

Comprehensive Data Pipeline Solutions Using Spark, Kafka, and AWS

Ronak Sengupta

Senior Software Engineer 2

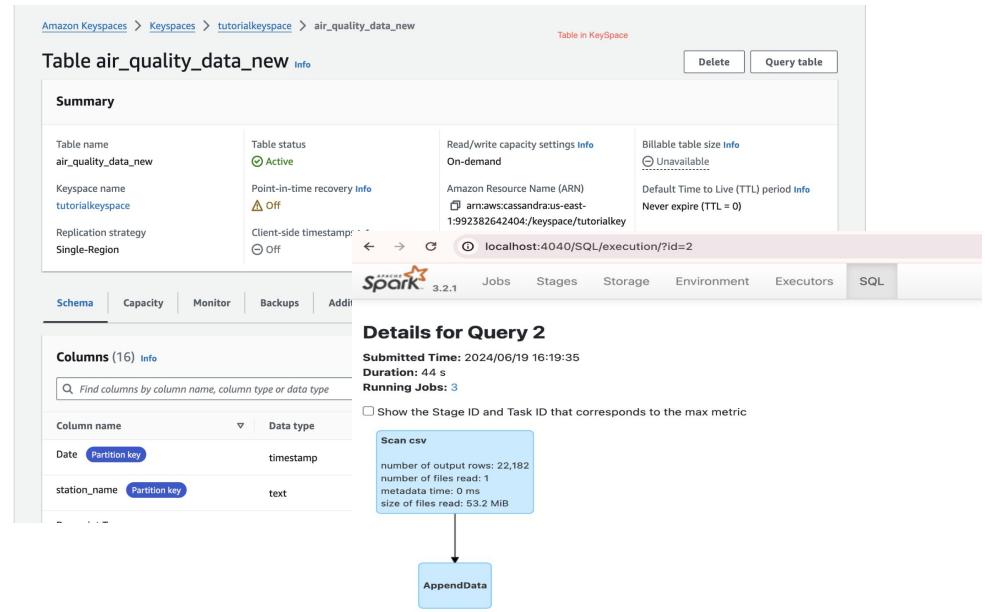
Payoda Technologies

Scenario 1 - Data Pipeline for performing Data conversions and performing aggregation operations

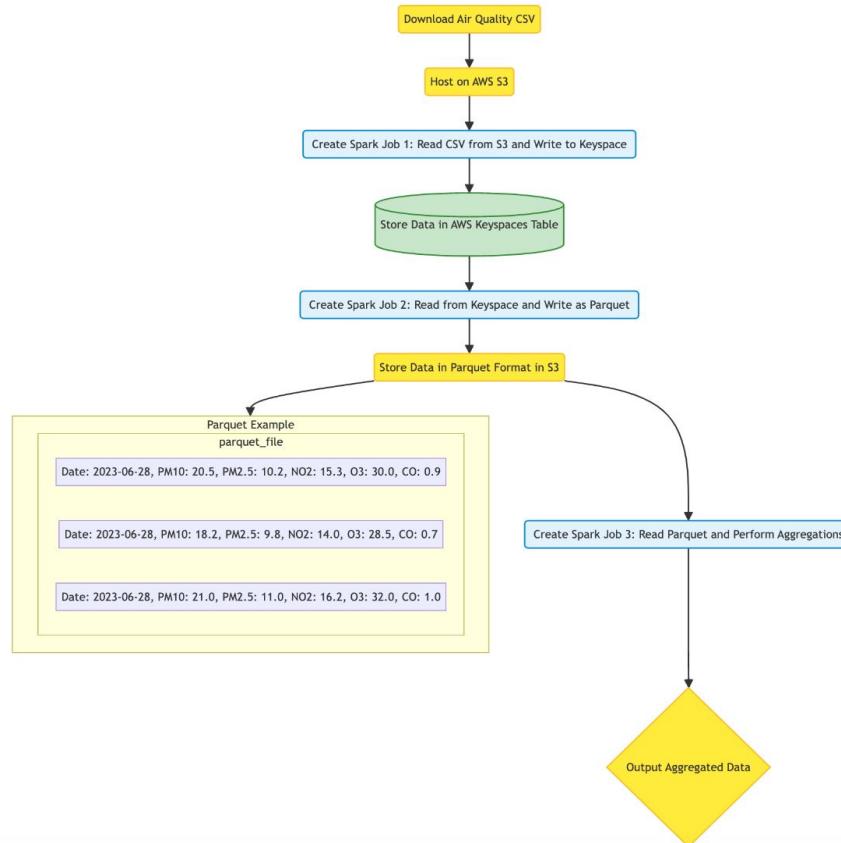
Description: In this project, I designed a data pipeline to analyze air quality data by seamlessly reading from AWS S3, writing to AWS Keyspaces for storage, converting and storing as Parquet format in S3, and performing crucial aggregation operations, designed by Apache Spark for efficient data processing. [Code Link:- <https://github.com/r1999-ron/SparkScenario1Task>]

Screenshot of aggregate operations

keyspace table screenshot & DAG Output



Flow Diagram - Scenario 1:



Scenario 2 - Streaming JOB using Kafka , Spark and Protobuf

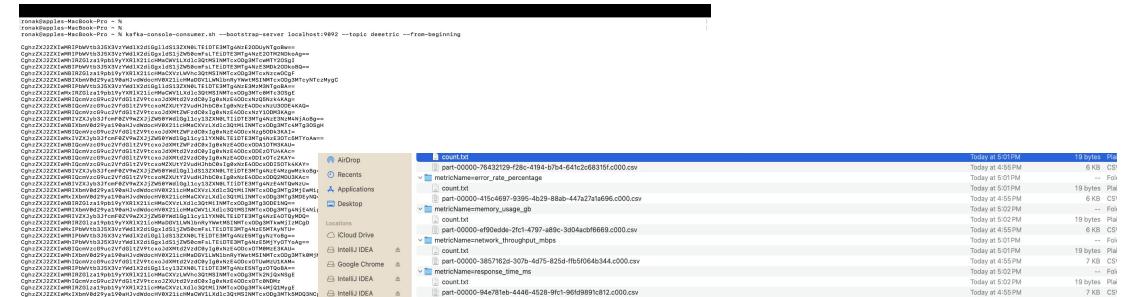
Description: In Scenario 2, a streaming pipeline using AKKA, Apache Spark, Protobuf, Spark Streaming, and Apache Kafka was implemented. An AKKA microservice generates JSON server metrics data posted to Kafka every 5 seconds. Spark Streaming converts JSON to Protobuf and publishes to another Kafka topic, then deserializes Protobuf to CSV files based on metric type. Spark Jobs aggregate these CSV files for comprehensive analysis, leveraging Spark and Kafka for scalable, fault-tolerant data processing. [Code Link:-

<https://github.com/r1999-ron/SparkScenario2Task> & <https://github.com/r1999-ron/SparkScenario2.2Task>

Screenshot of akka kafka topic

```
[zona@Apple-MacBook-Pro ~ % kafk-console-consumer.sh --bootstrap-server localhost:9092 --topic metric-message --from-beginning
```

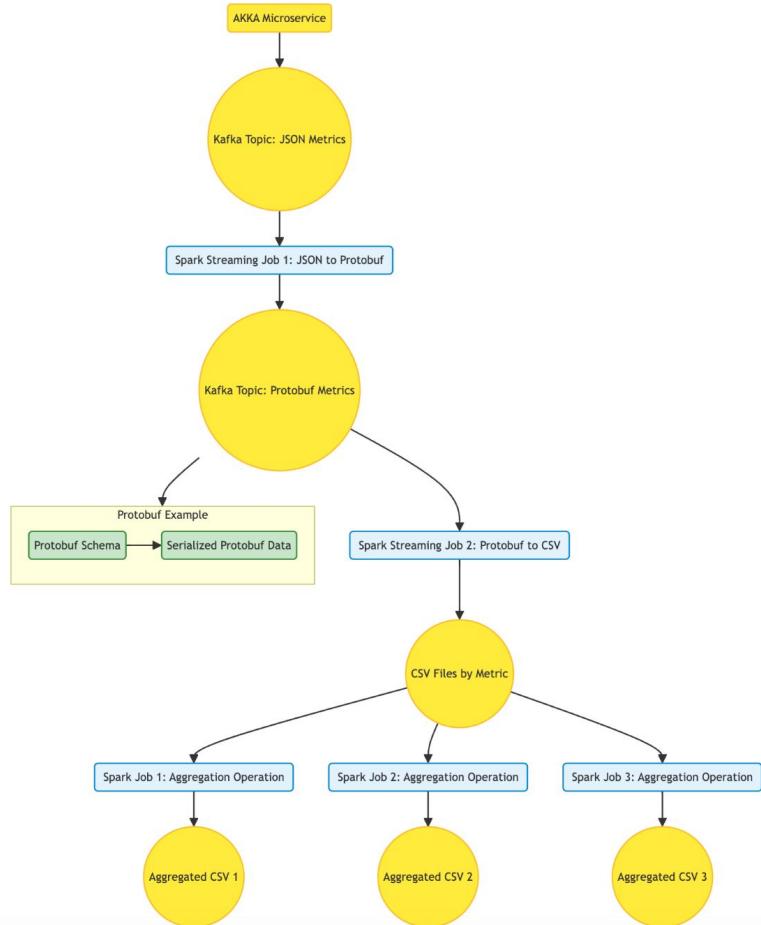
Screenshot of binary data(protoBuf format) & output csv screenshots



Aggregate Output

1	server_name	total_sum	count	average
2	server02	436	8	54.5

Flow Diagram - Scenario 2:



Scenario 3 - Executing Jobs on EMR

Description: - Scenario 3 utilizes AWS EMR, RDBMS, S3 Bucket, Spark, and Scala for data processing. It begins with setting up an EMR Cluster and Studio with a notebook. The dataset from Kaggle is stored in an S3 bucket, processed using an EMR Notebook to convert files to Parquet format and store them in another S3 bucket. Ten Spark Jobs on EMR perform operations like filtering and aggregation on the Parquet data, with results stored in MySQL tables for structured analysis and management. [Notebook Link:- <https://github.com/r1999-ron/SparkMandatoryTask3/blob/main/SparkMandatory3Task.ipynb>]

Screenshot of S3 bucket:

The screenshot shows the AWS S3 console interface for the 'emrfolder123' bucket. The 'Objects' tab is selected, displaying a list of six CSV files. Each file has a download icon, a copy URL icon, a copy URI icon, an open icon, a delete icon, and an actions dropdown. A 'Create folder' and 'Upload' button are also present. The table headers include Name, Type, Last modified, Size, and Storage class. All files are CSV type, last modified on June 25, 2024, at 10:51:24 UTC-05:30, and have a standard storage class.

Name	Type	Last modified	Size	Storage class
AP001.csv	CSV	June 25, 2024, 10:51:21 (UTC-05:30)	7.8 MB	Standard
AP002.csv	CSV	June 25, 2024, 10:51:22 (UTC-05:30)	4.5 MB	Standard
AP003.csv	CSV	June 25, 2024, 10:51:22 (UTC-05:30)	6.9 MB	Standard
AP004.csv	CSV	June 25, 2024, 10:51:23 (UTC-05:30)	7.0 MB	Standard
AP005.csv	CSV	June 25, 2024, 10:51:24 (UTC-05:30)	6.5 MB	Standard
AP006.csv	CSV	June 25, 2024, 10:51:24 (UTC-05:30)	739.1 KB	Standard

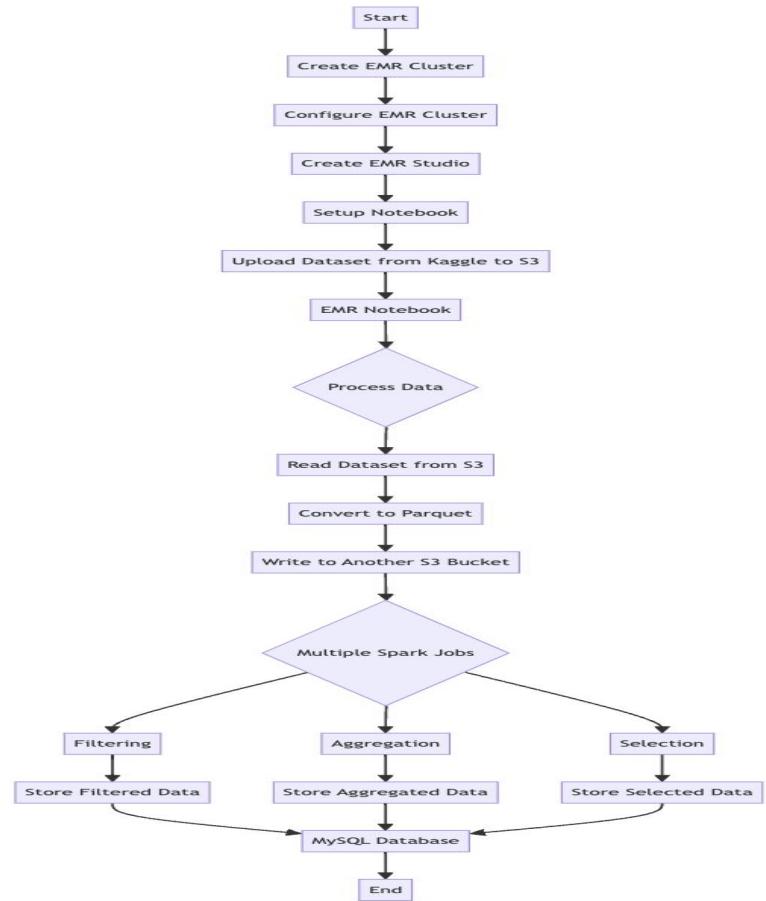
Spark Job filtration screenshot

```
[mysql]> select * from avg_ws_df limit 0,10;
```

From Date	Avg_WS
2023-02-10 10:00:00	266.8008053691275
2023-02-10 23:00:00	223.77164285714284
2023-02-13 16:00:00	244.7440604026846
2023-02-17 10:00:00	249.421821192053
2023-03-03 05:00:00	221.19941379310345
2023-03-06 13:00:00	271.3822580645161
2023-02-03 01:00:00	220.89655913978498
2023-02-04 11:00:00	277.2291467576792
2023-01-14 17:00:00	220.35160142348755
2023-01-29 10:00:00	235.0301742160279

10 rows in set (0.29 sec)

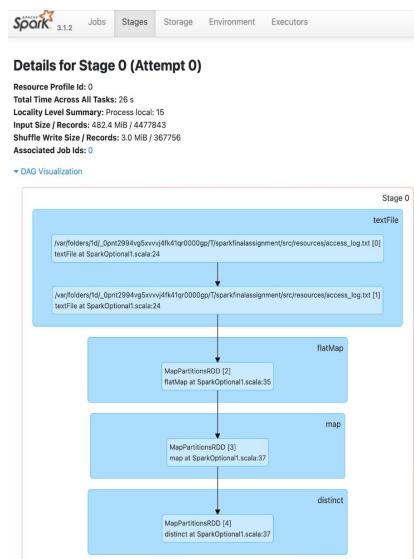
Flow Diagram of Scenario - 3



Optional Scenario 1

Description: Optional Scenario 1 uses RDDs in Apache Spark to process access log data from Kaggle in text format. Steps include reading the log file, converting it to RDDs, and extracting IP addresses, URLs, response statuses, and bytes sent. Key operations include finding unique IPs, grouping URLs by 200 status, counting 4xx responses, identifying requests > 5000 bytes, determining URLs with the most requests, and finding URLs with the most 404 errors. These operations offer insights into traffic patterns, response statuses, and error occurrences in the log data. [Code link:- <https://github.com/r1999-ron/SparkOptional1Task>]

Screenshot of DAG output



Screenshot of spark job operation

The screenshot shows the terminal window of the Spark application UI. The code in the terminal is as follows:

```
build.sbt
SPARKASSIGNMENTOPTIONAL1
> .bloop
> .metals
> .vscode
> project
> src/main-scala
  └── SparkOptional1.scala
> target
build.sbt
```

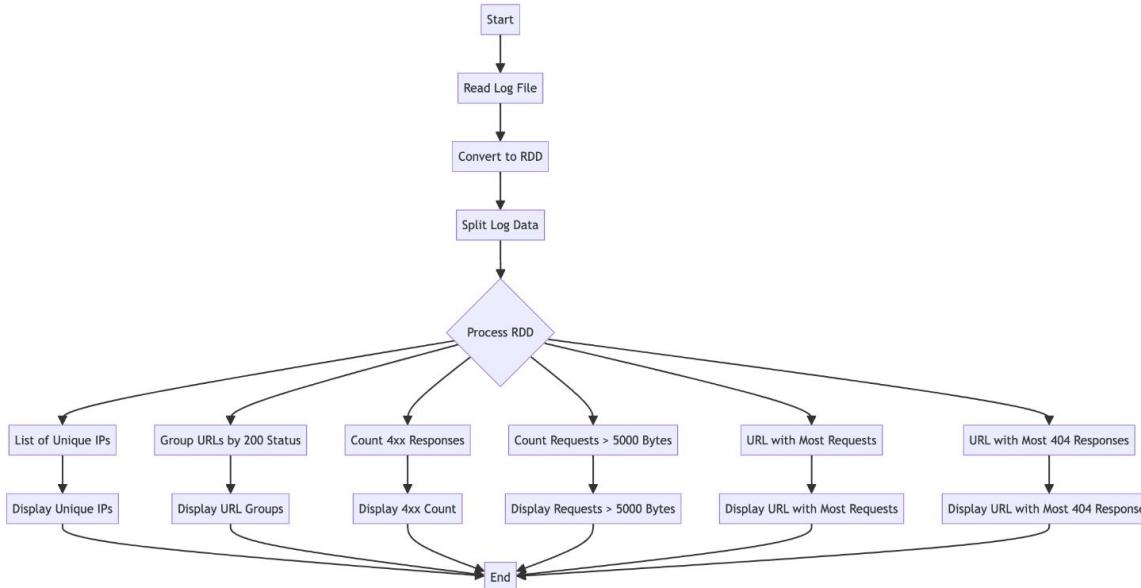
The code in `SparkOptional1.scala` is:

```
object Main extends App {
    val args: Array[String] = args
    val hostRequestedUrl = parsedLogs.map(_._2).countByValue().maxBy(_._2)
    println(s"URL with the most number of requests: ${hostRequestedUrl._1} (${hostRequestedUrl._2})")
    // URL with the most number of 404 responses
    val most404Url = parsedLogs.filter(_._3 == 404).map(_._2).countByValue().maxBy(_._2)
    // println("URL with the most number of 404 responses: ${most404Url._1} (${most404Url._2})")
    spark.stop()
}
```

The terminal shows the following logs:

```
[error] 24/06/23 23:42:46 INFO TaskSetManager: Finished task 0.0 in stage 1.0 (TID 10) in 246 ms on 192.168.0.100 (executor driver) (8/10)
[error] 24/06/23 23:42:46 INFO TaskSetManager: Finished task 7.0 in stage 1.0 (TID 17) in 251 ms on 192.168.0.100 (executor driver) (9/10)
[error] 24/06/23 23:42:46 INFO TaskSetManager: Finished task 6.0 in stage 1.0 (TID 16) in 251 ms on 192.168.0.100 (executor driver) (10/10)
[error] 24/06/23 23:42:46 INFO TaskSchedulerImpl: Removed TaskSet 1.0, whose tasks have all completed, from pool
[error] 24/06/23 23:42:46 INFO DAGScheduler: ResultStage 1 (countByValue at SparkOptional1.scala:43) finished in 0.261 s
[error] 24/06/23 23:42:46 INFO DAGScheduler: Job 0 is finished. Cancelling potential speculative or zombie tasks for this job
[error] 24/06/23 23:42:46 INFO TaskSchedulerImpl: Killing all running tasks in stage 1; Stage finished
[error] 24/06/23 23:42:46 INFO DAGScheduler: Job 0 finished; countByValue at SparkOptional1.scala:43, took 2.333704 s
[info] URL with the most number of requests: /assets/css/combined.css (117348)
[error] 24/06/23 23:42:46 INFO SparkUI: Stopped Spark web UI at http://192.168.0.100:4040
[error] 24/06/23 23:42:46 INFO MapOutputTrackerMasterEndpoint: MapOutputTrackerMasterEndpoint stopped!
[error] 24/06/23 23:42:46 INFO MemoryStore: MemoryStore cleared
```

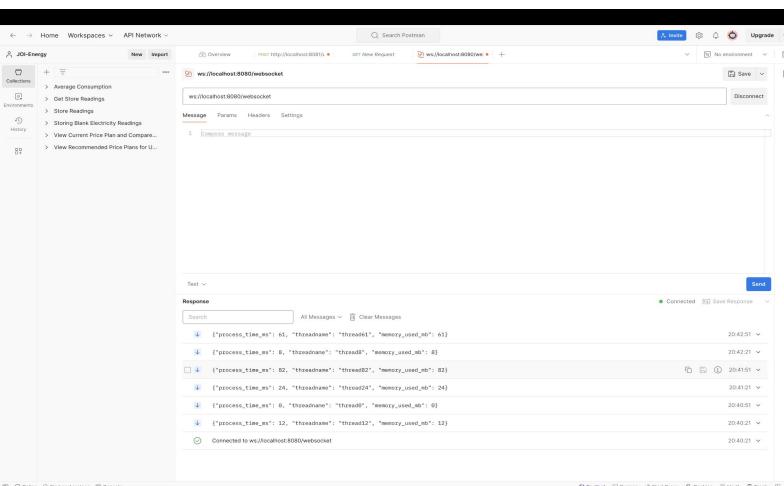
Flow Diagram - Optional 1



Optional Scenario 5

Description: Optional Scenario 5 involves creating a WebSocket using the Akka framework to produce JSON objects every 30 seconds with process time, thread name, and memory used. A Spark Streaming job consumes this WebSocket data, writes it to a Kafka topic in AVRO format, and another Spark Streaming job performs aggregation operations on the consumed data.[\[Code link:-
<https://github.com/r1999-ron/SparkOptional5Task>](https://github.com/r1999-ron/SparkOptional5Task)

Screenshot of postman websocket connection

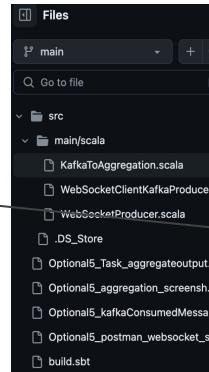
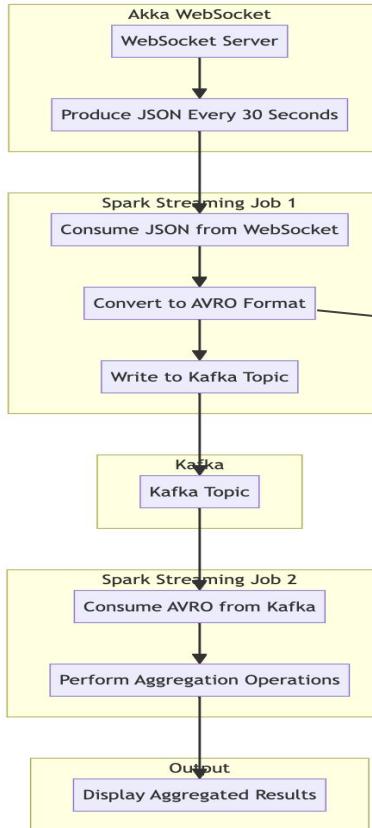


kafka consuming websocket data & aggregation output

```
(*process_time_ms": 29, "threadname": "thread29", "memory_used_mb": 29)
(*process_time_ms": 74, "threadname": "thread74", "memory_used_mb": 74)
(*process_time_ms": 12, "threadname": "thread12", "memory_used_mb": 12)
(*process_time_ms": 35, "threadname": "thread35", "memory_used_mb": 35)
(*process_time_ms": 16, "threadname": "thread16", "memory_used_mb": 16)
(*process_time_ms": 14, "threadname": "thread14", "memory_used_mb": 14)
(*process_time_ms": 93, "threadname": "thread93", "memory_used_mb": 93)
(*process_time_ms": 56, "threadname": "thread56", "memory_used_mb": 56)
(*process_time_ms": 37, "threadname": "thread37", "memory_used_mb": 37)
(*process_time_ms": 17, "threadname": "thread17", "memory_used_mb": 17)
(*process_time_ms": 39, "threadname": "thread39", "memory_used_mb": 39)
(*process_time_ms": 36, "threadname": "thread36", "memory_used_mb": 36)
(*process_time_ms": 58, "threadname": "thread58", "memory_used_mb": 58)
(*process_time_ms": 64, "threadname": "thread64", "memory_used_mb": 64)
(*process_time_ms": 11, "threadname": "thread11", "memory_used_mb": 11)
(*process_time_ms": 58, "threadname": "thread58", "memory_used_mb": 58)
(*process_time_ms": 16, "threadname": "thread16", "memory_used_mb": 16)
(*process_time_ms": 12, "threadname": "thread12", "memory_used_mb": 12)
(*process_time_ms": 55, "threadname": "thread55", "memory_used_mb": 55)
(*process_time_ms": 43, "threadname": "thread43", "memory_used_mb": 43)
(*process_time_ms": 98, "threadname": "thread98", "memory_used_mb": 98)
(*process_time_ms": 1, "threadname": "thread1", "memory_used_mb": 1)
(*process_time_ms": 4, "threadname": "thread4", "memory_used_mb": 4)
(*process_time_ms": 96, "threadname": "thread96", "memory_used_mb": 96)
(*process_time_ms": 55, "threadname": "thread55", "memory_used_mb": 55)
(*process_time_ms": 34, "threadname": "thread34", "memory_used_mb": 34)
(*process_time_ms": 35, "threadname": "thread35", "memory_used_mb": 35)
(*process_time_ms": 7, "threadname": "thread7", "memory_used_mb": 7)

{"threadname": "thread85", "avg_process_time": 85.0, "max_process_time": 85, "min_memory_used": 85, "max_memory_use
d": 85, "min_memory_used": 85}
{"threadname": "thread18", "avg_process_time": 18.0, "max_process_time": 18, "min_process_time": 18, "avg_memory_used": 18, "max_memory_us
e": 18, "min_memory_used": 18}
{"threadname": "thread44", "avg_process_time": 44.0, "max_process_time": 44, "min_process_time": 44, "avg_memory_used": 44.0, "total_memory_us
ed": 44, "min_memory_used": 44}
{"threadname": "thread11", "avg_process_time": 11.0, "max_process_time": 11, "min_process_time": 11, "avg_memory_used": 11, "max_memory_us
e": 11, "min_memory_used": 11}
{"threadname": "thread22", "avg_process_time": 22.0, "max_process_time": 22, "min_process_time": 22, "avg_memory_used": 22, "max_memory_us
e": 22, "min_memory_used": 22}
{"threadname": "thread57", "avg_process_time": 57.0, "max_process_time": 57, "min_process_time": 57, "avg_memory_used": 57, "max_memory_us
e": 57, "min_memory_used": 57}
{"threadname": "thread98", "avg_process_time": 98.0, "max_process_time": 98, "min_process_time": 98, "avg_memory_used": 98.0, "total_memory_us
ed": 98, "min_memory_used": 98}
{"threadname": "thread32", "avg_process_time": 32.0, "max_process_time": 32, "min_process_time": 32, "avg_memory_used": 32.0, "max_memory_us
e": 32, "min_memory_used": 32}
{"threadname": "thread73", "avg_process_time": 73.0, "max_process_time": 73, "min_process_time": 73, "avg_memory_used": 73.0, "total_memory_us
ed": 73, "min_memory_used": 73}
```

Flow Diagram - Optional Task 5



Screenshot of a code editor showing the `KafkaToAggregation.scala` code:

```
def main(args: Array[String]): Unit = {
    // Read from Kafka
    val df = spark.readStream
        .format("kafka")
        .option("kafka.bootstrap.servers", "localhost:9092")
        .option("subscribe", "websocket-topics")
        .option("startingOffsets", "earliest")
        .option("kafka.max.partition.fetch.bytes", "10485760")
        .load()

    // Define the schema
    val avroSchema = """{
        "type": "record",
        "name": "WebSocketRecord",
        "fields": [
            {"name": "process_time_ms", "type": "int"},
            {"name": "threadname", "type": "string"},
            {"name": "memory_used_mb", "type": "int"}
        ]
}"""

    df.writeStream
        .format("avro")
        .outputMode("append")
        .option("path", "/tmp/kafka-aggregation")
        .start()
}
```



THANK YOU