PESUNIVERSITY

EC Campus, Bengaluru

UE20CS343 **DATABASE TECHNOLOGIES (DBT)**

Project Report

Traffic Data Analysis using Apache Spark Streaming and Kafka.

PES2UG20CS457: Sehag A

PES2UG20CS458: Setti Durga Poojitha

PES2UG20CS461: Shreyas Sai Raman

Under the guidance of

Dr. Geetha D

Department of Computer Science and Engineering

B. Tech. CSE - 6th Semester

Jan – May 2023

Table of Contents

The following contents have to be tabulated like this:

1. Introduction:

Traffic data refers to the information collected about the movement of vehicles and people on roads, highways, and other transportation systems. It is a valuable source of information that can be used for traffic management, city planning, and public safety.

To process traffic data in real-time, we can use Apache Spark Streaming, Spark SQL, and Apache Kafka Streaming. We can implement a pipeline that reads traffic data from a streaming source, processes it using Spark SQL queries, and stores the processed data in a DBMS like MySQL.

Here are the steps we can follow to implement the pipeline:

1) Set up an Apache Kafka cluster with at least three topics to publish, subscribe, and produce data.

2) Implement a Spark Streaming application that reads traffic data from the Kafka topics which includes the traffic from various locations.

3) Store the processed data in a DBMS like MySQL.

Next,we implement a Spark batch application that reads traffic data from the database and processes it using Spark SQL queries.The results, accuracy, and performance are compared with the Spark Streaming application.

In conclusion, we can use Apache Spark Streaming, Spark SQL, Apache Kafka Streaming, and DBMS tools like MySQL to process traffic data in real-time and compare it with batch processing.

2.Installation of Software:

1. Apache Kafka - kafka_2.13-3.4.0
   https://kafka.apache.org/downloads

2. Spark - 3.3.2 - https://spark.apache.org/

3. Python - 3.8.16 - https://www.python.org/

a. Streaming Tools Used:

1. Apache Kafka
2. Apache Spark

b. DBMS:

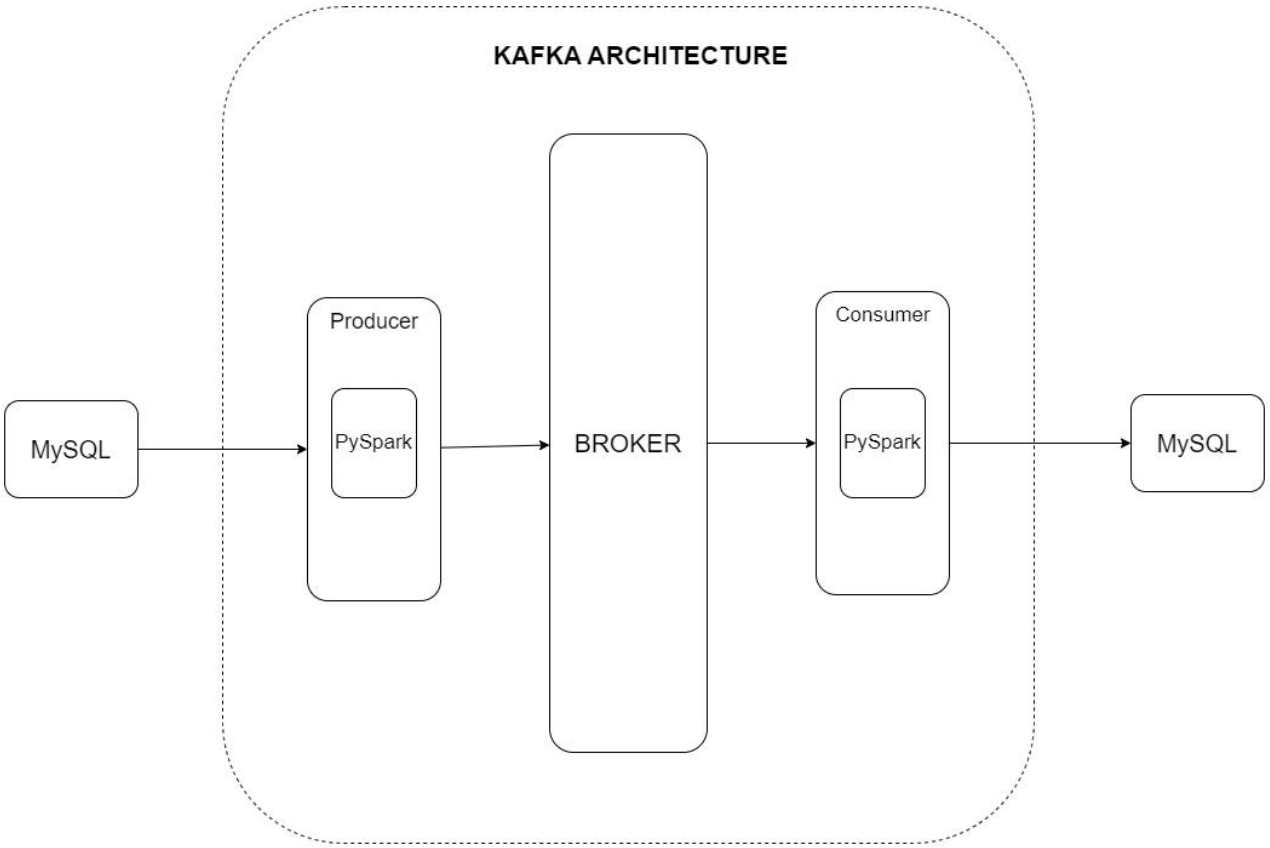
1. MYSQL database

3. Problem Description

1) The first step is to set up a Kafka cluster that can receive and store large streams of traffic data. Kafka acts as a message broker between the data sources and Spark.
2) A topic is a category or feed name to which records can be published. Here, a topic named "new-york",”san-franscisco” and “los-angeles” is created to publish traffic data records about New York, San Francisco and Los Angeles.
3) Apache Spark receives and processes data from Kafka for real-time stream processing. Set up Spark to receive data from Kafka and process it using Spark Streaming.
4) A tumbling window can be defined to receive data in fixed time intervals, such as every 5 minutes and . This allows for efficient processing of streaming data.

5) MySQL is a database we have chosen to store the processed traffic data. The processed traffic data is written to the MYSQL database for further analysis, querying and reporting.
6) We perform batch processing on the data which gets stored in the MYSQL database and the results are compared with the streaming mode of execution.

4. Architecture Diagram:



5. Input Data:

a. Source :

timestamp	location	speed
2022-04-23 01:23:45	New York	45
2022-04-23 01:23:50	Los Angeles	60
2022-04-23 01:24:01	Chicago	25
2022-04-23 01:24:05	Houston	10
2022-04-23 01:24:10	Phoenix	35
2022-04-23 01:24:15	New York	50
2022-04-23 01:24:20	Los Angeles	70
2022-04-23 01:24:25	Chicago	30
2022-04-23 01:24:30	Houston	15
2022-04-23 01:24:35	Phoenix	40
2022-04-23 01:24:40	New York	55
2022-04-23 01:24:45	Los Angeles	80
2022-04-23 01:24:50	Chicago	35
2022-04-23 01:24:55	Houston	20
2022-04-23 01:25:00	Phoenix	45
2022-04-23 08:00:00	New York	45.2

b. Description:

This input data contains traffic data with 3 columns:timestamp,location and speed. Each row represents a snapshot of the traffic conditions at that particular moment in time and location. This dataset is useful for analyzing traffic patterns and congestion in different locations and at different times.

6. Streaming Mode Experiment:

a. Windows

We First Start the Apache Kafka server

```
pes2ug20cs461@pes2ug20cs461:~$ sudo systemctl start kafka
[sudo] password for pes2ug20cs461:
pes2ug20cs461@pes2ug20cs461:~$ sudo systemctl status kafka
● kafka.service - Apache Kafka Server
     Loaded: loaded (/etc/systemd/system/kafka.service; disabled; vendor prese>
     Active: active (running) since Tue 2023-04-25 12:29:59 IST; 11h ago
       Docs: http://kafka.apache.org/documentation.html
   Main PID: 8452 (java)
      Tasks: 76 (limit: 3815)
     Memory: 729.1M
        CPU: 34min 32.090s
     CGroup: /system.slice/kafka.service
             └─8452 /usr/lib/jvm/java-8-openjdk-amd64/bin/java -Xmx1G -Xms1G ->

Apr 25 15:44:25 pes2ug20cs461 kafka-server-start.sh[8452]: [2023-04-25 15:44:2>
Apr 25 15:44:25 pes2ug20cs461 kafka-server-start.sh[8452]: [2023-04-25 15:44:2>
Apr 25 15:44:25 pes2ug20cs461 kafka-server-start.sh[8452]: [2023-04-25 15:44:2>
Apr 25 16:01:00 pes2ug20cs461 kafka-server-start.sh[8452]: [2023-04-25 16:01:0>
Apr 25 16:01:00 pes2ug20cs461 kafka-server-start.sh[8452]: [2023-04-25 16:01:0>
Apr 25 16:01:00 pes2ug20cs461 kafka-server-start.sh[8452]: [2023-04-25 16:01:0>
Apr 25 16:01:00 pes2ug20cs461 kafka-server-start.sh[8452]: [2023-04-25 16:01:0>
Apr 25 16:01:00 pes2ug20cs461 kafka-server-start.sh[8452]: [2023-04-25 16:01:0>
Apr 25 16:01:00 pes2ug20cs461 kafka-server-start.sh[8452]: [2023-04-25 16:01:0>
Apr 25 16:01:00 pes2ug20cs461 kafka-server-start.sh[8452]: [2023-04-25 16:01:0>
```

Publish the traffic topics to the consumer

```
pes2ug20cs461@pes2ug20cs461:~$ python3 kafkaproducer.py
23/04/27 14:39:59 WARN Utils: Your hostname, pes2ug20cs461 resolves to a loopback address: 127.0.1.1; us
ing 10.0.2.15 instead (on interface enp0s3)
23/04/27 14:39:59 WARN Utils: Set SPARK_LOCAL_IP if you need to bind to another address
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(newLevel).
23/04/27 14:40:00 WARN NativeCodeLoader: Unable to load native-hadoop library for your platform... using
 builtin-java classes where applicable
Topics Successfully sent
```

b. Workloads

We start the Spark streaming for Receiving Traffic Data from Kafka and processing it:

```
pes2ug20cs461@pes2ug20cs461:~$ python3 kafkaconsumer.py
23/04/26 00:17:04 WARN Utils: Your hostname, pes2ug20cs461 resolves to a loopba
ck address: 127.0.1.1; using 10.0.2.15 instead (on interface enp0s3)
23/04/26 00:17:04 WARN Utils: Set SPARK_LOCAL_IP if you need to bind to another
 address
:: loading settings :: url = jar:file:/opt/spark/jars/ivy-2.5.1.jar!/org/apache
/ivy/core/settings/ivysettings.xml
Ivy Default Cache set to: /home/pes2ug20cs461/.ivy2/cache
The jars for the packages stored in: /home/pes2ug20cs461/.ivy2/jars
org.apache.spark#spark-sql-kafka-0-10_2.12 added as a dependency
:: resolving dependencies :: org.apache.spark#spark-submit-parent-a4c969e6-bfa1
-4240-a61d-ae05ad3777d8;1.0
        confs: [default]
        found org.apache.spark#spark-sql-kafka-0-10_2.12;3.4.0 in central
        found org.apache.spark#spark-token-provider-kafka-0-10_2.12;3.4.0 in ce
ntral
        found org.apache.kafka#kafka-clients;3.3.2 in central
        found org.lz4#lz4-java;1.8.0 in central
        found org.xerial.snappy#snappy-java;1.1.9.1 in central
        found org.slf4j#slf4j-api;2.0.6 in central
        found org.apache.hadoop#hadoop-client-runtime;3.3.4 in central
        found org.apache.hadoop#hadoop-client-api;3.3.4 in central
        found commons-logging#commons-logging;1.1.3 in central
        found com.google.code.findbugs#jsr305;3.0.0 in central
        found org.apache.commons#commons-pool2;2.11.1 in central
:: resolution report :: resolve 618ms :: artifacts dl 28ms
        :: modules in use:
```

```
[Stage 5:============================>                            (98 + 4) / 200]
[Stage 5:=============================>                           (105 + 4) / 200]
[Stage 5:==============================>                          (109 + 4) / 200]
[Stage 5:================================>                        (113 + 5) / 200]
[Stage 5:=================================>                       (117 + 5) / 200]
[Stage 5:==================================>                      (122 + 4) / 200]
[Stage 5:===================================>                     (126 + 4) / 200]
[Stage 5:=====================================>                   (131 + 4) / 200]
[Stage 5:======================================>                  (136 + 5) / 200]
[Stage 5:=======================================>                 (141 + 4) / 200]
[Stage 5:========================================>                (146 + 4) / 200]
[Stage 5:========================================>                (149 + 4) / 200]
[Stage 5:=========================================>               (153 + 4) / 200]
[Stage 5:==========================================>              (157 + 4) / 200]
[Stage 5:===========================================>             (161 + 4) / 200]
[Stage 5:============================================>            (165 + 4) / 200]
[Stage 5:============================================>            (170 + 4) / 200]
[Stage 5:=============================================>           (175 + 4) / 200]
[Stage 5:==============================================>          (179 + 4) / 200]
[Stage 5:==============================================>          (181 + 4) / 200]
[Stage 5:===============================================>         (185 + 4) / 200]
[Stage 5:================================================>        (189 + 4) / 200]
[Stage 5:================================================>        (192 + 4) / 200]
[Stage 5:=================================================>       (196 + 4) / 200]
```

```
23/04/27 21:52:47 WARN AdminClientConfig: These configurations '[key.deserializer, value.deserializer, e
nable.auto.commit, max.poll.records, auto.offset.reset]' were supplied but are not used yet.
23/04/27 21:52:47 WARN AdminClientConfig: These configurations '[key.deserializer, value.deserializer, e
nable.auto.commit, max.poll.records, auto.offset.reset]' were supplied but are not used yet.
23/04/27 21:52:47 WARN AdminClientConfig: These configurations '[key.deserializer, value.deserializer, e
nable.auto.commit, max.poll.records, auto.offset.reset]' were supplied but are not used yet.
23/04/27 21:53:07 WARN ProcessingTimeExecutor: Current batch is falling behind. The trigger interval is
5000 milliseconds, but spent 20609 milliseconds
23/04/27 21:53:16 WARN ProcessingTimeExecutor: Current batch is falling behind. The trigger interval is
5000 milliseconds, but spent 28936 milliseconds
23/04/27 21:53:22 WARN ProcessingTimeExecutor: Current batch is falling behind. The trigger interval is
5000 milliseconds, but spent 34853 milliseconds
duration 40.66959595680237
```

Results are written to the MYSQL database in a separate table:

Server: localhost » Database: trafficdata » Table: traffic_results

start	end	location	avg_speed
2022-09-01 00:00:00	2022-09-01 00:05:00	Los Angeles34	44
2022-09-01 00:00:00	2022-09-01 00:05:00	Los Angeles89	33
2022-09-01 00:00:00	2022-09-01 00:05:00	Los Angeles61	54
2022-09-01 00:00:00	2022-09-01 00:05:00	Los Angeles85	90
2022-09-01 00:00:00	2022-09-01 00:05:00	Los Angeles92	90
2022-09-01 00:00:00	2022-09-01 00:05:00	Los Angeles84	43
2022-09-01 00:00:00	2022-09-01 00:05:00	Los Angeles40	65
2022-09-01 00:00:00	2022-09-01 00:05:00	Los Angeles5	83
2022-09-01 00:00:00	2022-09-01 00:05:00	Los Angeles41	41
2022-09-01 00:00:00	2022-09-01 00:05:00	Los Angeles28	22
2022-09-01 00:00:00	2022-09-01 00:05:00	Los Angeles22	92
2022-09-01 00:00:00	2022-09-01 00:05:00	Los Angeles25	98
2022-09-01 00:00:00	2022-09-01 00:05:00	Los Angeles29	22
2022-09-01 00:00:00	2022-09-01 00:05:00	Los Angeles73	55
2022-09-01 00:00:00	2022-09-01 00:05:00	Los Angeles71	23
2022-09-01 00:00:00	2022-09-01 00:05:00	Los Angeles39	27
2022-09-01 00:00:00	2022-09-01 00:05:00	Los Angeles2	73
2022-09-01 00:00:00	2022-09-01 00:05:00	Los Angeles23	16
2022-09-01 00:00:00	2022-09-01 00:05:00	Los Angeles4	40
2022-09-01 00:00:00	2022-09-01 00:05:00	Los Angeles87	72

c. Code like SQL Scripts

```
result1 = trafficDF1 \
    .filter(col("location").like("%Los Angeles%")) \
    .groupBy(window("timestamp", "5 minutes").alias("window"), "location") \
    .agg(avg("speed").alias("avg_speed")) \
    .filter(col("avg_speed")>10)\
    .select("window.start", "window.end", "location", "avg_speed")


result2 = trafficDF2 \
    .filter(col("location").like("%San Francisco%")) \
    .groupBy(window("timestamp", "5 minutes").alias("window"), "location") \
    .agg(avg("speed").alias("avg_speed")) \
    .filter(col("avg_speed")>10)\
    .select("window.start", "window.end", "location", "avg_speed")



result3 = trafficDF3 \
    .filter(col("location").like("%New York%")) \
    .groupBy(window("timestamp", "5 minutes").alias("window"), "location") \
    .agg(avg("speed").alias("avg_speed")) \
    .filter(col("avg_speed")>10)\
    .select("window.start", "window.end", "location", "avg_speed")
```

```
def write_to_mysql(df, epoch_id):
    # define the connection properties
    properties = {
        "user": "root",
        "password": ""
    }
    # write the DataFrame to MySQL
    df.write.jdbc(
        url="jdbc:mysql://localhost:3306/trafficdata",
        table="traffic_results",
        mode="append",
        properties=properties)
```

```
# define the streaming query
start_time=time.time()
result1.writeStream \
    .outputMode("complete") \
    .foreachBatch(write_to_mysql) \
    .trigger(processingTime='5 seconds') \
    .start()

result2.writeStream \
    .outputMode("complete") \
    .foreachBatch(write_to_mysql) \
    .trigger(processingTime='5 seconds') \
    .start()

result3.writeStream \
    .outputMode("complete") \
    .foreachBatch(write_to_mysql) \
    .trigger(processingTime='5 seconds') \
    .start()
```

The code defines three separate queries to calculate the average speed of the vehicles in three different cities (Los Angeles, San Francisco, and New York) over a 5-minute window. The resulting data is filtered to include only those vehicles with an average speed greater than 10 miles per hour. Finally, the query results are written to a MySQL database using the "writeStream" API.

The "foreachBatch" function is used to apply the "write_to_mysql" function to each batch of data generated by the query. The "trigger" function is used to set the processing time for each batch to 5 seconds. The "start" function is used to start the streaming query.

d. Inputs and Corresponding Results

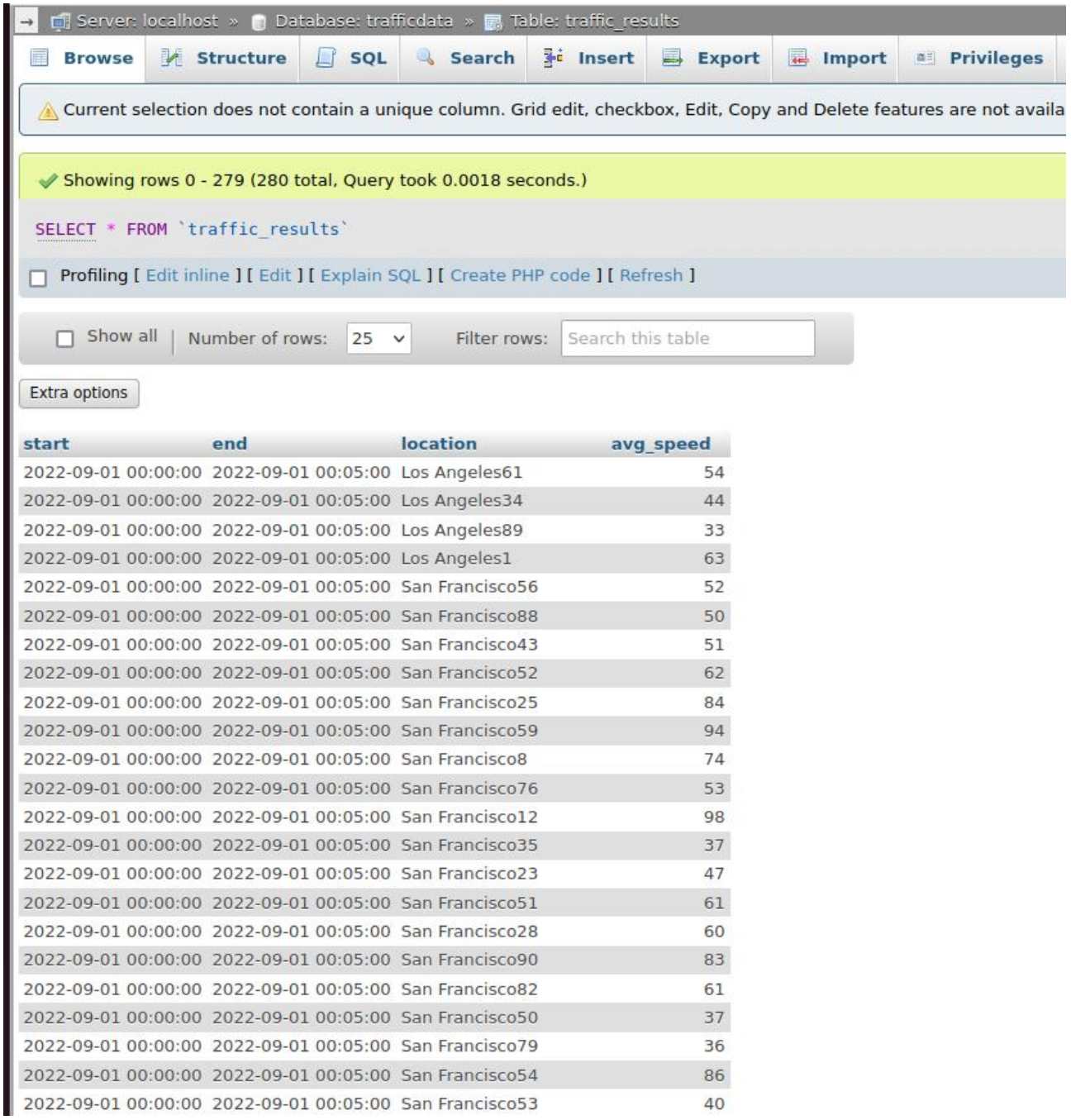
These are three separate Spark SQL queries that process data from three different data frames (trafficDF1, trafficDF2, and trafficDF3), each for a specific location (Los Angeles, New York, and San Francisco respectively), and calculate the average speed greater than 10 over a 5-minute tumbling window.

Each query starts with filtering the data frame to only include the records with the specified location, then groups the records by a 5-minute window based on the "timestamp" column and the "location" column. The aggregate function "avg" is applied to the "speed" column of the grouped data, and the resulting column is aliased as "avg_speed" and checks if it is greater than 10. Finally, the resulting data frame is selected to only include the "start" and "end" timestamps of the window, the "location", and the "avg_speed" columns.

The resulting data frames (result1, result2, result3) will each contain the average speed which is greater than 10 for their respective location over 5-minute windows.

```
trafficDF1 = trafficDF1 \
    .selectExpr("CAST(value AS STRING)") \
    .selectExpr("from_json(value, 'timestamp TIMESTAMP, location STRING, speed FLOAT') AS data") \
    .selectExpr("data.timestamp", "data.location", "data.speed")


trafficDF2 = trafficDF2 \
    .selectExpr("CAST(value AS STRING)") \
    .selectExpr("from_json(value, 'timestamp TIMESTAMP, location STRING, speed FLOAT') AS data") \
    .selectExpr("data.timestamp", "data.location", "data.speed")

# Perform some queries using Spark SQL

trafficDF3 = trafficDF3 \
    .selectExpr("CAST(value AS STRING)") \
    .selectExpr("from_json(value, 'timestamp TIMESTAMP, location STRING, speed FLOAT') AS data") \
    .selectExpr("data.timestamp", "data.location", "data.speed")



result1 = trafficDF1 \
    .filter(col("location").like("%Los Angeles%")) \
    .groupBy(window("timestamp", "5 minutes").alias("window"), "location") \
    .agg(avg("speed").alias("avg_speed")) \
    .filter(col("avg_speed")>10)\
    .select("window.start", "window.end", "location", "avg_speed")


result2 = trafficDF2 \
    .filter(col("location").like("%San Francisco%")) \
    .groupBy(window("timestamp", "5 minutes").alias("window"), "location") \
    .agg(avg("speed").alias("avg_speed")) \
    .filter(col("avg_speed")>10)\
    .select("window.start", "window.end", "location", "avg_speed")


result3 = trafficDF3 \
    .filter(col("location").like("%New York%")) \
    .groupBy(window("timestamp", "5 minutes").alias("window"), "location") \
    .agg(avg("speed").alias("avg_speed")) \
    .filter(col("avg_speed")>10)\
    .select("window.start", "window.end", "location", "avg_speed")
```

Results get stored in the new database table ‘traffic_results’:



start	end	location	avg_speed
2022-09-01 00:00:00	2022-09-01 00:05:00	Los Angeles61	54
2022-09-01 00:00:00	2022-09-01 00:05:00	Los Angeles34	44
2022-09-01 00:00:00	2022-09-01 00:05:00	Los Angeles89	33
2022-09-01 00:00:00	2022-09-01 00:05:00	Los Angeles1	63
2022-09-01 00:00:00	2022-09-01 00:05:00	San Francisco56	52
2022-09-01 00:00:00	2022-09-01 00:05:00	San Francisco88	50
2022-09-01 00:00:00	2022-09-01 00:05:00	San Francisco43	51
2022-09-01 00:00:00	2022-09-01 00:05:00	San Francisco52	62
2022-09-01 00:00:00	2022-09-01 00:05:00	San Francisco25	84
2022-09-01 00:00:00	2022-09-01 00:05:00	San Francisco59	94
2022-09-01 00:00:00	2022-09-01 00:05:00	San Francisco8	74
2022-09-01 00:00:00	2022-09-01 00:05:00	San Francisco76	53
2022-09-01 00:00:00	2022-09-01 00:05:00	San Francisco12	98
2022-09-01 00:00:00	2022-09-01 00:05:00	San Francisco35	37
2022-09-01 00:00:00	2022-09-01 00:05:00	San Francisco23	47
2022-09-01 00:00:00	2022-09-01 00:05:00	San Francisco51	61
2022-09-01 00:00:00	2022-09-01 00:05:00	San Francisco28	60
2022-09-01 00:00:00	2022-09-01 00:05:00	San Francisco90	83
2022-09-01 00:00:00	2022-09-01 00:05:00	San Francisco82	61
2022-09-01 00:00:00	2022-09-01 00:05:00	San Francisco50	37
2022-09-01 00:00:00	2022-09-01 00:05:00	San Francisco79	36
2022-09-01 00:00:00	2022-09-01 00:05:00	San Francisco54	86
2022-09-01 00:00:00	2022-09-01 00:05:00	San Francisco53	40

7. Batch Mode Experiment:

a. Description:

This is a batch mode experiment that performs some queries using Spark SQL on a MySQL database. First, it establishes a connection to the database and reads the data from the "traffic_data" table into a Spark DataFrame. Then it defines three different Spark queries using the tumbling window technique to aggregate the data and calculates the average speed for three different locations: Los Angeles, New York, and San Francisco. The results are printed using the show() method.

```
# Connect to MySQL
mydb=mysql.connector.connect(
    host="localhost",
    user="root",
    password="",
    database="trafficdata"
)

cursor = mydb.cursor()

spark =
SparkSession.builder.appName("TrafficData").config("spark.driver.extraClassPa
usr/share/java/mysql-connector-java-8.0.33.jar").getOrCreate()


trafficDF = spark.read \
            .format("jdbc")\
            .option("url", "jdbc:mysql://localhost/trafficdata") \
            .option("dbtable","traffic_data") \
            .option("user", "root") \
            .option("password","")\
            .load()
```

```
result1 = trafficDF \
    .filter(col("location").like("%Los Angeles%")) \
    .groupBy(window("timestamp", "5 minutes").alias("window"), "location") \
    .agg(avg("speed").alias("avg_speed")) \
    .filter(col("avg_speed")>10)\
    .select("window.start", "window.end", "location", "avg_speed")

result2 = trafficDF \
    .filter(col("location").like("%San Francisco%")) \
    .groupBy(window("timestamp", "5 minutes").alias("window"), "location") \
    .agg(avg("speed").alias("avg_speed")) \
    .filter(col("avg_speed")>10)\
    .select("window.start", "window.end", "location", "avg_speed")

result3 = trafficDF \
    .filter(col("location").like("%New York%")) \
    .groupBy(window("timestamp", "5 minutes").alias("window"), "location") \
    .agg(avg("speed").alias("avg_speed")) \
    .filter(col("avg_speed")>10)\
    .select("window.start", "window.end", "location", "avg_speed")

result1.createOrReplaceTempView("result1_temp_view")
result2.createOrReplaceTempView("result2_temp_view")
result3.createOrReplaceTempView("result3_temp_view")
```

```
result1.createOrReplaceTempView("result1_temp_view")
result2.createOrReplaceTempView("result2_temp_view")
result3.createOrReplaceTempView("result3_temp_view")


start_time=time.time()
#result1.show()
#result2.show()
#result3.show()
#spark.streams.awaitAnyTermination(timeout=40)
spark.sql("SELECT * FROM result1_temp_view").show()
spark.sql("SELECT * FROM result2_temp_view").show()
spark.sql("SELECT * FROM result3_temp_view").show()
```

b. Data Size and Results:

The data size in batch mode would depend on the specific data set being used. In this case, the data size can be determined by examining the "traffic_data" table in the MySQL database that is being read by the Spark application. The size of this table can be obtained using a query such as:

SELECT COUNT(*) FROM traffic_data

This will return the number of rows in the table, which can be used as an estimate for the data size in batch mode.

Results: The results are printed using the show() method which prints the results on the terminal.

```
pes2ug20cs461@pes2ug20cs461:~$ python3 kafkaconsumer1.py
23/04/27 21:45:42 WARN Utils: Your hostname, pes2ug20cs461 resolves to a loopback address: 127.0.1.1; us
ing 10.0.2.15 instead (on interface enp0s3)
23/04/27 21:45:42 WARN Utils: Set SPARK_LOCAL_IP if you need to bind to another address
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(newLevel).
23/04/27 21:45:43 WARN NativeCodeLoader: Unable to load native-hadoop library for your platform... using
 builtin-java classes where applicable
[Stage 0:>                                                          (0 + 1) / 1
                                                +-------------------+-------------------+---------
----+---------+
|              start|                end|     location|avg_speed|
+-------------------+-------------------+-------------+---------+
|2022-09-01 00:00:00|2022-09-01 00:05:00| Los Angeles1|     63.0|
|2022-09-01 00:00:00|2022-09-01 00:05:00|Los Angeles34|     44.0|
|2022-09-01 00:00:00|2022-09-01 00:05:00|Los Angeles61|     54.0|
|2022-09-01 00:00:00|2022-09-01 00:05:00|Los Angeles89|     33.0|
|2022-09-01 00:00:00|2022-09-01 00:05:00|Los Angeles84|     43.0|
|2022-09-01 00:00:00|2022-09-01 00:05:00|Los Angeles85|     90.0|
|2022-09-01 00:00:00|2022-09-01 00:05:00|Los Angeles92|     90.0|
|2022-09-01 00:00:00|2022-09-01 00:05:00|Los Angeles40|     65.0|
|2022-09-01 00:00:00|2022-09-01 00:05:00|Los Angeles28|     22.0|
|2022-09-01 00:00:00|2022-09-01 00:05:00|Los Angeles41|     41.0|
|2022-09-01 00:00:00|2022-09-01 00:05:00| Los Angeles5|     83.0|
|2022-09-01 00:00:00|2022-09-01 00:05:00|Los Angeles22|     92.0|
|2022-09-01 00:00:00|2022-09-01 00:05:00|Los Angeles73|     55.0|
|2022-09-01 00:00:00|2022-09-01 00:05:00|Los Angeles25|     98.0|
|2022-09-01 00:00:00|2022-09-01 00:05:00|Los Angeles29|     22.0|
|2022-09-01 00:00:00|2022-09-01 00:05:00|Los Angeles71|     23.0|
|2022-09-01 00:00:00|2022-09-01 00:05:00|Los Angeles39|     27.0|
|2022-09-01 00:00:00|2022-09-01 00:05:00| Los Angeles2|     73.0|
|2022-09-01 00:00:00|2022-09-01 00:05:00| Los Angeles4|     40.0|
|2022-09-01 00:00:00|2022-09-01 00:05:00|Los Angeles23|     16.0|
+-------------------+-------------------+-------------+---------+
only showing top 20 rows
```

start	end	location	avg_speed
2022-09-01 00:00:00	2022-09-01 00:05:00	San Francisco43	51.0
2022-09-01 00:00:00	2022-09-01 00:05:00	San Francisco88	50.0
2022-09-01 00:00:00	2022-09-01 00:05:00	San Francisco52	62.0
2022-09-01 00:00:00	2022-09-01 00:05:00	San Francisco25	84.0
2022-09-01 00:00:00	2022-09-01 00:05:00	San Francisco56	52.0
2022-09-01 00:00:00	2022-09-01 00:05:00	San Francisco59	94.0
2022-09-01 00:00:00	2022-09-01 00:05:00	San Francisco12	98.0
2022-09-01 00:00:00	2022-09-01 00:05:00	San Francisco76	53.0
2022-09-01 00:00:00	2022-09-01 00:05:00	San Francisco8	74.0
2022-09-01 00:00:00	2022-09-01 00:05:00	San Francisco35	37.0
2022-09-01 00:00:00	2022-09-01 00:05:00	San Francisco23	47.0
2022-09-01 00:00:00	2022-09-01 00:05:00	San Francisco51	61.0
2022-09-01 00:00:00	2022-09-01 00:05:00	San Francisco28	60.0
2022-09-01 00:00:00	2022-09-01 00:05:00	San Francisco50	37.0
2022-09-01 00:00:00	2022-09-01 00:05:00	San Francisco82	61.0
2022-09-01 00:00:00	2022-09-01 00:05:00	San Francisco90	83.0
2022-09-01 00:00:00	2022-09-01 00:05:00	San Francisco79	36.0
2022-09-01 00:00:00	2022-09-01 00:05:00	San Francisco54	86.0
2022-09-01 00:00:00	2022-09-01 00:05:00	San Francisco99	47.0
2022-09-01 00:00:00	2022-09-01 00:05:00	San Francisco53	40.0

only showing top 20 rows

start	end	location	avg_speed
2022-09-01 00:00:00	2022-09-01 00:05:00	New York89	36.0
2022-09-01 00:00:00	2022-09-01 00:05:00	New York77	72.0
2022-09-01 00:00:00	2022-09-01 00:05:00	New York19	82.2
2022-09-01 00:00:00	2022-09-01 00:05:00	New York51	90.6
2022-09-01 00:00:00	2022-09-01 00:05:00	New York65	62.8
2022-09-01 00:00:00	2022-09-01 00:05:00	New York57	45.6
2022-09-01 00:00:00	2022-09-01 00:05:00	New York68	50.4
2022-09-01 00:00:00	2022-09-01 00:05:00	New York20	53.0
2022-09-01 00:00:00	2022-09-01 00:05:00	New York25	32.6
2022-09-01 00:00:00	2022-09-01 00:05:00	New York67	61.6
2022-09-01 00:00:00	2022-09-01 00:05:00	New York45	58.2
2022-09-01 00:00:00	2022-09-01 00:05:00	New York5	33.6
2022-09-01 00:00:00	2022-09-01 00:05:00	New York4	54.4
2022-09-01 00:00:00	2022-09-01 00:05:00	New York27	55.0
2022-04-23 01:20:00	2022-04-23 01:25:00	New York	50.0
2022-09-01 00:00:00	2022-09-01 00:05:00	New York71	59.2
2022-09-01 00:00:00	2022-09-01 00:05:00	New York83	28.2
2022-09-01 00:00:00	2022-09-01 00:05:00	New York32	52.4
2022-09-01 00:00:00	2022-09-01 00:05:00	New York87	55.0
2022-09-01 00:00:00	2022-09-01 00:05:00	New York61	46.0

only showing top 20 rows

8. Comparison of Streaming & Batch Modes:

a. Results and Discussion:

```
23/04/27 21:53:07 WARN ProcessingTimeExecutor: Current batch is falling behind. The trigger interval i
5000 milliseconds, but spent 20609 milliseconds
23/04/27 21:53:16 WARN ProcessingTimeExecutor: Current batch is falling behind. The trigger interval i
5000 milliseconds, but spent 28936 milliseconds
23/04/27 21:53:22 WARN ProcessingTimeExecutor: Current batch is falling behind. The trigger interval i
5000 milliseconds, but spent 34853 milliseconds
duration 40.66959595680237
```

In the case of streaming mode, we see that the processing time of one or more batches exceeded the the trigger interval of 5000 milliseconds , as indicated in the warning message. This suggests that the workload may be too high or the processing resources may not be sufficient to handle the workload efficiently. Hence the duration is higher.

```
+-------------------+-------------------+----------+---------+
|              start|                end|  location|avg_speed|
+-------------------+-------------------+----------+---------+
|2022-09-01 00:00:00|2022-09-01 00:05:00|New York89|     36.0|
|2022-09-01 00:00:00|2022-09-01 00:05:00|New York77|     72.0|
|2022-09-01 00:00:00|2022-09-01 00:05:00|New York19|     82.2|
|2022-09-01 00:00:00|2022-09-01 00:05:00|New York51|     90.6|
|2022-09-01 00:00:00|2022-09-01 00:05:00|New York65|     62.8|
|2022-09-01 00:00:00|2022-09-01 00:05:00|New York57|     45.6|
|2022-09-01 00:00:00|2022-09-01 00:05:00|New York68|     50.4|
|2022-09-01 00:00:00|2022-09-01 00:05:00|New York20|     53.0|
|2022-09-01 00:00:00|2022-09-01 00:05:00|New York25|     32.6|
|2022-09-01 00:00:00|2022-09-01 00:05:00|New York67|     61.6|
|2022-09-01 00:00:00|2022-09-01 00:05:00|New York45|     58.2|
|2022-09-01 00:00:00|2022-09-01 00:05:00| New York5|     33.6|
|2022-09-01 00:00:00|2022-09-01 00:05:00| New York4|     54.4|
|2022-09-01 00:00:00|2022-09-01 00:05:00|New York27|     55.0|
|2022-04-23 01:20:00|2022-04-23 01:25:00|  New York|     50.0|
|2022-09-01 00:00:00|2022-09-01 00:05:00|New York71|     59.2|
|2022-09-01 00:00:00|2022-09-01 00:05:00|New York83|     28.2|
|2022-09-01 00:00:00|2022-09-01 00:05:00|New York32|     52.4|
|2022-09-01 00:00:00|2022-09-01 00:05:00|New York87|     55.0|
|2022-09-01 00:00:00|2022-09-01 00:05:00|New York61|     46.0|
+-------------------+-------------------+----------+---------+
only showing top 20 rows

duration 3.501192092895508
```

On the other hand, in the batch mode, the duration is much lower, indicating that the processing resources are capable of handling the workload efficiently.

From the given information, we can conclude that the batch mode has better performance than the streaming mode in terms of processing time.

9. Conclusion:

In batch processing, data is processed in fixed intervals of time or when a specific batch size is reached. All data in the batch is available at the start of the processing, and the output is usually written to a file or database. Batch processing is well-suited for large volumes of data that can be processed offline, such as historical data analysis.

In contrast, streaming processing processes data in real-time as it is generated. Data is processed continuously, and the output is often written to a sink or stream. Streaming processing is well-suited for applications that require real-time data processing and analysis, such as fraud detection, stock trading, and social media sentiment analysis.

In the code provided, Spark is used for both batch and streaming processing. In batch mode, traffic data is read from the database and processed using Spark SQL queries. In streaming mode, the data is read from a stream and processed in real-time, and the results are also written to the same MySQL database.

10. References:

Kafka Documentation: kafka.apache.org
Spark Documentation: spark.apache.org
Cassandra Documentation: cassandra.apache.org