Push: A Greedy Method For Efficient Task Assignment in Hadoop

Anton Petrov
Yale University
anton.petrov@yale.edu

ABSTRACT
One of the central philosophies of Apache Hadoop (the open source implementation of MapReduce [2] and GFS [5]) is to bring the computation to the input data rather than the other way around. This decreases the need to ship data over the network, reducing job response time. The current scheme of matching computational units to input data relies on a Round Robin heuristic, which has been shown to be sub-optimal. This paper presents a new method of assigning map tasks to input splits while maximizing data locality. Using a realistic simulation of a Hadoop cluster we compare the data locality achieved by our approach (which we term Push) to that of the default Round Robin scheduler and to the FlowScheduler algorithm, another proposed scheduling scheme. Finally, we evaluate the benefits of increased data locality on job response time by simulating a heterogeneous cluster with speculative execution of map tasks.

1. INTRODUCTION
Data has been termed the oil of the 21st century [7]. Data is being generated at an increasingly faster rate and companies are slowly realizing that by analyzing this data they can gain significant insights into their businesses. Retailers can find inefficiencies in their supply chain and inventory management schemes and also analyze their customers’ buying patterns to provide more personalized and tailored discount offers. Social media networks can obtain a more accurate understanding of their users’ interest and provide them with more relevant advertising. The challenge in leveraging the power of all this data is developing the tools required to store and analyze it.

Apache Hadoop is an open source implementation of the seminal MapReduce [2] and GFS [5] papers. After being evangelized by several high profile technology firms such as Yahoo!, Facebook, LinkedIn, Google and eBay, among others, Hadoop has become the de facto tool for storing and analyzing large quantities of data. Hadoop parallelizes its computation across all nodes and is designed to operate on a cluster of commodity machines where failure is assumed to be the norm. Hadoop is therefore highly tolerant of machine failures and quickly adapts when new nodes are added to the cluster or when existing ones fail.

Each file in Hadoop is broken down into chunks with adjustable size (typically 64 MB or 128 MB). Each such chunk is replicated across the cluster in order to provide a degree of fault tolerance. The default replication factor is 3, so when the primary (original) chunk is created, Hadoop is responsible for replicating that chunk to two other nodes resulting in three copies total. The replication factor provides the user with a tradeoff between fault tolerance and capacity.

Hadoop assumes a hierarchical network topology in which individual nodes are placed/grouped on racks and then groups of racks are connected together using high speed Ethernet. When a new chunk is created on a node, Hadoop replicates that chunk using a rack-aware placement policy. The first replica is placed on a random second node on the same rack as the original. The second replica is placed on a random node on a different rack than the original. This scheme decreases the inter-rack write traffic which generally speaking improves write performance. However, it does decrease read throughput because a chunk can be read from in parallel from two racks rather than three.

When a job is submitted to Hadoop, it is broken down into map and reduce tasks. Map tasks have a one to one mapping to input splits which sit logically on top of HDFS chunks [1]. Each map task processes a single input split and emits intermediate key value pairs which are then aggregated and sorted for the reduce tasks. Each reduce task is responsible for processing a particular subset of the intermediate key range. The result of the map and reduce tasks is then stored and replicated in HDFS.

Hadoop moves the computational units (map tasks) to the data chunks (input splits) by assigning map tasks to nodes which have the corresponding input split already sitting on disk. If a node gets assigned a task whose input is not local, that data is transferred from the closest node. Hence, the locality of an assignment is a measure of how many tasks are local with respect to the node to which they were assigned. High data locality is good because fewer input splits need to be transferred. Generally speaking, it is faster to transfer splits between nodes on the same rack than it is to transfer splits between nodes on different racks. Therefore, it is highly desirable for the Hadoop scheduler to seek a task assignment with the highest data locality. Fischer et. al. [3] showed that the current default Round Robin scheduler is sub-optimal in that respect. In this paper we present
Push, a greedy method that ‘pushes’ tasks to nodes using load information and we compare its data locality to that of Hadoop’s Round Robin and the FlowScheduler presented in [3].

The rest of the paper is organized as follows: Section 2 presents a detailed overview of the Round Robin scheduler along with the FlowScheduler and Push. Section 3 describes the experimental setup and its implementation. Section 4 presents a detailed overview of the Round Robin scheduler which determines which task is most appropriate and notifies the TaskTracker of the assignment. FlowScheduler and Push pre-compute an assignment prior to job execution and when a TaskTracker requests work, the task is chosen from the pre-computed list.

2. TASK ASSIGNMENT IN HADOOP

In this section we present in detail three scheduling algorithms: default Round Robin, FlowScheduler and Push. In Hadoop tasks are assigned reactively, meaning that when a TaskTracker has an empty task slot it contacts the JobTracker which determines which task is most appropriate and notifies the TaskTracker of the assignment. FlowScheduler and Push pre-compute an assignment prior to job execution and when a TaskTracker requests work, the task is chosen from the pre-computed list.

2.1 Round Robin

When the JobTracker receives a HeartBeat message (the communication protocol used to request work and report various statistics) from a TaskTracker, it checks these three conditions in order [10]:

- If there is a task whose corresponding input split is local to the requesting TaskTracker, then assign it.
- If there is a rack-local (on the same rack as the requesting node) task, then assign it.
- Else, pick the closest remote task and assign it.

Intuitively, Round Robin is sub-optimal because each node only cares about itself. A node will try to consume as many of its local tasks as possible, forcing other nodes to execute more rack-local and remote tasks. Out of the many possible assignments, a particular node will not ‘compromise’ by accepting a few rack-local and remote tasks so that other nodes may execute a higher number of local tasks. For a more formal proof showing that Round Robin deviates from the optimal assignment by a multiplicative factor please refer to Fischer et al. [3]. Despite its poor data locality, Round Robin has two advantages: it is fast and its simplicity makes it easier to implement and debug.

2.2 FlowScheduler

FlowScheduler relies on the cluster’s locality graph, which is constructed using input split placement information. The locality graph is a directed, acyclic graph whose node set includes all map tasks and all worker nodes. Every task has an outgoing edge degree equal to the replication factor, with an edge to a server if the input split corresponding to the task is local to that server.

Using this locality graph, the algorithm aims to compute a maximum flow which is then converted into an assignment. To find such a maximum flow, the graph is modified by adding two fictitious nodes, a source and a sink. The source node will have no incoming edges but will have one outgoing edge to every task in the cluster. The sink will have no outgoing edges but will have an incoming edge from each server. Let us define some notation so as to state the algorithm in a slightly more formal fashion. Let \( T \) be the set of tasks and \( S \) the set of servers where \( m = |T| \) and \( n = |S| \). Define an assignment to be a function \( A : T \rightarrow S \) assigning each task to a server. A partial assignment is an assignment where not every task is mapped to a server. Further, the input split placement is defined to be a subset of the Cartesian product of \( T \) and \( S \), namely \( \rho \subseteq T \times S \). If task \( t \in T \) is local on server \( s \in S \), then \((t, s) \in \rho\). The topology of the network is represented using a function \( F : s \rightarrow r \), mapping servers to rack IDs. Using the placement relation, \( \rho \), and the topology function, \( F \), we can construct a locality graph and deduce whether a task is local, rack-local or remote with respect to a given worker node.

The first part of FlowScheduler computes a partial assignment using the locality graph which is converted to a flow graph as outlined above. At most \( \tau \) tasks are assigned to every server, where \( \tau \) takes the values from 1 to \( m \). Every edge in the flow graph has capacity one, except for the edges from the slaves to the sink whose capacity is \( \tau \). Any tasks that are left over from the previous part are then greedily assigned to the server with lowest virtual load. The virtual load of a server, under a partial assignment \( \alpha \), is defined as the sum of the task weights for all the tasks assigned to that server. A task can have one of three weights: 1 (if it is local to the server), 2 (if it is rack-local) and 3 (if it is remote). Stated more formally, the virtual load of a node is:

\[
\sum_{t, \alpha(t) = a} w(t)
\]

where \( w(t) \) is the aforementioned weight function. As the algorithm computes a maximum flow for every value of \( \tau \), it also computes the virtual load of the assignment which is defined as the maximum virtual load of all servers in the cluster. The assignment that is output is the one whose virtual load is least. Here is the pseudocode for FlowScheduler:

```
Inputs: \( \rho \) (placement graph), \( F() \) (topology function), \( w() \) (weight function)
A, B; define as full assignments
\( \alpha; \) define as a partial assignment
A \leftarrow B
for \( \tau = 1 \rightarrow m \) do
  \alpha \leftarrow maxCover()
  A \leftarrow balAssign()
end for
if virtual load of \( A < B \) then
  B \leftarrow A
end if
return B
```

The \( maxCover() \) function computes a maximum flow on the locality graph using Ford Fulkerson; this is where the bulk of the computation occurs. Any maximum flow algorithm can be substituted for Ford Fulkerson. Rozyczki [8] has suggested using Dinic’s algorithm with link-cut trees thus reducing the runtime of \( maxCover() \) from \( O(nm^2) \) to \( O(\min(nm, \log(m))) \). As we shall see, Push provides excellent data locality and this optimization will not be needed.

The disadvantage of FlowScheduler is that it takes too long to run and is non trivial to implement and debug. On the other hand, Fischer et al. have shown that it provides...
data locality that is only an additive factor off from the optimal assignment in the worst case. The power of FlowScheduler lies in its ability to re-assign tasks based on the network topology. This is best explained using an example.

Consider Figure 1, showing a residual locality graph with a replication factor of two and with a flow of zero. The dashed edges represent augmenting paths along which flow is pushed. The capacity of the edges from the slave nodes to the sink all have capacity \( \tau \) which for this example we assume to be one. Thus the path source \( \rightarrow t1 \rightarrow s1 \rightarrow \text{sink} \) pushes one unit of flow and assigns task \( t1 \) to server \( s1 \). The other path source \( \rightarrow t2 \rightarrow s2 \rightarrow \text{sink} \) pushes another unit of flow and assigns task \( t2 \) to server \( s2 \). After updating the graph with this new flow and reconstructing the residual graph we observe that we can in fact re-assign \( t2 \) to \( s3 \) and assign \( t3 \) to \( s2 \). This way all three servers will have a virtual load of one and the maximum data locality will be achieved. Figure 2 illustrates the path which re-assigns \( t2 \) to \( t3 \) (source \( \rightarrow t3 \rightarrow s2 \rightarrow t2 \rightarrow s3 \rightarrow \text{sink} \)).

This ability to dynamically re-assign tasks in this manner makes FlowScheduler resilient to worst case adversaries. However, we believe that such graph topologies do not occur in real life clusters or if they do they are very rare. Thus FlowScheduler seems like an overkill solution. This insight was the inspiration for developing Push.

2.3 Push

Push is very similar to FlowScheduler, however once Push assigns a task to a server that task cannot be re-assigned later on. The pseudocode for Push is exactly the same as the pseudocode for FlowScheduler in the previous section, except that instead of running \( \text{maxCover()} \) we compute a partial assignment by running \( \text{push()} \). The \( \text{push()} \) function works by iterating over all tasks and for each task selecting the server with least current load on which the task is local and assigning it to that server. Any remaining tasks are assigned by running \( \text{balAssign()} \) which works similarly to the \( \text{balAssign()} \) in FlowScheduler except that our version does not allow remote assignments. The FlowScheduler version picks the machine with lowest virtual load from the entire cluster (which may result in a remote assignment) whereas the Push version of \( \text{balAssign()} \) assigns the task to the machine with lowest load on the same rack.

Since \( \text{push()} \) does not re-assign tasks, we do not need to use a flow maximizing algorithm which in turn means we no longer need to explicitly represent the locality graph. Instead we use two hashtables, one keeping track of which tasks are local on which machines and the second one keeping track of machine loads. The locality hashtable maps every task to an array of server IDs on which that task is local and the load hashtable is simply a mapping from a machine ID to that machine’s current virtual load. The algorithm computes \( m \) assignments by incrementing the value of \( \tau \) from 1 to \( m \), for each assignment computing the virtual load of the cluster and then returning the assignment with the lowest virtual load.

3. Experimental Setup & Implementation

For the benchmarking experiments, we chose to implement a simulation of a Hadoop cluster rather than incorporate our changes into the Hadoop codebase and run a full Hadoop instance. The primary reason for this choice is that a simulation allows us to have tighter control over the variables that change during experimentation. A Hadoop cluster running in a virtualized environment experiences unexpected changes in network and disk bandwidth causing large variance in performance. This makes it difficult to discern whether the gain/loss in speed/data locality is due to our approach or to unindented and random changes in the environment. This does not mean that our simulated Hadoop cluster will be entirely homogeneous, in fact our experiments include runs in a simulated heterogeneous environment, it just means that we can control the experiments with greater precision so we can better evaluate the merit of each algorithm. Another advantage of running a simulation is that we can perform experiments at a large scale to see if our proposed approach scales well in terms of runtime and memory usage. Performing these experiments at scale in the cloud would involve a heavy financial toll. In our opinion, a simulation is an excellent starting point and once the strengths and weaknesses of our approach are identified we can then incorporate it into the Hadoop codebase and run more realistic benchmarks.

We modeled our simulation as closely as possible to the current Hadoop implementation. A master thread hosts the NameNode and JobTracker objects that are responsible for replica and input tracking and job scheduling, respectively. Each worker node runs in a separate thread hosting the DataNode and TaskTracker objects. The TaskTracker of a node communicates with the JobTracker through periodic HeartBeat messages. Our entire codebase was implemented in Java and is highly object-oriented and modular so as to
easily incorporate it into the Hadoop codebase sometime in the future.

When a job is submitted through the command line, the user specifies the size of the fictitious input data set. This input file is broken down into input splits which are then distributed to worker nodes across the cluster. We created three separate heuristics to distribute the input splits, each of which is intended to model a different scenario. The first method is termed optimistic because it aims to spread out the data load across nodes as evenly as possible so that each node has pretty much the same number of input splits. The second method, random, models the current Hadoop replica placement scheme [10]. The original input split is first placed on a random node in the cluster, the first replica is placed on a random node on the same rack and the second replica is placed on a random node on a different rack. Finally, the last method, heterogeneous, aims to model a heterogeneous environment in which some nodes have a capacity limit and once that limit is filled, that node cannot accept any more input splits. This results in a heavily skewed load between nodes with some nodes having 2x or 3x more data than others.

When the input file has been split, TaskTrackers start requesting work from the JobTracker. Each TaskTracker has four task slots, meaning that it may execute up to four map tasks simultaneously. We measure job response time (how long it took for the job to execute from start to finish) as the highest number of HeartBeat messages sent from a single node to the JobTracker. This is an accurate measure because HeartBeats are sent at regular pre-defined intervals of 50ms and the job ends when the last node finishes its work (the one that sent the most HeartBeats). When assigning tasks, the JobTracker either polls a pre-computed list (this is the case with both FlowScheduler and Push) or determines the best appropriate task then and there (Round Robin).

Finally, to simulate a heterogeneous environment, when spawning the worker threads, we randomly label a fraction of them as slow. When assigning tasks we also label a predetermined fraction as slow. Rack-local tasks are by default 1.5x slower than local ones and remote tasks are 2x slower than local ones; this is done in order to simulate the latency of transferring the input splits across the network. Note that rack-local and remote tasks can actually be slower than 1.5x and 2x because those multiples reflect the transfer latency only. We wish to also model computational latency (due to bad records, or unusual input types) which is done through the random labelling of tasks, as outlined above.

### 3.1 Implementation Optimizations

Several optimizations were made to the FlowScheduler and Push algorithms in order to speed them up and reduce their main memory usage. The FlowScheduler optimizations are made with respect to the implementation of Sastry [9].

For FlowScheduler, we only keep a single representation of the locality graph. More traditional implementations of Ford Fulkerson rely on two copies of the graph: one keeping track of the flow and the other keeping track of the residual graph that is amended every time flow is pushed along a new augmenting path. Instead we keep track of the residual graph only, thus halving our memory consumption. To amend the residual graph after pushing flow along an augmenting path, we simply reverse the direction of each edge along the path, except for the edge connecting the servers to the sink (for them we lowered their capacity by one). All other edges have a capacity of one and so once flow is pushed along them, they get saturated and their direction changes in the residual graph. Initially, we represented the locality graph using an adjacency matrix. This approach is very wasteful for sparse graphs and its memory usage grows like the square of the number of nodes in the graph so it clearly will not scale well. Instead, we altered our representation by keeping track of all the outgoing edges from a given node, further reducing our memory consumption.

Instead of using Breadth First Search (BFS) to find an augmenting path in the residual graph, we used Depth First Search (DFS). This sped up the runtime because, when starting from the source, BFS first explores all neighboring task nodes (pushes all those nodes to its internal queue), then explores for each task all neighboring servers and only then reaches the sink. On the other hand, DFS picks the first neighboring task, then the first neighboring server to that task and then reaches the sink. Once we get to a task, it is highly likely that we will find an augmenting path through this task. This is because if flow was already passing through this task it would not have an edge from the source to itself – that edge would have been saturated, causing its direction to be reversed. The only time when we will not able to push flow through a task given that there exists an edge from the source to the task, is when all the servers to which the task is local have been filled to capacity. However, this starts to become a problem only towards the end of the algorithm.
at which point DFS will retreat from the task and pick the next one. Simply put BFS is not a good choice in this case because it spends a lot of time exploring nodes which will not be used in the augmenting path, whereas DFS does not.

The previous two optimizations are only relevant to the Ford Fulkerson algorithm used in \textit{maxCover}(). The following optimization is relevant to both FlowScheduler and Push. We reduced the number of iterations of the \texttt{for} loop that increments \( \tau \). The loop begins later and terminates earlier, without compromising the locality produced by the assignment. We start \( \tau \) at \( \lceil \text{numMaps}/\text{numNodes} \rceil \). This represents the ideal task distribution, where each node gets an equal number of tasks, all of which are local. A virtual load lower than this would be impossible. We also break from the \texttt{for} loop when all tasks are assigned by \textit{maxCover}() in FlowScheduler and \textit{push}() in Push. We can break from the loop then because any increase in server capacity (higher value of \( \tau \)) will not produce an assignment with a lower virtual load. In fact, since there are no unassigned tasks left over for \textit{balAssign}(), all subsequent assignments will be exactly the same. This optimization was crucial in speeding up the runtime of both algorithms.

### 4. RESULTS

This section describes the results of our experimentation. First, we compare the data locality achieved by the three algorithms, then we compare the time taken to compute the assignments and finally we present job completion times for Round Robin and Push within a heterogeneous execution environment, namely when some nodes and tasks are slowed down. The experiments were run on a 100 node cluster with the number of tasks to be scheduled ranging from 1,000 to 10,000. Each result is the average of ten runs. This is done because at each run the distribution of input splits within the cluster changes adding an element of randomness to the topology of the locality graph. The locality results represent the virtual load of the cluster, namely the node with the highest virtual load after the assignment was computed. The time taken results are all in milliseconds (ms) and represent the time it took for the algorithm to compute the assignment.

#### 4.1 Locality

Figure 3 presents the data locality achieved by each algorithm. The lower the virtual load, the higher the locality. First, notice that both Push and FlowScheduler achieve better locality than Round Robin for all three types of split distribution schemes. This was of course expected. Second, note that FlowScheduler achieves the ideal locality for \textit{random} and \textit{optimistic}. As previously mentioned, the ideal locality is when all the tasks have been evenly assigned to all the nodes and each node only has local tasks. Push lags behind this assignment by one point. This constant lag is a little peculiar and we do not have a really convincing explanation for it. Both Push and FlowScheduler were extensively tested and checked for accuracy and correctness so it is unlikely that this is a bug in the simulation. It is possible that this one point lag represents a single task re-assignment like the one described in Section 2.2. For a heterogeneous input split placement, the difference between FlowScheduler and Push is somewhat more pronounced. Nevertheless, it seems like Push provides an excellent approximation to FlowScheduler.

#### 4.2 Time Taken

For Push and FlowScheduler we simply recorded the time it took for the method that computed the assignment to execute. For Round Robin we measured the time it took to assign a task every time a TaskTracker requested one and then added up all the individual times. From the results table (Figure 4) we can see that, even with the extensive optimizations on FlowScheduler, the algorithm takes much too long to run. The disproportionately high runtime is not worth the extra locality. Further, even if we were to replace Ford Fulkerson with Dinic’s algorithm implemented using link-cut trees (also known as dynamic trees), the runtime would still be prohibitively high. Push is a little faster than Round Robin on a couple of occasions and this is due to the Round Robin implementation which frequently traverses long lists while Push makes use of a hashtable. Finally, note how there is a significant jump in runtime for all three algorithms when the input splits are distributed in a heterogeneous manner. This makes intuitive sense because given the load skew, each algorithm needs to spend more time balancing out the task assignment.

#### 4.3 Heterogeneous Execution

In the previous two sections we were not concerned with job execution time, rather we wanted to measure data locality and speed. However, the reason we are trying to

<table>
<thead>
<tr>
<th>Maps</th>
<th>Round Robin</th>
<th>Push</th>
<th>FlowScheduler</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Optimistic</td>
<td>Random</td>
<td>Heterogeneous</td>
</tr>
<tr>
<td>1000</td>
<td>21.9</td>
<td>22.1</td>
<td>31.7</td>
</tr>
<tr>
<td>2000</td>
<td>50.3</td>
<td>46.2</td>
<td>47.2</td>
</tr>
<tr>
<td>3000</td>
<td>54.4</td>
<td>54.5</td>
<td>57.3</td>
</tr>
<tr>
<td>4000</td>
<td>60.9</td>
<td>65.3</td>
<td>66.3</td>
</tr>
<tr>
<td>5000</td>
<td>65.4</td>
<td>66.8</td>
<td>78.9</td>
</tr>
<tr>
<td>6000</td>
<td>82.6</td>
<td>78.4</td>
<td>99.8</td>
</tr>
<tr>
<td>7000</td>
<td>90.4</td>
<td>93.9</td>
<td>116.8</td>
</tr>
<tr>
<td>8000</td>
<td>108.7</td>
<td>95.4</td>
<td>120.4</td>
</tr>
<tr>
<td>9000</td>
<td>116.6</td>
<td>109.6</td>
<td>160.5</td>
</tr>
<tr>
<td>10000</td>
<td>121.4</td>
<td>110.2</td>
<td>180.2</td>
</tr>
</tbody>
</table>

*Figure 4: Time Taken Results*
JobTracker the running. To determine which task is eligible for speculation, starts to speculate by re-assigning tasks that are already.

The HeartBeat (by reporting the highest number of HeartBeat messages in the cluster) in a heterogeneous cluster where speculative execution was turned on. Since Push achieves essentially the same locality as FlowScheduler we decided to only compare the performance of Round Robin to Push. Figure 5 shows the results of our experimentation.

Table: Heterogeneous Placement

<table>
<thead>
<tr>
<th>Maps</th>
<th>1000</th>
<th>2000</th>
<th>3000</th>
<th>4000</th>
<th>5000</th>
<th>6000</th>
<th>7000</th>
<th>8000</th>
<th>9000</th>
<th>10000</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Optimistic</td>
<td>Round Robin</td>
<td>Heterogeneous</td>
<td>Optimistic</td>
<td>Random</td>
<td>Heterogeneous</td>
<td>Optimistic</td>
<td>Random</td>
<td>Heterogeneous</td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td>43.8</td>
<td>39.7</td>
<td>42.9</td>
<td>43</td>
<td>43.3</td>
<td>44.9</td>
<td>76</td>
<td>76.5</td>
<td>81.1</td>
<td></td>
</tr>
<tr>
<td>2000</td>
<td>59.9</td>
<td>60.3</td>
<td>63.7</td>
<td>76</td>
<td>76.5</td>
<td>81.1</td>
<td>108.3</td>
<td>108.7</td>
<td>113.8</td>
<td></td>
</tr>
<tr>
<td>3000</td>
<td>77.3</td>
<td>76.5</td>
<td>82.8</td>
<td>140.7</td>
<td>143.3</td>
<td>154.8</td>
<td>172</td>
<td>176</td>
<td>187.7</td>
<td></td>
</tr>
<tr>
<td>4000</td>
<td>95.8</td>
<td>95.1</td>
<td>103.9</td>
<td>207.6</td>
<td>207.2</td>
<td>228.9</td>
<td>245.1</td>
<td>238.4</td>
<td>264.2</td>
<td></td>
</tr>
<tr>
<td>5000</td>
<td>115.5</td>
<td>114.3</td>
<td>124.7</td>
<td>273.7</td>
<td>277.6</td>
<td>302</td>
<td>308.4</td>
<td>304.1</td>
<td>335.4</td>
<td></td>
</tr>
<tr>
<td>6000</td>
<td>133.4</td>
<td>134.5</td>
<td>143.3</td>
<td>273.7</td>
<td>277.6</td>
<td>302</td>
<td>308.4</td>
<td>304.1</td>
<td>335.4</td>
<td></td>
</tr>
<tr>
<td>7000</td>
<td>151.7</td>
<td>148.7</td>
<td>165.9</td>
<td>273.7</td>
<td>277.6</td>
<td>302</td>
<td>308.4</td>
<td>304.1</td>
<td>335.4</td>
<td></td>
</tr>
<tr>
<td>8000</td>
<td>175.8</td>
<td>168.9</td>
<td>185.5</td>
<td>273.7</td>
<td>277.6</td>
<td>302</td>
<td>308.4</td>
<td>304.1</td>
<td>335.4</td>
<td></td>
</tr>
<tr>
<td>9000</td>
<td>191.2</td>
<td>189</td>
<td>203.2</td>
<td>273.7</td>
<td>277.6</td>
<td>302</td>
<td>308.4</td>
<td>304.1</td>
<td>335.4</td>
<td></td>
</tr>
<tr>
<td>10000</td>
<td>210.5</td>
<td>203.7</td>
<td>226.2</td>
<td>273.7</td>
<td>277.6</td>
<td>302</td>
<td>308.4</td>
<td>304.1</td>
<td>335.4</td>
<td></td>
</tr>
</tbody>
</table>

Figure 5: Heterogeneous Execution Results

achieve high data locality is because we want to reduce the frequency of replica transfers over the network and thus implicitly reduce job response time. We measured job response times for a perfectly homogeneous cluster, where all tasks take the same amount of time (in terms of computation, rack-local and remote tasks inherently take longer to process because they need to be transferred) and where each node runs equally fast. As you might expect, the higher locality immediately translated into a faster job response time. However, the assumption of homogeneity is unrealistic because Hadoop is meant to run in an environment where failure is the norm. As such, we ran experiments where 25 percent of the cluster nodes where slowed down and where 15 percent of the tasks took longer than average.

We attempt to combat this heterogeneity using speculative task execution. Our belief is that once the fast nodes have processed all the tasks they were pre-assigned, they can then 'steal' tasks from the slow nodes. Note that this type of reasoning would be incorrect in the context of Round Robin because there tasks are not pre-assigned and a node would 'free up' only at the last of wave of map tasks. In order to test our hypothesis we measured job execution time (by reporting the highest number of HeartBeat messages in the cluster) in a heterogeneous cluster where speculative execution was turned on. Since Push achieves essentially the same locality as FlowScheduler we decided to only compare the performance of Round Robin to Push. Figure 5 shows the results of our experimentation.

Round Robin uses the default Hadoop speculation procedure whereas Push implemnts the LATE (Longest Approximate Time to End) speculative scheduler [11]. The default Hadoop speculative algorithm works as follows. Each map task is given a progress score ranging from 0 to 1; the score is equal to the proportion of the input split that the map has processed up to date. When a node requests an additional task and there are none left, then the scheduler starts to speculate by re-assigning tasks that are already running. To determine which task is eligible for speculation, the JobTracker computes the average score of all running tasks and any task whose score is less than 0.2 of the average is eligible to be speculated. Further, a task can only be speculated unpon once. All tasks whose scores are less than 0.2 of the average are treated with the same priority and any ties are broken by locality. As outline by Zaharia et. al. [11], this scheme is sub-optimal in a heterogeneous environment. Please refer to that paper for a detailed discussion as to why this is the case.

The LATE scheduler revolves around the simple heuristic that at any given point we would like to speculate on the task that is going to finish last because this gives the speculative copy the best chance of overtaking the original and thus fulfilling its purpose. To implement this, the LATE scheduler also keeps track of a task’s progress score. It also computes the task’s progress rate = progress score/T, where T is the time for which the task has been running. Given the progress rate and current progress score, the expected end time of the task is equal to (1 - progress score)/progress rate. Thus, in Push, when a TaskTracker is done with the work that was assigned to it, the JobTracker builds a priority queue of all running tasks ordered by finish time and assigns a speculative copy of the task that will finish last.

Sadly, this scheme did not prove sufficient to lower the job execution time below the Round Robin execution time. From Figure 5, we can see that Push with LATE performed worst than Round Robin on all accounts. After some thought, we believe we have come up with a plausible explanation for this. During Round Robin scheduling there is large variance in node virtual load (i.e. the number of tasks each node ends up executing). This phenomenon is due to the scheduling algorithm and has nothing to do with the heterogeneous environment simulation (however, virtual load and replica placement are somewhat replated; optimistic and random providing less variance in virtual load than heterogeneous). Given this variance in load with Round Robin, during our experiments it was more often the case that the nodes with the lower virtual load were labelled as slow. This made slowed them down but not enough for them to become the limiting factor in the job execution time. The introduction of slow nodes essentially leveled the execution time so that most nodes finished around the same time. Of course, there were occasions when nodes with a high load were labelled as slow but those were rare, given the random nature of the experiment. On the other hand, with Push (and FlowScheduler for that matter) the distribution of virtual loads amongst nodes is very even. Again, this is a property of the scheduler. Thus, whenever any set of nodes is labelled as slow, those nodes instantly become the limiting factor and increase the job execution time.

The way to fix this would be to give the JobTracker information about which nodes in the cluster are going to be slow
prior to computing an assignment. Currently the assignment is computed with the assumption that all nodes will run at the same speed. This is the area that needs most work. We expected that the speculative execution would compensate appropriately but that unfortunately proved insufficient. Finally, it is also a perfectly valid assumption for the JobTracker to know which nodes are slow and which aren’t (it probably keeps statistics regarding execution speeds from previous jobs). We may be able to use a balancing heuristic similar to the one suggested in [6].

5. CONCLUSION

Data locality is not the end all be all. A high locality is not in itself sufficient in reducing job execution time. Our proposed method approximates FlowScheduler very well and despite the numerous optimizations incorporated in FlowScheduler, Push runs in a fraction of time making it a worthwhile alternative. The LATE speculative scheduling system has been shown to outperform the default Hadoop speculator. When combined with the suggested improvement in Push, we are confident that the resulting algorithm will prove very valuable.

6. FUTURE WORK

There are a number of possible avenues for future work beyond the aforementioned optimization.

- Incorporating Push into the Hadoop codebase and running the same experiments on a live cluster.
- Figuring out how task scheduling fits into MapReduce version 2, or YARN [4]. This version of MapReduce will soon be merged into the Hadoop codebase and will become standard. If we want our work to stay relevant we must figure out a way to incorporate it into YARN.
- Including the Push mechanism into the existing pluggable schedulers: Capacity Scheduler, Fair Scheduler and Hod Scheduler.

7. ACKNOWLEDGMENTS

I would like to thank Girish Sastry for introducing me to Hadoop and sparking my interest in large scale data management systems. Further, I would like to thank Professor Daniel Abadi for his kind understanding and guidance throughout this project. Finally, special thanks to Daniel Rozycki and Xueyuan Su for their insightful suggestions and discussions.

8. REFERENCES