Weather Stations Monitoring

1. Snippets of The Source Code

1.1. Weather Stations (Producer)

```
public class WeatherStationProducer {
    public int produce(){
        long s_no = 0;
        // Create a Kafka producer
        KafkaProducer<String, String> producer = new KafkaProducer<>(properties);
        GetData getData = new GetData(this.latitude,this.longitude);
        Weather weather = getData.getData();
        JSONArray temperature = weather.getTemperature();
        JSONArray humidity = weather.getHumidity();
        JSONArray windSpeed = weather.getWindSpeed();
        WeatherStatusMessage message = new WeatherStatusMessage(this.stationId);
        int count = 0;
        long currentUnixTimestamp = (System.currentTimeMillis() / 1000L)-1;
        while (true) {
            currentUnixTimestamp++;
            s_no++;
            if(isDrop()){
                if((s_no % 24) == 1) {
                    weather = getData.getData();
                    temperature = weather.getTemperature();
                    humidity = weather.getHumidity();
                    windSpeed = weather.getWindSpeed();
                    count = 0;
                }
                continue;
            }
            message.generateWeatherStatusMessage(s_no,currentUnixTimestamp, (Double) temperature.get(count), (Integer) humidity.get(count),
            String value = message.toString();
            count++;
            ProducerRecord<String, String> record = new ProducerRecord<>("weather-status-messages",value);
            producer.send(record);
            System.out.println("Sent message: " + value);
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            if((s_no % 24) == 1) {
                weather = getData.getData();
                temperature = weather.getTemperature();
                humidity = weather.getHumidity();
                windSpeed = weather.getWindSpeed();
```

```java
public class GetData {
    private String url = "";
    public GetData(String latitude,String longitude){
        this.url = "https://api.open-meteo.com/v1/forecast?latitude=" +
                latitude + "&longitude=" + longitude + "&hourly=relativehumidity_2m,windspeed_80m,temperature_80m&" +
                "current_weather=true&temperature_unit=fahrenheit&timeformat=unixtime&forecast_days=1&timezone=Africa%2FCairo";

    }
    private String fetchDataFromOpenMeteo(){
        URL url;
        HttpURLConnection conn;
        Scanner scanner;
        String responseBody = "";
        try {
            url = new URL(this.url);
            conn = (HttpURLConnection) url.openConnection();
            conn.setRequestMethod("GET");
            conn.setRequestProperty("Accept", "application/json");

            scanner = new Scanner(conn.getInputStream());
            while (scanner.hasNext()) {
                responseBody += scanner.nextLine();
            }
            scanner.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
        return responseBody.toString();
    }
    public  Weather getData() {
        String responseBody = this.fetchDataFromOpenMeteo();
        // Parse the JSON response from the API.
        JSONObject jsonObject = new JSONObject(responseBody.toString());
        JSONObject currentWeatherDaily = jsonObject.getJSONObject("hourly"); // get all data through all day hourly;
        JSONArray relativehumidity_2m = currentWeatherDaily.getJSONArray("relativehumidity_2m");
        JSONArray windspeed_80m = currentWeatherDaily.getJSONArray("windspeed_80m");
        JSONArray temperature_2m = currentWeatherDaily.getJSONArray("temperature_80m");

        Weather weather = new Weather(temperature_2m, relativehumidity_2m,windspeed_80m);
        return weather;
    }
```

1.2. Kafka

```
import java.util.Map;
import java.util.Properties;
import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class RainingTriggerProcessor {
    private static final Pattern messagePattern = Pattern.compile("\\{station_id=(\\d+), s_no=(\\d+), battery_status='(\\w+)', status_timestamp=(\\d+), " +
            "weather=\\{humidity=(\\d+(?:\\.\\d+)?), temperature=(\\d+(?:\\.\\d+)?), wind_speed=(\\d+(?:\\.\\d+)?)}}");

    private Properties config;
    public RainingTriggerProcessor(){
        this.config = new Properties();
        config.put(StreamsConfig.APPLICATION_ID_CONFIG, "raining-trigger-processor");
        Map<String, String> env = System.getenv();
        String kafkaBroker = env.get("KAFKA_BROKER");
        System.out.println(kafkaBroker);
        config.put("bootstrap.servers", kafkaBroker);
        config.put(StreamsConfig.DEFAULT_KEY_SERDE_CLASS_CONFIG, Serdes.String().getClass());
        config.put(StreamsConfig.DEFAULT_VALUE_SERDE_CLASS_CONFIG, Serdes.String().getClass());
    }
```

```
public class RainingTriggerProcessor {
    public void run(){
        KStream<String, String> weatherStream = builder.stream("weather-status-messages", Consumed.with(Serdes.String(), Serdes.String()));

        KStream<String, String> rainEvents = weatherStream
                .filter(
                        (key, value) -> {
                            Matcher matcher = messagePattern.matcher(value);
                            if (messagePattern.matcher(value).matches() && matcher.find()) {
                                return Integer.parseInt(matcher.group(5)) >= 70;
                            } else {
                                return false;
                            }
                        })
                .mapValues(
                        value -> {
                            Matcher matcher = messagePattern.matcher(value);
                            if (matcher.find()) {
                                return "RainingMessage{ station_id=" +
                                        matcher.group(1) +
                                        ", " +
                                        "s_no=" +
                                        matcher.group(2) +
                                        ", " +
                                        "battery_status=" +
                                        matcher.group(3) +
                                        ", " +
                                        "status_timestamp=" +
                                        matcher.group(4) +
                                        ", " +
                                        "weather='raining' }";
                            } else {
                                return "";
                            }

                        }
                )
                .peek(
                        (key, value) -> System.out.println(value)
                );

        rainEvents.to("raining-status-messages", Produced.with(Serdes.String(), Serdes.String()));
```

1.3. Base Central Station (Consumer)

```
public String get(String key) throws IOException {
    if (!this.keyToEntryMetaData.containsKey(key))
        return "NOT FOUND";
    EntryMetaData entryMetaData = keyToEntryMetaData.get(key);
    byte[] bytes = DiskReader.readEntryValueFromDisk(entryMetaData.getFileID(), entryMetaData.getValuePosition(),
entryMetaData.getValueSize());
    return BitCaskEntry.parseBytesToValue(bytes);
}
public void put(String key, String value) throws IOException {
    BitCaskEntry bitCaskEntry = new BitCaskEntry(key.getBytes().length,
            System.currentTimeMillis(), key, value);
    DiskResponse diskResponse = diskWriter.writeEntryToActiveFile(bitCaskEntry);
    EntryMetaData entryMetaData = new EntryMetaData(bitCaskEntry.getValueSize(), diskResponse.getValuePosition(),
            bitCaskEntry.getTimestamp(), diskResponse.getFileName());
    keyToEntryMetaData.put(key, entryMetaData);
}
```

```
public Command parseCommand(String input) throws InvalidCommandException {
    int cursor = 0;
    cursor = cleanWhiteSpace(input, cursor);
    if (cursor + 2 >= input.length()) {
        throw new InvalidCommandException("Operation needs to be specified in the command");
    }
    String operation = input.substring(cursor, cursor + 3).toUpperCase();
    cursor += 3;
    cursor = cleanWhiteSpace(input, cursor);
    if (cursor == input.length()) {
        throw new InvalidCommandException("Operands needs to be specified in the command");
    }
    return switch (operation) {
        case "GET" -> parseGetCommand(input, cursor);
        case "SET" -> parseSetCommand(input, cursor);
        default -> throw new InvalidCommandException("Operation not supported");
    };
}
```

```
public static byte[] readEntryValueFromDisk(String fileID, long valuePosition, int valueSize) throws IOException {
    byte[] value = new byte[valueSize];
    try (RandomAccessFile randomAccessFile = new RandomAccessFile(new File(dbDirectory + fileID), "r")) {
        randomAccessFile.seek(valuePosition);
        randomAccessFile.read(value, 0, valueSize);
    }
    return value;
}
```

```java
public static void readHintFile(File hintFile, Map<String, EntryMetaData> keyToEntryMetaData) throws IOException {
    BufferedInputStream bufferedInputStream = new BufferedInputStream(new FileInputStream(hintFile));
    byte[] bytes = bufferedInputStream.readAllBytes();
    int byteCursor = 0;
    while (byteCursor < bytes.length) {
        int entrySize = Ints.fromByteArray(Arrays.copyOfRange(bytes, byteCursor, byteCursor + 4));
        if (byteCursor + 4 + entrySize > bytes.length)
            break;
        byte[] entryBytes = Arrays.copyOfRange(bytes, byteCursor + 4, byteCursor + 4 + entrySize);
        int keySize = Ints.fromByteArray(Arrays.copyOfRange(bytes, byteCursor + 12, byteCursor + 16));
        String key = new String(Arrays.copyOfRange(bytes, byteCursor + 28, byteCursor + 28 + keySize),
                StandardCharsets.UTF_8);
        byteCursor += 4 + entrySize;
        EntryMetaData entryMetaData = EntryMetaData.buildEntryFromBytes(entryBytes,
hintFile.getName().substring(5));
        if (!keyToEntryMetaData.containsKey(key)) {
            keyToEntryMetaData.put(key, entryMetaData);
        }
    }
}
```

```java
private long writeToTheDisk(BitCaskEntry bitCaskEntry, File file, FileOutputStream fileOutputStream,
FileOutputStream fileOutputStreamReplica) throws IOException {
    BufferedOutputStream bufferedOutputStream = new BufferedOutputStream(fileOutputStream);
    BufferedOutputStream bufferedOutputStreamReplica = new BufferedOutputStream(fileOutputStreamReplica);
    byte[] bytesToWrite = bitCaskEntry.toBytes();
    bufferedOutputStream.write(Ints.toByteArray(bytesToWrite.length));
    bufferedOutputStreamReplica.write(Ints.toByteArray(bytesToWrite.length));
    bufferedOutputStream.flush();
    bufferedOutputStreamReplica.flush();
    long valuePosition = file.length() + 8 + 4 + bitCaskEntry.getKeySize() + 4;
    writeEntryToHintFile(file.getName(), new EntryMetaData(bitCaskEntry.getValueSize(), valuePosition,
            bitCaskEntry.getTimestamp(), file.getName()), bitCaskEntry.getKey());
    bufferedOutputStream.write(bytesToWrite);
    bufferedOutputStreamReplica.write(bytesToWrite);
    bufferedOutputStream.flush();
    bufferedOutputStreamReplica.flush();
    return valuePosition;
}
```

```java
public DiskResponse writeCompacted(BitCaskEntry bitCaskEntry, File file) throws IOException {
    File fileReplica = new File(directoryPath + "replica_" + file.getName());
    FileOutputStream compactFileOutputStream = new FileOutputStream(file, true);
    FileOutputStream compactFileOutputStreamReplica = new FileOutputStream(fileReplica, true);
    long valuePosition = writeToTheDisk(bitCaskEntry, file, compactFileOutputStream,
compactFileOutputStreamReplica);
    return new DiskResponse(file.getName(), valuePosition);
}
public DiskResponse writeEntryToActiveFile(BitCaskEntry bitCaskEntry) throws IOException {
    checkIfFileExceededSize();
    return writeEntry(bitCaskEntry);
}
```

```java
ScheduledExecutorService scheduledExecutorService = Executors.newScheduledThreadPool(1);
scheduledExecutorService.scheduleAtFixedRate(() -> {
    try {
        CommandFactory commandFactory = new CommandFactory();
        bitCask.mergeAndCompaction();
        for (int i = 1; i <= 10; i++) {
            Command command = commandFactory.parseCommand("get " + i);
            command.execute(bitCask);
        }
    } catch (IOException | InvalidCommandException e) {
        throw new RuntimeException(e);
    }
}, 5, 60, TimeUnit.SECONDS);
try (ServerSocket server = new ServerSocket(port)) {
    CommandFactory commandFactory = new CommandFactory();
    Socket client = server.accept();
    System.out.println("connected");
    String input;
    out = new PrintWriter(client.getOutputStream(), true);
    in = new BufferedReader(new InputStreamReader(client.getInputStream()));
    while ((input = in.readLine()) != null) {
        Command command = commandFactory.parseCommand(input);
        command.execute(bitCask);
        out.println(bitCask.get(input.split(" ")[1]));
    }
} catch (Exception e) {
    throw new RuntimeException(e);
}
```

```protobuf
syntax = "proto3";

option java_multiple_files = false;
option java_package = "bitCask.proto";
option java_outer_classname = "WeatherStatusProto";
option objc_class_prefix = "WSP";

package proto;

message WeatherStatus {
  int64 station_id = 1;
  int64 s_no = 2;
  string battery_status = 3;
  int64 status_timestamp = 4;
  Weather weather = 5;
}

message Weather {
  float humidity = 1;
  float temperature = 2;
  float wind_speed = 3;
}
```

2. Integrations

First, we use open meteo website to get real data for weather station and to do this we apply **Channel Adapter** to link our producers with website and change data to be suitable for system where we create class responsible for get data from open meteo and pass it to every producer , producer responsible only to determine position of stations in world.

Second, after each producer has its own data he sets up a connection with kafka according to a special topic and begins to send messages to the kafka broker through this topic.

Third, we create a centralBase Station which is responsible to consume all messages of all weather stations through kafka broker and then bitcask does its function which will talk about it clearly below. Consumers pass message of kafka to class which responsible to write this message to parquet file according to each weather station ID.

⇒ We already apply **Polling onsumer** pattern.

⇒ we apply **Invalid Channel** pattern , which If a message has any null value in this content , it will pass this message to RockDB on disk and not make the central station consume it.

⇒ We apply an **Aggregator pattern** , which collects every 10K message from each station and writes it in a parquet file separate from another parquet file of another station.

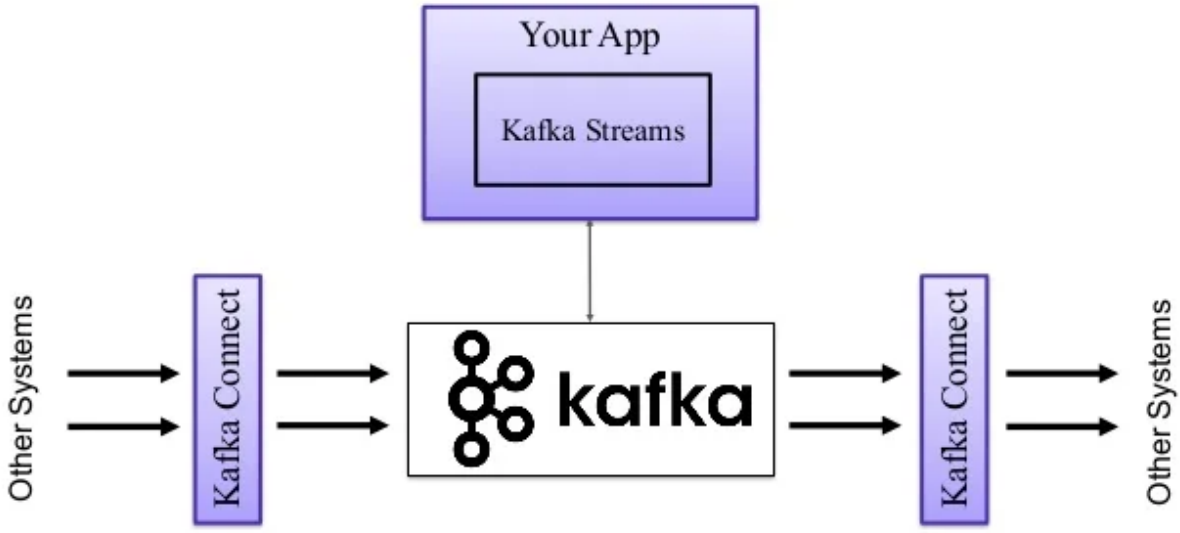
⇒ We apply an **Envelope Wrapper** and **Pipe and Filter patterns**, in kafka processor which unwraps each weather status message in the stream, parses it, filters on specific criteria (high humidity > 70) and then wraps it again as raining status message.

———————————————————————————

3. Kafka Processors

StreamsBuilder is used to define the processing topology. It creates a Kafka stream from the "weather-status-messages" topic, applies a filter to only keep messages with a precipitation probability greater than or equal to 70%, and then transforms each message into a new message type called "RainingMessage".

The transformed messages are logged using the peek() method, and then written to a new Kafka topic called "raining-status-messages". Finally, a KafkaStreams object is instantiated using the StreamsBuilder and configuration objects



4. Docker & K8s

The deployment process involves setting up Docker containers for weather stations, deploying them on Kubernetes (K8s), and configuring Kafka and ZooKeeper for data streaming and elastic search & kibana for analytics. Additionally, a Docker image for the base-central station, which acts as the consumer for Kafka, will be created. Also we made a persistent Volume to save the parquet files and the Bitcask files.

4.1. Dockerizing the Weather Stations:

- Create a Dockerfile for the weather station application. This file will define the necessary dependencies and configuration.
- Build the Docker image using the Dockerfile.
- Push the Docker image to Docker Hub or a private Docker registry for easy accessibility during deployment.

4.2. Deploying Weather Stations on Kubernetes (K8s):

- Install and configure Kubernetes on your desired environment.
- Create a Kubernetes deployment configuration file, specifying the desired number of weather station replicas and other relevant details.
- Apply the deployment configuration to deploy the weather station containers on the Kubernetes cluster.

4.3. Setting up Kafka and ZooKeeper:

- Obtain the necessary Kubernetes deployment files for Kafka and ZooKeeper. One option is to use the Bitnami Helm charts for Kafka and ZooKeeper.
- Modify the deployment files to suit your requirements, such as storage, resource allocation, and network configuration.

- Deploy ZooKeeper and Kafka using the modified deployment files, ensuring that the Kafka brokers are reachable by the weather stations.

4.4. Dockerizing the Base-Central Station:

- Create a Dockerfile for the base-central station, which acts as the consumer for the Kafka topics produced by the weather stations.
- Include the necessary dependencies and configuration in the Dockerfile.
- Build the Docker image for the base-central station.

4.5. Deploying the Base-Central Station:

- Modify the Kubernetes deployment configuration for the base-central station, specifying the appropriate Kafka topic consumer settings.
- Apply the deployment configuration to deploy the base-central station container on the Kubernetes cluster.

4.6. Creating Persistent Volume and Persistent Volume Claim

To ensure data persistence and availability for the Parquet files and Bitcask used in the weather monitoring system, we created a Persistent Volume (PV) and a Persistent Volume Claim (PVC) in Kubernetes. This will enable the storage of data even if the containers or pods are restarted or rescheduled.

1-Define Persistent Volume:

- Determine the storage requirements for the Parquet files and Bitcask, such as the size and access mode.
- Create a PV configuration file (e.g., pv.yaml) with the necessary specifications.
- Modify the metadata.name, spec.capacity.storage, and spec.hostPath.path fields to match your desired PV name, storage size, and the path to the actual storage location.

2-Apply the Persistent Volume:

- Apply the PV configuration using the following command

3- Define Persistent Volume Claim:

- Create a PVC configuration file (e.g., pvc.yaml) to define the requirements for the Parquet files and Bitcask storage.
- Modify the metadata.name and resources.requests.storage fields as needed to match your desired PVC name and storage size.

4- Update Deployment Configuration:

- In the deployment configuration file for the weather stations and the base-central station, add a volume and volume mount specification to the containers' configuration.
- Mount the Persistent Volume Claim (PVC) to the appropriate directory within the container where the Parquet files and Bitcask data will be stored.

4.7. Deploying Elasticsearch and Kibana

To enhance the functionality of the weather monitoring system, Elasticsearch and Kibana can be deployed to enable efficient data indexing, storage, and visualization.

4.8. Conclusion

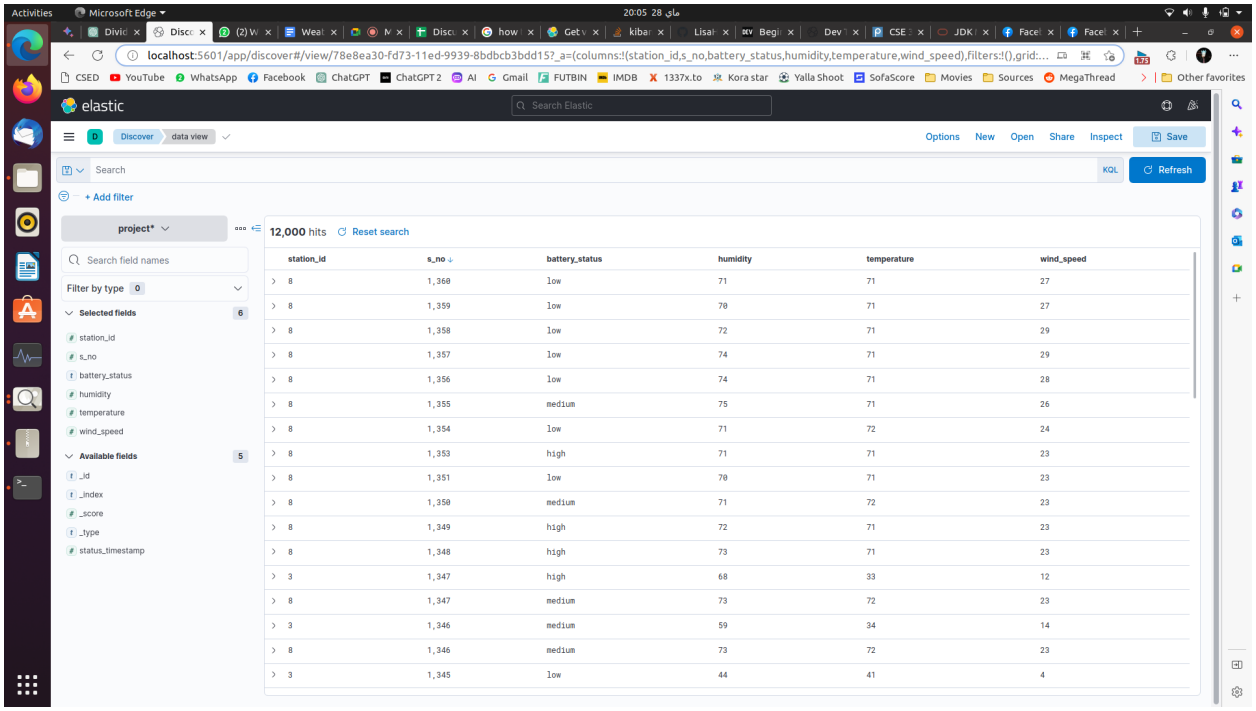
```
deffo@deffo:~/Downloads/WSM/deployments/centralStation$ kubectl get service
NAME                              TYPE           CLUSTER-IP       EXTERNAL-IP   PORT(S)
basestation-service               ClusterIP      10.101.192.203   <none>        3636/TCP
bitcask-service                   ClusterIP      10.98.14.172     <none>        4240/TCP
hello-kube-load-balancer-service  LoadBalancer   10.99.6.252      <pending>     80:32388/TCP
kubernetes                        ClusterIP      10.96.0.1        <none>        443/TCP
my-kafka                          ClusterIP      10.101.132.105   <none>        9092/TCP
my-kafka-headless                 ClusterIP      None             <none>        9092/TCP,9094/TCP,9093/TCP
postgres-cluster-ip-service       ClusterIP      10.107.133.183   <none>        5432/TCP
processor-service                 ClusterIP      10.104.109.221   <none>        3000/TCP
```

```
bitcask-69d44f478-2w8f9                  1/1     Running            1 (10m ago)      4h42m
bitcask-69d44f478-zzjrs                  1/1     Running            1 (8m48s ago)    4h42m
hello-kube-pod                           1/1     Running            2 (8m48s ago)    11d
my-kafka-0                               1/1     Running            1 (8m48s ago)    6h43m
postgres-deployment-6cc6896b69-ks55b     1/1     Running            2 (8m48s ago)    11d
processor-deployment-5d8f8dc999-hnpnj    1/1     Running            2 (8m13s ago)    5h1m
processor-deployment-5d8f8dc999-vqsjt    1/1     Running            2 (8m13s ago)    5h1m
station1-deployment-5b85b8b85b-lbk4m     1/1     Running            8 (8m13s ago)    5h31m
station10-deployment-545bfd6684-cbpml    1/1     Running            4 (8m8s ago)     5h30m
station2-deployment-757887f6db-26bb4     1/1     Running            3 (8m1s ago)     5h31m
station3-deployment-5687f8ff49-cjrq9     1/1     Running            2 (10m ago)      5h31m
station4-deployment-7f8648c584-9pr5c     1/1     Running            5 (8m2s ago)     5h31m
station5-deployment-598d6667b8-qfcxq     1/1     Running            3 (8m6s ago)     5h31m
station6-deployment-5b8b558669-8rkvg     1/1     Running            3 (8m12s ago)    5h31m
station7-deployment-6564847f65-2ldc4     1/1     Running            3 (8m9s ago)     5h30m
station8-deployment-b76f97957-xj7s2      1/1     Running            3 (8m5s ago)     5h30m
station9-deployment-59b66cb849-rjb5k     1/1     Running            3 (8m13s ago)    5h30m
```

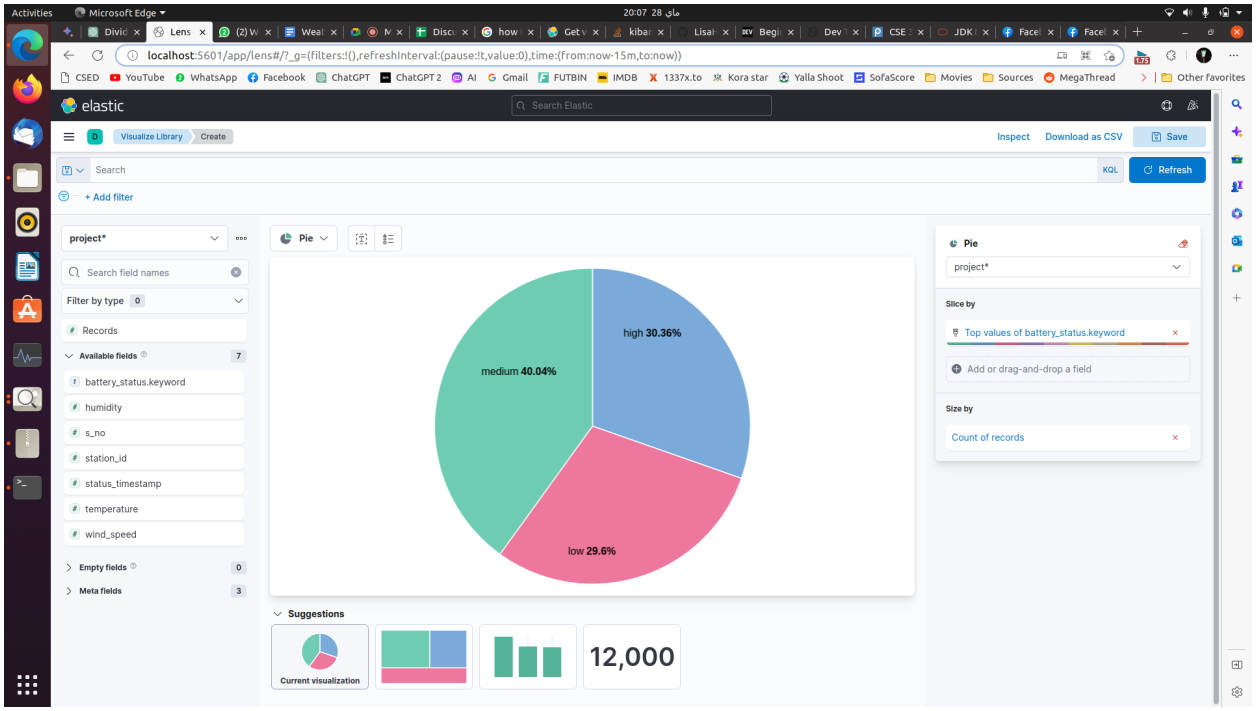
We have successfully deployed the weather monitoring system. The weather stations are running as Docker containers on Kubernetes, and Kafka and ZooKeeper are set up to enable data streaming. The base-central station, acting as the consumer, is also deployed and ready to process the data from the weather stations.

5. Elastic Search & Kibana

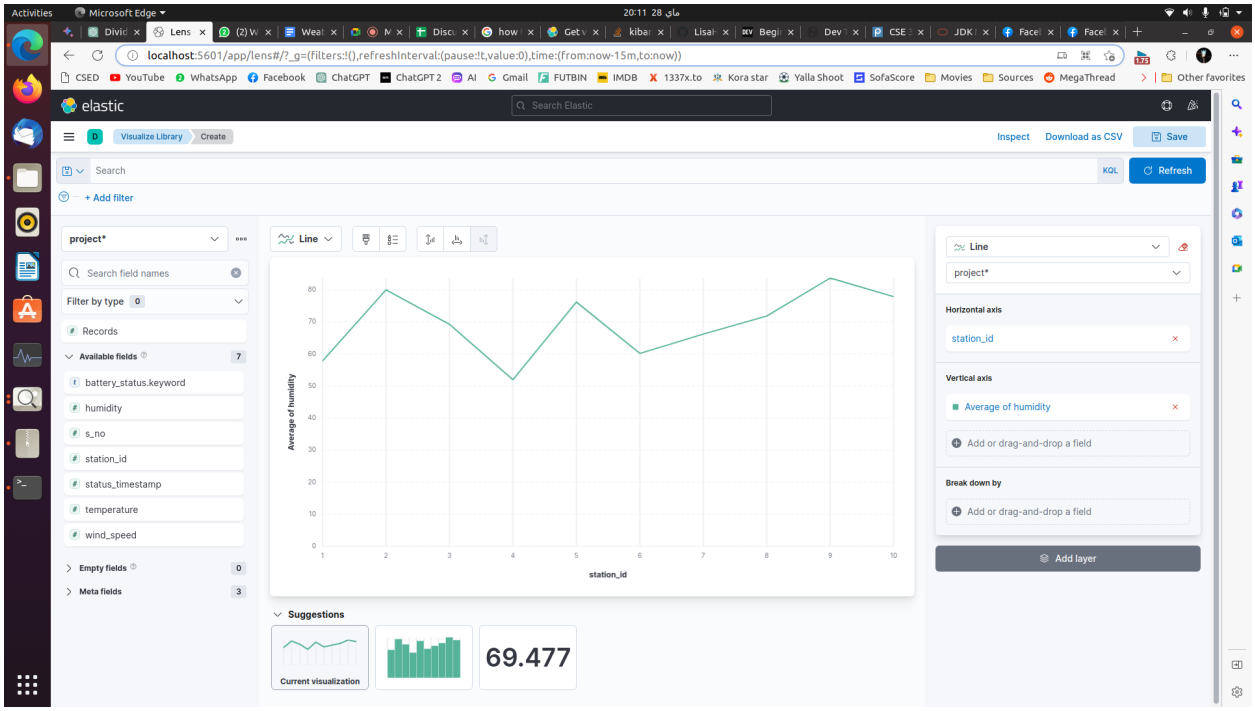
- Data Visualization through kibana



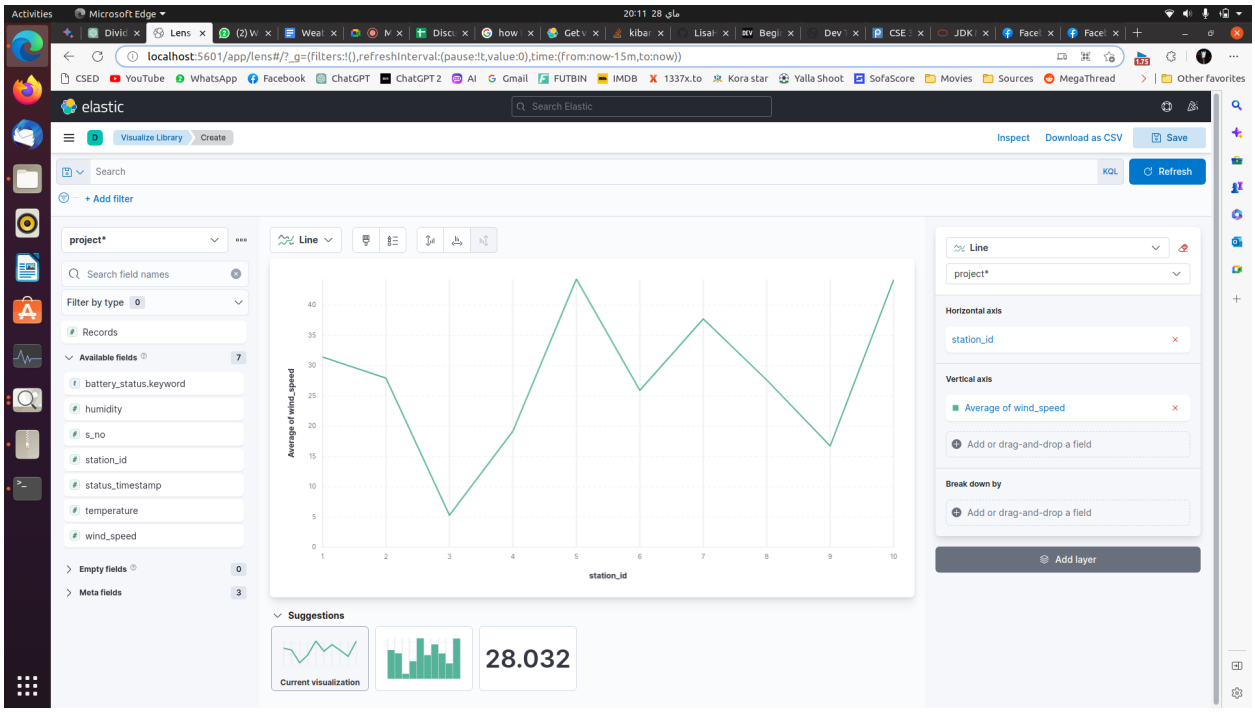
- Ratio of battery statues



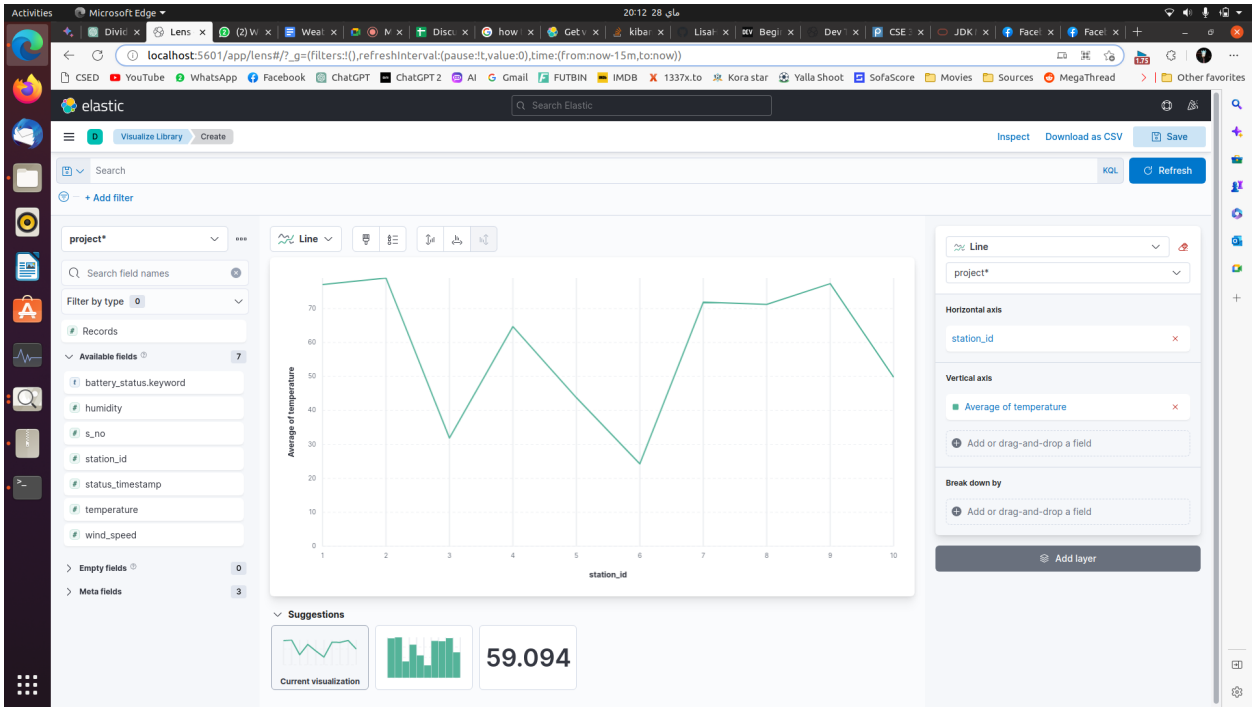
- Average humidity of each station



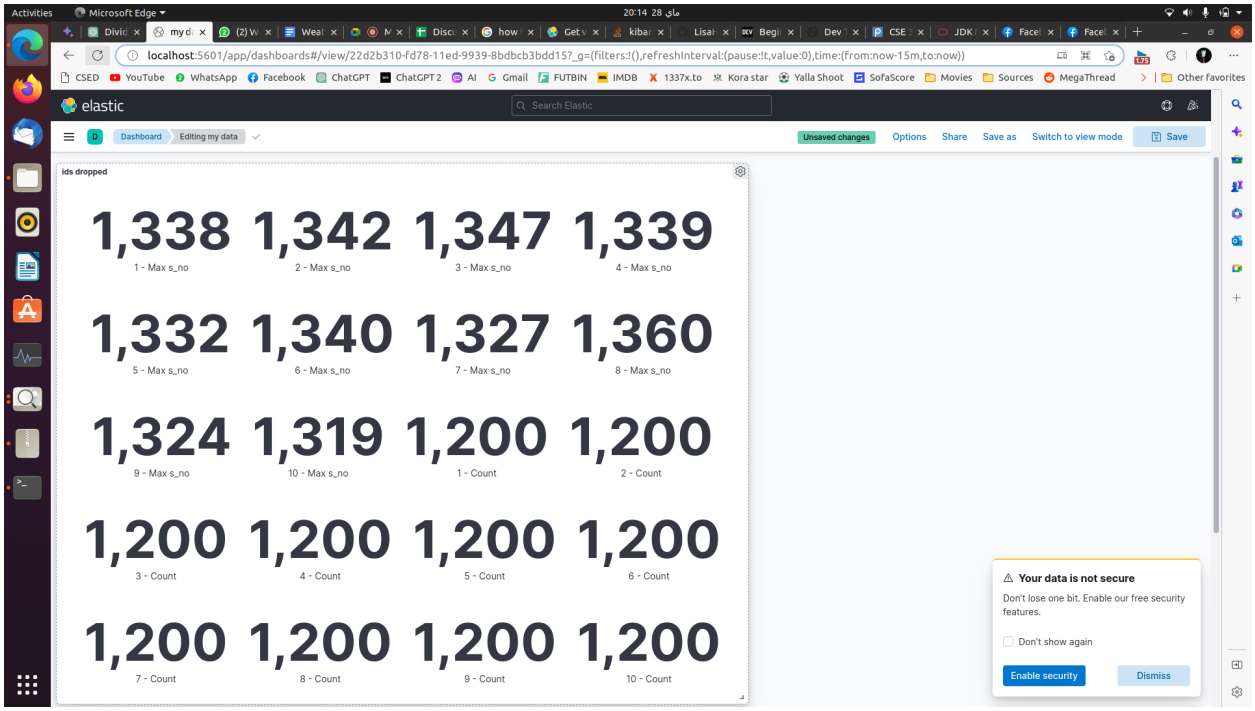
- Average wind speed of each station



- Average temperature of each station



- Max id and unique ids of every station



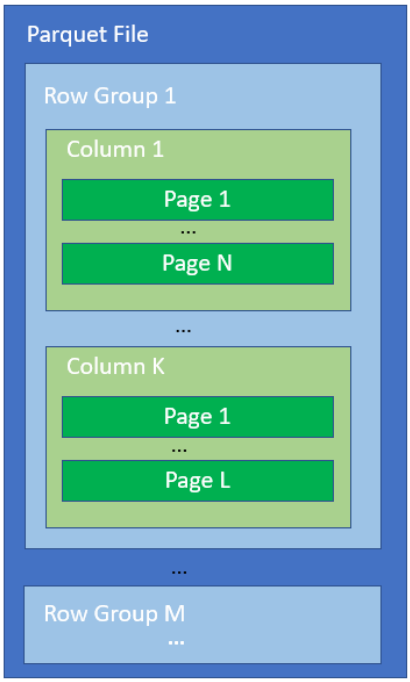
6. Parquet Files

WriteParquetFile that handles the writing of weather status data to Parquet files. Here is a summary of its functionality:

The class has several private variables, including the Parquet files directory path, root directory path, Hadoop file system object, Hadoop configuration object, Parquet schema, mappings for Parquet file size and version, and a storage for Parquet writers.

- The constructor initializes the Hadoop file system, creates the root directory if it doesn't exist, and initializes the mappings for Parquet file size and version.
- The write() method takes a WeatherStatusMessage object and an operating date as input and writes the weather status data to the appropriate Parquet file.
- It constructs the file path based on the date, station ID, version number, and operating date.
- It retrieves the Parquet writer from the storage associated with the file path. If it doesn't exist, a new writer is created using the createParquetWriter() method and added to the storage.
- The weather status data is written to the Parquet file using the writer, and the Parquet file size for the corresponding station ID is incremented.
- If the Parquet file size reaches a threshold (50 in this case), the writer is closed, the file is renamed with the ".parquet" extension, the size and version number for the station ID are reset, and the writer is removed from the storage.
- The code also includes a renameFile() method to rename a file by adding the ".parquet" extension and a createParquetWriter() method to create a Parquet writer with the specified options.

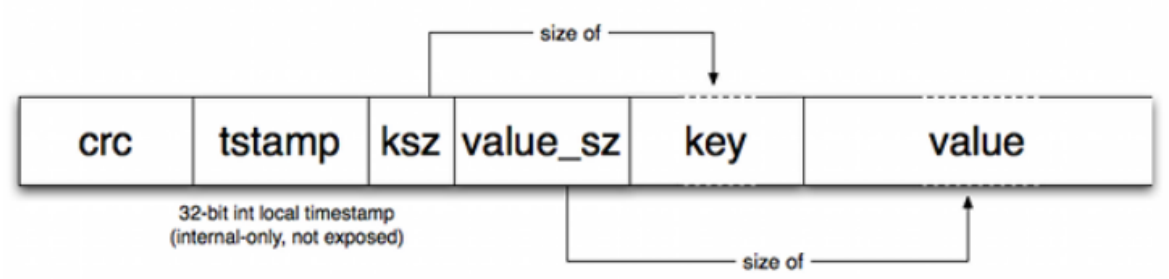
In summary, the WriteParquetFile class provides a mechanism to write weather status data to Parquet files, managing file paths, writers, sizes, and versions for efficient storage and retrieval of the data.



7. BitCask

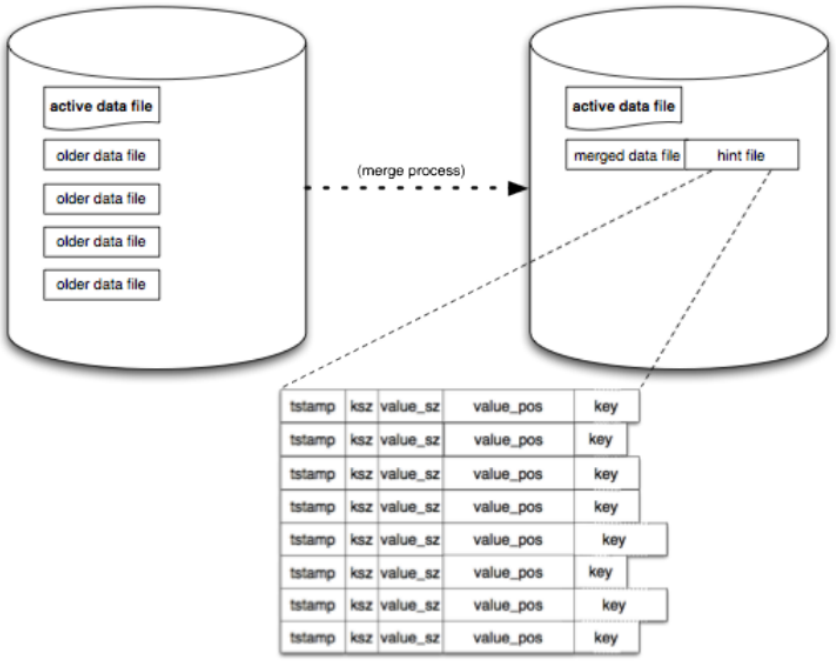
7.1. Implementation Details & Assumptions

- BitCask opens on 4240 port to receive requests from the base central station.
    - It receives a message like a command “Set 1 {humidity=34.3, etc}”
    - Parses the command and executes it.
    - Also there is a command for get.
- BitCask writes the record but firstly it writes the size of the record in 4 bytes then it writes the record like in the following figure (without CRC).

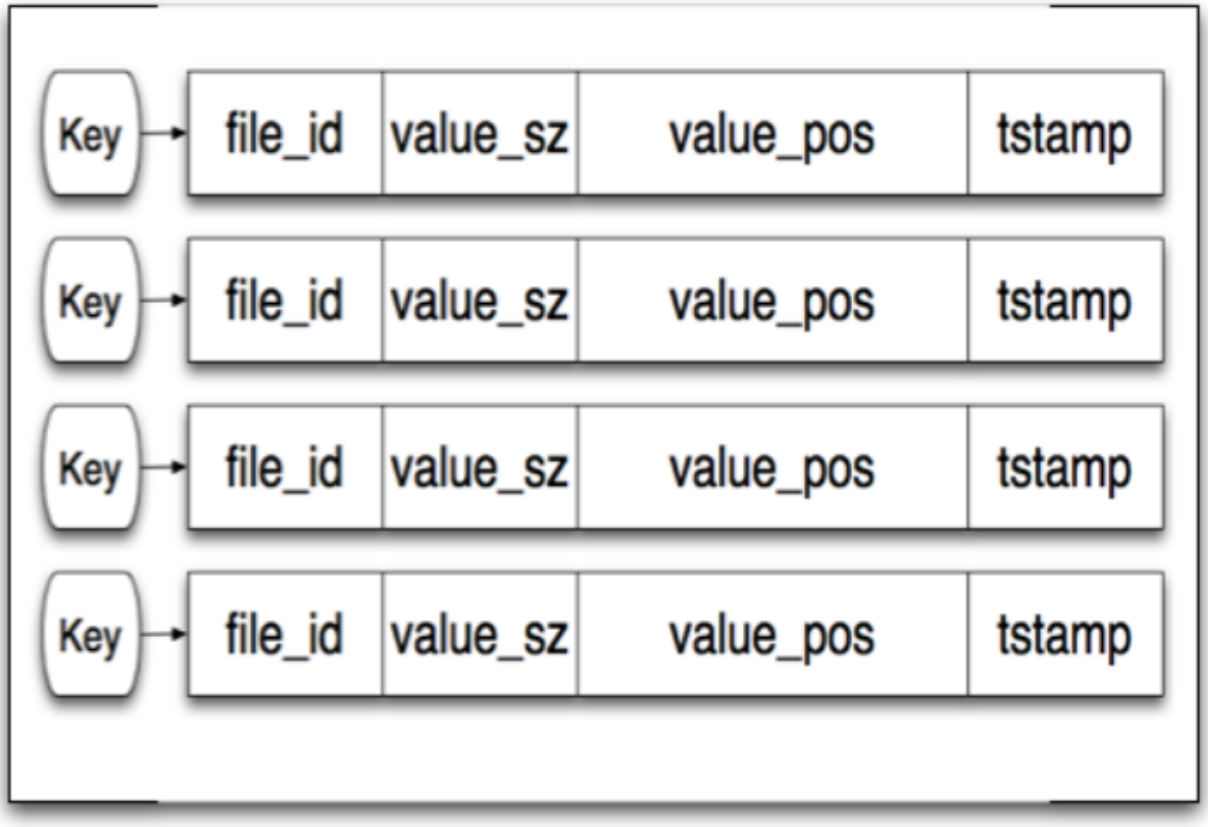


- BitCask writes each record, each with its schema (format for writing like explained in BitCask paper).
    - Data file.
    - Replica for the data file.
    - Hint file.
- Hint Files are created with each active file as well as after compaction, why?
    - Hint Files have almost fixed size for the record, each record consists of 24 bytes and the key byte representation which is 1 byte (UTF-8) in our weather status message.
    - After analyzing the file sizes, hint files are half the size of data files so it gives it advantage in reading fast (less bytes to process).
- BitCask stops writing to the same file after it reaches 32 kB, why this number?
    - So we can see the compaction running and test it.
- Integrated with ProtoBuf to compact the size of the weather status message so it takes almost 53 bytes for the whole record (key ➜ value) and the JSON value size is 130 bytes almost, it becomes 35 bytes.
- BitCask Merge & Compaction runs in a daemon thread that is scheduled to run every 60 seconds delayed from the start by 5 seconds, why?

- After almost 1 minute from running the stations, BitCask will be writing in its third file so there is some data files to be compacted.
- There was another solution that run a thread when the number of files reaches specific number.



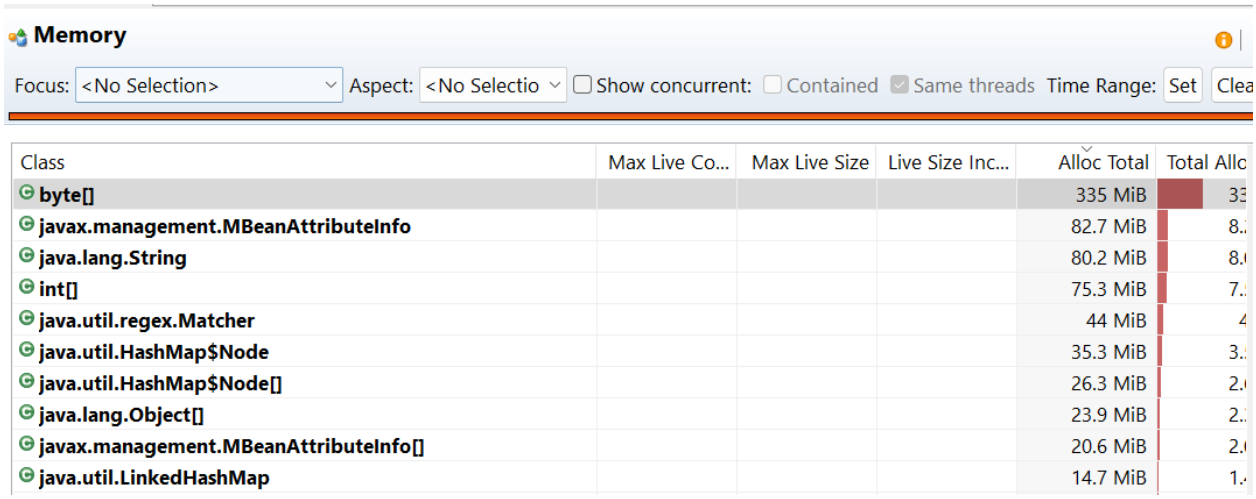
- BitCask has an in-memory structure called "**keydir**" which maps each key to the metadata of the value (value size, value position in data file, data file name).



- Hint Files help with recovery as they are used to rehash this "**keydir**" so it rebuilds the system fast.
- Recovery is executed as follows
    - List all files and filter them to replicas and hint files.
    - Sort descending on time so we have the most recent one first.
    - Skip the first one as it is the active file.
    - Read the hint file if it exists.
    - Read from the replica data file if the hint file is not found.
    - Update the "**keydir**"
- How BitCask reads the value of a given key:
    - Gets the value position of the value in the data file, the value size and gets the name of the data file.
    - Seeks the value position in this data file.
    - Returns the value.

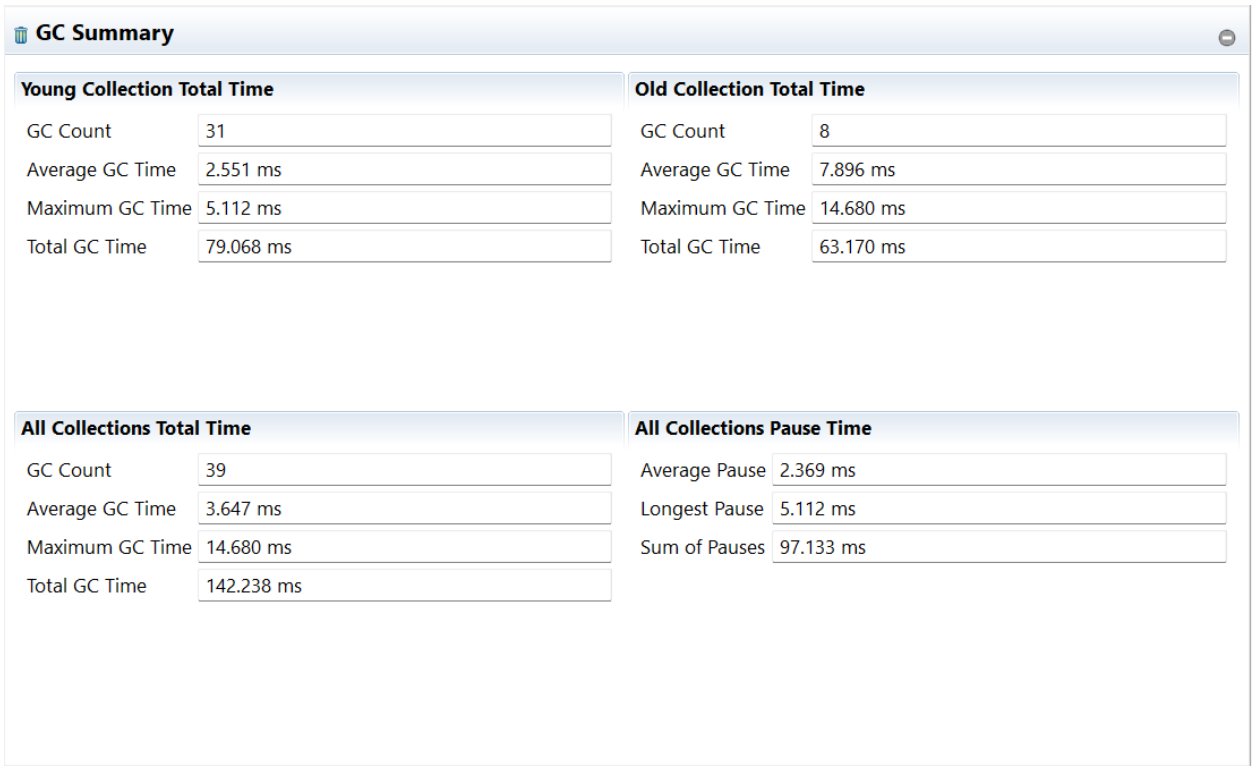
8. JFR

- JDK Mission Control (JMC) is used to visualize the JFR file.
- Top 10 Classes with highest total memory : -



Class	Max Live Co...	Max Live Size	Live Size Inc...	Alloc Total	Total Allc
byte[]				335 MiB	33
javax.management.MBeanAttributeInfo				82.7 MiB	8..
java.lang.String				80.2 MiB	8.(
int[]				75.3 MiB	7..
java.util.regex.Matcher				44 MiB	4
java.util.HashMap$Node				35.3 MiB	3..
java.util.HashMap$Node[]				26.3 MiB	2.(
java.lang.Object[]				23.9 MiB	2..
javax.management.MBeanAttributeInfo[]				20.6 MiB	2.(
java.util.LinkedHashMap				14.7 MiB	1..

- GC pauses count & maximum pause duration : -



- List of I/O operations : -