Apache Kafka

What is State?

Problem: Lets Consider a Customer Rewards application which should read invoices from a Kafka topic, and after filtering for prime customers, we have to compute reward points earned by the customer on a given invoice and sent a notification to the customer in real-time.

However, the application should Calculate and notify the customers with the loyalty points that he earned on the most recent purchase is a good thing to do, but it would make more sense for the customers if we can include the past rewards and notify him with the total points earned so far including the rewards earned on the current purchase.

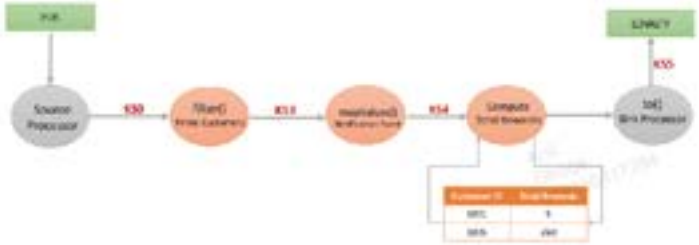
Solution:

We want to calculate loyalty points on each invoice, add it to the previously earned points by the customer and send him a notification with the total rewards accumulated up to the current bill. The following figure shows a topology for a possible solution approach.

The idea represented in the below figure is , We create a source processor to read a stream of invoices and a filter to receive only the prime customer invoices. Then we apply a mapValues() processor to transform the invoice into a notification object.

The mapValues() processor would also calculate the rewards earned on the current invoice and include it to the notification message. However, we want to add the current rewards to the previous points. We can do that by implementing a separate processor node which would query a table for the earlier total reward points by the customer.

Now the compute node would add the current value with the previous value and update the total in the notification message. We should also upgrade the new total points to the lookup table for the next use. Finally, we sink the notification in a different topic for the messaging service to consume and send an SMS.



Apache Kafka

In the above solution, the lookup table is the most critical component. The table maintains the current state of customer rewards. In a real-time streaming application, such tables with the data are termed as State. States are fundamental to an application. Almost every application maintains some state.

If we are coming from the database background, every time we talk about aggregation such as count(), avg(), sum(), and max(), we must maintain some states in between the events. Even if we think about joining two streams, we need to keep some state.

In the case of joins, the state would be the rows in each stream waiting to find a match in the other stream. Maintaining some states are often necessary and fundamental to building stream processing solution.

What is State Store?

A processor state could be as simple as a single key-value pair, or it could be a large lookup table or maybe a bunch of tables. Creating and maintaining state requires a state store. We can create an in-memory state store or may want to use a database or a file system to create a persistent state store. Whatever we chose to implement our state store, it must provide the following features:

1. Faster Performance
2. Fault tolerance

We might want to use a remote database such as PostgreSQL or MySQL to implement our state store. we may also want to use a distributed database such as HBase or Cassandra. However, remote databases are always subject to a performance bottleneck due to network delays. For A few transactions every second to hour, our remote databases could be faster.

However, a sizeable real-time application, processing millions of events would eventually start suffering from remote database performance problems.

Hence, Instead of using remote databases, Kafka Streams provide us to create fault-tolerant, in-memory state stores as well as local recoverable persistent state stores.

Kafka Streams persistent state stores are internally implemented using local Rocks DB instance. However, Streams API also allows us to create our own custom state store as well.

Stateful Vs Stateless

States are used by the processor in our topology. A processor may or may not need a state. We can classify processors in two categories depending upon whether a processor needs access to a state store.

1. Stateless Processors

Apache Kafka

2. Stateful Processors

Stateless transformations do not require state for processing, and they do not require a state store to be associated with them. some of the stateless processors such as mapValues(), filter(), flatMapValues(). Stateful transformations depend on the state for processing inputs and producing outputs require a state store.

All aggregating, joining and windowing operations in Kafka Streams are stateful operations, and hence they need access to a state store.

State Store Example

Creating Topology

Step-1: Like any other Kafka Stream application, we start by putting some basic configurations in a Java properties object.

Step-2: Then we create a stream builder instance and use it to open a source stream (KS0) to a Kafka topic.

Step-3: In the next operation, we apply a filter() processor on our source stream - KS0 and save the result in KS1.

Apache Kafka

```
1. Properties props = new Properties();
2. props.put(StreamsConfig.APPLICATION_ID_CONFIG,
   AppConfigs.applicationID);
3. props.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG,

   AppConfigs.bootstrapServers);
```

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

5. KStream KS0 = streamsBuilder.stream(AppConfigs.posTopic-
   Name,

      Consumed.with(PosSerdes.String(), PosSerdes.PosIn-
   voice()));


6. KStream KS1 = KS0.filter((Predicate<String, PosInvoice>)
   (key, value) ->

      value.getCustomerType()

         .equalsIgnoreCase(AppConfigs.CUSTOMER_TYPE-
   _PRIME));
```

Creating Store Builder

As per the topology defined in the above figure, the next step is to apply the transformValues() processor. The transformValues() processor is a stateful processor, and we want it to have access to a StateStore. So, the next logical step is to create a state store. However, instead of directly creating and instantiating a StateStore, we create a StoreBuilder.

```
StoreBuilder kvStoreBuilder = Stores.keyValueStoreBuilder(

   Stores.inMemoryKeyValueStore(AppConfigs.REWARDS_STORE_
NAME),

   Serdes.String(),

   Serdes.Double()

);

streamsBuilder.addStateStore(kvStoreBuilder);
```

Apache Kafka

The reason for creating a StoreBuilder is, This StoreBuilder is used by Kafka Streams as a factory to instantiate the actual state stores locally in application instances when and where needed. Instead, we create a StoreBuilder of our choice and add it to the StreamsBuilder, the StoreBuilder is internally manages to create StateStore.

Types of State Store

In the anbove example, we wanted to use a key/value state store because all we need to store is the customer id and her rewards. However, Kafka Streams 2.1 API offers three types of state stores and hence a corresponding store builder.

- KeyValueStore via Store.keyValueStoreBuilder()
- SessionStore via Store.sessionStoreBuilder()
- WindowStore via Store.windowStoreBuilder()

The KeyValueStore is available in the following variations.

- ✓ Stores.inMemoryKeyValueStore
- ✓ Stores.persistentKeyValueStore

In this example, we wanted to create an in-memory key/value state store. However, building a persistent key-value store is semantically same, and we can do that by just changing the first parameter of the Store.keyValueStoreBuilder() to Stores.persistentKeyValueStore.

Every state store is uniquely identified by a store name that we can set by passing a string name to Stores.inMemoryKeyValueStore(). These state stores are materialized, and hence we must provide a Serdes for the key and the value. So far, we created a store builder and added it to the streams builder. Now you are ready to extend the code shown and apply a transformValues() processor to KS1.

Creating Value Transformer

All the processors were accepting a lambda of one of the following types.

- ✓ (key, value) -> { }
- ✓ (value) -> { }

Accepting one of these lambda types is a typical case with many processors. However, the transformValues() is a bit advance type of processor. The transformValues () method takes an instance of a class that implements ValueTransformer interface

Apache Kafka

```
1. @SuppressWarnings("unchecked")
2. public class RewardsTransformer implements ValueTrans-
   former<PosInvoice, Notification> {
3.   private KeyValueStore<String, Double> stateStore;

4.   @Override
5.   public void init(ProcessorContext processorContext) {
6.     this.stateStore = (KeyValueStore) processorContext.getState-
     Store(
          AppConfigs.REWARDS_STORE_NAME );
7.   }

8.   @Override
9.   public Notification transform(PosInvoice invoice) {
10.    Notification notification = RecordBuilder.getNotification(in-
     voice);
11.    Double accumulatedRewards = stateStore.get(notifica-
     tion.getCustomerCardNo());
12.    Double totalRewards;

13.    if (accumulatedRewards != null)
14.      totalRewards = accumulatedRewards + notifica-
     tion.getEarnedLoyaltyPoints();
15.    else
16.      totalRewards = notification.getEarnedLoyaltyPoints();

17.    notification.setTotalLoyaltyPoints(totalRewards);
18.    stateStore.put(notification.getCustomerCardNo(),
            totalRewards);

19.    return notification;
20.  }

21.  @Override
22.  public void close() {
23.  }
24. }
```

The above code shows for a RewardsTransformer class that implements ValueTransformer<V, VR> interface. In this case, V is the input type, and the VR is the return type.

Apache Kafka

The RewardsTransformer is the place where we would define the actual logic to transform our PosInvoice into a Notification object.

```
@Override
public Notification transform(PosInvoice posInvoice) {
    Notification notification = new Notification()
        .withInvoiceNumber(posInvoice.getInvoiceNumber())
        .withCustomerCardNo(posInvoice.getCustomerCardNo())
        .withTotalAmount(posInvoice.getTotalAmount())
        .withEarnedLoyaltyPoints(posInvoice.getTotalAmount() * AppConfigs.LOYALTY_FACTOR)
        .withTotalLoyaltyPoints(0.0);
    Double accumulatedRewards = stateStore.get(notification.getCustomerCardNo());
    Double totalRewards;
    if (accumulatedRewards != null)
        totalRewards = accumulatedRewards + notification.getEarnedLoyaltyPoints();
    else
        totalRewards = notification.getEarnedLoyaltyPoints();
    notification.setTotalLoyaltyPoints(totalRewards);
    stateStore.put(notification.getCustomerCardNo(), totalRewards);

    return notification;
}
```

Creating RewardsTransformer is as simple as implementing ValueTransformer interface and overriding three methods.

The init() method is given a ProcessorContext and called only once. The processor context gives us access to a whole bunch of information, we use the processor context to get an instance of a state

store and initialize the reference to a private member for future usage.

```
private KeyValueStore<String, Double> stateStore;

@Override
public void init(ProcessorContext context) {
    this.stateStore = (KeyValueStore<String, Double>) context.getStateStore(AppConfigs.REWARDS_STORE_NAME);
}
```

At runtime, the StreamBuilder would create a state store using the StoreBuilder that we supplied. We can access that state store instance from the processor context, and that is what we are doing in the init() method.

```
StreamsBuilder builder = new StreamsBuilder();

KStream<String, PosInvoice> KS0 = builder.stream(
    AppConfigs.posTopicName,
    Consumed.with(AppSerdes.String(), AppSerdes.PosInvoice()))
.filter((key, value) ->
    value.getCustomerType().equalsIgnoreCase(AppConfigs.CUSTOMER_TYPE_PRIME));

StoreBuilder keyValueStoreBuilder = Stores.persistentKeyValueStore(
    AppConfigs.REWARDS_STORE_NAME),
    AppSerdes.String(), AppSerdes.Double()
);
builder.addStateStore(keyValueStoreBuilder);
```

The transform() method is called once for each Kafka message in the stream. This is the place where we implement our business logic. We have a straightforward business requirement.

Apache Kafka

- ✓ Create a Notification object using the PosInvoice which is passed to the transform method.

```
KStream<String, PosInvoice> KS0 = builder.stream(AppConfigs.POS_TOPIC,
    Consumed.with(AppSerdes.String(), AppSerdes.PosInvoice()))
.transformValues(() -> new RewardsTransformer(), AppConfigs.REWARDS_STORE_NAME)
.to(AppConfigs.notificationTopic,
    Produced.with(AppSerdes.String(), AppSerdes.Notification()));
```

- ✓ Get the accumulated reward points from the state store using customer id as a key.

```
Double accumulatedRewards = stateStore.get(notification.getCustomerCardNo());
```

- ✓ Compute the sum of the earned reward points and accumulated reward points.

```
if (accumulatedRewards != null)
    totalRewards = accumulatedRewards + notification.getEarnedLoyaltyPoints();
else
    totalRewards = notification.getEarnedLoyaltyPoints();
```

- ✓ Set the total reward points in the Notification object and update the state store for the future reference.

```
stateStore.put(notification.getCustomerCardNo(), totalRewards);
notification.setTotalLoyaltyPoints(totalRewards);
```

- ✓ Finally, return the notification back to the Kafka stream.

```
@Override
public Notification transform(PosInvoice posInvoice) {
    Notification notification = new Notification()
        .withInvoiceNumber(posInvoice.getInvoiceNumber())
        .withCustomerCardNo(posInvoice.getCustomerCardNo())
        .withTotalAmount(posInvoice.getTotalAmount())
        .withEarnedLoyaltyPoints(posInvoice.getTotalAmount() * AppConfigs.LOYALTY_FACTOR)
        .withTotalLoyaltyPoints(0.0);

    Double accumulatedRewards = stateStore.get(notification.getCustomerCardNo());

    Double totalRewards;
    if (accumulatedRewards != null)
        totalRewards = accumulatedRewards + notification.getEarnedLoyaltyPoints();
    else
        totalRewards = notification.getEarnedLoyaltyPoints();

    stateStore.put(notification.getCustomerCardNo(), totalRewards);
    notification.withTotalLoyaltyPoints(totalRewards);

    return notification;
}
```

That' all. The KeyValueStore is like a table where the key is the primary key of the table. We can query the table using the get() API and update the table using the put() API. Rest of the code is simple.

Apache Kafka

Kafka Streams Syntax Patterns

After defining our RewardsTransformer class, all we need is to supply it to the transformValues() processor. While learning Kafka Streams API, we will often notice this pattern where we create a custom class for implementing our business logic and supply an instance of the class to a Streams API. The creators of the Streams API wanted to offer us Java 8 lambda syntax to provide the instances of such custom classes, and hence, they implemented supplier classes to solve that problem.

So, to supply a ValueTransformer, they came up with a ValueTransformerSupplier class. Using Java 7 syntax, we can handover our RewardsTransformer as shown in below:

```
KStream KS3 = KS2.transformValues(
  new ValueTransformerSupplier() {
    @Override
    public ValueTransformer get() {
      return new RewardsTransformer();
    }
  },
  AppConfigs.REWARDS_STORE_NAME);
```

Java 7 syntax for supplying ValueTransformer reveals us the internal details of how providing custom classes is implemented by the Streams API. However, this design also allows us to leverage more convenient Java 8 lambda syntax because the ValueTransformerSupplier is a single method class. The lambda syntax for Java 8 is shown below:

```
KStream KS3 = KS2.transformValues(
  () -> new RewardsTransformer(),
  AppConfigs.REWARDS_STORE_NAME);
```

While learning Kafka Streams API, we will often notice this pattern where the Streams API offers us a convenient single method class to supply our business logic using Java 7 syntax, and at the same time, they also provide a convenient Java 8 lambda syntax.

Apache Kafka

For example, the mapValues() method that we use, offers a single method class ValueMapper<V, VR> as an argument type to mapValues(). If we look at the Javadoc for the KStream#mapValues(), we should get content like below:

mapValues(ValueMapper<? super V,? extends VR> mapper)
Transform the value of each input record into a new value (with possible new type) of the output record.

The mapValues() semantics shown in above allows us to use Java 7 as well as Java 8 lambda syntax as shown in below code:

```
1. //Java 7 Syntax for mapValues
   KStream<String, Notification> KS4 = KS2.mapValues(
     new ValueMapper<PosInvoice,Notification>() {
       @Override
       public Notification apply(PosInvoice invoice) {
         return RecordBuilder.getNotification(invoice);
       }
     }
   );

2. //Java 8 lambda Syntax for mapValues
   KStream<String, Notification> KS4 = KS2.mapValues(
     invoice -> RecordBuilder.getNotification(invoice)
   );

3. //Java 8 method reference Syntax for mapValues
```

```
KStream<String, Notification> K34 = K32.mapValues(
  RecordBuilder::getNotification
);
```

Apache Kafka

Starting Stream

Rest of the example code is straightforward. The below code shows the remaining application code.

```
K3.to(AppConfigs.notificationTopic,
    Produced.with(Serdes.String(), RecordSerdes.Notification()));

logger.info("Starting Kafka Streams");

KafkaStreams myStream = new KafkaStreams(streams-
Builder.build(), props);

myStream.start();
```

The K3 contains transformed Notification that also includes the total rewards for the customer. We sink the Notification to a Kafka topic. Finally, we create a KafkaStreams using the topology returned by the StreamBuilder#build() and start the stream.