# Distributed Systems (CS 425) - Fall 2024 - MP 4 Report

Akhil Sundaram(akhils7), Anurag Choudhary (anuragc3)

**Design Overview:** Rainstorm is a stream-processing framework built in golang using grpcs as a communication mechanism between the scheduler and workers. The first node in the cluster acts as the scheduler and the rest are workers (streaming tasks only allocated on the workers). **[Scheduler]** The scheduler/leader is responsible for allocating tasks to worker nodes, managing task IO streams, failure detection and recovery. It uses in memory data representations to keep track of : (1). Different tasks running on a specific node, (2). the various attributes related to a task and values needed for it to function, and (3). the progress for every task so far. (3) is done by maintaining checkpoints for each task that is regularly sent from workers to the leader. These representations help in providing an overview of the state of the system, and also help in reassigning tasks if any node fails during execution. **[Worker]** The worker node, upon receiving a request from the leader, spawns a Task goroutine that executes based on the provided stage, taskID, operator, custom parameters and input file. During execution, the worker periodically sends checkpoints and status updates back to the leader. After processing every N items (currently 10), the worker outputs the results to an intermediate file in Hydfs. It will also send a checkpoint to the server. Tasks in the subsequent stage read from these Hydfs files, typically based on their stageId+taskIDs. For aggregate stages, the previous stage's task determines the appropriate Hydfs file by hashing the keys of the key-value output to ensure correct data placement. **[Failures]** The worker is transparent about failures and starts processing data within a file range sent by the leader. It also maintains a LinesProcessed data store to detect failures from previous stages and discard duplicate entries from a previous stage. For the scheduler, in the case of failures it will bring up the failed task on another worker node with the same state as the previous failed node. (1) helps in deciding which node to assign new tasks/restart failed tasks on. It finds the node which has the least number of tasks running on it, and chooses that. Once a new node is chosen, we use the details from (2) and (3) to indicate to the new node that it should start processing from where the previous node left off. By doing this, we preserve exactly once processing semantics.[**Dataflow**] In RainStorm, each task's output is stored in a separate intermediate file in our flat-file system (hydfs). Therefore, if there are 3 tasks per stage, there are 3 output files. Initially, the leader spits the input data file into chunks (virtually, ie. though line number ranges). These ranges along with filename are passed to the first stage - source. In this stage, each task processes every line in the file within its allocated file ranges, and outputs the line as a key-value pair. This is then written to an intermediate file, along with metadata, such as which stage and task processed the current line. If input to the next stage requires aggregation, then the key is used to calculate a hash value that determines which file it will write to, else it writes to the file with the same task id as itself. These intermediate files are read periodically as input by the tasks in the next stage. At the last stage of the pipeline, all outputs are written to a single file, and thus outputs are interleaved. Due to this topology, for a single record, there is only one single hydfs file 'write' from each task of a stage. Checkpoints are sent to the leader in a batch of 10 records, which is also when the processed records are committed to hydfs. **MP1:** MP1 was used to grep the logs that are generated on the VM locally, and search for specific messages to help debugging.**MP2:** Rainstorm & Hydfs both utilizes the membership and failure detection mechanism of MP2 to identify nodes joining and leaving the cluster.**MP3:** MP3 is used as persistent storage for storing input, output and intermediate data for the streaming tasks that are started and allocated in the cluster.

**Measurements**:

To measure the performance of Rainstorm, we compare it with spark. We use two datasets and run both the applications on two different scenarios (tagged simple, and complex).

[**Dataset**]

Datasets used,

- City of Champaign; city owned property
- City of champaign; Traffic signs

Link - City of Champaign GIS - We use the csv data for both rainstorm and spark.

[**Workflow**]

Simple workflow: (true for both datasets)

- Source -> Filter Transform (by a pattern in line) -> Transform (two output columns)

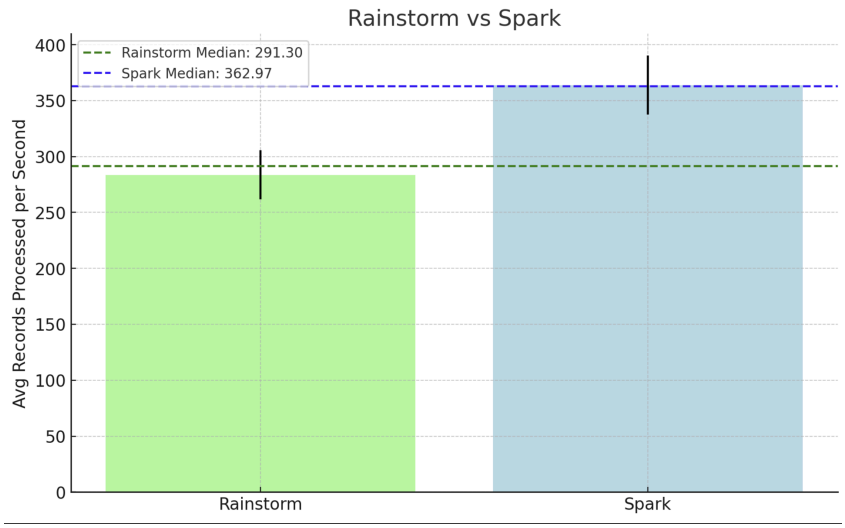
Complex workflow: (true for both datasets)

- Source -> Filter Transform (by column value) -> Count by key (another column value)

[**Execution**]

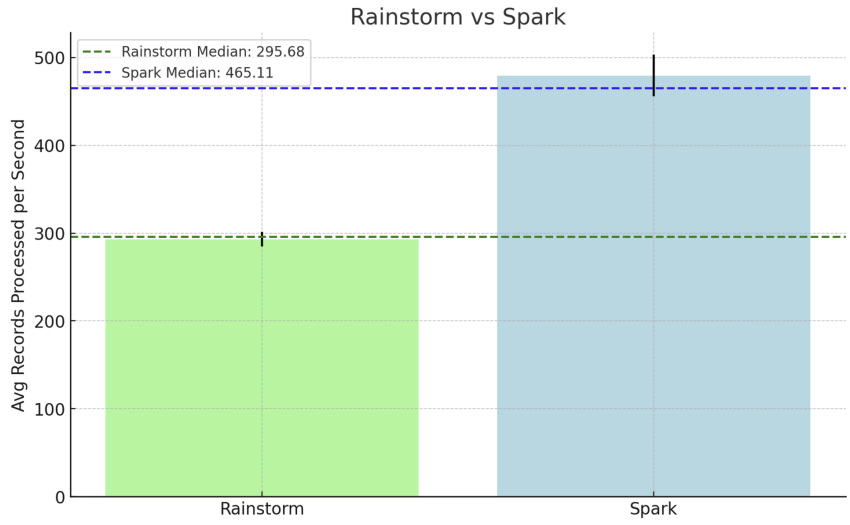
**Spark**: We set up a spark cluster on the 10 VMs provided to us. We then built pyspark files for each workflow. We also feed the input csv files and set up the configuration. We also set the batch size to 10.

**Rainstorm**: We built executable binary files for all the transform, filter-transform, source and count functions in golang. These are executed by RainStorm by each task based on the order we specify. Output and all intermediate files are stored in hydfs. Execution time is measured though go's time library.




**Spark vs Rainstorm - City of Champaign Traffic data. (Simple)**

We see that spark performs much better than rainstorm in general in all our tests. The input file was 10000 lines long. Overall the time taken was between 33 to 40 seconds over all the tests for rainstorm. However, as we show below the main delay seen is in the tasks of the first stage (source), which measured individually take up the bulk of all the time taken for the 33-40 second runs.



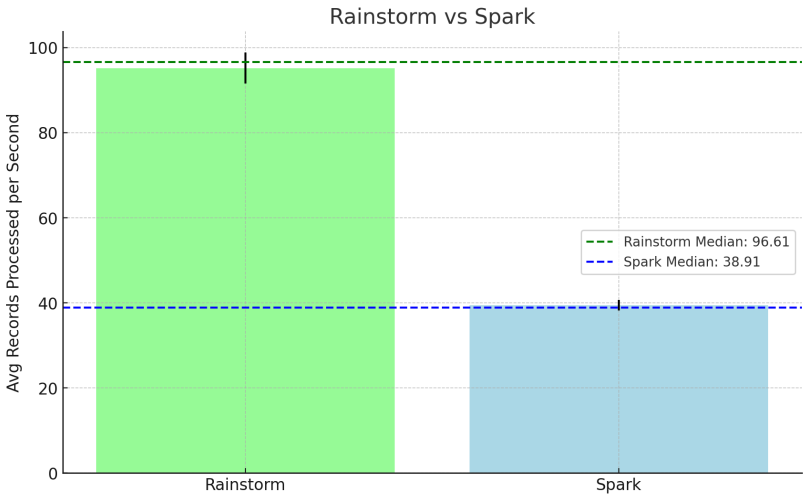

**Spark vs Rainstorm - City of Champaign Traffic data. (Complex)**

We see the same trend as above where spark beats rainstorm. Surprisingly, spark performs much better at the "complex" pipeline than the simple one seen above. We believe that spark may have some internal optimizations for aggregations between stages. Rainstorm, too, performs slightly more or less the same with or without aggregation. One interesting note is that almost all our runs in this experiment took very similar times - in rainstorm. (within a few seconds total).

```
Start time: 2024-12-08 23:02:16.814983051 -0600 CST m=+1093.041247956
Enter command: Time elapsed for Stage 0 task 1 on : 20.232339495s
Time elapsed for Stage 0 task 0 on : 20.292387401s
Time elapsed for Stage 0 task 2 on : 25.369599511s
Time elapsed for Stage 1 task 2 on : 28.397003052s
Time elapsed for Stage 1 task 0 on : 31.261765225s
Time elapsed for Stage 1 task 1 on : 31.814275416s
Time elapsed for Stage 2 task 2 on : 32.179157804s
Time elapsed for Stage 2 task 1 on : 32.386304634s
Time elapsed for Stage 2 task 0 on : 33.172345776s
Time taken for all streaming tasks to complete: 33.172374182s
```

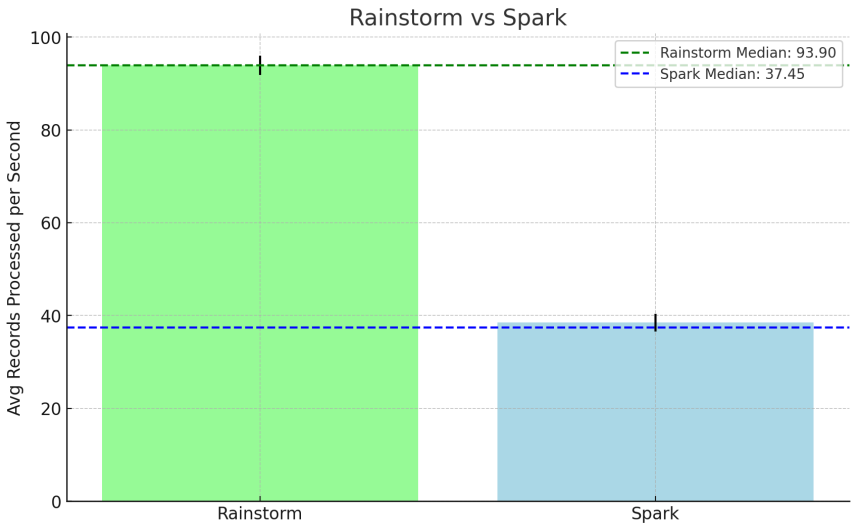
By measuring individual stages, we find that there is a good 20 seconds on average for any one task of a stage of rainstorm to complete. After which the upcoming stages complete in quicker, shorter bursts. We conclude that this may be the major part that we need to optimize as the streaming

application has to parse through all the data. Small changes in this step can lead to a bigger increase in performance.




**Spark vs Rainstorm - City of Champaign; city owned property (Simple)**

The City of Champaign data set has fewer records overall, close to 500 lines , and we see a big difference between the previous results. When the total input size is smaller, the efficiency of Spark Batch processing , Rainstorm processes nearly 90-100 records per second (with the overall time taken to be ~5-6 seconds). This is in sharp contrast to Spark, which has an average of ~40 records per second, which is less than half of what Rainstorm achieved. This could be attributed to the fact that Spark is specifically optimized for handling large-scale distributed data. Also , spark performs in-memory processing whenever possible, reducing the I/O overhead associated with reading and writing data to disk repeatedly. For smaller datasets, this approach does not boost performance as much as we notice for the previous graph.




**Spark vs Rainstorm - City of Champaign; city owned property (Complex)**

For the complex operators, we see that the same trend is maintained, and Rainstorm processes close to 95 records per second on average, as compared to ~40 records per second on average of Spark.
When comparing the graph as a whole against the results which we see with simple operators , performance of Spark and Rainstorm streaming frameworks with complex operators is almost the same as with simple operators (the difference is very less, ~ 2-3 records with the complex records being slightly worse). This is because the aggregation is taken care of by the worker nodes themselves, and nothing is sent to the leader for calculation. The only penalty would be from a few lines of hash computation based on key value. We believe that spark may perform similar computations, with better optimizations. We need to perform further analysis to confirm this finding.