**Confluent**:

* Confluent is an enterprise platform built on Apache Kafka. Essentially, Confluent is Kafka with some additional enterprise features.
* We will be using and installing Confluent Community, a free version of Confluent. It includes all the features Kafka and Confluent.

**Apache Kafka:**

* Kafka is known as “**Distributed streaming platform**”
* Publish and subscribe to streams of data records.
* Store the records in a fault tolerant and scalable fashion.
* Process streams of records in real time.
* Kafka is written in Java
* Originally created in LinkedIn.
* Become open source in 2011

**Topics:**

* Topics are at the core of everything you can do in Kafka
* A Topic is a data feed to which data records are published and from which they can be consumed.
* Publishers send data to a topic, and Subscribers read data from the topic. This is known as **Publish-Subscriber Messaging**, or simply **pub/sub** for short.

**The Topic Log:**

* A section of a Topic’s log
* Kafka topics each maintain a log.
* The **log** is an ordered, immutable list of data records.
* The log for a Kafka Topic is usually divided into multiple **partitions**. This is what allows Kafka to process data efficiently and in a scalable fashion.

Graphical user interface, application

Description automatically generated

**Offset:**

* Each record in a partition has a sequential, unique Id is called Offset.
* The sequential and Unique Id of a data record within a partition.

**Producers:**

* Something that writes data to a Topic.
* The producer is the publisher in the pub/sub model. The producer is simply an application that communicates with the cluster and can be a separate process or on a different server.
* Producers write data records to the Kafka topic.
* For each new record, The producer determines which partition to write to, often in a simple round-robin fashion. You can customize this to use a more sophisticated algorithm for determining which topic to write a new record to.

Graphical user interface

Description automatically generated

**Consumers:**

* Something that reads data from a Topic.
* A Consumer is the subscriber in the pub/sub model. Like producers, consumers are external applications that can be a separate process or on a different server from Kafka itself.
* Consumers read data from Kafka Topics.
* Each Consumer controls the offset it is currently reading from each partition, and consumers normally read records in order based on the offset.
* You can have any number of consumers for a topic, and they can all process the same records.
* Records are not deleted when they are consumed. They are only deleted based upon a configurable retention period.

Graphical user interface

Description automatically generated

**Consumer Groups:**

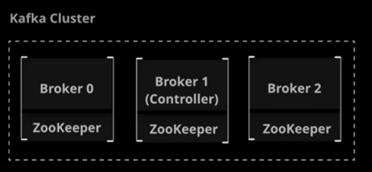
* A group of multiple consumers. Normally, multiple consumers can all consume the same record from a topic, but only one consumer in a consumer group will consume the record.
* By default, all consumers will process all records, but if you want to scale your record processing so that multiple instances can process the data without 2 instances processing the same record?
* You can place consumers into **Consumer Groups**. Each record will be consumed by exactly one consumer per consumer group.
* With consumer groups, Kafka dynamically assigns each partition to exactly one consumer in the groups. If you have more consumers than partitions, some of the consumers will be idle and will not process records.

A picture containing text, satellite

Description automatically generated

**Brokers**:

* The central component of Kafka architecture is the broker.
* Brokers are servers that make up a Kafka cluster (one or more brokers).
* Producers and Consumers communicate with brokers in order to publish and consume messages.



**Zookeeper:**

* Kafka depends on the underlying technology called Zookeeper.
* Zookeeper is a generalized cluster management tool. It manages the cluster and provides consistent, distributed places to store cluster configuration.
* Zookeeper coordinates communication throughout the cluster, adds and removes brokers, and monitors the status of the nodes in the cluster.
* It is often installed alongside Kafka, but can be maintained on a completely separate set of servers.
* Kafka officially deprecated Zookeeper version 3.5 onwards.
* Kraft metadata management is replacing the zookeeper.

**The Controller:**

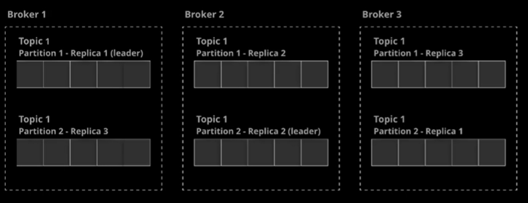
* In a Kafka cluster, one broker is dynamically designated as the controller.
* The controller coordinates the process of assigning partitions and data replicas to nodes in the cluster.
* Every cluster has exactly one controller. If controller goes down, another node will automatically become the controller.

**Networking:**

* Kafka uses a simple TCP protocol to handle messaging communication.

**Replication:**

* Kafka is designed with fault tolerance in mind. As a result, it includes built-in support for replication.
* **Replication** means storing multiple copies of any given piece of data.
* In Kafka, every topic is given a configurable replication factor.
* The **replication factor** is the number of replicas that will be kept on different brokers for each partition in the topic.



**Leaders:**

* In order to ensure that messages in a partition are kept in a consistent order across all replicas, Kafka choses a leader for each partition.
* The Leader handles all reads and writes for the partition. The leader is dynamically selected and the leader goes down, the cluster attempts to choose a new leader through a process called **Leader Election**.

**In-Sync Replicas:**

* Kafka maintains a list of **In-Sync Replicas (ISR)** for each partition.
* ISRs are replicas that are up to date with the leader. If a leader dies, the new leader is elected from among the ISRs.
* By default, if they are no remaining ISRs when a leader dies, Kafka waits until one becomes available. This means that producers will be on hold until a new leader can be elected.
* You can turn on **unclean leader election**, allowing the cluster to elect a non-in-sync replica in this scenario.

**Compaction**

* If multiple records with the **same key** are sent into the Kafka Topic, Kafka **compacts** and keeps only the **most recent value**.
* Older records are periodically deleted in the background.
* Avoiding unnecessary data storage while keeping **latest updates**
* Maintaining **latest state** (e.g., user profiles, configurations)
* Topics with compaction enabled are useful for **stateful applications** like Order status tracking, application configuration settings and user profiles

kafka-topics.sh --create --topic compacted-topic \

--bootstrap-server localhost:9092 \

--partitions 3 --replication-factor 1 \

--config **cleanup.policy=compact**

**Message Ordering**

**Single Partition Message Ordering**

* **Single Partition**: Use a single partition for the topic (this may limit scalability).
* **Partition Key**: Ensure related messages are sent with the same key so that they are routed to the same partition.

**Producer:**

ProducerRecord<Long, UserEvent> producerRecord = new ProducerRecord<>(Config.SINGLE\_PARTITION\_TOPIC, userEvent); Future<RecordMetadata> future = producer.send(producerRecord);

**Consumer:**

consumer = **new** **KafkaConsumer**<>(consumerProperties); consumer.subscribe(Collections.singletonList(Config.SINGLE\_PARTITION\_TOPIC));

ConsumerRecords<Long, UserEvent> records = consumer.poll(TIMEOUT\_WAIT\_FOR\_MESSAGES);

**Multiple Partition Message Ordering**

* Control the partition key so that related messages go to the same partition.
* In a case where you are consuming messages from multiple partitions, the consumer guarantees order only **within a single partition**. If your consumer listens to multiple partitions, you may not get messages in the order they were sent across the entire topic.

**Producer:**

Future<RecordMetadata> future = producer.send(**new** **ProducerRecord**<>(Config.MULTI\_PARTITION\_TOPIC, sequenceNumber, userEvent));

**Consumer:**

consumer.subscribe(Collections.singletonList(Config.MULTI\_PARTITION\_TOPIC));

ConsumerRecords<Long, UserEvent> records = consumer.poll(TIMEOUT\_WAIT\_FOR\_MESSAGES);

**Producer API:**

* Allows you to build producers that publishers messages to Kafka

**Consumer API:**

* Allows you to build consumers that read Kafka messages.

**Streams API:**

* Allows you to read from input topics, transform data, and output it to output topics

**Connect API:**

* Allows you to build custom connectors, which pull from or push to specific external systems.

**Admin Client API:**

* Allows you to manage and inspect higher-level objects like topics and brokers.

**Streams:**

* **Kafka streams** allow us to build applications that process Kafka data in real-time with ease.
* **A Kafka stream application** in an application where both the input and the output are stored in Kafka topics.
* **Kafka streams** is a client library API that makes it easy to build these applications.

**Streams Transformations:**

* Kafka Streams provides a robust set of tools for processing data. The Kafka cluster itself serves as the backend for data management and storage.

**Stateless Transformations** do not require any additional storage to manage the state.

* **Branch**: Splits a stream into multiple streams based on a predicate.
* **Filter**: Removes messages from the stream based on a condition.
* **FlatMap**: Takes input records and turns them into a different set of records.
* **ForEach**: Performs an arbitrary stateless operation on each record. This is a terminal operation and stops further processing.
* **GroupBy/GroupByKey**: Groups records by their key. This is required to perform stateful transformations. This can be used to group records that share the same key.
* **Map**: Allows you to read a record and produce a new, modified record.
* **Merge**: Merges two streams into one stream
* **Peek**: Similar to Foreach, but does not stop processing.

**Stateful Transformations**

* Require a state store to manage the state. Aggregations are stateful transformations that always operate on these groups of records sharing the same key.

**Aggregations:**

* **Aggregate**: Generates a new record from a calculation involving the grouped records.
* **Count**: Counts the number of records for each grouped key.
* **Reduce**: Combines the grouped records into a single record.

**Kafka Streams Joins:**

* Joins are used to combine streams into one new stream

**Co-partitioning:**

* When joining streams, the data must be co-partitioned.
* Same number of partitions for input topics.
* Same partitioning strategies for producers.
* You can avoid the need for co-partitioning by using a **GlobalKTable**.
* With **GlobalKTables**, all instances of your streams application will populate the local table with data from all-partitions.

**Kafka Streams Joins:**

* **Inner Join**: The new stream will contain only records that have a match in both joined streams.
* **Left Join**: The new stream will contain all records from the stream, but only matching records from the joined stream.
* **Outer Join**: The new stream will contain all records from both streams.

**Kafka Streams Windowing:** Windows are similar to groups in that they deal with a set of records with the same key. However, windows further subdivide groups into “**time buckets**”

* **Tumbling Time Windows:** Windows are based on time periods that never overlap or have gaps between them.
* **Hoping Time Windows:** Time-based, but can have overlaps or gaps between windows.
* **Sliding Time Windows:** These windows are dynamically based on the timestamps of records rather than a fixed point in time. They are only used in joins.
* **Session Windows:** Creates windows based on periods of activity. A group of records around the same timestamp will form a session window, whereas a period of “**idle time**” with no records in the group will not have a window.

**Late Arriving Records:**

* In real-world scenarios, it is always possible to receive out-of-order data.
* When records fall into a time window received after the end of that window’s grace period, they become known as **late-arriving records**.
* You can specify a **retention period** for a window. Kafka streams will retain old window buckets during this period so that late-arriving records can still be processed.
* Any records that arrive after the retention period has expired will not be processed.

**KStreams Vs KTables:**

* **Streams**: Each record is a self-contained piece of data in an unbounded set of data. New records do not replace an existing piece of data with a new value.

1. Credit card transactions in real-time
2. A real time log of attendees checking in to a conference.
3. A log of customer purchases which represent removal of items from a store’s inventory.

* **Tables**: Records represent a current state that can be overwritten/updated.

1. A user’s current available credit card balance.
2. A list of conference attendee names with a value indicating whether or not they have checked in.
3. A set of data containing the quantity of each item in a store’s inventory.

| **Feature** | **KStream** | **KTable** |
| --- | --- | --- |
| Type | **Event Stream** | **Change Log Stream** |
| Data | Immutable events | Mutable state (latest value per key) |
| Joins | Windowed Joins | Table-table joins |
| Example | Clickstream data, logs | User profiles, account balances |

| **Feature** | **Kafka Consumer API** | **Kafka Streams** |
| --- | --- | --- |
| Purpose | Consumes data from topics | Processes data in real time |
| State Management | No state management | Supports **stateful** and **stateless** processing |
| Scalability | Manual partition management | Automatic scaling based on partitions |
| Processing | Only consumption | Supports **transformations, filtering, joins, windowing** |

**Exactly-Once semantics**

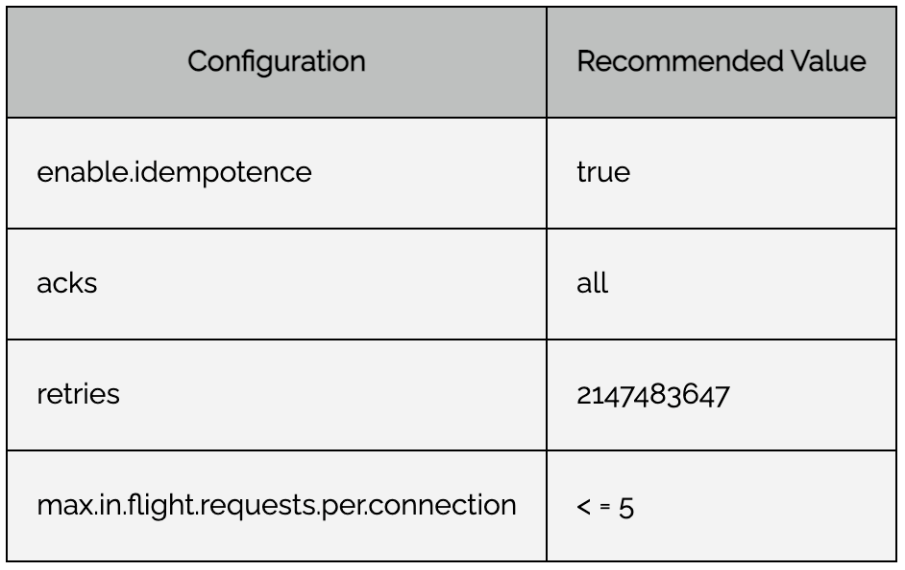
**Idempotent: Idempotent means that you can repeat the request and the result should be the same**

To achieve Exactly-Once semantic in Kafka, it uses below 3 property

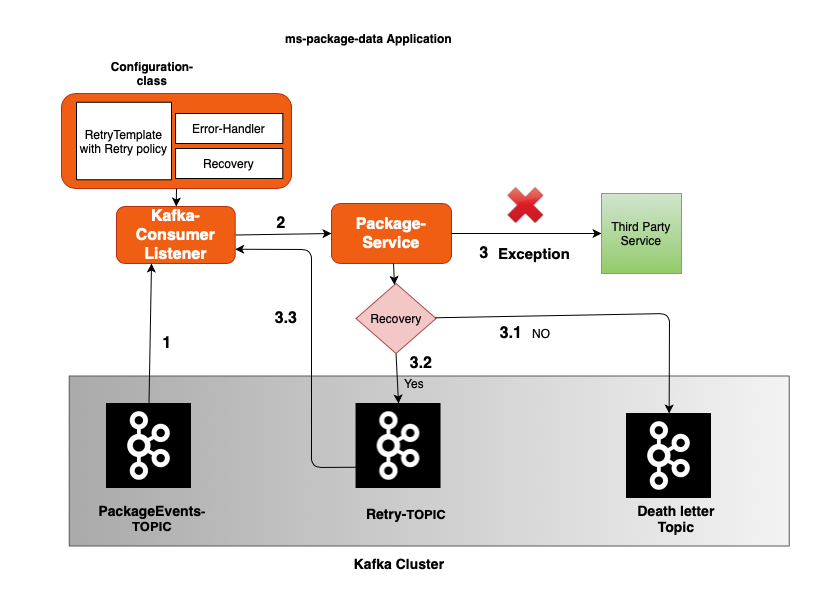
1. enable.idempotence=true (address a, b & c)
2. MAX\_IN\_FLIGHT\_REQUESTS\_PER\_CONNECTION=5(Producer will always have one in-flight request per connection)
3. isolation.level=read\_committed

The **max.in.flight.requests.per.connection** setting can be used to increase throughput by allowing the client to send multiple unacknowledged requests before blocking. However if the Producer is not Idempotent, max inflight requests is greater than 1, there is the risk of message re-ordering occurring when retrying due to transient errors. With the Idempotent Producer then up to 5 in-flight requests can be configured with message ordering guaranteed by Kafka.

For an Idempotent Producer, message ordering is guaranteed during retries by Kafka so long as the **max.in.flight.requests.per.connection** is not configured to be greater than 5. This configuration setting can be used to increase throughput by allowing the client to send multiple unacknowledged requests before blocking.



**Kafka Consumer Error Handling:**



## Kafka Retry:

* In return, RetryTemplate is set with Retry policy which specifies the maximum attempts you want to retry and what are the exceptions you want to retry and what are not to be retried.
* When the event is failed, even after retrying certain exceptions for the max number of retries, the recovery phase kicks in.

## **Kafka Recovery:**

* There is a handy method setRecoveryCallBack() on ConcurrentKafkaListenerContainerFactory where it accepts the Retry context parameter.

**Kafka Error handling:**

* For any exception in the process of the consumed event, an error is logged by Kafka “LoggingErrorHandler.class” in org.springframework.kafka.listener package,
* We can implement our own Error Handler by implementing the “ErrorHandler” interface.

package com.pack.events.consumer.config;

import lombok.extern.slf4j.Slf4j;

import org.springframework.beans.factory.ObjectProvider;

import org.springframework.boot.autoconfigure.kafka.ConcurrentKafkaListenerContainerFactoryConfigurer;

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

import org.springframework.dao.RecoverableDataAccessException;

import org.springframework.kafka.config.ConcurrentKafkaListenerContainerFactory;

import org.springframework.kafka.core.ConsumerFactory;

import org.springframework.retry.policy.SimpleRetryPolicy;

import org.springframework.retry.support.RetryTemplate;

import java.util.HashMap;

import java.util.Map;

import java.util.concurrent.TimeoutException;

@Configuration

@Slf4j

public class ConsumerConfig {

@Bean

**ConcurrentKafkaListenerContainerFactory**<?, ?> kafkaListenerContainerFactory(

ConcurrentKafkaListenerContainerFactoryConfigurer configurer,

ObjectProvider<ConsumerFactory<Object, Object>> kafkaConsumerFactory) {

ConcurrentKafkaListenerContainerFactory<Object, Object> factory = new ConcurrentKafkaListenerContainerFactory<>();

configurer.configure(factory, kafkaConsumerFactory);

factory.**setRetryTemplate**(retryTemplate());

factory.**setRecoveryCallback**((context -> {

if(context.getLastThrowable().getCause() instanceof RecoverableDataAccessException){

//here you can do your recovery mechanism where you can put back on to the topic using a Kafka producer

} else{

// here you can log things and throw some custom exception that Error handler will take care of ..

throw new RuntimeException(context.getLastThrowable().getMessage());

}

return null;

}));

factory.**setErrorHandler**(((exception, data) -> {

/\* here you can do you custom handling, I am just logging it same as default Error handler does

If you just want to log. you need not configure the error handler here. The default handler does it for you.

Generally, you will persist the failed records to DB for tracking the failed records. \*/

log.error("Error in process with Exception {} and the record is {}", exception, data);

}));

return factory;

}

private RetryTemplate retryTemplate() {

RetryTemplate retryTemplate = new RetryTemplate();

/\* here retry policy is used to set the number of attempts to retry and what exceptions you wanted to try and what you don't want to retry.\*/

retryTemplate.setRetryPolicy(getSimpleRetryPolicy());

return retryTemplate;

}

private SimpleRetryPolicy getSimpleRetryPolicy() {

Map<Class<? extends Throwable>, Boolean> exceptionMap = new HashMap<>();

exceptionMap.put(IllegalArgumentException.class, false);

exceptionMap.put(TimeoutException.class, true);

return new SimpleRetryPolicy(3,exceptionMap,true); } }

**When to use RabbitMQ**:

* RabbitMQ is ideal for simple use cases, you have certain advantages with low data traffic such as priority queue and flexible routing options.

**When to use Kafka**:

* If you need a commit log or multiple consumers, you can use Kafka for massive data and high throughput because RabbitMQ can’t help you with it.

**RabbitMQ vs Kafka**

1. Payload

**RabbitMQ**: It may have a large payload, for instance creating an order may have 45 different attributes.

**Apache Kafka**: Payload is very small in Kafka and its key-value pairs are sent across the stream.

3. Throughput

**RabbitMQ**: RabbitMQ gives a throughput of up to 4K -10K messages/sec.

**Apache Kafka**: With Kafka, one could get a throughput of up to 1 million messages/sec as Kafka leverages sequential disk I / O power and needs less hardware, which results in high throughput with only a small number of nodes.

4. Data Usage

**RabbitMQ**: It’s good for transactional data, what I meant by transactional data is user requests, order formation, placement of an order, these are forms of transactional items.

**Apache Kafka**: Kafka is really good for operational information, i.e. data are essentially about our process operations, i.e. statistics data for different kinds of auditing, logging. This kind of data tells about the safety and activities that are going on with the system.

5. Message Retention

**RabbitMQ**: RabbitMQ sends the message to the consumer and the message is removed from the queue once it has been processed and the acknowledgment has arrived.

**Apache Kafka**: Because Kafka is a log, there are always messages, you can monitor this by setting a retention policy for messages.

E.g. 7 days retention

6. Design

**RabbitMQ**: It uses a smart broker / dumb consumer model that focuses on consistently delivering messages to consumers that consume at approximately the same pace as the broker keeps track of consumer status.

**Apache Kafka**: Kafka uses a dumb broker / smart consumer. Kafka does not attempt to monitor the messages each user has read and retain only unread messages; rather, Kafka preserves all messages for a certain amount of time, and consumers are responsible for monitoring their position in each log (consumer state).

7. Topology

**RabbitMQ**: It has Exchange queue topology, where the producer sends a message to an exchange that is then routed to different queue bindings that consumers can consume.

**Apache Kafka**: Kafka only supports the publish-subscribe type of topology. This is where the producer sends a message across the stream to a topic within Kafka that is then consumed by a different consumer group

2. Data Flow

**RabbitMQ**: It has a distinct bounded flow of data, in other words, messages are created, sent, and received by the recipient of the consumer of the message.

**Apache Kafka**: It has an unbounded continuous flow of data, in other words, these key-value pairs are continuously streamed to the topic.

**RabbitMQ**

* **Message Ordering within a Queue**: RabbitMQ guarantees message order within a single queue. When a consumer consumes messages from that queue, they will be processed in the order they were received.
* **Queue Ordering**: RabbitMQ guarantees message ordering **within a single queue**.
* **Multiple Consumers**: If multiple consumers are consuming from a queue, RabbitMQ will distribute messages to consumers, which can break the order if you're processing in parallel. To preserve order, you should have only one consumer per queue or ensure that consumers process messages in the correct order.
* **Single Consumer per Queue:** To ensure message ordering is preserved, make sure that only one consumer is consuming messages from a queue at any given time.
* **Avoid Multiple Consumers on the Same Queue:** If you have multiple consumers on the same queue, RabbitMQ will round-robin messages to the consumers. This can break the order since messages may be processed out of sequence by different consumers. To avoid this, ensure that only one consumer is consuming messages from a queue.
* **Message Grouping**: You can group related messages with headers such as correlationId to preserve order within groups.

**Best Practices for Kafka Performance**

| **Category** | **Optimization** | **Benefits** |
| --- | --- | --- |
| **Partitions** | Increase partitions | Parallel processing |
| **Producer** | Use batch, compression, acks | Faster writes |
| **Consumer** | Increase concurrency, fetch size, async commit | Faster reads |
| **Broker** | Enable caching, tune retention | Lower disk usage, faster response |
| **Networking** | Increase buffer size, enable zero-copy | Reduce latency |
| **Streams** | Enable RocksDB, use multiple threads | Faster stream processing |
| **Scaling** | Add brokers, distribute topics | Higher throughput |

IOT Streams

This processor will:

1. **Group data by deviceId**.
2. **Apply a tumbling window of 10 seconds**.
3. **Compute the average temperature per window**.
4. **Write results to an output topic (iot-avg-temp)**.

@Component

public class IoTStreamWindowedProcessor {

@Bean

public KStream<String, IoTSensorData> processWindowedIoTStream(StreamsBuilder builder) {

// Define input topic

KStream<String, IoTSensorData> sensorStream = builder

.stream("iot-sensor-data",

Consumed.with(Serdes.String(), new JsonSerde<>(IoTSensorData.class)));

// Grouping by deviceId for windowed aggregation

KTable<Windowed<String>, Double> avgTempTable = sensorStream

.groupByKey()

.windowedBy(**TimeWindows.ofSizeWithNoGrace**(Duration.ofSeconds(10)))

.aggregate(

() -> 0.0, // Initial value

(key, value, aggregate) -> (aggregate + value.getTemperature()) / 2, // Compute avg

Materialized.with(Serdes.String(), Serdes.Double())

);

// Convert windowed key to string for output

avgTempTable.toStream()

.map((windowedKey, avgTemp) ->

KeyValue.pair(windowedKey.key(), avgTemp))

.to("iot-avg-temp", Produced.with(Serdes.String(), Serdes.Double()));

return sensorStream;

}

}

// Apply Sliding Window Aggregation

KTable<Windowed<String>, Double> avgTempTable = sensorStream

.groupByKey()

.windowedBy(

**SlidingWindows.ofTimeDifferenceWithNoGrace**(Duration.ofSeconds(10))

.advanceBy(Duration.ofSeconds(5)) // Sliding interval

)

.aggregate(

() -> 0.0, // Initial value

(key, value, aggregate) -> (aggregate + value.getTemperature()) / 2, // Compute avg

Materialized.with(Serdes.String(), Serdes.Double())

);

| **Feature** | **Tumbling Window** | **Sliding Window** |
| --- | --- | --- |

|  |  |  |
| --- | --- | --- |
| **Window Type** | Fixed intervals, non-overlapping | Overlapping windows |

|  |  |  |
| --- | --- | --- |
| **Window Size** | 10 seconds | 10 seconds |

|  |  |  |
| --- | --- | --- |
| **Window Step** | 10 seconds | 5 seconds |

|  |  |  |
| --- | --- | --- |
| **Granularity** | Coarse | More granular insights |

|  |  |  |
| --- | --- | --- |
| **Use Case** | Batch processing | Near real-time processing |