

Algorithm and Architecture

RainStorm employs a centralized coordination architecture with a single leader node and multiple worker nodes distributed across VMs:

- **Leader (Stage 0):** The leader serves as the control plane, responsible for:
 - Source process that reads input files from HyDFS and generates the initial stream
 - Task scheduling and distribution across worker VMs using round-robin allocation
 - Failure detection and automatic task recovery (monitors every 250ms)
 - Resource management and autoscaling decisions
 - RPC server (port 9001) for coordination and tuple acknowledgment tracking
- **Workers (Stages 1-N):** Each stage consists of multiple parallel worker tasks that:
 - Execute user-defined operator executables (filter, transform, or aggregation)
 - Process tuples received from the previous stage
 - Forward processed tuples to the next stage via hash partitioning
 - Maintain local state for exactly-once semantics
 - Report flow metrics for autoscaling decisions

Data Flow

Tuples flow through a multi-stage pipeline without barriers between stages:

1. Source: Leader reads from HyDFS, creates tuples with unique IDs (filename:linenumber), and distributes them to Stage 1 workers via hash partitioning
2. Processing: Each worker processes tuples through its operator executable (communicating via stdin/stdout), then forwards results to the next stage
3. Output: Final stage workers send results back to the leader, which writes to both console and HyDFS

Hash Partitioning

Tuples are distributed across workers using the FNV hash of the tuple key modulo the number of tasks in the target stage. This ensures load balancing and maintains key locality. When autoscaling is enabled, partitioning dynamically adapts to the current number of alive workers.

Exactly-Once Semantics

RainStorm guarantees exactly-once processing through:

- Unique Tuple IDs: Each tuple has a globally unique ID (filename:linenumber)
- Acknowledgment Chain: Each stage acknowledges processed tuples back to the sender
- Deduplication: Workers maintain AckedTuple maps to detect and discard duplicate tuples
- Persistent Logging: Each worker logs processed tuple IDs to HyDFS for state recovery after failures
- Retry Logic: Senders retry unacknowledged tuples with a resend queue.

Failure Recovery

When a worker fails:

1. Leader detects missing worker (within 250ms)
2. Leader restarts the task with the same identity on any available VM
3. Restarted worker recovers state by replaying its HyDFS log file
4. The worker rebuilds its AckedTuple map and resumes processing

Autoscaling

When enabled, RainStorm dynamically adjusts the number of tasks per stage:

- Monitoring: Leader tracks per-task flow rates every 250ms
- Scaling Down: If average flow < low watermark, remove one task

- Scaling Up: If average flow > high watermark, add one task
- Dynamic Partitioning: Hash partitioning automatically adapts to the new number of workers

a) **Past MP Use:**

MP1: Logging infrastructure for debugging

MP2: Failure detector for membership management and failure detection

MP3: HyDFS for persistent storage of input files, output files, and recovery logs

b) **LLM use:** In this MP, the LLM is used to help us debug the problem we have and help us with GoLang. It also helps us greatly in setting up environments and writing shell scripts for running the experiments.

c) **Measurements:**

Test1 dataset: From [gplus.tar.gz](#) in Stanford SNAP Repository:

<https://snap.stanford.edu/data/ego-Gplus.html>. This dataset consists of 'circles' from Google+. Google+

data was collected from users who had manually shared their circles using the 'share circle' feature.

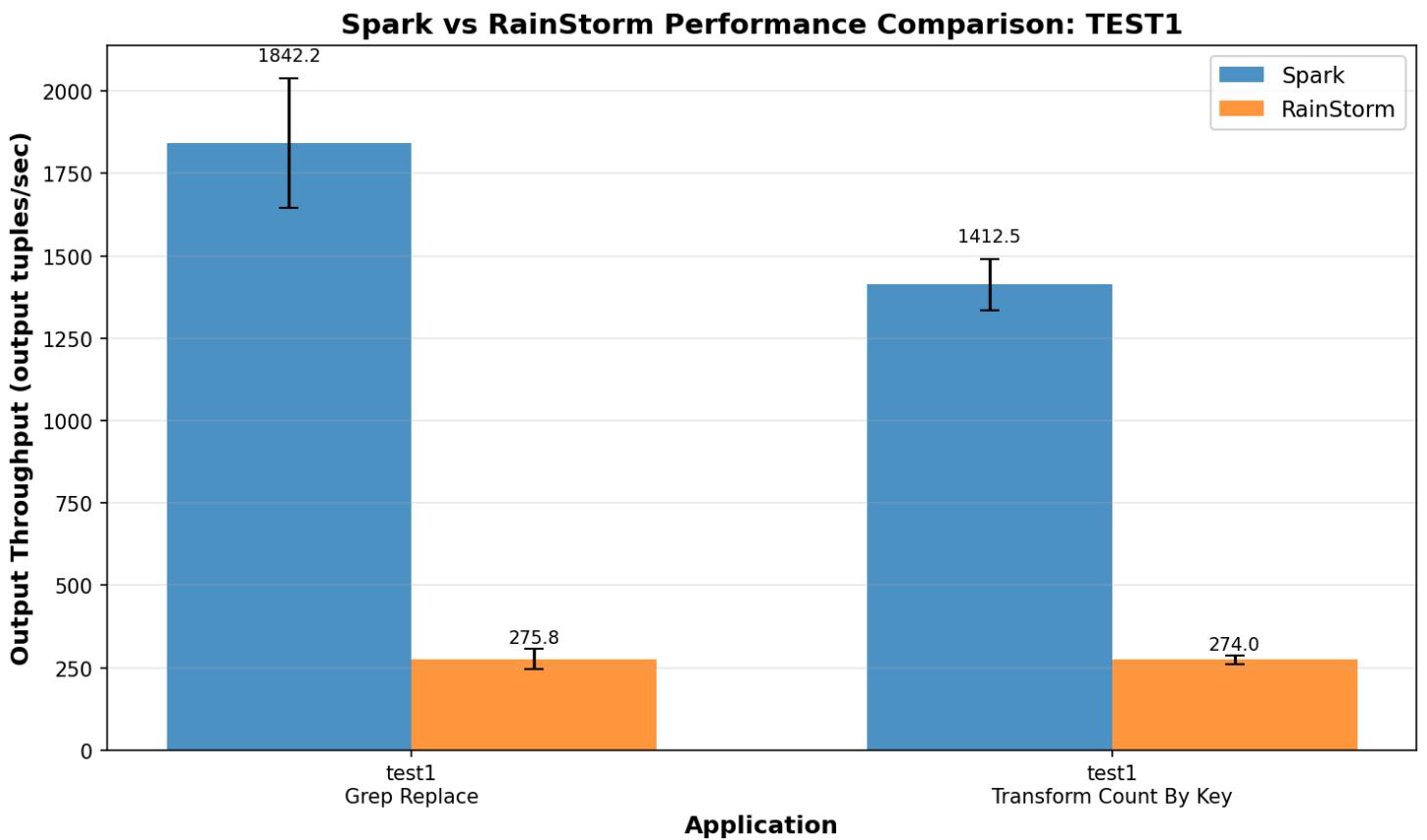
The dataset includes node features (profiles), circles, and ego networks.

Contains two columns, each of which is a number, separated by a space. Each column represents a directed edge in the graph. We only use the first 50k lines of it for faster experiments.

Example row: 103668714482917553215 116594199576805510790

Application 1: grep + replace (grep lines containing “1”, replace “1” with “2”)

Application 2: transform + count by key (transform records to key value pairs, count by key)



Spark outperforms RainStorm for both applications (6.68x faster for grep_replace, 5.16x faster for transform_count_by_key). Spark reads the entire input file from HDFS in one batch operation, processes all 50,000 tuples in memory, and writes output once at the end. RainStorm processes tuples one-by-one through stdin/stdout pipes and logs each processed tuple ID to HyDFS, creating 50,000+

small write operations. These frequent storage accesses (RPC calls, temp file writes, buffered flushes every 1 second) create I/O overhead that dominates processing time. For transform_count_by_key, RainStorm's tuple-by-tuple processing with per-tuple HyDFS logging means logging all 50,000 processed tuples, while Spark's batch aggregation processes everything in memory with minimal storage overhead.

Test2 dataset: Traffic Sign dataset from Champaign Map Database:

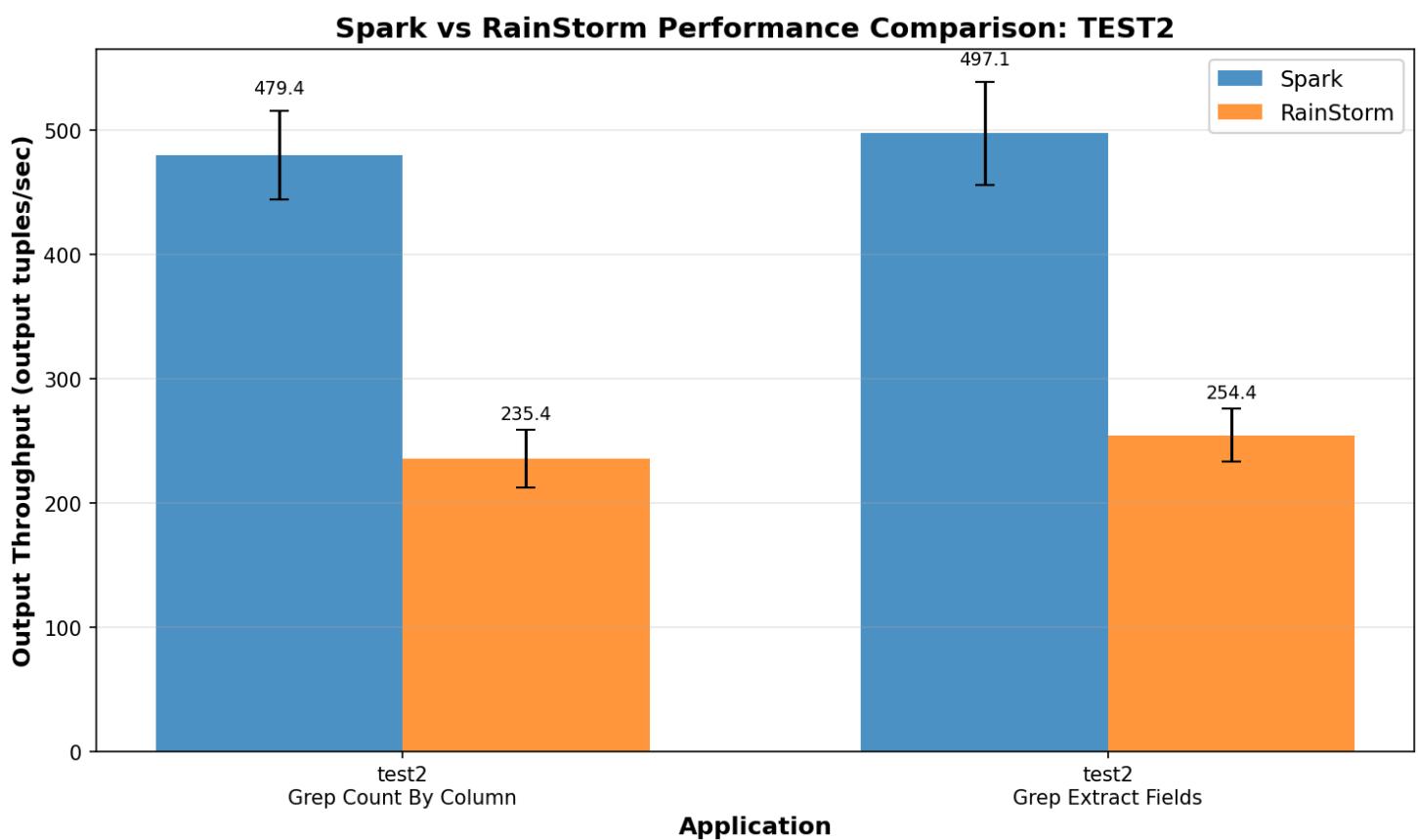
<https://gis-cityofchampaign.opendata.arcgis.com/datasets/cityofchampaign::traffic-signs/about>

Contains 15,547 records

The data set includes specifications about street signs in the City of Champaign. The data set covers signs owned by the city, IDOT, Champaign township, a small selection of private street name signs, and MTD.

Application 1: grep + count by column (grep lines containing “Champaign”, count by column 3)

Application 2: grep + extract (grep lines containing “AERIAL”, extract first three fields)



Spark outperforms RainStorm for both applications (2.04x faster for grep_count_by_column, 1.95x faster for grep_extract_fields). Again, Spark reads the entire 15,570-tuple file in one operation, processes in memory, and writes output once; RainStorm processes each tuple individually, creating thousands of small write operations. The storage access overhead from frequent HyDFS logging significantly slows down RainStorm compared to Spark's efficient bulk I/O operations. Even for simple transformations like field extraction, RainStorm's requirement to log every processed tuple for exactly-once semantics creates massive storage I/O overhead that Spark avoids through batch processing.