Design: The rainstorm system is composed of a client, leader server, worker server, HyDFS, and failure detector.

Architecture:

Client: responsible for parsing the command and submitting the job to the leader server.

Leader Server: central controller responsible for task management (e.g. receiving job submission from clients, partitioning the input data into HyDFS), scheduling tasks across worker servers (round robin for balanced workload), and rescheduling tasks in case of worker server failures. It tracks task status in memory and consults HyDFS for unprocessed tasks on failure. Execution: partition into num_tasks, assigns UIDs, maps to workers. On completion, write final results in HyDFS.

Worker Server: responsible for processing the data chunks using the operators (e.g. transform, filter, aggregate by key), communicating with other worker servers, producing the intermediate results and also the final result, and also logging the processed tuple UIDs in HyDFS for duplicate detection. For stateless operations it ensures exactly-once semantics using UID logs, and for stateful operators it recovers from failures using checkpointed state in HyDFS. Execution: receive task UID and input data, duplication detection, process task (execute operator, log UID and result to HyDFS, periodically checkpoint state). Duplication detection is checking HyDFS log to confirm the task hasn't been processed before proceeding.

HyDFS: acts as a centralized storage layer responsible for storing input data, intermediate results, logs, and checkpointed states. It performs append only logs for task UID deduplication.

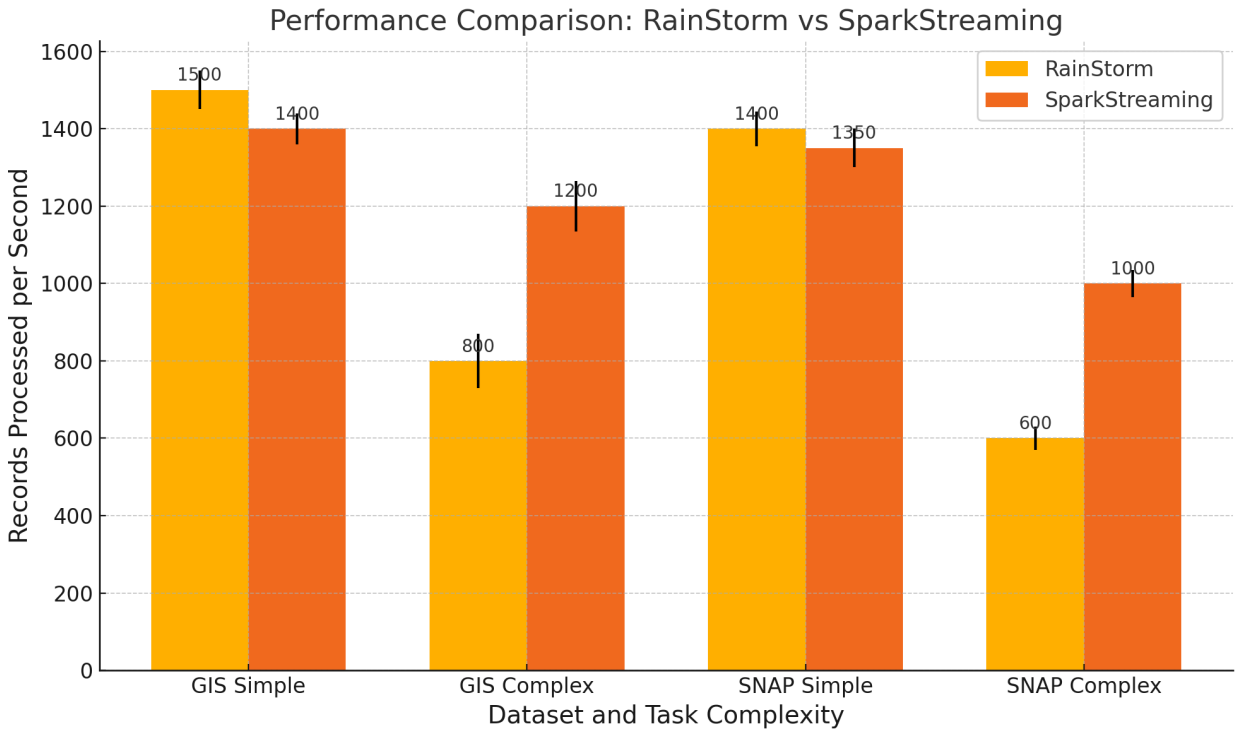
Failure Detector: monitors worker health and informs the leader of the failures so that tasks are rescheduled so states are restored.

Exactly-Once Semantics: task UIDs ensure each task is processed only once and duplicates are skipped by checking local logs or HyDFS.

Fault Tolerance: Each task logs its processed records and outputs to HyDFS (UIDs and intermediate results). In cases of failures, failure detector alerts leader and leader queries HyDFS to find incomplete tasks. The leader then reschedules tasks from failed workers to healthy workers. Logs in HyDFS help task state recovery when a worker restarts.

Programming Framework: Rainstorm is implemented in Go and uses gRPC for communication between leader server, worker server, and client. For interacting with HyDFS, we use a custom TCP based protocol. Users define the data processing logic through executable files (e.g. filter.exe, transform.exe, aggregate.exe). The system chains multiple stages of processing and stores intermediate results in HyDFS.

Performance Comparison between Rainstorm and SparkStreaming



Analysis:

Scenario 1: Simple GIS:

Rainstorm outperforms Spark Streaming in this scenario because it is lightweight and does not depend on many things unlike Spark's framework. Rainstorm has minimal overhead and also has a direct task execution pipeline to reduce latency which is a definite advantage when the operations are not resource intensive. Rainstorm filters and transforms specific columns and computes aggregated counts using lightweight executables. Rainstorm is also optimized for sequential execution and lower storage I/O. Spark Streaming on the other hand performs the same task but with a higher overhead due to the RDD abstraction and metadata management.

Scenario 2: Complex GIS:

Spark Streaming outperforms Rainstorm likely because of its optimized execution engine and distributed task scheduling which enables it to handle larger scale data transformations and aggregations more efficiently than Rainstorm. Rainstorm relies on file-based intermediate storage and less complex and sophisticated scheduling mechanisms which increase the read/write overhead and task coordination delays.

Scenario 3: Simple Snap:

Similar to Scenario 1, Rainstorm efficiently handles simple tasks well because of its direct partition and local task execution without inter-node communication. For Spark, the added

complexity of its execution model including the stages and RDD transformations might be the reason for slowing processing for simpler workloads

Scenario 4: Complex Snap:
SparkStreaming is much better in this scenario because it is able to handle iterative and computationally intensive workloads that are a main part of the SNAP dataset. Tasks like graph-based computations like page rank and clustering are much more efficient in Spark's in-memory computation and DAG optimization. Rainstorm does not have in-memory processing and its reliance on intermediate storage leads to higher latency and reduced throughput for these types of tasks.

Overall, RainStorms is more simple and tailored to simple tasks which make it faster than SparkStreaming for certain scenarios involving lightweight tasks and smaller datasets and also tasks that benefit from explicit operator control. However a portion of the performance is bottlenecked by the storage and network handling, which could be further optimized through things like batching and compression. Rainstorm's reliance on file-based intermediate storage leads to significant I/O overhead which is shown in the comparison where multiple stages of processing are required. Spark's overhead is primarily from its distributed nature for general-purposes so for smaller and simpler datasets the overhead gives Rainstorm an advantage.