Design:

Program flow:

We run two separate processes HDFS which is our MP_3 and RainStorm which is responsible for the tasks in MP_4. This allows us to avoid HDFS failure, and do not have to deal with coordinator blocking when quorum is not reached due to node failure. We spawn a failure detector instance inside RainStorm.

When RainStorm is invoked, we have a leader that assigns num_tasks for each stage and sends metadata about each task to workers currently running the fewest tasks. Upon receiving these tasks, workers start executing them. Op1 and op2 tasks do very little in the main task thread but listens for incoming records and processes them concurrently.

Exactly once:

Each task stores its output `{'unique_id': unique_id, 'key': key, 'value': value}` in `f"{RAINSTORM_DIR}/{task_id}_output.log"` on HDFS in batches of 10. During recovery, each task looks at this file and dishes out records in the file to downstream tasks. During failures, we ensure that recovery starts in every task including those that are newly assigned, by signaling the recovery threads after failure handling. Duplication is prevented by assigning a unique id to each record in the task. Identifying strings can be appended to the end of the id if needed to create new ids. This ensures that when a task fails, the recovered records will always have the same unique ids as they reach each stage, therefore we can just filter through the output files for duplicates, and discard them. We also send back to the upstream task when the record is processed, which we defined as finishing logging and sending the output downstream, courtesy to the homework spec. However we do not do anything with it since files are append-only in hdfs.

State recovery:

Stateful tasks uses its output file to aggregate results. Since we ensure no duplicates in the previous step, we simply read the output logs and aggregate on the record's key. For example, when counting words, we simply load the output file which should have all the previous <word, count> and we count all of those plus 1 to be our value.

Failure and node rejoin:

We route callback functions to a failure detector that triggers during failure and rejoins. We already mentioned failure handling. For node rejoins, we simply store the node in Leader's memory so it can consider it for future tasks.

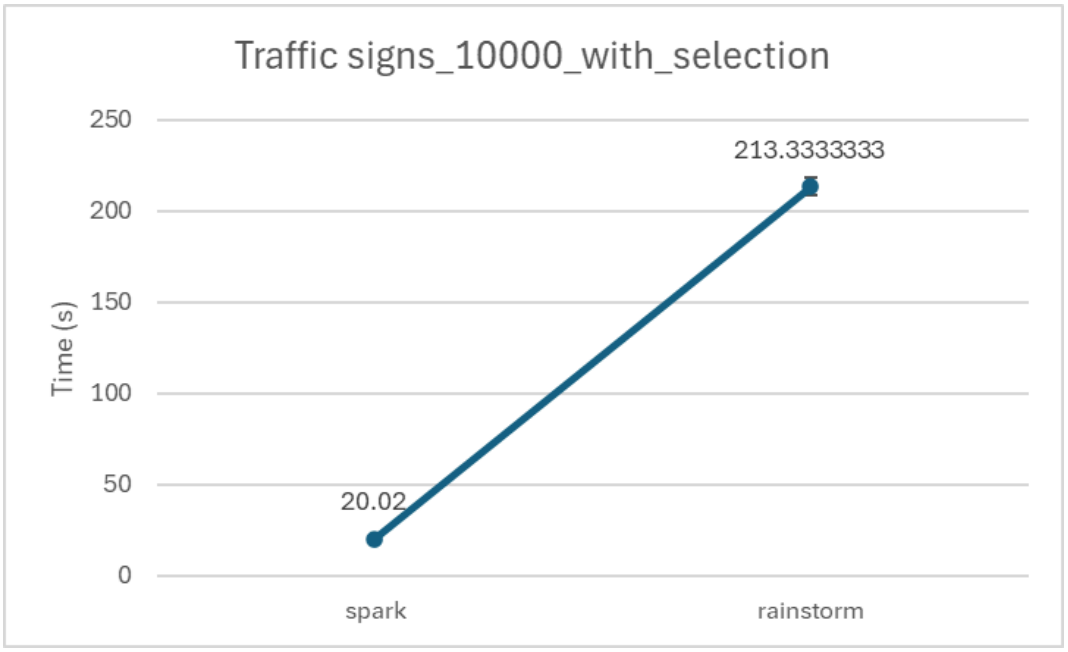
Generality:

We generalize the applications by letting them be any python3 executable that prints KEY: key, VALUE: val pairs to stdout which we will track and parse back to key, value.

Spark vs RainStorm

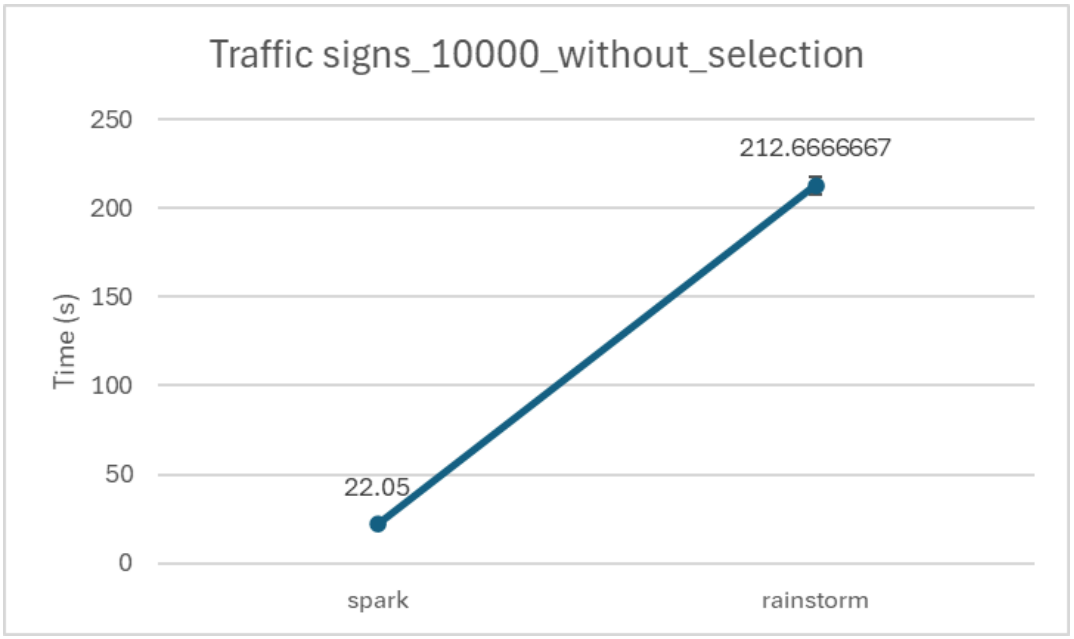
Scenario 1: DEMO application 1, complex

We take the dataset used in the demo, parsed to 10000 lines and apply application 1 in the demo to it. This is 2 operations, therefore complex.



As shown in the graph, spark does significantly better, probably because we are running out of bandwidth in our implementation and had to cap the rate at which source distributes new records, which is the main bottleneck for runtime. The extra bandwidth is the result of an additional layer of tcp traffic between HDFS and RainStorm, which could be avoided if HDFS is actually fail tolerant. So we need to find a way to handle quorum blocks.

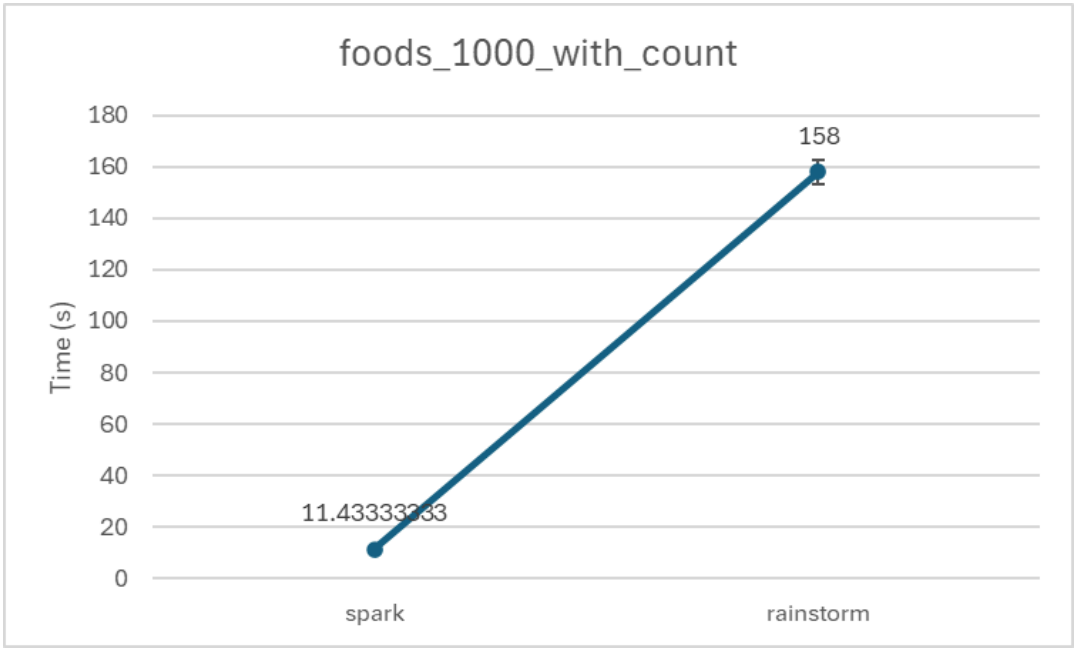
Scenario 2: DEMO application 1, simple
We omit the OBJECTID, Sign_Type selection, hence ‘simple’.



The performance does not improve in spark, which suggests that this extra stage is not a bottleneck in spark. Interestingly, when we ran py spark locally, the performance is faster than both Scenario 1 and 2. The bottleneck could be how workers communicate results to the Leader.

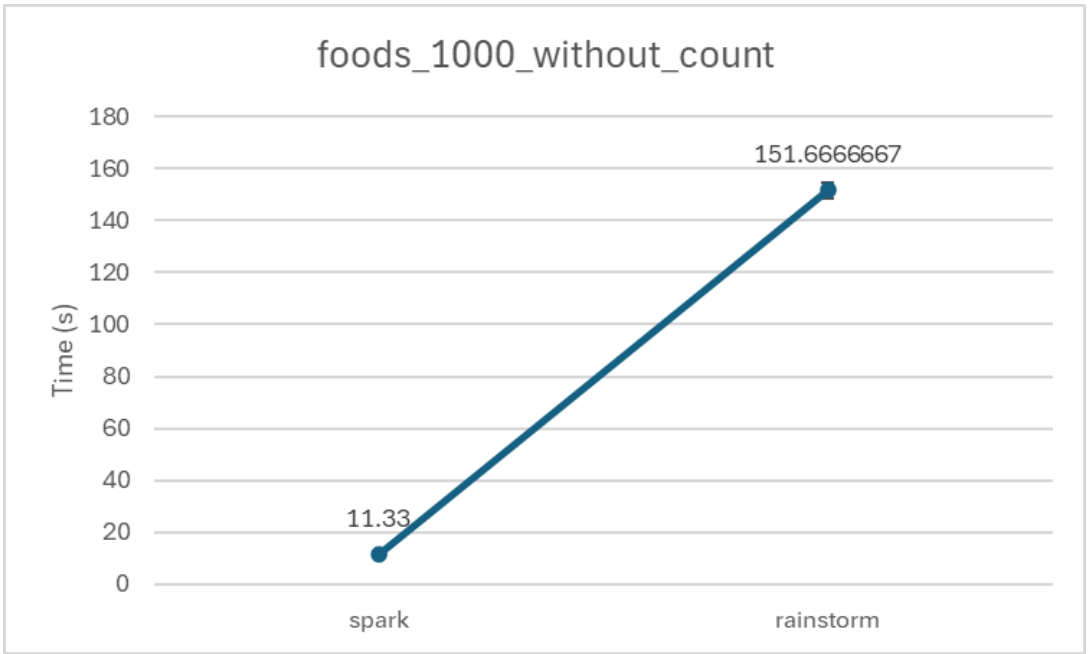
Scenario 3: SNAP review dataset, complex
We obtain a dataset from https://snap.stanford.edu/data/web-FineFoods.html and parse to 1000

records (approximately 10000 lines) to keep the load consistent across scenarios. We filter lines for review scores and count the 5 star reviews, which are 2 stages.



Again, spark is faster. We noticed load balancing issues on RainStorm, where some op2 tasks are more encumbered because the words assigned to them have more counts. We used sha1 hash for assignment which worked well for the demo dataset but does not work well here with one of the op2 nodes being mostly idle.

Scenario 4: review dataset, simple
We omit the count stage and just output the review score lines, hence ‘simple’.



Again, RainStorm seems to benefit for a little from the saved process execution in stage 2 (we apply a dummy program that just prints key, value as is). Spark does not seem to benefit, probably due to the cluster overhead that we previously mentioned.