Apache Kafka

TABLE OF CONTENTS

PREFACE

This cheatsheet will provide you with the essential knowledge, commands, and best practices to navigate the Kafka ecosystem with ease. This concise yet comprehensive resource is designed to be your go-to guide for all things Kafka.

INTRODUCTION

Apache Kafka is a robust, distributed streaming platform that has become an integral part of modern data processing and real-time event streaming architectures. It was originally developed by LinkedIn and open-sourced as an Apache Software Foundation project. Kafka provides a reliable, high-throughput, low-latency, and fault-tolerant way to publish, subscribe to, store, and process streams of data.

At its core, Kafka is a distributed event streaming system that is designed to handle a vast amount of data in a highly scalable, durable, and fault-tolerant manner. It's widely used for building real-time data pipelines, monitoring systems, log aggregation, and more. Kafka's architecture is built around a publish-subscribe model, where data producers (publishers) send records to topics, and data consumers (subscribers) read from those topics.

This cheatsheet serves as a quick reference guide to key concepts, components, and commands in Apache Kafka. Whether you're a developer, data engineer, or operations professional, it will help you navigate the Kafka ecosystem, set up your Kafka clusters, and perform common tasks with ease. From understanding Kafka's fundamental components to managing topics, producers, and consumers, you'll find the essential information you need to work effectively with Kafka in a compact and accessible format.

Let's dive into the world of Apache Kafka and unlock the power of real-time data streaming for your projects and applications.

INSTALLING AND RUNNING APACHE KAFKA

To install and start Apache Kafka on your local machine, follow these step-by-step instructions. We'll cover a basic setup for development and testing purposes.

- **Prerequisites**
  - Before you begin, ensure you have Java installed on your system, as Kafka is a Java-based application.
- **Download Apache Kafka**
  - Visit the Apache Kafka website's download page: Apache Kafka Downloads.
  - Download the latest stable version of Kafka, typically a binary release package. Choose the binary release that matches your operating system (e.g., `kafka_2.13-3.1.0.tgz` for a Unix-like system).
- **Extract the Kafka Archive**
  - Navigate to the directory where you downloaded Kafka.
  - Use the following command to extract the Kafka archive (replace the filename with your downloaded version): `tar -xzf kafka_2.13-3.1.0.tgz`
- **Start the Kafka Server (Broker)**
  - Change your working directory to the Kafka installation directory.
  - Kafka depends on Apache ZooKeeper for distributed coordination. You need to start ZooKeeper before starting Kafka. You can use the ZooKeeper scripts provided with Kafka, run the following command: `bin/zookeeper-server-start.sh config/zookeeper.properties` (leave the ZooKeeper terminal running in the background)
  - Open a new terminal window and start the Kafka server by running the following command: `bin/kafka-server-start.sh config/server.properties` (leave the Kafka server terminal running in the background)
- **Create a Kafka Topic**
  - We will create a Kafka topic named "test-topic" to help us follow along the rest of this cheatsheet. In a new terminal, use the following command: `bin/kafka-topics.sh --create --topic test-topic --bootstrap-server localhost:9092 --replication-factor 1 --partitions 1`

That's it! You now have Apache Kafka up and

running on your local machine. You can start producing and consuming messages, create more topics, and explore the rich set of features Kafka has to offer for event streaming and data processing.

APACHE KAFKA COMPONENTS

Apache Kafka consists of three major components, the role of each is summarized below:

System	Description
Publish/Subscribe	The Publish/Subscribe messaging system enables distributed streaming of events. It is designed to handle a vast amount of data in a highly scalable, durable, and fault-tolerant manner.
Kafka Connect	The Kafka Connect framework is used to connect external data sources (event producers) and data sinks (event consumers) to the platform.
Kafka Streams	Client library that enables processing of events/data in real-time.

As stated above, Apache Kafka runs in distributed clusters. A cluster node is being referred to as a Broker. Kafka Connect integrates Brokers with external clients that produce events or consume event data. Producers and consumers utilize Kafka's publish/subscribe messaging system to achieve instant exchange of event data between servers, processes, and applications. In case data handling is required, the Kafka Streams component can be used for real-time event processing.

THE PUBLISH-SUBSCRIBE SYSTEM

The publish/subscribe system is responsible for the distributed streaming of a vast amount of data in a highly scalable, durable, and fault-tolerant manner. Below are its main key concepts:

Concept	Description
Broker	A Kafka server instance that stores and manages data. Producers publish messages to brokers, and consumers retrieve messages from brokers. Kafka clusters consist of multiple brokers to ensure fault tolerance and scalability.
Topic	A logical channel or category for data in Kafka. It is used to categorize messages, allowing producers to publish data to a specific topic and consumers to subscribe to the topics of their interest. Topics can have multiple partitions for parallelism.
Partition	A way to horizontally distribute data within a topic. Each partition is an ordered, immutable sequence of messages. Partitions allow Kafka to achieve high throughput by distributing data across multiple brokers and enabling parallelism for consumers.
Offset	A unique sequential number assigned to each message within a topic partition.
Message (aka Record)	The means to transfer data between producers and consumers. It Contains a key, value, timestamp, and a header.

Concept	Description
Producer	A component that sends messages to Kafka topics. Producers can publish data to one or more topics, and they are responsible for specifying the topic and partition to which a message should be sent.
Consumer	A component that subscribes to one or more topics and reads messages from Kafka. Kafka supports different types of consumers, such as the high-level consumer API and the low-level consumer API. Consumers can work individually or in consumer groups for load balancing.
Consumer Group	A group of consumers that work together to consume data from a topic. Each partition within a topic is consumed by only one consumer within a group, ensuring parallel processing while maintaining order. Kafka automatically rebalances partitions among consumers in a group.
Zookeeper	Used for distributed coordination and management of Kafka clusters. While newer Kafka versions are designed to work without Zookeeper, older versions relied on it for maintaining cluster metadata and leader election. It plays a crucial role in cluster management and maintenance.

Concept	Description
Producer API	A client library or interface for producing data to Kafka. Popular programming languages like Java, Python, and others have Kafka producer libraries that allow developers to create producers.
Consumer API	A client library or interface for consuming data from Kafka. Like producer libraries, Kafka provides consumer libraries in various programming languages for reading data from Kafka topics.

Below is a Java code snippet that demonstrates how to create a Kafka producer and send a "Hello, World!" message to the "test-topic" topic created earlier.

```
import
org.apache.kafka.clients.producer.*;

import java.util.Properties;

public class KafkaProducerExample {
    public static void main(String[]
args) {
        // Set up Kafka producer
properties
        Properties props = new
Properties();
        props.put
("bootstrap.servers",
"localhost:9092");
        props.put("key.serializer",
"org.apache.kafka.common.serializati
on.StringSerializer");
        props.put(
"value.serializer",
"org.apache.kafka.common.serializati
on.StringSerializer");

        // Create a Kafka producer
```

```
        Producer<String, String>
producer = new KafkaProducer<>(
props);

        // Define the topic to send
the message to
        String topic = "test-topic";

        // The key for the message
        String key = "myKey";

        // The message to send
        String message = "Hello,
World!";

        // Create a ProducerRecord
with the topic and message
        ProducerRecord<String,
String> record = new
ProducerRecord<>(topic, key,
message);

        // Send the message to the
Kafka topic
        producer.send(record, new
Callback() {
            public void
onCompletion(RecordMetadata
metadata, Exception exception) {
                if (exception ==
null) {
                    System.out
.println("Message sent successfully
to topic " + metadata.topic());
                    System.out
.println("Partition: " + metadata
.partition());
                    System.out
.println("Offset: " + metadata
.offset());
                } else {
                    System.err
.println("Error sending message: " +
exception.getMessage());
                }
            }
        });

        // Close the producer when
done
        producer.close();
    }
}
```

In this code, we configure the Kafka producer with the necessary properties, create a producer instance, and use it to send a "Hello, World!" message with key "myKey" to the "test-topic" topic. As you can see, both the key and value are Strings, so we are using a `StringSerializer` to serialize the transferred data. It's wise to use the correct serializer based on the data types of our key and message. The callback function handles the acknowledgment of message delivery.

Below is a Java code snippet that demonstrates how to create a Kafka consumer to read messages from the "test-topic" topic.

```
import
org.apache.kafka.clients.consumer.*;
import
org.apache.kafka.common.serializatio
n.StringDeserializer;

import java.util.Collections;
import java.util.Properties;

public class KafkaConsumerExample {
    public static void main(String[]
args) {
        // Set up Kafka consumer
properties
        Properties props = new
Properties();
        props.put
("bootstrap.servers",
"localhost:9092");
        props.put("group.id", "test-
consumer-group"); // A consumer
group ID
        props.put(
"key.deserializer",
"org.apache.kafka.common.serializati
on.StringDeserializer");
        props.put
("value.deserializer",
"org.apache.kafka.common.serializati
```

```
on.StringDeserializer");

        // Create a Kafka consumer
        KafkaConsumer<String,
String> consumer = new
KafkaConsumer<>(props);

        // Subscribe to the "test-
topic" topic
        consumer.subscribe
(Collections.singletonList("test-
topic"));

        // Poll for new messages
        while (true) {
            ConsumerRecords<String,
String> records = consumer.poll(
100);

            for (ConsumerRecord
<String, String> record : records) {
                System.out.println
("Received message:");
                System.out.println
("Topic: " + record.topic());
                System.out.println
("Partition: " + record.
partition());
                System.out.println
("Offset: " + record.offset());
                System.out.println
("Key: " + record.key());
                System.out.println
("Value: " + record.value());
            }
            consumer.commitSync();
        }
    }
}
```

In this code, we configure the Kafka consumer with the necessary properties, create a consumer instance, and subscribe to the "test-topic" topic. The consumer continuously polls for new messages and processes them as they arrive since Records within a partition are always delivered to consumers in offset order.

By saving the offset of the last consumed message from each partition, the consumer can resume from where it left off in case of a restart. We can either configure automatic saving of message offsets, by setting the property `enable.auto.commit` to `true` when creating the consumer, or do it manually by issuing the `commitSync()` command after consuming a batch of messages, as depicted in the example above.

Keep in mind that a message is not removed from the broker immediately after it's consumed. Instead, it is retained according to a configured retention policy. Below are two common retention policies:

- `log.retention.bytes` - sets the maximum size of records retained in a partition.
- `log.retention.hours` - sets the maximum number of hours to keep a record on the broker.

THE KAFKA CONNECT FRAMEWORK

The Kafka Connect framework is an integral part of the Apache Kafka ecosystem designed for building and running scalable and reliable data integration solutions. It serves as a powerful platform for connecting Kafka with various external data sources and sinks, enabling seamless data transfer between Kafka topics and other systems or storage mechanisms. Here are some key aspects of the Kafka Connect framework:

- **Distributed, Scalable, and Fault-Tolerant**: Kafka Connect is built to be distributed and scalable, allowing you to deploy connectors across multiple nodes to handle high data volumes. It offers fault tolerance by redistributing work among connectors if one or more instances fail. Furthermore connectors are designed to distribute data efficiently across Kafka topics and manage data flow from source to destination, ensuring robust and scalable data pipelines.
- **Source and Sink Connectors**: Kafka Connectors come in two primary flavors: source connectors and sink connectors. Source connectors pull data from external systems into Kafka, while sink connectors write data from Kafka topics to external systems.
- **Pre-Built Connectors**: Kafka Connect offers a wide range of pre-built connectors for popular

systems and databases, such as databases (e.g., MySQL, PostgreSQL), cloud storage (e.g., Amazon S3), message queues (e.g., RabbitMQ), and more. These connectors simplify integration with these systems without the need for custom coding.

- **Configuration-Driven**: Connectors are configured using simple configuration files or via REST APIs. The configuration defines how data should be extracted, transformed, and loaded between Kafka and the external system.
- **Schema Support**: Kafka Connectors often support data serialization and deserialization using schemas. This allows connectors to handle structured data formats like Avro, JSON, or others, maintaining compatibility and data quality.
- **Transformation and Enrichment**: You can apply transformations and enrichments to data as it flows through Kafka Connect. These transformations can include filtering, mapping, and more, depending on your requirements.
- **Real-time Data Processing**: Kafka connectors facilitate real-time data streaming, enabling you to process and analyze data as it flows through your pipeline. This is crucial for applications that require timely insights and actions based on the data.
- **RESTful APIs**: Kafka Connect provides RESTful APIs for managing connectors, configurations, and monitoring the status of connectors. This makes it easier to interact with and manage the connectors programmatically. See the Apache Kafka documentation for more information.
- **Pluggable Architecture**: You can extend Kafka Connect by creating custom connectors for systems not covered by the pre-built connectors. This pluggable architecture encourages community contributions and supports a wide range of integration use cases.
- **Rich Ecosystem**: The Kafka Connect ecosystem has a growing collection of connectors available from various sources, including the Apache Kafka community, Confluent, and third-party developers. This ecosystem helps streamline integration with different technologies and data sources.
- **Monitoring and Error Handling**: Kafka Connect offers monitoring capabilities, allowing you to track the performance and status of connectors. Furthermore connectors often come with built-in error handling mechanisms, allowing them to handle various issues that may arise during data transfer, such as network interruptions, data format/schema mismatches, and more. This ensures data reliability and minimizes data loss.

Below is a step-by-step guide of how to stream data from a source file to a target file using the Kafka Connect framework with the Confluent Kafka Source and File Sink connectors. The File Source connector reads data from the source file and publishes it to a "test-topic." The File Sink connector subscribes to "test-topic" and writes the data to the target file as specified in its configuration.

Prerequisites

- Make sure you have Kafka and Kafka Connect set up and running.
- Download and install the Confluent Kafka Source and File Sink connectors.

Create a Kafka topic (Optional)

First, create a Kafka topic to which you want to publish the data from the source file. You can use the Kafka command-line tools for this:

```
kafka-topics.sh --create --topic
test-topic --partitions 1 --
replication-factor 1 --bootstrap-
server localhost:9092
```

Configure and start the File Source connector

Create a configuration file for the Kafka File Source connector. Here's an example configuration file (`file-source-config.properties`):

```
name=file-source-connector
connector.class=io.confluent.connect
.file.FileStreamSourceConnector
tasks.max=1
file=/path/to/source/source-file.txt
topic=test-topic
```

Update the `file` property with the actual path to

your source file.

Start the File Source connector using the Kafka Connect CLI:

```
connect-standalone config/connect-standalone.properties file-source-config.properties
```

Configure and start the File Sink connector

Create a configuration file for the Kafka File Sink connector. Here's an example configuration file (`file-sink-config.properties`):

```
name=file-sink-connector
connector.class=io.confluent.connect.file.FileStreamSinkConnector
tasks.max=1
file=/path/to/destination/destination-file.txt
topics=test-topic
format=value
```

Update the `file` property with the actual path to your target file.

Start the File sink connector using the Kafka Connect CLI:

```
connect-standalone config/connect-standalone.properties file-sink-config.properties
```

Kafka Connect is primarily designed to enable data streaming between systems, as-is. To perform complex transformations once the data is in Kafka, Kafka Streams comes into play. Nevertheless, Kafka Connect provides a mechanism to perform simple transformations per record. Let's see how to apply a custom transformation while streaming data between the files of the previous example.

Update the File Source connector configuration as shown below.

```
name=file-source-connector
connector.class=io.confluent.connect.file.FileStreamSourceConnector
tasks.max=1
file=/path/to/source/source-file.txt
topic=test-topic
transforms=uppercase
transforms.uppercase.type=org.apache.kafka.connect.transforms.RegexRouter
transforms.uppercase.regex=^.*$
transforms.uppercase.replacement=Uppercase$0
```

In this example, the `transforms` property specifies a transformation named "uppercase," which converts all data to uppercase using the `RegexRouter` transformation. The `regex` property matches the entire input, and the `replacement` property prepends "Uppercase" to each line.

KAFKA STREAMS

Kafka Streams is a powerful and lightweight stream processing library that is part of the Apache Kafka ecosystem. It allows you to build real-time data processing applications that can ingest, process, and output data from Kafka topics. Kafka Streams is designed to be easy to use, scalable, and fault-tolerant, making it an excellent choice for a wide range of stream processing use cases including real-time analytics, monitoring and alerting, fraud detection, recommendation engines, ETL (Extract, Transform, Load) processes, and more.

Here are some key characteristics and features of Kafka Streams:

- **Stream Processing Library**: Kafka Streams is not a separate service or framework; it is a library that you can include in your Java or Scala applications. This makes it easy to integrate stream processing directly into your existing Kafka-based infrastructure.
- **Real-time Data Processing**: Kafka Streams is designed for real-time processing of data streams. It allows you to analyze and transform data as it flows through the Kafka topics.
- **Exactly-Once Semantics**: Kafka Streams provides strong processing guarantees, including exactly-once processing semantics.

This ensures that each record is processed exactly once, even in the presence of failures.

- **Stateful Processing**: Kafka Streams supports stateful processing, allowing you to maintain and update local state stores as data is processed. This is useful for tasks like aggregations, joins, and windowed operations.
- **Windowed Aggregations**: Kafka Streams includes built-in support for windowed aggregations, enabling time-based operations on data, such as tumbling windows, hopping windows, and session windows.
- **Join Operations**: You can perform joins on Kafka Streams data, either by joining multiple Kafka topics or by joining a topic with a local state store.
- **Scaling and Fault Tolerance**: Kafka Streams applications can be easily scaled horizontally across multiple instances, and they are inherently fault-tolerant. In the event of an instance failure, processing can be seamlessly redistributed.
- **Integration with Kafka**: Kafka Streams is tightly integrated with Kafka, which means it benefits from Kafka's reliability and durability. It also allows for easy integration with the broader Kafka ecosystem.
- **Interactive Querying**: Kafka Streams enables interactive querying of state stores through an API. This means you can query the current state of your data at any point in time, which is valuable for building interactive applications. Details can be found in the Interactive Queries section of the Kafka documentation.
- **Flexible Deployment Options**: You can deploy Kafka Streams applications on a variety of platforms, including standalone applications, containers, cloud services, and Kubernetes.

Below is an example application that utilizes the Kafka Streams library to count the words from the "test-topic" created earlier.

```
import
org.apache.kafka.clients.consumer.Co
nsumerConfig;
import
org.apache.kafka.common.serializatio
n.Serdes;
import org.apache.kafka.streams.*;
import
org.apache.kafka.streams.kstream.KSt
ream;
import java.util.Properties;

public class WordCountStream {

    public static void main(String[]
args) {
        Properties props = new
Properties();
        props.put(StreamsConfig
.APPLICATION_ID_CONFIG, "word-count-
application");
        props.put(StreamsConfig
.BOOTSTRAP_SERVERS_CONFIG,
"localhost:9092");
        props.put(StreamsConfig
.PROCESSING_GUARANTEE_CONFIG,
"exactly_once_v2"); // Apply
exactly-once processing semantics
        props.put(ConsumerConfig
.AUTO_OFFSET_RESET_CONFIG,
"earliest");


        StreamsBuilder builder = new
StreamsBuilder();

        // Create a KStream from the
"test-topic"
        // Consumed.with() defines
the type of each record's key and
value data for de-serialization - in
this case both are translated to
Strings
        KStream<String, String>
textLines = builder.stream("test-
topic", Consumed.with(Serdes.
String(), Serdes.String()));

        // Split each line into
words and convert to lowercase
        KStream<String, String>
wordStream = textLines
            .flatMapValues(value ->
Arrays.asList(value.toLowerCase().sp
lit("\\W+")));
```

```
        // Group by the word and
count occurrences
        KTable<String, Long>
wordCounts = wordStream
            .groupBy((key, word) ->
word)
            .count();

        // Write the word counts to
a new Kafka topic "word-count-
output"
        // The Produced.with()
defines the type of each record's
key and value data for serialization
- in this case keys are Strings and
values are Longs
        wordCounts.to("word-count-
output", Produced.with(Serdes.
String(), Serdes.Long()));

        KafkaStreams streams = new
KafkaStreams(builder.build(),
props);
        streams.start();

        // Shutdown hook to
gracefully close the application
        Runtime.getRuntime
().addShutdownHook(new Thread
(streams::close));
    }
}
```

In the example above, we first set up the Kafka Streams configuration, including the Kafka broker(s) address. Then the `StreamsBuilder` is used to build the stream processing topology. A `KStream` is created from the "test-topic." Each line is split into words and converted to lowercase. Words are grouped and counted to create a `KTable` with word counts. The word counts are written to a new Kafka topic named "word-count-output" and finally, the Kafka Streams application is started, and a shutdown hook is added to close the application gracefully.

Let us delve into the nitty-gritty of the Kafka Streams API. Below you can find details regarding the basic concepts, utilities and classes.

A topic can be viewed as either a `KStream` or a `KTable`. With `KStream` each record is treated as an append to the stream, whereas with `KTable` each record is treated as an update to the value of its key in a table. So if a topic has two or more records with the same key, viewing it as a `KStream` allows for all records to be retained whereas viewing it as a `KTable` only the last record for each key is retained in the stream.

KSTREAM

Below are some basic functions of the `KStream` and examples of each.

`map`: Transforms each record in the stream by applying a mapping function. Here for every input record we return a new record which has its key and value equal to key of the input. As you can see records are represented by the `KeyValue` class.

```
KStream<String, String> sourceStream
= builder.stream("input-topic");
KStream<String, String> mappedStream
= sourceStream.map((key, value) ->
new KeyValue<>(key, key));
```

`filter`: Filters records based on a provided predicate. Here for every input record the filter function is executed. If it evaluates to true the record is retained in the new stream.

```
KStream<String, String> sourceStream
= builder.stream("input-topic");
KStream<String, String>
filteredStream = sourceStream.
filter((key, value) -> value
.startsWith("filter"));
```

`flatMap`: Transforms each input record into zero or more output records. Here we split the value of each input record to individual words and return each word in a new record with the same key as the input. As you can see records are represented by the `KeyValue` class.

```
KStream<String, String> sourceStream
```

```
= builder.stream("input-topic");
KStream<String, String>
flatMappedStream = sourceStream
.flatMap((key, value) -> Arrays
.asList(value.split("\\W+")).stream(
).map(v -> new KeyValue<>(key, v)));
```

`join`: Joins records of the input stream with records from another `KStream` or `KTable` if the keys from the records match (inner-join). Returns a stream of joined records with values combined based on the function provided. Here a joined record's value is the concatenation of the two input records values.

```
KStream<String, String> sourceStream
= builder.stream("input-topic");
KStream<String, String> joinStream =
sourceStream.join(anotherStream,
(value1, value2) -> value1 + " " +
value2);
```

`to`: Writes the stream to a Kafka topic.

```
KStream<String, String> sourceStream
= builder.stream("input-topic");
sourceStream.to("output-topic");
```

`foreach`: Performs a function on each record without producing a new output stream.

```
KStream<String, String> sourceStream
= builder.stream("input-topic");
sourceStream.foreach((key, value) ->
System.out.println("Key: " + key +
", Value: " + value));
```

`branch`: Splits the stream into multiple output streams based on a set of predicates. Here we split the input stream in three branched based on the value of each record.

```
KStream<String, String> sourceStream
= builder.stream("input-topic");
KStream<String, String>[] branches =
sourceStream.branch(
    (key, value) -> value.contains
("A"),
    (key, value) -> value.contains
("B"),
    (key, value) -> value.contains
("C")
);
```

`groupBy`: Groups records by a new key (specified by the mapping function), allowing you to perform aggregation and windowed operations. The method returns a `KGroupedStream` to allow specialized data processing. Here we just group by records existing keys.

```
KStream<String, String> sourceStream
= builder.stream("input-topic");
KGroupedStream<String, String>
groupedStream = sourceStream.
groupBy((key, value) -> key);
```

KGROUPEDSTREAM

A `KGroupedStream` is an intermediate representation that results from applying the `groupBy` operation to a `KStream`. It represents a stream of records that have been grouped based on a specific key, allowing for further aggregation and transformation of data. You can perform various operations on a `KGroupedStream`, such as aggregations and windowed operations, which make it a key component for analyzing and processing data in a Kafka Streams application. Lets see some examples.

`aggregate`: Aggregates records within a grouped stream using a provided aggregation function. The first attribute is a function that returns the initial aggregate value and the second is the aggregation function. Here we aggregate the length of the value of each record in a group and return the results for all groups in a `KTable`.

```
KTable<String, Long> aggregatedTable
= groupedStream.aggregate(
    () -> 0L,
    (key, value, aggregate) ->
aggregate + value.length());
```

`count`: Counts the number of records in each group of the grouped stream. Return the results for all groups in a `KTable`.

```
KTable<String, Long> countTable =
groupedStream.count();
```

`reduce`: Combines the values of records in each group of the grouped stream. Here we combine the values of all records of a group by concatenating them in a comma separated string. The results for all groups are returned in a `KTable`.

```
KTable<String, Long> countTable =
groupedStream.reduce((value1,
value2) -> value1 + ", " + value2);
```

`windowedBy`: Groups records into windows for time-based operations. Here we further group the events in the stream in 5 minute time windows.

```
TimeWindowedKStream<String, String>
windowedStream = groupedStream
.windowedBy(TimeWindows.of(Duration.
ofMinutes(5)));
```

KTABLE AND KGROUPEDTABLE

Details on the `KTable` and `KGroupedTable` operations can be found in the Kafka Documentation.

NOTABLE APACHE KAFKA ALTERNATIVES

These alternatives cover a spectrum of features and use cases, making them strong contenders for various messaging and event streaming scenarios. The choice of the most suitable alternative depends on your specific requirements, including factors such as scalability, cloud compatibility, and lightweight design.

Project	Description
Apache Pulsar	Apache Pulsar is a high-performance distributed messaging and event streaming platform. It offers multi-tenancy, strong durability, and geo-replication capabilities. Pulsar is known for its native support for fine-grained authorization and authentication.
Amazon Kinesis	Amazon Kinesis is a fully managed, cloud-based data streaming and processing service offered by AWS. It's designed for real-time data ingestion, storage, and processing at scale, making it well-suited for cloud-centric applications.
RabbitMQ	RabbitMQ is a robust open-source message broker that supports various messaging patterns, including publish-subscribe and work queues. It's known for its flexibility, support for multiple protocols, and extensive plugin ecosystem.

ADDITIONAL RESOURCES

This cheatsheet has provided you with essential information to kickstart your journey into the world of Apache Kafka. If you are seeking to deepen your knowledge, the following resources cover a wide range of learning materials, from official documentation and online courses to articles, and community resources. Dive into these Kafka-related materials to enhance your understanding and proficiency in this robust event streaming platform.

- Apache Kafka Documentation: The official documentation provides comprehensive information on Kafka's architecture,

configuration, and usage..

- Confluent Platform Training: Confluent, the company founded by the creators of Kafka, offers free online courses and tutorials on Kafka.
- LinkedIn Engineering Blog: LinkedIn's engineering blog often features Kafka-related articles, including best practices and use cases.
- Kafka Users Mailing List: A place to ask questions, share knowledge, and get help from the Kafka community.
- Apache Kafka GitHub Repository: The official source code repository for Kafka.
- Confluent GitHub Repository: Confluent's GitHub repository includes various Kafka-related projects and connectors.
- Confluent Community Hub: A marketplace for Kafka connectors, Kafka Streams applications, and other Kafka-related components shared by the community.