Apache Kafka

KStream – KStream Joins

KStream is an infinite, unbounded stream of records. we can join them with another KStream, KTable or with a GlobalKTable. However, if we try to join two infinite streams, sooner or later, we will end up consuming all our resources and crash our system. Hence KStream-KStream joins are always windowed joins, and non-windowed joins are not available.

When joining two streams, we must specify a boundary using a JoinWindows. Let's create a simple requirement scenario to get a sense of applicable use cases for stream-stream join and understand the mechanics of implementing such joins.

Problem – OTP Validation

Let's create a scenario of payment using a mobile wallet (mWallet). we want to pay bills for our Internet provider (iNet) that gives us an option to pay using our mWallet.

we log-in to the iNet application and select the mWallet option to initiate the bill payment. The iNet would ask for our linked mobile number.

Once we supply our cell number, the iNet would call the mWallet payment API. The mWallet payment API implement Apache Kafka based system, and for each payment request, it takes the following actions.

1. Generates an event and sends it to mWallet Kafka cluster. Figure below shows a sample **payment request event.**

```
100001 {"TransactionID": "100001", "CreatedTime":
1550149840000, "SourceAccountID": "131106", "TargetAccountID":
"15:837", "Amount": 1000, "OTP": 812960}
```

2. Generates an OTP and securely sends for transaction verification. This OTP is valid for five minutes.

Apache Kafka

3. Responds back to iNet with a transaction ID.

Now, the iNet would wait for us to enter the OTP. When we supply the OTP, the iNet again calls the mWallet with the TransactionID and OTP. This second call would generate a new event as shown as below:

```
100001 {"TransactionID": "100001", "CreatedTime":
1550150100000, "OTP": 812960}
```

Payment Confirmation Event

The second event is also pushed to Kafka. Now we need to write a streams application that joins these two streams and validates the OTP.

we can print the validation outcome on the console for this example. However, the actual implementation would want to sync the result into a key-value store where it can be queried by the mWallet for success and failure.

Solution - OTP Validation

Since the OTP is valid only for five minutes, this requirement makes an excellent use case for stream-stream join using a five-minute join window. Code below shows the first part of the implementation.

Apache Kafka

```
1. StreamsBuilder streamsBuilder = new StreamsBuilder();


2. KStream<String, PaymentRequest> KS0 = streams-
   Builder.stream(

      AppConfigs.paymentRequestTopicName,

      Consumed.with(AppSerdes.String(),

                AppSerdes.PaymentRequest())

          .withTimestampExtractor(AppTimeExtractor.Paymen-
   tRequest())

      );


3. KStream<String, PaymentConfirmation> KS1 = streams-
   Builder.stream(

      AppConfigs.paymentConfirmationTopicName,

      Consumed.with(AppSerdes.String(),

                AppSerdes.PaymentConfirmation())

          .withTimestampExtractor(AppTimeExtractor.Payment-
   Confirmation())
```

Line 1 -3

First three lines of code are straightforward, we created StreamBuilder and opened streams from two different Kafka topics. The first stream KS0 reads from a payment request topic where all initial payment requests are published. The second stream KS1 reads from the payment confirmation topic where the conformation events are posted.

```
StreamsBuilder streamsBuilder = new StreamsBuilder();
KStream<String, PaymentRequest> KS0 = streamsBuilder.stream(
    AppConfigs.paymentRequestTopicName,
    Consumed.with(AppSerdes.String(), AppSerdes.PaymentRequest())
        .withTimestampExtractor(AppTimeExtractor.PaymentRequest())
);

KStream<String, PaymentConfirmation> KS1 = streamsBuilder.stream(
    AppConfigs.paymentConfirmationTopicName,
    Consumed.with(AppSerdes.String(), AppSerdes.PaymentConfirmation())
        .withTimestampExtractor(AppTimeExtractor.PaymentConfirmation())
);
```

Apache Kafka

With these two streams created, we can join them and compare the OTP to authenticate the transaction. ***We created both the topics with the same number of partitions.***

```
kafka-topics.bat --bootstrap-server localhost:9092 --create --topic payment_request --partitions 5 --
replication-factor 3
```

```
kafka-topics.bat --bootstrap-server localhost:9092 --create --topic payment_confirmation --
partitions 5 --replication-factor 3
```

Both types of messages come with TransactionID as a key, and they use the same default partitioner.

Hence, we already satisfy the co-partitioning requirement for the join, and there is no need to repartition the data. We are ready to perform the join. Code below shows the join operation.

```
KStream<String, TransactionStatus> KS3 = KS0.join(KS1,

  (paymentRequest, paymentConfirmation) ->

    new TransactionStatus()

      .withTransactionID(paymentRequest.getTransac-
tionID())

      .withStatus(paymentRequest.getOTP()

        .equals(paymentConfirmation.getOTP()) ?"Suc-
cess" : "Failure"),

  JoinWindows.of(Duration.ofMinutes(5)),

  Joined.with(Appserdes.String(),

    AppSerdes.PaymentRequest(),

    Appserdes.PaymentConfirmation())

);
```

Line 4

We start with the first stream KS0 and join() it with KS1. The join() method takes three mandatory arguments, and the fourth argument is optional.

Apache Kafka

```
<VR, VR> KStream<K, VR> join(final KStream<K, VO> otherStream,
                             final ValueJoiner<? super V, ? super VO, ? extends VR> joiner,
                             final JoinWindows windows,
                             final Joined<K, V, VO> joined);
```

The first argument is obviously the other stream that we want to join. The second argument is a ValueJoiner lambda of the form (v1, v2) -> {} where v1 is a record from KS0 (left side) and the v2 is a record from KS1 (right side).

```
//implement streamsConfig.json file
KStream<String, TransactionStatus> KS3 = KS0.join(KS1,
    (paymentRequest, paymentConfirmation) ->
        new TransactionStatus()
            .withTransactionID(paymentRequest.getTransactionID())
            .withStatus(paymentRequest.getOTP()
                .equals(paymentConfirmation.getOTP()) ? "Success" : "Failure"),
    JoinWindows.of(Duration.ofMinutes(5)),
    Joined.with(AppSerdes.String(),
        AppSerdes.PaymentRequest(),
        AppSerdes.PaymentConfirmation())
);
```

The lambda method is triggered by the framework when a matching record is found. The body of the lambda method is where we would create an output record. The join logic and outcome in our example are straightforward.

We create a brand new TransactionStatus record as an outcome. Logic is as simple as we match the payment request OTP with the payment confirmation OTP and return success or failure.

The third argument is the JoinWindow that we use to define the duration constraint for the join operation. In the case of our example, if the paymentRequest and paymentConfirmation do not happen within five minutes, the join is not performed, and that's what we wanted. The last argument is to define the required Serdes. The first Serdes is for the common key among the two events. The second and the third Serdes are for the two events.

Let's test the application with some messages as shown in below:

1. 100001 {"TransactionID": "100001", "CreatedTime": 1550149860000, "SourceAccountID": "131100", "TargetAccountID": "151837", "Amount": 3000, "OTP": 852960}
2. 100002 {"TransactionID": "100002", "CreatedTime": 1550149920000, "SourceAccountID": "131200", "TargetAccountID": "151837", "Amount": 2000, "OTP": 931748}
3. 100003 {"TransactionID": "100003", "CreatedTime": 1550149980000, "SourceAccountID": "131300", "TargetAccountID": "151837", "Amount": 5000, "OTP": 591296}
4. 100004 {"TransactionID": "100004", "CreatedTime": 1550150100000, "SourceAccountID": "131400", "TargetAccountID": "151837", "Amount": 1000, "OTP": 283084}

Apache Kafka

- *Sample Payment Requests*

All the messages are timestamped is represented in UTC milliseconds. After sending these messages, the application does not show any outcome in the beginning because there are no matching confirmation records in the other topic yet. Let's throw some confirmation messages as shown in below:

1. 100001 {"TransactionID": "100001", "CreatedTime": 1582130100000, "OTP": 852960}
2. 100002 {"TransactionID": "100002", "CreatedTime": 1582130480000, "OTP": 931749}
3. 100004 {"TransactionID": "100001", "CreatedTime": 1582130040000, "OTP": 283046}

Sample Payment Confirmation

After sending these messages, our application should show the following outcome.

Transaction ID = 100001 Status = Success
Transaction ID = 100004 Status = Failure

Inner Join Output

We received four payment requests, but only two of them appeared in the outcome. The request 100001 was sent at 13:11 minute and confirmed at 13:15 minute with matching OTP. Hence, it comes as a success.

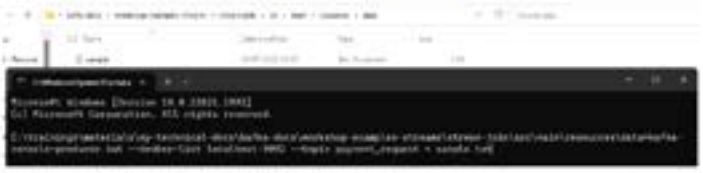
The request 100002 was sent at 13:12 but confirmed outside the five-minute window at 13:18. Hence, it is not considered for the Join and does not appear in the outcome.

The request 100003 was not confirmed, so it does not appear in the outcome. Finally, request 100004 is an odd one. It was generated at 13:15. However, somehow it was confirmed even earlier at 13:14. This record appears in the result because both the records fall in the five-minute window with a matching key. However, status is failed because their OTP didn't match.

Testing:

Step-1: start the console producer with the sample payment requests and payment confirmations as shown below:

Apache Kafka



```
kafka-console-producer.bat --broker-list localhost:9092 --topic payment_request < sample.txt
```

Sample.txt

```
100001:{"TransactionID": "100001", "CreatedTime":
1550149860000, "SourceAccountID": "131100",
"TargetAccountID": "151837", "Amount": 3000, "OTP":
852960}
100002:{"TransactionID": "100002", "CreatedTime":
1550149920000, "SourceAccountID": "131200",
```

```
"TargetAccountID": "151837", "Amount": 2000, "OTP":
931749}

100003:{"TransactionID": "100003", "CreatedTime":
1550149980000, "SourceAccountID": "131300",
"TargetAccountID": "151837", "Amount": 5000, "OTP":
591296}

100004:{"TransactionID": "100004", "CreatedTime":
1550150100000, "SourceAccountID": "131400",
"TargetAccountID": "151837", "Amount": 1000, "OTP":
283084}

100001:{"TransactionID": "100001", "CreatedTime":
1550150100000, "OTP": 052960}

100002:{"TransactionID": "100001", "CreatedTime":
1550150280000, "OTP": 931749}

100004:{"TransactionID": "100001", "CreatedTime":
```


Apache Kafka


```
1550150040000, "OTP": 283086}
```


Payment Request

Request ID	Created Time	OTP
100001	2019-02-14T13:13:00.00Z	052960
100002	2019-02-14T13:18:00.00Z	931749
100003	2019-02-14T13:13:00.00Z	591296
100004	2019-02-14T13:15:00.00Z	283084

Payment Confirmation

Request ID	Created Time	OTP
100001	2019-02-14T13:15:00.00Z	052960
100002	2019-02-14T13:18:00.00Z	931749
100001	2019-02-14T13:14:00.00Z	283086