NOTES

Docker-compose:

Zookeeper (zoo1)

Purpose: Zookeeper is a centralized service for maintaining configuration information and providing naming, distributed synchronization and clustering services. In the context of Kafka, it is used to manage the Kafka cluster, maintaining information on the status of brokers and topics.
Function as a bridge: Zookeeper acts as a bridge between the Kafka brokers (kafka1, kafka2, kafka3), helping to manage configuration and coordination between them.

Kafka Brokers (kafka1, kafka2, kafka3)

Purpose: Kafka brokers are responsible for storing and managing topics and messages in the Kafka cluster. They allow messages to be produced and consumed.
Configuring multiple brokers: Having multiple brokers increases the resilience and scalability of the Kafka cluster, enabling load balancing and fault tolerance.

Kafka Connect (kafka-connect)

Purpose: Kafka Connect is a tool for importing and exporting data between Kafka and other systems. It makes it easy to integrate data from different sources and destinations (such as databases, file systems, etc.) with Kafka.

Schema Registry (schema-registry)

Purpose: Schema Registry provides a schema management service for data in Avro format. It allows schema evolution (adding new fields, changing data types) without breaking data compatibility.
Function: It ensures that data producers and consumers are using the same schema, helping to avoid data compatibility problems in Kafka.

In summary, here is the function of each component:

- Zookeeper (zoo1): Coordinates and manages the Kafka cluster.
- Kafka Brokers (kafka1, kafka2, kafka3): Store and manage messages and topics.
- Kafka Connect (kafka-connect): Facilitates data integration between Kafka and other systems.
- Schema Registry (schema-registry): Manages schemas for Avro data in Kafka.
- Kafka UI (kafka-ui): Graphical interface for monitoring and managing Kafka.

Internal Kafka Connect topics

The topics mentioned in your docker-compose.yml have specific functions within Kafka Connect:

- **CONNECT_CONFIG_STORAGE_TOPIC: compose-connect-configs**
  - Stores the connector configurations.
  - Usefulness: Allows configurations to be shared and managed centrally.

CONNECT_OFFSET_STORAGE_TOPIC: compose-connect-offsets
Stores the offsets of the data processed by the connectors.
Usefulness: Ensures that data is processed only once and that processing can be resumed from the correct point in the event of failures.

CONNECT_STATUS_STORAGE_TOPIC: compose-connect-status
Stores the status of connectors and tasks.
Use: Monitors and manages the status of connectors, allowing restarts and recovery in the event of failures.

To manage the retention of messages in a Kafka topic, you need to configure the retention properties in Kafka. There are several options for this, depending on how you want to manage retention. The two main settings are retention time and retention size.

Kafka Retention Settings

Retention Time
log.retention.hours: Sets the time in hours that a message will be retained in the topic before it is eligible for deletion. The default value is 168 hours (7 days).
log.retention.minutes and log.retention.ms: Can be used to define retention in minutes or milliseconds respectively.

Retention Size
log.retention.bytes: Defines the maximum size in bytes that a log can reach before old messages are removed. The default value is -1, which means there is no size limit based on bytes.

Clean-Up Policy
log.cleanup.policy: Can be set to delete (to delete older messages when retention is reached) or compact (to compact logs).

Summary

Messages are automatically deleted after the configured retention period.
You should explicitly configure retention policies as required.
Use log.retention.hours, log.retention.bytes and log.cleanup.policy to control retention.

Apache Spark

uses memory (RAM) to process data, keeping most of the intermediate data and cache in memory to ensure fast processing. However, it can also use disk to store temporary and intermediate data when RAM is not enough. Spark's memory configuration can be adjusted to optimize performance based on the available system resources and the specific workload.

Configuring Memory in Spark

1. executor and driver memory

Executor Memory (spark.executor.memory): Amount of memory allocated to each executor. Each executor performs part of the work of a Spark application.

Driver Memory (spark.driver.memory): Amount of memory allocated to the driver, which coordinates the executors.

```
spark = SparkSession.builder \
   .appName("KafkaSparkApp") \
   .config("spark.executor.memory", "4g") \
   .config("spark.driver.memory", "4g") \
   .getOrCreate()
```

Driver: Coordinates the work of the executors and maintains the state of the SparkContext. In general, the driver doesn't need much memory unless it is dealing with a large number of tasks or accumulating a lot of results.

Executor: Executes the tasks sent by the driver. Each executor needs enough memory to process the data and perform the necessary operations.

2. Cache and Disk Storage

Disk (spark.local.dir): Local directory where intermediate data can be stored if there is not enough memory.

```
spark = SparkSession.builder \
   .appName("KafkaSparkApp") \
   .config("spark.local.dir", "/path/to/spark/temp") \
   .getOrCreate()
```

3. Other memory settings

Memory for Cache (spark.storage.memoryFraction): Fraction of the executor's memory that will be used to store cached data.

Memory for Shuffle (spark.shuffle.memoryFraction): Fraction of the executor's memory that will be used for shuffle operations.

```
spark = SparkSession.builder \
   .appName("KafkaSparkApp") \
   .config("spark.storage.memoryFraction", "0.3") \
   .config("spark.shuffle.memoryFraction", "0.2") \
   .getOrCreate()
```

How the memory works

RAM memory:
- Processing: Spark stores intermediate data in memory and caches it for fast processing.
- Caching: Used to store frequently accessed or reusable data in memory to speed up processing.

Disk Memory:

Spill to Disk: When RAM memory is not sufficient to store all intermediate data, Spark writes this data to temporary disk.
Checkpointing: Spark can store checkpoints on disk to ensure recovery in the event of failures.

Distribution of Tasks by the Driver

By default, the driver distributes tasks to executors fairly, using a scheduler that tries to balance the workload. Each task is a unit of work that can be executed by a runner, and the driver tries to maintain a balance so that no runner is overloaded or idle.

Other Relevant Driver Settings

spark.driver.maxResultSize: Sets the maximum size of the results that the driver can collect from all executors. This is useful to prevent the driver from running out of memory when collecting large results.

.config("spark.driver.maxResultSize", "1g")

spark.driver.bindAddress: The driver's bind address, which can be useful in environments with complex network configurations.

.config("spark.driver.bindAddress", "0.0.0.0")

Memory and Driver Cores: Adjusting these settings can improve application performance, especially in complex or data-intensive operations.
Directories and Extra Options: Configuring local directories and additional JVM options helps manage temporary storage and logging.
Task Distribution: The driver distributes tasks to executors evenly by default.

Functions of Spark Components

Spark Master:
Acts as the central point of control for the Spark cluster.
It manages resources and allocates tasks to workers.

Spark Workers:
Carry out the tasks assigned by the master.
They are responsible for carrying out the actual processing of the data.

Spark Submit:
This is the client that sends the Spark application to the master.
Once the master receives the submission, it distributes the tasks to the workers.

The script you are submitting (kafka_spark_app.py) is configured to read streams from Kafka, process the data and send it back to Kafka. As streams are continuous, the script will remain running continuously, processing new data as it arrives.

What is Watermarking?

Watermarking is a technique used in stream processing to deal with late data. In many streaming scenarios, especially when data is collected from distributed sources, there can be variation in the time the data arrives. A watermark defines a point in time up to which the streaming framework (such as Apache Spark) considers the data to be “on-time”. Data that arrives after this point may be considered late and may be discarded or processed differently.

When to Use Watermarking?

Disordered or Delayed Data: If your Kafka data may arrive late or out of order, using watermarks can be useful to ensure you don't process duplicate data or lose data.

Time Windows: If you are aggregating data into time windows, such as calculating averages or counts over time intervals, watermarks help you manage delayed data efficiently.

When Not to Use Watermarking?

Real-Time Data Without Significant Delays: If you're sure that your Kafka data always arrives in real time and you don't expect delayed data, you may not need watermarks.

Simple Processing: If you're simply reading data and not aggregating over time windows or concerned about the order of the data, watermarks may be unnecessary.

Apache Kafka provides guarantees about the order of data, but these guarantees are limited to the partition level. In other words, within the same partition, Kafka guarantees that the data will be consumed in the order in which it was produced. However, there are no order guarantees between different partitions of the same topic.
Kafka's ordering guarantees

Order within partition: Kafka guarantees that the records within a partition will be delivered in the order they were sent by the producer.
Order between partitions: There are no ordering guarantees between different partitions of a topic. This means that if you have multiple partitions, records may arrive out of order when consumed.

When to Worry About Data Order

Multiple Partitions: If the Kafka topic has multiple partitions, you may receive out-of-order data when reading from multiple partitions simultaneously.
Network delays: Even with a single partition, network delays can occur that result in data arriving out of order.

Using Watermarks and Time Windows

If you are aggregating data in time windows and need to ensure that the data is processed correctly, even if it arrives late, the use of watermarks is recommended. This is especially important in distributed scenarios where network delays or variation in data arrival times can occur.

By default, the Spring Kafka Listener starts consuming messages from the last confirmed offset. This means that it will only receive recent messages and not old messages that have already been consumed. If you want to consume old messages that are already in the cluster, you need to adjust the consumer configuration.
Options for consuming old messages

Set the Offset to earliest:
Set the consumer to start consuming from the beginning of the topic, i.e. from the first message.

```
@Bean public ConsumerFactory<String, Object> consumerFactory() { Map<String, Object>
props = new HashMap<>(); props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG,
"localhost:9092,localhost:9093,localhost:9094");
props.put(ConsumerConfig.GROUP_ID_CONFIG, "my-group");
props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
StringDeserializer.class);
props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
JsonDeserializer.class); props.put(JsonDeserializer.TRUSTED_PACKAGES, "*");
props.put(ErrorHandlingDeserializer.VALUE_DESERIALIZER_CLASS,
JsonDeserializer.class.getName()); props.put(JsonDeserializer.VALUE_DEFAULT_TYPE,
DataTypeKafka.class.getName());
props.put(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG, "earliest"); return new
DefaultKafkaConsumerFactory<>(props); }
```

```
window_agg_df = filtered_df \
   .withWatermark("timestamp", "1 minute") \
   .groupBy(window(col("timestamp"), "1 minute")) \
   .agg(avg("age").alias("average_age"))
```

Watermark: The watermark is set to 1 minute, which means that Spark waits up to 1 additional minute after the end of each window to receive delayed data. After this time, any data received with a timestamp within this window will be discarded.

windows: Spark creates 1-minute windows for aggregation. Each window is independent and calculates the average of the data received only within that specific minute.

```
final_query = final_json_df.writeStream \
   .format("kafka") \
   .option("kafka.bootstrap.servers", "kafka1:19092,kafka2:19093,kafka3:19094") \
   .option("topic", "aggregated_data") \
   .option("checkpointLocation", "/tmp/checkpoints/final1") \
   .outputMode("complete") \
   .start()
```

Output Mode

Streaming's output modes:

Append: Only new results are added to the output sink. This only works when data is output exactly once and not updated.

Update: Issues incremental updates to the output sink. Each update represents a change in the state of the aggregation.

Complete: Outputs the complete state of the aggregation for each micro-batch.