Spark Streaming + Kafka Integration Guide (Kafka broker version 0.10.0 or higher)

The Spark Streaming integration for Kafka 0.10 provides simple parallelism, 1:1 correspondence between Kafka partitions and Spark partitions, and access to offsets and metadata. However, because the newer integration uses the new Kafka consumer API instead of the simple API, there are notable differences in usage.

Linking

For Scala/Java applications using SBT/Maven project definitions, link your streaming application with the following artifact (see Linking section in the main programming guide for further information).

```
groupId = org.apache.spark
artifactId = spark-streaming-kafka-0-10_2.12
version = 3.3.0
```

Do not manually add dependencies on `org.apache.kafka` artifacts (e.g. `kafka-clients`). The `spark-streaming-kafka-0-10` artifact has the appropriate transitive dependencies already, and different versions may be incompatible in hard to diagnose ways.

Creating a Direct Stream

Note that the namespace for the import includes the version, org.apache.spark.streaming.kafka010

Scala | Java

```
import java.util.*;
import org.apache.spark.SparkConf;
import org.apache.spark.TaskContext;
import org.apache.spark.api.java.*;
import org.apache.spark.api.java.function.*;
import org.apache.spark.streaming.api.java.*;
import org.apache.spark.streaming.kafka010.*;
import org.apache.kafka.clients.consumer.ConsumerRecord;
import org.apache.kafka.common.TopicPartition;
import org.apache.kafka.common.serialization.StringDeserializer;
import scala.Tuple2;

Map<String, Object> kafkaParams = new HashMap<>();
kafkaParams.put("bootstrap.servers", "localhost:9092,anotherhost:9092");
kafkaParams.put("key.deserializer", StringDeserializer.class);
kafkaParams.put("value.deserializer", StringDeserializer.class);
kafkaParams.put("group.id", "use_a_separate_group_id_for_each_stream");
kafkaParams.put("auto.offset.reset", "latest");
kafkaParams.put("enable.auto.commit", false);

Collection<String> topics = Arrays.asList("topicA", "topicB");

JavaInputDStream<ConsumerRecord<String, String>> stream =
  KafkaUtils.createDirectStream(
    streamingContext,
    LocationStrategies.PreferConsistent(),
    ConsumerStrategies.<String, String>Subscribe(topics, kafkaParams)
  );

stream.mapToPair(record -> new Tuple2<>(record.key(), record.value()));
```

For possible kafkaParams, see Kafka consumer config docs. If your Spark batch duration is larger than the default Kafka heartbeat session timeout (30 seconds), increase heartbeat.interval.ms and session.timeout.ms appropriately. For batches larger than 5 minutes, this will require changing group.max.session.timeout.ms on the broker. Note that the example sets enable.auto.commit to false, for discussion see Storing Offsets below.

LocationStrategies

The new Kafka consumer API will pre-fetch messages into buffers. Therefore it is important for performance reasons that the Spark integration keep cached consumers on executors (rather than recreating them for each batch), and prefer to schedule partitions on the host locations that have the appropriate consumers.

In most cases, you should use `LocationStrategies.PreferConsistent` as shown above. This will distribute partitions evenly across available executors. If your executors are on the same hosts as your Kafka brokers, use `PreferBrokers`, which will prefer to schedule partitions on the Kafka leader for that partition. Finally, if you have a significant skew in load among partitions, use `PreferFixed`. This allows you to specify an explicit mapping of partitions to hosts (any unspecified partitions will use a consistent location).

The cache for consumers has a default maximum size of 64. If you expect to be handling more than (64 * number of executors) Kafka partitions, you can change this setting via `spark.streaming.kafka.consumer.cache.maxCapacity`.

If you would like to disable the caching for Kafka consumers, you can set `spark.streaming.kafka.consumer.cache.enabled` to `false`.

The cache is keyed by topicpartition and group.id, so use a **separate** `group.id` for each call to `createDirectStream`.

ConsumerStrategies

The new Kafka consumer API has a number of different ways to specify topics, some of which require considerable post-object-instantiation setup. `ConsumerStrategies` provides an abstraction that allows Spark to obtain properly configured consumers even after restart from checkpoint.

`ConsumerStrategies.Subscribe`, as shown above, allows you to subscribe to a fixed collection of topics. `SubscribePattern` allows you to use a regex to specify topics of interest. Note that unlike the 0.8 integration, using `Subscribe` or `SubscribePattern` should respond to adding partitions during a running stream. Finally, `Assign` allows you to specify a fixed collection of partitions. All three strategies have overloaded constructors that allow you to specify the starting offset for a particular partition.

If you have specific consumer setup needs that are not met by the options above, `ConsumerStrategy` is a public class that you can extend.

Creating an RDD

If you have a use case that is better suited to batch processing, you can create an RDD for a defined range of offsets.

Scala | Java

```
// Import dependencies and create kafka params as in Create Direct Stream above

OffsetRange[] offsetRanges = {
  // topic, partition, inclusive starting offset, exclusive ending offset
  OffsetRange.create("test", 0, 0, 100),
  OffsetRange.create("test", 1, 0, 100)
};

JavaRDD<ConsumerRecord<String, String>> rdd = KafkaUtils.createRDD(
  sparkContext,
  kafkaParams,
  offsetRanges,
  LocationStrategies.PreferConsistent()
);
```

Note that you cannot use `PreferBrokers`, because without the stream there is not a driver-side consumer to automatically look up broker metadata for you. Use `PreferFixed` with your own metadata lookups if necessary.

Obtaining Offsets

Scala | Java

```
stream.foreachRDD(rdd -> {
  OffsetRange[] offsetRanges = ((HasOffsetRanges) rdd.rdd()).offsetRanges();
  rdd.foreachPartition(consumerRecords -> {
    OffsetRange o = offsetRanges[TaskContext.get().partitionId()];
    System.out.println(
      o.topic() + " " + o.partition() + " " + o.fromOffset() + " " + o.untilOffset());
  });
});
```

Note that the typecast to `HasOffsetRanges` will only succeed if it is done in the first method called on the result of `createDirectStream`, not later down a chain of methods. Be aware that the one-to-one mapping between RDD partition and Kafka partition does not remain after any methods that shuffle or repartition, e.g. reduceByKey() or window().

Storing Offsets

Kafka delivery semantics in the case of failure depend on how and when offsets are stored. Spark output operations are at-least-once. So if you want the equivalent of exactly-once semantics, you must either store offsets after an idempotent output, or store offsets in an atomic transaction alongside output. With this integration, you have 3 options, in order of increasing reliability (and code complexity), for how to store offsets.

Checkpoints

If you enable Spark checkpointing, offsets will be stored in the checkpoint. This is easy to enable, but there are drawbacks. Your output operation must be idempotent, since you will get repeated outputs; transactions are not an option. Furthermore, you cannot recover from a checkpoint if your application code has changed. For planned upgrades, you can mitigate this by running the new code at the same time as the old code (since outputs need to be idempotent anyway, they should not clash). But for unplanned failures that require code changes, you will lose data unless you have another way to identify known good starting offsets.

Kafka itself

Kafka has an offset commit API that stores offsets in a special Kafka topic. By default, the new consumer will periodically auto-commit offsets. This is almost certainly not what you want, because messages successfully polled by the consumer may not yet have resulted in a Spark output operation, resulting in undefined semantics. This is why the stream example above sets "enable.auto.commit" to false. However, you can commit offsets to Kafka after you know your output has been stored, using the `commitAsync` API. The benefit as compared to checkpoints is that Kafka is a durable store regardless of changes to your application code. However, Kafka is not transactional, so your outputs must still be idempotent.

Scala | Java

```
stream.foreachRDD(rdd -> {
  OffsetRange[] offsetRanges = ((HasOffsetRanges) rdd.rdd()).offsetRanges();

  // some time later, after outputs have completed
  ((CanCommitOffsets) stream.inputDStream()).commitAsync(offsetRanges);
});
```

Your own data store

For data stores that support transactions, saving offsets in the same transaction as the results can keep the two in sync, even in failure situations. If you're careful about detecting repeated or skipped offset ranges, rolling back the transaction prevents duplicated or lost messages from affecting results. This gives the equivalent of exactly-once semantics. It is also possible to use this tactic even for outputs that result from aggregations, which are typically hard to make idempotent.

Scala | Java

```
// The details depend on your data store, but the general idea looks like this

// begin from the offsets committed to the database
Map<TopicPartition, Long> fromOffsets = new HashMap<>();
for (resultSet : selectOffsetsFromYourDatabase)
  fromOffsets.put(new TopicPartition(resultSet.string("topic"), resultSet.int("partition")), resultSet.long("offset"));
}

JavaInputDStream<ConsumerRecord<String, String>> stream = KafkaUtils.createDirectStream(
  streamingContext,
  LocationStrategies.PreferConsistent(),
  ConsumerStrategies.<String, String>Assign(fromOffsets.keySet(), kafkaParams, fromOffsets)
);

stream.foreachRDD(rdd -> {
  OffsetRange[] offsetRanges = ((HasOffsetRanges) rdd.rdd()).offsetRanges();

  Object results = yourCalculation(rdd);

  // begin your transaction

  // update results
  // update offsets where the end of existing offsets matches the beginning of this batch of offsets
  // assert that offsets were updated correctly

  // end your transaction
});
```

SSL / TLS

The new Kafka consumer supports SSL. To enable it, set kafkaParams appropriately before passing to `createDirectStream` / `createRDD`. Note that this only applies to communication between Spark and Kafka brokers; you are still responsible for separately securing Spark inter-node communication.

Scala | Java

```
Map<String, Object> kafkaParams = new HashMap<String, Object>();
// the usual params, make sure to change the port in bootstrap.servers if 9092 is not TLS
kafkaParams.put("security.protocol", "SSL");
kafkaParams.put("ssl.truststore.location", "/some-directory/kafka.client.truststore.jks");
kafkaParams.put("ssl.truststore.password", "test1234");
kafkaParams.put("ssl.keystore.location", "/some-directory/kafka.client.keystore.jks");
kafkaParams.put("ssl.keystore.password", "test1234");
kafkaParams.put("ssl.key.password", "test1234");
```

Deploying

As with any Spark applications, `spark-submit` is used to launch your application.

For Scala and Java applications, if you are using SBT or Maven for project management, then package `spark-streaming-kafka-0-10_2.12` and its dependencies into the application JAR. Make sure `spark-core_2.12` and `spark-streaming_2.12` are marked as `provided` dependencies as those are already present in a Spark installation. Then use `spark-submit` to launch your application (see Deploying section in the main programming guide).

Security

See Structured Streaming Security.

Additional Caveats

- Kafka native sink is not available so delegation token used only on consumer side.