Apache Kafka

Aggregates and Tables

Computing aggregates in real-time over a stream of data is a complex problem to solve. Hence, before we go to understand KTable and aggregation, let's create a simple example to understand the realtime usecase.

Problem – Streaming Wordcount

Streaming word count is an excellent example that can help us to understand some core basics of KTable and aggregation. The requirement is simple as such:

Step-1: create a Kafka topic and use command line producer to push some text to the topic. Let's assume we created a topic and sent the two messages as shown in below figure:

```
Hello World
Hello Kafka Streams
```

Step-2:

On the other side, we will create a Kafka Streams application to consume data from the topic in real time. The streams application should break the text message into words and count the frequency of unique words. Finally, it should print the outcome to the console.

For the above two messages, we expect the result as shown in below.

Key	Value
hello	2
world	1
kafka	1
streams	1

However, since it is a stream processing application, we must continuously monitor the topic for new incoming messages and revise the outcome.

Apache Kafka

Solution–Streaming Wordcount

```
1. public static void main(final String[] args) {

2.   final Properties props = new Properties();
3.   props.put(StreamsConfig.APPLICATION_ID_CONFIG, "Streaming-
     WordCount");
4.   props.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG,
     "localhost:9092");
5.   props.put(StreamsConfig.STATE_DIR_CONFIG, "state-store");
6.   props.put(StreamsConfig.DEFAULT_KEY_SERDE_CLASS_CON-
     FIG,
          Serdes.String().getClass().getName());
7.   props.put(StreamsConfig.DEFAULT_VALUE_SERDE_CLASS_CON-
     FIG,
          Serdes.String().getClass().getName());

8.   logger.info("Start Reading Messages");

9.   StreamsBuilder streamBuilder = new StreamsBuilder();
10.  KStream<String, String> KS0 = streamBuilder.stream("stream-
     ing-word-count");

11.  KStream<String, String> KS1 = KS0.flatMapValues(
          value -> Arrays.asList(value.toLowerCase().split(" "))
          );

12.  KGroupedStream<String, String> KGS2 = KS1.groupBy((key,
     value) -> value);
```

```
13. KTable<String, Long> KTS3 = KGS2.count();

14. KTS3.toStream().peek(
        (k, v) -> logger.info("key = " + k + " value = " + v.toString())
    );

15. KafkaStreams streams = new KafkaStreams(stream-
    Builder.build(), props);
16. streams.start();
17. Runtime.getRuntime().addShutdownHook(new Thread-
    (streams::close));
18. }
```

Configuration (Line 2 – 7) :

Apache Kafka

Like any other Kafka Streams application, we start StreamingWordCount by setting up a Java Properties object. Same as other examples, we put APPLICATION_ID_CONFIG and BOOTSTRAP_SERVERS_CONFIG. Then we put the default key/value Serdes.

There is one interesting thing to notice in this example. We are going to send text messages using console producer utility. By default, the console producer sends a message with a null message key. The message value would be a text string. However, they will be sent without a message key.

This example also defines a **STATE_DIR_CONFIG** for an apparent reason. however, we are going to create a table (KTable in this case) to store the word count. *Creating a KTable would internally create a local state store.* We do not want Kafka to generate the state store at a default location. Hence, we specify a root directory for the Rocks DB state store.

Creating Source Stream (Line 9 – 11):

Next step is to create a StreamBuilder and then create a source KStream (KS0). Then we use KS0 to create another KStream (KS1) by applying the flatMapValues() method.

The flatMapValues() will produce a stream of one or more values. In this case, it will split the text into individual words and result in a stream of words.

```
KStream<String, String> ks1 = ks0.flatMapValues(value->Arrays.asList(value.split(" ")));
```

Now that we have broken our stream of text (KS0) into the stream of words (KS1), we are ready to aggregate the new stream. But before that, we have to notice the following observations on KS1.

1. The key for every message in KS1 would be null.

2. Value for every message in KS1 is a word.

The KS1 is not a table but a stream. However, we can visualize the KS1 as a table below:

Key	Value
null	hello
null	world
null	hello
null	kafka
Null	streams

If we are asked to compute the word count on the above table, we can easily do it using the following SQL.

```
SELECT count(value)
FROM KS1
GROUP BY value
```

Apache Kafka

```
22 •  select count(*) count, words from word_count group by words;
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

count	words

2	hello
1	world
1	kafka
1	stream

Computing aggregates on a table using SQL looks easy.

Computing Aggregation:

Computing Aggregation is a two-step process.

1. Group By

2. Aggregate

The first step is to group our data by specifying a Group By clause. We have done that in the last line of the SQL shown below.

```
SELECT count(value)
FROM KS1
GROUP BY value
```

```
22 •   select count(*) count, words from word_count group by words;
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

count	words
2	hello
1	world
1	kafka
1	stream

Once the data is grouped, we apply an aggregation formula. In our case, the aggregation aggregation formula is count(), and we used it in the first line of the SQL.

Kafka Streams also take a similar two-step approach. Before we can compute an aggregate, we must group our stream. However, there is one limitation. The Kafka Streams can be grouped only on a key. This limitation does not constraint us in any other way except that we must create an appropriate key for our grouping requirement.

Apache Kafka

However, in our example, the message key is null, and the message value is a word. We want to group our data on words such as hello and kafka. So, all we need to do is to copy the message value into the key. How to do it?

There are many ways. However, when we are creating a key for grouping, we recommend using the groupBy() method.

Grouping KStream (Line 12 - 13)

Grouping is always performed by the key. If our stream already comes with the desired key, we can simply group our KStream using groupByKey().

The groupByKey() method does not expect any argument. However, we may need to provide a Serdes because grouping may require automatic repartition.

Alternatively, we can use groupBy() method when our stream does not have a key, or we want to change the key for the grouping operation. Since our messages in this example are without a key, so we are using the groupBy() method.

Using groupBy() is straightforward. The groupBy() method takes a (key,value) -> {} lambda. we can implement our business logic in the lambda body and return a new key for grouping our stream.

Line 12 in the above code is simply returns the value to become the key in the resulting stream.

```
KGroupedStream<String, String> kgs2 = ks1.groupBy((k,words)->words);
KTable<String, Long> kst3 = kgs2.count();
```

The groupBy() method creates a KGroupedStream KGS2. we can't do anything on a KGroupedStream except applying an aggregation formula. In our example, the formula is simply KGS2.count().

StateStore:

The outcome of an aggregation formula is always a KTable. In our example, KTS3 is a Kafka table backed by a local state store. We can visualize the KTS3 as shown in above example.

Key	Value
hello	2
world	1
kafka	1
streams	1

Word Count KTable

Apache Kafka

This table is stored in a local state store that is automatically created behind the scene. We might be wondering, why do we need to keep this table in a state store as we have already computed the word count and printed it on the console. That's all we wanted to do. Why keep it in the state store.

The answer is straightforward. We wanted to create a streaming word count. The above table represents the state of the world count so far. We need to update the state when new records are received. Now assume that we submitted two more messages as shown below:

Real-time streams
Streams Programming

Sample Messages

All the unique words in these two massages should also get incorporated in our word count state store. Let's include them in the table shown above, and the new state should be like the one shown below:

Key	Value
hello	2
world	1
kafka	1
streams	3
Real-time	1
Programming	1

There were two new unique words (real-time and programming), and one old word (streams) appeared twice. Hence, the count for the streams should be incremented by two, and both the new words should be added to the state with a count value as one.

What if we send a few more new messages? The state would keep getting updates for the new and old words appearing in the message. It is evident that we cannot do such updates unless we maintain the current values in a state store. That's why the state stores and the ability to create and manage tables are an integral part of any stream processing framework.

Now since we have seen KTable and Aggregates in action, it is time to formalize following:

- ✓ Kafka Streams KTable
- ✓ Computing Aggregates

KTable

A KTable is an abstraction of an update stream or often referred to as a change-log stream. We can visualize it as a table with a primary key. Each data record in a KStream is an UPSERT. ***More precisely, every record flowing to KTable is an UPDATE of the last value of the same key.*** If the

Apache Kafka

record for the same key does not exist yet, the update will be considered as an INSERT. Hence the record for the same key is overwritten by the new record.

Also, null values are interpreted especially. A new record with a null value represents a DELETE for the record's key. This also means that we can't have a KTable without a key. ***Having a key is essential for the KTable to work***, just as we cannot update a record in a database table without having the key.

KTable can be transformed using KTable transformation methods. Most of the KStream transformation methods that we have know, are also available for KTable.

- ✓ We can apply filter(), mapValues(), transformValues() and groupBy() transformation on KTable.
- ✓ We can convert a KTable to a KStream using the toStream() method.
- ✓ We can join KTable with another KTable or with a KStream.

Creating KTable

We can create a KTable using following methods.

1. KTable from a single Kafka Topic

2. Transformation on another KTable

3. Aggregation on a KStream

We have already seen an example to create a KTable using an aggregation on KStream in above code, Let's create a simple example to demonstrate the first two methods of producing a KTable.

Problem – Streaming Table:

We want to create a simple Kafka streams application that subscribes to a Kafka topic and creates an update stream, i.e. KTable. Further, we want to apply a filter() transformation on the original KTable to result in a new KTable. Finally, we want to convert the last KTable to a KStream and print the contents to the console.

This problem statement would create a super simple Streams application. However, we need something like this to demonstrate some critical KTable concepts.

Solution – Streaming Table:

Apache Kafka

```
1. public static void main(final String[] args) {


2.   final Properties props = new Properties();
3.   props.put(StreamsConfig.APPLICATION_ID_CONFIG, "Stream-
     ingTable");
4.   props.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG,
     "localhost:9092");
5.   props.put(StreamsConfig.STATE_DIR_CONFIG, "state-store");
6.   props.put(StreamsConfig.DEFAULT_KEY_SERDE_CLASS_CON-
     FIG,

         Serdes.String().getClass().getName());
7.   props.put(StreamsConfig.DEFAULT_VALUE_SERDE_CLASS_CON-
     FIG,

         Serdes.String().getClass().getName());
8.   props.put(StreamsConfig.DEFAULT_TIMESTAMP_EXTRACTOR_
     CLASS_CONFIG,

         WallclockTimestampExtractor.class.getName());


9.   StreamsBuilder streamBuilder = new StreamsBuilder();
10.  KTable<String, String> KT0 = streamBuilder.table("stock-
     tick");


11.  KTable<String, String> KT1 = KT0.filter((key, value) -> key.con-
     tains("HDFCBANK"));


12.  KStream<String, String> KS2 = KT1.toStream();
13.  KS2.peek((k, v) -> System.out.println("Key = " + k + " Value = " +
     v));


14.  KafkaStreams streams = new KafkaStreams(stream-
     Builder.build(), props);
15.  streams.start();
16.  Runtime.getRuntime().addShutdownHook(new Thread-
     (streams::close));
17. }
```

This simple example covers creating a KTable from Kafka topic and creating a KTable by transforming another KTable. The example also shows converting a KTable to a KStream. Let's try to understand the example code.

Line 2 – 7:

Apache Kafka

setting up Kafka Streams configuration.

Line 9 – 10:

we create a StreamBuilder and get a KTable using the table() method on the StreamBuilder. The table() method is like the stream() method that we used in earlier examples.

The table() method takes a topic name and creates a KTable by subscribing the Kafka topic. We already know that the KTable is an update stream and the records with the same key will be overwritten.

So, at any point in time, the KTable KT0 would have only one record for each key.

Line 11

In the next step we apply a filter() method on the KT0 to remove all other records except those where the key is equal to HDFCBANK.

The filter method in KTable works differently compared to the KStream#filter(). Let's try to understand it. Follow steps defined below to test the above example.

Create a topic (streaming-key-value) using below command:

kafka-topics.bat --create --zookeeper localhost:2181 --replication-factor 1 --partitions 1 --topic stock-tick

Start a Kafka console producer using below command:

kafka-console-producer.bat --broker-list localhost:9092 --topic stock-tick --property parse.key=true --property key.separator=":"

Send following messages: to Kafka topic using the console producer: HDFCBANK:2120 TCS:1879

Start StreamingTable example application:

We should see the following output. Key = HDFCBANK Value = 2120 Key = TCS Value = null

The filter method passes the HDFCBANK from KT0 to KT1, and hence it shows up in the output with a correct key/value pair.

However, the filter method rejects the TCS, and hence a tombstone (delete) record is forwarded to KT1. This is to ensure that if there is a record for TCS, it should be deleted and hence KT1 shows up as a null value for TCS.

This is where the KTable#filter() behaviour is different from KStream#filter(). Instead of simply holding up the filtered record, it forwards a tombstone (delete) record. The above example gives us

Apache Kafka

a general idea of how to use a KTable. However, while testing this example, we might realize that the output takes a while to show up on the console. The delay is caused by caching of KTable records.