COURSE WORK: BIG DATA ANALYTICS TECHNOLOGY

MSc Data Science: Coventry University UK (**2024.2 BATCH**)

Name: P D Lolitha Lakshan Weerasinghe

Index Number: COMScDS242P-003

Table of Contents

Source Code

- GitHub Repo : https://github.com/lolitha-lakshan-1/bigdata-assignment
- NIBM One Drive : https://nibm-my.sharepoint.com/:u:/g/personal/comscds242p-003_student_nibm_lk/IQDsD-L-e362Q6pPy5K0RvwMAQl4TpUnvFGoYv25MMCf-UM?e=whBcDx

Higer Quality Report in Doc Format : https://github.com/lolitha-lakshan-1/bigdata-assignment/blob/main/COMScDS242P-003_Report.docx

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

Part 1: Implementation and Performance Comparison

Hadoop Cluster Implementation Overview

The Hadoop cluster was implemented using Docker which consists of NameNode, DataNode and YARN services. Resource configuration was done using the configuration files in the hadoop_config directory which was mounted into all the containers; therefore all Hadoop daemons and jobs used the same configuration. The core-site.xml was used to set the filesystem to hdfs://namenode:9000 and the temp directory to /opt/hadoop/tmp and hdfs-site.xml was used to set the HDFS replication factor to dfs.replication = 1, which matches a single-DataNode setup. The yarn-site.xml was used to point YARN to the limit the ResourceManager and NodeManager resources to 1024 MB and container allocations between 128 MB and 1024 MB and 1 cpu core.

Hadoop Job Implementation

The Hadoop implementation uses two MapReduce jobs to summarize the SNAP edge-list data. The InDegreeJob1Hadoop job reads each edge, skips commented lines and counts the number of times each node appears as destination to get that node's in-degree. The InDegreeJob2Hadoop job groups those in-degree results to count how many nodes have each degree value, producing a (degree, count) distribution. The InDegreeMainHadoop runs both jobs in order and combines the final output into one CSV for reporting.

Source: https://github.com/lolitha-lakshan-1/bigdata-assignment/tree/main/Task_1/hadoop

Hadoop Performance Tracking

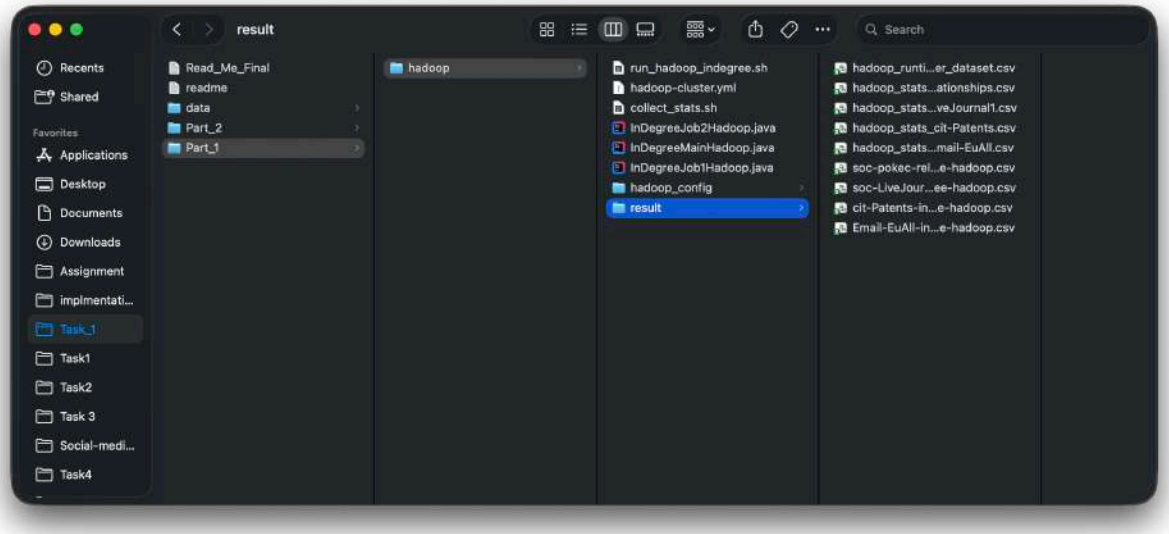
Job performance for each dataset is tracked using a bash script (run_hadoop_indegree.sh) which executes the hadoop job per dataset. It logs the job duration and resource usage. For

each dataset, the script first stage the dataset file from DATASET_ROOT into the NameNode and uploaded it to HDFS under /input/<dataset>-hadoop.txt. Then it measures the end-to-end execution time by capturing Unix start/end timestamps around the actual job command and duration is appended to the hadoop_runtime_per_dataset.csv file.

In parallel, the CPU, memory usage/percentage, network I/O, disk I/O, and PID metrics are written into a per dataset csv with dataset name hadoop_stats_<dataset>.csv. This was done using the executing docker stats every 5 seconds for containers. The charts are built using stats from these csv files.

Source : https://github.com/lolitha-lakshan-1/bigdata-assignment/blob/main/Task_1/hadoop/run_hadoop_indegree.sh

Result : https://github.com/lolitha-lakshan-1/bigdata-assignment/tree/main/Task_1/hadoop/results/PreOptimization



Spark Cluster Implementation Overview

The Spark cluster was implemented using Docker, defined in spark-cluster.yml, which consists of a spark-master and a spark-worker service. The master container starts the Spark Master process, exposes port 7077 for the cluster URL and the worker container starts a Spark Worker process and connects back to the master using port 7077. Both containers

mount to the same host dataset directory "/data", so the Spark jobs can access the input files across the cluster.

The run_spark_indegree.sh script compiles InDegreeSpark.java, packages it into spark-indegree.jar, copies the JAR into the master container under /tmp/indegree, and then runs the job using spark-submit on the master with --master spark://spark-master:7077 and spark.default.parallelism=4, passing the dataset name as the input parameter.

Source : https://github.com/lolitha-lakshan-1/bigdata-assignment/blob/main/Task_1/spark/spark-cluster.yml

Spark Job Implementation

The Spark implementation uses a single Spark job to process the dataset, and it is built using the RDD-based Spark API. Intermediate results are represented as key–value tuples using JavaPairRDD. The job reads the input file where each line represents a directed edge as a pair of node IDs. The source node and the destination node. It skips empty lines and comment lines that start with #, then extracts the destination node from each edge and emits a tuple (destinationNode, 1). These tuples are aggregated to calculate the in-degree of each node. Thereafter, the job converts those results into (inDegreeValue, 1) tuples and aggregates again to count how many nodes share the same in-degree, producing a sorted degree distribution. The final output is written as a single CSV file in the format degree count for reporting.

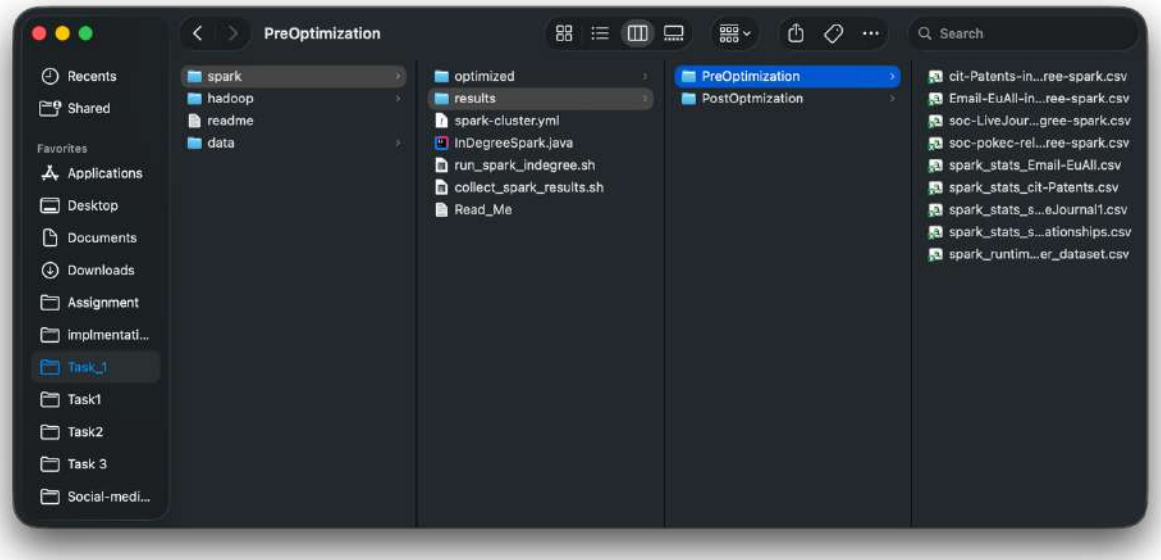
Source: https://github.com/lolitha-lakshan-1/bigdata-assignment/blob/main/Task_1/spark/InDegreeSpark.java

Spark Performance Tracking

Job performance for each dataset is tracked using the shell script run_spark_indegree.sh which executes the Spark job per dataset and logs both duration and resource utilization. For each dataset discovered under DATASET_ROOT, the script runs the Spark application using spark-submit on the cluster master and measures duration time by capturing Unix

start and end timestamps around the spark-submit command then the duration is appended to spark_runtime_per_dataset.csv.

In parallel, the script starts a background loop that executes every 5 seconds using docker stats command and writes CPU, memory usage/percentage, network I/O, disk I/O, and PID metrics into a per-dataset CSV named spark_stats_<dataset>.csv



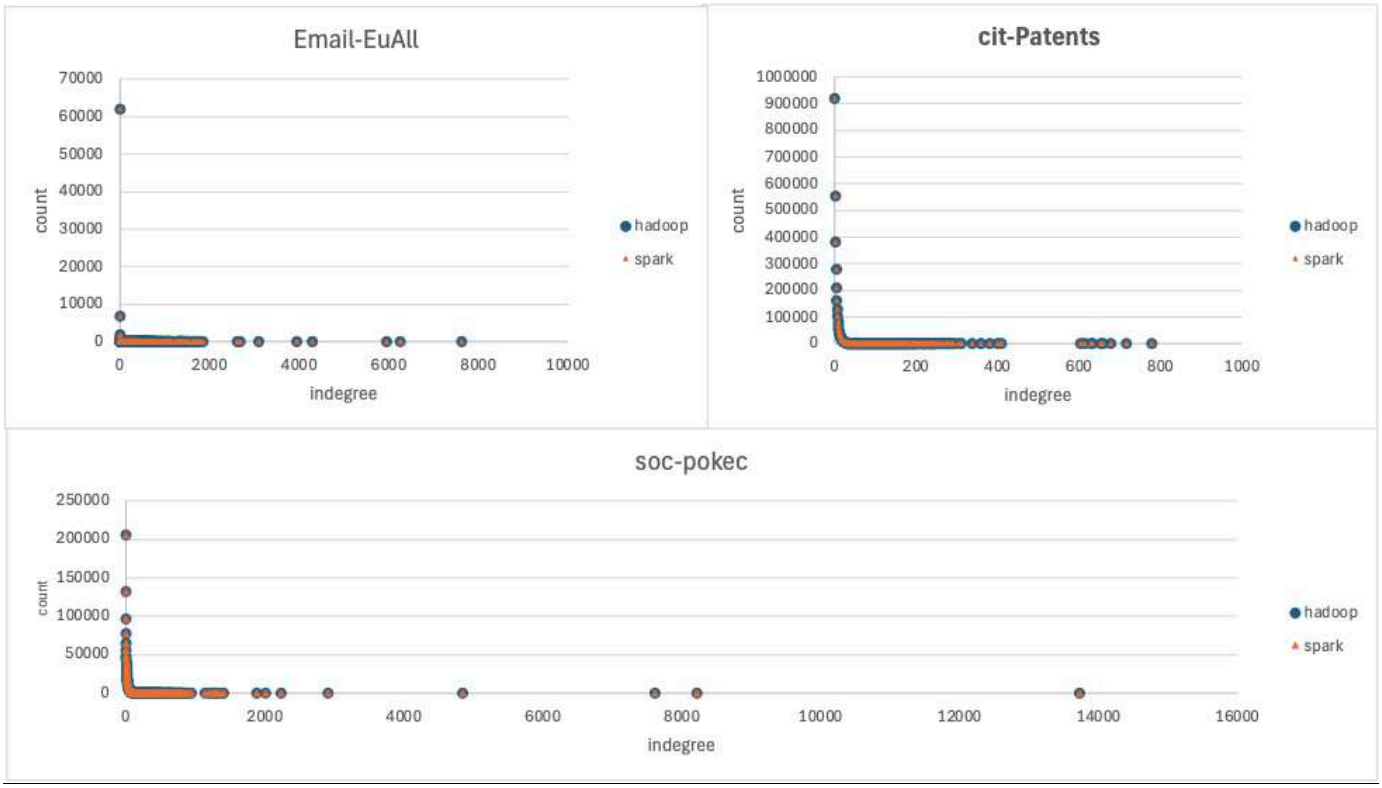
Test Bench

MacBook Air M1 with 8GB RAM and 8-core CPU

In-degree distribution plots

Email-EuAll , cit-Patents and soc-pokec-relationships datasets were executed on the Spark and Hadoop clusters. Their sizes are respectively 5MB, 280MB and 423MB.

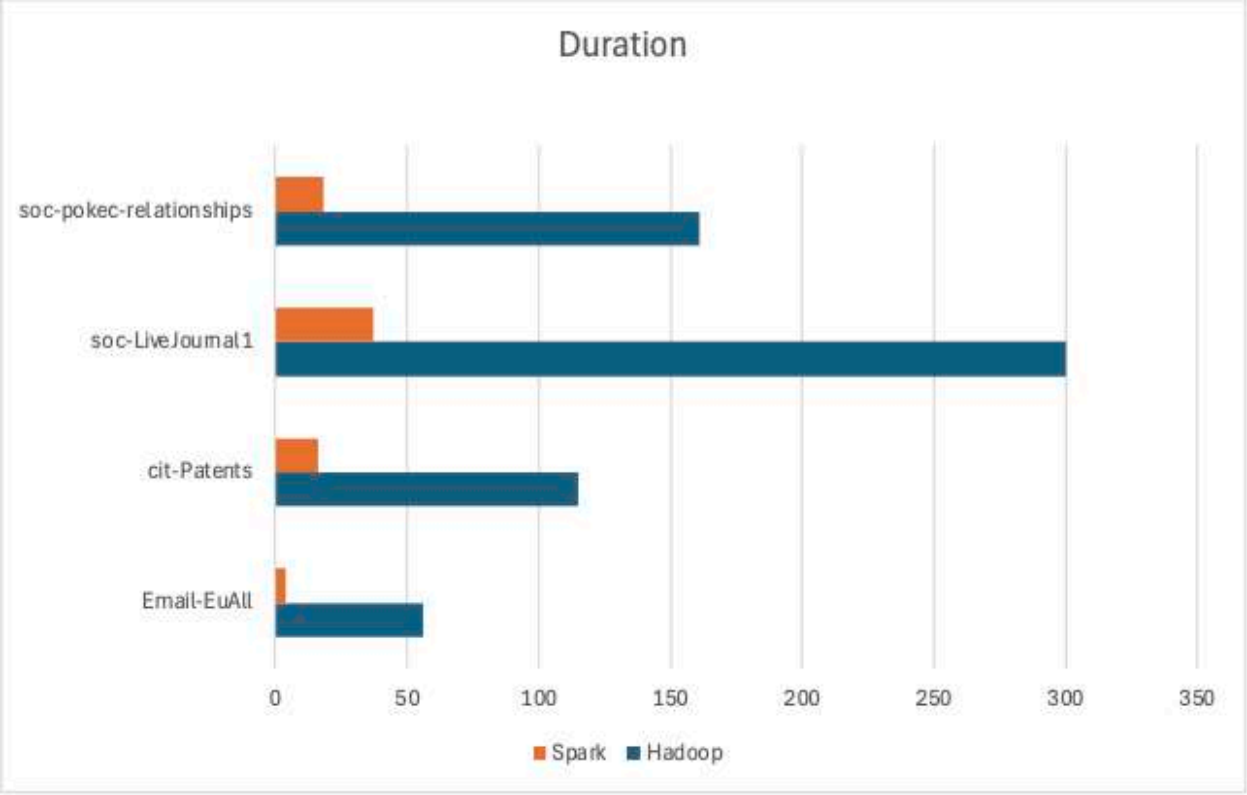
According to the below plots, Hadoop and Spark plots perfectly overlaps for each other for the datasets which confirms that **results are same for both execution methods.**



Performance Metrics [Includes soc-LiveJournal1 dataset]

Excel Sheets With Raw Data : https://github.com/lolitha-lakshan-1/bigdata-assignment/tree/main/Task_1/comparison/pre_optimization

Execution Duration



Dataset	Size	Hadoop Time(s)	Spark Time(s)	Speedup (Hadoop/Spark)
Email-EuAll	5 MB	56	4	14.00x
cit-Patents	280 MB	115	16	7.19x
soc-pokec-relationships	423 MB	161	18	8.94x
soc-LiveJournal1	1.08 GB	300	37	8.11x

- Spark is 7.19x to 14x faster across datasets than Hadoop.
- On the smallest dataset (Email-EuAll, 5MB), Hadoop takes 56 seconds. In contrast, it takes only 115 seconds to process a file 56 times larger (280MB). This indicates that Hadoop has a very high startup cost.

- Spark's processing time for the largest dataset (1.08GB) is only 37 seconds, which is still faster than Hadoop's time for the smallest dataset 56 seconds.

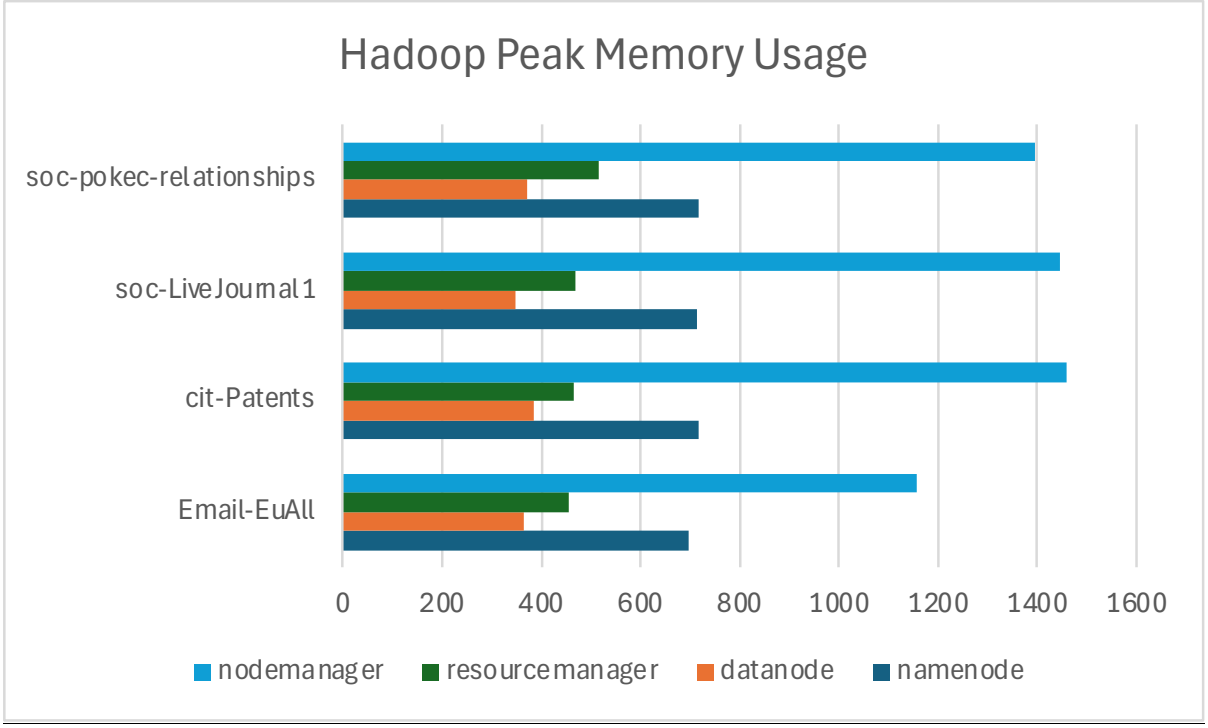
Memory Usage

Peak Memory Usage

It was found out Hadoop components (NameNode, DataNode, ResourceManager) show very little variation in peak memory usage regardless of the dataset size. For example, the NameNode peak only changes by ~20 megabytes between a 5MB and a 1GB file. This suggests Hadoop pre-allocates or maintains a high baseline memory.

Hadoop Peak Memory Usage (MiB)

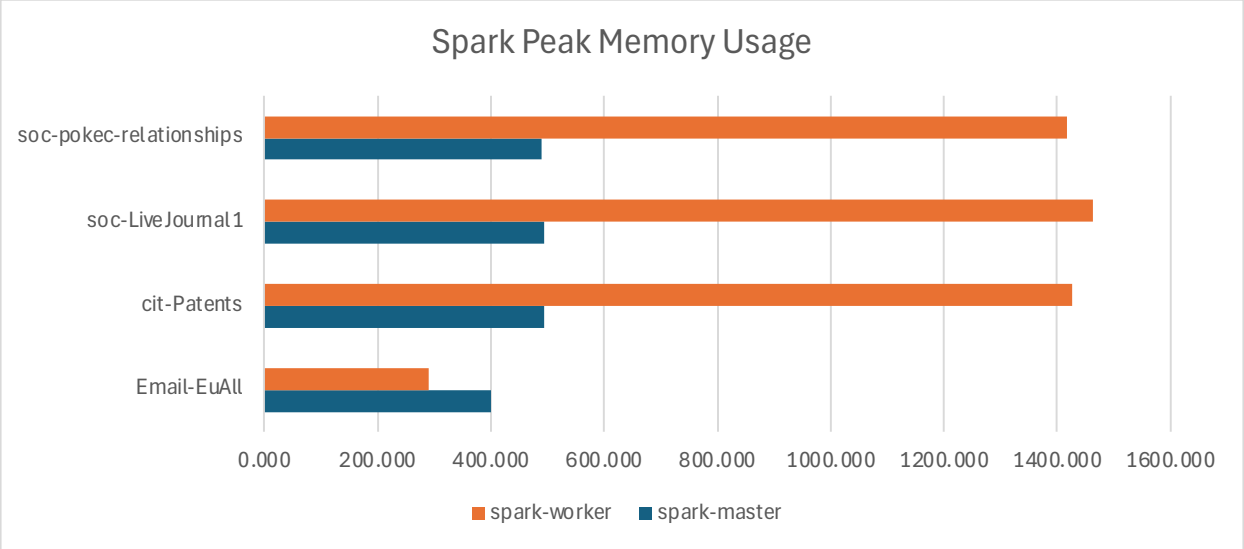
Dataset	namenode	datanode	resourcemanager	nodemanager
Email-EuAll	698.1	364.3	455.9	1158.144
cit-Patents	718.9	383.8	464.9	1458.176
soc-LiveJournal1	715.4	348.5	470.2	1445.888
soc-pokec-relationships	717.9	371.1	516.2	1395.712



- The nodemanager is consistently the largest consumer of memory. It accounts for nearly half of the total peak memory usage for every dataset.
- The namenode and resourcemanager memory peak remains stable regardless of the dataset size.
- The datanode uses the least amount of memory in the cluster, maintaining a steady range between 348.5 and 383.8 megabytes.
- cit-Patents and soc-pokec-relationships show the highest total peak memory usage, largely driven by higher NodeManager requirements.Email-EuAll (the smallest dataset) has the lowest overall memory footprint.

Spark Peak Memory Usage (MiB)

Dataset	spark-master	spark-worker
Email-EuAll	399.800	291.100
cit-Patents	494.000	1427.456
soc-LiveJournal1	494.700	1462.272
soc-pokec-relationships	489.700	1416.192

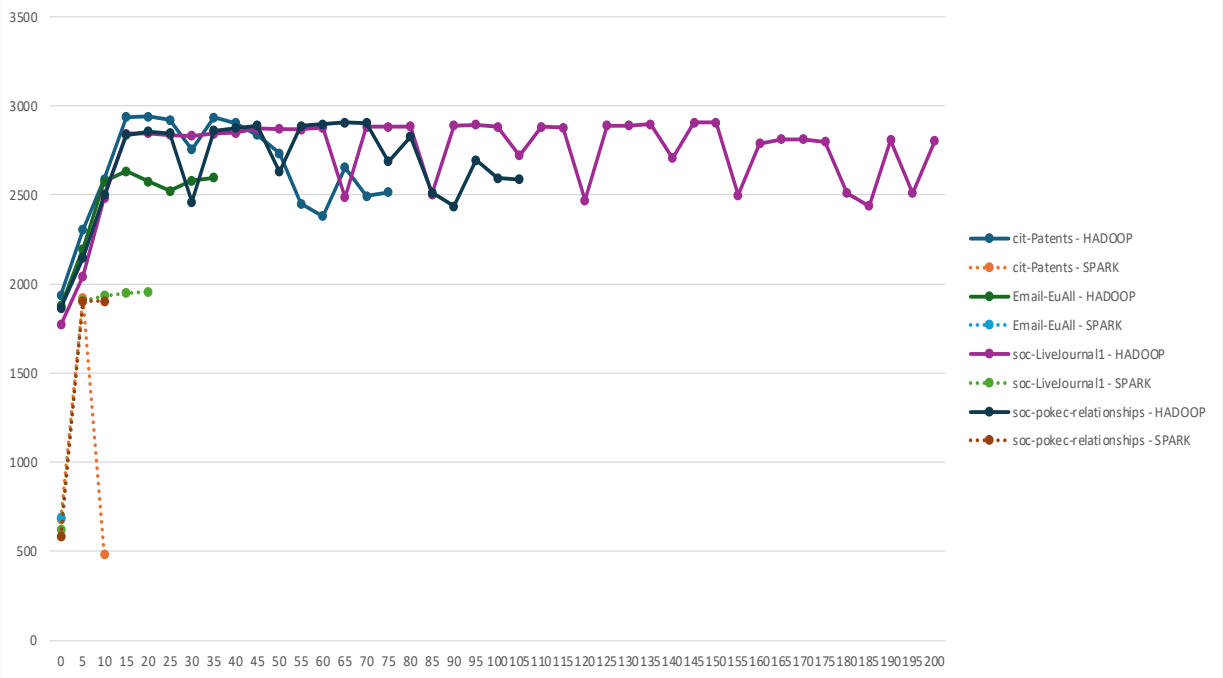


- Spark shows a unique behavior with the smallest dataset, Email-EuAll where spark-master (399.800) uses more memory than the spark-worker (291.100). Suggests

that the administrative overhead of the master node exceeds the actual resources needed for the computation.

- Regardless of whether the dataset is 280MB (cit-Patents) or over 1GB (soc-LiveJournal1), the master stays within a narrow range of 489.7 to 494.7 Mibs.
- The worker memory usage scales sharply as the dataset transitions from small to medium sizes, then plateaus.M emory usage jumps from 291.100 (Email-EuAll) to 1427.456 (cit-Patents).
- However, Spark is much more sensitive to data size. The Spark-Worker peak memory usage jumps from 291.1 for the 5MB file to 1462.2 for the 1GB file.
- Therefore, Spark's memory footprint is small for small tasks and scales up to meet large tasks.

Hadoop vs Spark Memory Usage



For aggregate total of memory usage by all nodes, the chart indicates that Hadoop jobs consumers about 2.5GB - 3GB of memory consistently and the jobs executes for a longer period as well. Meantime Spark jobs peak around 2GB of memory and finish the execution instantly. Therefore, Hadoop maintains a heavy footprint even for the smallest tasks as

depicted above. Furthermore, the solid line for Hadoop depicts a "sawtooth" pattern, where memory fluctuating between 2.5 and 3.0 GB. This indicates shuffle and sort phases of MapReduce. The Spark jobs shoot straight up and straight down. This confirms Spark is loading the graph into memory, processing it instantly.

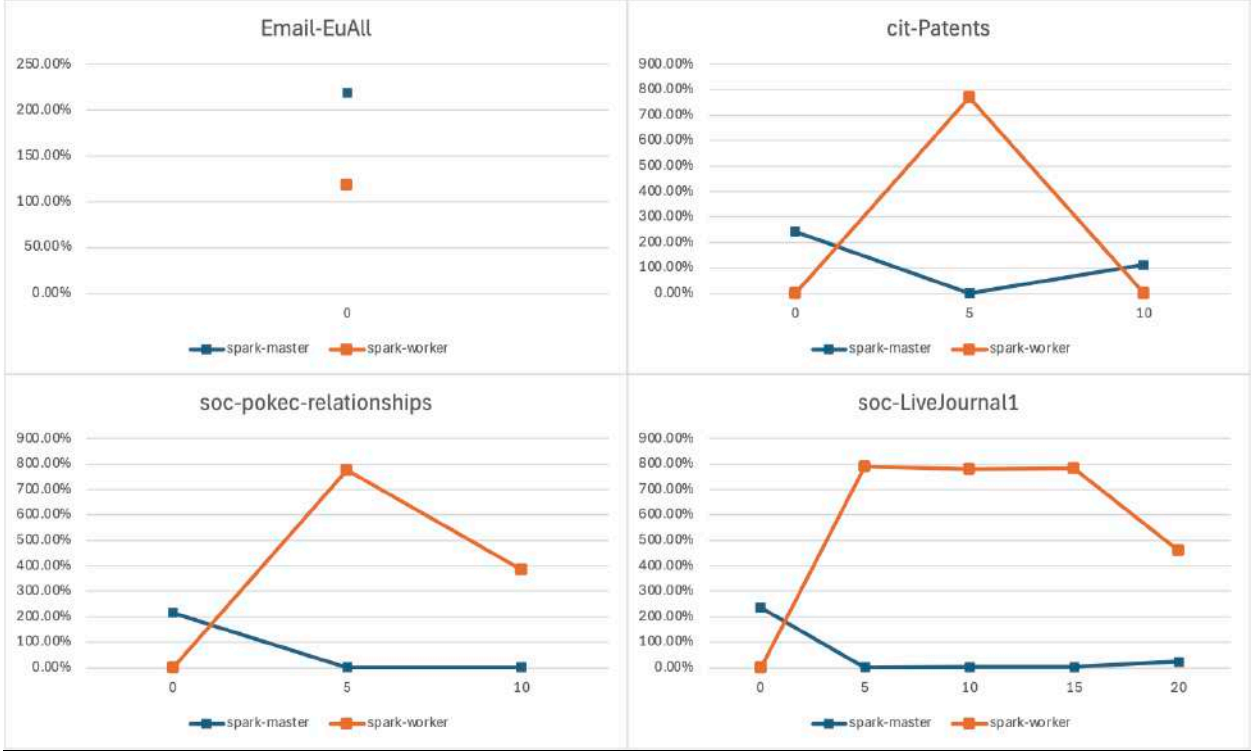
CPU utilization

Hadoop CPU Utilization

The values are aggregate across multiple CPU cores (e.g., 200% = 2 full cores utilizing 100% capacity).

- The NodeManager is responsible for launching and managing the actual containers where Map and Reduce tasks run. The high CPU usage confirms that the computation is happening there. Therefore, the green line dominates every chart, staying between 100% and 200%. Also, the "sawtooth" pattern indicates the map reduce steps.
- NameNode (Orange Line) there is a sharp spike at the begining reaching ~150-200% which immediately drops to near 0%.
- DataNode & ResourceManager (Blue & Light Blue Lines) hover near 0% for the entire duration of the job.
- The chart proves that Hadoop's CPU load is almost entirely on the NodeManagers.
- It seems Hadoop has bottle neck of CPU utilization, only utilizing 2 cores of the available 8 cores. This could be a configuration issue related to docker.

Spark CPU Utilization

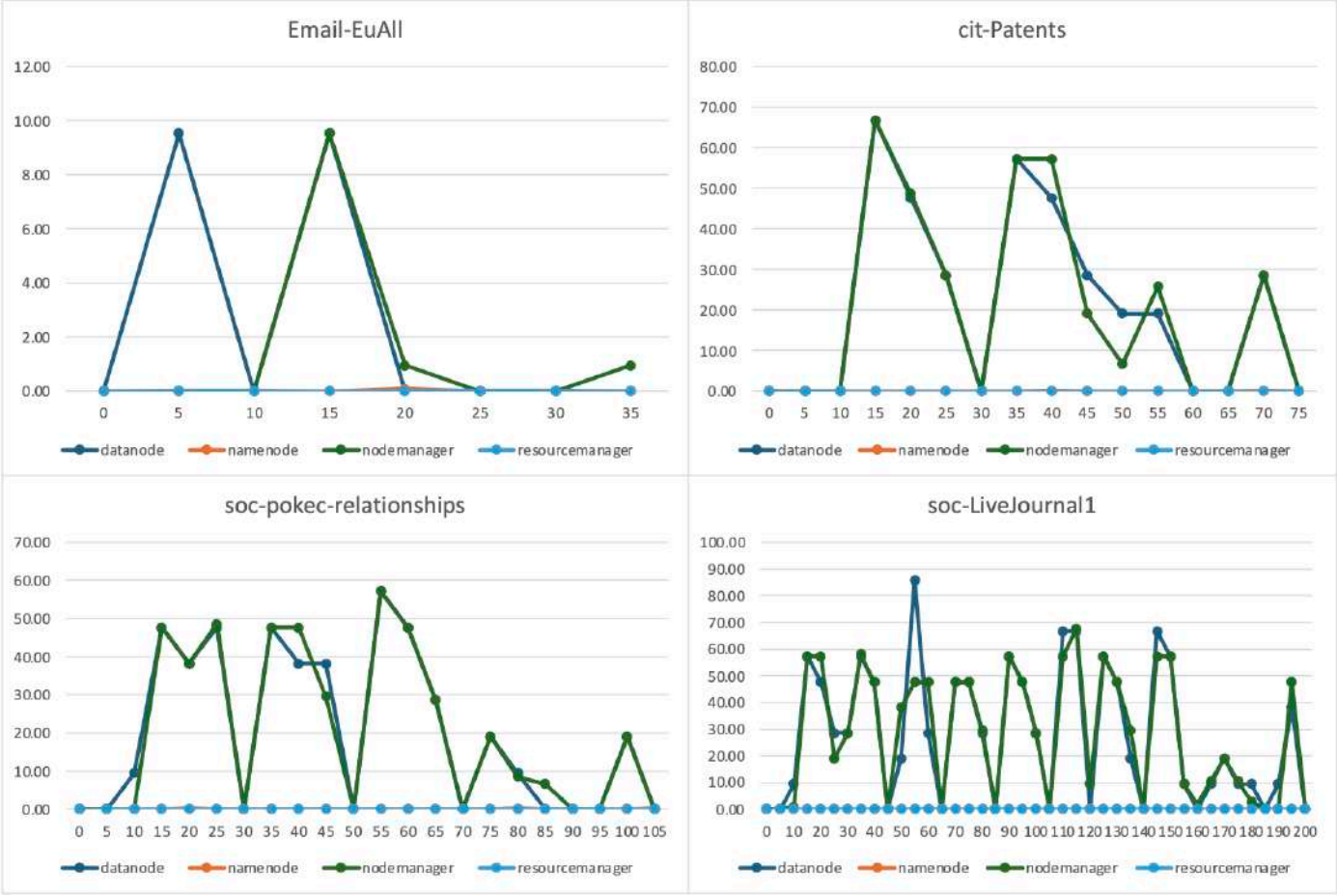


- Unlike Hadoop, which hovered around 100-200% CPU usage, the Spark Worker (Orange) skyrockets to 800% utilization almost immediately for the three larger datasets (cit-Patents, soc-pokec, and soc-LiveJournal1) and 800% peak suggests the worker is fully utilizing the 8-cores in the machine CPU.
- Email-EuAll dataset is so small that the job finishes instantly utilizing only 2 cores.
- cit-Patents & soc-pokec both show a "triangle" shape. The Worker hits 800% at T=5 seconds but is already ramping down by T=10.
- soc-LiveJournal1 this is the only dataset that sustains the load. The Worker hits 800% and stays there for roughly 10-15 seconds. This confirms it is the most computationally intensive job.

Network Utilization

The network utilization values were derived from the net_io field, which is recorded as received / sent traffic for each container. First the received side and the sent side were converted into MiB so everything is in one unit. Then a total network value per row was formed as net_total_mib = net_rx_mib + net_tx_mib. Because Docker's net_io is a cumulative counter, job overhead was measured as the change per 5-second interval for each dataset and each node, the overhead at time t is net_total_mib_delta(t) = net_total_mib(t) − net_total_mib(t−5).

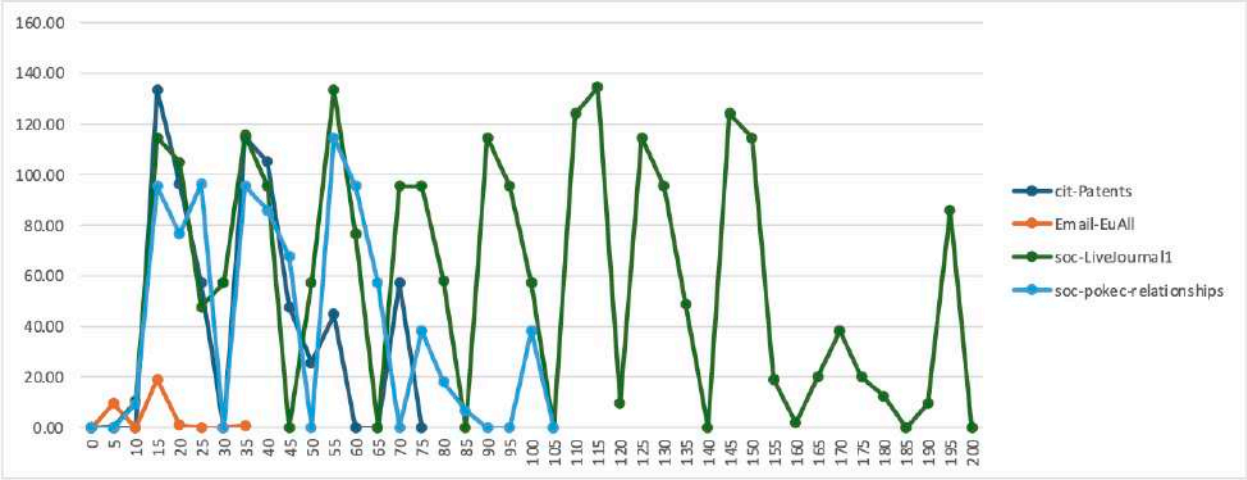
Hadoop Network Utilization

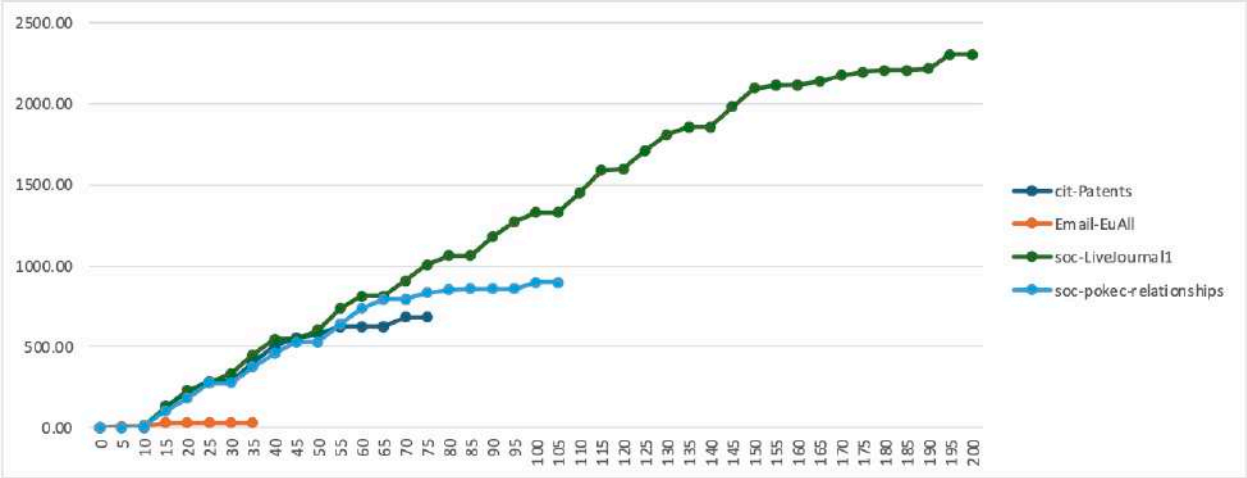


- In all four graphs the DataNode (Blue) and NodeManager (Green) lines move almost perfectly in sync. This confirms that network traffic in Hadoop is driven by data movement between workers. Also, there are sharp peaks followed by drops to near zero which indicates bursty behavior. This would represent the shuffle phase of MapReduce. And the NameNode (Orange) and ResourceManager (Light Blue) remain flat at 0.00 for almost the entire duration across jobs.
- In the smallest job Email-EuAll, the data transfer happens in two quick bursts.
- cit-Patents shows a spike of activity between T=10 and T=60. This indicates a heavy shuffle phase where a large portion of the job execution time was spent just moving data between nodes.

- The soc-pokec dataset shows a series of repeated, medium-intensity spikes. This indicates a constant, rhythmic network shuffling throughout the job.
- soc-LiveJournal1 as the largest dataset, this shows lot of activity. The network is busy constantly from T=15 to T=200, with repeated spikes hitting 60-80 MiB. This suggests the job was network-bound for significant periods, constantly shuffling data back and forth.

Hadoop Throughput & Cumulative Network IO Usage

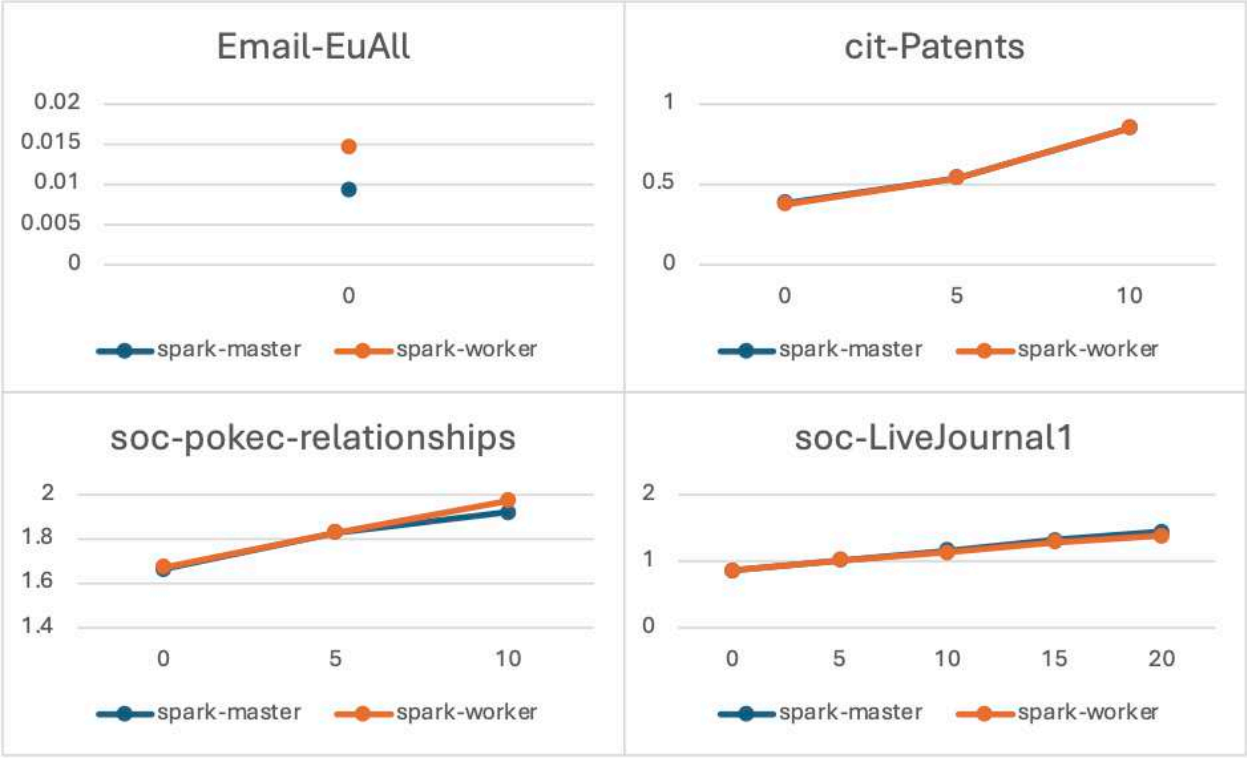




The total Network IO profiles demonstrate a direct correlation between data transfer volume and graph edge count. soc-LiveJournal1 (69 million edges) depicts the most extensive network footprint, sustaining a volatile throughput between 10–140 MiB for nearly 200 seconds. The mid-tier datasets such as soc-pokec-relationships (30.6 million edges)

displays a multi-phase pattern with peaks around 115 MiB, while cit-Patents (16.5 million edges) is characterized by a singular, intense initial burst reaching ~135 MiB before dropping off. In contrast, the minimal Email-EuAll (0.42 million edges) requires negligible bandwidth, peaking at only ~20 MiB, confirming that network activity scales aggressively with the volume of edges requiring shuffle.

Spark Network Utilization



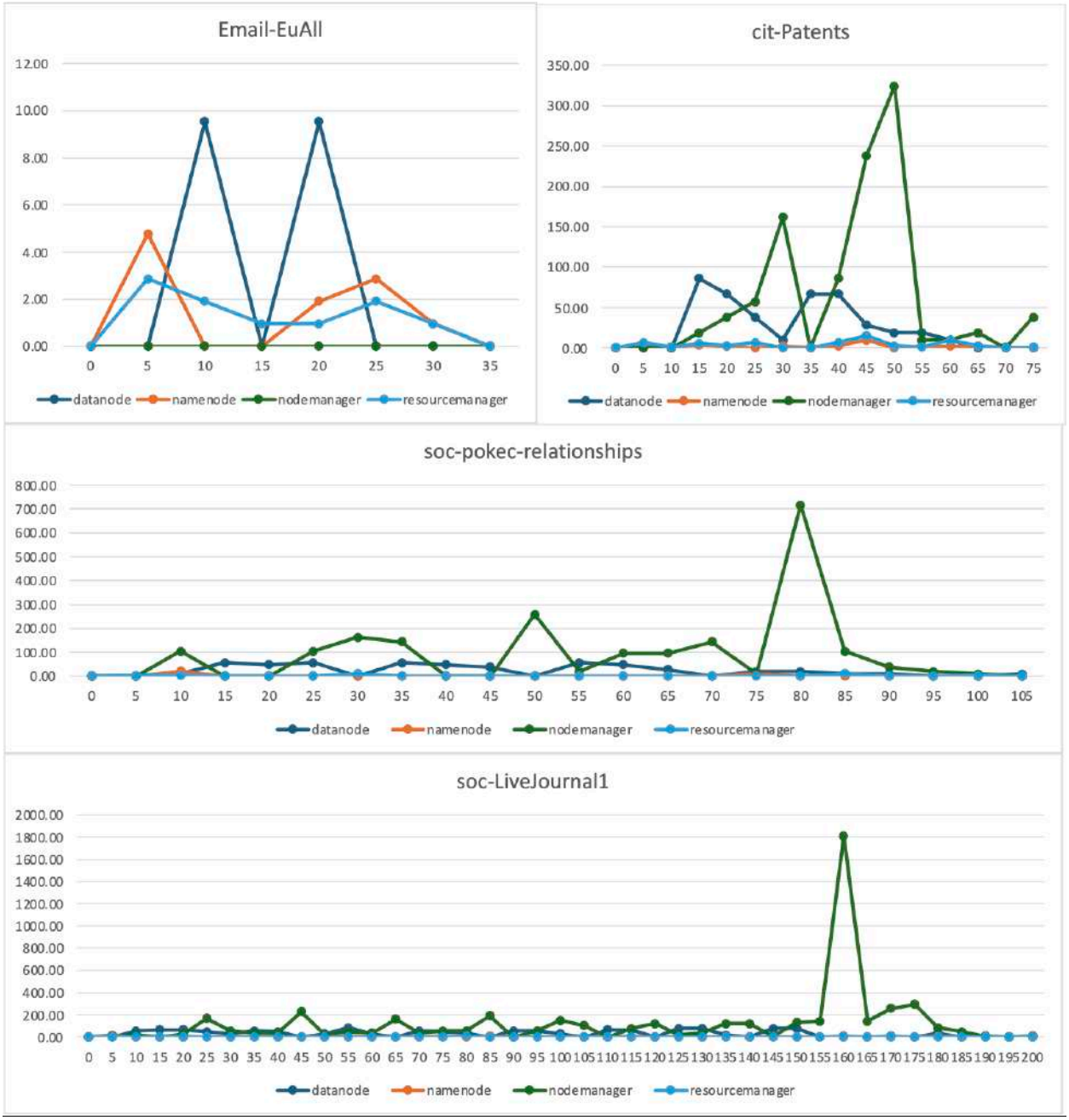
- According to the graphs the Worker nodes shows higher network activity in every dataset than the master nodes. Since Spark Executors resides in Worker nodes, this is as expected as Master nodes act as the cluster manger in a control plane.
- Email-EuAll & cit-Patents shows the lowest network footprint. This Spark job fits mostly in memory, and therefore the data exchange is minimal because the relationships in these datasets are less dense.
- soc-LiveJournal1 & soc-pokec-relationships depicts a jump in Worker network traffic here. Because these are larger social network graphs with high connectivity, Spark must move data between workers.

Hadoop vs Spark Network IO

Based on the findings, there is a massive disparity in network utilization with Hadoop as it generates heavy traffic in the hundreds of MiB, Spark operates almost entirely in the kB range, peaking at only ~1.9 MiB for the same tasks. This ~400x reduction occurs because Hadoop's architecture requires constant data movement between the NodeManager and DataNode for storage and shuffling, whereas Spark processes data in-memory, bypassing the network interface for most of the computation.

Disk Utilization

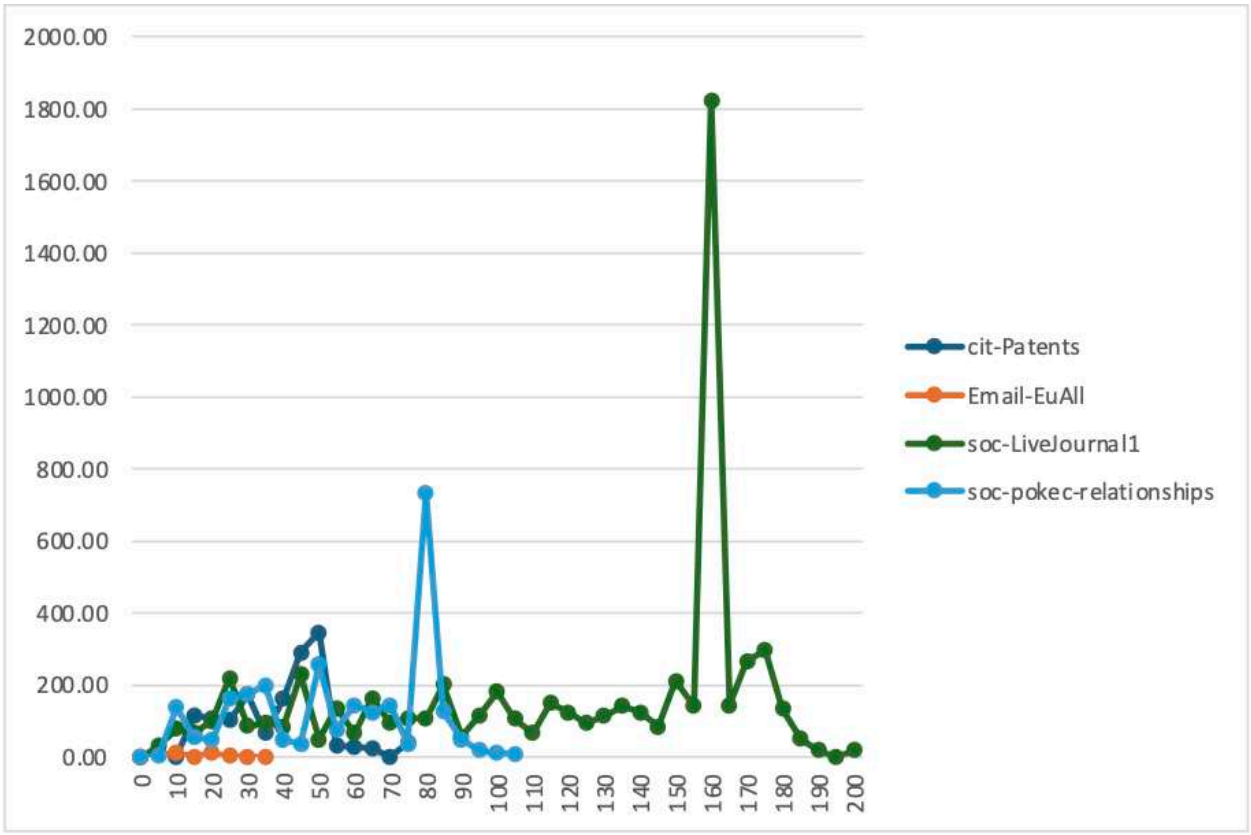
Hadoop Disk Utilization



NameNode and ResourceManager remains idle, using less than 65 MiB of disk per job. It seems NodeManager's activity is the primary driver of disk I/O while DataNode maintains a steady but lower load.

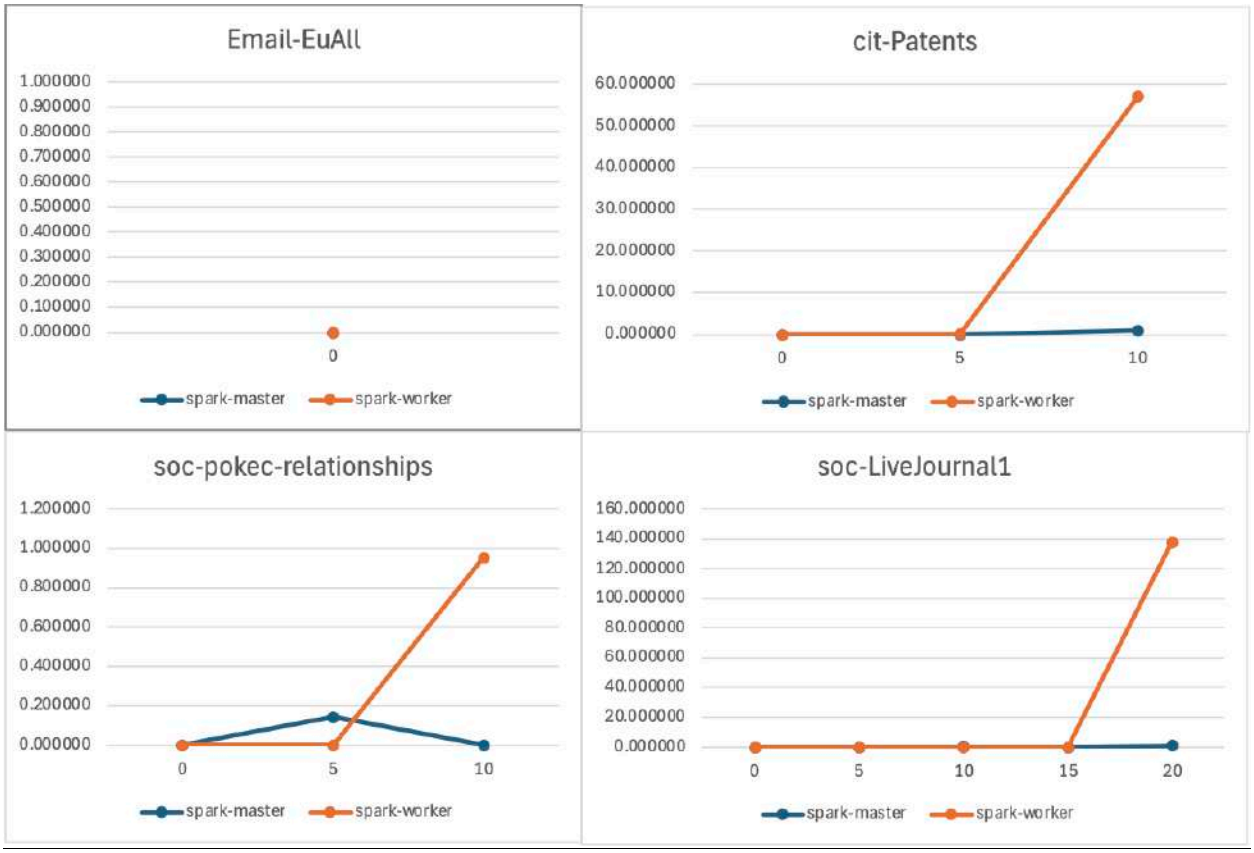
For simple datasets like Email-EuAll, total disk activity is minimal, often staying under 20 MiB. For large dataset like soc-LiveJournal1 causes disk I/O to surge. For the soc-LiveJournal1 dataset, the NodeManager disk io shoots up to a peak of 1,812 MiB in a single interval. This indicates a massive burst of temporary data being written to disk during the shuffle confirming that the "shuffle" creates the highest stress on the disk.

Total Disk IO per Dataset



The chart illustrates the total Disk I/O across four datasets, revealing a correlation between workload complexity and disk IO. The Email-EuAll dataset miniscule footprint. As the workloads scales to cit-Patents and soc-pokec-relationships, disk usage grows linearly, reaching approximately 0.3 GiB and 0.78 GiB. The most significant finding is the disproportionate spike for the soc-LiveJournal1 dataset, which dominates the chart with a peak usage of approximately 1.8GiB which is 2.5x than the soc-pokec-relationships. There seems to be an exponential increase in disk access compared with the dataset sizes.

Spark Disk Utilization



The soc-pokec-relationships dataset traces the highest curve, yet it reaches a peak of only 3.89 MiB at the 10-second mark. The soc-LiveJournal1 workload is even lighter, peaking at 2.82 MiB at the 20-second mark, while cit-Patents registers a maximum of just 1.70 MiB. The Email-EuAll dataset remains a flat baseline near zero.

Spark vs Hadoop Disk IO

There is wide contrast between the disk usage between Hadoop and Spark. While Hadoop's disk usage spiked to 1,822 MiB for the largest workload, Spark's peak activity is over 450 times lower, topping out at less than 4 MiB. This contrast demonstrates how Spark's in-memory architecture fundamentally shifts the execution model, transforming a gigabyte-scale storage workload into a minimal megabyte-scale task.

Performance Conclusion

Hadoops has a disk-centric design and Spark's has a memory-centric architecture. Therefore, Hadoop intermediate data must be written to disk before the next stage begins, resulting in the gigabyte-scale spikes observed in the analysis. Spark, conversely, utilizes an in-memory approach, which allows it to retain intermediate results in RAM. This architectural advantage makes Spark lightweight compared to Hadoop.

Part 2: Scalability and Optimization Analysis

Please refer Part 1 charts and descriptions for performance metrics with soc-LiveJournal1.

Optimizations

Hadoop Optimization

To improve the performance of the Hadoop job, a combiner was introduced. The combiner performs partial aggregation on each mapper prior to the shuffle phase. This reduces the number of intermediate records sent over the network. This decreases the shuffle overhead bandwidth usage and lowers overall job runtime. Since the combiner reduces shuffle volume, so reducers receive and merge map outputs faster and can begin the reduce phase sooner.

Dataset	Size	Hadoop Time	Optimized Hadoop Time
Email-EuAll	5 MB	56	56
cit-Patents	280 MB	115	106
soc-pokec-relationships	423 MB	161	144
soc-LiveJournal1	1.08 GB	300	267

The optimization had no measurable impact on the smallest dataset. Email-EuAll remained at 56 seconds before and after optimization. For the larger datasets, runtime decreased consistently. cit-Patents improved from 115 seconds to 106 seconds, soc-pokec-relationships improved from 161 seconds to 144 seconds, and soc-LiveJournal1 improved

from 300 seconds to 267 seconds. Overall, the benefit increased with dataset size, showing the optimization is more effective when shuffle volume is higher.

Spark Optimization

The Spark workloads were optimized by increasing the number of partitions Spark uses for parallel operations. This was done using the spark.default.parallelism=8 property. Initially this was 4 parallel partitions. This will allow Spark to use more CPU cores at the same time making counting and aggregation work is split into more tasks.

Dataset	Size	Spark Time	Optimized Spark Time
Email-EuAll	5 MB	4	4
cit-Patents	280 MB	16	16
soc-pokec-relationships	423 MB	18	18
soc-LiveJournal1	1.08 GB	37	36

The Spark parallelism change had almost no effect on runtime across the datasets. Email-EuAll stayed at 4 seconds, cit-Patents stayed at 16 seconds, and soc-pokec-relationships stayed at 18 seconds after optimization. Only soc-LiveJournal1 showed a small improvement from 37 seconds to 36 seconds. This indicates the workload was already efficiently parallelized.

Analysis

Hadoop and Spark show different performance patterns because they process data in different ways. Hadoop writes intermediate results to disk between stages and spends more time in shuffle and file I O. Spark keeps more data in memory and reduces repeated disk writes. This gives Spark a large advantage in end-to-end runtime. The Hadoop combiner reduced shuffle volume and improved runtime more on larger datasets, which matches the observed gains.

Spark is better suited for processing large scale graph data in this experiment. It completed all datasets much faster than Hadoop. The Hadoop optimization improved results but Hadoop remained slower, especially on the largest dataset. The Spark parallelism increase

produced almost no change, which suggests Spark was already using available resources efficiently for this workload. This is proven from the cpu usage charts as well as it Spark is utilizing all the cores available.

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

Part 1: Setup and Environment Configuration

Source : https://github.com/lolitha-lakshan-1/bigdata-assignment/tree/main/Task_2/part-1

1. Installed and configured a Kafka cluster with Docker. The cluster is configured with "kafka", "kafka-exporter", "kafka-connect", "prometheus", "grafana" and "zookeeper" applications.

2. A side cart pod was configured to create the Kafka topics "traffic.raw" and "traffic.processed". The side cart pod will also wait until the Kafka pod get initialized, by monitoring the health check endpoint to prevent premature termination of Kafka Connect.

3. Kafka Connect will support ingesting data through the api and export data to an another medium / storage etc.

4. Kafka Exporter, Prometheus and Grafana will help with the visualization of stats related to the cluster.

5. Following Grafana dashboard panels was developed to cover the aspects of health and performance of the cluster, and the Prometheus was configured as a data source of the dashboard. Grafana has following dashboard panels.

- "Incoming Message Rate" which displays the throughput. A high rate on traffic.raw with low on traffic.processed indicates a processing bottleneck.

```
"sum by (topic) (rate(kafka_topic_partition_current_offset{topic=~\"traffic.*\"}[1m]))"
```

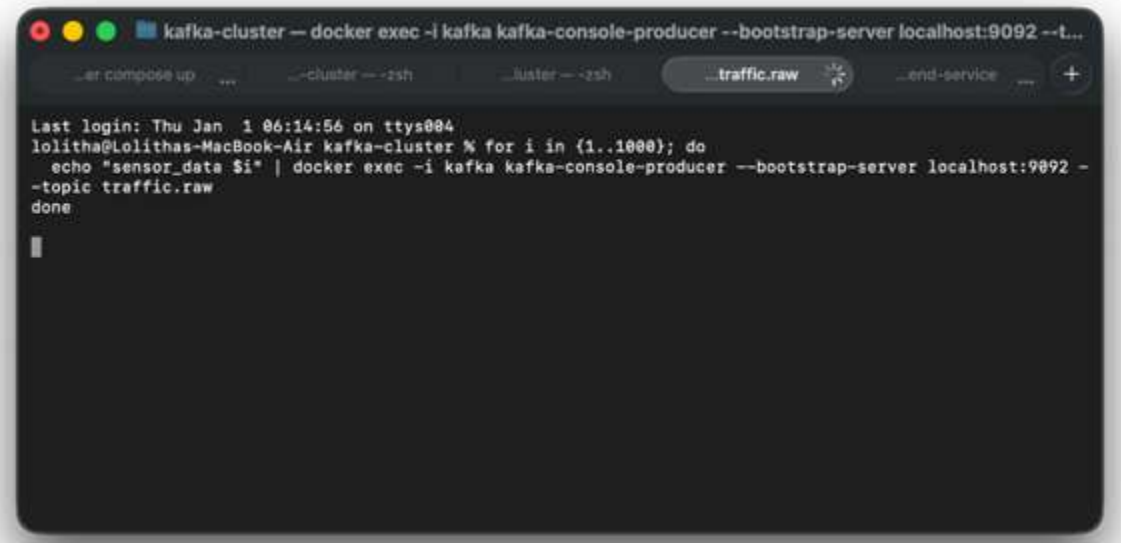
- Total Messages Processed which indicates volume.

```
"sum by (topic) (kafka_topic_partition_current_offset{topic=~\"traffic.*\"})"
```

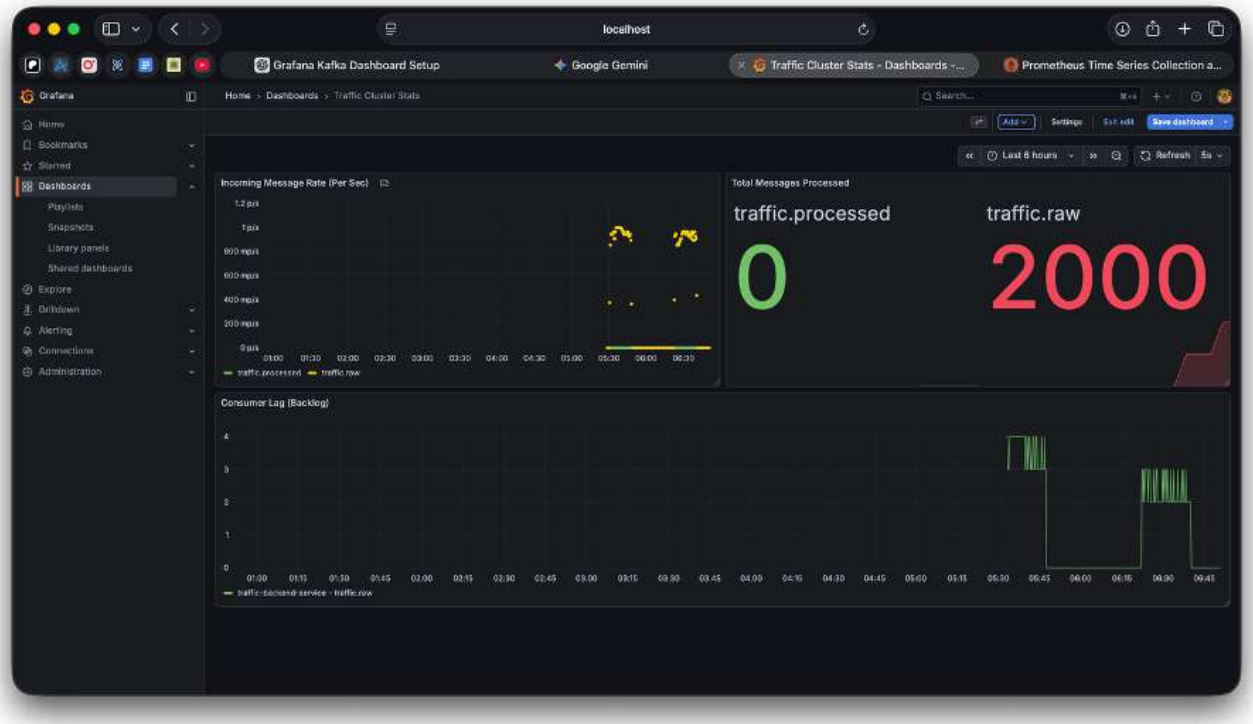
- Consumer Lag panel indicates how far behind consumers are in reading messages from topics which indicates Processing Health.

```
"sum by (consumergroup, topic) (kafka_consumergroup_lag{topic=~\"traffic.*\"})"
```

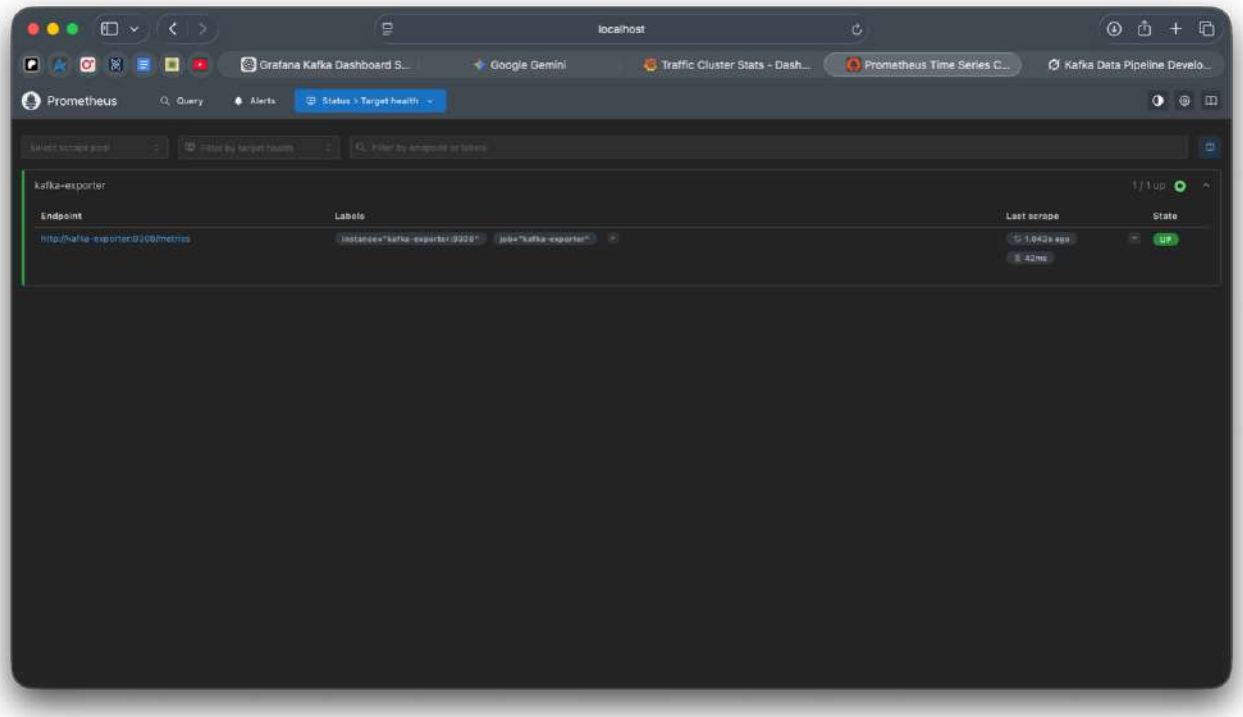
6. Data ingestion was simulated using the command to ingest 1000 rows through a loop.



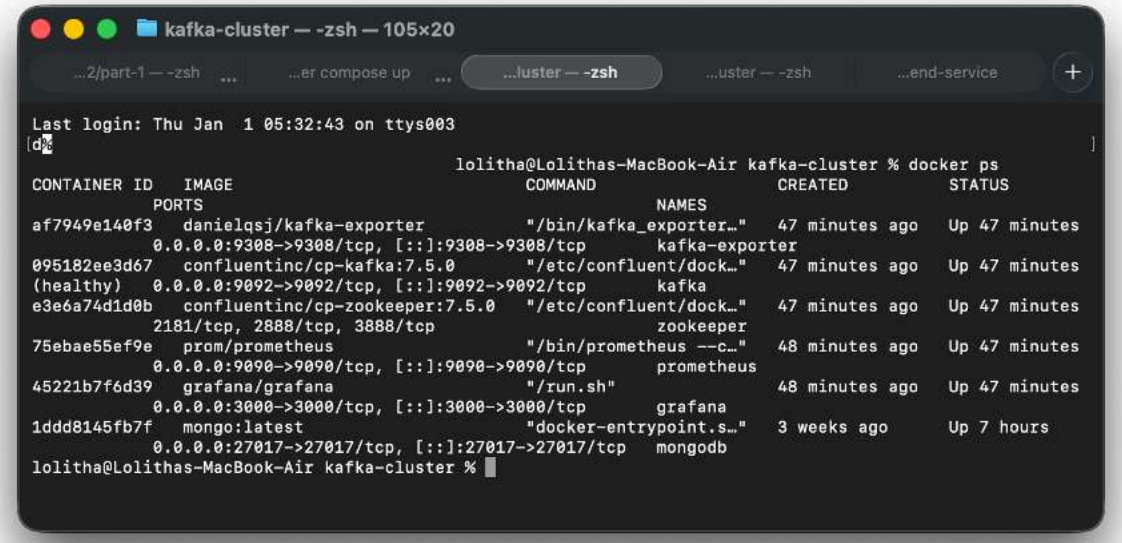
Grafana Dashboard



Prometheus Exporter



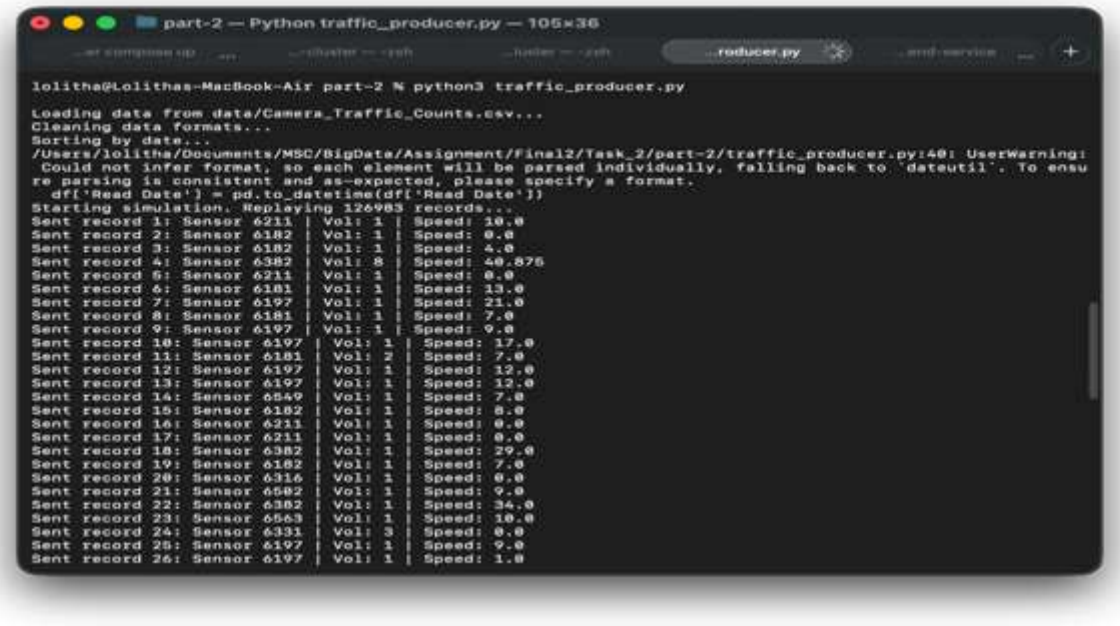
Environment Setup



Part 2: Data Source and Preprocessing

A Python script was written as the Kafka Producer which simulate traffic data every 5 seconds by reading the CSV. This script uses the kafka library and data is sent as JSON messages to the "traffic.raw" kafka topic. Prior to the conversion the script cleans the data by removing commas from the columns (e.g., "1,234" → 1234), also cleans and converts numeric values to integer. Furthermore, drop any rows if column values are missing.

Source: https://github.com/lolitha-lakshan-1/bigdata-assignment/blob/main/Task_2/part-2/traffic_producer.py



Part 3: Streaming Data Processing and Analysis

3.1 Data Ingestion

Modified the previously written “traffic_producer.py” script to send data continuously without a delay.

3.2 Processing and Computation

Wrote a real time stream processor using python to consume data from a Kafka topic (traffic.raw), it aggregates data into hourly averages and daily summaries. Thereafter it writes to a PostgreSQL database and publish the processed data to the "traffic.processed" Kafka topic.

Source : https://github.com/lolitha-lakshan-1/bigdata-assignment/blob/main/Task_2/part-3/traffic_processor.py

The "traffic_processor.py" stream processor has following main helper functions to calculate the statistics.

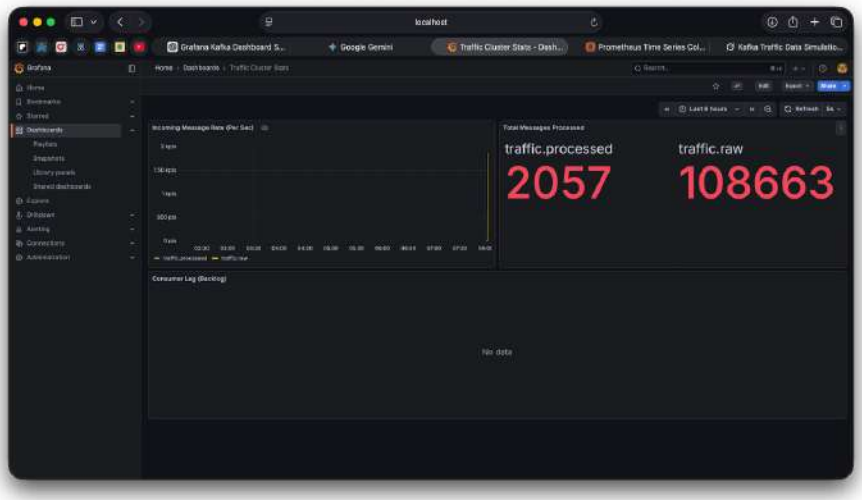
The update_running_totals function processes each incoming traffic record. For the given record, it adds the record's volume to sum_vol and the speed to sum_speed within the hourly_stats dictionary entry for that sensor. It also increments the count of records seen for that sensor in the current hour. Simultaneously, it compares the current volume against the peak_volume in daily_state. If higher, it updates the peak_volume and sets peak_sensor to the current sensor_id. Finally, it adds the sensor_id to both the active_sensors_today set (tracking sensors reporting on the current day) and the all_known_sensors set (tracking every sensor ever encountered).

The calculate_hourly_averages function is called for each new hour. It iterates over the accumulated hourly_stats dictionary from the current hour and computes avg_vol as sum_vol / count and avg_speed as sum_speed / count for each sensor with data. It builds a list of db_rows for database insertion and creates corresponding kafka_messages. After processing, hourly_stats is reset to an empty dictionary for the new hour.

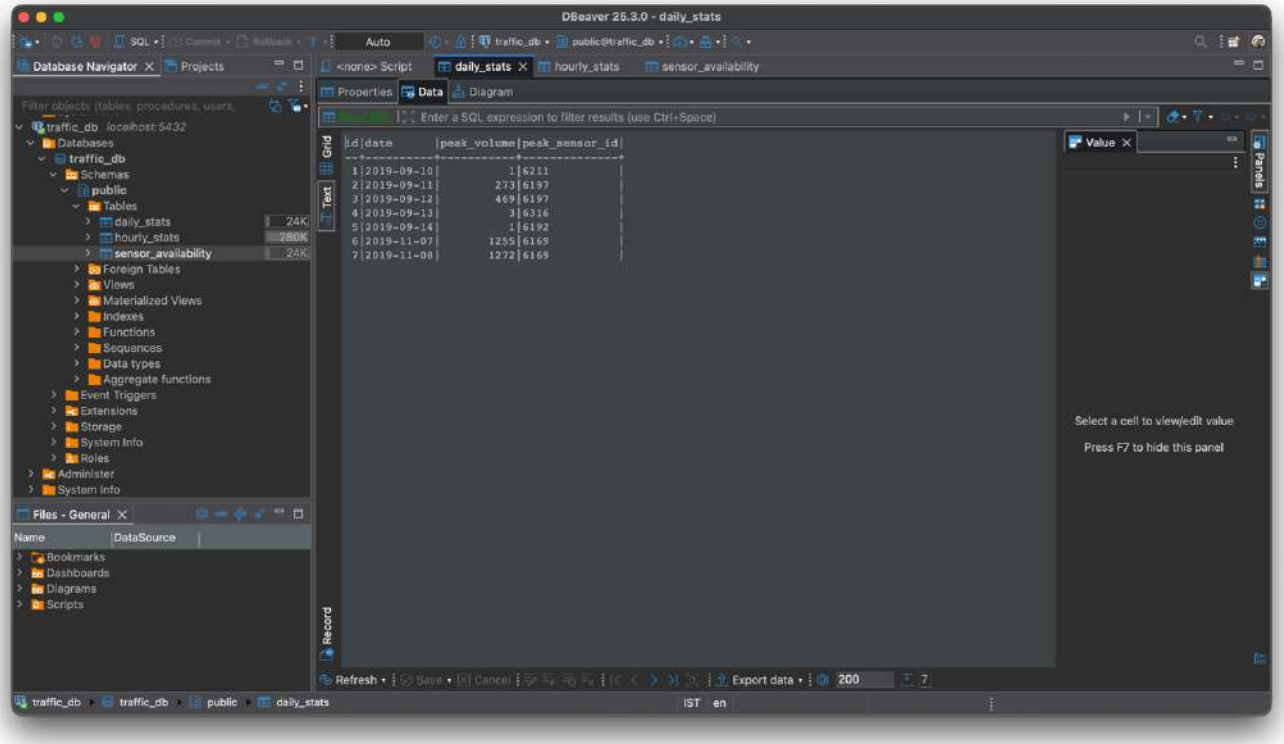
The calculate_daily_summary function executes when the date changes. It uses the data accumulated in daily_state to determine the day's peak_volume and associated peak_sensor, counts the number of active_sensors_today, and computes availability_pct as the percentage of active_sensors_today relative to the total size of all_known_sensors. It returns a summary dictionary containing the date, peak values, active count, and availability percentage. Subsequently, the daily tracking fields—peak_volume, peak_sensor, and active_sensors_today—are reset, while all_known_sensors persists across days.

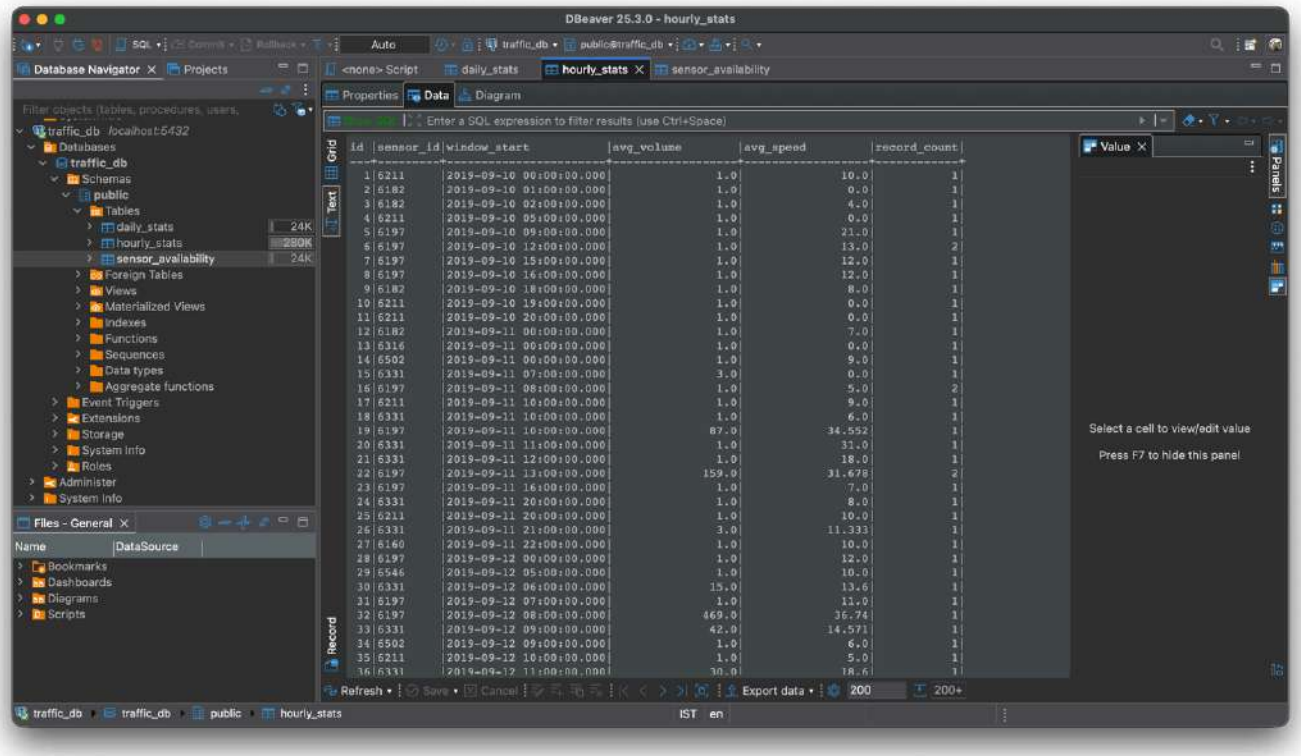
3.3 Output and Persistence

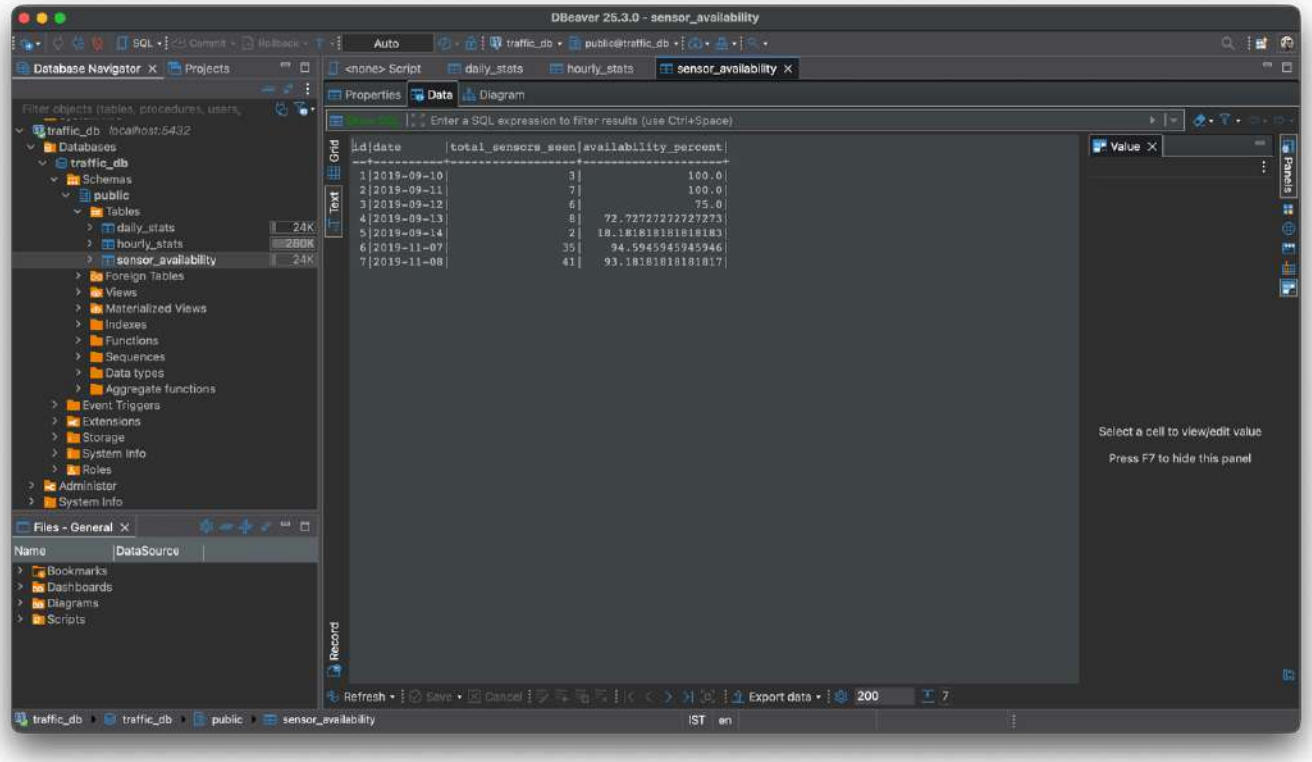
Kafka Data Processed Topic



Data Stored in Postgres DB

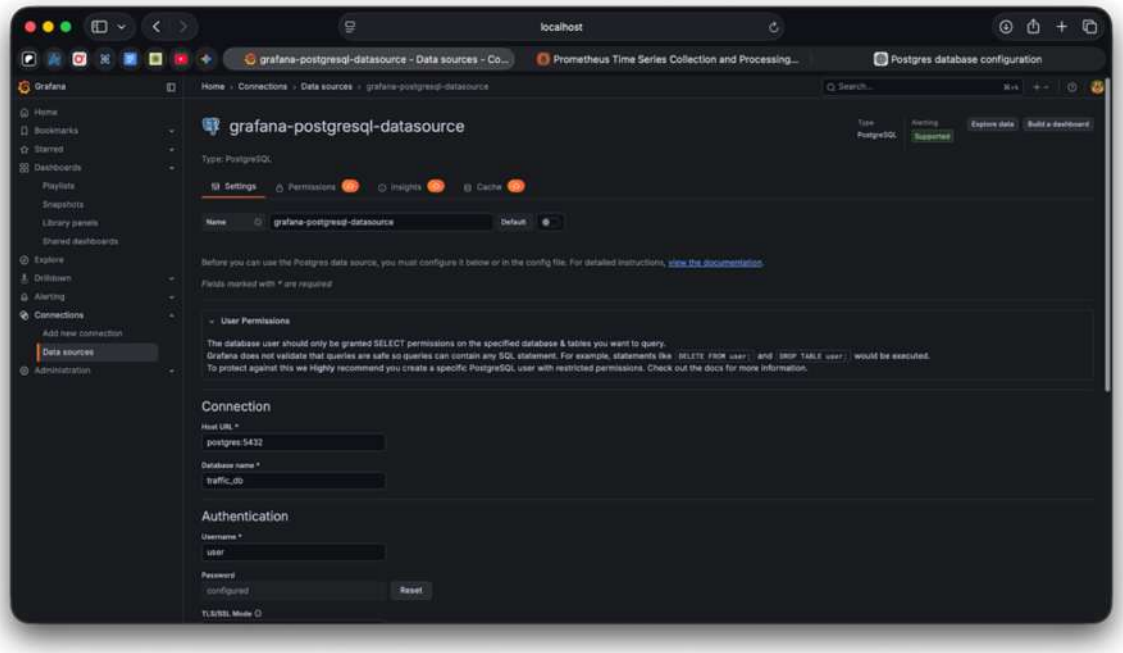



DBeaver 25.3.0 - hourly_stats
Database Navigator
Projects
hourly_stats
sensor_availability
Enter a SQL expression to filter results (use Ctrl+Space)
Value
Select a cell to view/edit value
Press F7 to hide this panel


DBeaver 25.3.0 - sensor_availability
Database Navigator
Projects
hourly_stats
sensor_availability
Enter a SQL expression to filter results (use Ctrl+Space)
Value
Select a cell to view/edit value
Press F7 to hide this panel

Part 4: Visualization and Reporting

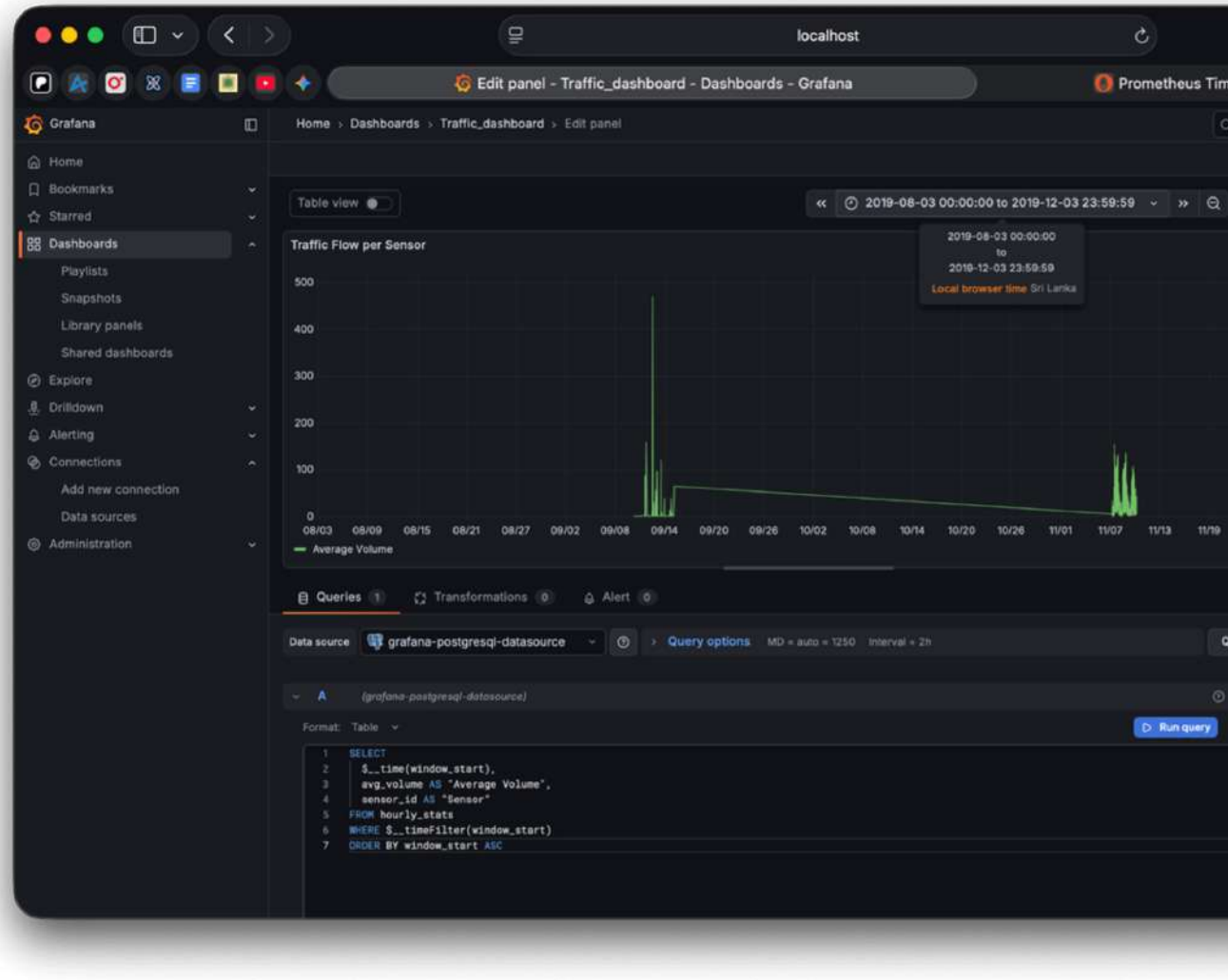
1. First updated the Grafana datasource with the Postgres datasource and added the db name and db credentials.



2. Created the dashboard panels using the UI and following queries.

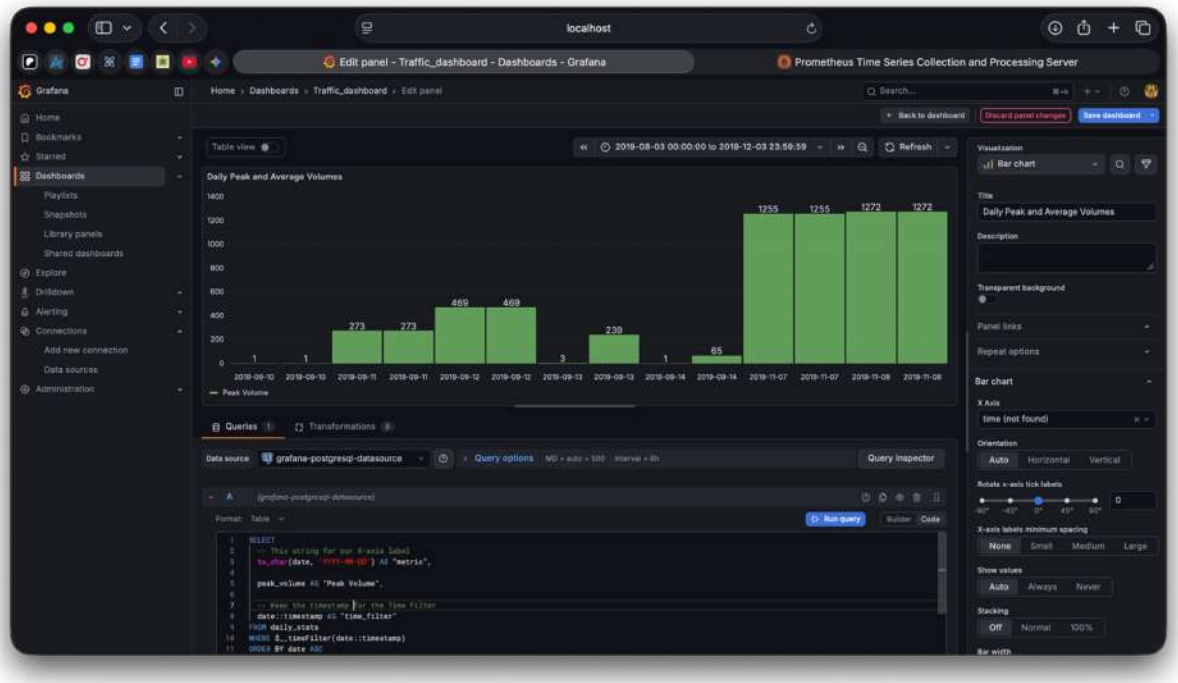
Source: https://github.com/lolitha-lakshan-1/bigdata-assignment/blob/main/Task_2/part-4/kafka-cluster/grafana/provisioning/dashboards/traffic_dashboard.json

Time-series chart for traffic flow per sensor



This query works by selecting the window_start as the timestamp and the avg_volume as the plotted value. The sensor_id is cast to text using, which forces Grafana to treat the ID as a label rather than a number, resulting in a distinct line for every sensor.

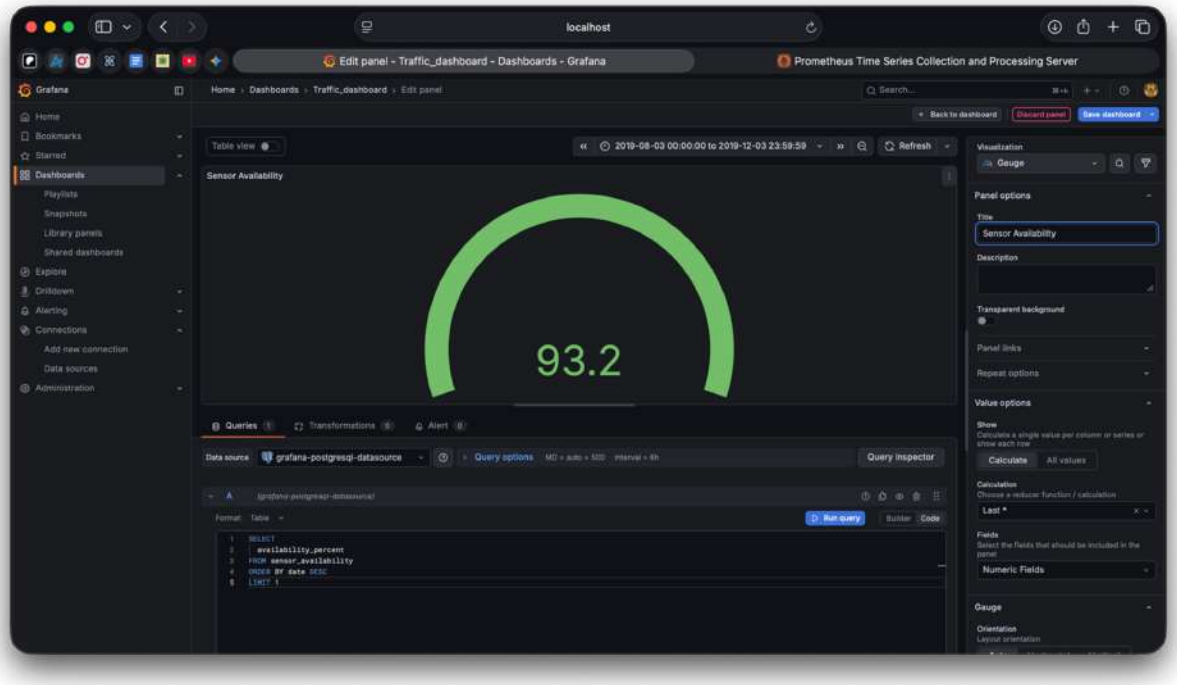
Bar chart for daily peak and average volumes




```
SELECT
to_char(date, 'YYYY-MM-DD') AS "metric",
peak_volume AS "Peak Volume",
date::timestamp AS "time_filter"
FROM daily_stats
WHERE $__timeFilter(date::timestamp)
ORDER BY date ASC
```

Since Grafana's time-axis requires precise timestamps rather than simple dates, date column had to be cased to place the bars correctly on the X-axis.

Gauge indicators or tables for sensor availability and Health




```
SELECT
  availability_percent
FROM sensor_availability
ORDER BY date DESC
LIMIT 1
```

The Sensor Availability gauge was created with the above query which will fetch most recent record by ordering the data descending.

Task 3: Graph Databases using Neo4j and Docker Objective

Part 1 — Understanding Graph Databases

Fraud Detection In Financial Services

Major financial institutions like JPMorgan Chase utilize graph databases to uncover sophisticated fraud rings or money laundering schemes. The traditional relational databases struggle in this context as indirect connections would require complex JOIN queries to uncover these sorts of data. In a graph data base Nodes will represent entities such as Customers, Accounts, ATMs as well as meta data such as ip addresses, device ids and phone numbers.

Graph relationships will include actions such as `TRANSFERRED_TO`, `LOGGED_IN_WITH`, or `SHARES_DEVICE`. Therefore, a graph database will help investigators to instantly traverse through the network to spot non-obvious patterns, such as 20 seemingly unrelated accounts all accessing banking services from the same mobile device ID. In a graph database these kinds of patterns would be easy to spot and discover than a traditional relational database. In a traditional database this information will be buried under columns, rows and may be distributed across tables.

- Useful Query: Cycle Detection like find a path where money leaves Account A, passes through 5 intermediaries, and returns to an account held by A's known associate.

Source: Neo4j Financial Services Use Cases & JPMorgan Chase AI Research

Food Discovery

Uber Eats built a Food Knowledge Graph to address the zero-result problem which occurs when a specific search term yields no exact match, which could user to abandon the app. A standard database would simply return and empty results for non-existing search term, but a graph structure has the capability to understand the *semantic* relationships between food items, allowing for intelligent substitutions.

In a graph database, nodes can be utilized save entities such as menu items, cuisines, food attributes such as Spicy, Vegan. Furthermore, relationships can be utilized to define associations of the nodes such as IS`_A_TYPE_OF` or `OFTEN_ORDERED_WITH`.

With these relationships graph databases will allow to expand the query to look for associated food items if no direct food items were found. For an example If a user searches for a specific dish that isn't available, the graph knows that "Udon" is related to "Ramen" via the "Japanese Noodle" parent node, allowing the app to recommend a relevant alternative instantly.

- Useful Query: Semantic Similarity (e.g., Find all restaurants serving items connected to the Japanese Cuisine)

Source: Uber Engineering Blog: Food Discovery with Uber Eats

Contextual Search & Recommendations In Ecommerce

eBay developed a distributed knowledge graph (internally called "Akutan") to better understand the massive inventory of over 1 billion items, moving beyond simple keyword matching. Relational databases are rigid and can struggle to define with the highly variable attributes of diverse products (e.g., a "screen size" matters for phones but not for sneakers), whereas a graph flexibly links products to real-world concepts. In an ecommerce site

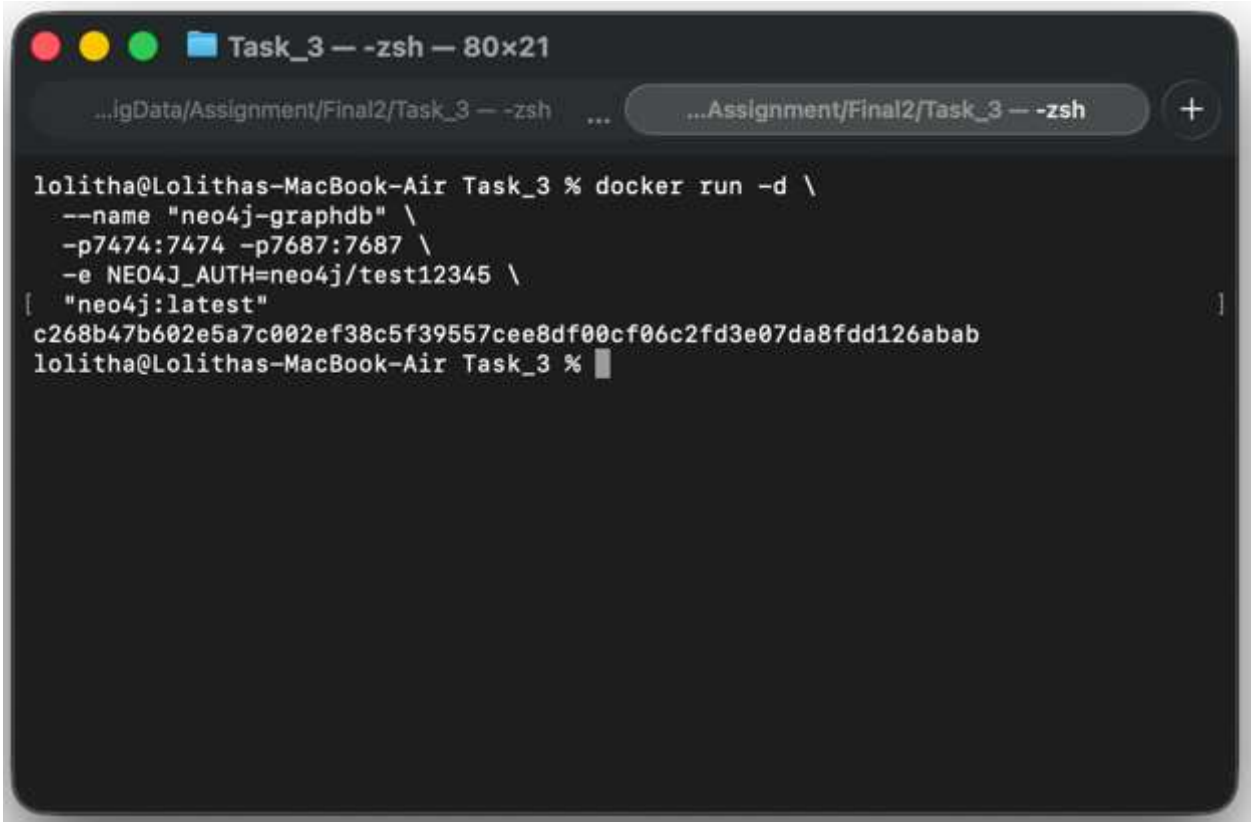
Products (iPhone 12), Events (Super Bowl), and Entities (Celebrities, Brands) can be stored as graph database nodes and connections such as like `COMPATIBLE_WITH`, `USED_BY`, or `RELEASED_DURING` can be expressed using graph database relationships. These relationships would help with complex questions where keyword search will fail for an example a specific phone case `IS_COMPATIBLE_WITH` a specific phone model, eBay can filter out thousands of irrelevant results that just happen to have similar keywords in the title.

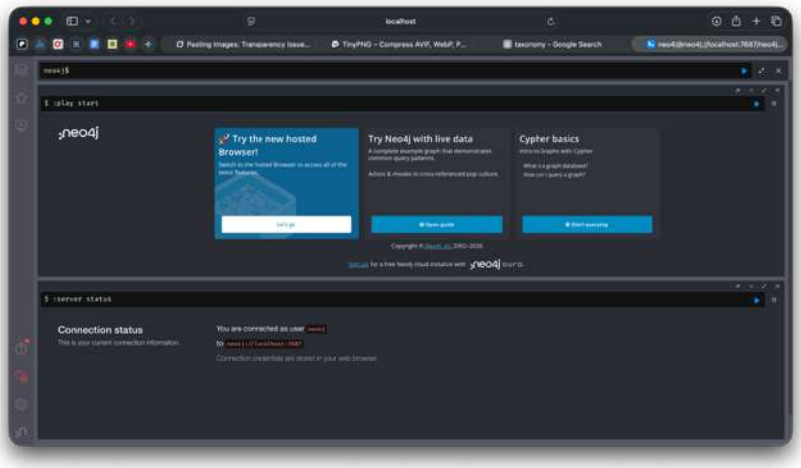
- Useful Query: Pathfinding (e.g., Return all distinct accessories linked via `COMPATIBLE_WITH` edges to the specific electronics item the user is currently viewing).

Source: eBay Innovation Blog: Akutan Knowledge Graph

Part 2 — Implementation with Neo4j (Docker Setup)

Implementation





Dataset

The dataset simulates a financial fraud ring. It connects people, accounts, and digital footprint to reveal suspicious activity.

Following nodes are present in the dataset,

- Customers: 5 individuals (C1–C5).
- Accounts: 6 bank accounts (A1–A6).
- Infrastructure: 3 Devices (Phones/Tablets) and 2 IP Addresses.

And Nodes are connected using following relationships,

- Customers OWNS accounts.
- Customers LOGGED_IN_WITH devices and LOGGED_IN_FROM IP addresses.
- Accounts TRANSFERRED_TO other accounts with amount and currency.

As an overview the C1 Customer controls two accounts, A1 and A2, and money transfers to A1 and moves step by step through other customers' accounts, then returns to A1, which creates a clear circular transfer loop. The login links make the connections feel intentional because C1 and C2 share the same device and the same IP address, C3 and C4 share another device and IP, and C5 also logs in from the same IP as C1 and C2 which suggests a money mule.

fraud_data_Cnames_customers_login.cypher

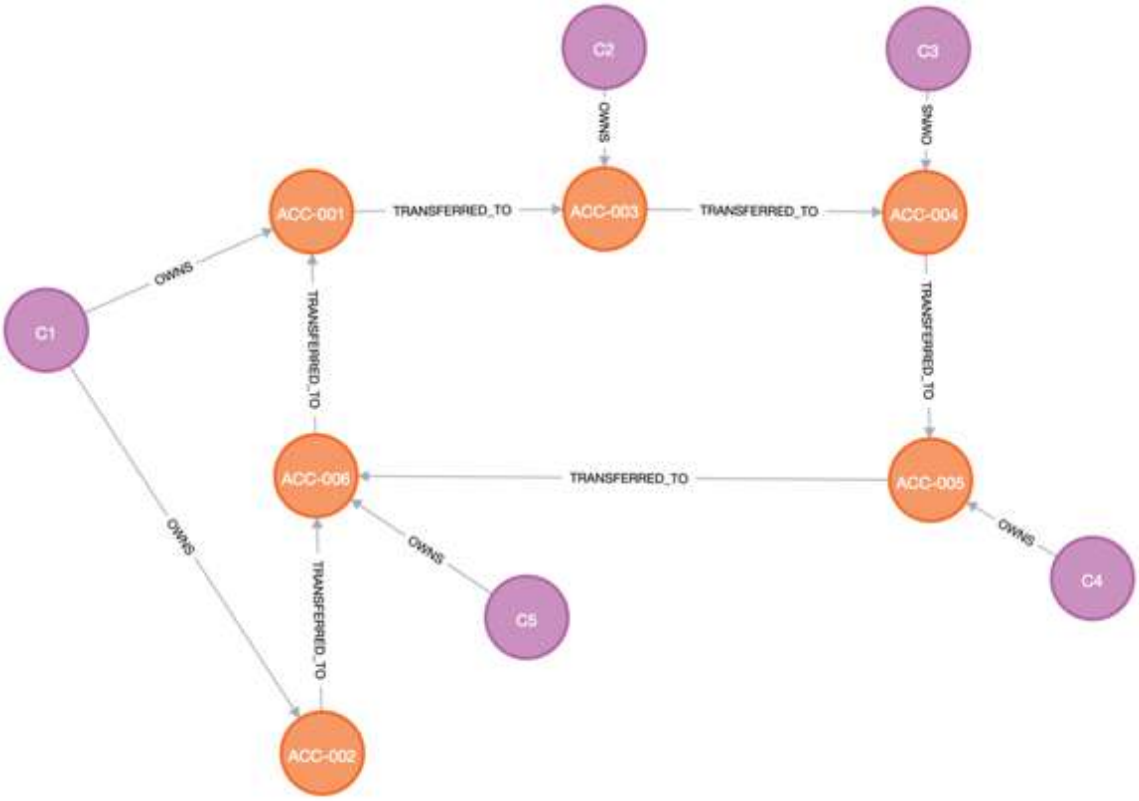
```
MATCH (n)
DETACH DELETE n;

CREATE
  (c1:Customer {id: 'C1', name: 'C1'})-[:OWNS]->(a1:Account {id: 'A1', accountNo: 'ACC-001'}),
  (c1)-[:OWNS]->(a2:Account {id: 'A2', accountNo: 'ACC-002'}),

  (c2:Customer {id: 'C2', name: 'C2'})-[:OWNS]->(a3:Account {id: 'A3', accountNo: 'ACC-003'}),
  (c3:Customer {id: 'C3', name: 'C3'})-[:OWNS]->(a4:Account {id: 'A4', accountNo: 'ACC-004'}),
  (c4:Customer {id: 'C4', name: 'C4'})-[:OWNS]->(a5:Account {id: 'A5', accountNo: 'ACC-005'}),
  (c5:Customer {id: 'C5', name: 'C5'})-[:OWNS]->(a6:Account {id: 'A6', accountNo: 'ACC-006'}),

  (d1:Device {id: 'D1', deviceId: 'dev-xyz', type: 'Phone'}),
  (d2:Device {id: 'D2', deviceId: 'dev-abc', type: 'Phone'}),
  (d3:Device {id: 'D3', deviceId: 'dev-789', type: 'Tablet'}),

  (ip1:IPAddress {id: 'IP1', address: '10.0.0.10'}),
  (ip2:IPAddress {id: 'IP2', address: '10.0.0.20'}),

  // Money transfers (fraud ring pattern)
  (a1)-[:TRANSFERRED_TO {amount: 2500, currency: 'USD'}]->(a3),
  (a3)-[:TRANSFERRED_TO {amount: 2600, currency: 'USD'}]->(a4),
  (a4)-[:TRANSFERRED_TO {amount: 2700, currency: 'USD'}]->(a5),
  (a5)-[:TRANSFERRED_TO {amount: 2800, currency: 'USD'}]->(a6),
  (a2)-[:TRANSFERRED_TO {amount: 5000, currency: 'USD'}]->(a6),
  (a6)-[:TRANSFERRED_TO {amount: 2400, currency: 'USD'}]->(a1),

  (c1)-[:LOGGED_IN_WITH]->(d1),
  (c2)-[:LOGGED_IN_WITH]->(d1),
  (c3)-[:LOGGED_IN_WITH]->(d2),
  (c4)-[:LOGGED_IN_WITH]->(d2),
  (c5)-[:LOGGED_IN_WITH]->(d3),

  (c1)-[:LOGGED_IN_FROM]->(ip1),
  (c2)-[:LOGGED_IN_FROM]->(ip1),
  (c5)-[:LOGGED_IN_FROM]->(ip1),
  (c3)-[:LOGGED_IN_FROM]->(ip2),
  (c4)-[:LOGGED_IN_FROM]->(ip2);
```

Queries

1. Find Associates of C1Customer

```
MATCH (startCustomer:Customer {name: 'C1'})-[:OWNS]->(startAccount:Account)
MATCH paths = (startAccount)-[:TRANSFERRED_TO*1..4]-(:Account)<-[:OWNS]-(otherCustomer:Customer)
RETURN paths, otherCustomer;
```

The query is trying to answer "Which other customers are connected to C1 through fund transfers?" and it does this by starting from C1's accounts and following 1– 4 TRANSFERRED_TO hops between accounts to find any other customers who receive money directly or indirectly.

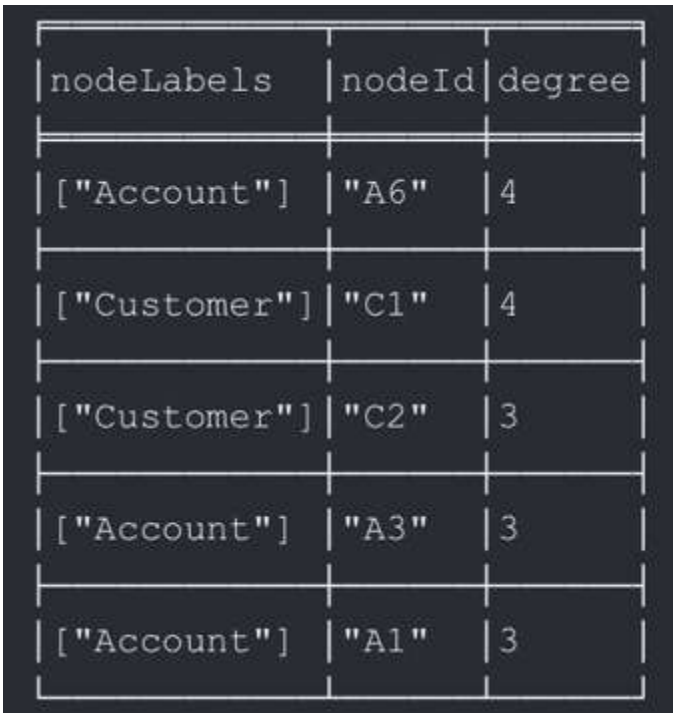


The result shows that money flowing out of ACC-001 and ACC-002 and eventually reaches accounts owned by C2, C3, C4, and C5. This means all four of those customers are transactionally connected to C1, forming a small network of potentially related or suspicious activity.

2. Most connected node

```
MATCH (node) RETURN labels(node) AS nodeLabels, node.id AS nodeId, COUNT { (node)--() } AS degree ORDER BY degree DESC LIMIT 5;
```

This query tries to find which nodes act as hubs in the fraud graph by counting how many relationships each node has. It counts all connections and then returns the top 5 most connected nodes.



nodeLabels	nodeId	degree
["Account"]	"A6"	4
["Customer"]	"C1"	4
["Customer"]	"C2"	3
["Account"]	"A3"	3
["Account"]	"A1"	3

The result shows that account A6 and customer C1 both have degree of 4, meaning they each touch four other nodes, making them strong candidates as suspicious entities. Accounts A3 and A1, and customer C2, also have relatively high degrees (3), indicating they participate in multiple transfers or relationships.

- A6 – owned by customer C5; receives money from A5 and A2 and sends money to A1.
- C1 – owns accounts A1 and A2, has LOGGED_IN_WITH → D1 and has LOGGED_IN_FROM → IP1.
- C2 – owns account A3, has LOGGED_IN_WITH → D1 and has LOGGED_IN_FROM → IP1.
- A3 – owned by C2, receives money from A1, sends money to A4.
- A1 – owned by C1, receives money from A6, sends money to A3.

3. Customers sharing both a device and an IP

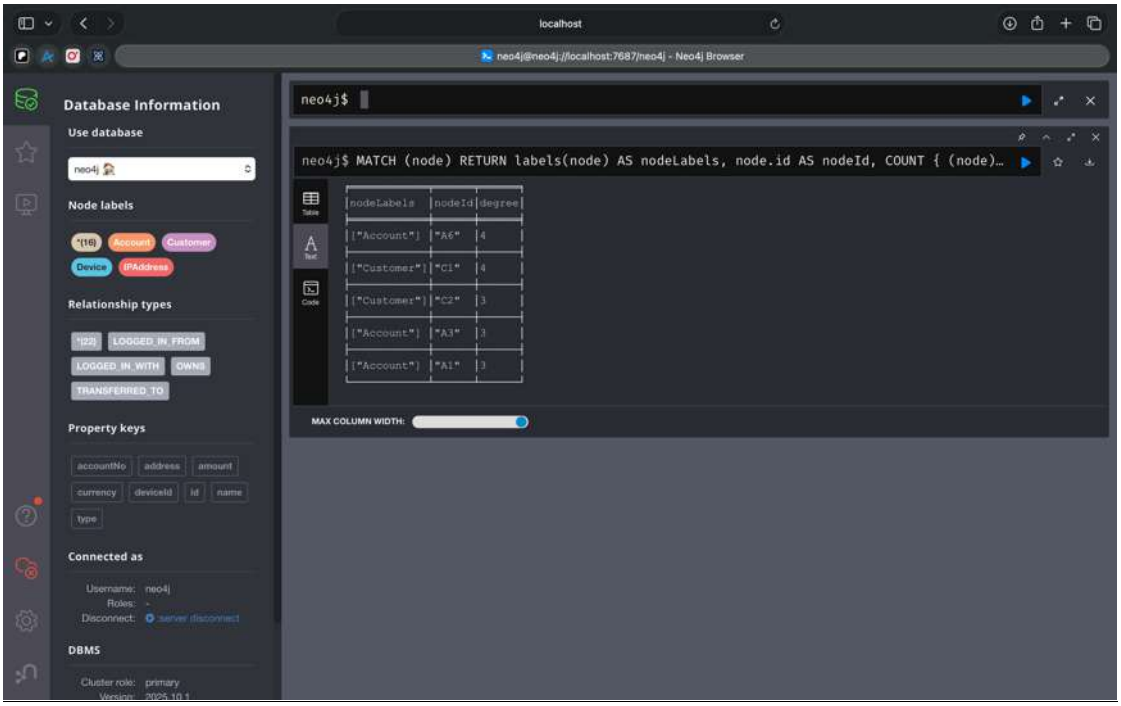
This query looks for pairs of customers who share both the same device and the same IP address, which is a strong signal that they might be the same person or closely colluding accounts.

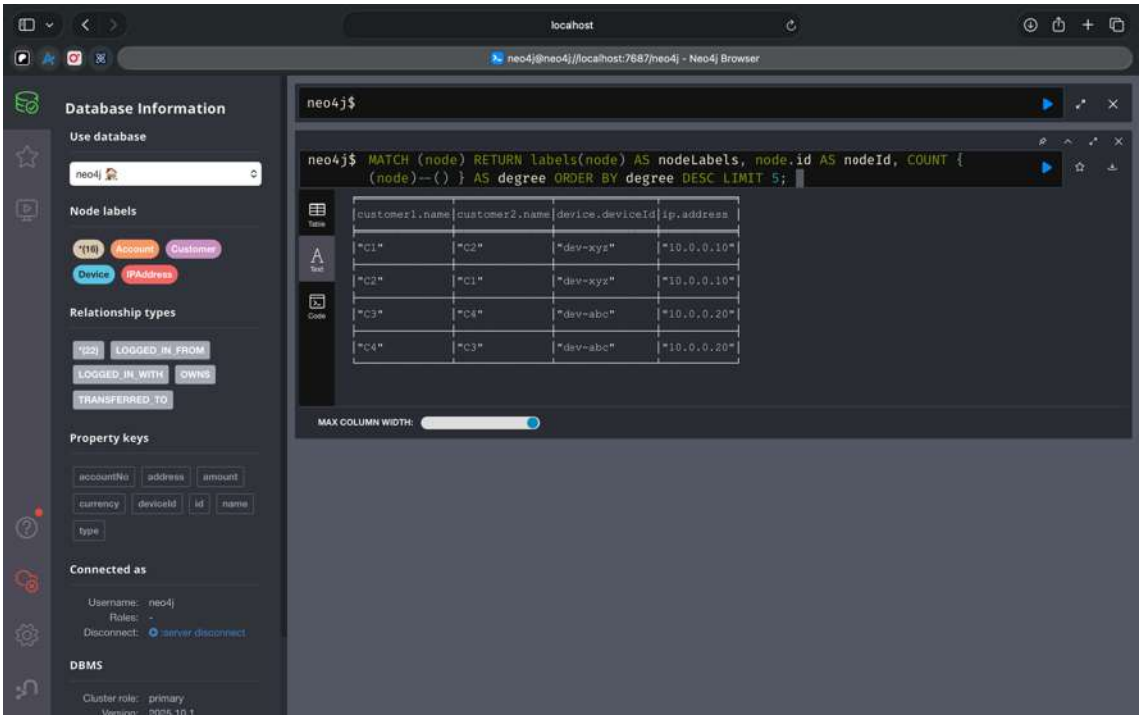
```
MATCH
  (customer1:Customer)-[:LOGGED_IN_WITH]->(device:Device)<-[:LOGGED_IN_WITH]-
(customer2:Customer),
  (customer1)-[:LOGGED_IN_FROM]->(ip:IPAddress)<-[:LOGGED_IN_FROM]-(customer2)
WHERE customer1 <> customer2
RETURN DISTINCT
  customer1.name,
  customer2.name,
  device.deviceId,
  ip.address
ORDER BY customer1.name, customer2.name;
```

customer1.name	customer2.name	device.deviceId	ip.address
"C1"	"C2"	"dev-xyz"	"10.0.0.10"
"C2"	"C1"	"dev-xyz"	"10.0.0.10"
"C3"	"C4"	"dev-abc"	"10.0.0.20"
"C4"	"C3"	"dev-abc"	"10.0.0.20"

The result shows C1–C2 sharing dev-xyz and 10.0.0.10, and C3–C4 sharing dev-abc and 10.0.0.20. These pairs should get high interest from a fraud perspective because multiple customer profiles appear to be operated from the same physical device and network.

Screenshots





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

Part 1 — Understanding Watermarks

*4.1.1 What are **watermarks** in stream processing?*

The real time systems consume never ending stream of data. This data is used to generate outputs such as sums, averages or counts. This is usually done by grouping the events into time windows. However, due to unforeseen circumstances data can arrive out of order. A watermark is a rule which specifies whether the time window for processing the data elapsed or not, rather than waiting forever for late events to arrive.

For an example imagine a city traffic system where the road sensors send the number of cars every minute. Due to a slow network a reading measured at 10:01 might arrive late at 10:03. Without watermarks, the system will close the 10:00–10:01 window too early which would result missing the late reading thus resulting wrong traffic stats,

With watermarks, we can say specify wait up to 30 seconds for late data, then close the window and publish the result which gives a good balance between freshness and accuracy specially for live dashboards.

4.1.2 Watermark Generation Strategies

The three main strategies for generating watermarks are Periodic, Punctuated, and Event-Time-Based.

Periodic watermarks

The periodic watermark strategy generates watermarks at fixed intervals based on processing time or the system clock (e.g., every 100ms). The system calculates the watermark by finding the maximum event time observed during the period and then subtracting a pre-configured 'allowed lateness' delay. This technique ensures that the system can consistently advance without stalling, even when the incoming data stream is inactive or when data is sparse.

Punctuated watermarks

Punctuated watermarks are created only when a special marker event appears in the data stream. This marker event carries a flag or metadata which tells the stream processor that time has elapsed. Therefore, punctuated watermarks do not use a timer; instead, the stream processor waits for these marker events and uses the marker event metadata to generate the new watermark. However, the system will stop advancing the time (stall) if these marker events do not arrive as expected.

Event-time-based watermarks

Event-Time-Based Watermarks are based on the actual event time in the data record itself. e.g. when a social media post was actually made. Rather than considering event received time. Therefore, system is focused on measuring the actual timeline of events, regardless of network delays or out-of-order arrival. The watermark advances only when new, later-timestamped data arrives.

```
Watermark = Max Event Time Seen – Fixed Delay
```

Strategy	Pros	Cons
Periodic Watermarks	• Low stall risk • Low processing overhead, as checks occur only at set intervals. • Skew can be easily adjusted to balance accuracy and latency.	• Always introduces a delay, increasing latency. • Cannot handle extremely late events.
Punctuated Watermarks	• Provides the fastest possible update.	• High Stall Risk and High Overhead. • Requires checking every record for the marker flag. • Source Dependency: Relies on the producer to correctly generate the marker events.
Event-Time-Based Watermarks	• Essential for consistent results regardless of network delays or processing lag.	• Requires handling out-of-order data, which introduces complexity in stream processor logic and state management.

4.1.3) Handling late or out-of-order events

Periodic Watermarks

Periodic Watermarks closes the time window as soon as duration is elapsed. The time is based on the system clock rather than the event time therefore the lagging events are considered late. If the network lags watermarks will be generated continuously which would result dropping valid data. To mitigate these following strategies can the implemented.

- Tune Allowed Lateness. In some systems it is possible to keep a window state around for a specific time after the watermark has passed. If a late event arrives during this grace period, the window is re-opened, the calculation is updated, and a corrected result is emitted.
- Implement a Dead Letter Queue and process the data later.

Punctuated Watermarks

The biggest risk with punctuated watermarks is that if the data source goes quiet or marker events does not arrive, the watermark will stop moving which would result a stagnant window. To mitigate this following can be done,

- Implement a timeout, window will move after a timeout occurs.

- If the source sends late event after the mark event handling that with a Dead Letter Queue and process later.

- Use Heartbeats events and move the window when the data stream is empty.

Event-Time-Based Watermarks

For Event-Time-Based Watermarks it is possible to,

- Configure the system to define a specific duration of "allowed lateness" after the watermark has passed the window's end time and handle late date. xw

- Dead Letter Queue and process later.

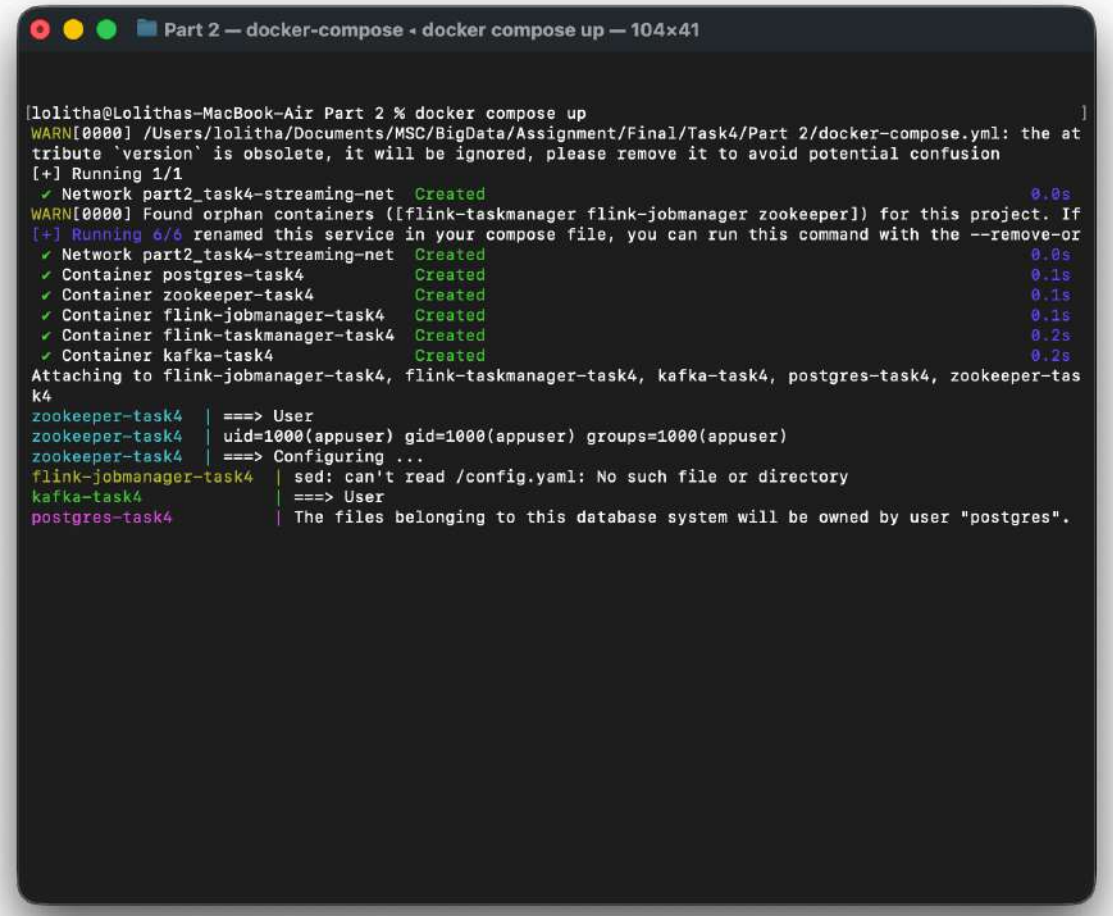
What is best for best for real-world social media streams?

The social media streaming platform have high-volume, spikes of data and globally distributed therefore we should expect to handle bursty and variable network latency of data. The best strategy to handle this would be Event-Time-Based Watermarks with Bounded Out-of-Orderness.

Since it tracks the true progress of the event stream, ensuring that results are based on the actual time the events occurred, despite events arriving out of order. The "Fixed Delay" parameter will allow to balance the need of accuracy and latency where for social media platforms the accuracy is most probably eventually consistent.

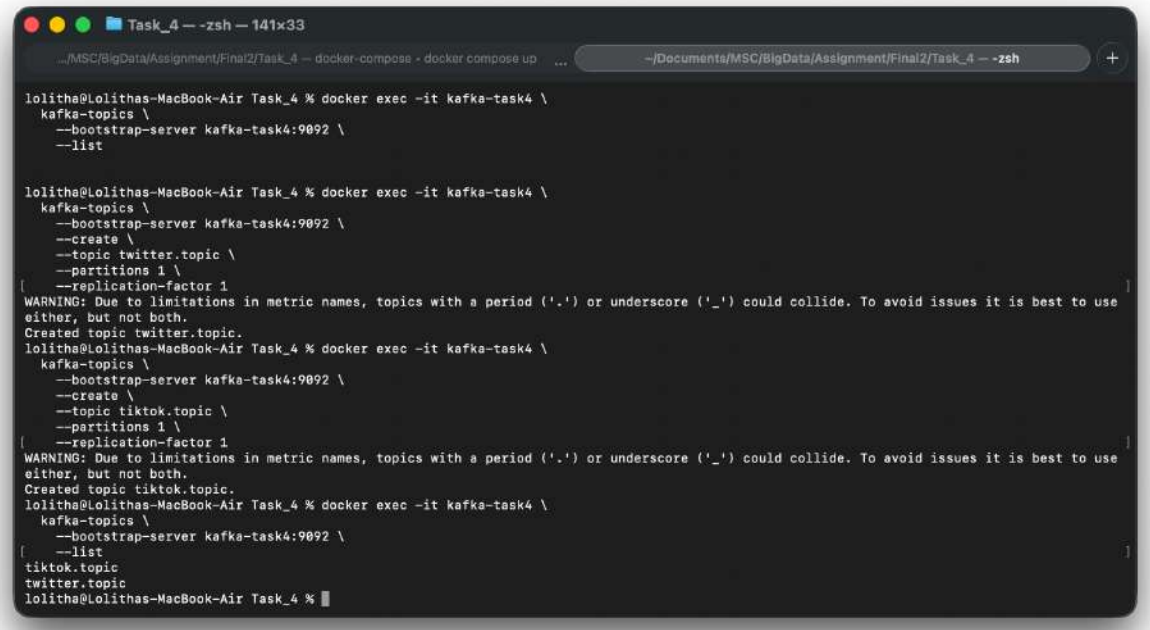
Part 2 — Implementation using Docker (Kafka + Flink)

Step 1 — Environment Setup

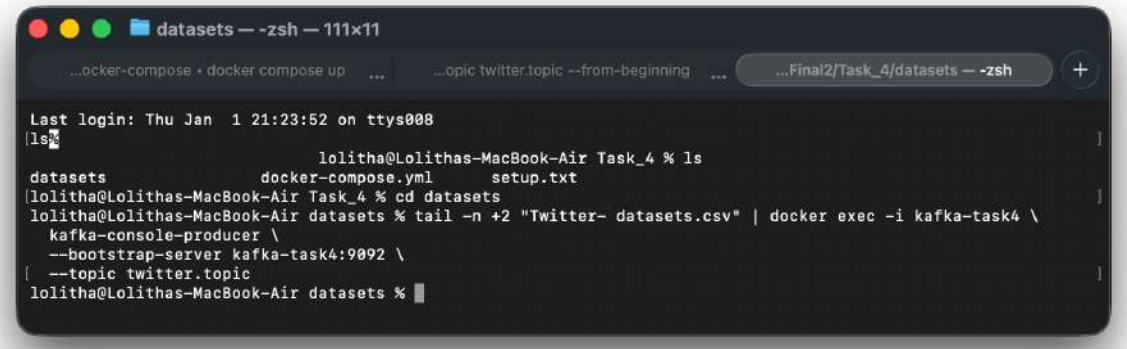


Step 2 — Data and Kafka Topics

Create Topics



Send Data To Topic



Read Data From Topic

Step 3 — Flink Stream Processing

Flink Streaming Application Implementation

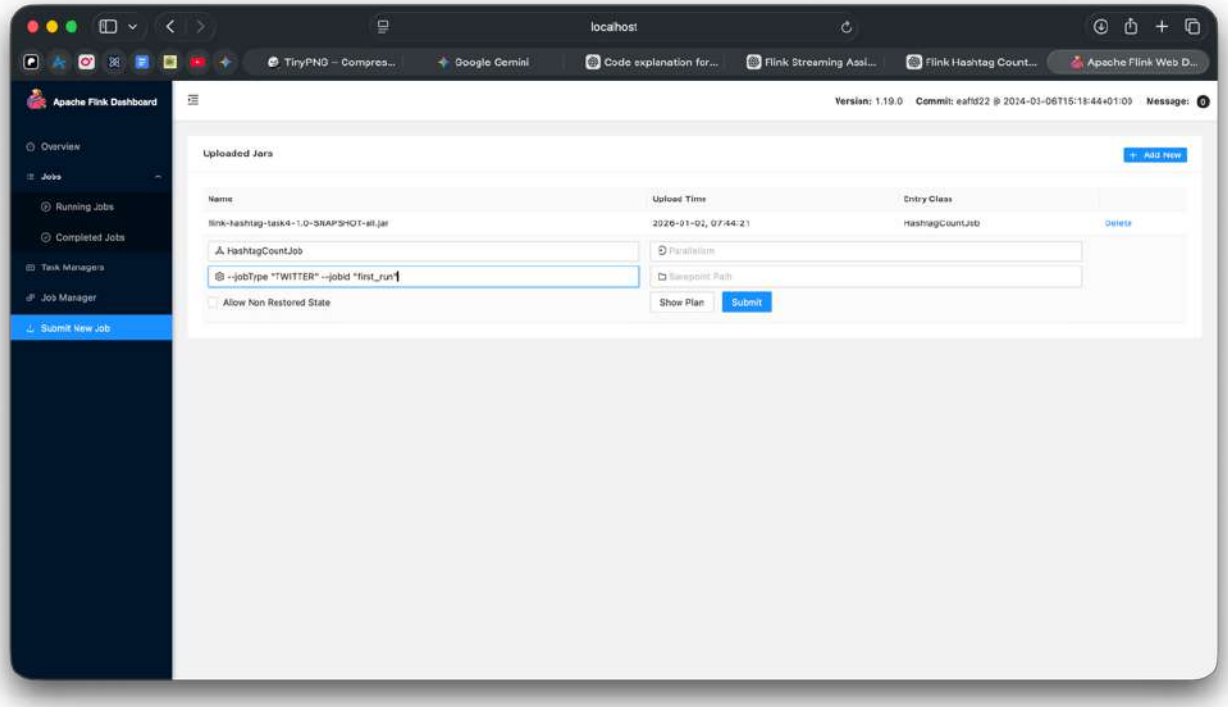
A Flink Stream Application was developed using Java to handle Twitter and TikTok datasets which would count the hashtags #GloboNews and #SAE.

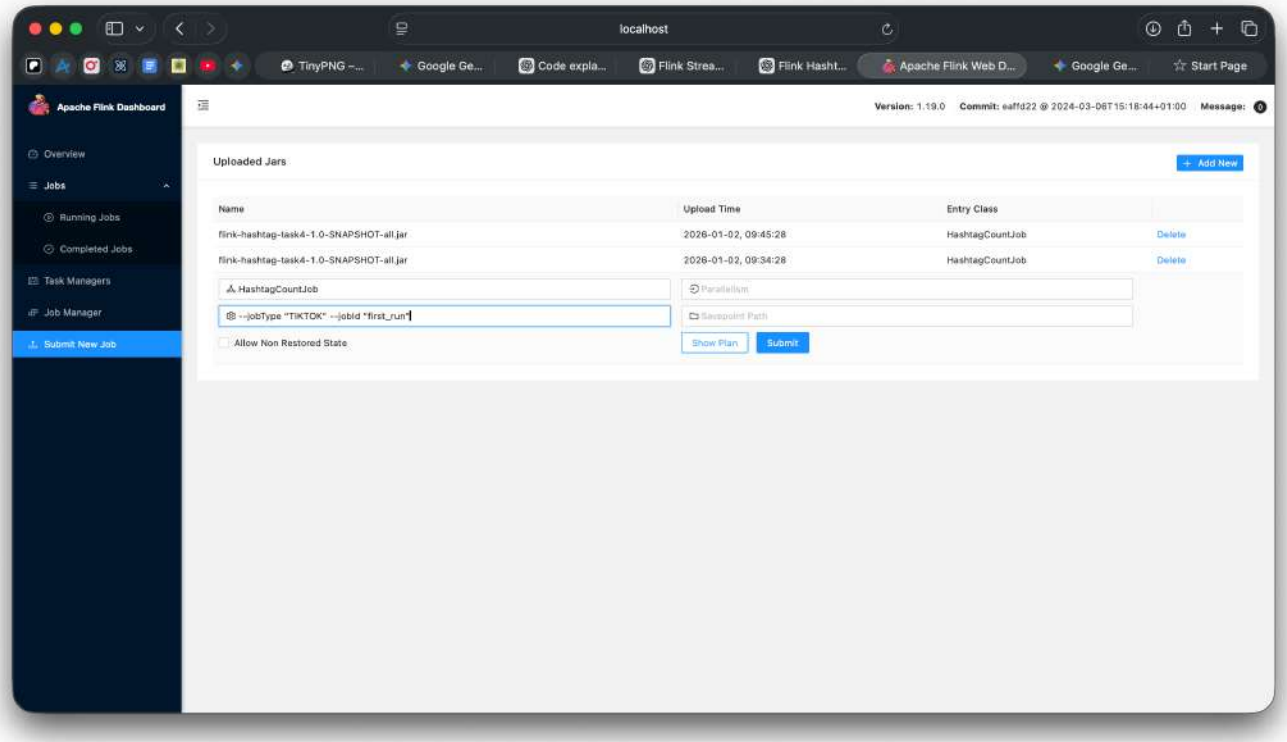
Two Kafka topics were created, one carrying Twitter CSV records and the other carrying TikTok CSV records. “kafka-console-producer” was used to push the records to the kafka topics. The Flink Application consume the messages using the “flink-connector-kafka” dependency and then process the data to output results to the PostgresSQL.

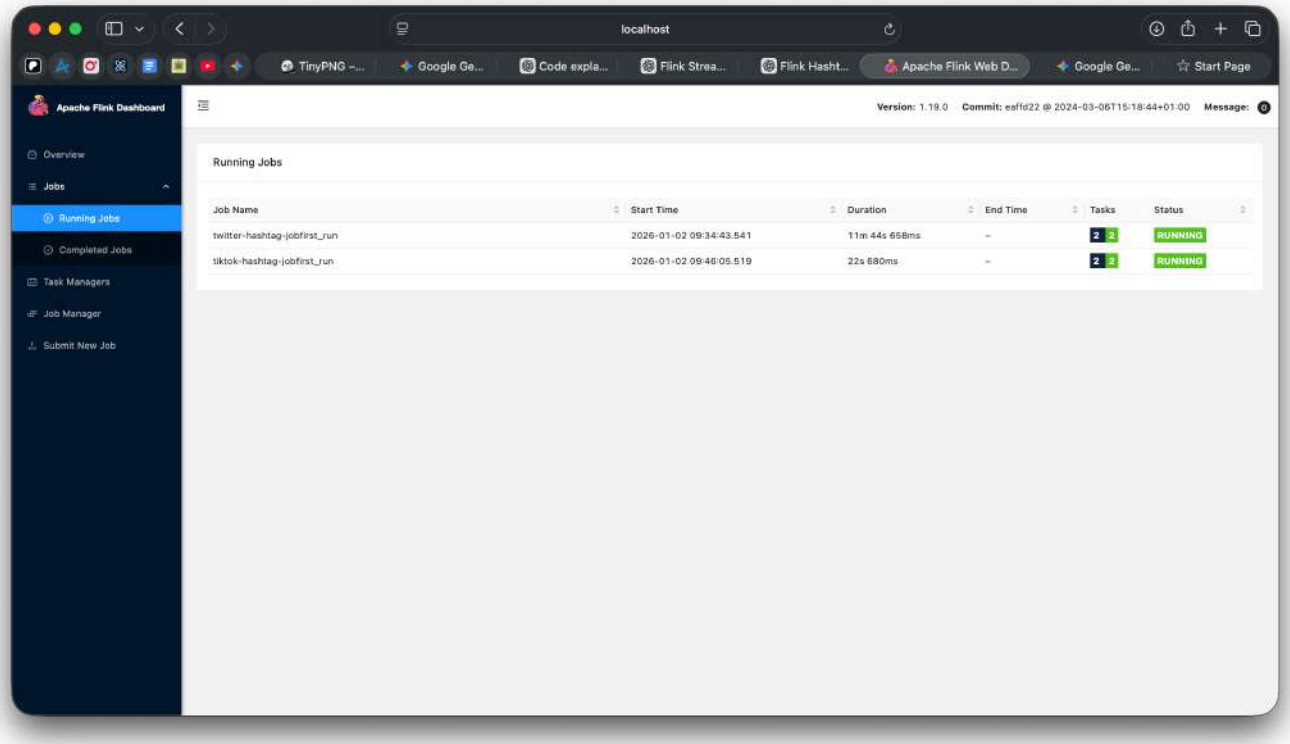
Heart of the Flink Application is “HashtagCountJob” which carries out the following duties:

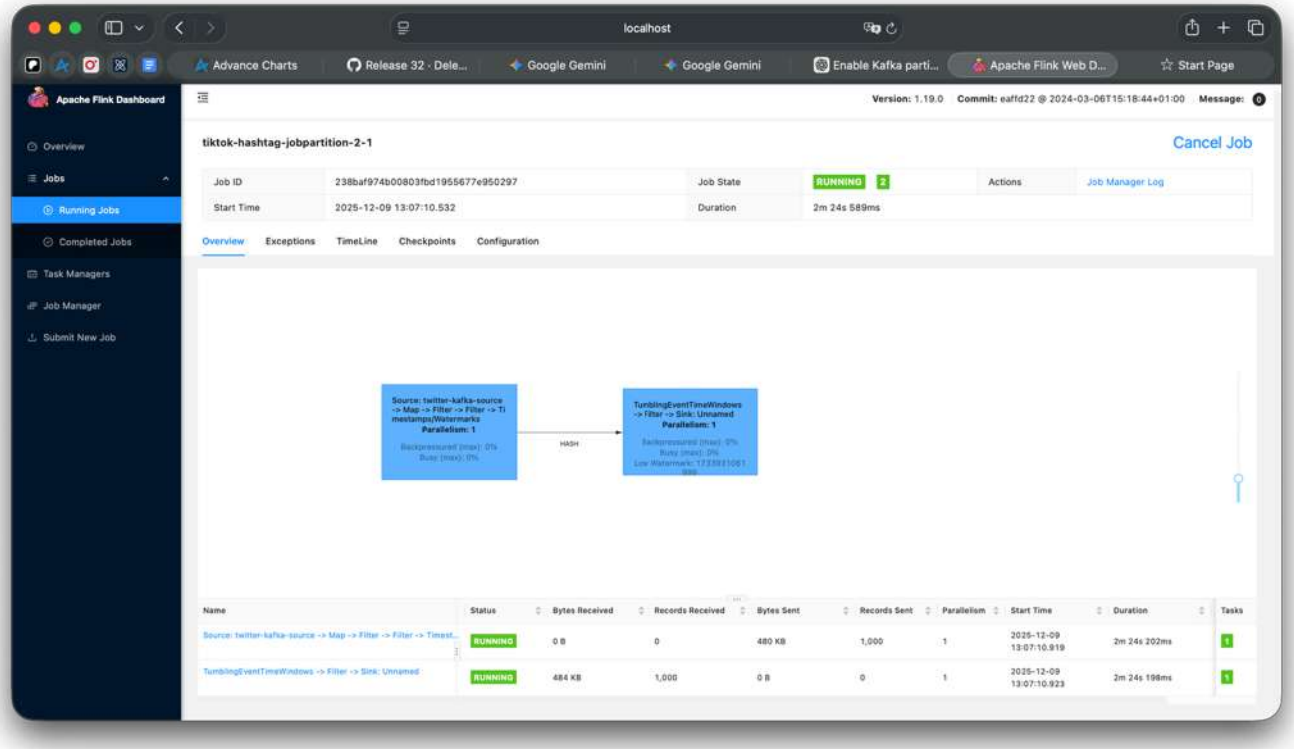
- Creates the StreamExecutionEnvironment.
- Reads command line args to decide which dataset to run on (Twitter or TikTok) and picks the correct Kafka topic, group id, hashtag, and job name.
- Then it reads Kafka messages as plain strings using the Kafka source (SimpleStringSchema) from "flink-connector-kafka" lib.
- EventParser class identifies and extract the event time using a regex and parses each string into an Event.
- It assigns event time and then HashtagCountJob assign watermarks.
- HashtagCountJob groups the events and runs a tumbling event time window.
- The window length is controlled by WINDOW_SECONDS = 15. The job uses bounded out of order watermarks with a tolerance of WATERMARK_DELAY_SECONDS = 5. Therefore, Flink allows events to arrive up to 5 seconds late before a window is considered complete. The job does not configure allowedLateness. So, events arriving after the watermark passes the window end will not be included in that window’s counts.
- For each window it outputs a Result that contains rawEventCount and hashtagEventCount.
- Finally, it saves the results to PostgreSQL using via a JDBC sink.
- The performance is captured using a small script which will record the memory and cpu usage every 1 second using docker stats command for the “flink-taskmanager-task4” container.
- Source : https://github.com/lolitha-lakshan-1/bigdata-assignment/tree/main/Task_4/flink-hashtag-count-gradle/src/main/java

Flink Job Submission






localhost
TinyPNG –...
Google Ge...
Code expla...
Flink Strea...
Flink Hasht...
Apache Flink Web D...
Google Ge...
Start Page
Apache Flink Dashboard
Version: 1.19.0 Commit: eaffd22 @ 2024-03-06T15:18:44+01:00 Message:
Overview
Jobs
Running Jobs
Completed Jobs
Task Managers
Job Manager
Submit New Job
Running Jobs
Job Name
Start Time
Duration
End Time
Tasks
Status
twitter-hashtag-jobfirst_run
2026-01-02 09:34:43.541
11m 44s 658ms
RUNNING
tiktok-hashtag-jobfirst_run
2026-01-02 09:46:05.519
22s 680ms
RUNNING

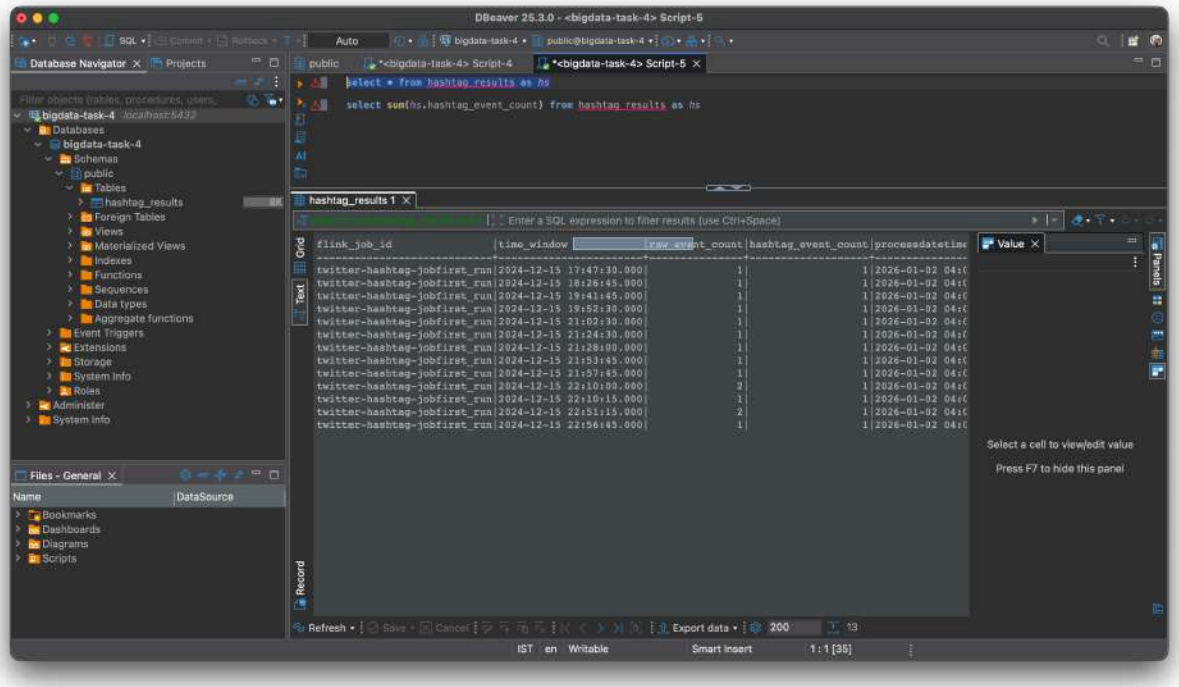

localhost
Advance Charts
Release 32 · Dele...
Google Gemini
Google Gemini
Enable Kafka parti...
Apache Flink Web D...
Start Page
Apache Flink Dashboard
Version: 1.19.0 Commit: eaffd22 @ 2024-03-06T15:18:44+01:00 Message:
Overview
Jobs
Running Jobs
Completed Jobs
Task Managers
Job Manager
Submit New Job
tiktok-hashtag-jobpartition-2-1
Cancel Job
Job ID
238baf974b00803fbd1955677e950297
Job State
RUNNING
Actions
Job Manager Log
Start Time
2025-12-09 13:07:10.532
Duration
2m 24s 589ms
Overview
Exceptions
TimeLine
Checkpoints
Configuration
HASH
Name
Status
Bytes Received
Records Received
Bytes Sent
Records Sent
Parallelism
Start Time
Duration
Tasks
RUNNING
0 B
0
480 KB
1,000
1
2025-12-09 13:07:10.919
2m 24s 202ms
RUNNING
484 KB
1,000
0 B
0
1
2025-12-09 13:07:10.923
2m 24s 198ms

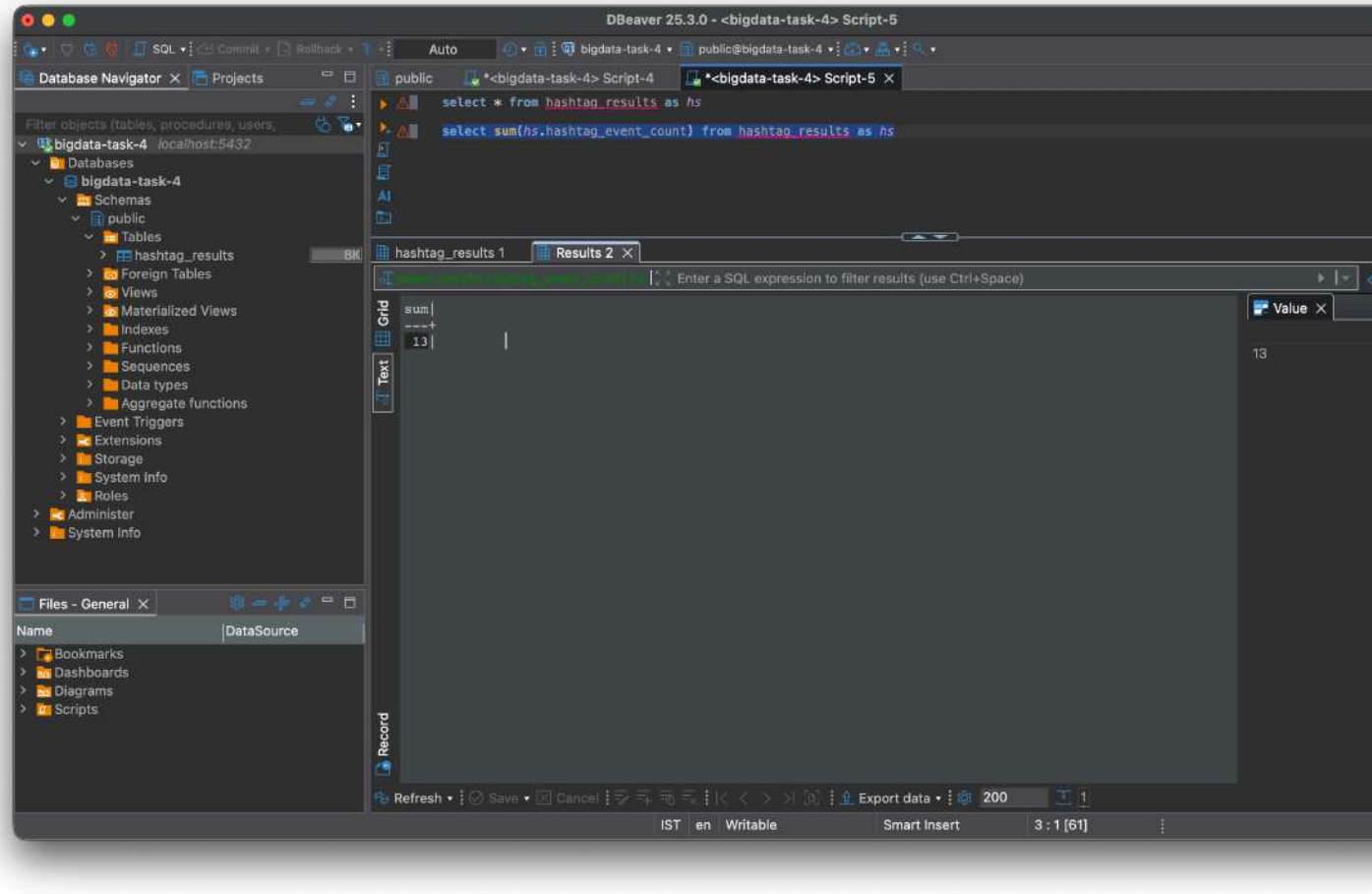
Result Accuracy and Outputs

Twitter Dataset

Expected “#GloboNews” Hashtag Count: 13

Flink Output :

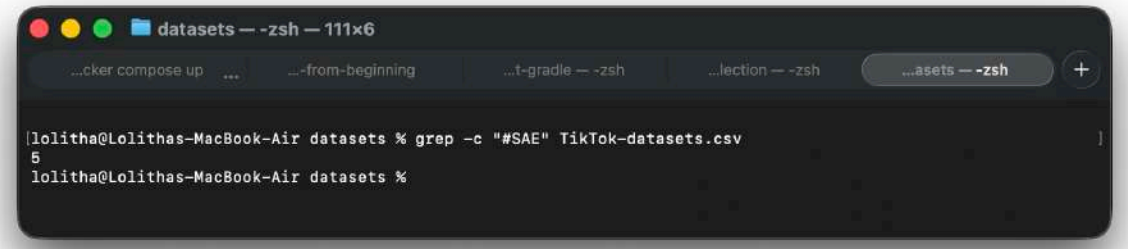




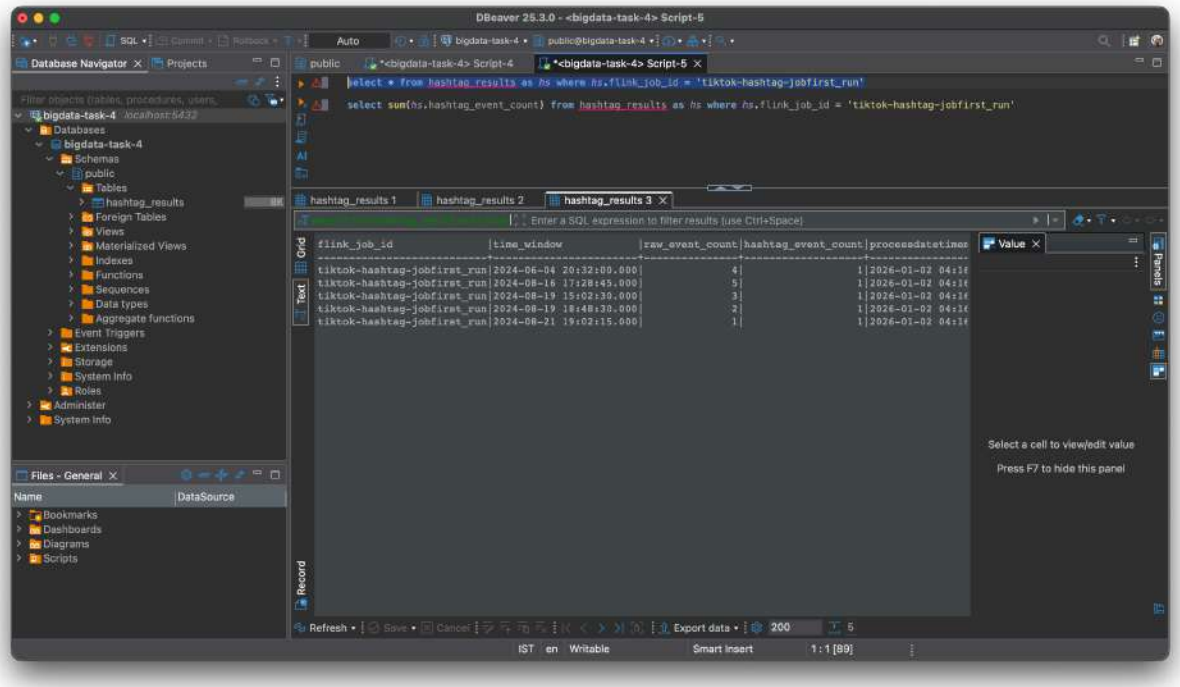
"#GloboNews" hashtag count is 13 therefore, result is accurate.

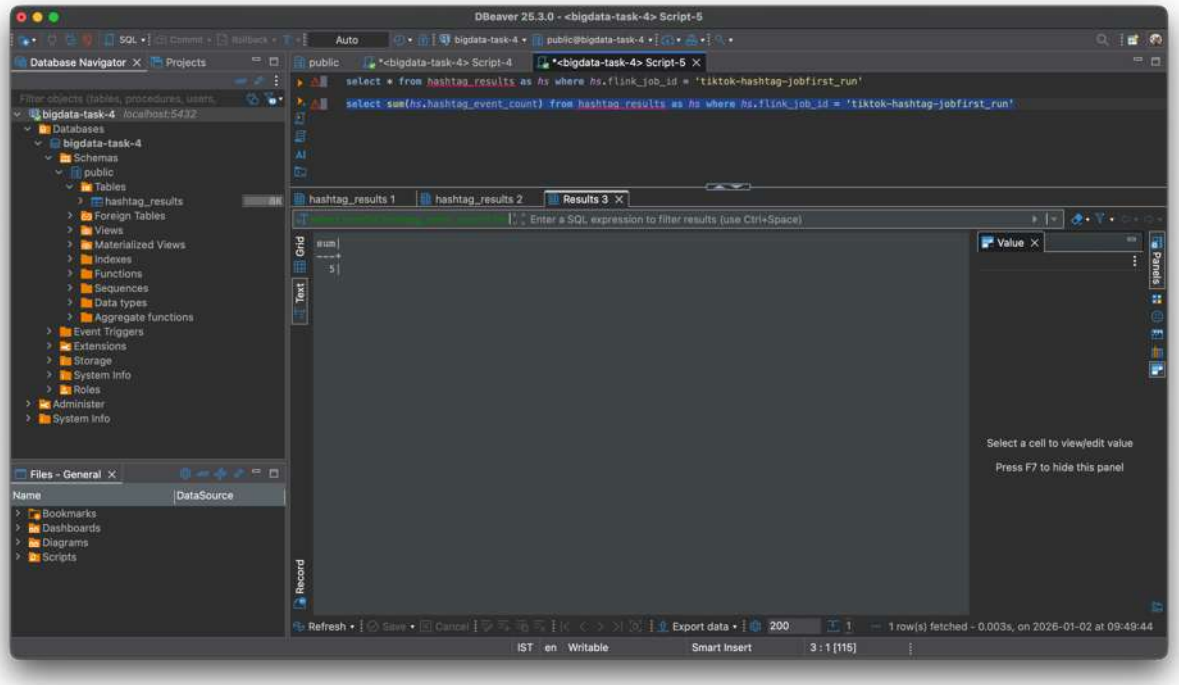
TikTok Dataset

Expected "#SAE" Hashtag Count: 5



Flink Output:

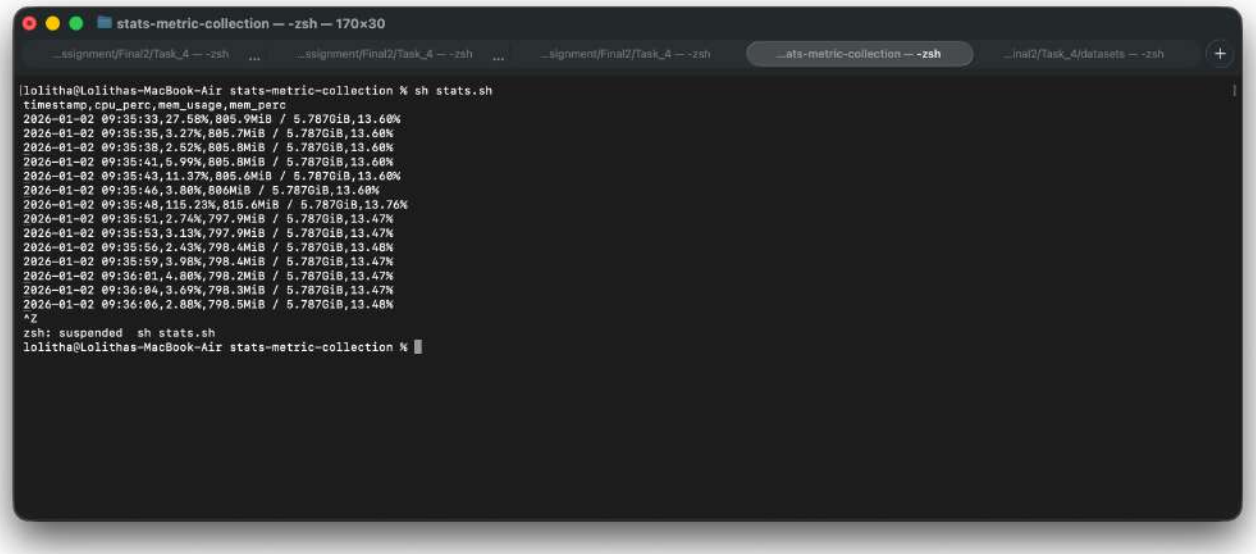




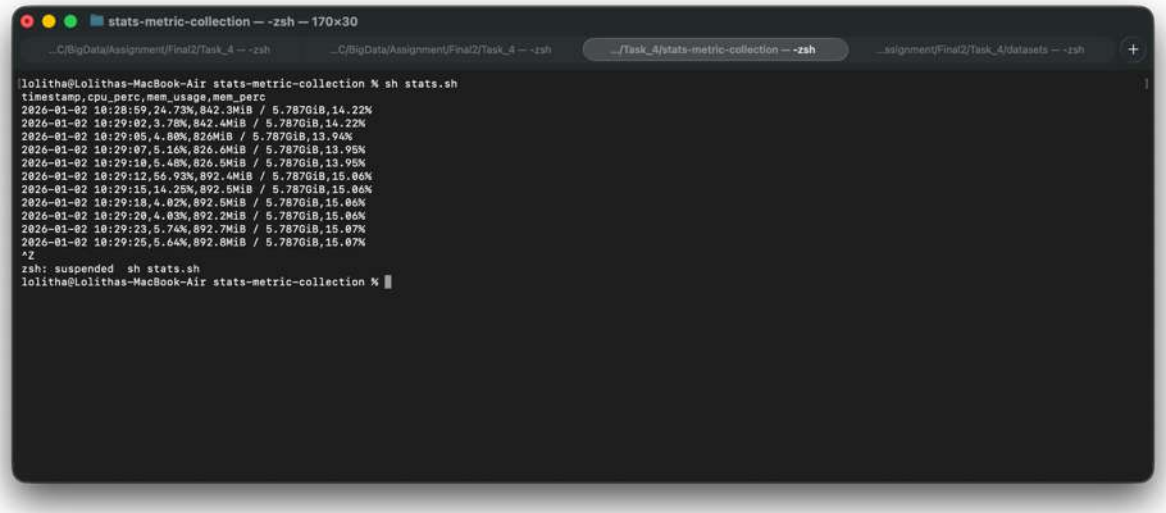
“#SAE” hashtag count is 5 therefore, result is accurate.

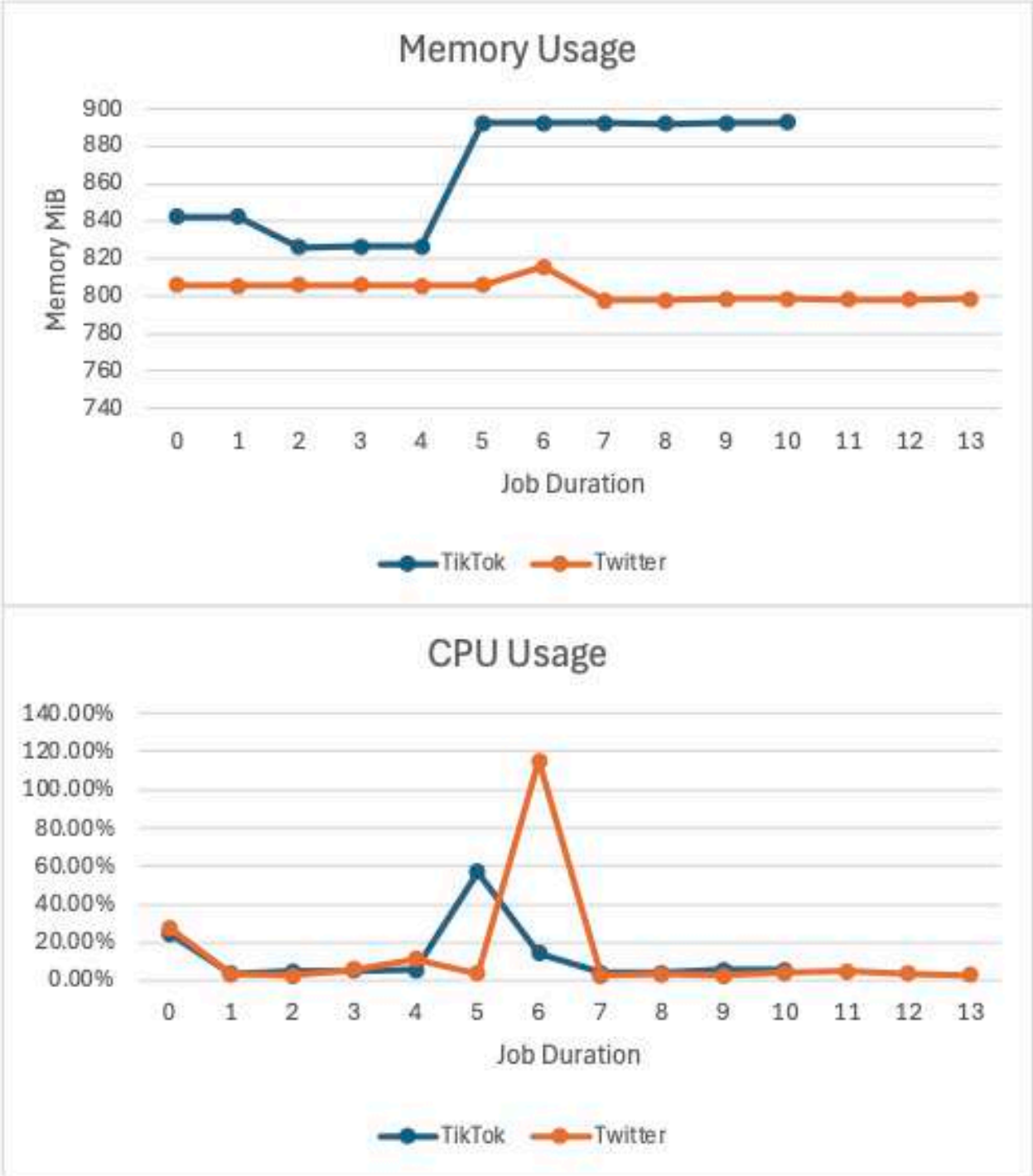
Performance

Metrics for Twitter Job



Metrics for TikTok Job





Twitter dataset size is around 1 MB and TikTok dataset size is around 500 kilobytes. Flink jobs completed in a span of 2 – 3 seconds after receiving data. Containers were restarted before each job, and it was noted that Flink allocated more base memory when it came to the tiktok job. The Tiktok job recorded a ~+60 MiB spike of memory compared to ~+20 MiB spike of Twitter Job. Furthermore, on both occasions there were CPU spikes observed, Twitter Job recorded a spike 2X compared to TikTok.

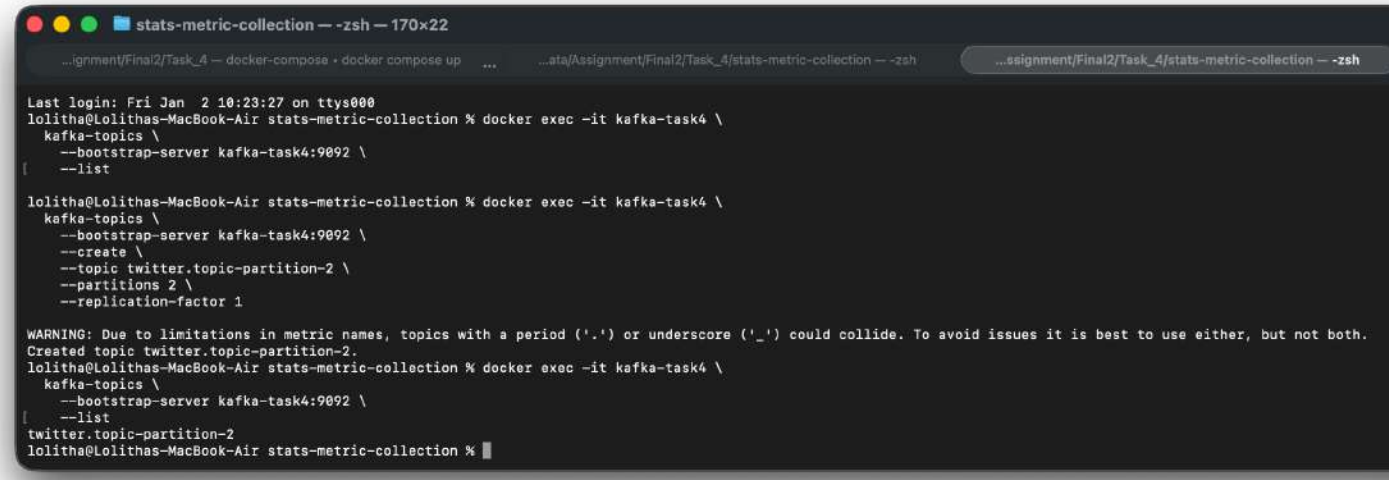
Step 4 — Scaling Experiment

Implementation

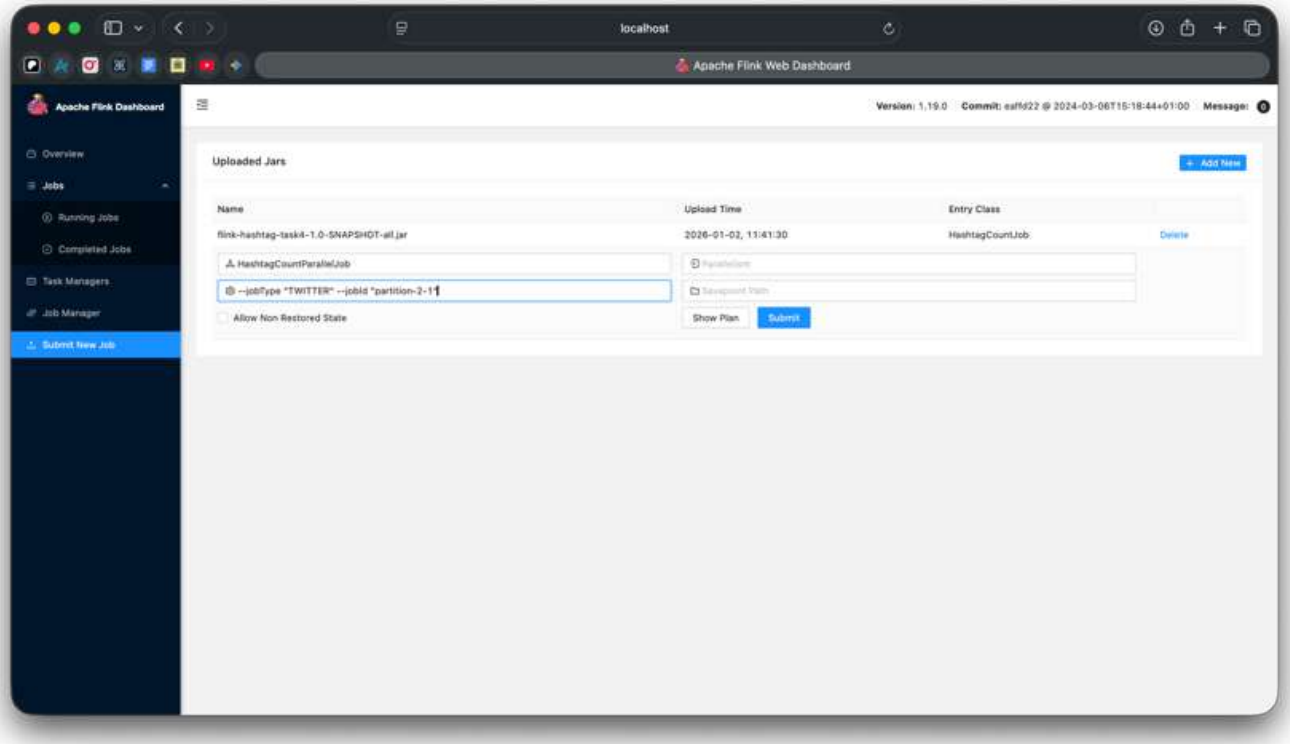
The parallel Flink job is implemented in “HashtagCountParallelJob” class which carries out the following duties:

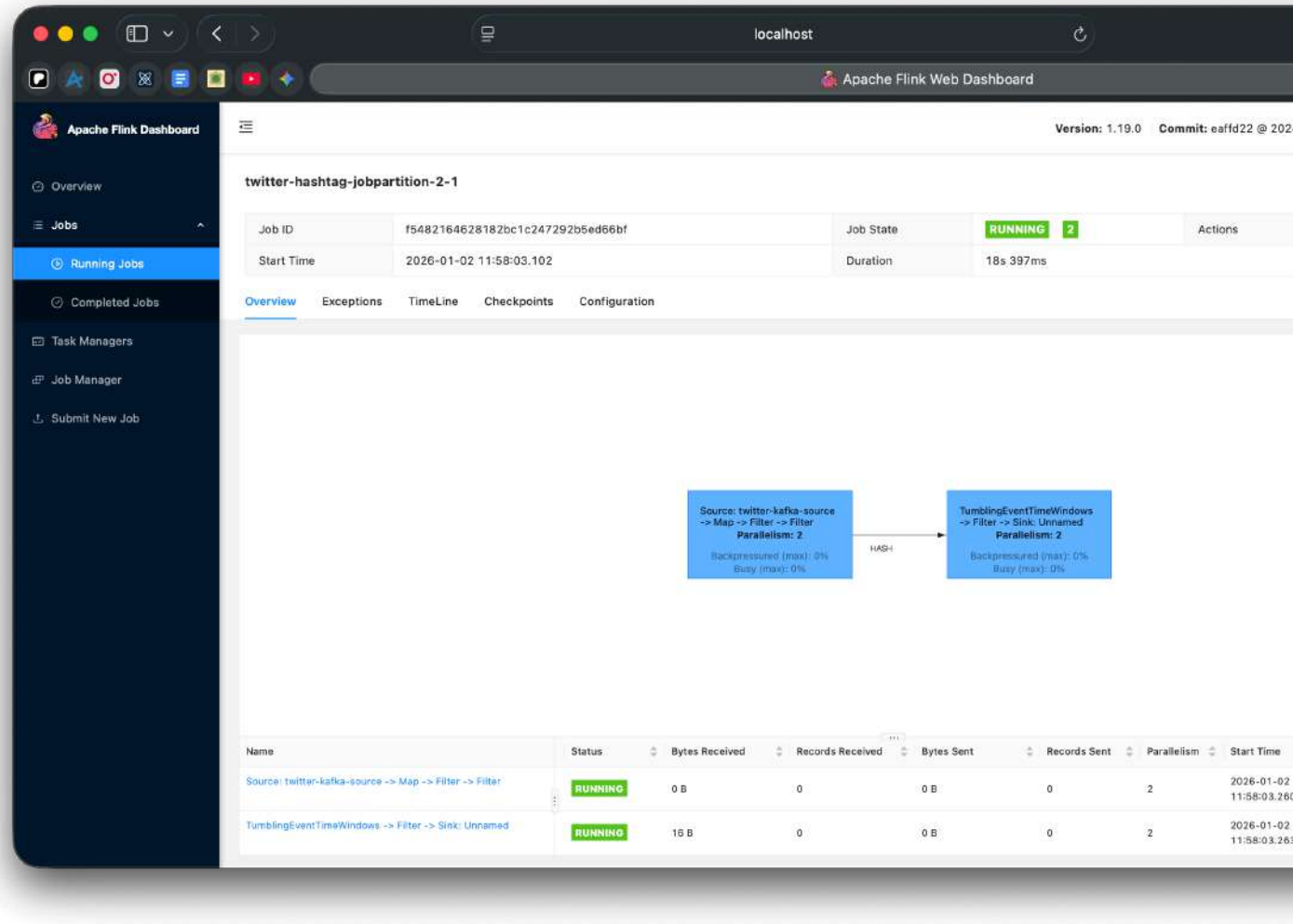
- It creates the StreamExecutionEnvironment and reads command line args to decide which dataset to run on (Twitter or TikTok) and picks the correct Kafka topic, group id, hashtag, and job name.
- It uses the two new partition topics to support higher parallel consumption. The topics are “twitter.topic-partition-2” and “tiktok.topic-partition-2” created with partitions size of 2.
- Increased Flink operator parallelism by calling environment.setParallelism(2). This allows the Kafka source to consume the two topic partitions in parallel.
- It reads Kafka messages as plain strings using the Kafka source with SimpleStringSchema from the flink-connector-kafka library and assigned watermarks at the Kafka source. On the single partition job this was done after passing the event.
- The watermark strategy is bounded out of order and uses watermark delay of 5 seconds. The timestamp assigner extracts event time by parsing each record with EventParser.parse(record) and returning event.getEventTimeMillis() and filters any invalid timestamps.
- It groups events and runs a tumbling event time window. The window length is controlled by WINDOW_SECONDS = 15. The job does not configure allowedLateness. So events arriving after the watermark passes the window end will not be included in that window’s counts.
- For each window it outputs a Result that contains rawEventCount and hashtagEventCount.
- Finally, it saves the results to PostgreSQL using a JDBC sink.
- The performance is captured using the same docker stats script which records CPU and memory usage every 1 second for the Flink containers.

Source : https://github.com/lolitha-lakshan-1/bigdata-assignment/blob/main/Task_4/flink-hashtag-count-gradle/src/main/java/HashtagCountParallelJob.java



Flink Job Submission





Now Flink Job indicates the parallelism of 2.

Result Accuracy and Outputs

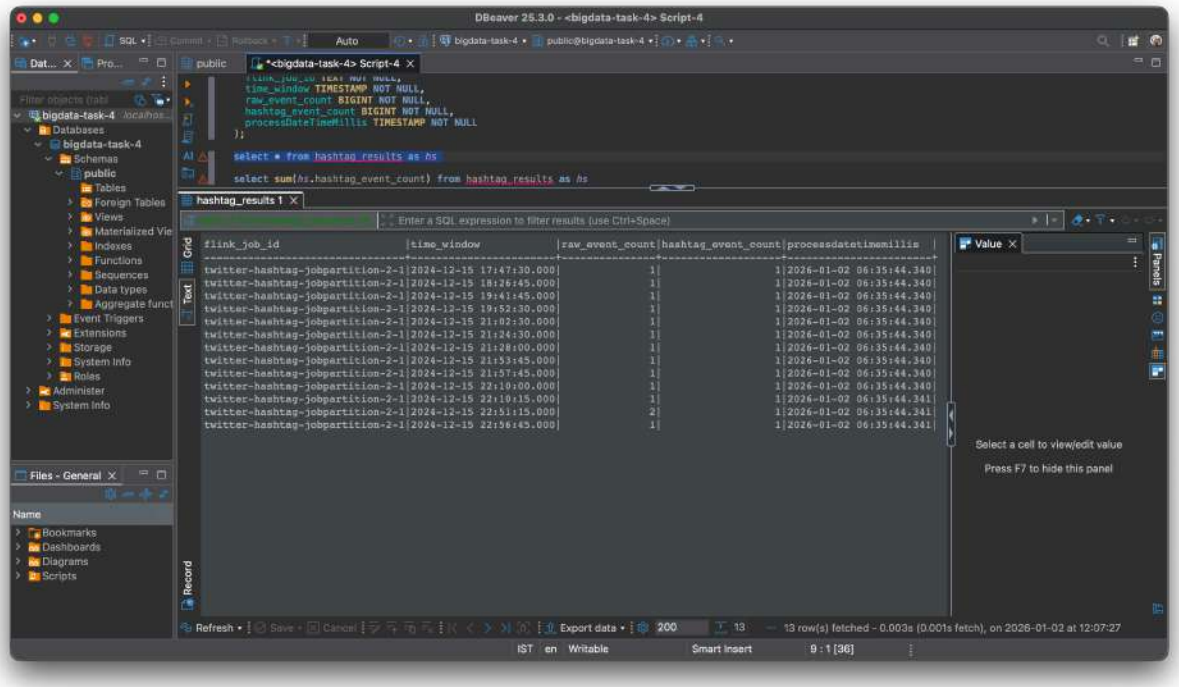
Twitter Dataset

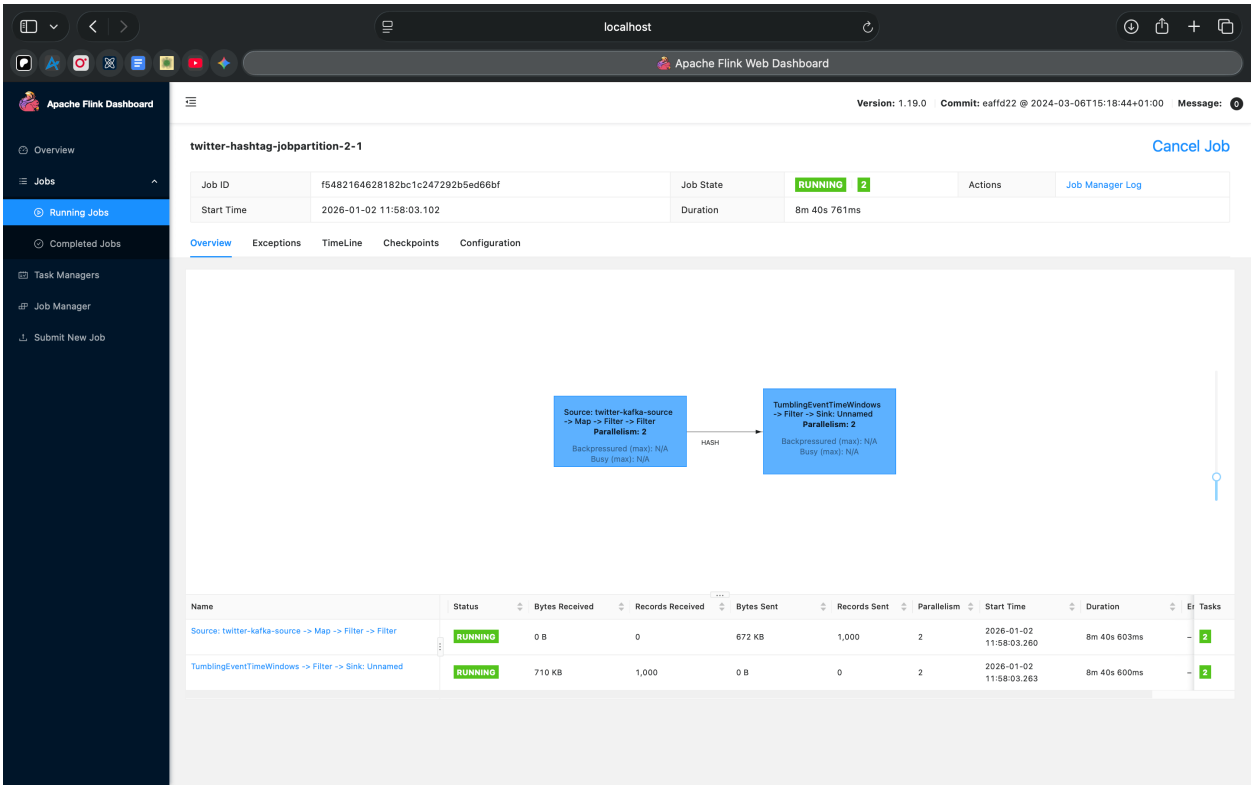
Expected “#GloboNews” Hashtag Count: 13

Even with parallelism enabled Flink returned hashtag count of 13 for “#GloboNews” hashtag.

Therefore, result is accurate.

Flink Output:



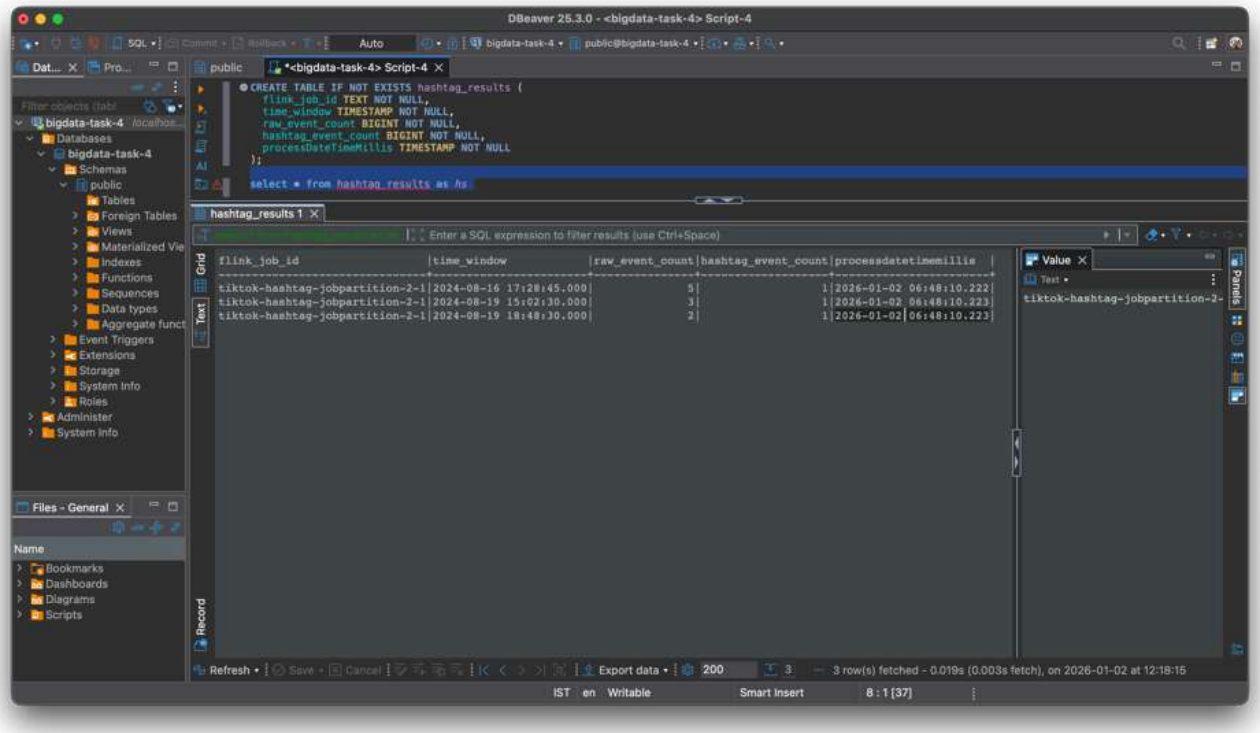


TikTok Dataset

Expected “#SAE” Hashtag Count: 5

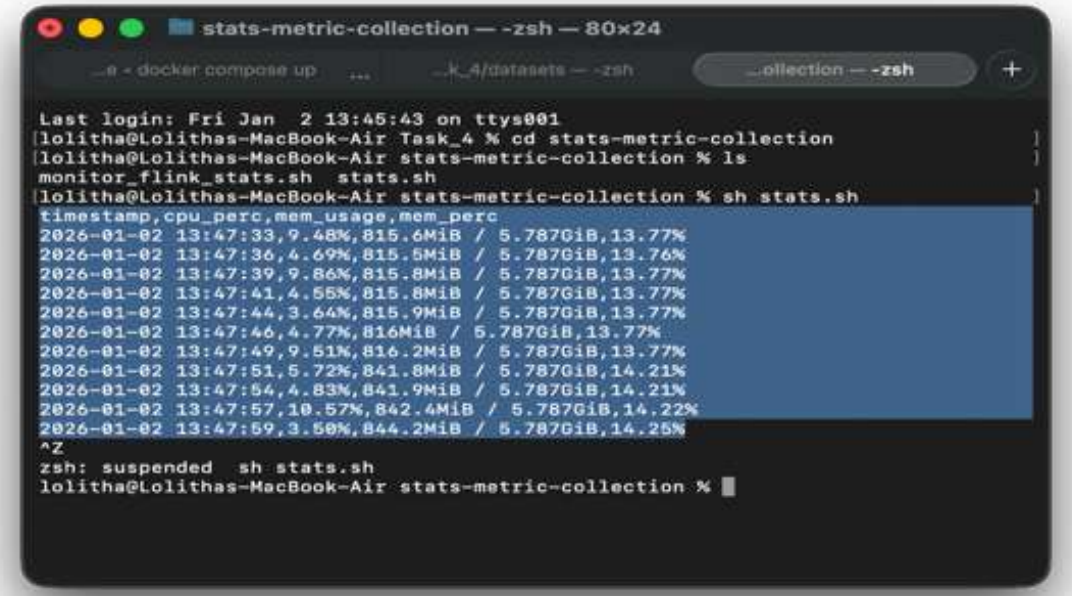
However, Flink Parallel job return “#SAE” Hashtag Count: 3 which is incorrect.

Therefore result was inaccurate .

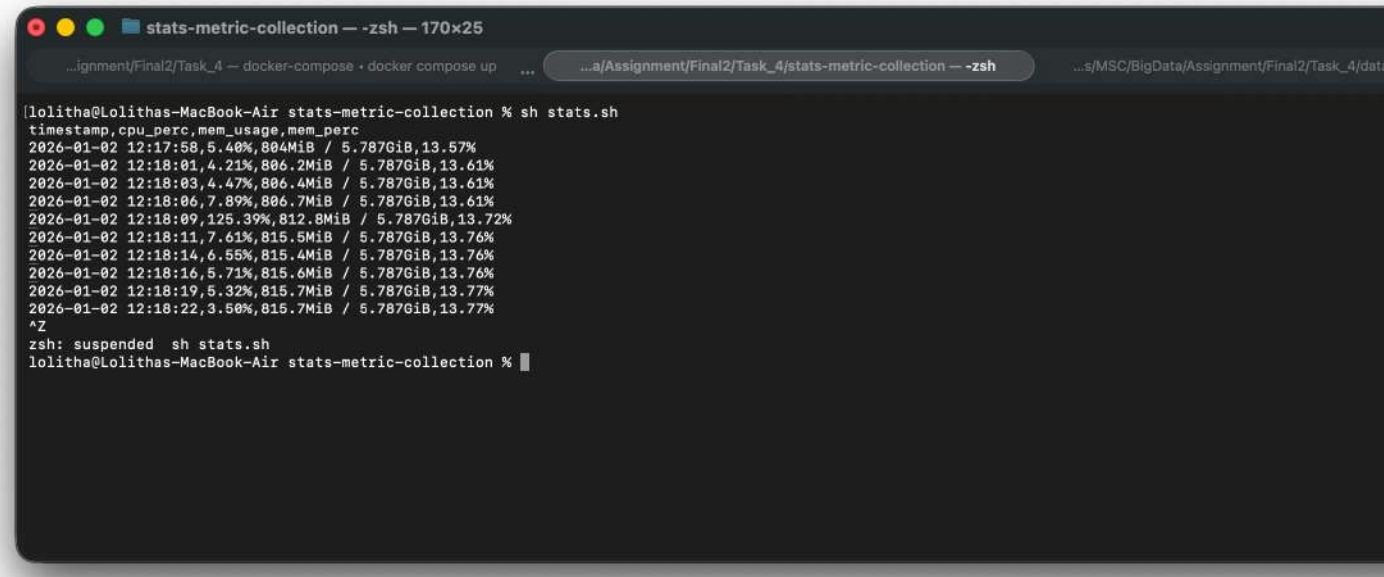


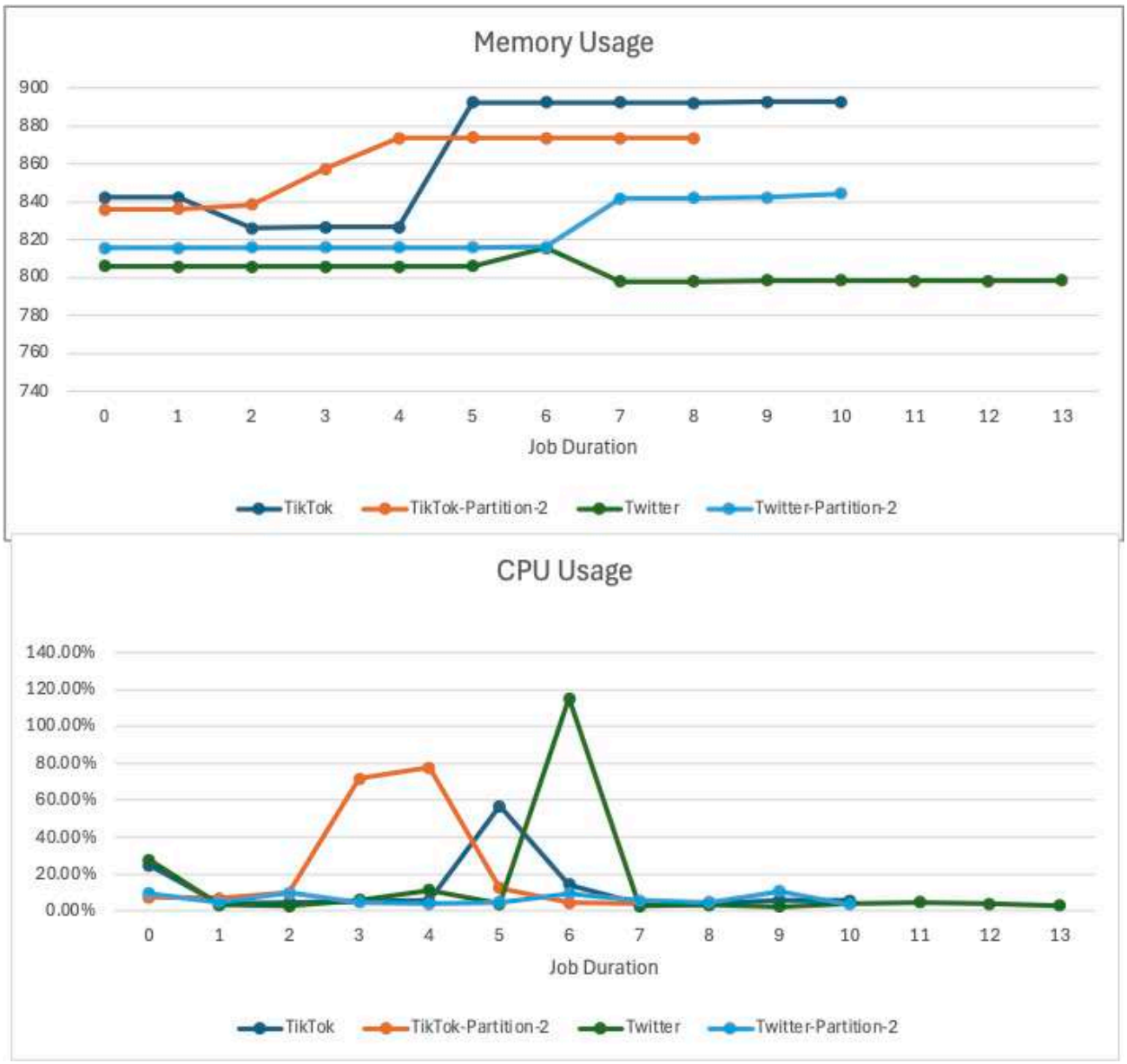
Performance

Metrics for Twitter Job



Metrics for TikTok Job





Memory usage stays mostly steady across the run. This suggests there is no memory pressure. CPU usage shows short spikes at a few time points. These spikes likely come from bursty input and window processing. Overall changes in performance are driven more by CPU bursts than by memory growth.

Task 5

Application of Big Data Technologies in the Development, Verification and Deployment of Large Language Models (LLMs)

Big Data is field of handling datasets with enormous volume, high velocity, higher variety, higher veracity and much of a potential value. The data driven decision making, which is a crucial process of almost all the industries of fields in the current world highly relies on the information retrieved and generated by processing these Big Data. This process requires key techniques and technologies such as distributed systems and parallel processing frameworks to play with the massive and diverse datasets. Large Language Models is a recently advanced Artificial Intelligence technology which can understand, generating and interacting with human languages using NLP techniques. These LLMs are called “Large” because they are trained on massive datasets with billions or even trillions of parameters to learn and understand the different traits of human languages and to generate response in more humanized manner. The integration of Big Data and Large Language Models comes where the Big Data principles and technologies become the fundamental to all the stages of LLMs, since the primary challenges in Development, Verification and Deployment of LLMs are to manage and handle the massive text corpora used in there.

The Development and Training phase of LLMs involves ingestion of petabyte-scale and diverse text datasets into the LLM pipeline. Parallel data acquisition, preprocessing and distributed training optimization can be distinguished as the two critical areas in this development and training phase. The Big Data frameworks for Petabyte-Scale data pipelines comes to the play to handle massively parallel data acquisition and preprocessing. The distributed platforms like Apache Spark, a tool for large scale data Extract-Transform-Load (ETL) process, Ray Data & Dask, a framework offers parallelization and execution of larger-than-memory and the specialized toolkits like Data Prep Kit (DPK), which is an open-source toolkit to streamline the complex process of quality assessment and transformation of

natural language and code data for LLMs can be identified as the major frameworks for handle large scale data processes[1]. The above platforms ensure the advanced data hygiene by handling the scale, complexity and fuzzy deduplication (identifying and merging approximately same records). The distributed training optimization, which implies overcoming the hardware limitations for during the development process is acquired by using the frameworks such as DeepSeed, which reduces the memory consumption with the Big Data techniques such as Zero Redundancy Optimizer (ZeRO) which partitions the model states across distributed nodes in GPU. This process makes the development and training much efficient by ensuring that each state of the model is stored only once[2].

In the Verification phase of the LLMs the Big Data methodologies are applied to test the LLMs against its biasness, potential failures and other shortcomings at a large scale. This phase uses Big Data applications for three distinct purposes: standardize comparisons for quantitative model performance, large-scale bias, fairness and robustness assessment and synthesizing data for testing and development. The verification process of LLMs needs standardized, large datasets and high-throughput frameworks. Standardized Benchmarking is one of the main tasks to get standardized tasks and datasets to quantitatively assess the reasoning, language understanding and factual accuracy of LLM models [3]. Super General Language Understanding Evaluation (SuperGLUE) is one of the most popular benchmark designed for performance evaluation in Natural Language Understanding (NLU) in LLMs. Also, the use of Big Data techniques to continuously refresh, govern and create unpolluted dataset to mitigate the data contamination issue can be seen as one of the key applications [4] . The bias evaluation process of the LLMs has now shifted to internal bias mitigation strategies, typically involve computationally intensive processes such as detecting and normalizing sensitive attributes within the model [5]. These strategies require distributed systems to handle the scaled analysis of the model's internal data flow.

The final phase of LLM lifecycle is the Deployment phase, typically known as LLM Deployment and Inference (LLMOps). This phase gives the Big Data systems challenge of

managing the high cost and low-latency demands of large models. The optimization of the GPU resources is a crucial need to manage the cost at deployment and there comes the use of various Big Data techniques. One of the key techniques used for this purpose is PagedAttention which uses a core Big Data resource optimization principle of effectively eliminating fragmentation to boost the throughput significantly [6]. This technique mainly applies the distributed memory management theory to LLM's Key-Value cache to partition the cache to allocate the GPU only when needed. Also, GPU utilization is maximized by processing incoming requests in parallel using Continuous Batching techniques which uses the parallel processing principles in Big Data. Furthermore, for the multi-node performance of the LLMs, these systems use Distributing Serving Architectures which share cache as shared to handle highly volatile distributed data resources at the deployment phase. However, still there are some ethical and big data challenges such as, LLMs memorizing personal information and the black box nature of LLM responses which are to be handled in deployment phase.

The above comprehensive summary of the applications on Big Data technologies and techniques throughout all the phases in LLM life cycle shows that, Big Data is not just a supplement to LLMs, but those theories are the foundation for the existence of the LLMs since they are obviously "Large" in nature and work with enormous amounts of data at each phase. In brief, the development phase relies on the distributed processing frameworks and system optimization techniques to execute advanced data hygiene and to manage massive memory demands, the verification phase transforms into a Big Data problem of utilizing high-throughput distributed systems and execute internal bias mitigation strategies to evaluate the LLMs quantitatively and finally, the deployment phase mitigate the challenges of feasibility by utilizing the distributed systems concepts such as memory virtualization and dynamic scheduling principles. Therefore, it is evident that many of Big Data concepts do a crucial intervention through Development, Verification and Deployment phases of the LLMs.

References:

[1] D. Wood et al., “Data-Prep-Kit: getting your data ready for LLM application development,” Nov. 13, 2024, arXiv: arXiv:2409.18164. doi: 10.48550/arXiv.2409.18164.

[2] S. Rajbhandari, J. Rasley, O. Ruwase, and Y. He, “ZeRO: Memory Optimizations Toward Training Trillion Parameter Models,” May 13, 2020, arXiv: arXiv:1910.02054. doi: 10.48550/arXiv.1910.02054.

[3] S. Chen et al., “Recent Advances in Large Langauge Model Benchmarks against Data Contamination: From Static to Dynamic Evaluation,” Sept. 29, 2025, arXiv: arXiv:2502.17521. doi: 10.48550/arXiv.2502.17521.

[4] Y. Cheng, Y. Chang, and Y. Wu, “A Survey on Data Contamination for Large Language Models,” June 05, 2025, arXiv: arXiv:2502.14425. doi: 10.48550/arXiv.2502.14425.

[5] A. Abhishek, L. Erickson, and T. Bandopadhyay, “BEATS: Bias Evaluation and Assessment Test Suite for Large Language Models,” Mar. 31, 2025, arXiv: arXiv:2503.24310. doi: 10.48550/arXiv.2503.24310.

[6] W. Kwon et al., “Efficient Memory Management for Large Language Model Serving with PagedAttention,” Sept. 12, 2023, arXiv: arXiv:2309.06180. doi: 10.48550/arXiv.2309.06180.