MP4 Report

Aarya Bhatia aaryab2@illinois.edu, Ricky Hsu rickyhh2@illinois.edu

Architecture

The MapReduce framework consists of nodes running an SDFS and MapleJuice RPC server, and a Failure Detector (FD). The FD notifies servers of node changes using callback functions.

MP3: Added support for file writes in Truncate and Append mode. Also, allow the client to specify an offset and range for file reads.

MP4

Leader: The leader manages job queues, where each job is run sequentially and generates several tasks. Each task is assigned to a worker through a partition function (hash, range).

Jobs: A MapJob divides a file into batches of at most 100 lines called MapTasks. A ReduceJob groups files by key (as there may be multiple intermediate files per key) and associates each key with a ReduceTask. Jobs undergo Start, Task, and Finish phases, managing worker allocation, task scheduling, output aggregation and worker deallocation.

Workers: Worker nodes run multiple executors as goroutines, communicating through channels, and executing tasks with inputs/outputs to/from SDFS.

A worker downloads the executable and spawns a pool of executor threads of a given size. After each task is received and executed, the output from the executable is parsed as key-value pairs and aggregated with the existing tuples. An ack is sent to the leader for each completed task. Each worker's output file is different so that they can run in parallel.

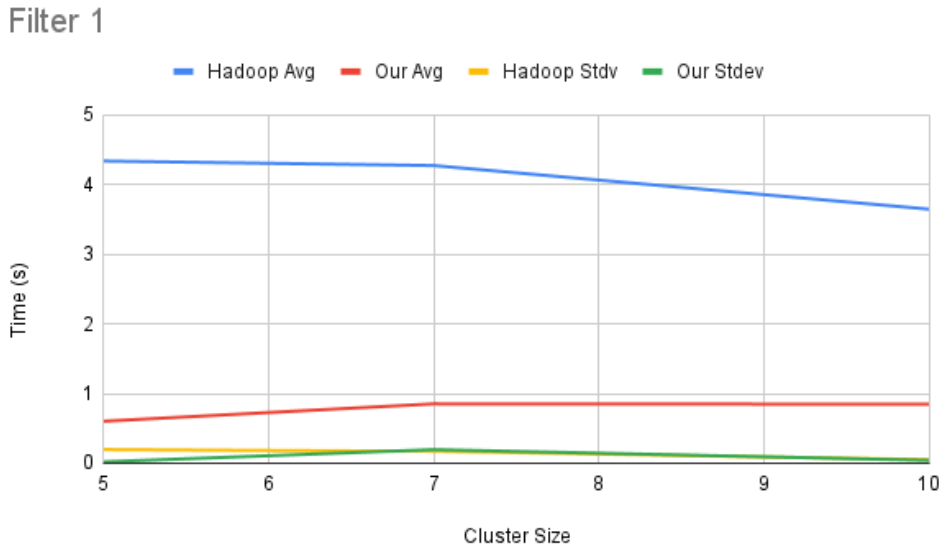
Failure Handling: Failure handling involves task rescheduling, job retries, and removal of output files from failed workers.

MapReduce: Map and reduce executables follow specific stdin/stdout formats, and receive user args and system args like filename and key. The input to a map program is lines of text and for a reduce program it is lines containing values for a given key linewise. The output from both programs should contain lines with "key: value", multiple keys are ok.

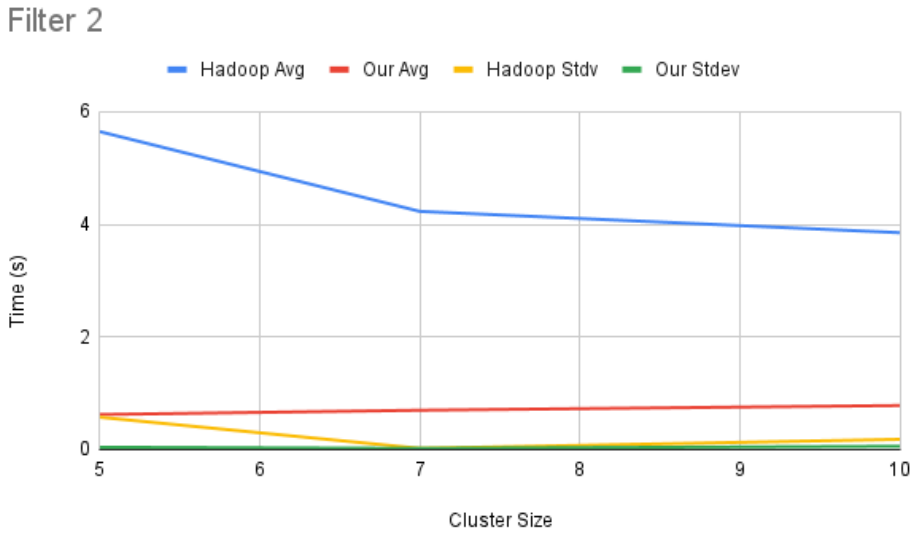
Experiments

- We write a map and reduce executable file (language agnostic) that is uploaded to SDFS before the job.
- For filter queries, the map emits tuples such as (key=filename, value=line of text).
- For join queries, the map emits tuples such as (file1.column1, file1 fields…) or (file2.column2, file2 fields…). The reducer combines the fields of both files and emits (key, file1 fields…, file2 fields…) where each file has csv data.

For simple filter, we used the TrafficSignalIntersection dataset with the regex "Video,Radio". MapleJuice performed significantly better than Hadoop.



For complex filter, we ran a filter query on the regex "[^,]+,{3,}[^,]+" and had a similar outcome.



For the joins we generated custom data with "customers" and "orders".
For the simple case, we joined on customer.ID = orders.customerID where each customer had multiple orders (one-to-many). Hadoop to perform significantly better than MapleJuice. Most of the bottleneck came from the reduce phase, although our map

phase performed better than Hadoop for both joins. For simple joins, the final number of output lines was equal to the number of orders.

For the complex case, we modified our dataset so that each customer had an ip_address picked randomly from a fixed pool. We performed an "inner join" on customer.ip_address = orders.ip_address (many-to-many). MapleJuice blew up on large datasets. The number of output lines increased quadratically with input size.

Conclusion: We gained some performance in the map phase by processing lines in chunks unlike hadoop. However, we could optimize more in the I/O for the reduce phase. The main bottleneck for map phase was having too many keys - more file uploads to SDFS. The main bottleneck for reduce phase was having too few keys - large load on fewer executors and more idle executors. Specifically to join, we can have a separate parallelized line processing segment in the reduce phase. This will help parallelize the reduction portion as well.