MP4 Report - CS425

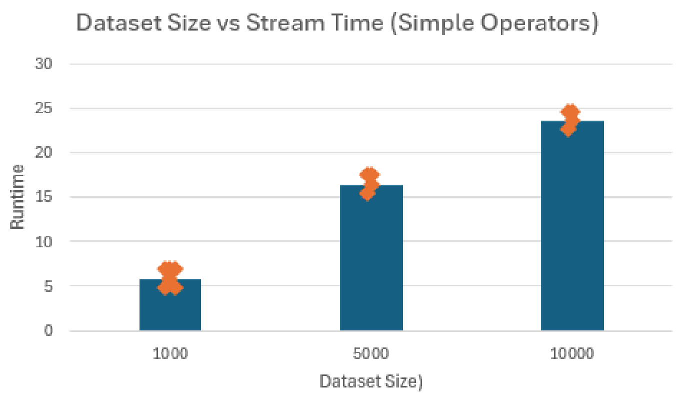
Carson Sprague (cs104) & Rohan Kumar (rohandk2)

Overall Design:

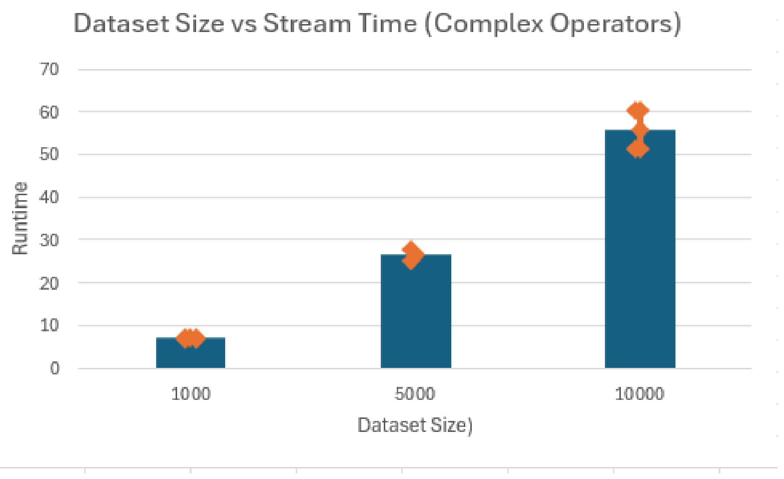
For this MP, we wanted to leverage our code from the previous MPs to facilitate the development of RainStorm, our distributed stream processing system. We had to be extremely careful about the design choices of our system, as there were several moving parts which all touch each other in some way. We made use of HTTP, TCP and UDP protocols across the entire system, each for different tasks. We used HTTP to send requests to and from our HyDFS from MP3, UDP for the Ping/Ack failure detection from MP2, and TCP to send tuple data between nodes in our stream processing system.

Our RainStorm implementation is as follows. We have a central leader which initializes the roles for each node, dividing the nine remaining worker nodes into their respective stages (Source, OP1, and OP2). We then retrieve our input data file from HyDFS and use a hash partitioning function to evenly distribute the data amongst the three source nodes. We use TCP to open up listeners on each node to listen for incoming data, and begin sending tuples from our source nodes to our executable nodes. At each executable node, we first check whether the tuple received was a duplicate and skip it if necessary. Once a tuple is sent to our transformation nodes, we use the exec.Command function to run our executable, passing in the tuple as special input. By using the exec.Command function we are able to pass in different executables as parameters to make our system agnostic. After a tuple is processed, it is sent to a log file in HyDFS, which contains the timestamps of when the job was started, when the job was finished, and the tuple's unique ID to avoid duplicate tuple processing. Once the tuple is logged and processed, we use TCP to send our tuple as a string to the next node. At the OP2 node, the tuple is sent back to the leader, where it is logged to a final output file in HyDFS and printed to the leader's screen. To handle duplicates, we associate each line with a unique identifier in the form of <filename:linenumber>,line. Each node has a map of those UIDs with the associated line, and whenever a duplicate appears it is discarded.

Performance:



For our testing we used the City of Champaign Traffic Sign dataset (https://gis-cityofchampaign.opendata.arcgis.com/datasets/f5aaae43ed6642cc944117f4f4221adb_37/explore). For our simple operator test, we used a sequence of filtering based on a parameter and then a split to select useful fields to us. A real world usage of this being 'I need to find all of the parking signs, but only need the ID's of those signs and none of the extra columns.' We found here that our stream framework was pretty quick, with what looked like a logarithmic trendline as the dataset grew. For a sample of 1000 signs, we got a time of about 6 seconds, compared to a dataset of 10,000 signs we had a time of about 25 seconds. This was also a stateless operation, which is likely also why it was fairly quick (much less processing behind the scenes).



In our complex operator test, we used a stateful sequence of filtering based on one category/column, and then keeping a count of the items in another category/column out of that filtered set. This would be equivalent to doing some complex database work to find the count of types of signs that have a specific type of pole (punched telespar!). We found that our system really chugged with this. Compared to the somewhat-logarithmic curve that we had in the stateless operator example, we had a linear curve here (6 seconds for 1000 signs, 60 seconds for 10,000 signs). We attribute this to the increased complexity of keeping track of the count in persistent storage, which incurs quite a bit of overhead with our HyDFS.

Spark Streaming Comparison

Our system, while functional, was really not up to spec of Spark's implementation. There's a couple reasons for this, one of which being a student project, and the others being due to implementation. Spark is known for using batch processing, while we processed all of our records individually, meaning every time a line of data was processed we would send it out immediately, compared to how Spark would accumulate a couple entries and then send it. This reduces the load on the network by a factor of N, N being the amount of entries it collects before sending it out. Also, we used our persistent storage system (HyDFS) to handle failures, and again our HyDFS system is written by students and is not the most efficient way of handling things. To improve our system in the future, we would implement a sort of batch-streaming feature to reduce the load on our network. This would again reduce messages by a factor of N. Also, we would cache some of our data so that reading/writing to entries would be much faster than writing to the HyDFS.