Rainstorm is a distributed stream processing system with a centralized scheduler responsible for task distribution, coordination, and fault tolerance. Here's a breakdown of its system design:

1. Roles and Responsibilities:

Scheduler/Leader: The introducer node in the system acts as the Rainstorm scheduler. It orchestrates the entire stream processing pipeline, including:

- **Job Initialization:** Validating operators, dividing the input data into chunks, assigning tasks to worker nodes, and initiating the pipeline.
- **Task Distribution:** Assigning tasks to worker nodes in a round-robin fashion, excluding itself. It requests a free port from each worker for each assigned task.
- **Routing:** Maintains routing information (**operatorToVmPorts**) to direct tuples between stages based on key hashing. The scheduler acts as the central routing point for all inter-task communication
- **Fault Tolerance:** Handling node failures by reassigning tasks from the failed node to other active workers. This involves selecting a suitable replacement node (**findNodeWithFewestTasks**), requesting a free port on that node, and updating the routing table.
- **Output Collection:** Receiving the final output tuples from the last stage of the pipeline and writing them to the designated output HyDFS file. And outputting to the console. The scheduler also acknowledges these final tuples.

Worker Nodes: All nodes in the system, except the leader, can function as workers. They execute the assigned tasks and communicate with the scheduler and other workers.

- **Task Execution:** Workers listen on assigned ports for incoming tuples. Upon receiving a tuple, they:
- **Screen Input:** Check a local buffer (UIDBuf) and persistent log file to discard duplicate tuples resulting from retransmission or task reassignment.
- **Apply Operator:** Process the tuple using the assigned operator.
- **Send Output:** Send the processed tuple(s) to the next stage in the pipeline, using routing information from the scheduler.
- **Acknowledgement (ACK):** Implement an ACK mechanism to ensure reliable tuple delivery and processing. Workers send ACKs back to the previous stage upon reception of a corresponding ACK from the subsequent stage.

2. Data Flow and Pipeline Execution:

- **Input Data:** Rainstorm reads input data from a HyDFS file. The scheduler divides this file into chunks, assigning each chunk to a different source task.
- **Source Tasks:** Source tasks read their assigned file chunk line by line, generate tuples, and send them to the next stage in the pipeline. Each tuple is assigned a unique ID based on filename and linenumber
- **Intermediate Operators:** Worker nodes execute user-defined operations on the incoming tuples. Operators can be stateless or stateful. Stateful operators maintain their state in a local state file, which is restored upon task reassignment. Operators can also be filters, dropping tuples that don't match specified criteria.
- **Output Collection:** The final stage operators send their output tuples to the scheduler. The scheduler writes these tuples to the output HyDFS file and sends ACKs back to the last stage

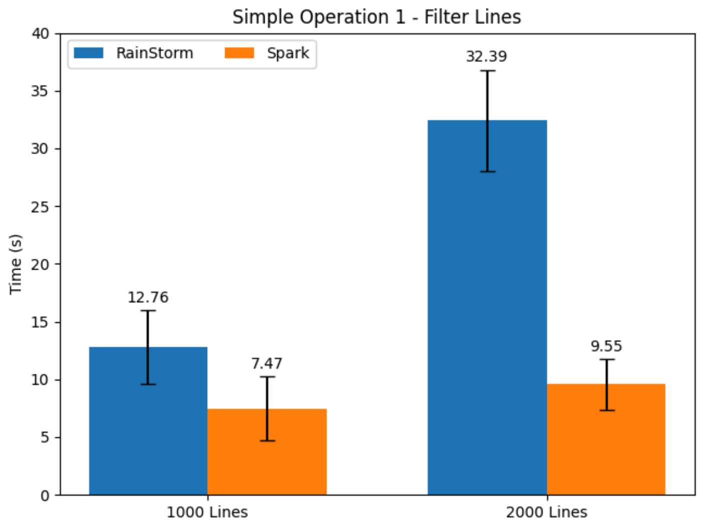
3. Fault Tolerance:

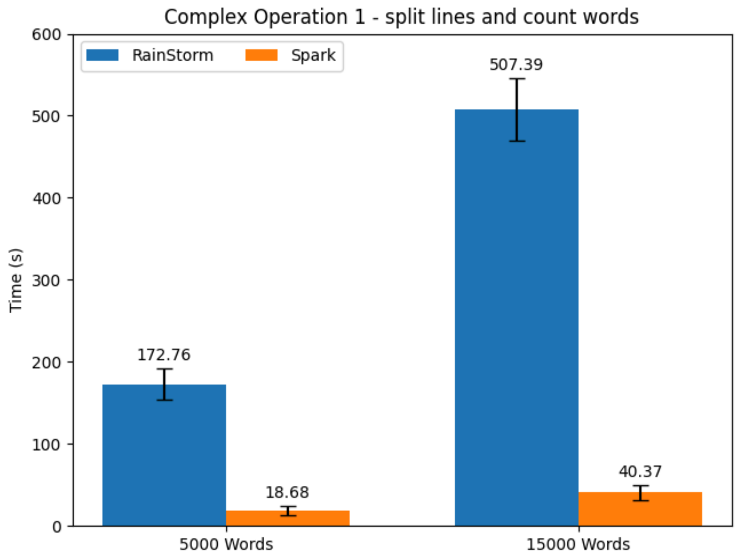
Task Rescheduling: When a worker node fails, the scheduler detects the failure through its failure detection mechanism (using pings and timeouts). The scheduler then reassigns any tasks that were running on the failed node to other active workers.
State Recovery: Stateful operators persist their state to a local state file. When a stateful task is reassigned, the new worker restores the operator's state from the corresponding state file.
Duplicate Handling: The screening mechanism at each worker node, using the UID buffer and persistent log, prevents duplicate processing of tuples during retransmission or task reassignment.

Performance Testing -

Dataset 1

For dataset 1, we used the shakespeare sonnets dataset





For our simple operation, we filtered for lines with the word “shall” and outputted the first character of each of these lines. Perhaps not the most useful operation, but a benchmark nonetheless. For our complex operation, we performed the operation specified in the MP4 document; splitting lines into words, and counting the number of each word.

For this first dataset, RainStorm performs significantly worse than Spark. One potential reason for this is the way in which we divide up our file data initially; we assign each source node an even chunk of the input file, but because the key for the output of the source nodes is specified to be filename:linenumber, and our line numbers for evenly divided chunks will tend to hash to the same node (i.e. line 0 and line 3 will hash to the same node) this tends to make all of our source outputs hash to a single op1 node. This leaves us with 1-2 idle nodes for the majority of the source operation, where Spark presumably divides up its data evenly among its worker nodes.

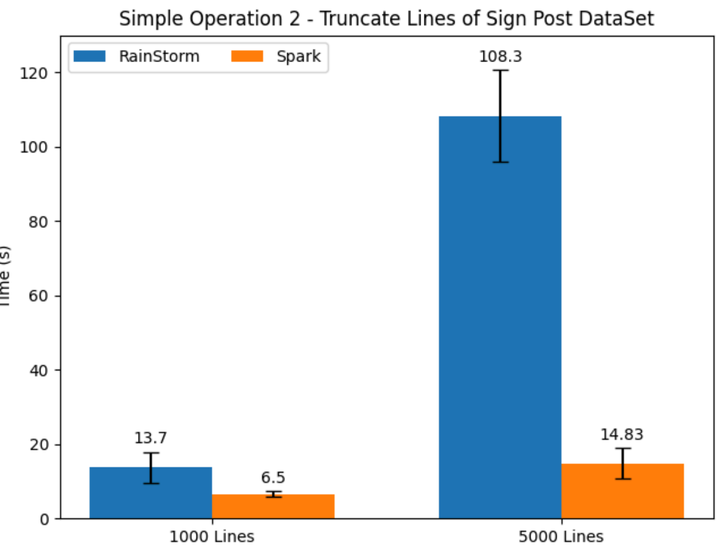
Another distinct characteristic of the above plots is the markedly worse performance of RainStorm for a larger input dataset. Because tasks check over every entry of their respective log file before performing an operation and write to their log file every time an operation is completed, tasks need to check over an ever-growing set of data to ensure exactly once delivery, resulting in ever-slower operations

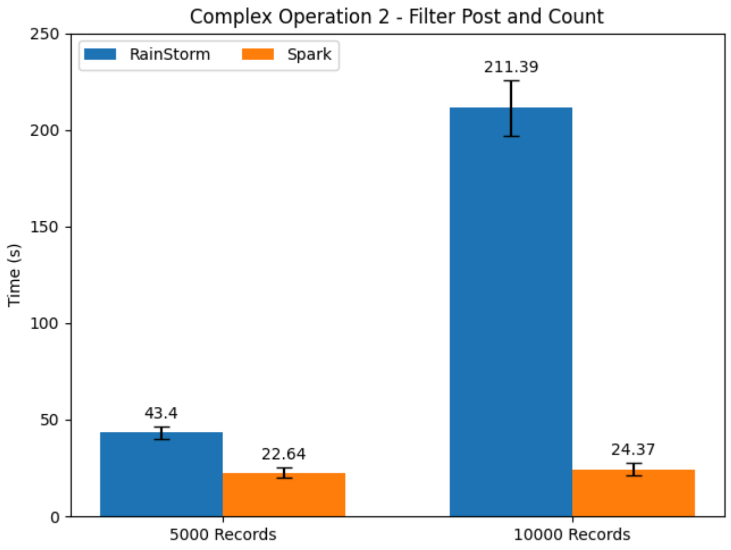
A knock against RainStorm over Spark for this particular dataset is the downright abysmal performance on the complex operation above, where Spark nearly is approximately an order of magnitude faster than

RainStorm. This can likely be attributed to the fact that this complex operation requires us to maintain–and accordingly check–state for all operations, adding additional overhead to the final layer. The particular operation we chose to run as our complex operation here (split lines and count words) is also quite bad for performance, because for this operator sequence the number of records grows from op1 to op2 with word splitting, giving us all the more data to log and check over.

Dataset 2

For dataset 2, we used the Demo data set of Sign Posts





For our simple operation in the above plot, we filtered for lines in the sign post dataset which contained the word “No Left Turn”, and outputted the “Category” column of each of these records. For the second operation, we counted the number of each category among records with the Unpunched Telespar Sign Post attribute.

For the above plots, we note similar performance to the sonnets dataset for our simple operation, with worse performance for the 5000 line Sign Post case over the 2000 line Shakespeare presumably due to the aforementioned infinitely growing log files. However, we note a sharp improvement in the complex operator performance over the Sonnets dataset (though RainStorm still loses quite handily to Spark). This is likely because instead of splitting lines and adding more pieces of data for the final layer to process, the complex operation applied to this dataset filters out posts, reducing the number of records supplied to the final layer and reducing the time taken to iterate over the final log file accordingly.

Overall, while designing RainStorm, our focus was not on creating the most performant streaming framework, we instead on getting things right and then iterating towards making things faster. We expected RainStorm to be slower than Spark, and a gap this large is somewhat expected when competing against one of the top streaming frameworks available in the market today.