompute()` and `beginSort()`. These are located in methods inside `MapTask.java`, namely `runNewMapper` and `runOldMapper`. If the new API is used, the `runNewMapper` method is used, similarly for the old API. I discovered that `beginSort()` existed in both methods. On the other hand, `beginCompute()` was present in `runOldMapper` but not in `runNewMapper`. I attempted to fix `runNewMapper` by looking at code from `runOldMapper`, presumably a version from an earlier version. After the patch, the `computeStartAt` and `timeSoFar` attributes (`timeSoFar` contributes to the remaining timer) in `TaskProgress` were reasonable. But `sortStartAt` was still initialized to 0. I discovered that after line of 883 in `runNewMapper`, `beginSort()` was not being called, even though there were no blocking statements. Line 883 contains the line `mapper.run(mapperContext)`, which runs a mapper. It is likely that there is some sort of failure here; interestingly, even if the code after this line is not called, the tasks can still complete properly. Debugging this would necessitate building custom files with respect to objects of type `org.apache.hadoop.mapreduce.Mapper<INKEY, INVALUE, OUTKEY, OUTVALUE>`, as this file was not altered in the SkewTune implementation. There is not much reason this line of code should behave strangely, unless the context passed to it is unusual.

As a stopgap solution, I tried to replace calls of `runNewMapper` with `runOldMapper`, but this did not work. I got the following error message: `java.lang.ClassCastException: org.apache.hadoop.mapreduce.Mapper<INKEY, INVALUE, OUTKEY, OUTVALUE> cannot be cast to org.apache.hadoop.mapred.InputSplit`, which arises from mixing of old API’s and new API’s.

At this point in the semester, I did not have a chance to debug the issue further. Without a doubt, however, the current implementation of SkewTune given by the authors does not work properly. This is certainly a violation of the “developer transparency” design requirement for SkewTune.
It was necessary to add logging statements for each of the methods discussed in the previous paragraph, to locate the issue and to set up a pipeline of scripts to facilitate modification, compiling, and copying of modified SkewTune implementations to all nodes.

8 SkewTune Evaluation Results

Even though the estimates for the tasks are inaccurate (essentially, the current implementation assigns the remaining time for every task as infinity), SkewTune still performs some repartitioning. I examine an instance of SkewTune running on the quarter-graph PageRank test that has been the workhorse so far.

Submitting the job via SkewTune, the job spawns three additional jobs. Measurements for the jobs are as such:

```
jonathoncai@master:~/CloudGen$ python measure/dist.py
~/hadoop/logs/history/done/job_201505070426_0002_jonathoncai
(/default-rack/master, 259202)
(/default-rack/slave2, 444909)
(/default-rack/slave3, 602371)
(/default-rack/slave1, 3980)
jonathoncai@master:~/CloudGen$ python measure/dist.py
~/hadoop/logs/history/done/job_201505070426_0003_jonathoncai
(/default-rack/master, 149776)
(/default-rack/slave2, 140878)
(/default-rack/slave3, 231926)
(/default-rack/slave1, 473277)
jonathoncai@master:~/CloudGen$ python measure/dist.py
~/hadoop/logs/history/done/job_201505070426_0004_jonathoncai
(/default-rack/slave1, 114511)
(/default-rack/slave2, 80207)
(/default-rack/slave3, 80222)
(/default-rack/master, 161880)
jonathoncai@master:~/CloudGen$ python measure/dist.py
~/hadoop/logs/history/done/job_201505070426_0005_jonathoncai
(/default-rack/slave1, 114511)
(/default-rack/master, 204786)
(/default-rack/slave2, 159021)
(/default-rack/slave3, 99031)
(/default-rack/slave1, 261284)
```

The original job is job_201505070426_0002_jonathoncai. It should be noted that the skew of the job is improved; in the second job that is launched, by the time the job completes, slave1, which would normally be under-utilized under submission without SkewTune, now performs 473277 milliseconds of computation. It is difficult to interpret the precise distribution of work for this sequence of jobs, as in reality, the repartitioning is being performed over slots (cores) while I am measuring time in terms of nodes. So, one cannot just add the times over the nodes and aggregate them together for a final time distribution. I am parsing logs which do not provide slot-level detail, although in theory, it should be possible
to extract this information. Normally, as in the original SkewTune paper, what is measured is the total time it takes to run the job, not even the distribution of time over nodes, as I have tried to do here.

Over three trials, the same PageRank job runs 1.7 times slower with SkewTune than without, making SkewTune’s current implementation less than ideal. This is likely because of the inaccurate resource estimates.

9 Conclusions and Future Work

I examined the behavior of an existing implementation of SkewTune and discover, approximately, the reason for its failure to re-partition tasks properly. I have provided an explicit direction for debugging SkewTune (namely, by examining line 883 in `MapTask.java`).

In retrospect, many of the issues I dealt with could be resolved by moving to a more recent API. I was dealing with an old version of Hadoop and an older mitigator system from 2012.

Along the way, I gained valuable knowledge about Hadoop, MapReduce, PageRank, systems programming, and a greater appreciation for what it takes to build a coherent system. I would be excited about investigating resource allocation frameworks that utilize a more modern API, such as Spark. The scar tissue I have developed over the course of this project will be valuable in future investigations.

References


