UNIVERSITY OF PADOVA
Department of Physics and Astronomy 'Galileo Galilei'
MSc in Physics of Data

Streaming processing of cosmic rays using Drift Tubes detectors

Gabriele Bertinelli - Roben Bhatti - Diego Bonato - Martina Cacciola

July 12th, 2023

Goal of the project

- Our project aims to create a simulated data processing network for a particle physics detector, enabling **real-time** monitoring of the results through a dashboard.

- The dataset consists of multiple text files in comma-separated values (CSV) format, hosted on Cloud Veneto's cloud storage bucket.

- Our objective is to simulate a continuous DAQ stream by injecting the provided dataset into a Kafka **topic**. After performing data cleansing, we package the extracted information into individual **messages** per batch and inject them into a new Kafka topic.

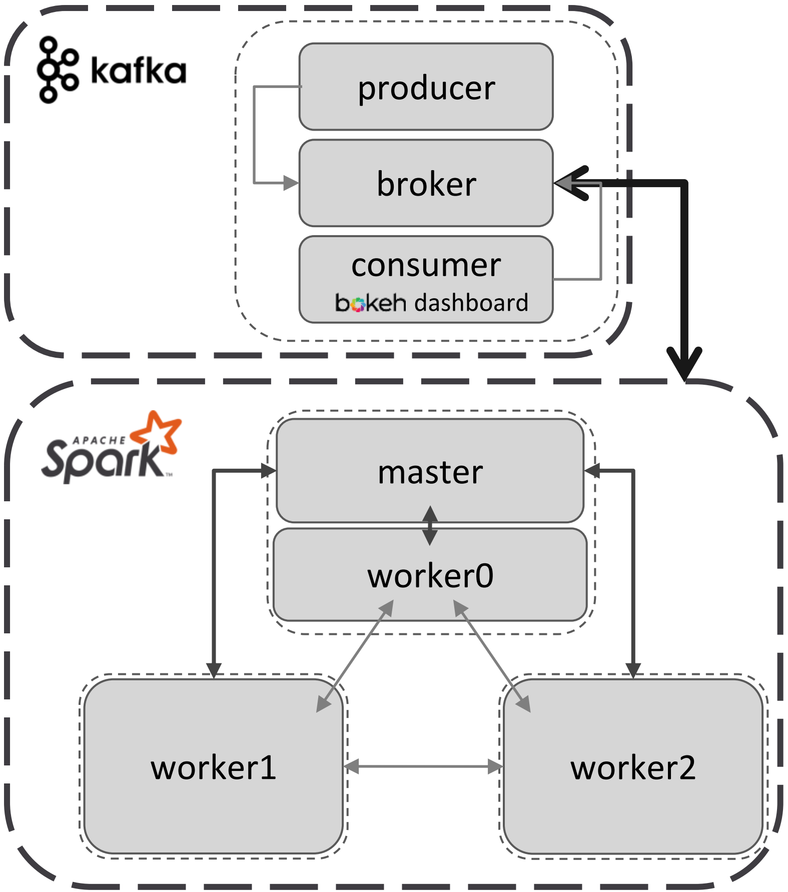
Tools

To implement our project, we utilize various frameworks and Python packages, including:

- **Kafka 3.3.2**: This distributed event streaming platform is employed to manage the data streaming process. We interface with Kafka using the ***kafka*** package.

- **Spark 3.3.2**: This cluster computing framework is used for data analytics, enabling distributed computation.

- **Bokeh 3.1.1**: This Python package is utilized to enhance the visual representation of our data through interactive plotting.


kafka
producer
broker
consumer
bokeh dashboard
APACHE Spark
master
worker0
worker1
worker2

Network architecture

- Spark master → 2 cores
- worker0 → 2 cores

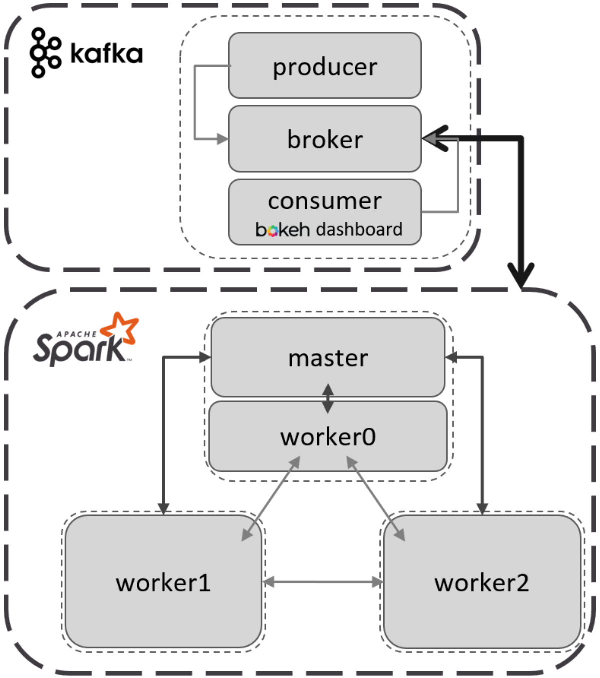
} 8 GB RAM

- worker1 →
- worker2 →

} 4 cores 8 GB RAM

- Kafka broker
- producer
- consumer

} 4 cores 8 GB RAM



Producer

- Connect to the cluster → KAFKA_BOOTSTRAP_SERVER
- Define a producer
- Create two topics → data_raw, data_clean
  - num_partitions
  - replicaton_factor (=1)

```
data_raw = NewTopic(name='data_raw',
                    num_partitions=100,
                    replication_factor=1)

data_clean = NewTopic(name='data_clean',
                      num_partitions=1,
                      replication_factor=1)

kafka_admin.create_topics(new_topics=[data_raw, data_clean])
```

Producer

- Connect to s3 bucket

- Access each file by `` `Key` `` value → save it in a Pandas Dataframe

- Loop in the dataframe → append each row to a message
- Send message asynchronously → `KafkaProducer.send()`
- Full batch → `KafkaProducer.flush()`

```
df=pd.read_csv(s3_client.get_object(Bucket='mapd-minidt-stream', Key=obj['Key']).get('Body'), sep=' ')
'...'
        # append a record to the msg to send
        producer.send('data_raw', row)
'...'
        if batch_counter==batch_size:
            producer.flush()
```

Spark analysis

- Session and Context creation

- To test performance → we varied these parameters
  - `spark.executor.instances`: n° of executors
  - `spark.executor.cores`: n° of CPU cores for each executor
  - `spark.sql.shuffle.partitions`: n° partitions used when shuffling for joins/aggregations
  - `spark.sql.execution.arrow.pyspark.enabled`: in memory columnar format → no major differences → left `` `true` ``

- Producer creation → send the final message to the `data_clean` topic

Spark analysis

```python
# read streaming df from kafka
inputDF = spark\
    .readStream\
    .format("kafka")\
    .option("kafka.bootstrap.servers", KAFKA_BOOTSTRAP_SERVER)\
    .option("kafkaConsumer.pollTimeoutMs", 1000)\
    .option('subscribe', 'data_raw')\
    .option("startingOffsets", "latest") \
    .load()
```

```python
# extract the value from the kafka message
rawraw_data = inputDF.select(col("value").cast("string")).alias("csv")

# split the csv line in the corresponding fields
raw_data = rawraw_data.selectExpr("cast(split(value, ',')[0] as int) as HEAD",
                                   "cast(split(value, ',')[1] as int) as FPGA",
                                   "cast(split(value, ',')[2] as int) as TDC_CHANNEL",
                                   "cast(split(value, ',')[3] as long) as ORBIT_CNT",
                                   "cast(split(value, ',')[4] as int) as BX_COUNTER",
                                   "cast(split(value, ',')[5] as double) as TDC_MEAS")
```

- Select useful data, add `CHAMBER` column

Spark analysis

- Create a function to be applied to each batch, `batch_processing`

- Tasks:
  - total number of processed hits: `hit_count`
  - total number of processed hits per chamber: `hit_count_chamber`
  - histogram of the counts of active `TDC_CHANNEL`, per chamber: `ch*_tdc_counts_list`
  - histogram of the total number of active `TDC_CHANNEL` in each `ORBIT_CNT`, per chamber: `ch*_orbit_counts_list`

- Create JSON message → send to topic

Spark analysis

```python
def batch_processing(df, epoch_id):

    df = df.persist()

    # 1: total number of processed hits, post-cleansing (1 value per batch)
    hit_count = df.count()

    # 2: total number of processed hits, post-cleansing, per chamber (4 values per batch)
    hit_count_chamber = df.groupby('CHAMBER').agg(count('TDC_CHANNEL').alias('HIT_COUNT'))\
                        .sort("CHAMBER").select('HIT_COUNT')\
                        .agg(collect_list('HIT_COUNT')).collect()
```

Spark analysis

```python
def batch_processing(df, epoch_id):
'...'

    # 3: histogram of the counts of active TDC_CHANNEL, per chamber (4 arrays per batch)
    tdc_counts = df.groupby(['CHAMBER','TDC_CHANNEL']).agg(count('TDC_CHANNEL').alias('TDC_COUNTS'))
    tdc_counts = tdc_counts.persist()
    # Filter the tdc_counts DataFrame for each chamber
    ch*_tdc_counts = tdc_counts.filter(tdc_counts.CHAMBER == *).select('TDC_CHANNEL','TDC_COUNTS')\
                    .sort("TDC_CHANNEL").toPandas()
    #Save it in a list
    ch*_tdc_channels_list = list(ch*_tdc_counts['TDC_CHANNEL'])
    ch*_tdc_counts_list   = list(ch*_tdc_counts['TDC_COUNTS'])

    # 4: histogram of the total number of active TDC_CHANNEL in each ORBIT_CNT, per chamber (4 arrays per batch)
    orbit_count=df.groupby(['CHAMBER','ORBIT_CNT']).agg(countDistinct("TDC_CHANNEL").alias('TDC_ORBIT'))
    orbit_count = orbit_count.persist()
    ch*_orbit_counts = orbit_count.filter(orbit_count.CHAMBER == *).select('ORBIT_CNT','TDC_ORBIT')\
                    .sort("ORBIT_CNT").toPandas()
    #Save it in a list
    ch*_orbit_list          = list(ch*_orbit_counts['ORBIT_CNT'])
    ch*_orbit_counts_list   = list(ch*_orbit_counts['TDC_ORBIT'])

    df.unpersist()
    tdc_counts.unpersist()
    orbit_count.unpersist()
```

Spark analysis

```
def batch_processing(df, epoch_id):
'...'
    msg = { 'msg_ID': ID,
            'hit_count': hit_count,
            'hit_count_chamber': hit_count_chamber[0][0],

            'tdc_counts_chamber': {
                '0': {
                    'bin_edges': ch0_tdc_channels_list,
                    'hist_counts': ch0_tdc_counts_list
                },
                '...'
            'active_tdc_chamber': {
                '0': {
                    'bin_edges': ch0_orbit_list,
                    'hist_counts': ch0_orbit_counts_list
                },
                '...'
                }}
    producer.send('data_clean', json.dumps(msg).encode('utf-8'))
    producer.flush()
```

Spark analysis

- Apply the function to the streaming dataset

```
query = raw_data.writeStream\
            .outputMode("update")\
            .foreachBatch(batch_processing)\
            .option("checkpointLocation", "checkpoint")\
            .trigger(processingTime='5 seconds')\
            .start()
query.awaitTermination(5)
```

Consumer - Dashboard

- Creation of the consumer

```
# Initialize Kafka consumer by subscribing to the topic
consumer = KafkaConsumer('data_clean',
                         bootstrap_servers = KAFKA_BOOTSTRAP_SERVER)
```

- Poll messages

```
def polling():
    for msg in consumer:
        value = json.loads(msg.value.decode('utf-8'))  # Convert the message value to a dictionary
        break
    return value
```

```
def create_value():
    #instantiate dictionary
    combined_dict = {}
    poll = polling()

    x = poll["msg_ID"]
    y = poll['hit_count']
    combined_dict["p1"] = {"x": [x], "y": [y]}
    '...'
return combined_dict
```

Consumer - Dashboard

- Create periodic function

```
def update():
    #poll data and filter them
    new_data = create_value()

    #stream new data to dashboard
    source1.stream(new_data["p1"], rollover=10)
    source2.stream(new_data["p2_0"], rollover=10)

'...'
```

- Layout configuration and axis formatting

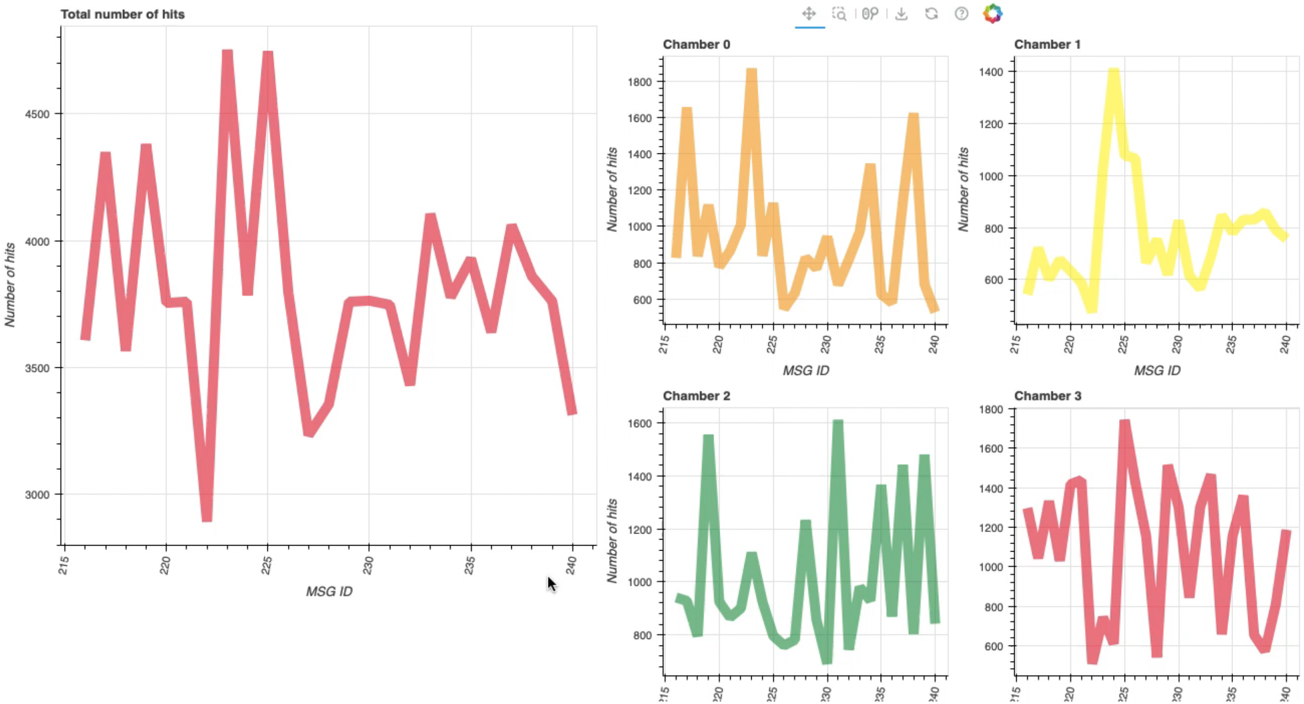
```
curdoc().add_root(grid_with_title)
curdoc().add_periodic_callback(update,500)
```

- Launch the dashboard → `bokeh serve --show consumer_dashboard.ipynb`

Cosmic Rays Analysis

Diego Bonato, Martina Cacciola, Gabriele Bertinelli, Roben Bhatti

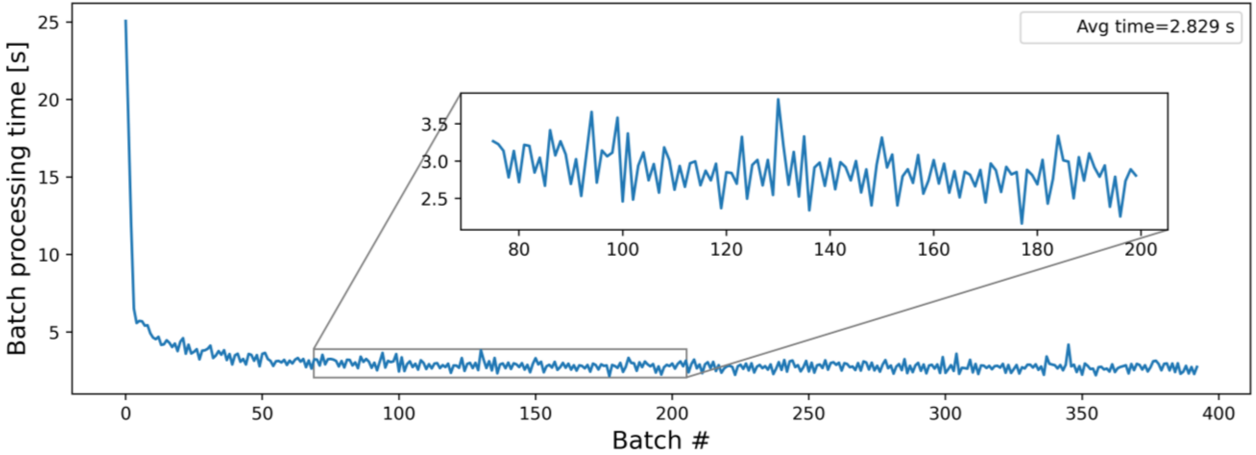
Total number of processed hits

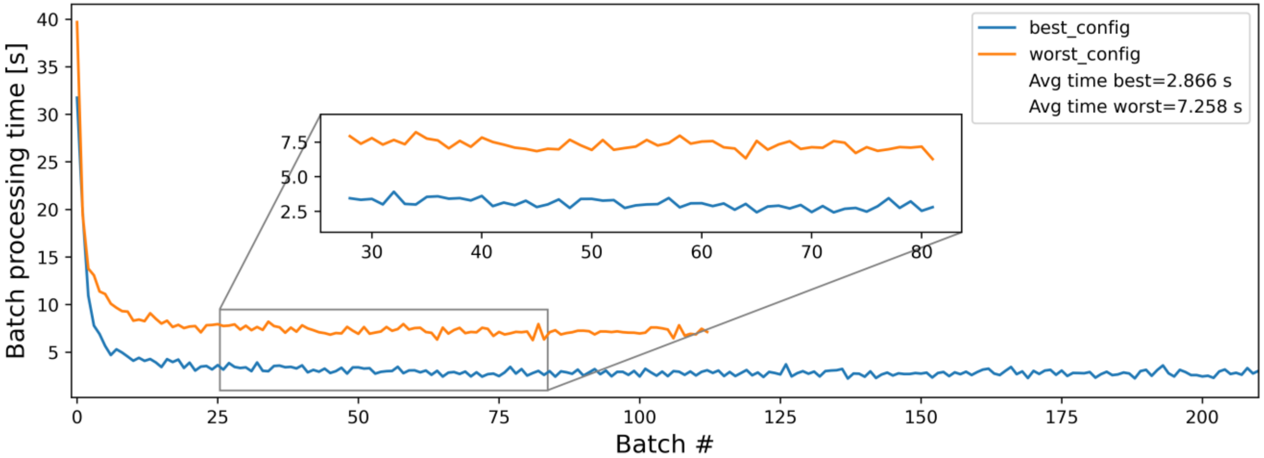


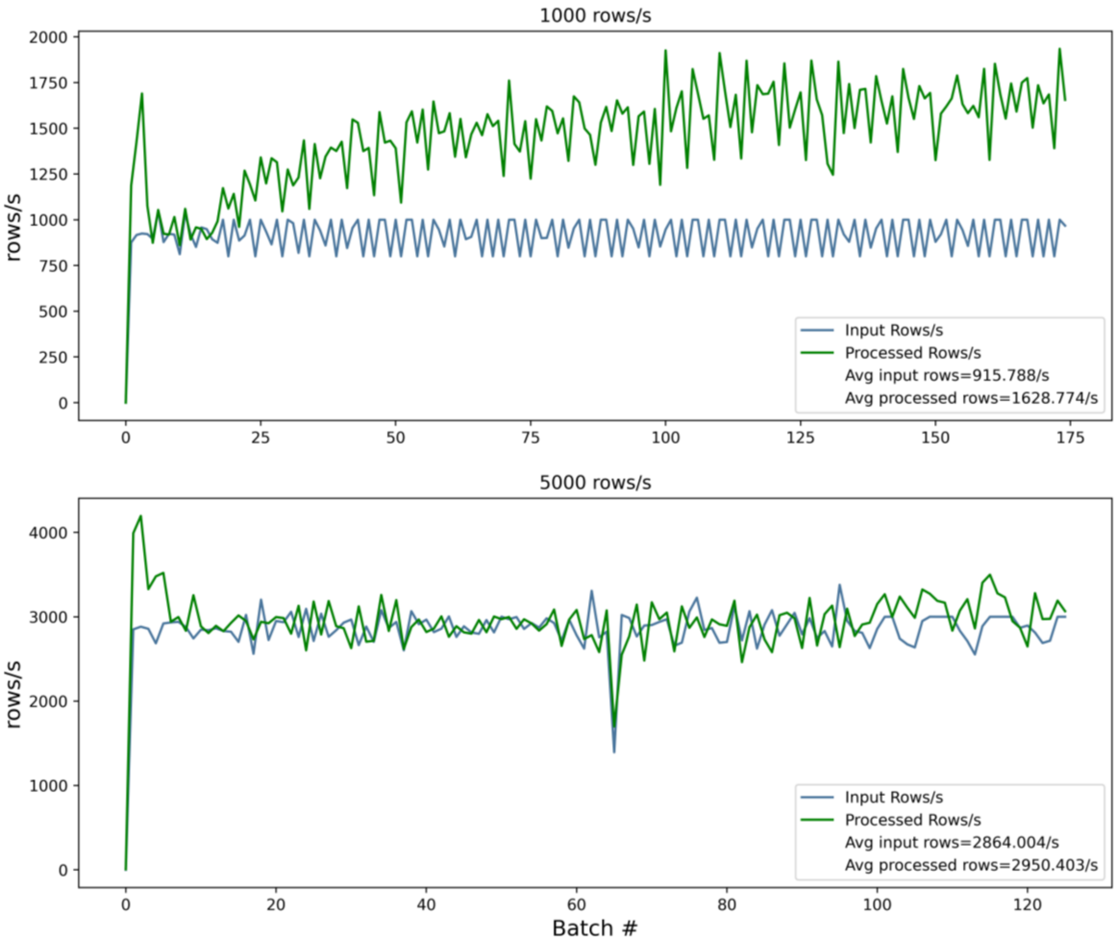
Metrics analysis

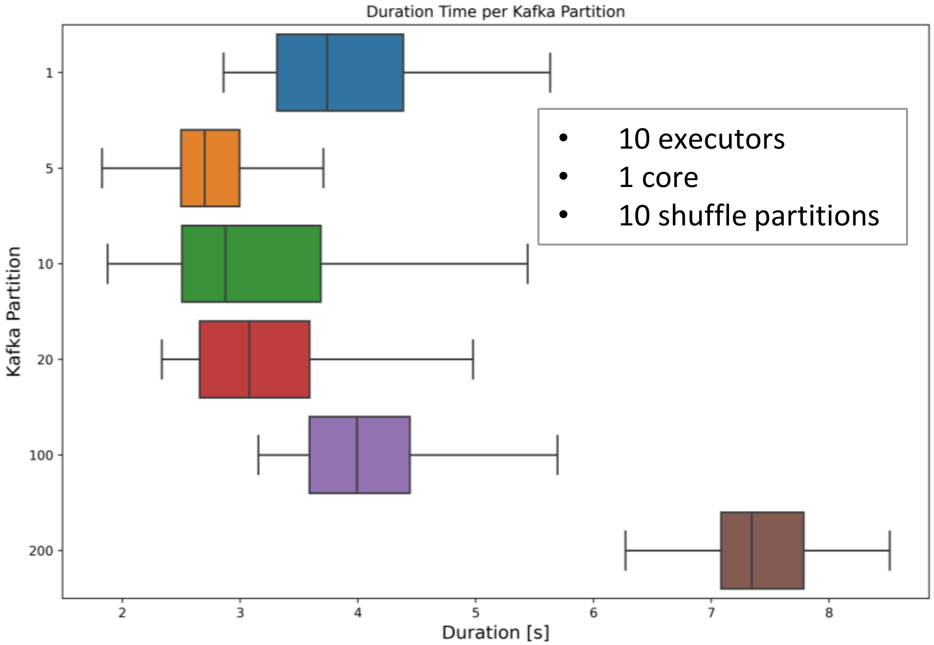
We conducted a study to examine how the performance of the network scales by altering the following parameters:

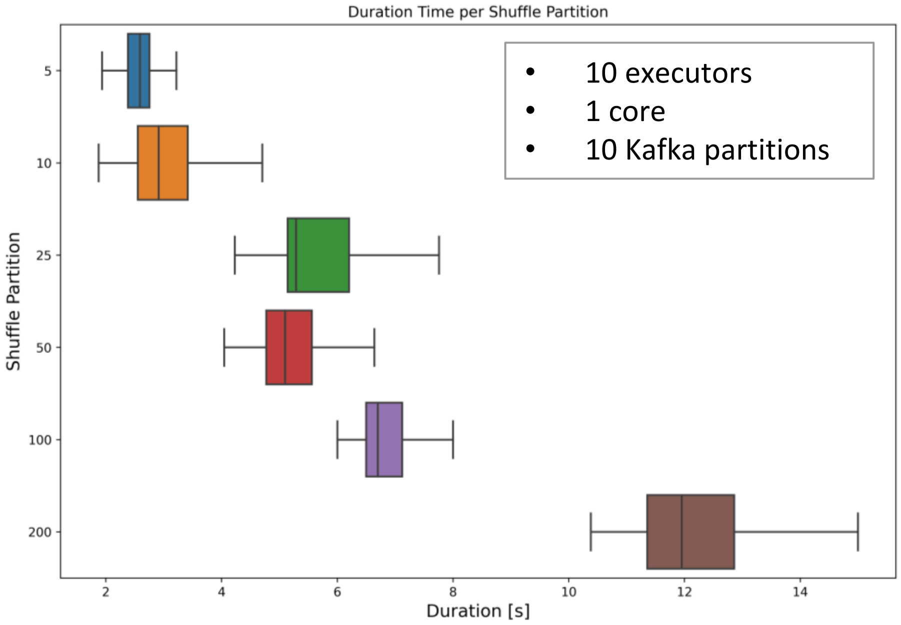
- Number of executors (and cores) → **10 (1 core)**, 3 (2, 4, 4 cores), 5 (2 cores)
- Number of Kafka partitions → 1, 5, **10**, 100, 200
- Number of shuffle partitions → 5, **10**, 25, 50, 100, 200
- Input rows → **1000**, 5000 rows/s

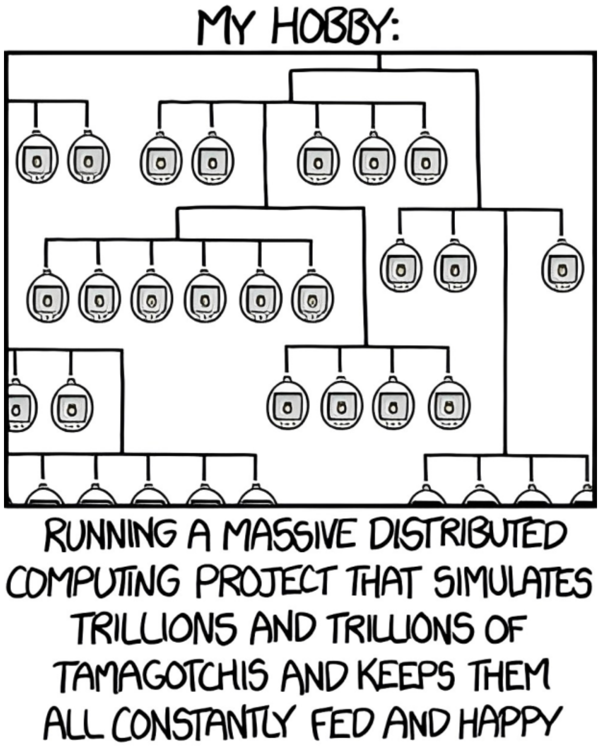

Avg time=2.829 s
Batch processing time [s]
Batch #


best_config
worst_config
Avg time best=2.866 s
Avg time worst=7.258 s
Batch processing time [s]
Batch #


1000 rows/s
rows/s
2000
1750
1500
1250
1000
750
500
250
0
0
25
50
75
100
125
150
175
Input Rows/s
Processed Rows/s
Avg input rows=915.788/s
Avg processed rows=1628.774/s
5000 rows/s
rows/s
4000
3000
2000
1000
0
0
20
40
60
80
100
120
Batch #
Input Rows/s
Processed Rows/s
Avg input rows=2864.004/s
Avg processed rows=2950.403/s


Duration Time per Kafka Partition
Kafka Partition
1
5
10
20
100
200
2
3
4
5
6
7
8
Duration [s]
• 10 executors
• 1 core
• 10 shuffle partitions


Duration Time per Shuffle Partition
Shuffle Partition
5
10
25
50
100
200
2
4
6
8
10
12
14
Duration [s]
• 10 executors
• 1 core
• 10 Kafka partitions


MY HOBBY:
RUNNING A MASSIVE DISTRIBUTED COMPUTING PROJECT THAT SIMULATES TRILLIONS AND TRILLIONS OF TAMAGOTCHIS AND KEEPS THEM ALL CONSTANTLY FED AND HAPPY

Credit: XKCD