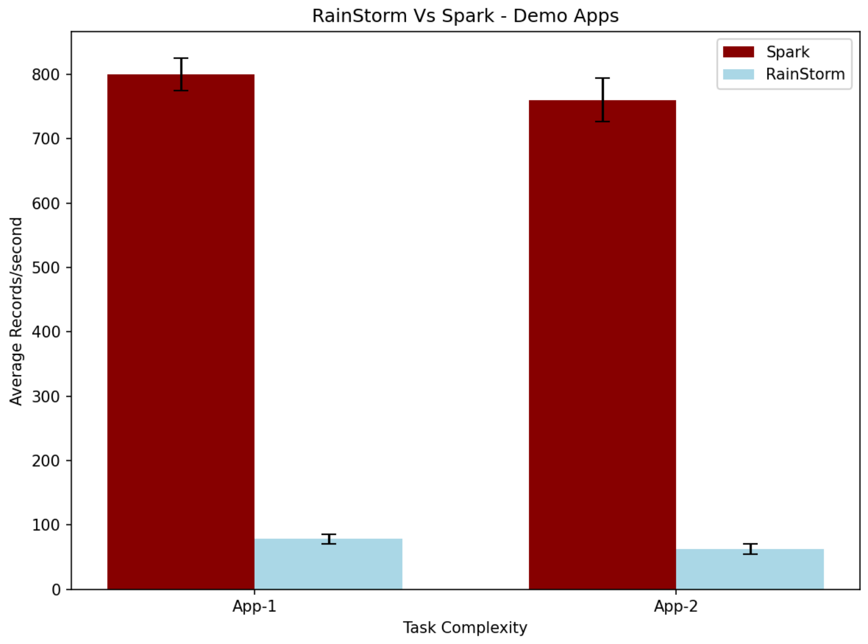
CS 425 / ECE 428 MP 4 Report

Andy Min (akmin2), Satyam Singh (satyams2)

Design

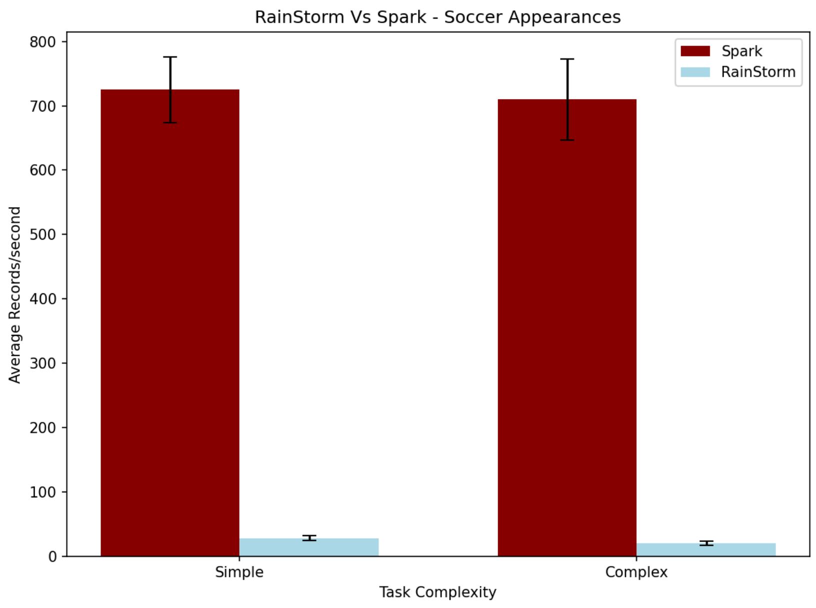
Core: When RainStorm is invoked, it designates the machine that invoked it as the leader. The leader then assigns tasks to the workers based on the number of tasks per stage. The leader stores the assignment and also makes sure to pass this information to the workers. The source workers start processing the distributed file line by line and send the outputs to the corresponding op1 worker based on the hashed key. The source workers only process lines that hash to their index/assignment to prevent double processing. Each task is tagged with an unique ID, which is checked in op1 and op2 to prevent duplicates. Based on the current stage, either op1.exe or op2.exe (both Python scripts that process key-value tuples in this case) is invoked. The task is then sent to the next stage through an RPC request, which retries until it is successfully completed to ensure no data is lost. Finally, the outputs of op2 are sent to the leader to be printed and written to the output DFS file. We also make sure to reset all states before running RainStorm so states from old runs don't impact the current run.
Failures: Failures during op1 and op2 are handled by logging three main states to the corresponding log file: received, outputted, and acknowledged. A callback registered with the leader is triggered when a worker fails to assign a new worker, which parses the log file to recover the previous state of the failed machine. The new worker is selected by checking the number of tasks assigned to each machine, and the one with the least work is chosen. This new worker processes the distributed log file to reconstruct the old task states and resumes from where the failed machine left off.
State Recovery: For counting operations, every time a key's count is incremented, the update is logged to the log file. In the event of a failure, the process follows the steps described above, aggregating all key increment updates and storing them in a dictionary. When a key processed by the failed worker needs to be incremented, the new worker starts from the logged count instead of 0.
Key Decisions: We decided to use RPC to simplify acknowledgments and make remote function calls easier. Batch logging was also implemented, where updates are appended to a local file and synced with the DFS log file every 100ms.

Spark Comparison



The graph refers to the TrafficSign dataset used for the apps made for the demo.

From the graph above, it's evident that our RainStorm program is significantly slower than Spark, which aligns with our implementation limitations. Several factors contribute to this performance gap. Firstly, we have relatively large timeouts configured for suspicion (4 seconds) and RPC acknowledgments (500 ms), which greatly affect our efficiency. During these timeouts, other processes may be blocked, as we continually print debug messages to the terminal. This constant logging adds overhead to the system. Additionally, our Hybrid Distributed File System (HyDFS) logic places a significant strain on the system. Currently, each append operation is saved as a separate file. This design means that every operation involves opening or creating a new file, which becomes increasingly expensive when there are multiple appends to log files. To improve efficiency, we need to redesign HyDFS to handle appends more effectively—potentially by consolidating all appends into a single file. This approach would reduce the overhead of file creation and improve system performance. When comparing the more complex App-2 to the simpler App-1, we observe a slight decrease in the records processed per second. This is expected because each record in App-2 undergoes more intensive processing, which naturally slows down the program. For more complex tasks, we also need to maintain state information to handle potential failures. This requires additional logging and the use of extra data structures, further contributing to the performance drop. Interestingly, the observed dip in performance between App-1 and App-2 was less pronounced than anticipated. This makes sense given that the complexity difference between the two tasks is not drastic in our case.



This data set is the appearances.csv from the Football Data from Transfermarkt from Kaggle. This csv contains 1,048,576 entries and is roughly 129.75 MB.
Link to the data set: https://www.kaggle.com/datasets/davidcariboo/player-scores

Similar to the previous graph, we can observe that Spark outperforms our version of RainStorm in terms of speed. Many of the reasons discussed earlier also apply in this scenario, but the structure of this dataset significantly influences the observed performance differences. Specifically, this dataset contains a small number of columns but a much larger number of entries (rows). Given that our current RainStorm design is not fully optimized, the higher frequency of data flowing through the system exposes the weaknesses of our implementation more clearly. For instance, the sharp increase in the number of records leads to more frequent appends to distributed log files, which in turn results in the creation of a substantial number of local files. This proliferation of local files drastically slows down RainStorm’s runtime, particularly in the latter stages of execution. By that point, the program has accumulated a large number of files, introducing significant overhead that hampers its efficiency. On the other hand, Spark demonstrates relative stability even with a large number of entries. This stability can be attributed to Spark's mature design and robust architecture. Spark efficiently manages its distributed operations through mechanisms like partitioning, in-memory processing, and optimized file handling, minimizing the impact of data size on its performance. In contrast, RainStorm suffers from a noticeable dip in performance under these conditions due to its reliance on less efficient file operations and higher overhead.