KafkaIO Dynamic Read

https://issues.apache.org/jira/browse/BEAM-11325
Boyuan Zhang
boyuanz@apache.org

Problem Summary

It has been demanded for a while that KafkaIO can deal with any topic/partition that is added/removed dynamically during pipeline execution time. When KafkaIO users want to repartition existing topics, they have to stop the current running pipeline, update the pipeline code and re-launch it because current KafkaIO implementation has to know the topics and partitions during pipeline construction time. It's a huge burden for Beam users to maintain such a process.

Thus, we want to have KafkaIO be able to read from topics/partitions that are added when the pipeline is executing. Also, KafkaIO should be able to stop reading from one certain partition when under following conditions:

- When the topic/partition is removed or
- When the pipeline author wants to stop reading from this topic/partition

Proposed Solution

For the ability of reading from new added topics/partition, given that we already have ReadFromKafkaDoFn which is an Splittable DoFn taking any KafkaSourceDescriptor as input elements, all we need to do is to have a *WatchKafkaTopicPartitionDoFn* that is able to watch for growth of topics/partitions and emit corresponding KafkaSourceDescriptor to ReadFromKafkaDoFn. We are going to expose this ability from KafkaIO.Read API and users who are interested in this ability need to change the pipeline code. For Dataflow specific, this change is not update-compatible.

For supporting stopping read from topic/partitions, more changes are required in existing ReadFromKafkaDoFn:

- For deleted topic/partitions, the processElement method needs to return ProcessContinuation.stop() when the current partition is no longer in the current topic. Such information can be retrieved by KafkaConsumer.partitionFor(topic) API.
- For any topic/partitions, the *ReadFromKafkaDoFn* and *WatchKafkaTopicPartitionDoFn* will allow the pipeline author to register a stopRead(TopicPartition topicPartition) callback. When it's the time for *WatchKafkaTopicPartitionDoFn* to update, it should call this callback to determine whether one topic/partition has been stopped. Before the ReadFromKafkaDoFn starts processing TopicPartition, it will call provided stopRead(TopicPartition current) to decide whether to stop reading.

Here are supported scenarios with this proposed solution for certain given bootstrap_server strings:

- Certain topic/partition is added/deleted.
- Certain topic/partition is added, then removed but added again.
- Certain topic/partition is stopped if the stopRead(TopicPartition) is provided
- Certain topic/partition is added, then stopped but added again is stopRead(TopicPartition) is provided. It may involve race conditions. Please refer to race condition discussion.

We are going to focus on the discussion of WatchKafkaTopicPartitionDoFn since the removal support in ReadFromKafkaDoFn is very straightforward.

WatchKafkaTopicPartitionDoFn Design

There are three key points of this DoFn:

- to track current topics/partitions that have been processed by downstream ReadFromKafkaDoFn and
- to regularly query the set of current available topics/partitions and
- to emit any new added TopicPartitions

Thus, we are going to use BagState to persist existing TopicPartitions and use processing-time Timer to do the update regularly.

The major portion of the DoFn looks like:

```
// The input here is the dummy input with empty key and value.
class WatchKafkaTopicPartitionDoFn extends DoFn<KV<byte[], byte[]>,
KafkaSourceDescriptor> {

  private final String timerId = "watch_timer";
  final String bagStateId = "topic_partition_set";
  // The duration is specified by users.
  private long checkMills;

  // The provided function to check whether current TopicPartition should be
  //stopped.
  private SerializableFunction<TopicPartition, Boolean> stopReadFn;

  @TimerId(timerId)
  private final TimerSpec spec = TimerSpecs.timer(TimeDomain.PROCESSING_TIME);

  @StateId(bagStateId)
  private final StateSpec<BagState<TopicPartition>> bagStateSpec =
StateSpecs.bag();
```

```
  private Set<TopicPartition> getAllTopicPartitions() {
    Set<TopicPartition> current = new HashSet<>();
    try (Consumer<byte[], byte[]> kafkaConsumer =
consumerFactoryFn.apply(offsetConsumerConfig)) {
      for (Map.Entry<String, List<PartitionInfo>> topicInfo :
          kafkaConsumer.listTopics().entrySet()) {
        for (PartitionInfo partition : topicInfo.getValue()) {
          current.add(new TopicPartition(topicInfo.getKey(),
partition.partition()));
        }
      }
    }
    return current;
  }

  @ProcessElement
  public void processElement(
      @TimerId("watch_timer") Timer timer,
      @StateId(bagStateId) BagState<TopicPartition> existingTopicPartitions,
      OutputReceiver<KafkaSourceDescriptor> outputReceiver) {
    // For the first time, we emit all available TopicPartition and write them into
State.
    Set<TopicPartition> current = getAllTopicPartitions();
    current.forEach(
        topicPartition -> {
          existingTopicPartitions.add(topicPartition);
          outputReceiver.output(
              KafkaSourceDescriptor.create(
                  topicPartition, startOffset, startReadTime,
ImmutableList.of(bootstrapServer)));
        });

    timer.set(Instant.now().plus(checkMills));
  }

  @OnTimer(timerId)
  public void onTimer(
      @StateId(bagStateId) BagState<TopicPartition> existingTopicPartitions,
      OutputReceiver<KafkaSourceDescriptor> outputReceiver) {
    // It's the time to check whether there is any update.
    Set<TopicPartition> lastUpdated = new HashSet<>();
    existingTopicPartitions.read().forEach(e -> lastUpdated.add(e));
    existingTopicPartitions.clear();

    Set<TopicPartition> currentAll = getAllTopicPartitions();

    // Emit new added TopicPartitions.
```

```
    Set<TopicPartition> newAdded = Sets.difference(currentAll, lastUpdated);
    newAdded.forEach(
        topicPartition -> {
          if (stopRead(topicPartition)) continue;
          outputReceiver.output(
              KafkaSourceDescriptor.create(
                  topicPartition, startOffset, startReadTime,
ImmutableList.of(bootstrapServer)));
        });

    // Update the State.
    currentAll.forEach(
        topicPartition -> {
          if (stopRead(topicPartition)) continue;
          existingTopicPartitions.add(topicPartition);
        });

    // Reset the timer.
    timer.set(Instant.now().plus(checkDuration.getMillis()));
  }
}
```

KafkaIO API Exposure and X-Lang Support

There are 3 more APIs we want KafkaIO.Read to expose to take advantage of WatchKafkaTopicPartitionDoFn:

- dynamicRead(Duration interval)
  Within it enabled, the KafkaIO.Read will read any available TopicPartition with the frequency of specific time interval.
- withTopicPattern(String topicPattern, Duration interval)
  Different from dynamicRead(), it will only keep watching topics that match the provided pattern.
- withStopReadFn(SerializableFunction)

We will also expose these attributes into ReadfromKafka in python SDK for x-lang usages.

Race Condition Discussion

Race conditions may happen under 2 supported cases:

- A TopicPartition is removed, then added back again
- A TopicPartition is stopped, then want to read it again

When race condition happens, it will result in the stopped/removed TopicPartition failing to be emitted to ReadFromKafkaDoFn again. Or ReadFromKafkaDoFn will output replicated records.

The major cause for such race condition is that both WatchKafkaTopicPartitionDoFn and ReadFromKafkaDoFn react to the signal from removed/stopped TopicPartition but we cannot guarantee that both DoFns perform related actions at the same time.

Here is one example for failing to emit new added TopicPartition:

- A WatchKafkaTopicPartitionDoFn is configured with updating the current tracking set every 1 hour.
- One TopicPartition A is tracked by the WatchKafkaTopicPartitionDoFn at 10:00AM and ReadFromKafkaDoFn starts to read from TopicPartition A immediately.
- At 10:30AM, the WatchKafkaTopicPartitionDoFn notices that the TopicPartition has been stopped/removed, so it stops reading from it and returns ProcessContinuation.stop().
- At 10:45 the pipeline author wants to read from TopicPartition A again.
- At 11:00AM when WatchKafkaTopicPartitionDoFn is invoked by firing timer, it doesn't know that TopicPartition A has been stopped/removed. All it knows is that TopicPartition A is still an active TopicPartition and it will not emit TopicPartition A again.

Another race condition example for producing duplicate records:

- At 10:00AM, ReadFromKafkaDoFn is processing TopicPartition A
- At 10:05AM, ReadFromKafkaDoFn starts to process other TopicPartitions(sdf-initiated checkpoint or runner-issued checkpoint happens)
- At 10:10AM, WatchKafkaTopicPartitionDoFn knows that TopicPartition A is stopped/removed
- At 10:15AM, WatchKafkaTopicPartitonDoFn knows that TopicPartition A is added again and emits TopicPartition A again
- At 10:20AM, ReadFromKafkaDoFn starts to process resumed TopicPartition A but at the same time ReadFromKafkaDoFn is also processing the new emitted TopicPartitionA.

The race condition is avoidable if the pipeline author configures the WatchKafkaTopicPartitionDoFn with reasonable timer duration and is careful about the remoral/stop-addiction cases.

Implementation

KafkaIO Removal/Stop Support: https://github.com/apache/beam/pull/13710
KafkaIO Dynamic Read Support: https://github.com/apache/beam/pull/13750