MP4 Report

by: Nicolas Yepes (nyepes2), Ivo Kusijanovic (ilk2)

Architecture:
Rainstorm first must choose a leader, which we assume will not fail. The leader will receive the request to start a job. It will look for tasks to be the readers, it will first try nodes that have the local file if none are available it will look for the one with least number of tasks. Then it will look for nodes with the least number of tasks to run the next stages. The nodes that should run the tasks will start a subprocess and keep references to stdin and stdout of the processes. The reader nodes will hash the line number and determine which node to send. Although all nodes run a failure detector, the leader will send a packet telling nodes what nodes to send the data to. Specifically the leader asks to change the sending sockets of each vm *only* and let the receiving ones timeout an exit. Eventually the restarted socket will communicate back with the task. In order to distinguish between a single vm running multiple tasks we keep track of what stage the vm is working in. If it has to work in the same stage with multiple instances the system will consider it as if it was one. Each vm has a thread safe dictionary with the ids of what it has already processed. To guarantee exactly once. Additionally we must append and merge files often in order to recover from failures. This file updates after the data is sent to the next vm, and allows the new vm to know what it has already processed before. Stateful operators will operate very similarly by collecting the changes, rebuilding the state and feeding it back to the process.

Programming Framework:
In order for a program to run with RainStorm it must take input from stdin and output it to stdout. The first value the program must output is a way to distinguish between Stateful or Stateless programs,it does when the user lets know on start what it is. This program will encode in a json the input that it ran and the outputs from that input. Another feature is that the program should be able to run for an indeterminate long amount of time until without it exiting (assuming correct input) which is done by continuously reading from stdin. Most of the framework details were abstracted away, and the user is expected to write a function that takes in a key and a value and produces a list of outputs. These outputs are then sent back to the main process that will assign them ids and send them to the next stage. Stateful operators should only output the keys with the new values as a list of key value pairs.

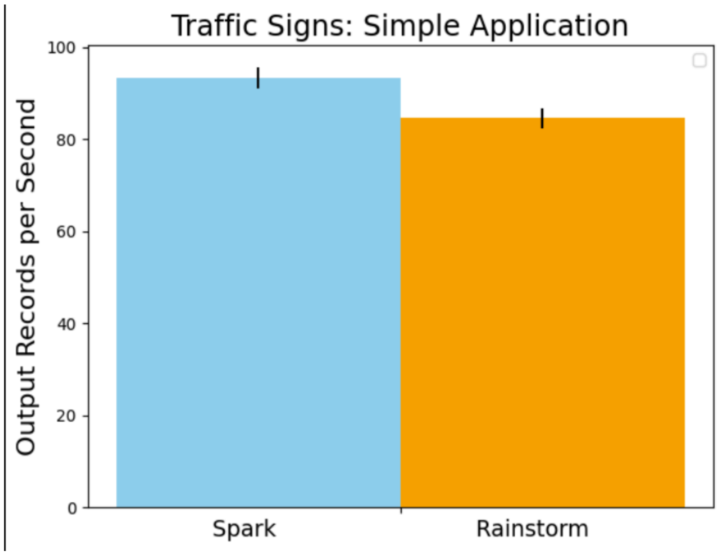
Rainstorm vs Spark:

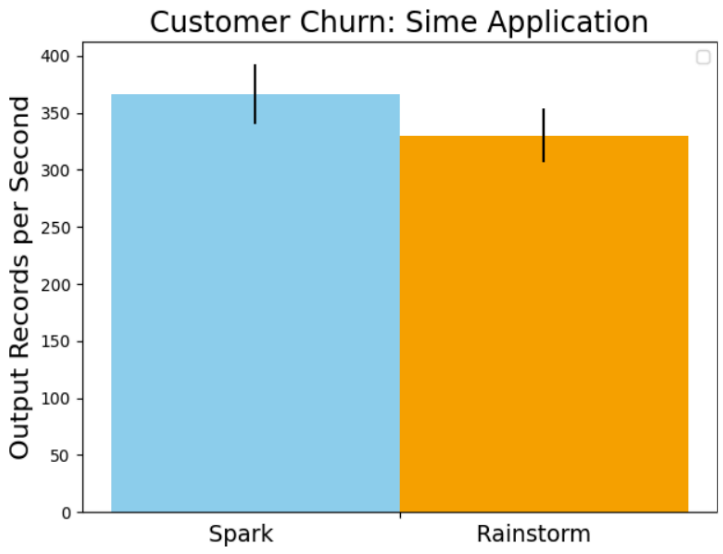
The datasets used can be found at Traffic Signs and at Customer Churn. Both of these datasets are stored in .csv format and have 10,000 entries. Now, we run the same applications (i.e. one simple and one complex) on both datasets.

To make a fair comparison we use output records per second metric, which in other words is the number of key-value pairs out to the file over the time it took from submitting the task. Furthermore, we use the same topology of 3 readers, 3 intermediate workers, and 3 final workers. In both we make checkpoints for failure recovery purposes.

Additionally, we used Microsoft Azure to store the input datasets so that all worker nodes of the Spark cluster had access to the dataset. Now, this is a setup for Spark unlike RainStorm who makes the readers the VMs which hold a replica of the file.

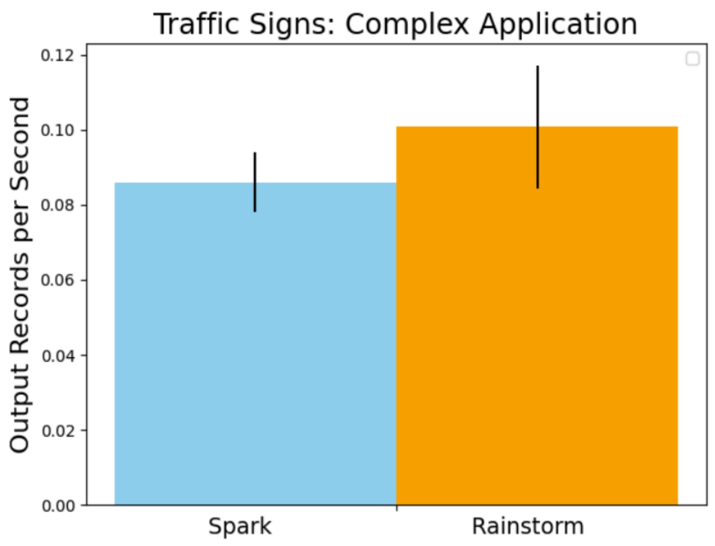
Moving on to the actual testing, the first application (i.e. simple) entails doing two operations; the first is a filter by pattern on the entry, and the second is a column selection. For the Traffic Signs dataset we used the pattern "*Unpunched Telespar*" and the selected columns OBJECTID, and Sign_Type. For the Customer Churn dataset we used the pattern "*France*" and selected the columns CustomerId and Surname. See bar charts below

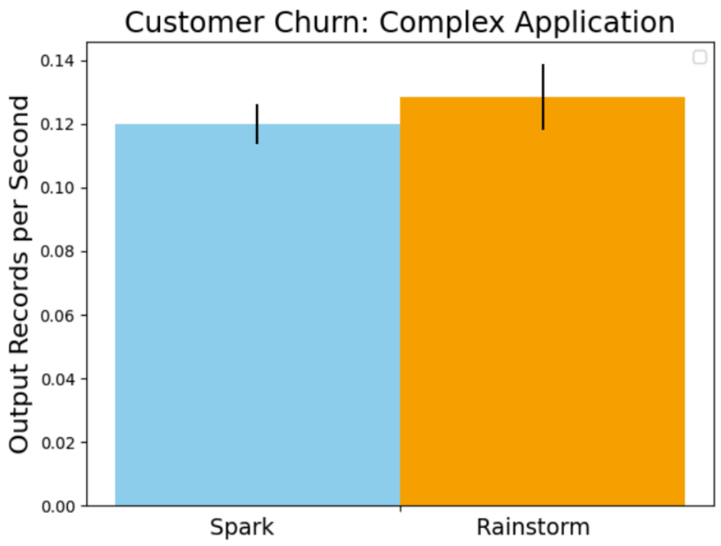




As we can see above, Storm has a better performance than our system in terms of output records per second. This is to be expected as Spark Streaming is meant to be scalable. This may also suggest that our current system isn't well suited for scalable applications with 100,000+ entries. We believe that this problem spans from the massive overhead for communication and failure detection and use of multiple threads in our system. Other approaches should be explored to make the system slimmer and as scalable and perhaps better than Spark. Additionally, we notice that both Spark Streaming and Rainstorm perform better on the customer churn dataset. This is most likely due to the smaller quantity of columns in the dataset and lack of escape characters. This makes streaming faster as less bytes have to be sent over the network.

The second application (i.e. complex) entails again doing two operations; the first is filtering entries based on the value of a selected column, and the second is an aggregate count based on the value of a selected column. For the Traffic Signs dataset we filtered entries based on the Sign_Type column being equal to Stop, and then doing an aggregate count based on the value of the category column. The first is filtering the entries based on the Gender column being equal to Female, the second is doing an aggregate count based on the IsActiveMember column. The results are showcased below.





Although we had a worse output record per second in the first application, we outperformed Spark in the complex application. Although this seems promising, we believe that this is a product of having very few output records. Spark Streaming is very scalable so it takes about the same time in outputting 2 key-values or 30,000 key-values based on the chained operations. Therefore, it suffers from performance using the output records per second metric when a small amount of records are expected. Still, both systems perform best on the Customer Churn dataset.

Also note that our system outputs all changes to the aggregate as we don't use a barrier. Therefore, to make it a fair comparison with Spark we just counted the unique record with the maximum key. This might have worsened the performance of our system due to the multiple writes to the HyDFS.

Overall, we have made two finds. The first is that both systems perform better when given csv datasets whose rows have few columns and lack escape characters. This is expected as less bytes have to be passed over the network. The second and perhaps more interesting finding is that our system outperforms Spark for small expected output records. However, we believe that this is due to the scalable nature of Spark, which sacrifices excellent performance on smaller output records. Therefore, if we have 100,000+ output records we definitely expect Spark to beat Rainstorm by a landslide. On a final note, we have to consider some shortcomings of our testing. Using Azure might have negatively affected the metric for Spark Streaming, but it was needed due to the setup it required. More testing should be done to setup spark to use out HyDFS.