Project: Weather Stations Monitoring

1. Overview

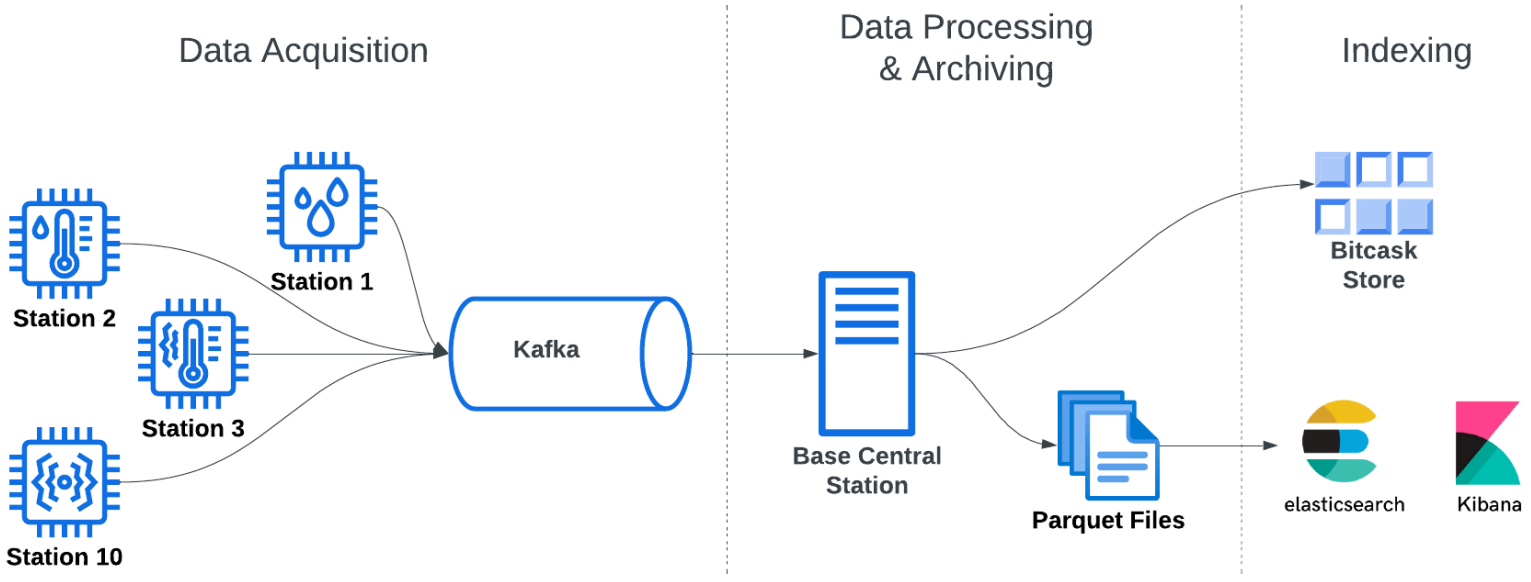
The Internet of Things (IoT) is an important source of data streams in the modern digital world. The “Things” are huge in count and emit messages in very high frequency which flood the global internet. Hence, efficient stream processing is inevitable.

One use case is the distributed weather stations use case. Each **“weather station”** emits readings for the current weather status to the **“central base station”** for persistence and analysis. In this project, you will be required to implement the architecture of a weather monitoring system.

2. System Architecture

The system is composed of three stages:

- **Data Acquisition:** multiple weather stations that feed a queueing service (Kafka) with their readings.
- **Data Processing & Archiving:** The base central station is consuming the streamed data and archiving all data in the form of Parquet files.
- **Indexing:** two variants of index are maintained
  - Key-value store (Bitcask) for the latest reading from each individual station
  - ElasticSearch / Kibana that are running over the Parquet files



3. Installation

For the purpose of this lab, we will use a kubernetes application containing a cluster of:

- 10 weather stations
- 1 Kafka service + 1 Zookeeper service (for Kafka)
- 1 Elastic + Kibana service
- 1 central base station service

. To set up this cluster, you need first to:

- Install Docker
- Pull Bitnami Kafka Docker Image (Includes zookeeper)
- Pull image containing ElasticSearch and Kibana (or feel free to use separate images)

4. Implementation

A) Write Weather Station Mock

Each weather station should **output a status message every 1 second** to report its sampled weather status. Check the Weather Status Message below.
To simulate data with query-able nature, you will have to:

- Randomly, change the **battery_status** field by the following specs:
  - Low = 30% of messages per service
  - Medium = 40% of messages per service
  - High = 30% of messages per service
- Randomly drop messages on a 10% rate

Weather Status Message

The weather status message should be of the following schema:

```
{
   "station_id": 1, // Long
   "s_no": 1, // Long auto-incremental with each message per service
   "battery_status": "low", // String of (low, medium, high)
   "status_timestamp": 1681521224, // Long Unix timestamp
   "weather": {
       "humidity": 35, // Integer percentage
       "temperature": 100, // Integer in fahrenheit
       "wind_speed": 13, // Integer km/h
   }
}
```

B) Set up Weather Station to connect to Kafka

Use the produce API to send messages to the Kafka server. You should use the java programmatic API for this task.
To add the kafka clients dependencies, use the needed line from this link.

Try getting a simple example working first before progressing. A sample produce API is as follows:

```
Properties properties = new Properties();
properties.setProperty(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG,
"127.0.0.1:9092");
properties.setProperty(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
StringSerializer.class.getName());
properties.setProperty(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
StringSerializer.class.getName());
try (KafkaProducer<String,String> producer = new KafkaProducer<>(properties)) {
   ProducerRecord<String, String> record = new ProducerRecord<>("my_first",
"Hey Kafka!");
   producer.send(record);
}
```

You can use this to confirm this API works on your machine.

```
./bin/kafka-console-consumer.sh --bootstrap-server 127.0.0.1:9092 --topic
my_first --from-beginning
```

Output should display the message above.

C) Implement Raining Triggers in Kafka Processors

You are required to use Kafka Processors to detect when it's raining. You should use Kafka Processor to detect if humidity is **higher than 70%**. The processor should output a special message to a specific topic in Kafka.

Note : You can also use Kafka DSL to implement such features.

D) Implement Central Station

Implement BitCask Riak to store updated view of weather status

You should maintain a key value store of the station's statuses. To do this efficiently, you are going to implement the BitCask Riak to maintain an updated store of each station status.

The implementation should be as discussed in lectures with some notes:

- You are required to implement hint files to help in rehash for recovery (will be tested)
- Compaction should be scheduled to run over the segment files to avoid disrupting active readers
- You are NOT required to implement checksums to detect errors
- You are NOT required to implement tombstones for deletions as there is no point in deleting some weather station ID entry for sport.

You should also implement a **BitCask Client** to be able to view its content. The client should call endpoints of your choice and design in the Central Station to achieve what's required. The client should have the following requirements:

1. Should be runnable through a bash script (or equivalent in Python if you prefer) as follows:
    a. `./bitcask_client.sh --view-all` This should print all keys with their latest values to a file named with the timestamp of the current time with extension **.csv**. The file should be a CSV file with 2 columns key and value. Example of file name is "1746034451.csv".
    b. `./bitcask_client.sh --view --key=SOME_KEY` This should print to stdout the value of the associated key.
    c. `./bitcask_client.sh --perf --clients=100` This should start 100 threads each trying to query all keys stored in bitcask and output file similar to requirement no.1 but append thread number of the timestamped CSV file name. Example file name is "1746034451_thread_1.csv".

This client will be used for:

1. Testing the system during development to make sure your implementation of the BitCask is correct.
2. Testing during discussions by TAs.

Implement Archiving of all Weather Statuses

Aside from maintaining an updated LSM for updated statuses, you should archive all weather statuses history for all stations.

We can append all weather statuses into parquet files. For this purpose, your central server should write all received statuses to parquet files and **partition them by time and station ID**.

You should write records in batches to the parquet to avoid blocking on IO frequently. Common batch size could be **10K records**.

E) Set up Historical Weather Statuses Analysis

You should direct all weather statuses to ElasticSearch for indexing and querying by Kibana. The use cases for such analysis are:

- Count of low-battery statuses per station (should confirm percentages above)
- Count of dropped messages per station (should confirm percentages above)

In order to do that, connect parquet files as a data source for ElasticSearch so you can index using ElasticSearch and visualize using Kibana.

F) Deploy using Kubernetes

You are required to deploy all of this using Docker. In order to do that, you need to do few steps:

- Write Dockerfile for the central server
- Write Dockerfile for the weather stations
- Write K8s yaml file to incorporate:
    - 10 services with image for the weather station
    - Service with image for the central server you wrote.
    - 2 services with kafka & zookeeper images
    - Shared storage for writing parquet files and BitCask Riak LSM

G) Profile Central Station using JFR

Since the Central Station is the heart of this Data Intensive Application, it is of the utmost importance that we make sure that our code is free of any unnecessary overheads. One way to do this is to use a profiling tool to measure execution time, memory consumption, GC pauses and their durations, etc..

Java Flight Recorder (JFR) is a tool for collecting diagnostic and profiling data about a running Java application. It is integrated into the Java Virtual Machine (JVM) and causes almost no performance overhead, so it can be used even in heavily loaded production environments.

You are required to profile your central station using JFR (Java Flight Recorder). You should run the system for 1 minute and report the following:

- Top 10 Classes with highest total memory
- GC pauses count
- GC maximum pause duration
- List of I/O operations

5. Deliverables

You are required to deliver the following:

- Source Code
- Docker & Kubernetes files:
  - K8s yaml file
  - Weather Station Dockerfile
  - Central Server Dockerfile
- Screenshots for Kibana visualization confirming:
  - Battery status distribution (30% low - 40% medium - 30% high)
  - 10% dropped messages
- Sample Parquet File
- Sample BitCask Riak LSM directory
- Report containing all of the above

6. Bonus

- Open-Meteo (https://open-meteo.com/) is an open-source weather API with free access for non-commercial use. In addition to collecting the data from your weather stations, it is required that you integrate with it and collect data from it and feed it to your queuing service (Kafka)
  Hint: you will need to implement a Channel Adapter
- We studied in the Enterprise Integration Patterns multiple common patterns that perfectly fits in this project, it is required that you use 5 to 6 patterns in your implementation
  Hint: you may use some of these patterns (or more): dead-letter channel, claim check, invalid-message channel, polling consumer, idempotent receiver, envelope wrapper.

7. Notes

- You should work in groups of **four**

- All team members should be ready to answer any question during discussion
- Any cheating will be severely penalized.