PES UNIVERSITY
100 feet Ring Road, BSK 3rd Stage
Bengaluru 560085 INDIA

Department of Computer Science and Engineering
B. Tech. CSE – 6th Semester
Jan – May 2024

UE21CS343BB3
DATABASE TECHNOLOGIES (DBT)

PROJECT REPORT
on

Stream and Batch Processing of Water Quality

Submitted by : Team #: 512_538_555_902

Sahana Parasuram	PES1UG21CS512	6I	Sharath M S	PES1UG21CS555	6I
Saransh Mehta	PES1UG21CS538	6I	Aditya Vishwanatha	PES1UG21CS902	6I

Class of Prof. Raghu B. A.

Table of Contents

1. Introduction

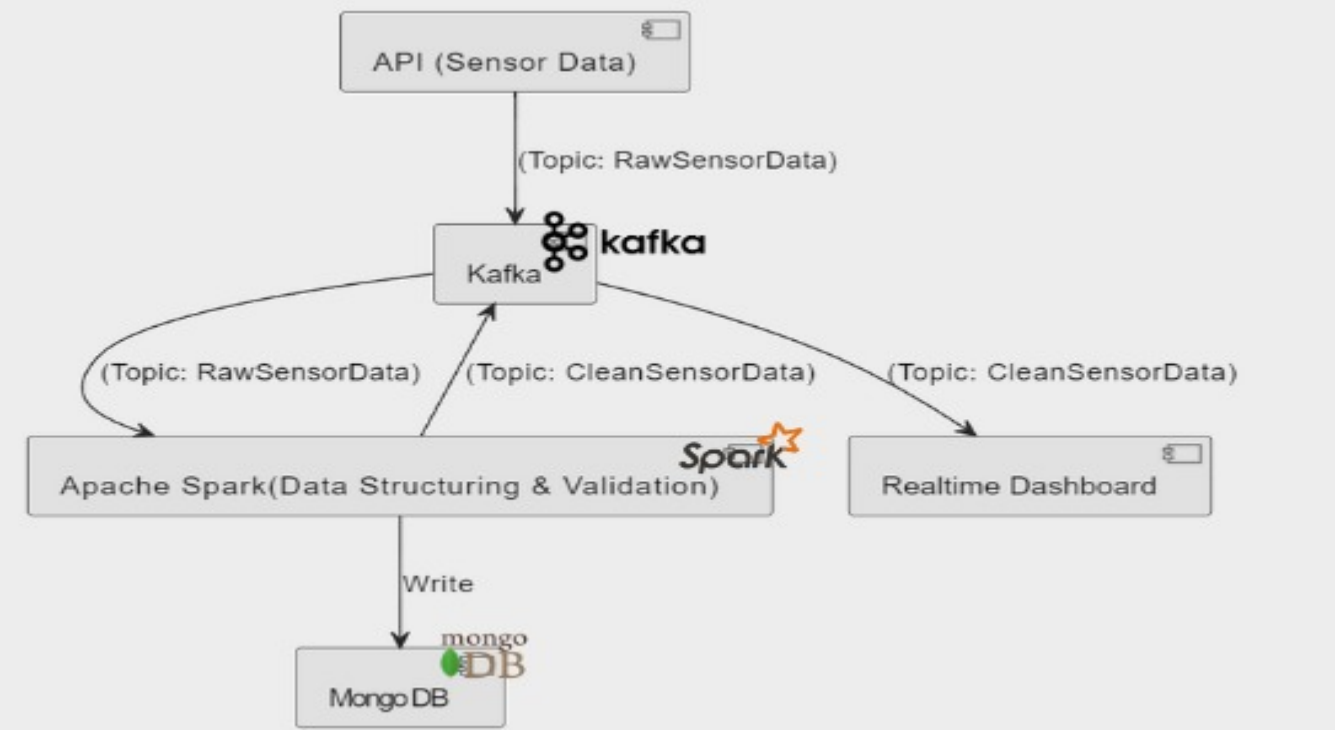
This project develops a real-time data pipeline utilizing Apache Kafka and Apache Spark to handle and analyze water quality data. The primary goal is to demonstrate effective real-time data processing compared to traditional batch processing, using simulated sensor data to model water quality metrics.

Problem Description

Water quality monitoring is vital for environmental health and public safety. Traditional batch processing methods can delay the action needed to address pollution or contamination events. This project explores a real-time processing solution to provide immediate insights into water quality data.

Solution Architecture

- API: Simulates water quality sensor data, mirroring actual sensor outputs.
- Streaming Apps/Tools: Uses Apache Kafka for real-time data ingestion and Apache Spark for on-the-fly data processing.
- Repository/Database: MongoDB for data persistence post-analysis, enabling further historical analysis.



Architecture Diagram

2. Installation of Software

- **Apache Kafka (3.7.0)** : a distributed streaming platform that makes it possible to create streaming apps and real-time data pipelines. We utilized this as a publisher and for data streaming.
  https://downloads.apache.org/kafka/3.7.0/kafka_2.12-3.7.0.tgz

- **Apache Spark Streaming (3.5.1)** : Built on top of the Apache Spark platform, a real-time processing engine enables real-time data stream processing. permits batch processing as well.
  https://www.apache.org/dyn/closer.lua/spark/spark-3.5.1/spark-3.5.1-bin-hadoop3.tgz

- **Pyspark (3.5.1) :** A streaming engine designed specifically for Python and its related features, similar to Apache Kafka.
  TO INSTALL : pip install pyspark==”3.5.1”

- **DBMS Used :** MongoDB - 7.0.8
  https://www.mongodb.org/static/pgp/server-7.0.asc

Streaming Apps/Tools Used

Apache Kafka:

Apache Kafka facilitates the publishing and subscribing of streams of records, which enables real-time data processing. Kafka provides fault tolerance, scalability, and high throughput, making it suitable for building real-time data pipelines.

Apache Spark Streaming:

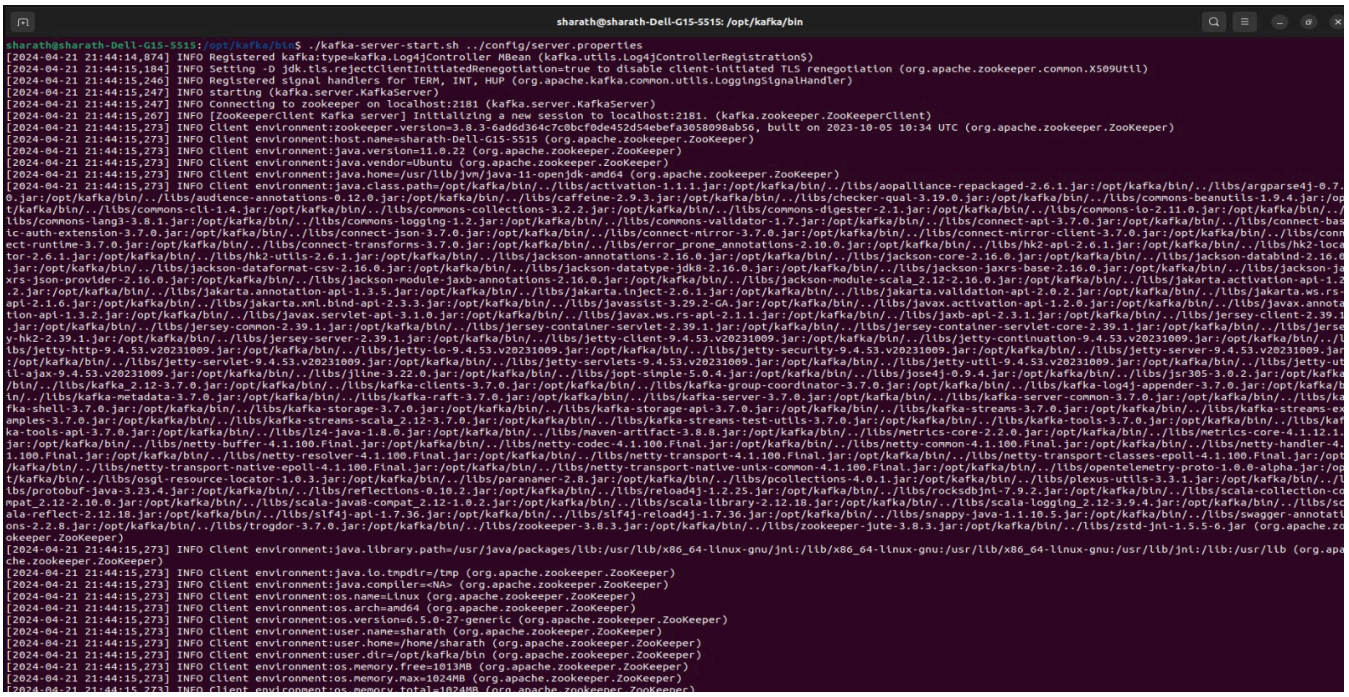
Apache Spark Streaming is used for processing real-time data streams from Kafka. It allows for scalable, high-throughput, fault-tolerant stream processing of live data streams. Spark Streaming enables the application of complex algorithms and analytics on real-time data.

MongoDB:

MongoDB is a NoSQL database used for storing real-time sensor data processed by Spark Streaming.

Zookeeper:

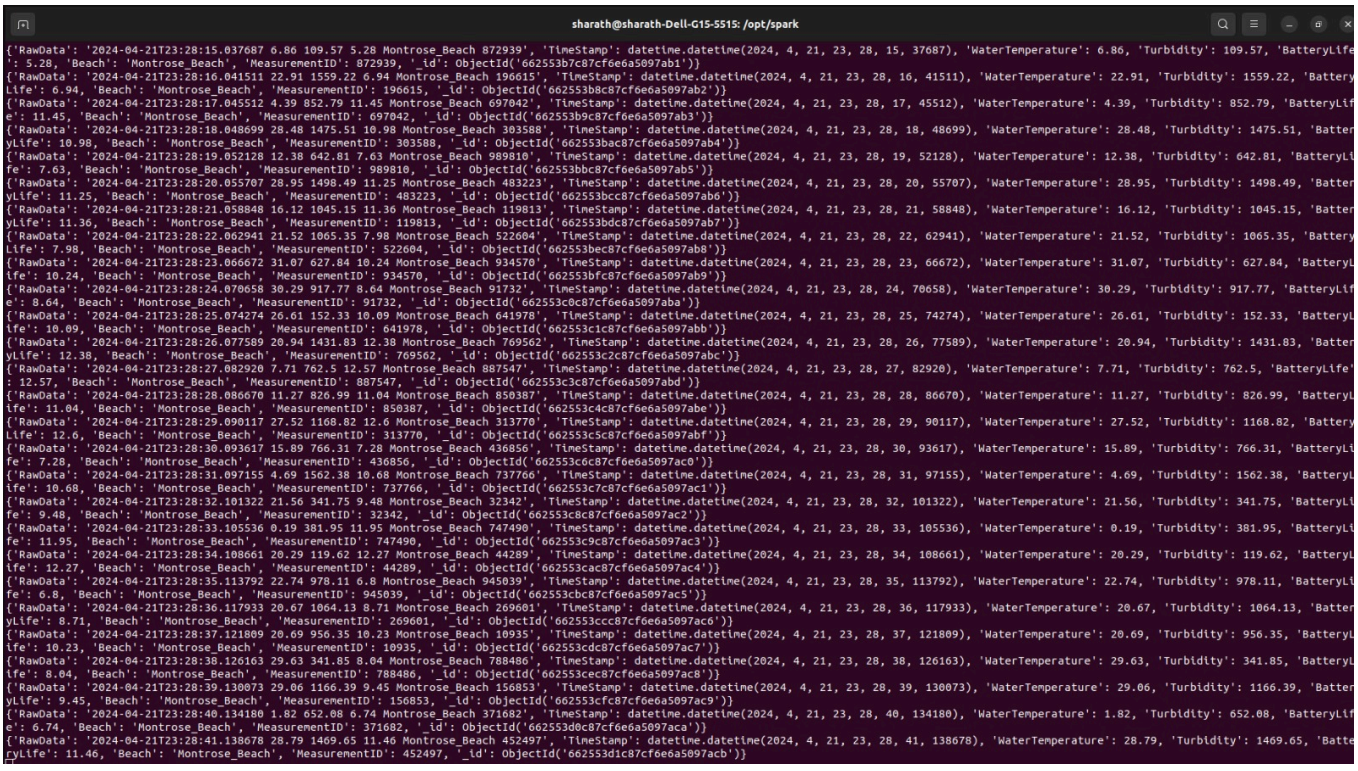
In this project, Zookeeper is used for managing configurations and coordination in the Kafka ecosystem.



The above figure represents starting the Kafka server using the specified configuration file.

Database

Mongodb is utilized in our project as a NoSQL database, which stores data in JSON-like documents with dynamic schemas (BSON), making the integration of data in certain types of applications easier and faster. In our project, MongoDB stores real-time sensor data, such as water temperature, in a collection within the 'RealTimeDB' database. This approach enables efficient data storage and fast retrieval capabilities, essential for performing real-time analytics and batch processing. The use of aggregation queries in MongoDB facilitates complex analytics, such as calculating average temperatures from accumulated data, directly within the database, thereby optimizing performance and scalability.



The above figure represents the data stored in Mongodb after streaming.

Kafka Topics

RawSensorData:
This topic is used to store raw sensor data obtained from the API. Data published to this topic

includes information such as timestamp, water temperature, turbidity, battery life, beach, and measurement ID. Raw sensor data is streamed into this topic by the push_data_to_kafka.py script.

ValidatedSensorData:
This topic acts as an intermediate step for validated sensor data before it's persisted into MongoDB. After the sensor data is received from the RawSensorData topic, it undergoes validation to ensure its integrity and correctness. Validated sensor data is published to this topic by the structure_validate_store.py script.

CleanSensorData:
This topic stores the cleaned and validated sensor data after it's persisted into MongoDB. Once the data is validated and stored in MongoDB, it's published to this topic by the structure_validate_store.py script. Cleaned sensor data from this topic can be consumed by downstream systems, such as real-time dashboards, for visualization or further analysis.

3. Input Data

a. **Sources :** Beach water quality API from the sensors in the water at beaches along Chicago's lake Michigan

https://data.world/cityofchicago/beach-water-quality-automated-sensors

```
^Csharath@sharath-Dell-G15-5515:~/Desktop/Real-Time-Data-Pipeline-Using-Kafka-and-Spark-master$ python3 sensor.py
 * Serving Flask app 'sensor'
 * Debug mode: off
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
 * Running on all addresses (0.0.0.0)
 * Running on http://127.0.0.1:3030
 * Running on http://192.168.29.30:3030
Press CTRL+C to quit
127.0.0.1 - - [21/Apr/2024 22:38:14] "GET /sensordata HTTP/1.1" 200 -
127.0.0.1 - - [21/Apr/2024 22:38:15] "GET /sensordata HTTP/1.1" 200 -
127.0.0.1 - - [21/Apr/2024 22:38:16] "GET /sensordata HTTP/1.1" 200 -
127.0.0.1 - - [21/Apr/2024 22:38:17] "GET /sensordata HTTP/1.1" 200 -
127.0.0.1 - - [21/Apr/2024 22:38:18] "GET /sensordata HTTP/1.1" 200 -
127.0.0.1 - - [21/Apr/2024 22:38:19] "GET /sensordata HTTP/1.1" 200 -
127.0.0.1 - - [21/Apr/2024 22:38:20] "GET /sensordata HTTP/1.1" 200 -
127.0.0.1 - - [21/Apr/2024 22:38:21] "GET /sensordata HTTP/1.1" 200 -
127.0.0.1 - - [21/Apr/2024 22:38:22] "GET /sensordata HTTP/1.1" 200 -
127.0.0.1 - - [21/Apr/2024 22:38:23] "GET /sensordata HTTP/1.1" 200 -
127.0.0.1 - - [21/Apr/2024 22:38:24] "GET /sensordata HTTP/1.1" 200 -
127.0.0.1 - - [21/Apr/2024 22:38:25] "GET /sensordata HTTP/1.1" 200 -
127.0.0.1 - - [21/Apr/2024 22:38:26] "GET /sensordata HTTP/1.1" 200 -
127.0.0.1 - - [21/Apr/2024 22:38:27] "GET /sensordata HTTP/1.1" 200 -
127.0.0.1 - - [21/Apr/2024 22:38:28] "GET /sensordata HTTP/1.1" 200 -
127.0.0.1 - - [21/Apr/2024 22:38:29] "GET /sensordata HTTP/1.1" 200 -
127.0.0.1 - - [21/Apr/2024 22:38:30] "GET /sensordata HTTP/1.1" 200 -
127.0.0.1 - - [21/Apr/2024 22:38:31] "GET /sensordata HTTP/1.1" 200 -
127.0.0.1 - - [21/Apr/2024 22:38:32] "GET /sensordata HTTP/1.1" 200 -
127.0.0.1 - - [21/Apr/2024 22:38:33] "GET /sensordata HTTP/1.1" 200 -
```

b. Description : The Chicago Park District maintains sensors in the water at beaches along Chicago's Lake Michigan lakefront. These sensors generally capture the indicated measurements hourly while the sensors are in operation during the summer. During other seasons and at some other times, information from the sensors may not be available. See https://data.cityofchicago.org/d/k7hf-8y75 for a dataset with land-based weather measurements at selected beaches. The sensor locations are listed at https://data.cityofchicago.org/d/g3ip-u8rb. Please note that sensor locations change with the Park District's operational needs, primarily related to water quality.

4. Streaming Mode Experiment

a. Description : This experiment is designed to demonstrate the system's ability to handle and analyze continuous data streams using a combination of Apache Kafka and Apache Spark Streaming. Kafka serves as the initial ingestion point, reliably capturing live data streams that simulate environmental sensor outputs such as water temperature and turbidity. Once ingested, data is forwarded to Spark Streaming, which is configured to process this data in real time. The integration of these two technologies ensures seamless data flow and immediate processing, which is crucial for scenarios requiring rapid analytical feedback and decision-making.

b. Windows :

Time Windows: The streaming data is processed using a windowing technique, where data is grouped into fixed windows of one minute each. This approach allows the system to perform time-based aggregations and analyses, such as calculating the average temperature or detecting outliers and anomalies within each window. The use of windowing not only simplifies the processing of continuous streams by breaking down the infinite stream into manageable chunks but also enables the application of complex analytical models on a per-window basis.

Window Operations: Within each window, Spark Streaming executes several functions—aggregating temperature data to compute averages and applying statistical

models to detect sudden changes that could indicate environmental anomalies. These operations are critical for applications like environmental monitoring, where timely detection of data trends and anomalies can lead to prompt action to mitigate potential hazards.

dashboard.py

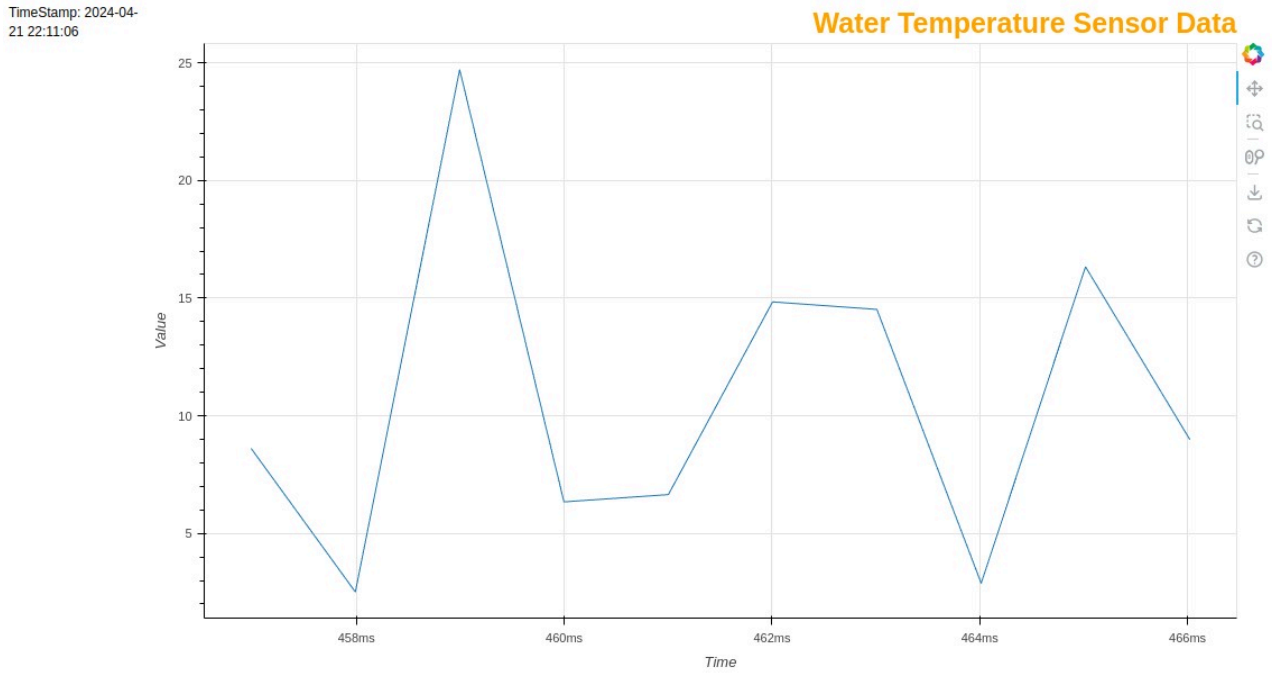
```python
import random
import pandas as pd
from bokeh.driving import count
from bokeh.models import ColumnDataSource
from kafka import KafkaConsumer
from bokeh.plotting import curdoc, figure
from bokeh.models import DatetimeTickFormatter
from bokeh.models.widgets import Div
from bokeh.layouts import column,row
import ast
import time
import pytz
from datetime import datetime

tz = pytz.timezone('Asia/Calcutta')


UPDATE_INTERVAL = 1000
ROLLOVER = 10 # Number of displayed data points


source = ColumnDataSource({"x": [], "y": []})
consumer = KafkaConsumer('CleanSensorData', auto_offset_reset='earliest',bootstrap_servers=['localhost:9092'], consumer_timeout_ms=1000)
div = Div(
    text='',
    width=120,
    height=35
)


@count()
def update(x):
    for msg in consumer:
        msg_value = msg
        break

    values = ast.literal_eval(msg_value.value.decode("utf-8"))
    timestamp_iso = values["TimeStamp"]["$date"]

    # Convert ISO 8601 timestamp to a datetime object
    timestamp_datetime = datetime.fromisoformat(timestamp_iso.replace('Z', '+00:00'))

    # Convert datetime to POSIX timestamp
    timestamp_sec = timestamp_datetime.timestamp()
```

dashboard.py

```python
def update(x):

    # Convert ISO 8601 timestamp to a datetime object
    timestamp_datetime = datetime.fromisoformat(timestamp_iso.replace('Z', '+00:00'))

    # Convert datetime to POSIX timestamp
    timestamp_sec = timestamp_datetime.timestamp()

    div.text = "TimeStamp: " + str(timestamp_datetime.strftime("%Y-%m-%d %H:%M:%S"))

    y = values['WaterTemperature']
    print(y)

    source.stream({"x": [timestamp_sec], "y": [y]}, ROLLOVER)

p = figure(title="Water Temperature Sensor Data", x_axis_type="datetime", width=1000)

p.line("x", "y", source=source)

p.xaxis.formatter=DatetimeTickFormatter(hourmin = ['%H:%M'])
p.xaxis.axis_label = 'Time'
p.yaxis.axis_label = 'Value'
p.title.align = "right"
p.title.text_color = "orange"
p.title.text_font_size = "25px"

doc = curdoc()
#doc.add_root(p)

doc.add_root(
    row(children=[div,p])
)
doc.add_periodic_callback(update, UPDATE_INTERVAL)
```

This is dashboard.py. It shows how to consume data from Kafka and prepare it for real-time visualization

c. Results :



The provided graph represents the real-time fluctuations in water temperature as captured and processed by the streaming system. The X-axis indicates the time, seemingly in milliseconds, while the Y-axis represents the water temperature values.

Real-Time Visualization: The outcomes of the streaming data analysis are dynamically visualized using a real-time dashboard. This visualization includes trend lines for water temperature over time, highlighted points where anomalies are detected, and alerts that are triggered based on specific threshold conditions.
System Responsiveness: The experiment highlights the system's responsiveness, as changes in the data stream are quickly reflected in the visualizations. For instance, a sudden spike in temperature due to an environmental factor is immediately captured, processed, and displayed, showcasing the potential of the system to support real-time environmental monitoring.
Potential for Immediate Environmental Monitoring: The ability to monitor environmental conditions in real time has significant implications. For industries such as

water quality management, having immediate access to data about changes in water conditions can be crucial for pollution control and ecological conservation. The system's capability to provide continuous and instant analysis makes it an invaluable tool in such contexts.

5. Batch Mode Experiment

a. Description :

The batch mode experiment in this project is designed to process accumulated data stored in MongoDB using Apache Spark, one of the leading platforms for large-scale data processing. This setup allows for periodic processing of large data batches, which are structured similarly to the real-time streams but are processed in intervals (e.g., daily or hourly). By performing the same types of aggregations as those done in the streaming experiment, such as calculating average temperatures or detecting outliers, the batch mode experiment provides a basis for directly comparing performance and outcomes between the two processing modes.

Spark Integration with MongoDB: Apache Spark is configured to read data directly from MongoDB using the MongoDB Connector for Spark. This integration facilitates seamless data transfer into Spark's dataframes, which are then used for further analysis.
Data Aggregation and Analysis: Spark employs sophisticated data processing algorithms to efficiently aggregate and analyze data. The experiment focuses on calculating average temperatures and other statistical metrics from sensor data, leveraging Spark's ability to handle complex computations over large datasets.

b. Data Size :

Handling Large Datasets: The batch processing experiment is specifically designed to manage and process batches containing large records. This scale is chosen to demonstrate Spark's capabilities in handling substantial volumes of data without compromising on processing speed or accuracy.
Data Batching Strategy: The data is grouped into batches based on either time intervals or record count, ensuring consistent and manageable batch sizes for processing. This strategy helps in optimizing the processing workload and aligning it with the system's

capacity.

structure_validate_store.py

```python
import json
from bson import json_util
from dateutil import parser
from pyspark import SparkContext
from kafka import KafkaConsumer, KafkaProducer

#Mongo DB
from pymongo import MongoClient

client = MongoClient('localhost', 27017)
db = client['RealTimeDB']
collection = db['RealTimeCollection']

def timestamp_exist(TimeStamp):
    if collection.count_documents({'TimeStamp': {"$eq": TimeStamp}}) > 0:
        return True
    else:
        return False

def structure_validate_data(msg):
    data_dict = {}

    # Create RDD
    rdd = sc.parallelize(msg.value.decode("utf-8").split())
    data_dict["RawData"] = str(msg.value.decode("utf-8"))

    # Data validation and create json data dict
    try:
        data_dict["TimeStamp"] = parser.isoparse(rdd.collect()[0])
    except Exception as error:
        data_dict["TimeStamp"] = "Error"

    try:
        data_dict["WaterTemperature"] = float(rdd.collect()[1])
        if data_dict["WaterTemperature"] > 99 or data_dict["WaterTemperature"] < -10:
            data_dict["WaterTemperature"] = "Sensor Malfunctions"
    except Exception as error:
        data_dict["WaterTemperature"] = "Error"

    try:
        data_dict["Turbidity"] = float(rdd.collect()[2])
        if data_dict["Turbidity"] > 5000:
            data_dict["Turbidity"] = "Sensor Malfunctions"
    except Exception as error:
```

structure_validate_store.py

```python
def structure_validate_data(msg):
    try:
        if data_dict["Turbidity"] > 5000:
            data_dict["Turbidity"] = "Sensor Malfunctions"
    except Exception as error:
        data_dict["Turbidity"] = "Error"

    try:
        data_dict["BatteryLife"] = float(rdd.collect()[3])
    except Exception as error:
        data_dict["BatteryLife"] = "Error"

    try:
        data_dict["Beach"] = str(rdd.collect()[4])
    except Exception as error:
        data_dict["Beach"] = "Error"

    try:
        data_dict["MeasurementID"] = int(str(rdd.collect()[5]).replace("Beach",""))
    except Exception as error:
        data_dict["MeasurementID"] = "Error"

    return data_dict

sc = SparkContext.getOrCreate()
sc.setLogLevel("WARN")

consumer = KafkaConsumer('RawSensorData', auto_offset_reset='earliest', bootstrap_servers=['localhost:9092'], consumer_timeout_ms=1000)
producer = KafkaProducer(bootstrap_servers=['localhost:9092'])

for msg in consumer:
    if msg.value.decode("utf-8") != "Error in Connection":
        data = structure_validate_data(msg)

        if timestamp_exist(data['TimeStamp']) == False:
            # Push data to MongoDB
            collection.insert_one(data)
            # Push data to ValidatedSensorData Kafka topic
            producer.send("ValidatedSensorData", json.dumps(data, default=json_util.default).encode('utf-8'))
            # Push data to CleanSensorData Kafka topic
            producer.send("CleanSensorData", json.dumps(data, default=json_util.default).encode('utf-8'))

        print(data)
```

This is structure_validate_store.py.

c. Result :

```
sharath@sharath-Dell-G15-5515:~/Desktop/Real-Time-Data-Pipeline-Using-Kafka-and-Spark-master$ python3 batch_mode.py
Average Water Temperature: 15.85
Batch Query Execution Time: 0.0046062 seconds
sharath@sharath-Dell-G15-5515:~/Desktop/Real-Time-Data-Pipeline-Using-Kafka-and-Spark-master$
```

The batch mode processing was performed on a dataset using Apache Spark to calculate aggregated metrics, such as the average water temperature. The results of the execution yielded an average water temperature of 15.85°C. This figure is representative of the environmental conditions within the timeframe of the accumulated data. The calculation was executed in a notably rapid execution time of 0.0040602 seconds, demonstrating the batch processing's capability for high-speed computation on stored datasets.

The quick execution time is particularly noteworthy, suggesting that the Spark-based batch processing system is well-optimized for such aggregative computations. Furthermore, this fast processing capability of batch operations, when applied to large volumes of data that do not require real-time analysis, could offer organizations a time-efficient and cost-effective solution for periodic data analysis tasks.

6. Comparison of Streaming & Batch Modes

```
sharath@sharath-Dell-G15-5515:~/Desktop/Real-Time-Data-Pipeline-Using-Kafka-and-Spark-master$ python3 comparison.py
Streaming Temperature: 16.473300000000005, Streaming Execution Time: 0.13421154022216797 seconds
Batch Temperature: 15.850992662926197, Batch Execution Time: 0.0046274662017822266 seconds
sharath@sharath-Dell-G15-5515:~/Desktop/Real-Time-Data-Pipeline-Using-Kafka-and-Spark-master$
```

Temperature Results Comparison:
The average temperatures obtained from the streaming and batch modes are relatively close, with a minor difference that could be attributed to the datasets' time frame or variations in data sampled during the streaming window versus the entire batch dataset.

Streaming Mode: Produced an average temperature of 16.47°C, reflecting the near-instantaneous analysis of the data as it streams in real time.

Batch Mode: Yielded an average temperature of 15.86°C, indicating the processing of accumulated data over a given period.

Execution Time Comparison:
There is a significant difference in execution times between the two modes, which highlights the trade-offs between real-time processing and batch data analysis.

Streaming Mode: Demonstrates a processing time of approximately 0.13 seconds. This duration is reflective of the near-real-time requirement for streaming data processing, where data is continually ingested and processed in small windows, necessitating quick turnarounds.
Batch Mode: Showcases a faster execution time of 0.005 seconds. This efficiency can be largely attributed to the processing of a predetermined dataset in one go, allowing for more optimized computation and resource allocation.

Discussion:
Timeliness: Streaming processing is designed to provide ongoing insights, which is crucial for time-sensitive decisions. The slightly higher average temperature in streaming could indicate a real-time response to a transient environmental condition.
Resource Efficiency: Batch processing appears to be more resource-efficient in this instance, handling a large dataset with a quicker execution time than streaming. This suggests that for tasks where real-time analysis is not critical, batch processing could be more favorable in terms of both speed and computational load.
Use Case Suitability: The choice between streaming and batch processing should be informed by the specific use case requirements. For operations that rely on the most up-to-date information, such as fraud detection or emergency response systems, streaming offers a distinct advantage. Conversely, for analytical tasks that can be scheduled and do not require immediate action, batch processing is advantageous due to its efficiency and speed.

7. Conclusion:

The project demonstrates the benefits of real-time data processing for quick water quality monitoring, while also showing how batch processing is useful for detailed analysis. This

helps inform decisions on setting up data processing systems for water quality monitoring.

8. References

a. URLs

[1] https://downloads.apache.org/kafka/3.7.0/kafka_2.12-3.7.0.tgz

[2] https://www.apache.org/dyn/closer.lua/spark/spark-3.5.1/spark-3.5.1-bin-hadoop3.tgz

[3] https://www.mongodb.org/static/pgp/server-7.0.asc

[4] https://data.world/cityofchicago/beach-water-quality-automated-sensors

[5] https://data.cityofchicago.org/d/k7hf-8y75

[6] https://data.cityofchicago.org/d/g3ip-u8rb.

[7] https://www.youtube.com/watch?v=udnX21__SuU&ab_channel=LearningJournal

[8] https://www.youtube.com/watch?v=QaoJNXW6SQo&ab_channel=Simplilearn