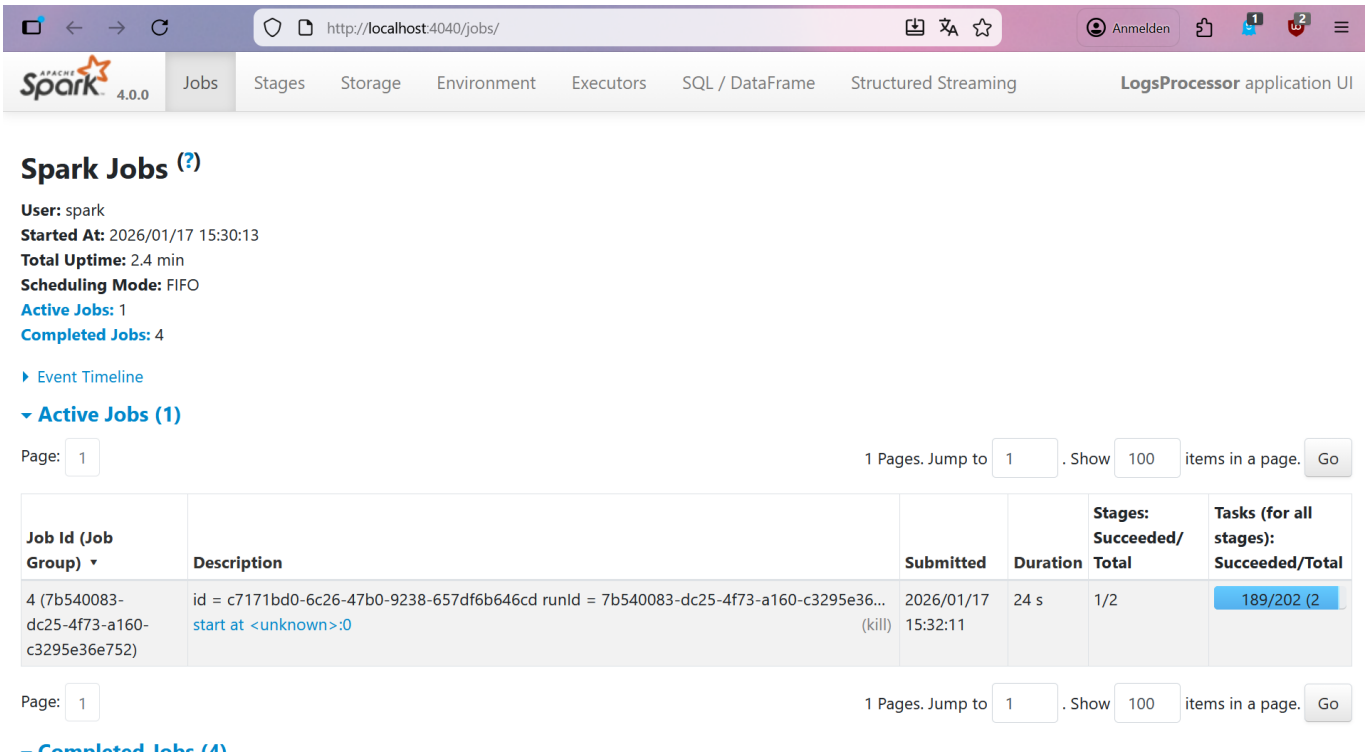
# Activity 1

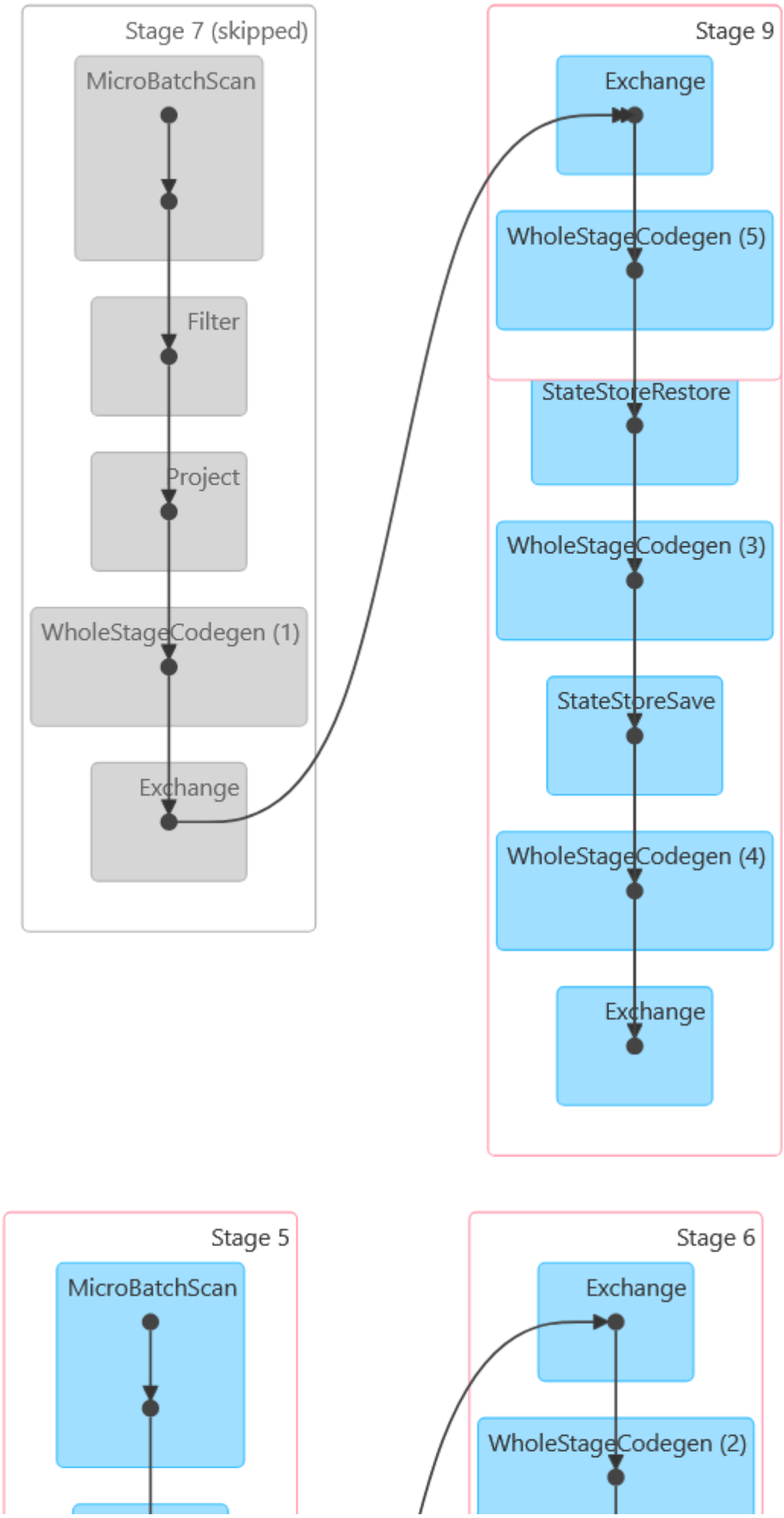
## Part 1

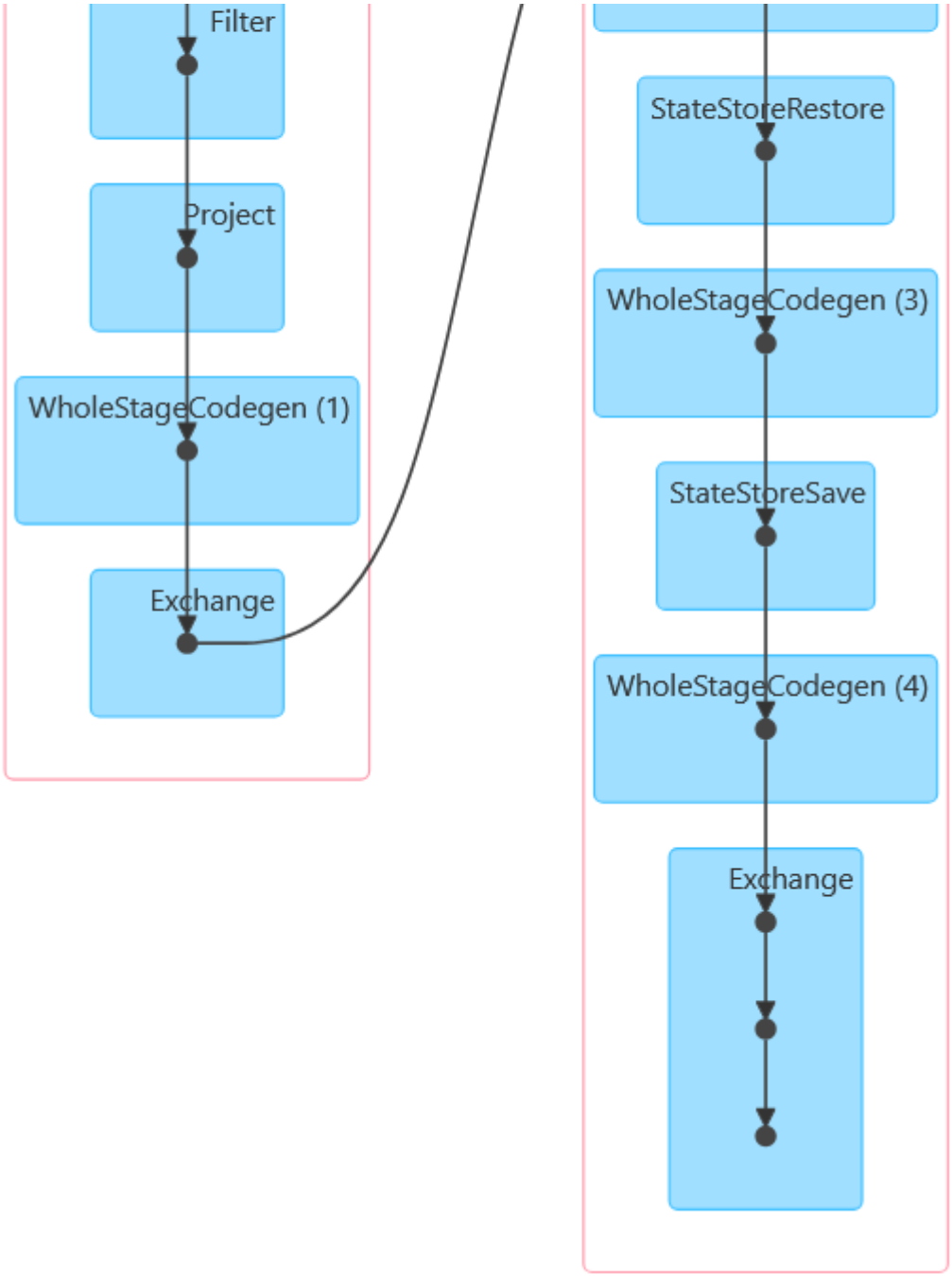



## Part 2

a

▾ DAG Visualization



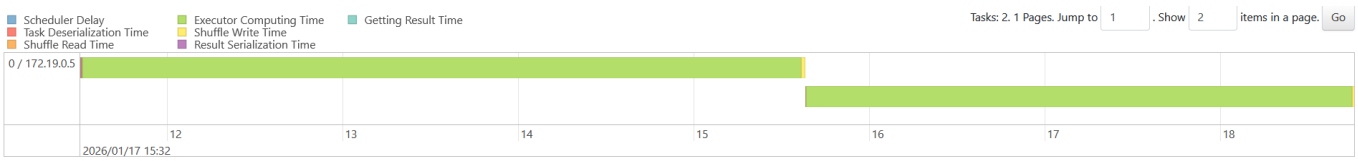



One with skip and one without it.

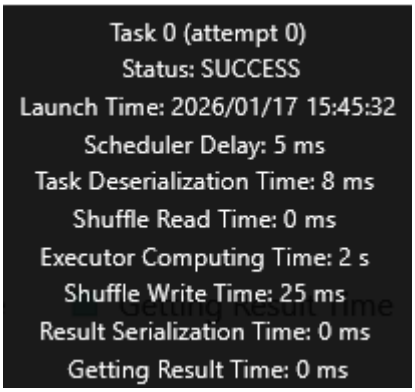
b

| Shuffle Read | Shuffle Write |
|---|---|
| 12.6 KiB | |
| | 12.6 KiB |
| 7.9 KiB | |
| 12.6 KiB | 7.9 KiB |
| 12.6 KiB | |
| | 12.6 KiB |
| 7.9 KiB | |
| 12.7 KiB | 7.9 KiB |

shuffle write found in a stage




Example of how the Tasks then look like if you hover over just one of them as I found it interesting.




c

For right now we only have one worker and one driver, so the worker has all the work. But from what I understand this is how it should be for one worker and one driver.

| | RDD Blocks | Storage Memory | Disk Used | Cores | Active Tasks | Failed Tasks | Complete Tasks | Total Tasks | Task Time (GC Time) | Input | Shuffle Read | Shuffle Write | Excluded |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Active(2)** | 0 | 17.7 MiB / 848.3 MiB | 0.0 B | 1 | 2 | 0 | 11770 | 11772 | 33 min (11 s) | 0.0 B | 774.9 KiB | 484.6 KiB | 0 |
| **Dead(0)** | 0 | 0.0 B / 0.0 B | 0.0 B | 0 | 0 | 0 | 0 | 0 | 0.0 ms (0.0 ms) | 0.0 B | 0.0 B | 0.0 B | 0 |
| **Total(2)** | 0 | 17.7 MiB / 848.3 MiB | 0.0 B | 1 | 2 | 0 | 11770 | 11772 | 33 min (11 s) | 0.0 B | 774.9 KiB | 484.6 KiB | 0 |

**Executors**

Show 20 entries Search:

| Executor ID | Address | Status | RDD Blocks | Storage Memory | Disk Used | Cores | Active Tasks | Failed Tasks | Complete Tasks | Total Tasks | Task Time (GC Time) | Input | Shuffle Read | Shuffle Write | Logs | Thread Dump | Heap Histogram | Add Time | Remove Time |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| driver | b9ec450fed29:42325 | Active | 0 | 8.9 MiB / 434.4 MiB | 0.0 B | 0 | 0 | 0 | 0 | 0 | 17 min (1 s) | 0.0 B | 0.0 B | 0.0 B | | Thread Dump | Heap Histogram | 2026-01-17 16:30:15 | - |
| 0 | 172.19.0.5:46657 | Active | 0 | 8.9 MiB / 413.9 MiB | 0.0 B | 1 | 2 | 0 | 11770 | 11772 | 16 min (10 s) | 0.0 B | 774.9 KiB | 484.6 KiB | stdout stderr | Thread Dump | Heap Histogram | 2026-01-17 16:30:22 | - |

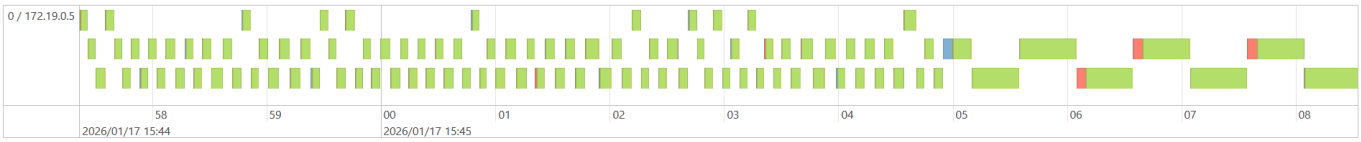
# Part 3

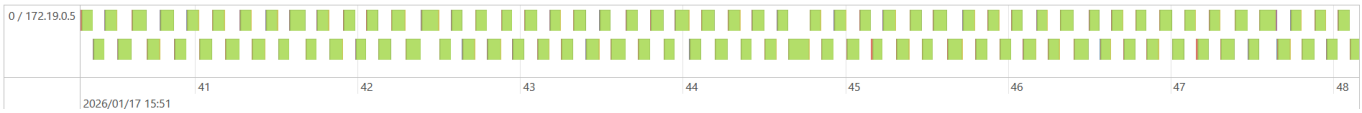
## 1

Stage 168 and 113 for me with 35 seconds.

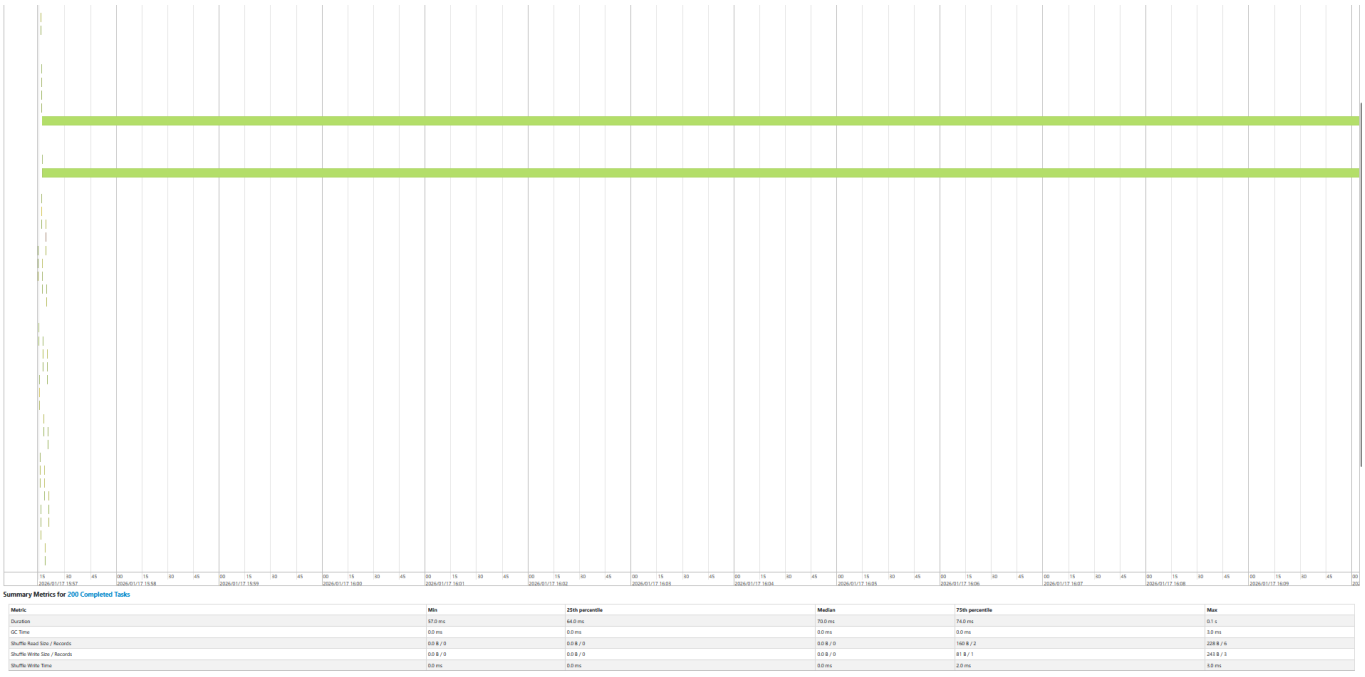
| Stage Id | Description | Submitted | Duration | Tasks: Succeeded/Total | Input | Output | Shuffle Read | Shuffle Write |
|---|---|---|---|---|---|---|---|---|
| 168 | id = c7171bd0-6c26-47b0-9238-657df6b646cd runId = 7b540083-dc25-4f73-a160-c3295e36e752 batch = 33 start at <unknown>:0 +details | 2026/01/17 15:51:40 | 35 s | 200/200 | | | 12.6 KiB | 8.0 KiB |
| 113 | id = c7171bd0-6c26-47b0-9238-657df6b646cd runId = 7b540083-dc25-4f73-a160-c3295e36e752 batch = 22 start at <unknown>:0 +details | 2026/01/17 15:44:57 | 35 s | 200/200 | | | 12.6 KiB | 7.9 KiB |

From what I can tell our biggest bottlneck right now is the limited parallelism.




These two pictures are event Timelines from 2 of the slowest ones.



This one was done with the same amount of tasks in less then half of the time.

But as you can see it has like one 2 long bars running at the same time and lots of short ones scattered in the beginning.

We only added 1 Executor Core, 1 Executor and 1 GB of memory.

2

| Storage Memory |
| --- |
| 11 MiB / 434.4 MiB |
| 11 MiB / 413.9 MiB |

So in total it is right now using 22.7 MiB out of 848.3 MiB which are available to it

3

In Spark Structured Streaming, performance and scalability are about how well the system processes streaming data and how it handles growing workloads.

Structured Streaming usually works with micro-batches, meaning incoming data is processed in small chunks. Smaller batches give lower latency but more overhead, while larger batches increase throughput but also delay results.

Performance depends on how efficiently Spark uses cluster resources like CPU and memory. Operations such as joins, aggregations, and stateful processing can slow things down if they require a lot of memory or shuffling.

Scalability means Spark can handle higher data rates by adding more executors or increasing parallelism. However, bottlenecks like network shuffles, state size, and slow sinks can limit how much it actually scales.

# Activity 2

change to the spark call

```
spark-submit \
  --master spark://spark-master:7077 \
  --packages org.apache.spark:spark-sql-kafka-0-10_2.13:4.0.0 \
  --num-executors 2 \
  --executor-cores 2 \
  --executor-memory 2G \
  /opt/spark-apps/spark_structured_streaming_logs_processing.py
```

Changes to docker file in Exercise 3

```
  spark-worker:
    image: bitnamilegacy/spark:4.0.0
    depends_on:
      - spark-master
    environment:
      - SPARK_MODE=worker
      - SPARK_MASTER_URL=spark://spark-master:7077
      - SPARK_WORKER_CORES=4
      - SPARK_WORKER_MEMORY=4G
    networks:
      - streaming-net
    deploy:
      replicas: 1
      resources:
        limits:
          memory: 4095M
          cpus: '4'
```

Changes to Docker file in Load-generator

```
services:
  generator:
    image: adrianovogel/hgb-load-gen:latest
    extra_hosts:
      - "host.docker.internal:host-gateway"
    environment:
      - KAFKA_BROKER=host.docker.internal:9095
      - KAFKA_TOPIC=logs
      - TARGET_RPS=10000
      - ADDITIONAL_TERM=crash
      - ADDITIONAL_TERM_RATE=100
    deploy:
      replicas: 4  # This enables to run more instances (containers)
      resources:
        limits:
          memory: 2048M
          cpus: '1'
    networks:
      - streaming-net
```

# Monitoring

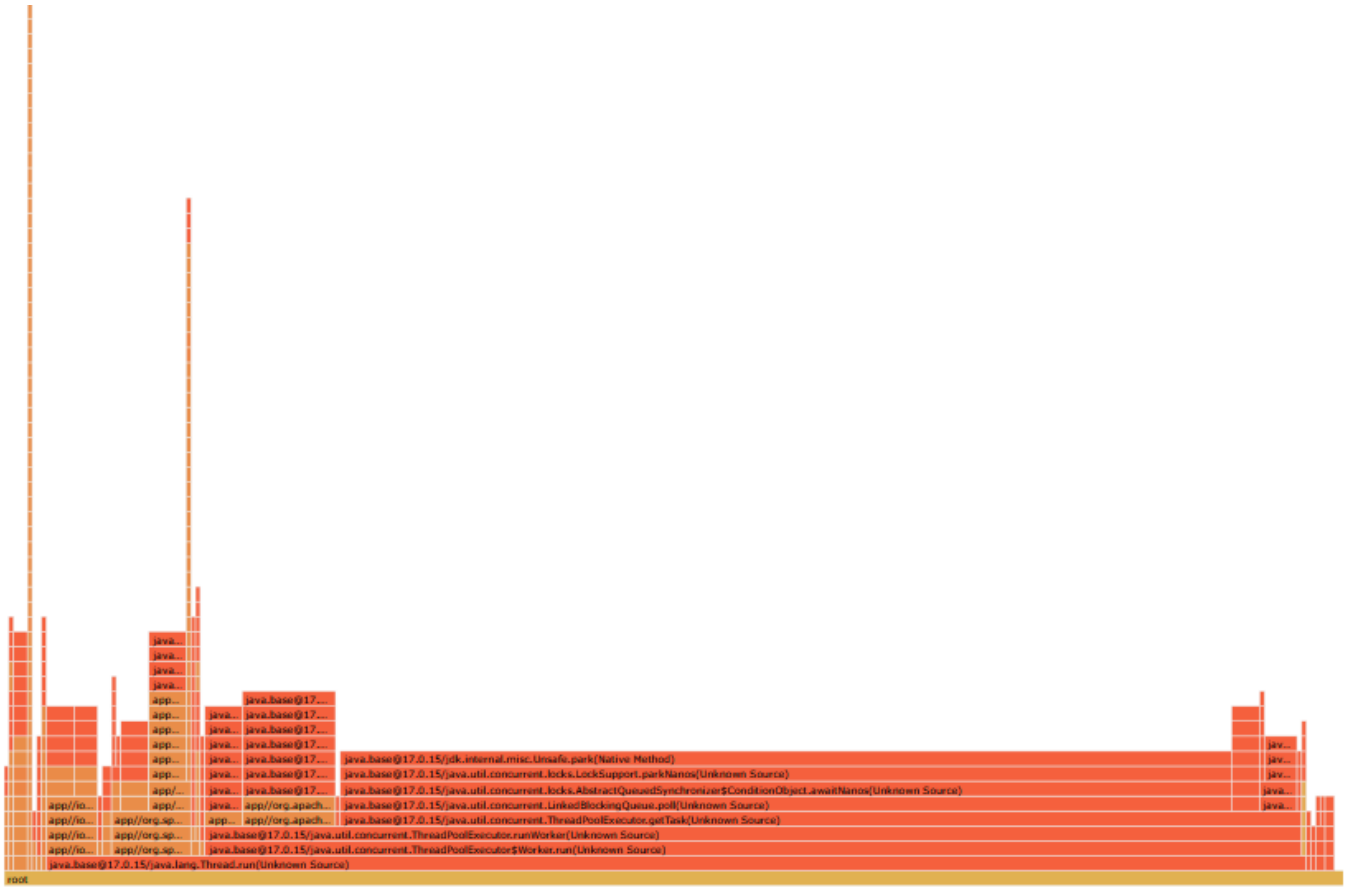
## Input Rate vs. Process Rate

| | |
|---|---|
| 35671.60 | 61106.03 |

I think I do not have enough avg Input per second right now but it at least shows me that it works (I was a little bit scared that this all would be to much for my laptop, that is why it might be like that.)

## The Executor Tab

The lovely flame chart from the Threat dump

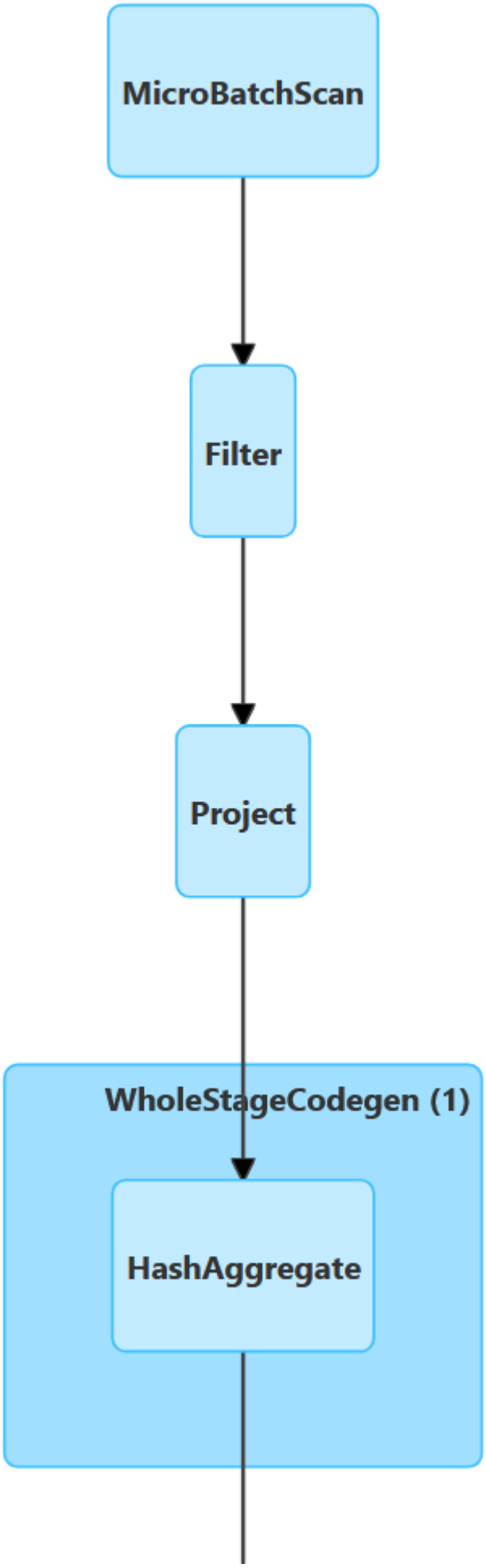


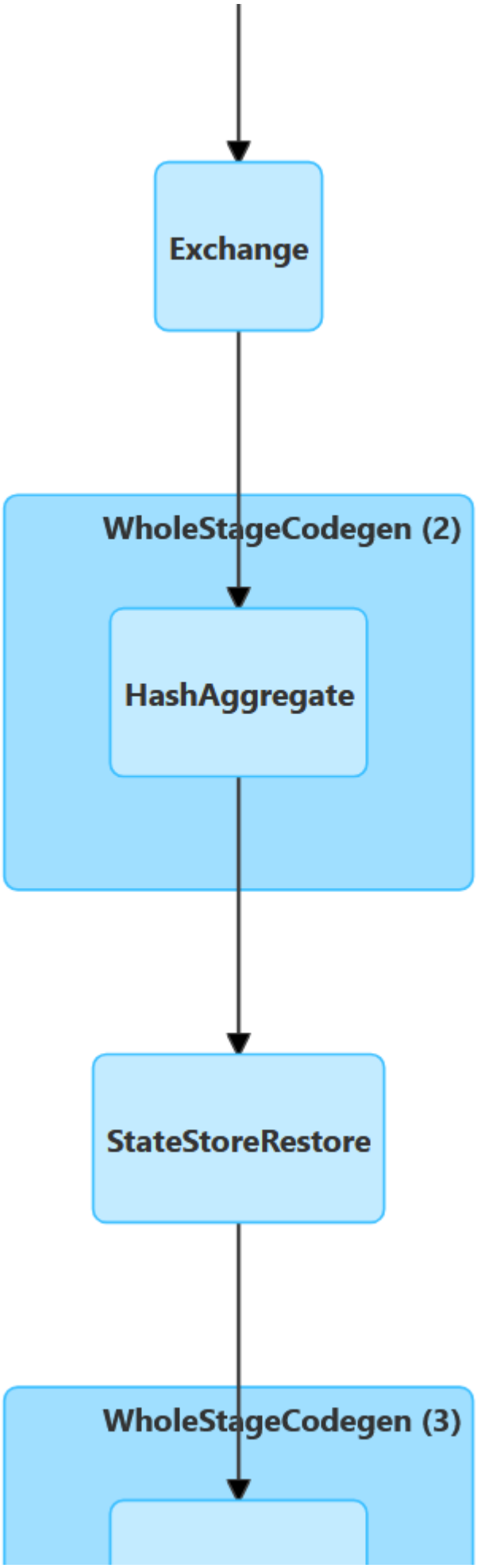

And the tasks

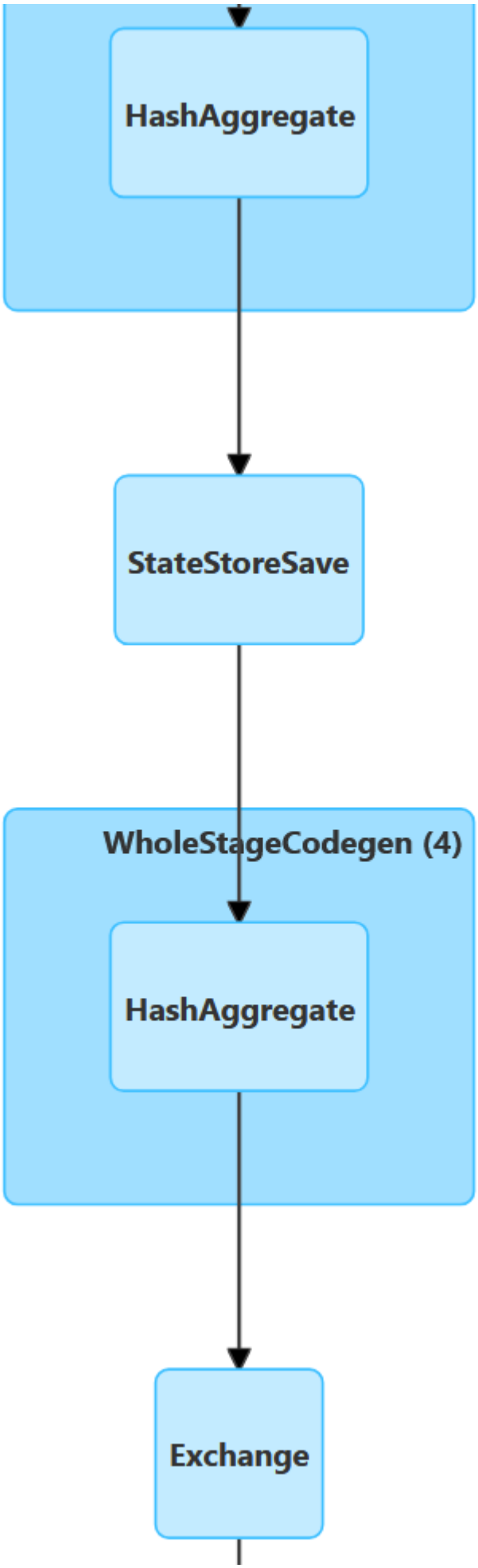
| Complete Tasks | Total Tasks |
|---|---|
| 0 | 0 |
| 5810 | 5810 |
| 5800 | 5800 |

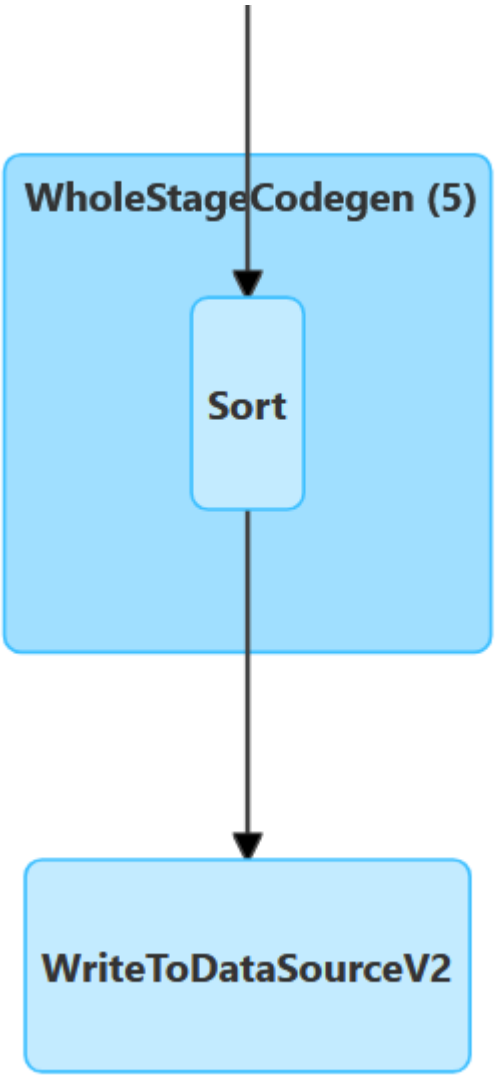
I think they are quite evenly distributed which is good. Again the one at the top is the driver therefore this does not count.

The SQL/Query Tab


MicroBatchScan
Filter
Project
WholeStageCodegen (1)
HashAggregate


Exchange
WholeStageCodegen (2)
HashAggregate
StateStoreRestore
WholeStageCodegen (3)


HashAggregate
StateStoreSave
WholeStageCodegen (4)
HashAggregate
Exchange




This is sadly the best way I am able to show the DAG as it is just a line downwards.

It has two exchange blocks. I cannot fully state if this is now being evenly distributed across all of my cores, but I can say if there is a problem it is in the exchange as I read in the details down:

(5) Exchange Input [2]: [source_ip#18, count#32L] Arguments: hashpartitioning(source_ip#18, 200), REQUIRED_BY_STATEFUL_OPERATOR, [plan_id=15493]

Which is also true for the other partitions. Now if one source IP is more frequent then the other (the one thing I cannot tell you right now as I do not know how I would figure that out), then it would be distributed to this again, so small number of partitions recieve the most data would be the skewness I think. But right now we only have 2 Partitions and I do not know how I can check this.