MP4 Report

By Sagar Abhyankar (sra9) and Aditya Kulkarni (aak14)

Architecture & Design

The MP is implemented using go-lang without using the traditional RPCs, instead, we use sockets in go. In this MP we have built RainStorm, a streaming platform like Apache Spark with batching. We follow exactly one semantics even in the case of failures. The MP uses HYDFS from MP3 for all of its log and data recovery requirements, which inherently provides fault tolerance in case of crashes by recovering files. The HYDFS further down the line uses MP2's SWIM protocol in suspect mode for failure detection. RainStorm communicates with HYDFS using a self-TCP pipe over which updates are relayed in both directions, eg operation requests such as get, append, or merge and failure notifications such as addition and removal of nodes. We use a separate dedicated port for the operations of RainStorm using which RainStorm can communicate tasks and updates necessary in the RainStorm layer. At the core RainStorm models each task as a container, each task has around 15 variables that control and dictate the metadata, input/output nodes, operator names/states, etc. This vastly generalizes the architecture and hence RainStorm can perform a range of queries just by changing the binary files of the two operators. RainStorm also can tolerate up to two failures which is limited solely by the underlying HYDFS layer replication number. In case of a failure/s, once the leader is notified, it quickly reschedules the task on the VM running minimum tasks at that point in time. Each task has a doReplay Boolean variable, which when set to true indicates to the new task that it first needs to recover the previous state before continuing the processing, the leader simply needs to first merge the task logfile of the lost task and then reschedule the task with the doReplay=True, then send resume messages to any task that had paused because of this failure. In case of multiple failures, the leader schedules tasks in topological order to ensure proper dependency resolution. The state recovery is done with the use of log files, each running task has a log file associated with it where it continuously appends records that are processed, but not yet acknowledged by the immediate next stage as a buffer. If the failure detection layer (MP2) notifies about a failure node to which the current task was sending values, the task automatically pauses and does not relay any further tuples till the Leader tells it to resume the task (with a new output node address). When resuming, the task replays all the entries in the buffer to ensure that the new task does not miss any records. Each stage forgets about the records after they have been acknowledged by the next stage. The last stage also maintains a meta file along with the task log file to ensure that no duplicate records are processed. Stateful operators are expected to do state maintenance by themselves (by having a state file).In cases where more than one task runs on the same VM, we provide multiple state files for an operator to prevent state overwrites. In case of a crash, this state file is repopulated using the last buffer from that particular task and the processing continues. We implement batching to improve performance by reducing I/O tasks and also prevent excessive memory usage resulting from the otherwise continuous task log files appends. The batch size for the source stage is 300 and for the second stage, it is 250. Finally, each of the VMs can run multiple tasks on it, all the communication between VMs is done using the NodeID-TaskID pair as the key (NodeID from Hydfs).

Comparison With Spark Streaming:

For the comparison we chose the following datasets.

1. https://giscityofchampaign.opendata.arcgis.com/datasets/cityofchampaign::parking-meter-spaces/about (parking meter spaces champaign, extrapolated to 20,000 lines)
2. https://www.kaggle.com/datasets/datasnaek/chess (Chess Game Dataset from Lichess, extrapolated with permutation and combinations to 100,000 lines)

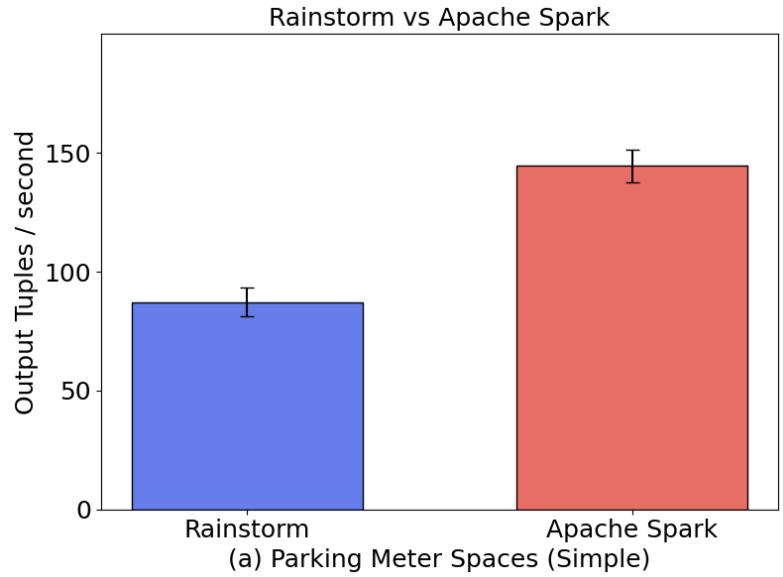
Following are the queries and the comparisons when run against Spark Streaming.

MP4 Report

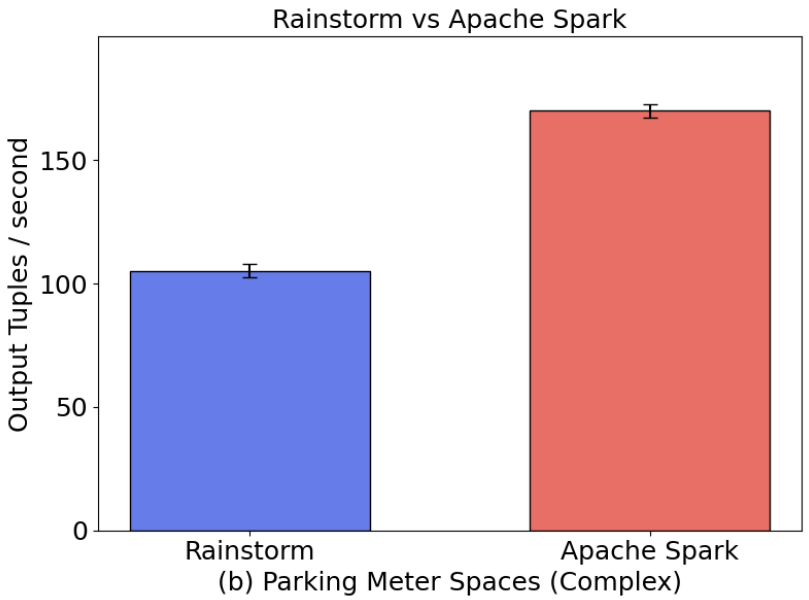
By Sagar Abhyankar (sra9) and Aditya Kulkarni (aak14)

a. Scenario: Champaign County Parking Meter Spaces
   1. Simple: Active ="no" & blockNum = 1000
   2. Complex: Number of parking meters per zoneid with rate less than 0.5

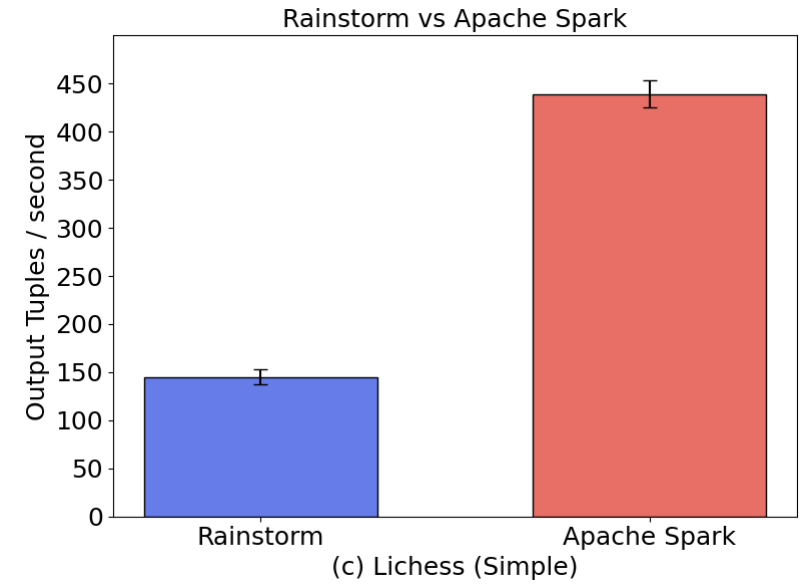
b. Scenario: Lichess
   1. Simple: Winner = White & victory_status=Mate
   2. Complex: Winner = White & victory_status in [Mate,Resign,Out of time] & turns > 40 & opening name in [Scandinavian Defense Sicilian Defense Indian Game Caro-Kann Defense Italian Game] GROUP BY opening name.



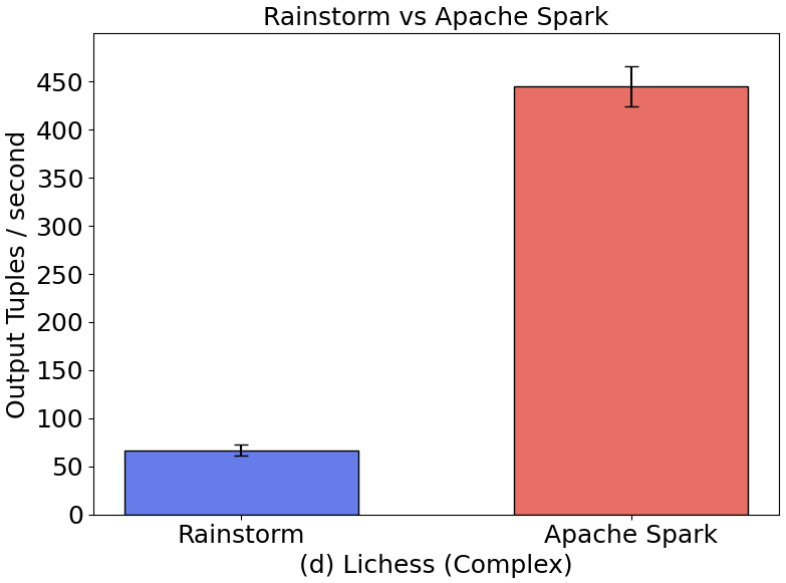
(a) Parking Meter Spaces (Simple)



(b) Parking Meter Spaces (Complex)



(c) Lichess (Simple)



(d) Lichess (Complex)

To ensure fairness in comparison, we have set the batch size to 100 and number of partitions to 3 in both RainStorm and Spark Streaming. We have calculated the output tuples per second by dividing the total number of records by the metric of total time taken for the query processing, this

gives us the output tuples/second. There are significantly more records (3x-4x) to be processed in the second stage in the Lichess dataset compared to the other dataset. From the graphs we see that Apache Spark requires a constant amount of time to schedule the tasks and display the result of the task irrespective of the number of records if further optimized RainStorm could beat Spark streaming for a lower number of records range as spark streaming will still require nearly the same amount of time to do the initial scheduling, but RainStorm being a more lightweight and targeted (barebones) streaming platform, it requires less time to handle all the tasks planning and setup. This is evident from (a) and (b) where the number of records to process for the first dataset is about 20000, and RainStorm and spark streaming's performance is comparable. We also observe that simple queries on average take more time to process than complex queries but, on the counter, a single stateful operator invocation (used for complex queries) in RainStorm is slower than a stateless one. So, what is the issue? In the scenarios we have tested, simple queries have a lot more tuples to process in the second stage compared to the complex queries, hence although the simple sequence of operators should be faster, there are a lot more tuples to be processed making it effectively slower than the complex sequence of operators.

For Spark Streaming, we implemented ReadStream and WriteStream operations, even for a single CSV file, to maintain streaming capabilities and ensure fair comparison with RainStorm's streaming architecture. While traditional read/write operations would suffice for static data, this approach enables seamless handling of both static and real-time data sources without architectural modifications. The streaming paradigm ensures consistent testing conditions across both systems while maintaining future scalability.

From these tests, we observed that RainStorm's performance is vastly affected by the total number of tuples processed, which has a direct correlation with the amount of times RainStorm interacted with the HYDFS and also the number of times an operator was invoked. In our HYDFS implementation, we store appends as logical chunks and they are not merged until necessary, even if this makes the write speed fast, to ensure that RainStorm doesn't pace too ahead of hydfs, which may cause it to lose the fault tolerance properties, we implement a static 1 second time delay every time RainStorm has to interact with hydfs. This ensures that when a task on RainStorm stores buffer on hydfs, it is guaranteed that the write was actually made and not just queued (volatile) on the underlying hydfs layer. If we removed the static one-second delay, RainStorm achieved a peak performance of about 650 tuples/ second, but it is important to note that the hydfs layer would lag much behind and hence this isn't a stable output speed. Batching helped to improve performance significantly while still ensuring fault tolerance. We were limited to processing files as large as 30-40mbs because the appends created even with batching for the task log files (buffers) on rainstorm would overwhelm the VMs memory capacity, but the runtime for both was several minutes and hence the output/tuples is a stable measure nonetheless.