Anant Goyal, anantg2@illinois.edu
Abuzar Hussain Mohammad, ahm7@illinois.edu

Design: For our Rainstorm design we went with the approach of having one Rainstorm leader process (always running on VM1) along with its associated leader server. On the remaining VM's we start by having Worker Servers always listening to requests from the leader process. As the leader process requests worker tasks to be started and distributes that request to the respective worker servers, the servers then spawn a worker task process and based on the Rainstorm stage they were assigned by the leader, the worker task subsequently spawns a subprocess which applies the operator (identity, transform, filter etc.). The leader (now aware of all tasks on all VM's) shares a "routing path" with each task to inform the task of its downstream task. The leader process now acts as a "stage 0 task" to begin sending tuples (based on the hash of the tuple mod # of tasks in stage 1) evenly to all tasks in stage 1. Stdin of the worker process passes the tuple to the operator process which outputs it via Stdout where the worker process can forward it to its downstream task (again based on the hash mod # of downstream tasks) and the cycle continues. Tasks in the final stage write to an output file on HyDFS instead of forwarding the tuple. A few important notes: Firstly, upon receiving a tuple each task sends an 'ACK' back to its upstream task. Secondly after processing a tuple each task adds the unique tuple Id to a hashset along with writing it to a task specific append only log. In case of task failures the task forwarding the tuple will detect that its downstream task has failed (since it stops receiving ACK's) and notifies the leader process of this failure. The leader process starts the journey of restarting the failed task on a new VM by first sending it a "routing path" and secondly sending it the failed tasks append only log. This allows the newly started task to replay the failed tasks tuples preventing it from processing any duplicated tuples. Lastly the leader also informs the tasks in the upstream stage of the new worker task so that they can route their tuples correctly. To handle autoscaling we have each worker task reporting their current "load" back to the leader which determines whether tasks need to be added or removed in that stage based on the comparison of the task load to the average LW and HW. Scaling up initiates the same process as starting a new task, which scaling down initiates the same process as kill (minus the restarting of the scaled down process). When forwarding tuples from one process to another we also encode additional information such as the stage id of the sending task, the unique tuple Id and a key which consists of the filename:line number that can be hashed. To account for some dropped tuples we also have each worker task running a "retry tuples" thread which checks which tuples it is waiting to hear an ACK for, and retries sending those periodically, up to a maximum of 3 times. If the worker task receives a tuple that was processed before, it still sends an ACK to the upstream target. Our operators were written in Python for quick tuple processing while the rest of our program is written in Java. The biggest challenge we faced was making our log and output file appends faster as those bottlenecked the tuples being sent. As a result we had to modify our MP3 append behaviour to just write out to a single file rather than wait for replicas to be created, append to and ACK'd.

Anant Goyal, anantg2@illinois.edu
Abuzar Hussain Mohammad, ahm7@illinois.edu

Past MP Use: While we did not disable our MP2 failure detector it was not useful for MP4 since we were not handling VM fails (instead focusing on process failures). Our MP3 was also only somewhat useful as we had to modify our MP3 append logic to handle much higher volumes of appends and throughput. MP1 was very useful to debug our local task logs.

LLM Use: We primarily used ChatGPT as our LLM of choice. After initial design discussions we used the LLM mainly to refine certain decisions such as how to handle task restarts and forwarding of tuples. We then used it to generate helper functions such as our Main commands like list_tasks and kill_task. Overall, our experience using an LLM for MP4 was very good and we found it sped up development and debugging significantly.
LLM Session File: /misc/Stream processing architecture.webarchive

Measurements: The datasets we used to compare Rainstorm vs. Spark was from the Wikipedia Dataset, specifically the English portion of the Wikicorpus. Combining a few of the raw text files, we generated two ~100MB and ~150MB text files containing english words which we used as our datasets. Wikipedia Dataset: http://www.cs.upc.edu/~nlp/wikicorpus/

- For Application 1 we used a 2 stage grep + replace where the first stage filtered lines on a word and the second stage replaced occurrences of one word with another.
- For Application 2 we used a 2 stage grep + AggregateByKey where the first stage filtered lines on a word and the second stage gave each word and # occurrences.
- To keep both Rainstorm and Spark streaming consistent we used N_tasks_per_stage = 3 and ran both rainstorm and spark on all 10 VM's

For both datasets we noticed the tuples/sec throughput did not vary much. This is likely since both datasets were Wikipedia corpus containing english text with the only difference being the size of the files. Varying the type of input files may cause the throughput rates to fluctuate more. Most glaringly, however, we notice our rainstorm performing significantly worse when compared to Spark. On average our rainstorm (for both application 1 and 2 on both datasets) was producing output tuples at a rate of approximately 290 tuples/second. While this was okay for smaller file sizes up to a few MB's, as the file size reached close to 100MB our rainstorm was taking upwards of many minutes whereas we noticed Spark would finish processing in ~20 seconds. As such, Spark output tuples per time processing was significantly faster and in the range of 10's of thousands per second. We also noticed the standard deviation for both rainstorm and spark was not much between trials and stayed very consistent.

Anant Goyal, anantg2@illinois.edu
Abuzar Hussain Mohammad, ahm7@illinois.edu

Analysis: There are many reasons for why Spark significantly outperformed Rainstorm some of which being optimizations Spark makes while other being design flaws in Rainstorm itself. Spark's core operators run in Scala with optimized memory management and a native dataframe engine. Rainstorm on the other hand launches separate Java processes which in turn launches separate python scripts that process line-by-line I/O. Rainstorm also serializes each input line as a json object then has to deserialize that object. Rainstorm emphasizes simplicity and correctness while spark optimizes for performance and is generally a production quality stream processor. Our comparison results align with these expectations.