Project 2 Report

Team Members
Aaron Knodell - 420007922

Time Spent
About 10 hours of research and 40 hours of development.

Link To Repository
https://github.tamu.edu/aknodell/689-18-a/tree/master/P2

Final Version of Code For Grading
https://github.tamu.edu/aknodell/689-18-a/tree/a720c285ce016901dc90f0e82b5012f373c199d3/P2

Project Specification
Design and implement a simplified version of Hadoop's default scheduler and FairScheduler for distributing execution of MapReduce jobs. Tasks included designing a virtual representation of a cluster, designing a process for executing a task on the cluster, implementing the default scheduler, and implementing a simplified version of the FairScheduler.

Background Research
Most of the research involved learning about Hadoop's architecture and how the different scheduling algorithms operated.

Hadoop's architecture essentially consists of a ResourceManager that manages the hardware and NodeManagers on each node of the cluster. The ResourceManager contains a Scheduler and an ApplicationsManager. The ApplicationsManager is responsible for accepting incoming jobs, while the Scheduler is responsible for scheduling existing jobs. The NodeManager is responsible for managing the hardware and communicating to the ResourceManager when a task is completed.

The default scheduler was surprisingly difficult to find information on. Most sources just referred to it as a simple FIFO queue, however one source implied that jobs were sorted by priority first, then by submission time. This seemed more likely to me, at least for more recent versions of Hadoop since priority was included in the JobConf class documentation.

The FairScheduler algorithm was unsurprisingly much more complex. Essentially, all jobs are assigned to separate pools, usually by application. Each pool is assigned a fair share of resources that it has access to. This is based on specified minimum resources for each pool, as well as a relative weight given to each pool. The pools have queues for the jobs that they contain with different ordering protocols: the default FIFO, the fair algorithm or DRF

(DominantResourceFairnessPolicy). When a resource becomes available, the algorithm selects the next task to execute based on which pool is the most underserved according to it's determined fair share of the resources and then according to the internal scheduling protocol of the pool's queue. Configurations for each pool and queue are read from an xml file.

The default FIFO queue works the same as the default Hadoop scheduler: jobs are sorted by priority and then by time. The FairScheduler is designed to take the priority of the job into account, but to allow lower priority jobs a chance to start running earlier. Essentially, each priority level doubles the share of resources allocated to the queue compared to the level below it. That is, a HIGH priority job get twice as many resources as a NORMAL level job.

Design

I decided to start by designing a representation of the hardware. Machines are represented by a Cluster class that holds a list of NodeManagers. Each machine has a specified amount of memory, and a number of slots for map tasks and for reduce tasks, which correspond to cores.

The ResourceManager is also represented by a class and contains classes for the Scheduler and ApplicationsManager.

The configurations are loaded by a Backend file that performs validation for the filenames and for each configuration contained in the file and automatically loads them. The program runs the entire scheduling sequence automatically, with the user only specifying which scheduling protocol they want to use for the run.

Machines are loaded into a the Cluster object, which is also contained in the ResourceManager so that it has direct access to it. Pools are loaded into the ApplicationsManager and Jobs are loaded into the pools. Pools are accessible by the outside ResourceManager. The Scheduler does not own any objects, but it is passed the pools and the cluster.

NodeManagers exist just to run tasks and communicate back to the ResourceManager when they complete.

Implementation

Config Files

The configuration for the system is loaded from three files. The cluster is configured by the file "clusterConfig.txt", with each line being the configuration for a machine in the form:

```
<memory> <numberOfMapSlots> <numberOfReduceSlots>
```

Memory is measured in gigabytes. All values must be integers.

The pools are loaded from “poolConfig.txt”, with each line containing the configuration for a pool in the form:

```
<poolName> <queueType> <weight> <minMapSlots> <minReduceSlots>
```

PoolName and queueType are strings, weight, minMapSlots, and minReduceSlots are integers. Additionally, queueType must be either “fifo” or “fair”.

Finally, jobs configurations are loaded from the file “jobConfig.txt”. Each line is a configuration of the format:

```
<poolName> <numMapTasks> <numReduceTasks> <memPerMapTask> <memPerReduceTask> <priority(optional)>
```

All the values are integers except for poolName, which is a string. If the poolName does not exist in the system, the job will be assigned to the default pool, as long as all the other values are valid. Priority defaults to 1 if it is not defined.

Hardware

The biggest challenges by far were coordinating knowledge of the system between the various parts. This was mainly accomplished by storing all the running classes within the ResourceManager class. This isn’t necessarily an accurate representation of an actual system, but it was necessary for sharing vital information in the most efficient way possible, especially while maintaining simple execution of the program.

Each NodeManager had a number of defined spaces for map tasks and for reduce tasks. This was based on the documentation for Hadoop.

General Scheduling Algorithm

Both scheduling schemes used a similar base algorithm, with differences being in the details of the implementations. The general algorithm is:

```
while  remaining_tasks_in_active_jobs > 0:
       get_next_task_from_active_jobs
       assign_task_to_available_slot
       if remaining_tasks_in_active_jobs == 0:
              start_next_job_in_active_pools
```

In practice, since the map and reduce tasks were separated, I had the scheduler run two threads, one for map tasks, and one for reduce tasks. The reduce tasks thread had an additional check to make sure that all the maps for a job had been completed before starting the reduce tasks.

Fair Scheduler

The fair scheduler begins by calculating the fair shares for each pool and paring down the list of pools to just those that are active (have jobs assigned to them). It then determines which pool is the most underused, which is simply pool with the fewest running tasks in proportion to its fair share. It starts a task from that pool, and continues the process until all jobs are completed.

Calculating Fair Shares

Fair shares were assigned to pools according to a three possibilities. Before assigning pools, the "total fair shares" were calculated by multiplying each pool by its minimum slots and summing the results. If that was less than the available slots in the cluster, the shares were scaled up.

If it was more, the minimum shares were summed. If that was less than the total, then the extra slots were allocated according to the "total fair shares" proportions. If the minimum shares were greater than the available slots, then the minimums were scaled down. Float values were used to prevent any pool from being assigned zero as a "fair share".

FIFOQueue

The FIFOQueue is used for pools that have the "fifo" type. It's implemented as a basic python list. To add a new job, the queue traverses its elements from the beginning until it reaches a job that has a lower priority or an equal priority and a more recent timestamp and inserts the new element directly in front of it.

FairQueue

The FairQueue is used for pools that have the "fair" type. Implementing the "fairness" of the queue was challenging. What I eventually came up with was a 31 step cycle where 16 spots meant a task from a VERY_HIGH priority job was executed, 8 spots mean a task from a HIGH priority job was executed, and so on down to VERY_LOW.

FIFO Scheduler

The FIFO scheduler is actually implemented the same way as the Fair Scheduler. The only difference is that instead of assigning jobs to different pools, all the jobs are assigned to the default pool which uses a FIFOQueue. Then the scheduler runs normally, with the selected pool always being the default pool.

Simulating Running Tasks

Runnings tasks were simulated within the NodeManagers. Each NodeManager was its own thread that ran in parallel. For each task in the NodeManager, a random number would be generating and the task would have a 20% chance of completing. Then the thread would sleep of 0.25 seconds. This was to simulate the variability of runtimes so the results wouldn't be the same for every run, and so that the Scheduler would have time to schedule new tasks when old ones were completed.

Instructions for Running Code

The code is written in python 2.7 and requires the following files to be in the same directory in order to work correctly:

- schedulerCli.py
- schedulerBackend.py
- schedulerConstants.py
- resourceManager.py
- nodeManager.py
- queues.py
- clusterConfig.txt
- poolConfig.txt
- jobConfig.txt

To run the code, enter the command "python schedulerCli.py". The user will then be prompted to choose either the FIFO scheduler or the FairScheduler. Once the scheduler is selected, the code will load the configurations from the config files, and execute the jobs automatically. Once the code is finished, a goodbye message is printed and the program ends. A log of when each job was started and completed in written to schedulerLog.txt. Changes to the cluster, pools, or jobs can be made by editing the config files directly.

Conclusions

When the FairScheduler was compared to the default FIFO scheduler over several runs, there wasn't a significant difference in overall time to execute the tasks. This makes sense, since the limit on overall performance is mainly the size of the cluster, not the scheduling mechanism. However, the was an obvious difference in when jobs were started. Even when the pools were very unevenly weighted (12 to 1), jobs from the lowly weight pool were still started early in the process.

A stretch goal for this project was to implement the Capacity scheduler designed by Yahoo, in addition to the the Fair scheduler. However, there was not time to begin any serious implementation. The Fair scheduler was much more complex that anticipated, and it took a significant amount of time to design an implementation of the infrastructure that would work for multiple scheduling schemes. Even then, there were two major re-factors to the default scheduler code involving how entities in the system were represented that occured after a working solution had been found.

This also resulted in several more advanced features of the system architecture and the Fair scheduler being left unimplemented. Most of these, like task failure, would have been interesting both for the challenge and in the interest of creating a more realistic representation of the system, but were given less of a priority in the interest of getting a basic working system in place before automating errors within that system.

Originally an interactive system was envisioned. However, implementing the system to providing feedback while accepting user input would have been a substantial task and seemed like it was outside the scope of the original idea submission.

Ultimately, this project was a reminder that even a fairly simple algorithm can become quite complex when it comes to the actual implementation.

References

J. V. Gautam, H. B. Prajapati, V. K. Dabhi and S. Chaudhary, "A survey on job scheduling algorithms in Big data processing," *2015 IEEE International Conference on Electrical, Computer and Communication Technologies (ICECCT)*, Coimbatore, 2015, pp. 1-11.

Hadoop Fair Scheduler Design Documents:
https://svn.apache.org/repos/asf/hadoop/common/branches/MAPREDUCE-233/src/contrib/fairscheduler/designdoc/fair_scheduler_design_doc.pdf
https://hadoop.apache.org/docs/r1.2.1/fair_scheduler.pdf

Hadoop Online Documentation:
http://hadoop.apache.org/docs/r3.1.0/hadoop-yarn/hadoop-yarn-site/YARN.html
http://hadoop.apache.org/docs/r3.1.0/hadoop-yarn/hadoop-yarn-site/NodeLabel.html
https://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/FairScheduler.html
https://hadoop.apache.org/docs/r2.6.2/api/org/apache/hadoop/mapred/JobConf.html
https://hadoop.apache.org/docs/r2.6.2/api/org/apache/hadoop/mapred/JobPriority.html