CS 425 MP4 Report

GA6 – Saurabh Darekar(sdare1) & Ankit Chavan (auc3)

Design of the Stream Processing Application

1. **Streaming:**
   - **Initialization:** A designated leader reads the Rainstorm command and, based on the membership list, allocates tasks to members. The members then create worker tasks, which are managed by a Worker Manager at each member. These tasks handle the streaming process.
   - **Task Allocation:** The leader assigns specific roles to members, such as reading lines, filtering, aggregating, etc. Once roles are assigned, the leader signals all members to start processing.
   - **Output Display:** The final tuples processed by the aggregator are continuously displayed on the leader's screen.
2. **Logging and Batching:** Each worker task sends logs to HyDFS. Aggregators also send their current state and the data processed thus far to HyDFS. Data and logs are appended to HyDFS in batches to avoid excessive load on the machines.
3. **Executing Custom Code:**
   - **Dynamic Classes:** The application is built in Java and uses dynamic classes to pass custom code or executables. Since Java doesn't handle executables directly, we compile Java code and pass the generated class files to the application.
   - **Processing Key-Value Pairs:** The application sends key-value pairs to these classes and receives key-value pairs in return.
4. **Fault Tolerance:**
   - **Failure Detection:** The application relies on the MP2 Failure Detector to identify failures.
   - **State and Data Management:** Logs, data, and worker task states are stored in MP3 HyDFS.
   - **Task Recovery:** When the leader detects a failure, it initiates new worker tasks on other nodes and transfers the failed node's data and logs to the new tasks.
   - **Replaying Lines:** The leader reviews the logs to determine where the source tasks should resume processing and instructs them to replay the required lines.
   - **Duplicate Handling:** Aggregator nodes discard duplicate tuples that are generated after source replays some lines, ensuring that the final output remains consistent and adheres to the "exactly-once" processing guarantee.

CS 425 MP4 Report
GA6 – Saurabh Darekar(sdare1) & Ankit Chavan (auc3)

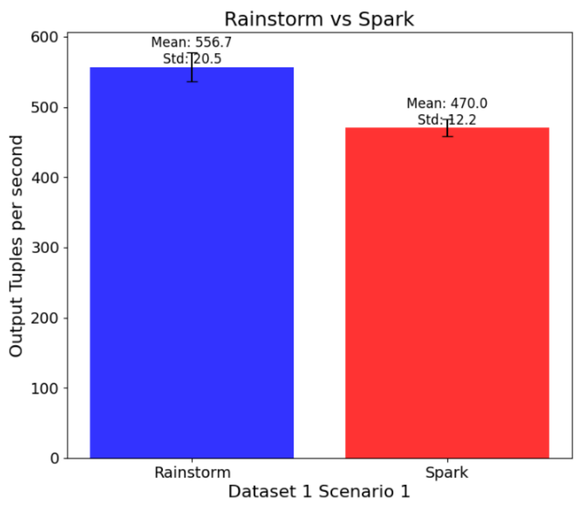
We used the City of Champaign's dataset for Traffic Signs as our first dataset & City of Champaign's Street Lights dataset as our second dataset.

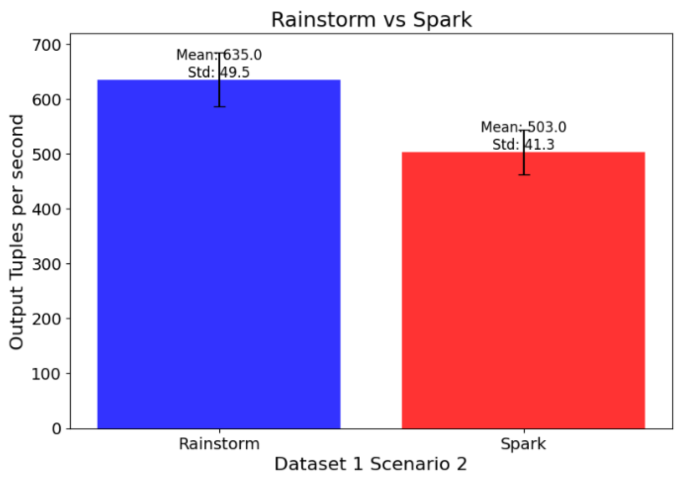
1. Traffic Signs Dataset
   I. Scenario 1 (Simple Operator):

In this scenario, we output the columns "OBJECTID" and "Sign_Type" from rows containing a specific pattern, X. When executed on Spark, this operation took approximately 11–12 seconds to complete. In contrast, when the same test case was run on Rainstorm, it completed in 8–9 seconds. We plotted a graph using the formula Total Input Rows / Time Taken to calculate output tuples per second. For all subsequent scenarios, graphs were plotted using similar calculations.

   II. Scenario 2 (Complex operator):

Here, we first filtered the data to include rows where the "Sign Post" column matches a specific pattern, X. After filtering, we calculated the count of values in the "category" column. In this case, Spark took about 9–11 seconds, whereas Rainstorm completed the task in 6–8 seconds.



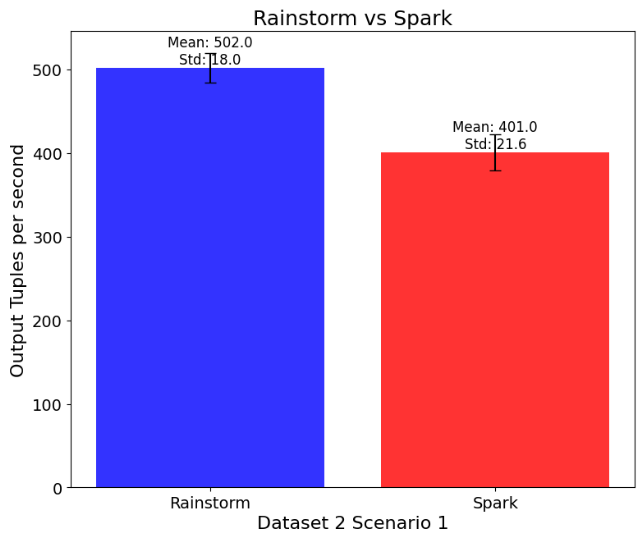


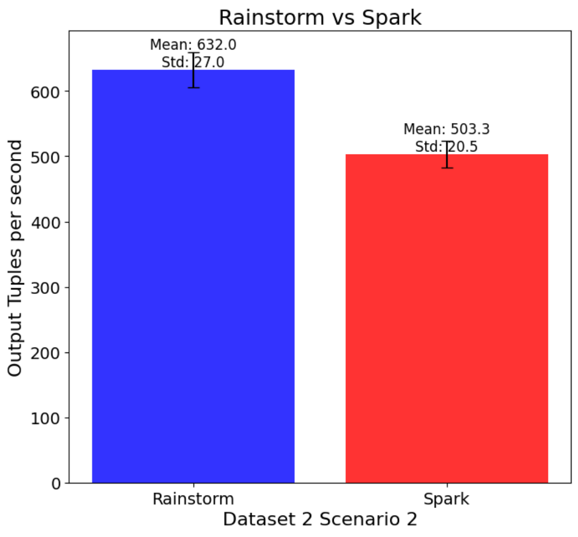
2. Street Lights dataset

   I. Scenario 1 (Simple Operator):

      In this case, we output the columns "OBJECTID" and "Pole_Material" from rows containing a specific pattern, X. This operation took 14–16 seconds on Spark, whereas Rainstorm completed it in 11–13 seconds.

II. Scenario 2 (Complex Operator):
For this scenario, we first filtered the data to include rows where the “Luminaire_Type” column matches a specific pattern, X. After filtering, we calculated the count of values in the “Pole_Material” column. Spark took 12–14 seconds to run this , while Rainstorm completed the task in 9–11 seconds.





From all observations, we found that Rainstorm consistently outperforms Apache Spark by a slight margin. This difference may be due to the Rainstorm requiring fewer initializations and setup steps compared to Spark when executing a given job. We also found that, on the same dataset, our simple operator scenario took more time compared to the complex operator scenario. This could be because the simple operator was writing more lines to the console and to the files compared to the complex operator scenario.