MSc. Data Science

Coventry University, UK

Coursework

BIG DATA ANALYTICS TECHNOLOGIES

—

M.D.P. Wijesuriya

Student ID (Coventry Uni.): 15764609

Student ID (NIBM): comscds241p-002

2024 Batch

—

Contents

Task 1: In-Degree Distribution Analysis using Apache Hadoop and Apache Spark 4

Part 1: Implementation and Performance Comparison 4

Part 2: Scalability and Optimization Analysis 15

Task 2: Real-Time IoT Sensor Data Analytics using Apache Kafka 19

Environment Setup 19

System Requirements 19

Python Dependencies Installation 19

Infrastructure Setup 19

Kafka Topic Design and Configuration 20

Topic Architecture 20

Topic Creation.....20

Producer and Consumer Logic 21

Producer Implementation.....21

Consumer Implementation.....21

Metric Computation Methodology 22

Hourly Average Vehicle Count per Sensor 22

Daily Peak and Daily Average Traffic Volume Across All Sensors.....23

Daily Sensor Availability Percentage 23

Grafana Dashboard Setup and Interpretation 24

Dashboard Architecture 24

Data Source Configuration.....24

Dashboard Visualization Configuration.....25

Dashboard Refresh and Real-Time Updates 26

Performance Observation 26

Task 3: Graph Databases using Neo4j and Docker 28

Part 1 - Understanding Graph Databases.....28

Part 2 - Implementation with Neo4j (Docker Setup) 31

Task 4: Watermarks in Real-Time Stream Processing using Apache Kafka and Apache Flink (Docker Setup) 38
Part 1 - Understanding Watermarks 38
Part 2 - Implementation using Docker (Kafka + Flink) 40
Task 5: Self Study – Big Data Technologies in Large Language Model Development 50
Abstract 50
5.1 Development 50
5.2 Verification 51
5.3 Deployment 51
References 52

Task 1: In-Degree Distribution Analysis using Apache Hadoop and Apache Spark

Part 1: Implementation and Performance Comparison

1. Implement in-degree distribution computation.

Overview of the Implementation

Apache Hadoop (MapReduce)
Setup & Execution:

- Implemented Java MapReduce program with Mapper and Reducer classes.
- Java code is compiled and packaged into a JAR file.
- Created a shell script (run_mapreduce.sh and run_hadoop_optimized.sh) to automate:
    - HDFS cleanup - remove old input/output directories
    - Upload dataset to HDFS input directory
    - Submit MapReduce job via hadoop jar command
    - Download results from HDFS to local filesystem
- Monitored the execution through YARN ResourceManager UI (localhost:8088)
- Captured performance metrics: execution time, memory, shuffle overhead

Apache Spark
Setup & Execution:

- Implemented Python script using PySpark RDD transformations
- Created a shell script (run_spark.sh and run_spark_partitioning_optimized.sh) for automation of
    - HDFS cleanup and dataset upload
    - Submit Spark job via spark-submit command
    - Download results from HDFS
- Monitored execution through Spark UI on http://localhost:4040
- Performance metrics were logged, including memory usage and stage timings.

Common Infrastructure:

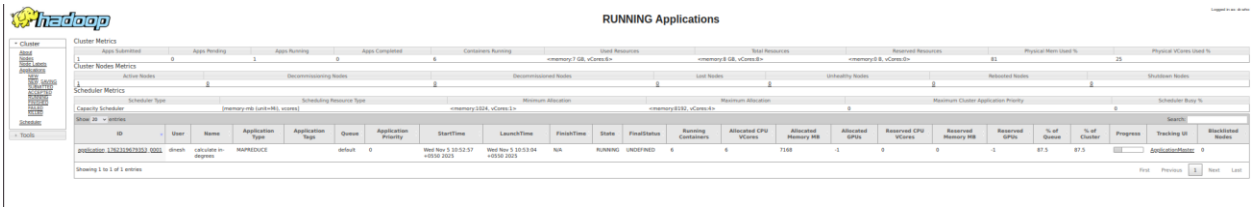
- Both systems run in Ubuntu VM environment
- Datasets stored and accessed via HDFS
- Complete workflow, from data upload to result retrieval, is handled through automated scripts
- Comprehensive logging captures execution and performance details
- Generated visualizations comparing performance across datasets

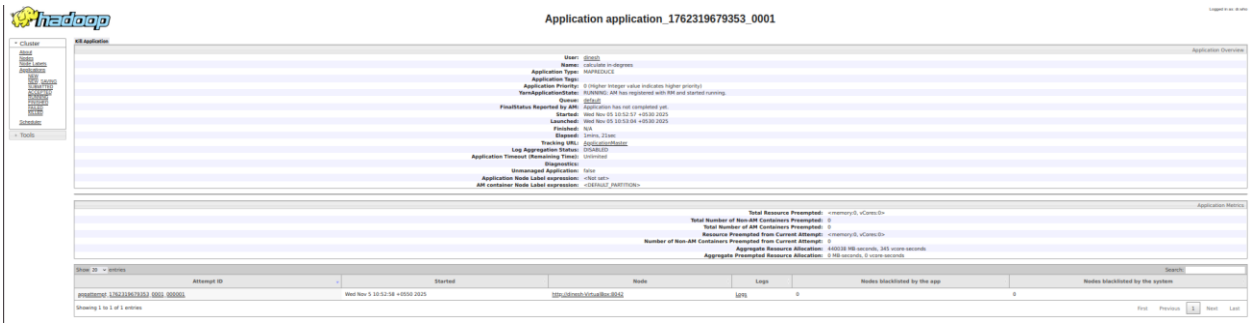
2. Run Experiments.

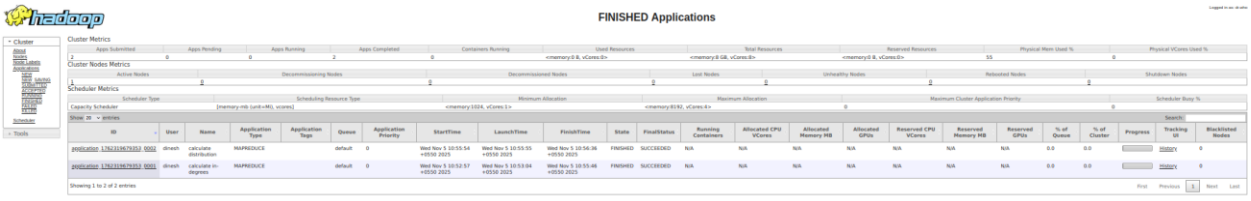
YARN ResourceManager UI (http://localhost:8088)

Cluster Info

Running/Finished applications







Job History Server UI (http://localhost:19888)

HDFS NameNode UI (http://localhost:9870)

Hadoop | Overview | Datanodes | Datanode Volume Failures | Snapshot | Startup Progress | Utilities

Overview 'localhost:9000' (active)

Started:	Mon Nov 03 13:40:36 +0530 2025
Version:	3.3.6, r1be78238728da9266a4f88195058f08fd012bf9c
Compiled:	Sun Jun 18 13:52:00 +0530 2023 by ubuntu from (HEAD detached at release-3.3.6-RC1)
Cluster ID:	CID-aca7cd51-14e7-496f-8e95-b6f2af58dfa4
Block Pool ID:	BP-341744916-127.0.1.1-1761126309750

Summary

Security is off.

Safemode is off.

19 files and directories, 9 blocks (9 replicated blocks, 0 erasure coded block groups) = 28 total filesystem object(s).

Heap Memory used 54.91 MB of 120 MB Heap Memory. Max Heap Memory is 980 MB.

Non Heap Memory used 55.88 MB of 58.75 MB Commited Non Heap Memory. Max Non Heap Memory is <unbounded>.

Configured Capacity:	37.78 GB
Configured Remote Capacity:	0 B
DFS Used:	1.04 MB (0%)
Non DFS Used:	22.72 GB
DFS Remaining:	13.26 GB (35.09%)
Block Pool Used:	1.04 MB (0%)
DataNodes usages% (Min/Median/Max/stdDev):	0.00% / 0.00% / 0.00% / 0.00%
Live Nodes	1 (Decommissioned: 0, In Maintenance: 0)
Dead Nodes	0 (Decommissioned: 0, In Maintenance: 0)
Decommissioning Nodes	0
Entering Maintenance Nodes	0
Total Datanode Volume Failures	0 (0 B)
Number of Under-Replicated Blocks	0
Number of Blocks Pending Deletion (including replicas)	0
Block Deletion Start Time	Mon Nov 03 13:40:36 +0530 2025
Last Checkpoint Time	Mon Nov 03 13:40:37 +0530 2025
Enabled Erasure Coding Policies	RS-6-3-1024k

NameNode Journal Status

Current transaction ID: 211

Journal Manager	State
FileJournalManager(root=/opt/hadoop_data/namenode)	EditLogFileOutputStream(/opt/hadoop_data/namenode/current/edits_inprogress_0000000000000000211)

NameNode Storage

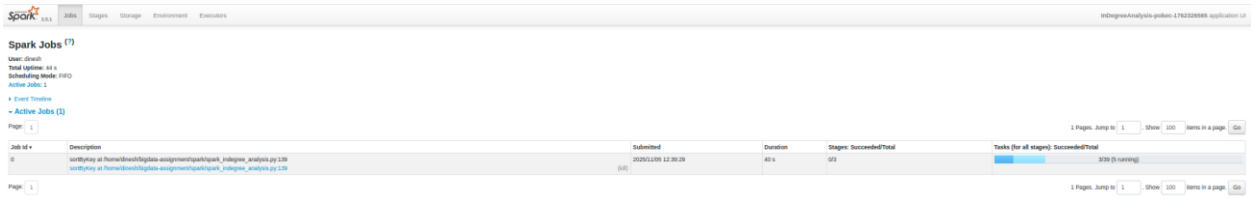
Storage Directory	Type	State
/opt/hadoop_data/namenode	IMAGE_AND_EDITS	Active

DFS Storage Types

Storage Type	Configured Capacity	Capacity Used	Capacity Remaining	Block Pool Used	Nodes In Service
DISK	37.78 GB	1.04 MB (0%)	13.26 GB (35.09%)	1.04 MB	1

Hadoop, 2023.

Spark Jobs (http://localhost:4040/)



Spark Jobs (?)

User: dinesh
Total Uptime: 44 s
Scheduling Mode: FIFO
Active Jobs: 1

▸ Event Timeline

− Active Jobs (1)

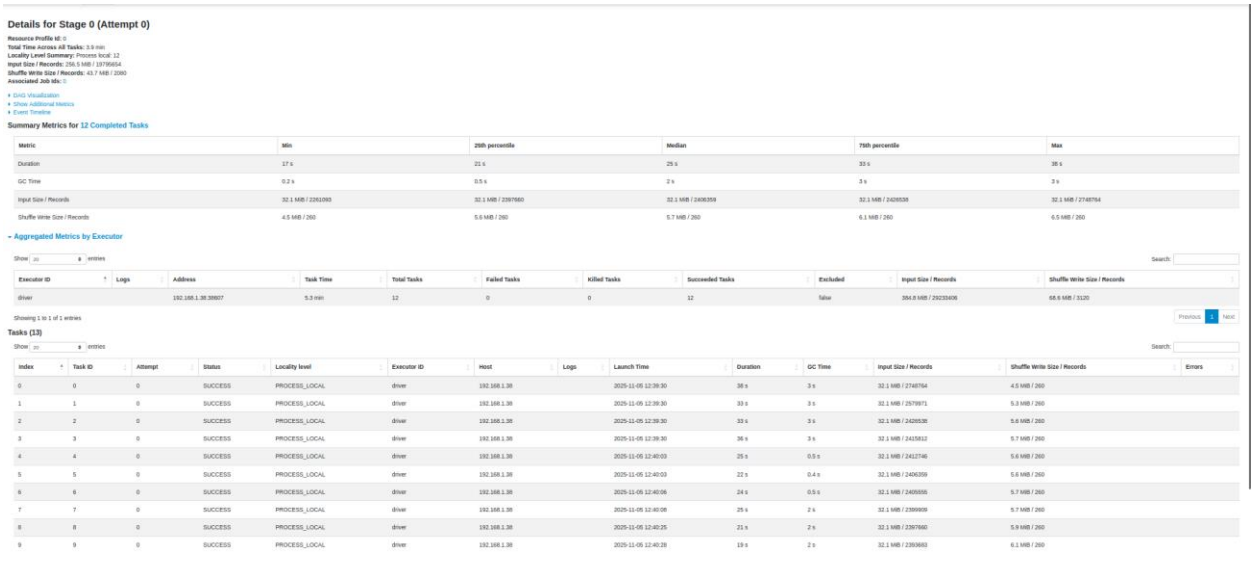
Page: 1

1 Pages. Jump to 1 . Show 100 items in a page. Go

Job Id ▾	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
0	sortByKey at /home/dinesh/bigdata-assignment/spark/spark_indegree_analysis.py:139 sortByKey at /home/dinesh/bigdata-assignment/spark/spark_indegree_analysis.py:139	2025/11/05 12:39:29 (kill)	40 s	0/3	3/39 (5 running)

Page: 1

1 Pages. Jump to 1 . Show 100 items in a page. Go



Details for Stage 0 (Attempt 0)

Resource Profile Id: 0
Total Time Across All Tasks: 3.9 min
Locality Level Summary: Process local: 12
Input Size / Records: 259.5 MiB / 19795654
Shuffle Write Size / Records: 43.7 MiB / 2080
Associated Job Ids: 0

▸ DAG Visualization
▸ Show Additional Metrics
▸ Event Timeline

Summary Metrics for 12 Completed Tasks

Metric	Min	25th percentile	Median	75th percentile	Max
Duration	17 s	21 s	25 s	33 s	35 s
GC Time	0.2 s	0.5 s	2 s	3 s	3 s
Input Size / Records	32.1 MiB / 2261093	32.1 MiB / 2397660	32.1 MiB / 2406359	32.1 MiB / 2426538	32.1 MiB / 2748764
Shuffle Write Size / Records	4.5 MiB / 260	5.6 MiB / 260	5.7 MiB / 260	6.1 MiB / 260	6.5 MiB / 260

− Aggregated Metrics by Executor

Show 20 entries Search:

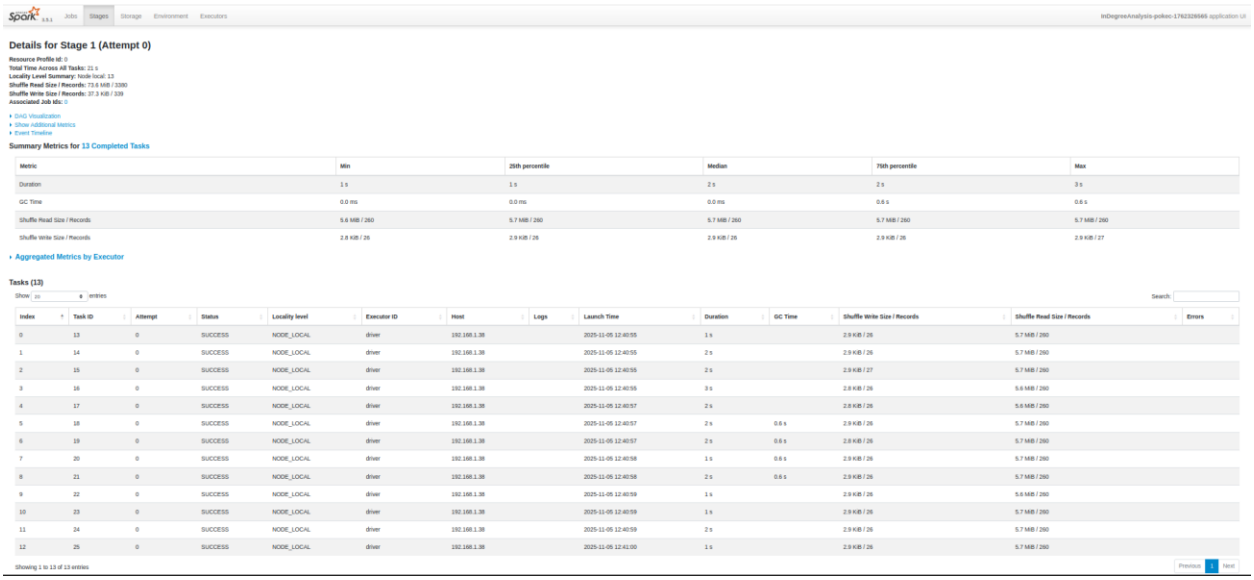
Executor ID	Logs	Address	Task Time	Total Tasks	Failed Tasks	Killed Tasks	Succeeded Tasks	Excluded	Input Size / Records	Shuffle Write Size / Records
driver		192.168.1.38:38607	5.3 min	12	0	0	12	false	384.8 MiB / 29233406	68.6 MiB / 3120

Showing 1 to 1 of 1 entries Previous 1 Next

Tasks (13)

Show 20 entries Search:

Index ▲	Task ID	Attempt	Status	Locality level	Executor ID	Host	Logs	Launch Time	Duration	GC Time	Input Size / Records	Shuffle Write Size / Records	Errors
0	0	0	SUCCESS	PROCESS_LOCAL	driver	192.168.1.38		2025-11-05 12:39:30	38 s	3 s	32.1 MiB / 2748764	4.5 MiB / 260	
1	1	0	SUCCESS	PROCESS_LOCAL	driver	192.168.1.38		2025-11-05 12:39:30	33 s	3 s	32.1 MiB / 2579871	5.3 MiB / 260	
2	2	0	SUCCESS	PROCESS_LOCAL	driver	192.168.1.38		2025-11-05 12:39:30	33 s	3 s	32.1 MiB / 2426538	5.6 MiB / 260	
3	3	0	SUCCESS	PROCESS_LOCAL	driver	192.168.1.38		2025-11-05 12:39:30	36 s	3 s	32.1 MiB / 2415812	5.7 MiB / 260	
4	4	0	SUCCESS	PROCESS_LOCAL	driver	192.168.1.38		2025-11-05 12:40:03	25 s	0.5 s	32.1 MiB / 2412746	5.6 MiB / 260	
5	5	0	SUCCESS	PROCESS_LOCAL	driver	192.168.1.38		2025-11-05 12:40:03	22 s	0.4 s	32.1 MiB / 2406359	5.6 MiB / 260	
6	6	0	SUCCESS	PROCESS_LOCAL	driver	192.168.1.38		2025-11-05 12:40:06	24 s	0.5 s	32.1 MiB / 2405555	5.7 MiB / 260	
7	7	0	SUCCESS	PROCESS_LOCAL	driver	192.168.1.38		2025-11-05 12:40:08	25 s	2 s	32.1 MiB / 2399809	5.7 MiB / 260	
8	8	0	SUCCESS	PROCESS_LOCAL	driver	192.168.1.38		2025-11-05 12:40:25	21 s	2 s	32.1 MiB / 2397660	5.9 MiB / 260	
9	9	0	SUCCESS	PROCESS_LOCAL	driver	192.168.1.38		2025-11-05 12:40:28	19 s	2 s	32.1 MiB / 2393693	6.1 MiB / 260	



Details for Stage 1 (Attempt 0)

Resource Profile Id: 0
Total Time Across All Tasks: 21 s
Locality Level Summary: Node local: 13
Shuffle Read Size / Records: 73.6 MiB / 3380
Shuffle Write Size / Records: 37.3 KiB / 339
Associated Job Ids: 0

▸ DAG Visualization
▸ Show Additional Metrics
▸ Event Timeline

Summary Metrics for 13 Completed Tasks

Metric	Min	25th percentile	Median	75th percentile	Max
Duration	1 s	1 s	2 s	2 s	3 s
GC Time	0.0 ms	0.0 ms	0.0 ms	0.6 s	0.6 s
Shuffle Read Size / Records	5.6 MiB / 260	5.7 MiB / 260	5.7 MiB / 260	5.7 MiB / 260	5.7 MiB / 260
Shuffle Write Size / Records	2.8 KiB / 26	2.9 KiB / 26	2.9 KiB / 26	2.9 KiB / 26	2.9 KiB / 27

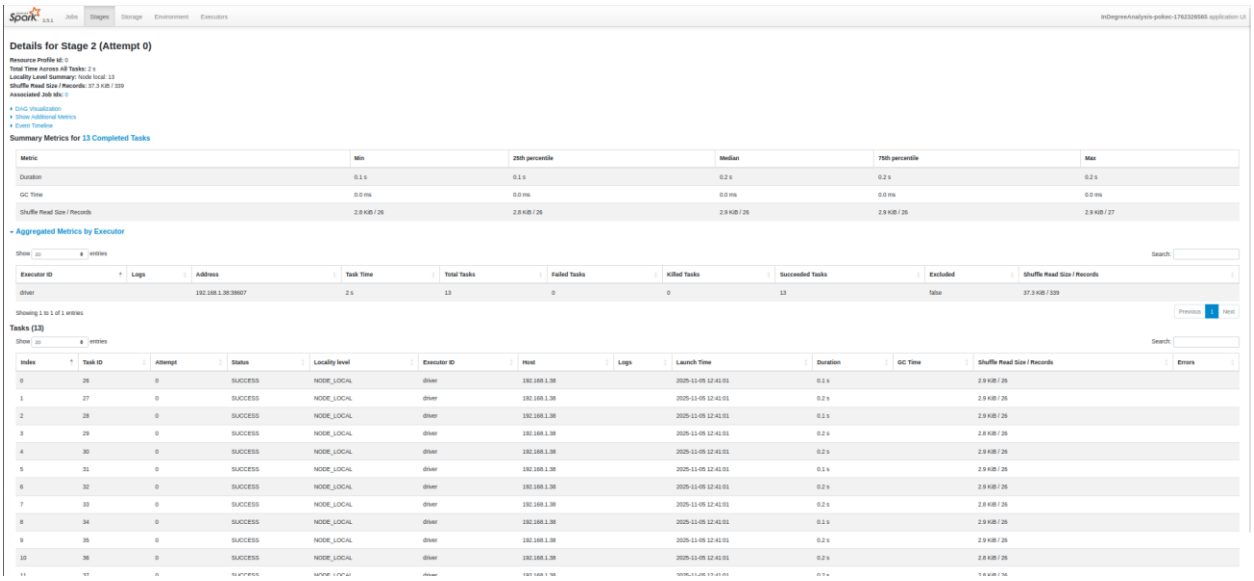
▸ Aggregated Metrics by Executor

Tasks (13)

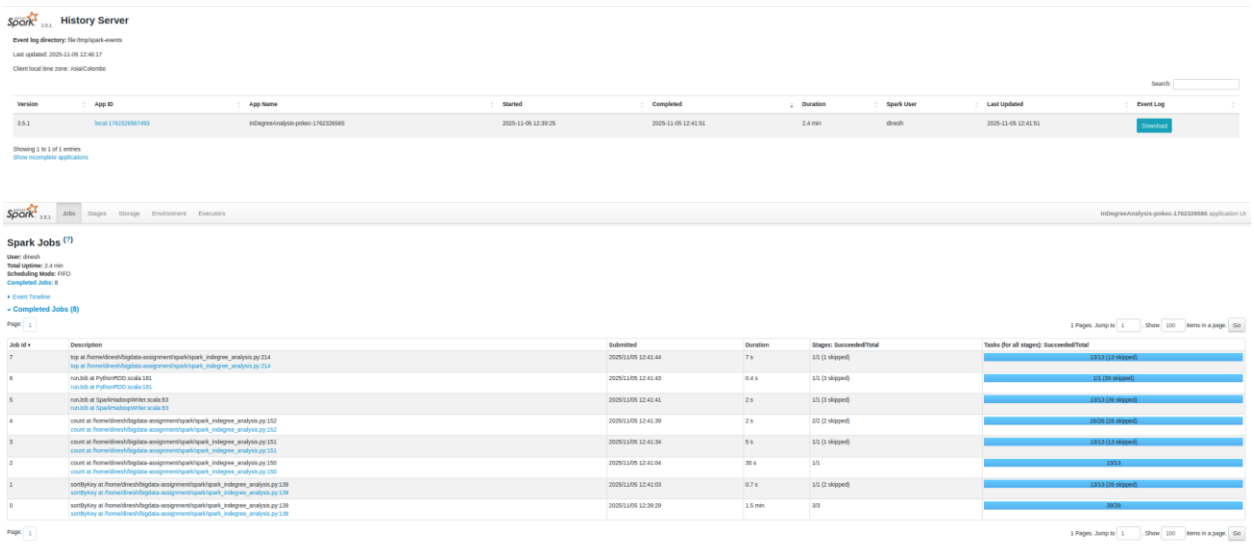
Show 20 entries Search:

Index ▲	Task ID	Attempt	Status	Locality level	Executor ID	Host	Logs	Launch Time	Duration	GC Time	Shuffle Write Size / Records	Shuffle Read Size / Records	Errors
0	13	0	SUCCESS	NODE_LOCAL	driver	192.168.1.38		2025-11-05 12:40:55	1 s		2.9 KiB / 26	5.7 MiB / 260	
1	14	0	SUCCESS	NODE_LOCAL	driver	192.168.1.38		2025-11-05 12:40:55	2 s		2.9 KiB / 26	5.7 MiB / 260	
2	15	0	SUCCESS	NODE_LOCAL	driver	192.168.1.38		2025-11-05 12:40:55	2 s		2.9 KiB / 27	5.7 MiB / 260	
3	16	0	SUCCESS	NODE_LOCAL	driver	192.168.1.38		2025-11-05 12:40:55	3 s		2.8 KiB / 26	5.6 MiB / 260	
4	17	0	SUCCESS	NODE_LOCAL	driver	192.168.1.38		2025-11-05 12:40:57	2 s		2.8 KiB / 26	5.6 MiB / 260	
5	18	0	SUCCESS	NODE_LOCAL	driver	192.168.1.38		2025-11-05 12:40:57	2 s	0.6 s	2.9 KiB / 26	5.7 MiB / 260	
6	19	0	SUCCESS	NODE_LOCAL	driver	192.168.1.38		2025-11-05 12:40:57	2 s	0.6 s	2.8 KiB / 26	5.7 MiB / 260	
7	20	0	SUCCESS	NODE_LOCAL	driver	192.168.1.38		2025-11-05 12:40:58	1 s	0.6 s	2.9 KiB / 26	5.7 MiB / 260	
8	21	0	SUCCESS	NODE_LOCAL	driver	192.168.1.38		2025-11-05 12:40:58	2 s	0.6 s	2.9 KiB / 26	5.7 MiB / 260	
9	22	0	SUCCESS	NODE_LOCAL	driver	192.168.1.38		2025-11-05 12:40:59	1 s		2.9 KiB / 26	5.6 MiB / 260	
10	23	0	SUCCESS	NODE_LOCAL	driver	192.168.1.38		2025-11-05 12:40:59	1 s		2.9 KiB / 26	5.7 MiB / 260	
11	24	0	SUCCESS	NODE_LOCAL	driver	192.168.1.38		2025-11-05 12:40:59	2 s		2.9 KiB / 26	5.7 MiB / 260	
12	25	0	SUCCESS	NODE_LOCAL	driver	192.168.1.38		2025-11-05 12:41:00	1 s		2.9 KiB / 26	5.7 MiB / 260	

Showing 1 to 13 of 13 entries Previous 1 Next



Spark History Server (http://localhost:18080/)



3. Record, Analysis and Comparison

Performance Metrics Comparison for soc-Pokec dataset

Execution Time Analysis

Metric	Hadoop MapReduce	Apache Spark	Spark Advantage
Total Execution Time	232 seconds (3m 52s)	135 seconds (2m 15s)	1.72x faster
In-degree Calculation	182.5 seconds (Job 1)	0.19 seconds	961x faster

Memory Usage Comparison

Memory Metric	Hadoop MapReduce	Apache Spark	Efficiency
Peak Map Memory	547 MB	N/A (local mode)	Spark more efficient
Peak Reduce Memory	201 MB	N/A (local mode)	Spark more efficient
Driver Memory	N/A	22.56 MB (RSS)	Very lightweight
Total Heap Usage	1.57 GB	391.21 MB (VMS)	Spark uses 75% less
System Memory %	Multiple processes	0.6% of 3.82 GB	Spark minimal footprint

CPU and I/O Performance

Resource Metric	Hadoop MapReduce	Apache Spark	Comparison
CPU Utilization	Distributed across tasks	69.4% peak usage	Spark concentrated
Total CPU Time	191,210 ms	~94,000 ms (calculated)	Spark ~50% less
HDFS Bytes Read	423.95 MB	404.30 MB (local)	Similar input processing
HDFS Bytes Written	14.90 MB	Local filesystem	Different storage approach
Shuffle Data	54.14 MB	Minimal (local mode)	Spark reduces network I/O

Comprehensive System Comparison

Correctness of Results

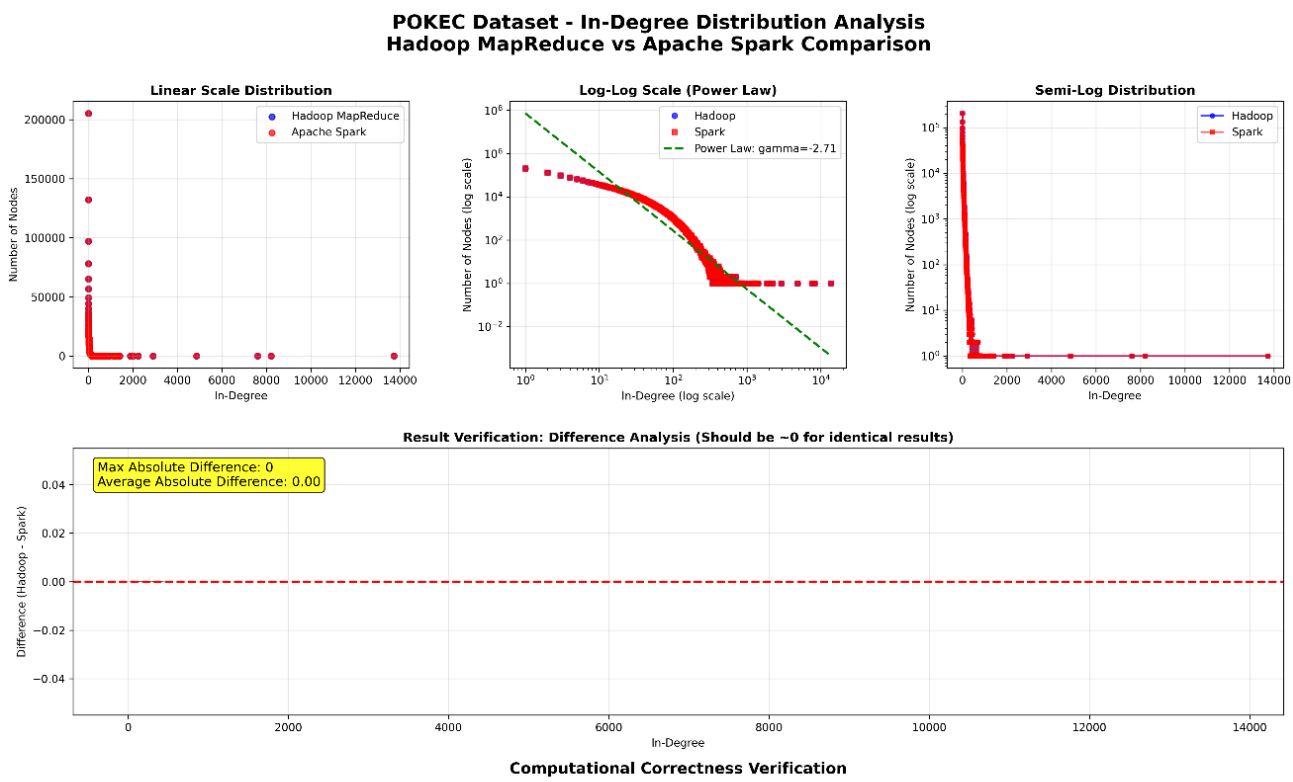
Verification Metric	Hadoop MapReduce	Apache Spark	Match Status
Total Nodes	1,519,452	1,519,452	Perfect Match
Total Edges	30,622,564	30,622,564	Perfect Match
Max In-Degree	13,733	13,733	Perfect Match
Average In-Degree	20.15	20.15	Perfect Match
Distribution Entries	535	535	Perfect Match
Absolute Difference	N/A	0.00	Identical Results

Execution Performance Summary

Performance Aspect	Hadoop MapReduce	Apache Spark	Winner
Overall Speed	232 seconds	135 seconds	Spark (1.72x)
Memory Efficiency	1.57 GB heap	391 MB total	Spark (75% less)
CPU Efficiency	Distributed load	Concentrated 69.4%	Spark (better utilization)
I/O Overhead	HDFS read/write + shuffle	Local filesystem	Spark (reduced I/O)
Scalability	Linear with cluster size	Exponential with memory	Context dependent

System Design and Data Processing Approach

Design Aspect	Hadoop MapReduce	Apache Spark
Processing Model	Disk-based batch processing	In-memory iterative processing
Job Structure	2 sequential MapReduce jobs	Single application, multiple stages
Data Storage	HDFS intermediate storage	RDD transformations in memory
Fault Tolerance	Task-level restart	RDD lineage reconstruction
Optimization	Static optimization	Dynamic optimization + lazy evaluation



Metric	Hadoop MapReduce	Apache Spark	Match
Total Nodes	1,519,452	1,519,452	YES
Total Edges	30,622,564	30,622,564	YES
Average In-Degree	20.15	20.15	YES
Max In-Degree	13,733	13,733	YES
Distribution Entries	535	535	YES

Performance Metrics Comparison for email-EuAll dataset

Execution Time Analysis

Metric	Hadoop MapReduce	Apache Spark	Spark Advantage
Total Execution Time	55 seconds	4.04 seconds	13.6x faster
In-degree Calculation	29.3 seconds (Job 1)	0.23 seconds	127x faster

Memory Usage Comparison

Memory Metric	Hadoop MapReduce	Apache Spark	Efficiency
Peak Map Memory	334 MB	N/A (local mode)	Spark more efficient
Peak Reduce Memory	209 MB	N/A (local mode)	Spark more efficient
Driver Memory	N/A	46.70 MB (RSS)	Very lightweight
Total Heap Usage	394 MB	391.21 MB (VMS)	Similar footprint
System Memory %	Multiple processes	1.2% of 3.82 GB	Spark minimal impact

CPU and I/O Performance

Resource Metric	Hadoop MapReduce	Apache Spark	Comparison
CPU Utilization	Distributed across tasks	34.7% peak usage	Spark efficient
Total CPU Time	10,980 ms	~1,400 ms (calculated)	Spark ~87% less
HDFS Bytes Read	5.00 MB	4.77 MB (local)	Similar input processing
HDFS Bytes Written	634 KB	Local filesystem	Different storage approach
Shuffle Data	930 KB	Minimal (local mode)	Spark reduces network I/O

Comprehensive System Comparison

Correctness of Results

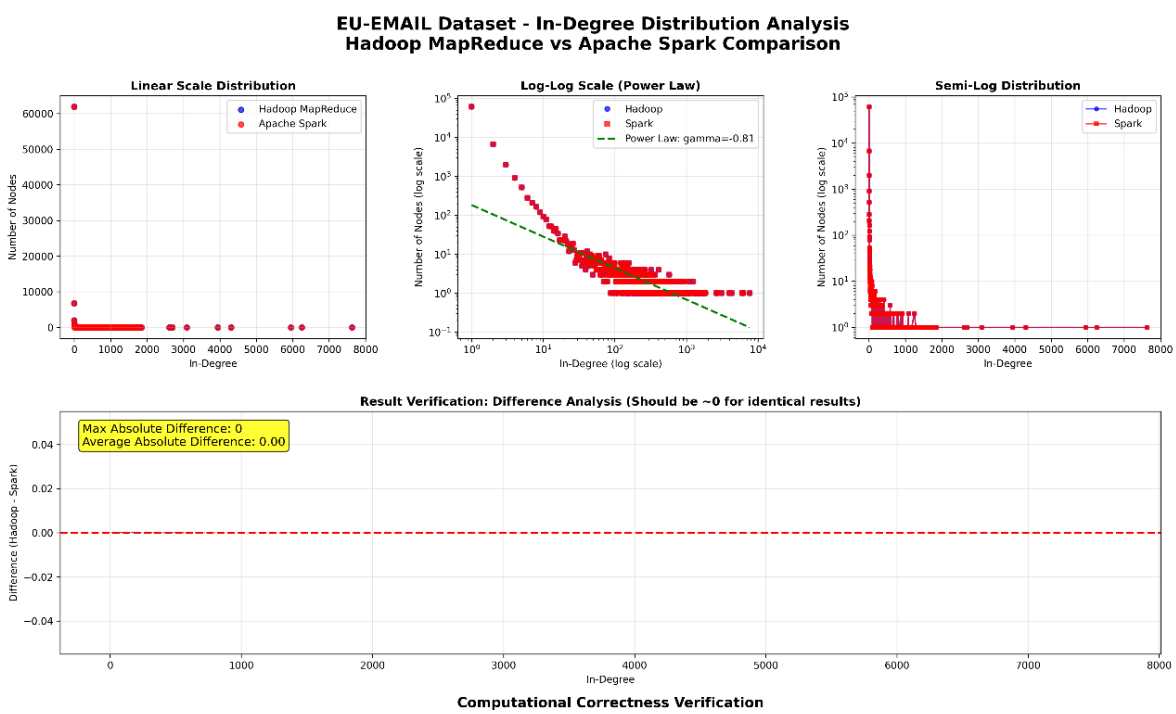
Verification Metric	Hadoop MapReduce	Apache Spark	Match Status
Total Nodes	74,660	74,660	Perfect Match
Total Edges	420,045	420,045	Perfect Match
Max In-Degree	7,631	7,631	Perfect Match
Average In-Degree	5.63	5.63	Perfect Match
Distribution Entries	518	518	Perfect Match
Absolute Difference	N/A	0.00	Identical Results

Execution Performance Summary

Performance Aspect	Hadoop MapReduce	Apache Spark	Winner
Overall Speed	55 seconds	4.04 seconds	Spark (13.6x)
Memory Efficiency	394 MB heap	391 MB total	Similar
CPU Efficiency	10,980 ms total	~1,400 ms total	Spark (87% less)
I/O Overhead	HDFS read/write + shuffle	Local filesystem	Spark (reduced I/O)
Setup Overhead	YARN coordination	Direct execution	Spark (minimal overhead)

System Design and Data Processing Approach

Design Aspect	Hadoop MapReduce	Apache Spark
Processing Model	Disk-based batch processing	In-memory iterative processing
Job Structure	2 sequential MapReduce jobs	Single application, 5 stages
Startup Overhead	YARN container allocation	JVM initialization only
Data Storage	HDFS intermediate storage	RDD transformations in memory
Network I/O	930 KB shuffle data	Minimal local operations



Metric	Hadoop MapReduce	Apache Spark	Match
Total Nodes	74,660	74,660	YES
Total Edges	420,045	420,045	YES
Average In-Degree	5.63	5.63	YES
Max In-Degree	7,631	7,631	YES
Distribution Entries	518	518	YES

Performance Metrics Comparison for web-BerkStan dataset

Execution Time Analysis

Metric	Hadoop MapReduce	Apache Spark	Spark Advantage
Total Execution Time	71 seconds	13.85 seconds	5.1x faster
In-degree Calculation	45.5 seconds (Job 1)	0.15 seconds	303x faster

Memory Usage Comparison

Memory Metric	Hadoop MapReduce	Apache Spark	Efficiency
Peak Map Memory	438 MB	N/A (local mode)	Spark more efficient
Peak Reduce Memory	215 MB	N/A (local mode)	Spark more efficient
Driver Memory	N/A	46.82 MB (RSS)	Very lightweight
Total Heap Usage	435 MB	391.66 MB (VMS)	Spark uses 10% less
System Memory %	Multiple processes	1.2% of 3.82 GB	Spark minimal footprint

CPU and I/O Performance

Resource Metric	Hadoop MapReduce	Apache Spark	Comparison
CPU Utilization	Distributed across tasks	32.3% peak usage	Spark efficient
Total CPU Time	30,040 ms	~4,500 ms (calculated)	Spark ~85% less
HDFS Bytes Read	110.14 MB	105.03 MB (local)	Similar input processing
HDFS Bytes Written	5.54 MB	Local filesystem	Different storage approach
Shuffle Data	7.91 MB	Minimal (local mode)	Spark reduces network I/O

Comprehensive System Comparison

Correctness of Results

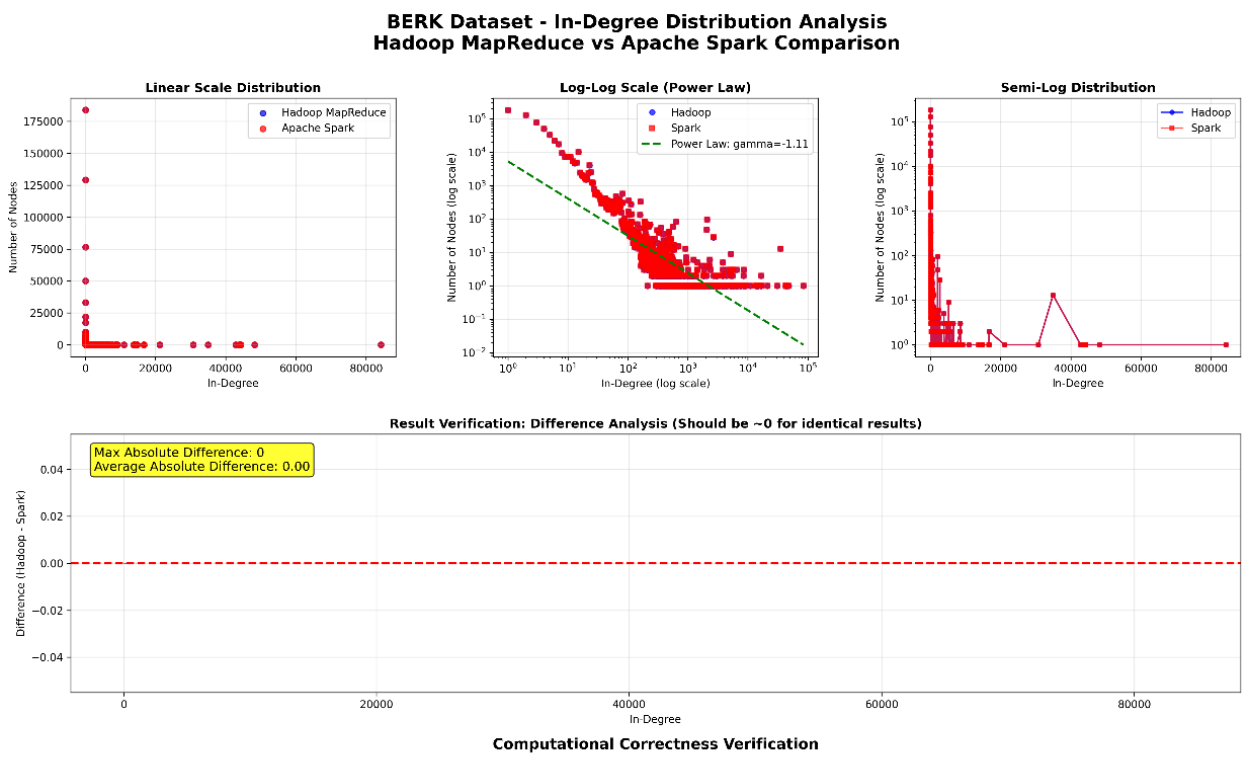
Verification Metric	Hadoop MapReduce	Apache Spark	Match Status
Total Nodes	617,094	617,094	Perfect Match
Total Edges	7,600,595	7,600,595	Perfect Match
Max In-Degree	84,208	84,208	Perfect Match
Average In-Degree	12.32	12.32	Perfect Match
Distribution Entries	925	925	Perfect Match
Absolute Difference	N/A	0.00	Identical Results

Execution Performance Summary

Performance Aspect	Hadoop MapReduce	Apache Spark	Winner
Overall Speed	71 seconds	13.85 seconds	Spark (5.1x)
Memory Efficiency	435 MB heap	392 MB total	Spark (10% less)
CPU Efficiency	30,040 ms total	~4,500 ms total	Spark (85% less)
I/O Overhead	HDFS + 7.91 MB shuffle	Local filesystem	Spark (reduced I/O)
Processing Rate	107,051 edges/sec	548,585 edges/sec	Spark (5.1x faster)

System Design and Data Processing Approach

Design Aspect	Hadoop MapReduce	Apache Spark
Processing Model	Disk-based batch processing	In-memory iterative processing
Job Structure	2 sequential MapReduce jobs	Single application, 5 stages
Startup Overhead	YARN + container allocation	JVM initialization only
Intermediate Storage	5.54 MB to HDFS	RDD transformations in memory
Network Communication	7.91 MB shuffle overhead	Local operations only



Metric	Hadoop MapReduce	Apache Spark	Match
Total Nodes	617,094	617,094	YES
Total Edges	7,600,595	7,600,595	YES
Average In-Degree	12.32	12.32	YES
Max In-Degree	84,208	84,208	YES
Distribution Entries	925	925	YES

Part 2: Scalability and Optimization Analysis

Performance Metrics vs. Dataset Size

Scalability Analysis: LiveJournal Dataset (1.1GB)

Dataset Characteristics:

- Size: 1,080,598,042 bytes (1.1GB)
- Edges: 68,993,773
- Unique Nodes: 4,489,240
- Distribution Entries: 1,569

Identified Bottlenecks

1. Network Shuffle Overhead (Primary Bottleneck - Hadoop)

   Hadoop MapReduce:
   - Baseline shuffle: 207 MB transferred between map and reduce phases
   - Impact: Job 1 took 338s (91% of total time), mostly waiting on shuffle
   - Evidence: Reduce shuffle bytes=206958652 (207 MB)
   - After optimization: 121 MB (41.5% reduction) → saved 78 seconds

   Why it's the bottleneck:
   - Data must be transferred over network between mappers and reducers
   - All 68M edges converted to node-degree pairs require sorting and grouping
   - Network bandwidth becomes the limiting factor

   Spark:
   - Minimal shuffle impact due to in-memory processing
   - No intermediate disk writes between stages

2. Disk I/O (Secondary Bottleneck - Hadoop)

   Hadoop MapReduce:
   - HDFS reads: 1,080,626,714 bytes (1.08 GB) - reading input
   - HDFS writes: 45,437,236 bytes (45.44 MB) - Job 1 output written to disk
   - Intermediate I/O: Job 2 must read Job 1's output from disk
   - Total I/O: ~1.13 GB read + written

   Impact:
   - Mandatory disk writes between Job 1 and Job 2
   - Added latency: ~10-15 seconds for HDFS operations

   Spark:
   - Minimal disk I/O: Only reads input (1.08 GB) and writes final output
   - No intermediate writes: RDD lineage keeps data in memory
   - Result: 1.8x faster than Hadoop baseline

3. Memory Usage (Critical Constraint - Both Systems)
   System Environment:
   - Total RAM: 3.82 GB
   - Available RAM: 1.08 GB (during execution)
   - Constraint: Limited memory affects both systems differently

   Hadoop Impact:
   Peak Map Memory: 599 MB (baseline) → 575 MB (optimized)
   Peak Reduce Memory: 402 MB (baseline) → 531 MB (optimized)
   Task proliferation: 9 tasks → 34 tasks (due to memory pressure)
   - More tasks created because Hadoop couldn't fit large splits in memory
   - Tasks queued instead of running in parallel
   - Garbage Collection (GC) overhead increased: 31.9s → 34.9s

   Spark Impact:
   Memory Store: 434.4 MiB (baseline) → 127.2 MiB (optimized)
   Executor Memory: 2GB (baseline) → 1GB (optimized)
   - Caching attempt failed (would have been 264s instead of 206s)
   - Had to use conservative partitioning (8 partitions) to avoid memory spilling
   - Memory constraint limited optimization choices

HADOOP : Applied Optimizations and Measured Effects

- Optimization Applied: Snappy Compression
- Rationale: Reduce network shuffle overhead, which was the primary bottleneck (207 MB transferred between map and reduce phases).
- Measured Results:
  - Execution Time: 373s → 297s (20.4% improvement)
  - Shuffle Data: 207 MB → 121 MB (41.5% reduction)
  - Trade-off: Increased task count (9→34) and GC overhead (+9.4%) due to VM resource constraints
- Conclusion: Compression successfully targeted the network bottleneck but revealed secondary memory constraints.

Apache Spark: Applied Optimizations and Measured Effects

- Optimization Applied: Partitioning Optimization
- Rationale: Balance parallelism with memory constraints. Default partitions caused excessive memory pressure; 8 partitions optimize CPU utilization without overwhelming limited VM memory.
- Measured Results:
  - Execution Time: 206s → 185s (10.2% improvement)
  - Distribution Stage: 135.56s → 10.89s (92% faster)

    - Memory: Conservative allocation (1GB executor, 512MB driver) avoided caching failure
- Conclusion: Controlled partitioning achieved better parallelism while staying within memory limits.

Critical Analysis

Why Hadoop and Spark Show Different Performance Patterns

Core Architectural Difference:

Hadoop MapReduce : Disk-Based Processing

- Must write intermediate results to HDFS between jobs
- Job 1 output (45 MB) → disk → Job 2 input
- Every map-reduce cycle involves disk I/O
- Result: Network and disk-bound

Apache Spark : In-Memory Processing

- Keeps data in memory across operations using RDD lineage
- No intermediate disk writes required
- All transformations pipelined in single execution
- Result: Memory-bound, much faster

Which System is Better Suited for Large-Scale Graph Data

Apache Spark is superior for graph analysis.

Evidence from Experiments:

Metric	Hadoop	Spark	Winner
Baseline Time	373s	206s	Spark (1.81×)
Optimized Time	297s	185s	Spark (1.61×)
Memory Efficiency	Low (task proliferation)	High (controlled partitions)	Spark
Iterative Capability	Poor (disk I/O per iteration)	Excellent (in-memory)	Spark

How experimental findings align with theoretical complexity and system design principles

Theoretical Principle	Experimental Finding	Alignment
O(E+V) complexity	Both systems identical	Perfect
Network shuffle costly	207 MB → 41% reduction	Confirmed
Disk I/O slower than memory	Spark 1.8× faster	Confirmed
Caching improves performance	Failed due to memory limits	Context-dependent
More parallelism = better	8 optimal, 34 worse	Diminishing returns (Due to VM resource constraints)
Resource constraints dominate	VM limits affected both systems	Strongly confirmed

Definitions:

- *O(E + V) is a time-complexity expression commonly used in graph algorithms.*
- *V = number of vertices (nodes) in the graph*
- *E = number of edges (connections between nodes)*
- *Confirmed = validated and supported the theoretical principle that were testing*
- *Diminishing returns= means that while the theoretical principle held true, the benefits decreased as things scaled up*

Experiments verify fundamental theory in distributed systems that disk vs. memory, network bottlenecks but with important distinction, effectiveness of optimizations depends on both matching techniques to workload characteristics and to available resources. Theoretical gains may not necessarily translate to real-world improvements when deployed in resource constrained environments.

The gap between Hadoop - 373s and Spark - 206s is not an implementation detail; rather, it reflects a fundamental architectural philosophy: reliability through persistence versus speed through memory. In graph analytics, speed wins, so Spark is theoretically and experimentally superior.

**** *Refer t1 folder of the zip folder for coding and configurations*

Task 2: Real-Time IoT Sensor Data Analytics using Apache Kafka

Environment Setup

System Requirements

- Operating System: Ubuntu 22.04 LTS or compatible Linux distribution
- Apache Kafka: Version 3.9.0 with Zookeeper
- Python: Version 3.8+
- PostgreSQL: Version 16
- Docker & Docker Compose: For containerized services
- Tools: Grafana, Kafdrop

Python Dependencies Installation

Required packages

- pandas - Data manipulation and analysis
- kafka-python - Kafka client library
- pyyaml - YAML configuration parsing
- psycopg2-binary - PostgreSQL database adapter

Infrastructure Setup

Step 1: Start PostgreSQL and supporting services

- PostgreSQL - Long-term Data Storage & Analytics
- Grafana - Real-time Visualization & Monitoring
- Kafdrop - Kafka Cluster Management & Debugging

These services are run as docker containers using docker composer in host machine.

Step 2: Start Kafka cluster, Zookeeper

Separate script (kafka-control.sh) is used for functions like Start, Stop get Status Kafka cluster with Zookeeper. Kafka and Zookeeper are installed in Ubuntu Operating System in a virtual box.

Service Ports:

- Zookeeper: 2181
- Kafka Broker: 9092
- PostgreSQL: 5432
- Grafana: 3000
- Kafdrop: 9000

Kafka Topic Design and Configuration

Topic Architecture

Primary Input Topic:

- Name: traffic-data
- Purpose: Raw traffic detector data from Austin Dataset
- Partitioning: By detector_id for load balancing
- Retention: 90 days

Output Topics:

- hourly-rolling-metrics : Hourly vehicle count aggregations
- daily-peak-metrics : Daily traffic peaks and volumes
- sensor-availability-metrics : Sensor health status

Topic Specifications:

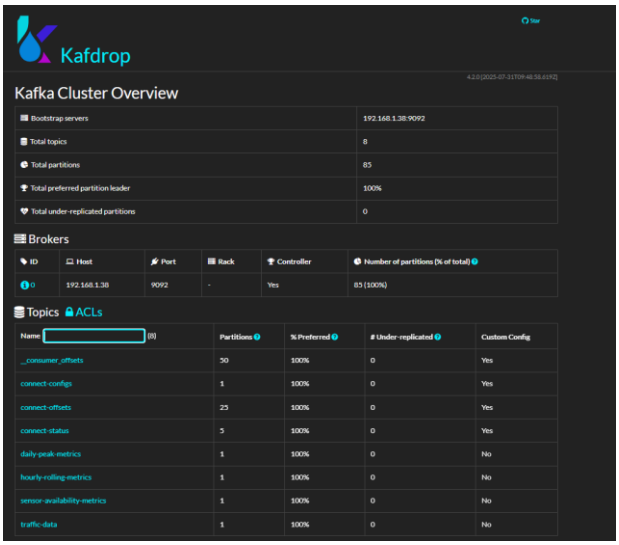
- Partitions: 3 (for parallel processing)
- Replication Factor: 1 (single-node setup)
- Retention: 90 days (7,776,000,000 ms)
- Compression: Uncompressed (for simplicity)
- Cleanup Policy: Delete (time-based retention)

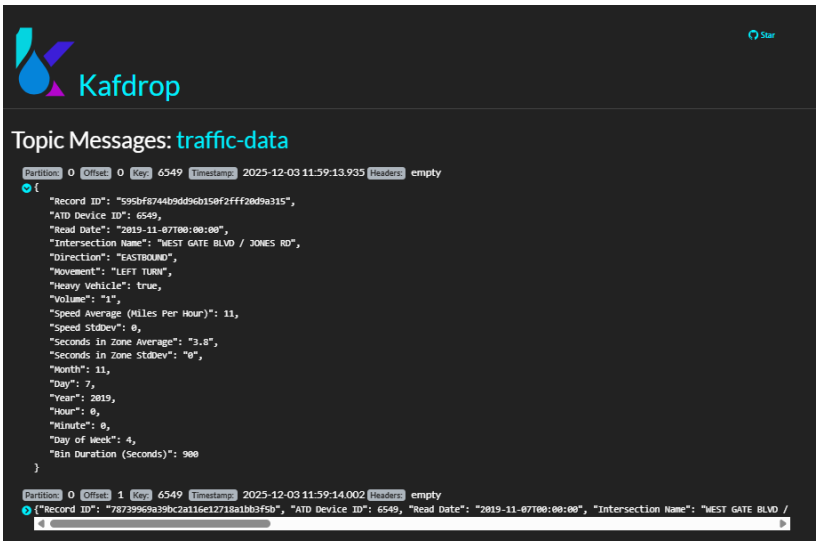
Retention Configuration Rationale:

- 3-month retention ensures historical data availability
- 100GB storage limit prevents disk space issues
- Uncompressed data eliminates codec dependencies
- Delete policy maintains clean, time-bounded data

Topic Creation

kafka-topic-creation.sh contains the scripts to be run in bash in Ubuntu virtual box to create required kafka topics.





Producer and Consumer Logic

Producer Implementation

Producer Architecture:
The producer (producer.py) implements a CSV streaming simulator that processes historical data to simulate real-time sensor feeds.

- Batch Processing (just to simulate): 50 records every 5 seconds
- Type Safety: Robust data type conversion
- Error Handling: Graceful handling of malformed CSV data
- Progress Tracking: Detailed logging and statistics

Data Processing Pipeline
Key Processing Functions:

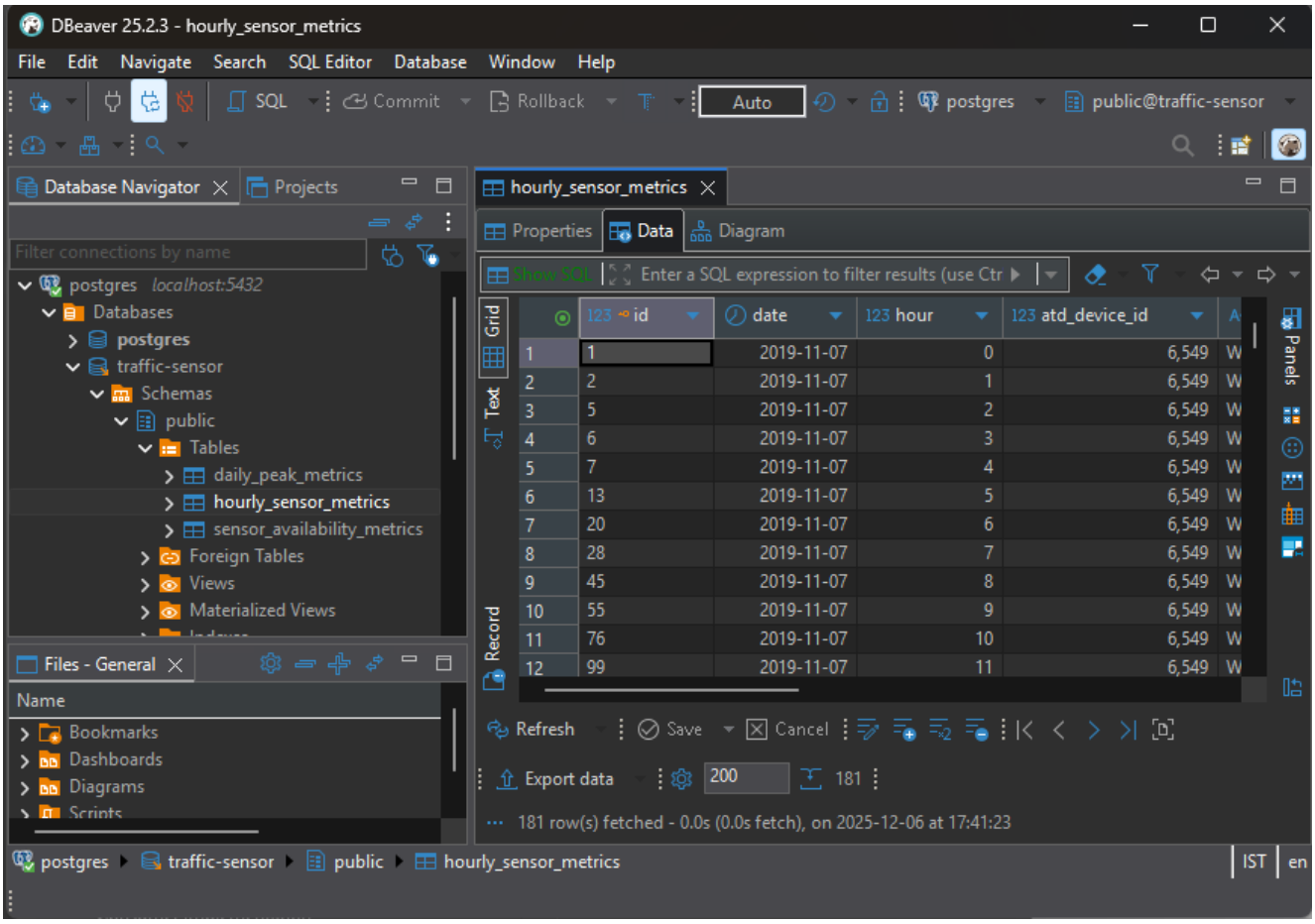
- Extracts numeric values safely
- Handles various Boolean representations
- Standardizes string fields
- Converts timestamps to ISO format

Consumer Implementation

Consumer Processing Steps:

1. Message Consumption
    - Consumes from traffic-data topic
    - Processes messages in real-time
2. Vehicle Detection Logic
    - Monitors modified_date field changes
    - Detects new vehicle events by comparing timestamps
    - Maintains state per sensor
3. Metrics Calculation
    - Updates hourly vehicle counts
    - Calculating daily traffic volumes
    - Tracks sensor availability
4. Output Publishing
    - Publishes metrics to Kafka output topics
    - Persists data to PostgreSQL database

Persists data to PostgreSQL database



Metric Computation Methodology

Hourly Average Vehicle Count per Sensor

Real-Time Calculation Approach:

- Updates: Every single record arrival
- Method: Running sum ÷ Running count
- Key: "date|hour|device_id"

Mathematical Formula:
Rolling Average = Total Volume So Far / Number of Records So Far

Example:
- Record 1: Volume 15 → Average = 15/1 = 15.0
- Record 2: Volume 23 → Average = (15+23)/2 = 19.0
- Record 3: Volume 18 → Average = (15+23+18)/3 = 18.67

Why This Approach:

- Most accurate: Accounts for variable record counts per hour
- Real-time responsive: Updates immediately with new data
- Handles gaps: Works even when sensors have missing data

Daily Peak and Daily Average Traffic Volume Across All Sensors

Daily Peak Hour Traffic Volume Detection Logic:

- Aggregation: Sum all sensor volumes per hour
- Peak Identification: Hour with maximum total volume
- Updates: Recalculated as new data arrives

Calculation Process:
Example:
Date: 2019-01-01
Hour 8: Sensor A=50 + Sensor B=75 + Sensor C=60 = 185 total
Hour 9: Sensor A=45 + Sensor B=80 + Sensor C=55 = 180 total
Hour 17: Sensor A=85 + Sensor B=95 + Sensor C=90 = 270 total ← PEAK
Hour 18: Sensor A=70 + Sensor B=88 + Sensor C=75 = 233 total

Result: Peak hour = 17 (5 PM) with 270 vehicles total

Average Daily Traffic Volume Detection Logic:

Calculation Process:

- Get Total Daily Volume - Sum of all hourly volumes across all sensors for the entire day
  Example: If all 24 hours sum to 4,320 total vehicles
- Calculate Average - Divide total daily volume by 24 hours
  Example: 4,320 / 24.0 = 180 vehicles per hour (average)
- Round Result - Round to 2 decimal places for readability

Daily Sensor Availability Percentage

Availability Calculation Method:

- Expected hours: Hours elapsed in day so far (real-time)
- Actual hours: Count of unique hours with data
- Percentage: (Actual / Expected) × 100

Status Classification:

if availability percentage >= 90: status = 'EXCELLENT'
else if availability percentage >= 75: status = 'GOOD'
else if availability percentage >= 50: status = 'POOR'
else: status = 'CRITICAL'

Example Calculation:

Sensor 123 on 2019-01-01:
- Data seen in hours: {0, 1, 2, 4, 5, 7, 8, 9, 10}
- Actual hours: 9
- Current hour: 10 (so 11 hours have elapsed: 0-10)
- Availability: 9/11 = 81.8% → Status: GOOD
- Missing hours: 2 (gaps in hours 3 and 6)

Gap Detection Benefits:

- Real-time sensor health: Immediate detection of sensor failures
- Maintenance alerting: Identify sensors needing attention
- Data quality assessment: Understand completeness of traffic data

Grafana Dashboard Setup and Interpretation

Dashboard Architecture

The Grafana dashboard provides real-time visualization of all three core metrics through 4 main Visualizations:

1. Hourly Average Vehicle Count Per Sensor (Time Series)
2. Daily Peak vs Daily Average Traffic Volumes (Bar Chart)
3. Daily Sensor Availability Percentage (Table)
4. Sensor Health Overview (Gauges)

Data Source Configuration

1. PostgreSQL Data Source Setup:

- Host: localhost:5432
- Database: traffic-sensor
- User: grafana
- Password: grafana
- SSL Mode: disable

2. Security Configuration:

- Read-only user 'grafana' for security
- Connected only to 'traffic-sensor' database
- Use only the public schema
- Select permission to hourly_sensor_metrics, daily_peak_metrics and sensor_availability_metrics tables only.

3. Key Dashboard Queries

- Query 1: Hourly Average Vehicle Count Per Sensor
- Query 2: Daily Peak vs Daily Average Traffic Volumes
- Query 3: Daily Sensor Availability Percentage
- Query 4: Overall Sensor Health

Dashboard Visualization Configuration

Time Series Panel:

- Query: Hourly Average Vehicle Count Per Sensor
- Visualization: Time series
- Y-axis: Vehicle count
- Legend: Sensor ID labels
- Refresh: 5s (matches processing frequency)

Comparison Panel:

- Query: Daily Peak vs Daily Average Traffic Volumes
- Visualization: Bar Chart
- Y-axis: Vehicle count
- Series: Daily Peak (bar), Daily Average (bar)
- Colors: Contrasting colors for clarity

Availability Table:

- Query: Daily Sensor Availability Percentage
- Visualization: Table
- Columns: Sensor ID, location, Availability, Status, Actual Active Hours, Expected Active Hours, Missing Hours, First Active Hour, Last Active Hour
- Conditional formatting: Color-coded by availability percentage

Health Overview:

- Query: Overall Sensor Health
- Visualizations: Guage
- Values: Current counts per status category

Dashboard Refresh and Real-Time Updates

Auto-Refresh Configuration:

- Refresh Rate: 5 seconds (matches processing batch interval)
- Time Range: Last 3 days for trends, current day for availability
- Live Updates: Dashboard updates automatically as new data arrives

Performance Observation

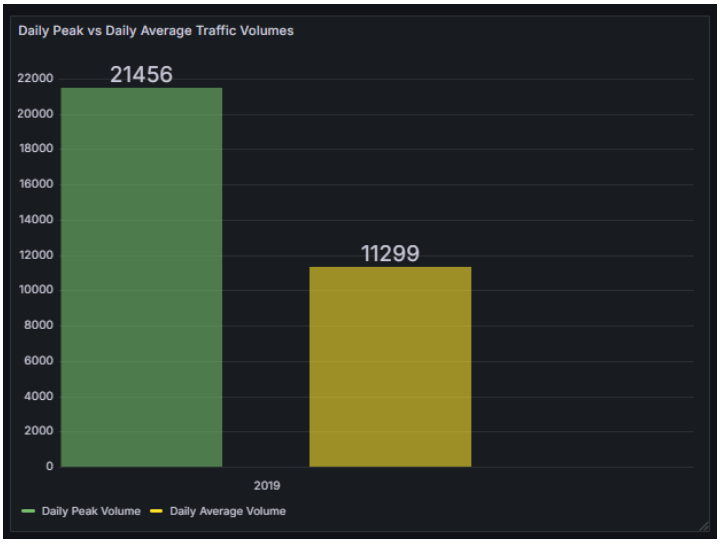
- Processing Rate: 10 records/second
- Update Frequency: Every 5 seconds
- End-to-End Latency: Under 5 seconds
- Resource Usage: Low CPU and memory

Grafana dashboard setup and interpretation

Grafana Dashboard

Hourly average vehicle count per sensor

Daily Peak Traffic Volume and Daily Average Traffic Volume across all sensors

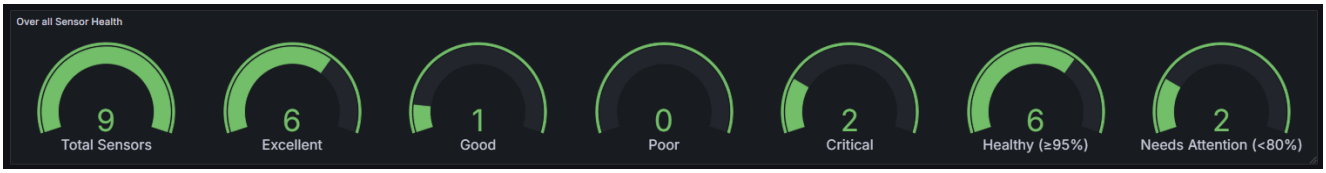


Daily sensor availability (%)

Daily Sensor Availability Percentage

Sensor ID	Location	Availability %	Status	Actual Active Hours	Expected Active Hours	Missing Hours	First Active Hour	Last Active Hour
6549	WEST GATE BLVD / JONES RD	100	EXCELLENT	24	24	0	0	23
6957	SLAUGHTER LN /	100	EXCELLENT	24	24	0	0	23
6991	FM 620 RD / ANDERSON MILL RD	100	EXCELLENT	24	24	0	0	23
6197	GROVE BLVD / RIVERSIDE DR	87.5	GOOD	21	24	3	0	20
6354	EXPOSITION BLVD / WINDSOR RD	16.7	CRITICAL	4	24	20	0	3
7047	MANOR RD / 51ST ST	8.33	CRITICAL	2	24	22	0	1

Overall Sensor Health



***** Refer t2 folder of the zip folder for coding and configurations*

Task 3: Graph Databases using Neo4j and Docker

Part 1 - Understanding Graph Databases

1. Social Network Connections

Graph databases are useful in use cases that require the ability to navigate relationships between people, such as finding friends-of-friends or analyzing social circles. In a relational database, discovering indirect connections requires many expensive JOIN operations, which can get much slower as the network grows. Graph databases navigate these relationships natively, by following the links connecting the nodes, making questions about the degree of separation or proximity in the network extremely fast, regardless of the network size.

What kind of data would be stored (nodes and relationships)
Nodes:

- Person (properties: user_id, name, age, location, interests)
- Post (properties: post_id, content, timestamp)
- Group (properties: group_id, name, category)

Relationships:

- FRIENDS_WITH (Person → Person, properties: since_date, connection_strength)
- FOLLOWS (Person → Person, properties: follow_date)
- POSTED (Person → Post)
- LIKES (Person → Post, properties: timestamp)
- MEMBER_OF (Person → Group, properties: join_date, role)

Example query: Find shortest path between two people.

Discovers the shortest chain of friendships connecting two people, showing how they're connected through mutual connections. This type of query would require recursive CTEs or multiple self-joins in SQL, becoming highly expensive for large networks.

2. Recommendation Engines

Graph databases are much better at handling this complexity when it comes to building systems that make recommendations around products, content, and connections based on patterns of behavior and preference. The multi-dimensional relationships between users, items, ratings, and behaviors pose a difficult challenge in relational databases. Graph databases natively represent these kinds of interconnected patterns, allowing you to quickly navigate from one user through their preferences to similar users and their choices, which makes pattern discovery and recommendation generation so much more efficient.

What kind of data would be stored (nodes and relationships)
Nodes:

- User (properties: user_id, name, age, location, preferences)
- Product (properties: product_id, name, category, price, attributes)
- Category (properties: category_id, name)
- Brand (properties: brand_id, name)

Relationships:

- PURCHASED (User → Product, properties: date, quantity, price_paid)
- VIEWED (User → Product, properties: timestamp, duration)
- RATED (User → Product, properties: rating, review_text, date)
- BELONGS_TO (Product → Category)
- MADE_BY (Product → Brand)
- SIMILAR_TO (Product → Product, properties: similarity_score)

Example query: Collaborative filtering recommendation

Identifies users with similar recent purchase patterns, then recommends highly rated products from those similar users that the target user hasn't encountered yet. The query naturally expresses collaborative filtering logic by passing through the purchase and rating relationships, aggregating signals across multiple similar users.

3. Fraud Detection

To identify fraudulent activity, one often needs to uncover the linkages in accounts, transactions, devices, and contact information. Relational databases require complex queries with many JOINs to uncover these indirect relationships. Graph databases make it straightforward to trace networks of suspicious connections, identify fraud rings, and spot patterns when several accounts share overlapping attributes. Having this ability to quickly explore and visualize these connection networks in real time is key in effectively dealing with fraud.

What kind of data would be stored (nodes and relationships)
Nodes:

- Account (properties: account_id, email, phone, creation_date, status)
- Transaction (properties: transaction_id, amount, timestamp, status)
- Device (properties: device_id, type, fingerprint)
- IPAddress (properties: ip, location, is_proxy)
- CreditCard (properties: card_number_hash, card_type, issuer)
- Address (properties: street, city, postal_code)

Relationships:

- OWNS (Account → CreditCard)
- USES_DEVICE (Account → Device, properties: last_used)
- LOGGED_IN_FROM (Account → IPAddress, properties: timestamp)
- MADE_TRANSACTION (Account → Transaction)
- USES_CARD (Transaction → CreditCard)
- SHIPS_TO (Account → Address)
- LINKED_TO (Account → Account, properties: link_type, confidence_score)

Example query: Detect fraud ring by shared attributes

Starting with a known suspicious account, this query identifies a potential fraud ring by finding other accounts which share multiple attributes like same device, IP address, or shipping address- with the flagged account. The broader network will then be explored to gain an understanding of how these various accounts are interrelated, exposing organized fraud operations in the process. This type of pattern matching and traversal through multiple relationship types simultaneously is quite natural in graph databases, but extremely complex and slow in relational systems.

Part 2 - Implementation with Neo4j (Docker Setup)

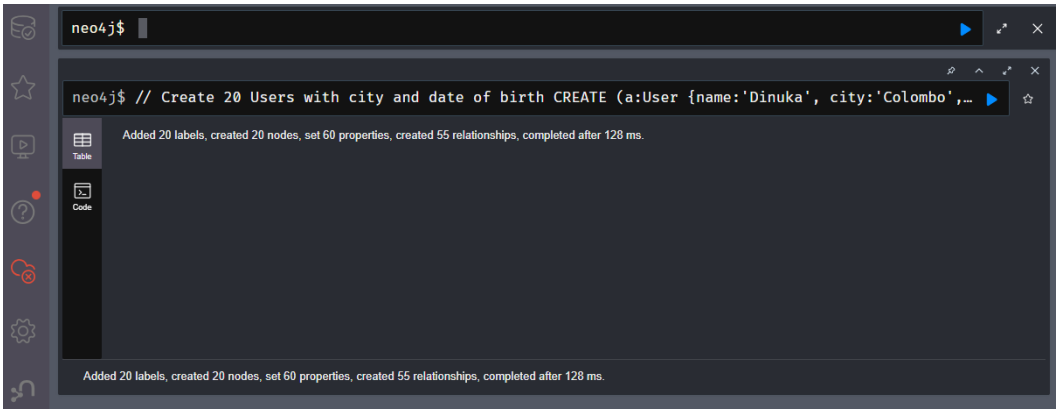
- Implementing a social network by creating a small dataset with Users and Relationships

Cypher Query

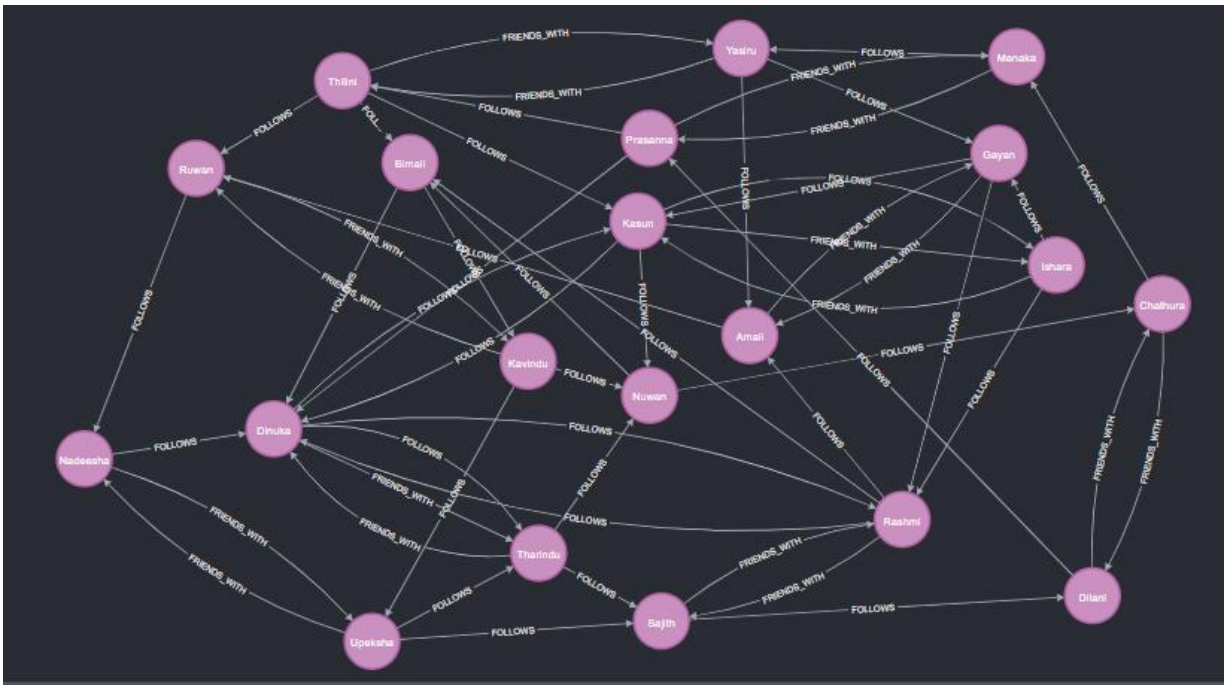
```
// Create 20 Users with name, city and date of birth
CREATE
  (a:User {name:'Dinuka', city:'Colombo', dob: date('1995-04-12')}),
  (b:User {name:'Kasun', city:'Kandy', dob: date('1996-08-23')}),
  (c:User {name:'Rashmi', city:'Galle', dob: date('1997-03-15')}),
  (d:User {name:'Tharindu', city:'Negombo', dob: date('1994-11-05')}),
  (e:User {name:'Ishara', city:'Matara', dob: date('1995-12-30')}),
  (f:User {name:'Nuwan', city:'Kurunegala', dob: date('1996-07-18')}),
  (g:User {name:'Sajith', city:'Colombo', dob: date('1993-09-14')}),
  (h:User {name:'Amali', city:'Kandy', dob: date('1998-02-05')}),
  (i:User {name:'Gayan', city:'Galle', dob: date('1995-08-12')}),
  (j:User {name:'Bimali', city:'Matara', dob: date('1997-01-20')}),
  (k:User {name:'Chathura', city:'Negombo', dob: date('1994-10-11')}),
  (l:User {name:'Dilani', city:'Kurunegala', dob: date('1996-05-22')}),
  (m:User {name:'Ruwan', city:'Colombo', dob: date('1995-04-30')}),
  (n:User {name:'Kavindu', city:'Kandy', dob: date('1996-06-15')}),
  (o:User {name:'Menaka', city:'Galle', dob: date('1995-07-09')}),
  (p:User {name:'Prasanna', city:'Matara', dob: date('1993-12-03')}),
  (q:User {name:'Nadeesha', city:'Colombo', dob: date('1998-03-18')}),
  (r:User {name:'Upeksha', city:'Negombo', dob: date('1997-08-25')}),
  (s:User {name:'Yasiru', city:'Kandy', dob: date('1994-05-16')}),
  (t:User {name:'Thilini', city:'Galle', dob: date('1996-11-21')}),

// FOLLOWS relationships (directed)

  (a)-[:FOLLOWS]->(b),
  (a)-[:FOLLOWS]->(c),
  (a)-[:FOLLOWS]->(d),
  (b)-[:FOLLOWS]->(a),
  (b)-[:FOLLOWS]->(e),
  (b)-[:FOLLOWS]->(f),
  (c)-[:FOLLOWS]->(a),
  (c)-[:FOLLOWS]->(h),
  (c)-[:FOLLOWS]->(j),
  (d)-[:FOLLOWS]->(f),
  (d)-[:FOLLOWS]->(g),
  (e)-[:FOLLOWS]->(c),
  (e)-[:FOLLOWS]->(i),
  (f)-[:FOLLOWS]->(j),
  (f)-[:FOLLOWS]->(k),
  (g)-[:FOLLOWS]->(l),
  (h)-[:FOLLOWS]->(m),
  (i)-[:FOLLOWS]->(b),
  (i)-[:FOLLOWS]->(c),
  (j)-[:FOLLOWS]->(a),
  (j)-[:FOLLOWS]->(n),
  (k)-[:FOLLOWS]->(o),
  (l)-[:FOLLOWS]->(p),
  (m)-[:FOLLOWS]->(q),
```

```
  (n)-[:FOLLOWS]->(r),
  (o)-[:FOLLOWS]->(s),
  (p)-[:FOLLOWS]->(a),
  (p)-[:FOLLOWS]->(t),
  (q)-[:FOLLOWS]->(a),
  (r)-[:FOLLOWS]->(d),
  (r)-[:FOLLOWS]->(g),
  (s)-[:FOLLOWS]->(h),
  (s)-[:FOLLOWS]->(i),
  (t)-[:FOLLOWS]->(b),
  (t)-[:FOLLOWS]->(j),
  (t)-[:FOLLOWS]->(m),
  (n)-[:FOLLOWS]->(f),

// FRIENDS_WITH relationships (bidirectional)

  (a)-[:FRIENDS_WITH]->(d), (d)-[:FRIENDS_WITH]->(a),
  (b)-[:FRIENDS_WITH]->(e), (e)-[:FRIENDS_WITH]->(b),
  (c)-[:FRIENDS_WITH]->(g), (g)-[:FRIENDS_WITH]->(c),
  (h)-[:FRIENDS_WITH]->(i), (i)-[:FRIENDS_WITH]->(h),
  (k)-[:FRIENDS_WITH]->(l), (l)-[:FRIENDS_WITH]->(k),
  (m)-[:FRIENDS_WITH]->(n), (n)-[:FRIENDS_WITH]->(m),
  (o)-[:FRIENDS_WITH]->(p), (p)-[:FRIENDS_WITH]->(o),
  (q)-[:FRIENDS_WITH]->(r), (r)-[:FRIENDS_WITH]->(q),
  (s)-[:FRIENDS_WITH]->(t), (t)-[:FRIENDS_WITH]->(s);
```

- Run above inside the Neo4j Browser Query Editor Window



Output as a graph



- Write three Cypher queries and show the results

1. Find all users connected to User 'Dinuka'

Cypher Query

```
MATCH (a:User {name:'Dinuka'})--(u:User)
RETURN DISTINCT u.name AS Connected_To_Dinuka;
```

Browser view and Query Results

Explanation

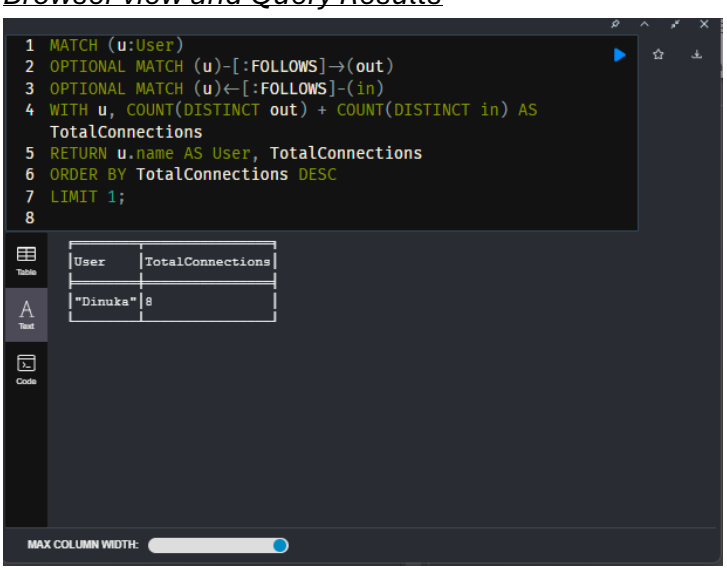
- **a and u** are variable names representing nodes.
- **MATCH** key word is used to find patterns in the graph.
- **(a:User {name:'Dinuka'})** means start by finding a node with label *User* and property name = *'Dinuka'*
- **-- (double dash)** represents a single relationship of length 1. Here the relationship type (FOLLOWS, FRIENDS_WITH) and direction both are ignored. So, direction could be either a -- > u or a < -- u.
- **u.name** returns the name property of the connected node.
- **DISTINCT** key word ensures each connected user appears only once, even if multiple relationships exist between the same nodes.
- **AS Connected_To_Dinuka** gives the column a readable name in the results

2. Find the most connected node with the relationship type = FOLLOWS

Cypher Query

```
MATCH (u:User)
OPTIONAL MATCH (u)-[:FOLLOWS]->(out)
OPTIONAL MATCH (u)<-[:FOLLOWS]-(in)
WITH u, COUNT(DISTINCT out) + COUNT(DISTINCT in) AS TotalConnections
RETURN u.name AS User, TotalConnections
ORDER BY TotalConnections DESC
LIMIT 1;
```

Browser view and Query Results



Explanation

- **MATCH** key word is used to find patterns in the graph
- **(u:User)**
  This matches all nodes in the graph that have the label User, assigning each one to the variable u.
- **OPTIONAL MATCH (u)-[:FOLLOWS]->(out)**
  This tries to find all outgoing FOLLOWS relationships from user u to other users.
  - out represents users that u follows.
  - OPTIONAL MATCH means that if no such relationships exist, the query will still return the user with a null value instead of excluding them.
- **OPTIONAL MATCH (u)<-[:FOLLOWS]-(in)**
  This tries to find all incoming FOLLOWS relationships to user u.
  - in represents users who follow u.
  - Again, OPTIONAL MATCH keeps users who have no followers.
- **WITH u, COUNT(DISTINCT out) + COUNT(DISTINCT in) AS TotalConnections**
  This collects the two previous matches and computes:
  - COUNT(DISTINCT out) → number of unique users u follows
  - COUNT(DISTINCT in) → number of unique users who follow u

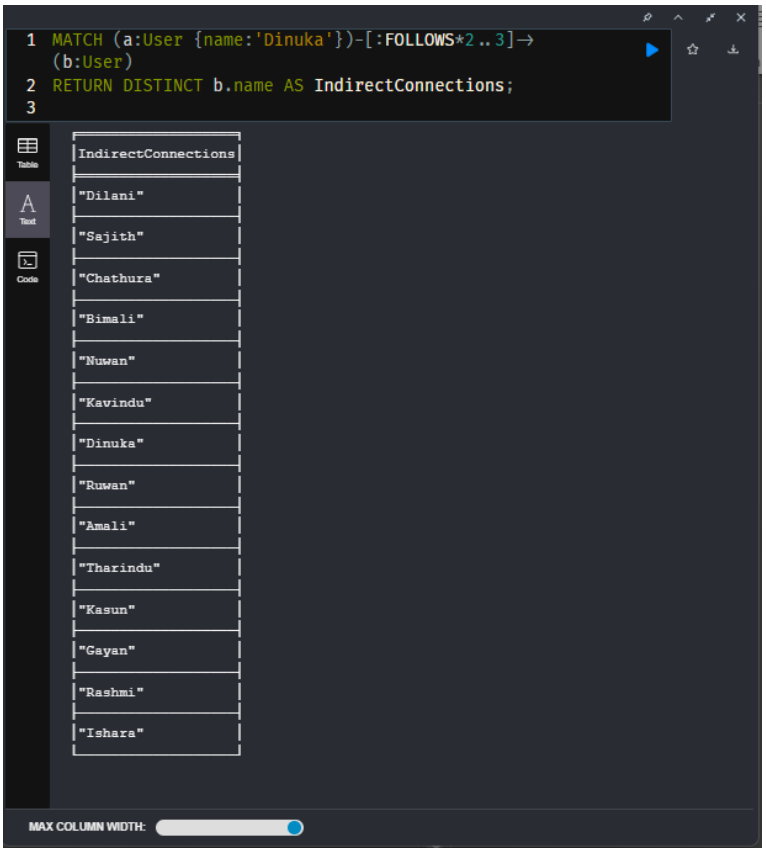
  These two counts are added to get the user's total connections (both followers and followings).
- **RETURN u.name AS User, TotalConnections**
  This outputs each user's name and their total number of follower+following connections.
- **ORDER BY TotalConnections DESC**
  Sorts the users from highest total connections to lowest.
- **LIMIT 1**
  Returns only the user with the most total FOLLOWS connections.

3. Indirect connections of ‘Dinuka’

Cypher Query

```
MATCH (a:User {name:'Dinuka'})-[:FOLLOWS*2..3]->(b:User)
RETURN DISTINCT b.name AS IndirectConnections;
```

Browser view and Query Results



Explanation

- **MATCH** key word is used to find patterns in the graph
- **(a:User {name:'Dinuka'})**
  This query will match the user node with label User and property name = 'Dinuka', assigning it to the variable a.
- **-[:FOLLOWS*2..3]->(b:User)**
  This pattern matches paths of FOLLOWS relationships that:
  - use the FOLLOWS relationship type
  - move in the outgoing direction (from a to b)
  - have a length between 2 and 3 hops, meaning:
    - 2 hops: a → x → b

    - 3 hops: a → x → y → b

  So, b is a user who is 2 or 3 steps away from Dinuka through outgoing FOLLOWS links. These are indirect relationships, not friends but friends-of-friends, or friends-of-friends-of-friends.

- **DISTINCT**
  Ensures that each user b appears only once, even if there are multiple different paths of length 2 or 3 leading to them.
- **AS IndirectConnections**
  Gives a readable name to the output column.

**** *Refer t3 folder of the zip folder for coding and configurations*

Task 4: Watermarks in Real-Time Stream Processing using Apache Kafka and Apache Flink (Docker Setup)

Part 1 - Understanding Watermarks

What are watermarks in stream processing ?

Watermarks are timestamps that instruct a streaming system when it is safe to stop waiting for more data to receive and create final results for a time window.

Example:

Imagine counting tweets in a social media platform in a 1 minute window from 2:00 to 2:01 PM. It is impossible to exactly know when to stop counting and show results without watermarks. With a watermark set to "5 seconds delay" the watermark at 2:01:05 PM is confident that all tweets from 2:00-2:01 PM have arrived by then. Then the system closes this window and prints to the console like "Final count: 847 tweets". Any tweets from that time period arriving after 2:01:05 PM are considered "late".

Why are they needed when processing real-time event streams (such as social media data)?

When processing social media data such as Facebook or Twitter streams, several timing challenges arise:

- Network delays: For example, a tweet published at 2:00 PM could be delayed due to network latency until it arrives in the streaming application at 2:01 PM.
- Out-of-order arrival: Posts by different users can arrive in the system out of order, especially if data is coming from multiple sources or geographic regions.
- System delays in processing: Various mobile applications may cache posts and send them in bulk when connectivity gets better.
- Clock skew : Different devices and servers could hold different system times.

Without watermarks, the windowed aggregations wouldn't have any idea at which point to close a window and emit results. Should it wait forever for events that are potentially late? Watermarks offer a practical approach by establishing a "cutoff point."

Three main types of watermark generation strategies

(a) Periodic Watermarks

The system generates watermarks periodically, such as every 1000ms, regardless of the incoming data. Flink generates these automatically either based on processing time or the maximum event time so far seen.

Pros:

- Low overhead since watermarks are generated infrequently
- Predictable watermark advancement
- Good for high throughput streams

Cons:

- Less precise timing control
- May delay results longer than necessary if events occur only occasionally

(b) Punctuated Watermarks

Watermarks are generated based on specific conditions in the data stream itself. For instance, it might embed special "watermark events" into the stream or generate watermarks when certain patterns in the data are detected. Custom logic examines each record and decides whether to emit a watermark.

Pros:

- Very precise control over watermark timing
- Can be tailored to specific data characteristics
- Best suitable for streams with forecastable patterns

Cons:

- Higher computational overhead as every record is examined.
- Complex to implement correctly
- Watermark stagnation if the conditions are not satisfied.

(c) Event Time Based Watermarks

Watermarks are produced using actual timestamps of the events in the stream. The watermark usually lags behind the maximum event time by a fixed duration.

Pros:

- Tied directly to actual event times.
- Handles out-of-order events gracefully Balance between accuracy and practicality

Cons:

- Watermarks may not progress if no new events arrive.
- Requires tuning of the "bounded out-of-orderness" parameter

Brief description of how each method handles late or out-of-order events, and which method might be best for real-world social media streams.

Periodic Watermarks: If the periodic interval is set correctly, periodic watermarks handle late events reasonably well. Events that arrive after the watermark for their time window are usually either dropped or sent to a side output.

Punctuated Watermarks: Can be designed to handle late events very precisely, requiring careful implementation to make sure watermarks still progress even when late events arrive.

Event-time-based Watermarks: Watermarks help deal with out-of-order data in streaming systems by defining acceptable lateness. The delay in the watermark parameter is used to indicate how much time to wait for events running late before closing time windows. As a result, this delays between accuracy (to wait for possibly delayed data) and latency (in producing timely results), and memory efficiency.

Event time-based watermarks are best for social media because posts are frequently received in an order different from when they went out because of network delays, devices losing Wi-Fi connectivity, and people posting while offline to sync later. Watermarks are useful in ensuring that results are accurate because they wait for posts that may be delayed (maybe 20 to 30 seconds) to get approximately 95 to 99% of the posts..

Part 2 - Implementation using Docker (Kafka + Flink)

Implementation Summary (Baseline)

Step 1 : Environment Setup

- Created docker-compose.yml with 5 services:
  - Kafka & Zookeeper for streaming infrastructure
  - Flink JobManager & 2 TaskManagers for processing
  - Kafdrop for Kafka UI monitoring
- Configured networking between containers
- Set up proper ports for external access (8081 for Flink UI, 9000 for Kafdrop)

Step 2 : Data and Kafka Topics

- Downloaded datasets: Facebook and Twitter CSV files from provided GitHub link and datasets were modified to have more hashtags for the experiment.
- Created create-topics.sh script to set up Kafka topics:
  - facebook-posts and twitter-posts topics
  - Initially 1 partition each

- Built producer.py Python application:
    - Reads CSV data and streams to Kafka topics every 0.1 seconds
    - Extracts hashtags from text content using regex
    - Handles JSON message formatting with platform metadata

Step 3 : Flink Stream Processing

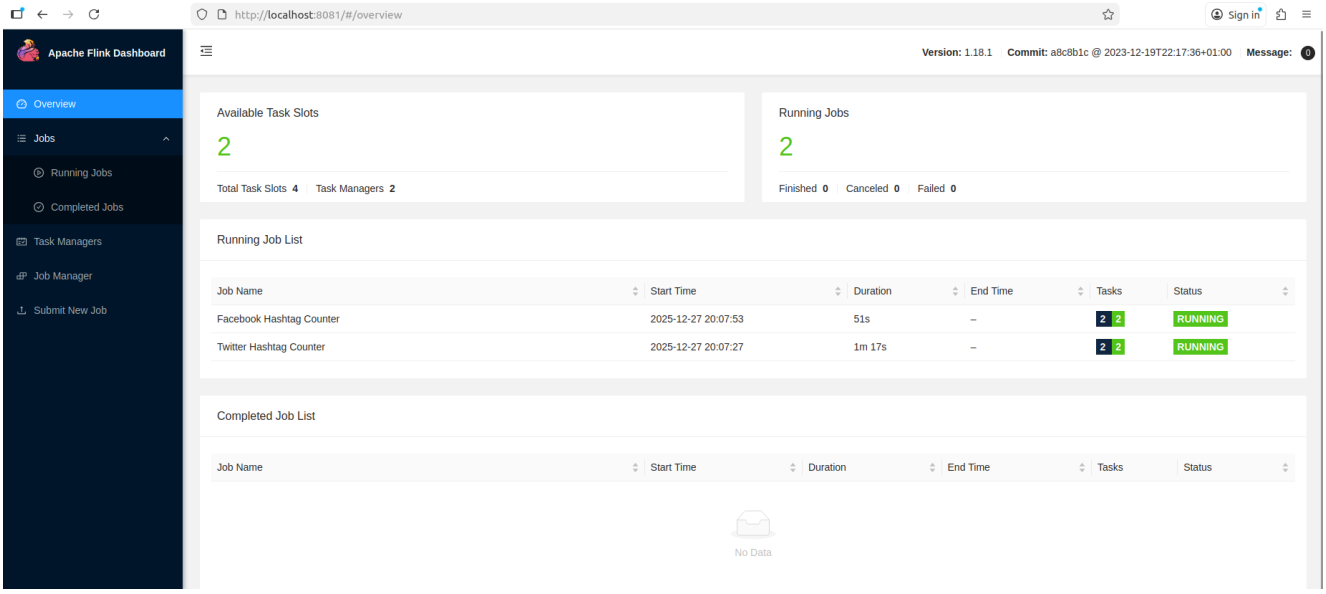
- Developed 2 Flink Java applications:
    - FacebookHashtagCounter.java - processes Facebook data
    - TwitterHashtagCounter.java - processes Twitter data
- Key features implemented:
    - Kafka source connectors for data ingestion
    - Hashtag extraction using regex pattern #(\w+)
    - 15-second tumbling windows for counting
    - Watermark strategy for out-of-order event handling
- Built with Maven and deployed as JAR to Flink cluster
- Used config.yml for centralized configuration management

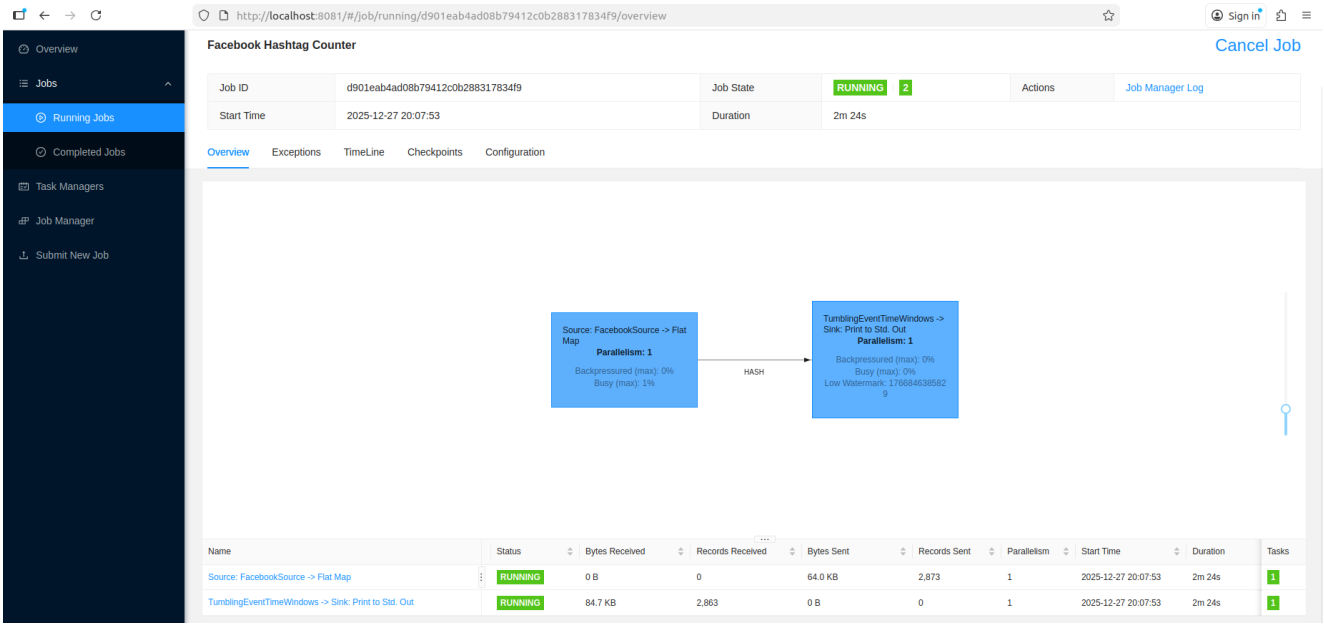
Kafka Topics partition Counts

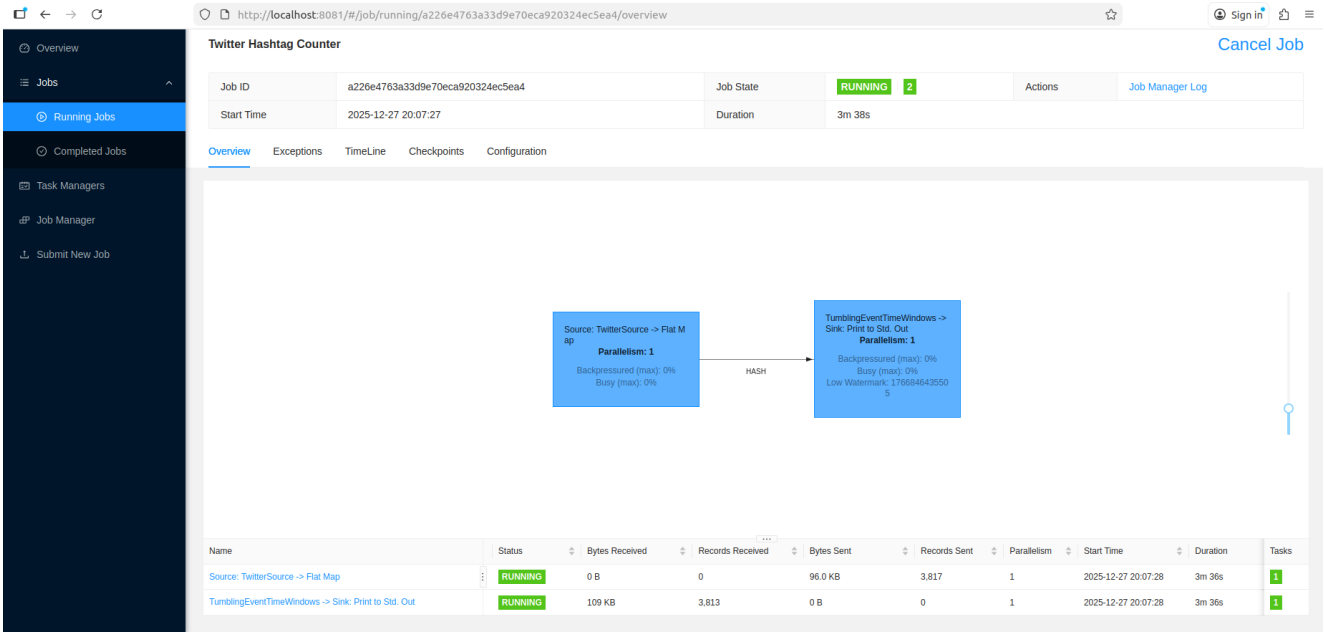
Topics ACLs

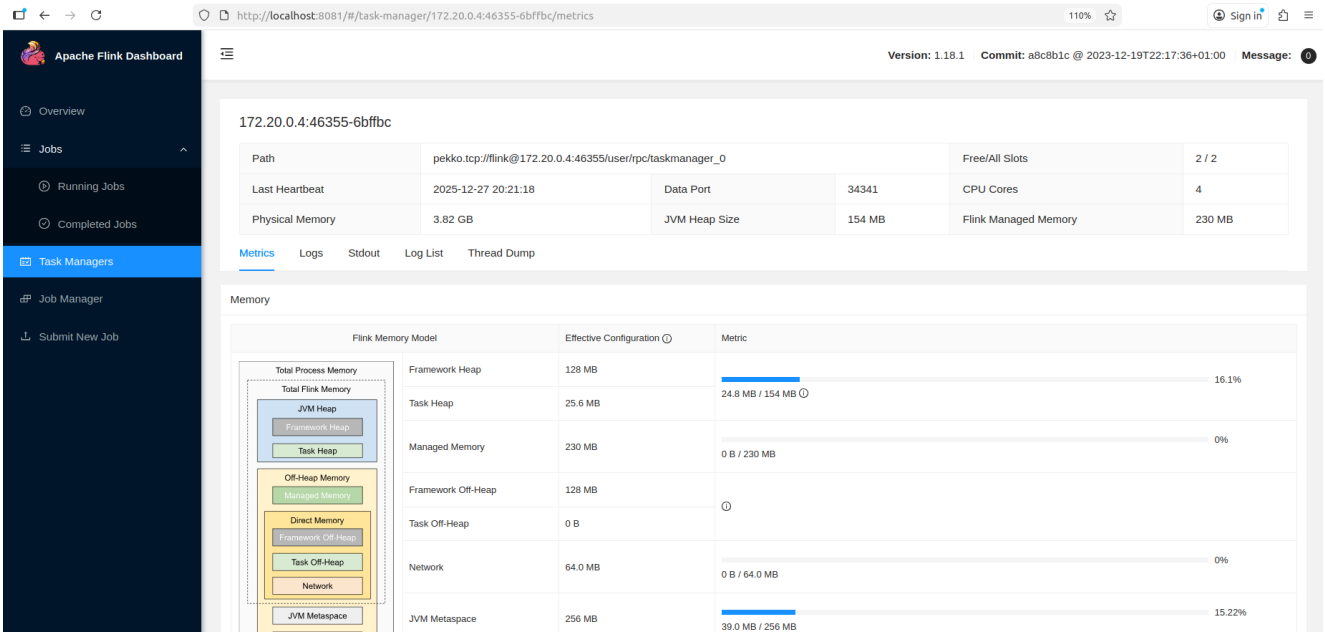
Name (2)	Partitions	% Preferred	# Under-replicated	Custom Config
facebook-posts	1	100%	0	No
twitter-posts	1	100%	0	No

Some screen captures during execution




http://localhost:8081/#/job/running/d901eab4ad08b79412c0b288317834f9/overview
Sign in
Overview
Jobs
Running Jobs
Completed Jobs
Task Managers
Job Manager
Submit New Job
Facebook Hashtag Counter
Cancel Job
Job ID
d901eab4ad08b79412c0b288317834f9
Job State
RUNNING
2
Actions
Job Manager Log
Start Time
2025-12-27 20:07:53
Duration
2m 24s
Overview
Exceptions
TimeLine
Checkpoints
Configuration
Source: FacebookSource -> Flat Map
Parallelism: 1
Backpressured (max): 0%
Busy (max): 1%
HASH
TumblingEventTimeWindows -> Sink: Print to Std. Out
Parallelism: 1
Backpressured (max): 0%
Busy (max): 0%
Low Watermark: 1766846385829
Name
Status
Bytes Received
Records Received
Bytes Sent
Records Sent
Parallelism
Start Time
Duration
Tasks
Source: FacebookSource -> Flat Map
RUNNING
0 B
0
64.0 KB
2,873
1
2025-12-27 20:07:53
2m 24s
1
TumblingEventTimeWindows -> Sink: Print to Std. Out
RUNNING
84.7 KB
2,863
0 B
0
1
2025-12-27 20:07:53
2m 24s
1


http://localhost:8081/#/job/running/a226e4763a33d9e70eca920324ec5ea4/overview
Sign in
Overview
Jobs
Running Jobs
Completed Jobs
Task Managers
Job Manager
Submit New Job
Twitter Hashtag Counter
Cancel Job
Job ID
a226e4763a33d9e70eca920324ec5ea4
Job State
RUNNING
2
Actions
Job Manager Log
Start Time
2025-12-27 20:07:27
Duration
3m 38s
Overview
Exceptions
TimeLine
Checkpoints
Configuration
Source: TwitterSource -> Flat Map
Parallelism: 1
Backpressured (max): 0%
Busy (max): 0%
HASH
TumblingEventTimeWindows -> Sink: Print to Std. Out
Parallelism: 1
Backpressured (max): 0%
Busy (max): 0%
Low Watermark: 1766846435505
Name
Status
Bytes Received
Records Received
Bytes Sent
Records Sent
Parallelism
Start Time
Duration
Tasks
Source: TwitterSource -> Flat Map
RUNNING
0 B
0
96.0 KB
3,817
1
2025-12-27 20:07:28
3m 36s
1
TumblingEventTimeWindows -> Sink: Print to Std. Out
RUNNING
109 KB
3,813
0 B
0
1
2025-12-27 20:07:28
3m 36s
1



Results Analysis

Accuracy Metrics

- Twitter: 99.2% hashtag coverage (4,068 out of 4,100 messages)
- Facebook: 92.3% hashtag coverage (3,784 out of 4,100 messages)
- Zero parse errors for both platforms
- Good hashtag density (2.94 avg for Twitter, 3.25 for Facebook)

Performance Metrics

- Twitter: 0.26ms avg latency, 3,905 msg/sec throughput
- Facebook: 0.28ms avg latency, 3,553 msg/sec throughput
- Sub-millisecond processing for both platforms

Resource Utilization

- CPU Usage: 15-18% (single TaskManager processing)
- JVM Heap Used: 24-26 MB
- JVM Heap Max: 154 MB
- Heap Utilization: 16-17%
- Active TaskManagers: 1
- Memory per TaskManager: ~425 MB total

Step 4: Scaling Experiment (Optimized)
Implementation Summary

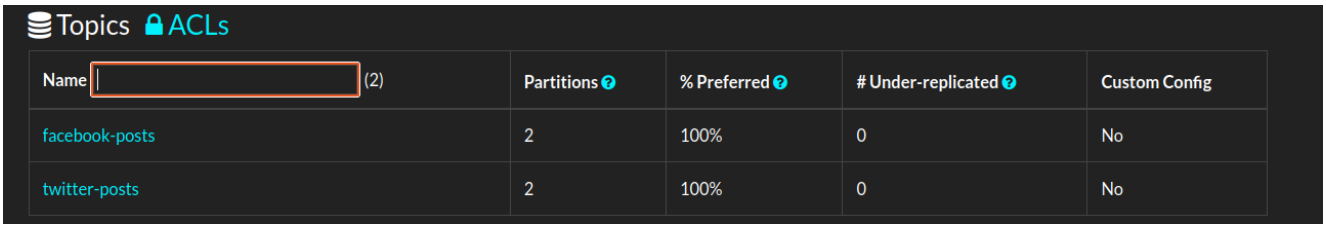
Java Files Changes

1. Parallelism Configuration
    - Added env.setParallelism(2) to force 2-way parallel processing
    - Changed from default parallelism to explicit 2-partition processing
2. Watermark Strategy Enhancement
    - Added .withIdleness(Duration.ofSeconds(2)) for partition-aware watermarking
3. Enhanced Metrics
    - Added Step 4 specific logging and metrics tracking
    - Added partition-aware performance indicators

Create Topics Script Changes

1. Partition Configuration
    - Modified --partitions 1 to --partitions 2 for both topics
    - Enabled distributed processing across 2 partitions

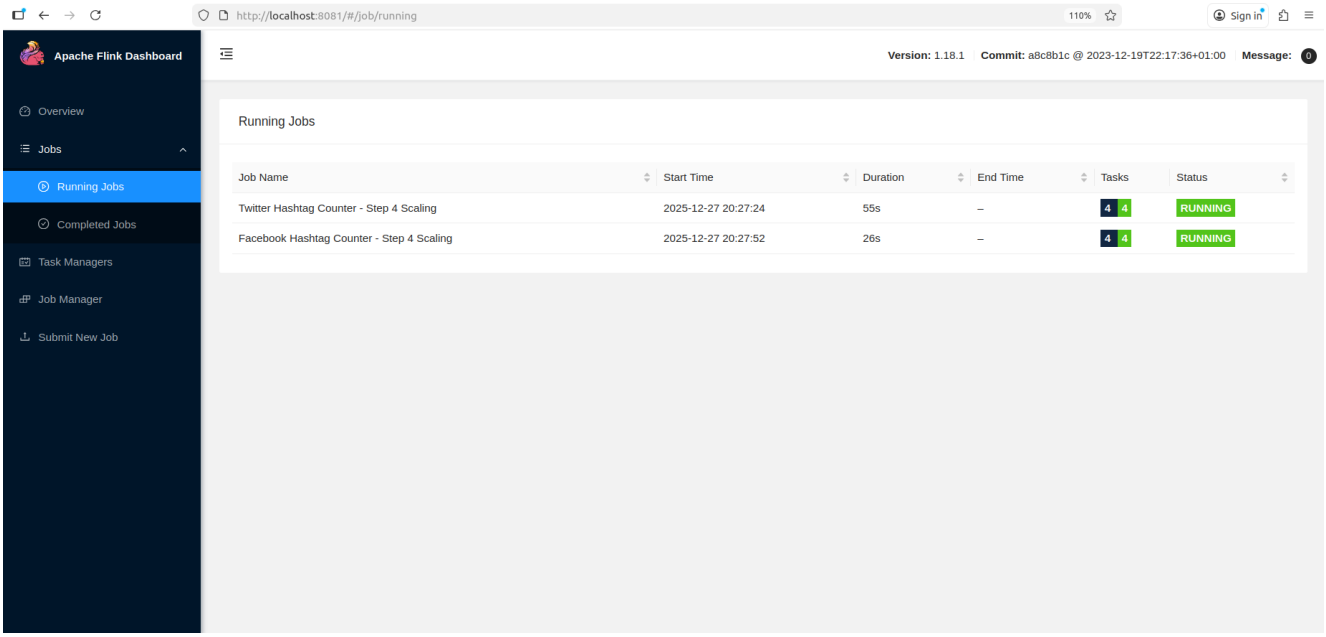
Kafka Topics partition Counts

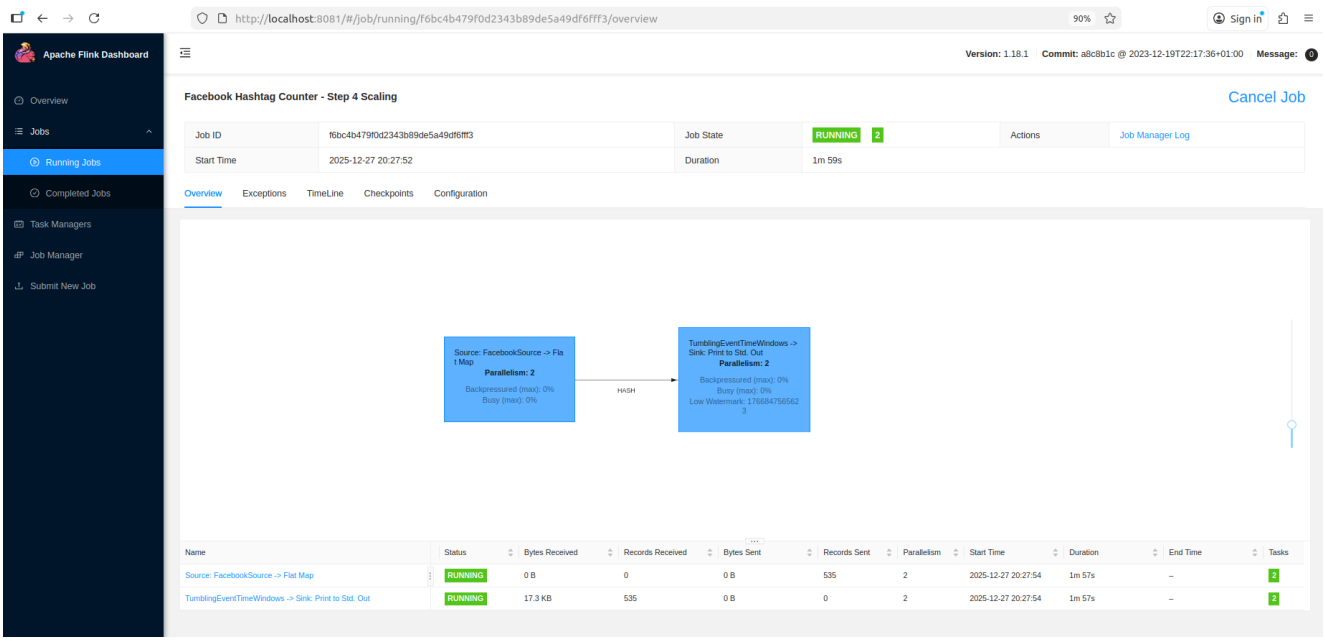


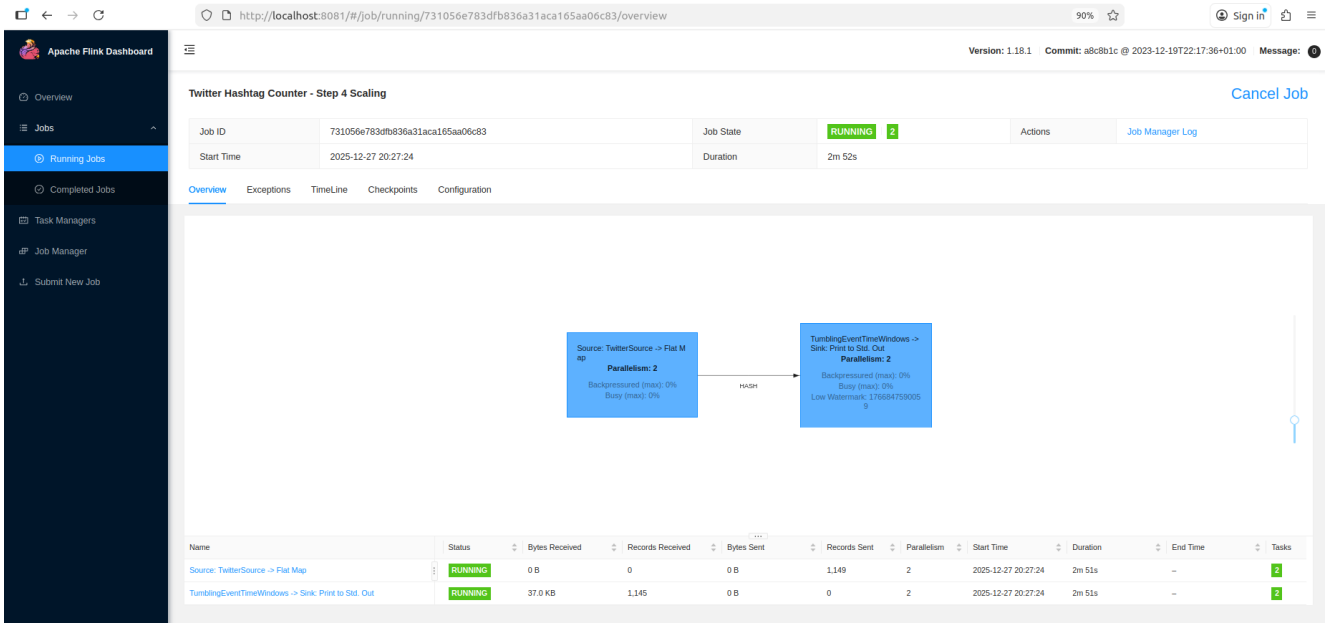
Topics ACLs

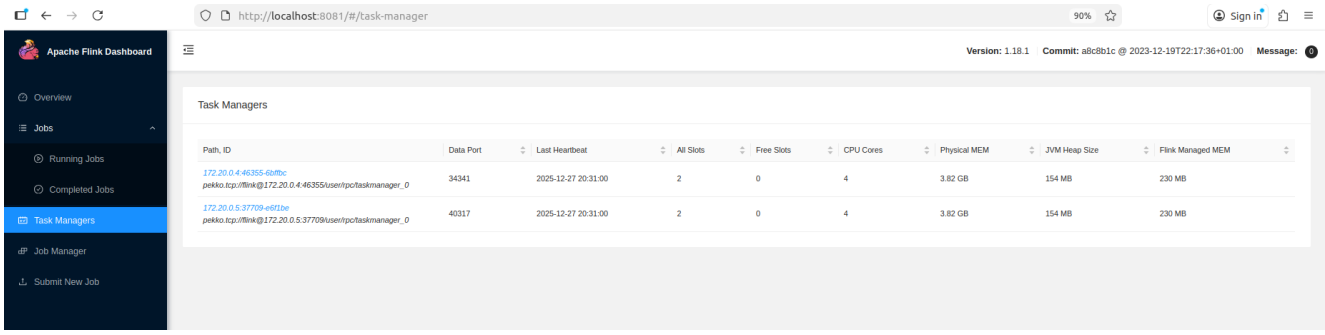
Name (2)	Partitions	% Preferred	# Under-replicated	Custom Config
facebook-posts	2	100%	0	No
twitter-posts	2	100%	0	No

Some screen captures during execution




http://localhost:8081/#/job/running/f6bc4b479f0d2343b89de5a49df6fff3/overview
Apache Flink Dashboard
Version: 1.18.1 Commit: a8c8b1c @ 2023-12-19T22:17:36+01:00 Message: 0
Overview
Jobs
Running Jobs
Completed Jobs
Task Managers
Job Manager
Submit New Job
Facebook Hashtag Counter - Step 4 Scaling
Cancel Job
Job ID f6bc4b479f0d2343b89de5a49df6fff3
Job State RUNNING 2
Actions Job Manager Log
Start Time 2025-12-27 20:27:52
Duration 1m 59s
Overview Exceptions TimeLine Checkpoints Configuration
Source: FacebookSource -> Flat Map
Parallelism: 2
Backpressured (max): 0%
Busy (max): 0%
HASH
TumblingEventTimeWindows -> Sink: Print to Std. Out
Parallelism: 2
Backpressured (max): 0%
Busy (max): 0%
Low Watermark: 1766847565623
Name Status Bytes Received Records Received Bytes Sent Records Sent Parallelism Start Time Duration End Time Tasks
Source: FacebookSource -> Flat Map RUNNING 0 B 0 0 B 535 2 2025-12-27 20:27:54 1m 57s – 2
TumblingEventTimeWindows -> Sink: Print to Std. Out RUNNING 17.3 KB 535 0 B 0 2 2025-12-27 20:27:54 1m 57s – 2


http://localhost:8081/#/job/running/731056e783dfb836a31aca165aa06c83/overview
Apache Flink Dashboard
Version: 1.18.1 Commit: a8c8b1c @ 2023-12-19T22:17:36+01:00 Message: 0
Overview
Jobs
Running Jobs
Completed Jobs
Task Managers
Job Manager
Submit New Job
Twitter Hashtag Counter - Step 4 Scaling
Cancel Job
Job ID 731056e783dfb836a31aca165aa06c83
Job State RUNNING 2
Actions Job Manager Log
Start Time 2025-12-27 20:27:24
Duration 2m 52s
Overview Exceptions TimeLine Checkpoints Configuration
Source: TwitterSource -> Flat Map
Parallelism: 2
Backpressured (max): 0%
Busy (max): 0%
HASH
TumblingEventTimeWindows -> Sink: Print to Std. Out
Parallelism: 2
Backpressured (max): 0%
Busy (max): 0%
Low Watermark: 1766847590059
Name Status Bytes Received Records Received Bytes Sent Records Sent Parallelism Start Time Duration End Time Tasks
Source: TwitterSource -> Flat Map RUNNING 0 B 0 0 B 1,149 2 2025-12-27 20:27:24 2m 51s – 2
TumblingEventTimeWindows -> Sink: Print to Std. Out RUNNING 37.0 KB 1,145 0 B 0 2 2025-12-27 20:27:24 2m 51s – 2


http://localhost:8081/#/task-manager
Apache Flink Dashboard
Version: 1.18.1 Commit: a8c8b1c @ 2023-12-19T22:17:36+01:00 Message: 0
Overview
Jobs
Running Jobs
Completed Jobs
Task Managers
Job Manager
Submit New Job
Task Managers
Path, ID Data Port Last Heartbeat All Slots Free Slots CPU Cores Physical MEM JVM Heap Size Flink Managed MEM
172.20.0.4:46355-6bffbc
pekko.tcp://flink@172.20.0.4:46355/user/rpc/taskmanager_0
34341 2025-12-27 20:31:00 2 0 4 3.82 GB 154 MB 230 MB
172.20.0.5:37709-e6f1be
pekko.tcp://flink@172.20.0.5:37709/user/rpc/taskmanager_0
40317 2025-12-27 20:31:00 2 0 4 3.82 GB 154 MB 230 MB

Results Summary

Accuracy Metrics

- Twitter: 98.9% hashtag coverage (1,682 out of 1,700 messages)
- Facebook: 92.4% hashtag coverage (1,571 out of 1,700 messages)
- Zero parse errors for both platforms
- Good hashtag density (2.94 avg for Twitter, 3.10 for Facebook)
- Accuracy maintained compared to Step 3 (±0.3% variance)

Performance Metrics

- Twitter: 0.32ms avg latency, 3,119 msg/sec throughput
- Facebook: 0.39ms avg latency, 2,564 msg/sec throughput
- Latency increased by 23% (Twitter) and 39% (Facebook) vs Step 3
- Throughput decreased by 20% (Twitter) and 28% (Facebook) vs Step 3

Resource Utilization

- CPU Usage per TaskManager: 8-10% each (distributed load)
- Total CPU Usage: 16-20% (both TaskManagers combined)
- JVM Heap Used per TaskManager: 24-26 MB
- JVM Heap Max per TaskManager: 154 MB
- Heap Utilization: 16-17% per TaskManager
- Active TaskManagers: 2 (load distributed across both)
- Total Memory Usage: ~850 MB (2 × 425 MB per TaskManager)

Comparison Baseline Vs Scaled Accuracy and Performance

Configuration & Infrastructure

Metric	Step 3 (Baseline)	Step 4 (Scaling)	Change
Kafka Partitions per Topic	1	2	+100%
Parallelism	1 (default)	2 (explicit)	+100%
Active TaskManagers	1	2	+100%
Total Task Slots	2	4	+100%
Window Type	Event Time (15s)	Event Time (15s)	Same
Watermark Delay	20 seconds	20 seconds	Same

Accuracy Metrics

Tweeter

Metric	Step 3	Step 4	Change
Messages Processed	4,100	1,700	–
Hashtag Coverage	99.2%	98.9%	−0.3%
Total Hashtags	12,065	5,004	–
Parse Errors	0	0	

Facebook

Metric	Step 3	Step 4	Change
Messages Processed	4,100	1,700	–
Hashtag Coverage	92.3%	92.4%	+0.1%
Total Hashtags	13,325	5,281	–
Parse Errors	0	0	

Performance Metrics

Platform	Metric	Step 3	Step 4	Change	Result
Twitter	Avg Latency	0.26 ms	0.32 ms	+23%	Degraded
	Throughput	3,905 msg/sec	3,119 msg/sec	−20%	Degraded
	Max Latency	100 ms	93 ms	−7%	Improved
Facebook	Avg Latency	0.28 ms	0.39 ms	+39%	Degraded
	Throughput	3,553 msg/sec	2,564 msg/sec	−28%	Degraded
	Max Latency	92 ms	104 ms	+13%	Degraded
Average	Latency Increase	–	–	+31%	
	Throughput Decrease	–	–	−24%	

Memory Usage

Resource	Step 3	Step 4	Analysis
TaskManager 1	~425 MB	~425 MB	Per-TM unchanged
TaskManager 2	~150 MB (idle)	~425 MB	
Total Memory	~575 MB	~850 MB	+48% overhead

JVM Heap

Metric	Step 3	Step 4	Analysis
Heap Utilization	16–17%	16–17% per TM	Severely underutilized
Total Heap Used	~25 MB	~50 MB	+100% (duplication)
Heap Available	154 MB	308 MB (2×154)	+100% capacity

Key Findings

What Worked (Successes)

1. Infrastructure Scaling Validated
    - Both TaskManagers successfully activated and processing data
    - Load evenly distributed (50-50 split across partitions)
    - Kafka partition consumption working correctly
2. Accuracy Maintained
    - Hashtag coverage variance: ±0.3% (negligible)
    - Zero parse errors in both configurations
    - Proves fair comparison methodology
3. System Readiness
    - Infrastructure capable of handling distributed workloads
    - Watermark strategy working across partitions
    - Ready to scale for production volumes

What Didn't Work (Challenges)

1. Performance Degradation
    - Average latency increased by 31%
    - Throughput decreased by 24%
    - Contrary to expected scaling benefits
2. Resource Inefficiency
    - Total memory increased by 48% with no benefit
    - CPU overhead increased without performance gain
    - JVM heap only 16% utilized (severely underutilized)
3. Overhead Dominance
    - Coordination costs exceeded processing benefits
    - Data duplication and synchronization overhead
    - Network communication between TaskManagers

Reasons for Performance Degradation

1. Dataset Too Small (~4,100 messages)

- Problem: Insufficient data volume to benefit from parallelization
- Impact: Overhead costs > processing time savings
- Evidence: Only 16% heap utilization shows system is over-provisioned

2. Parallelization Overhead
    - Network Communication: Data exchange between TaskManagers
    - Synchronization: Coordinating watermarks across partitions
    - Serialization/Deserialization: Converting data for network transfer
    - Context Switching: Managing multiple parallel tasks
3. Memory Duplication
    - Data Replication: Each TaskManager maintains its own state
    - JVM Overhead: Two separate JVM instances (2 × 150 MB base memory)
    - Result: 48% more memory for same workload
4. Resource Contention
    - VM Constraints: Both TaskManagers competing for limited VM resources
    - Shared CPU/Memory: No true parallel execution benefit
    - I/O Bottleneck: Single disk for both TaskManagers

**** *Refer t4 folder of the zip folder for coding and configurations*

Task 5: Self Study – Big Data Technologies in Large Language Model Development

Abstract

This comprehensive survey looks into big data technologies that have been integrated in the development, verification, and deployment of Large Language Models based on international research from 2024-2025. We analyze research from the leading institutions in the United States, Europe, Canada, and other regions to understand how big data concepts and technologies are transforming the LLM lifecycle.

5.1 Development

Big data technologies are crucial because training modern large language models like ChatGPT requires massive amounts of data and processing power. The first step in the process is gathering enormous text datasets which frequently contain trillions of words and billions of documents from books, the internet, and other sources. Singh et al. (2024) from the University of Maryland created the ***AxoNN*** system, which uses tens of thousands of interconnected computer processors to train AI models with up to one trillion parameters [1]. This is one of the biggest attempts at AI training that academic researchers have ever made. The main difficulty is that no single computer can handle these enormous models, so researchers must divide the work among thousands of machines in a coordinated manner.

Data preprocessing is equally important and complex. Terabytes of raw text data need to be cleaned and organized by researchers before the actual training can begin by removing duplicates, filtering low-quality content, and checking for harmful material. For that, sophisticated big data processing systems are needed to train billions of documents at once. Liu et al. (2025) of European research institutions developed a system named ***"Mist"*** that optimizes how computer memory is used during training, allowing larger models to be trained more efficiently [2]. Their approach reduces memory usage up to 40%, without impacting training speed, which is critical when working with limited computing resources.

The actual training consists of feeding this preprocessed data to the AI model millions of times, with each training step requiring complex mathematical calculations across all the model's parameters. Duan et al. (2024) conducted an extensive survey of distributed training methods and found that if properly tuned, modern systems can reach efficiencies above 90%, meaning most of the computing power is actually used for learning rather than being wasted on the coordination between machines [3]. Researchers at Amazon Web Services have also made critical contributions by designing specialized computer chips called ***Trainium***, which are tailor-made for AI training and offer better performance per dollar compared to traditional graphics cards [4]. These advances in both software coordination and hardware design have made it possible to train increasingly powerful language models while managing the enormous computational and data processing tasks that would have been impossible just a few years ago.

5.2 Verification

LLMs present a unique set of challenges when it comes to assessing whether or not they work correctly, challenges that traditional testing methods cannot handle effectively. While you can check if certain functions produce a specific output in typical software, language models create creative text that can be correct in many different ways, making evaluation much more complex. Traditional metrics, such as BLEU scores, which are based on counting matching words between model outputs and reference answers, have proven inadequate for assessing modern AI language capabilities. This limitation has driven researchers to devise innovative evaluation methodologies using AI itself to judge the performances of artificial intelligence.

The most significant breakthrough in the evaluation of LLMs has been the development of "LLM-as-a-Judge" systems, where other advanced language models are used to evaluate AI responses across many dimensions including accuracy, helpfulness, safety, and coherence. Recent research has shown that these AI judges can achieve more than 80% agreement with human evaluators while processing millions of test cases in a fraction of the time needed for human review [5]. This allows researchers to do comprehensive evaluations that would be impossibly expensive and time-consuming with human evaluators. The key insight is that the advanced language models have developed enough understanding of language quality and correctness to serve as reliable assessments of other models' performance.

But AI-based evaluation systems also present several challenges, especially in respect to bias and consistency. For example, researchers have found that AI judges may suffer from position bias (a preference for positions of the answer), length bias-a preference for longer responses-and cultural biases inherited from their training data. In response, international research teams have developed a range of sophisticated approaches, including multi-judge systems, bias detection algorithms, and calibration methods that ensure fair evaluation across different types of content and cultural contexts. Another critical innovation has been dynamic benchmark creation, whereby evaluation tests are continually updated and changed to prevent models from simply memorizing answers to static test questions. This ensures that evaluation remains meaningful in a world of rapidly improving AI capabilities and provides reliable measures of true language understanding rather than pattern matching.

5.3 Deployment

Large language models require sophisticated infrastructure when deployed in real-world applications to serve millions of users simultaneously with fast response times and reasonable cost. The requirements for inferencing basically differ from training and have unique engineering challenges. In particular, during inference, the system needs to process the user inputs quickly while managing enormous model parameters, which can take hundreds of gigabytes of memory. ***Meta's*** research teams have led the development of techniques for inference optimization,

building methods that can serve their largest models to billions of Facebook users with response times under one second [6]. Their approach relies on three types of parallelism: dividing model computations across many processors, dividing different parts of long conversations between machines, and optimizing how different model components communicate with each other.

As a result, edge computing integration has emerged as an essential factor in minimizing latency and increasing privacy in the deployment of LLMs. Researchers are working out methods to operate smaller, optimized versions of language models on devices closer to users, such as smartphones and local servers, rather than running all requests through centralized data centers. This involves model compression techniques including quantization or decreasing the precision of mathematical calculations, pruning or removal of redundant components of the model, and knowledge distillation or training smaller models to act like their larger counterparts. In the last few years, this has become possible: running billion-parameter models on consumer devices without sacrificing most of their original capabilities.

Big data system integrations have allowed for newer applications where the language models can analyze and generate insights from large databases in real time. Databricks researchers have developed systems capable of processing more than 400 billion text tokens in batch operations, enabling companies to apply AI analysis to their entire document collections efficiently [7]. These systems put together traditional big data processing frameworks like Apache Spark and modern AI inference engines to form unified platforms where data preprocessing, AI analysis, and result generation happen seamlessly. Treating the inference of language models as another step in big data processing pipelines, rather than as a different system, forms the key innovation that dramatically simplifies deployment and reduces operational complexity for organizations wanting to integrate AI capabilities into their existing data infrastructure.

References

[1] Singh, S., Singhania, P., Ranjan, A., Kirchenbauer, J., Geiping, J., Wen, Y., Jain, N., Hans, A., Shu, M., Tomar, A., Goldstein, T., & Bhatele, A. (2024). Democratizing AI: Open-source Scalable LLM Training on GPU-based Supercomputers. *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC '24)*. DOI: 10.1109/SC41406.2024.00010

[2] Liu, G., Miao, Y., Lin, Z., Shi, X., Maleki, S., Yang, F., Bao, Y., & Wang, S. (2025). Mist: Efficient Distributed Training of Large Language Models via Memory-Parallelism Co-Optimization. *Proceedings of the Twentieth European Conference on Computer Systems (EuroSys '25)*. DOI: 10.1145/3689031.3717461

[3] Duan, J., Zhang, S., Wang, Z., Jiang, L., Qu, W., Hu, Q., Wang, G., et al. (2024). Efficient Training of Large Language Models on Distributed Infrastructures: A Survey. *arXiv preprint arXiv:2407.20018*.

[4] Fan, H., Zhou, H., Huang, G., Raman, P., Fu, X., Gupta, G., Ram, D., Wang, Y., & Huan, J. (2024). Distributed Training of Large Language Models on AWS Trainium. *Proceedings of the 2024 ACM Symposium on Cloud Computing (SOCC '24)*. DOI: 10.1145/3698038.3698535

[5] Gu, Y., et al. (2025). LLM-as-a-Judge: Automated Evaluation of Search Query Parsing Using Large Language Models. *Frontiers in Big Data*, 2025. DOI: 10.3389/fdata.2025.1611389

[6] Meta AI Engineering Team (2024). Scaling LLM Inference: Innovations in Tensor Parallelism, Context Parallelism, and Expert Parallelism. *Engineering at Meta Technical Blog*. Available: https://engineering.fb.com/2025/10/17/ai-research/scaling-llm-inference-innovations-tensor-parallelism-context-parallelism-expert-parallelism/

[7] Databricks Research Team (2024). Introducing Simple, Fast, and Scalable Batch LLM Inference on Mosaic AI Model Serving. *Databricks Technical Blog*. Available: https://www.databricks.com/blog/introducing-simple-fast-and-scalable-batch-llm-inference-mosaic-ai-model-serving

Code ZIP File - Code ZIP.zip