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

Structured Streaming integration for Kafka 0.10 to read data from and write data to Kafka.

Linking

For Scala/Java applications using SBT/Maven project definitions, link your application with the following artifact:

```
groupId = org.apache.spark
artifactId = spark-sql-kafka-0-10_2.12
version = 3.0.1
```

Please note that to use the headers functionality, your Kafka client version should be version 0.11.0.0 or up.

For Python applications, you need to add this above library and its dependencies when deploying your application. See the Deploying subsection below.

For experimenting on `spark-shell`, you need to add this above library and its dependencies too when invoking `spark-shell`. Also, see the Deploying subsection below.

Reading Data from Kafka

Creating a Kafka Source for Streaming Queries

Scala Java Python

```
# Subscribe to 1 topic
df = spark \
  .readStream \
  .format("kafka") \
  .option("kafka.bootstrap.servers", "host1:port1,host2:port2") \
  .option("subscribe", "topic1") \
  .load()
df.selectExpr("CAST(key AS STRING)", "CAST(value AS STRING)")

# Subscribe to 1 topic, with headers
val df = spark \
  .readStream \
  .format("kafka") \
  .option("kafka.bootstrap.servers", "host1:port1,host2:port2") \
  .option("subscribe", "topic1") \
  .option("includeHeaders", "true") \
  .load()
df.selectExpr("CAST(key AS STRING)", "CAST(value AS STRING)", "headers")

# Subscribe to multiple topics
df = spark \
  .readStream \
  .format("kafka") \
  .option("kafka.bootstrap.servers", "host1:port1,host2:port2") \
  .option("subscribe", "topic1,topic2") \
  .load()
df.selectExpr("CAST(key AS STRING)", "CAST(value AS STRING)")

# Subscribe to a pattern
df = spark \
  .readStream \
  .format("kafka") \
  .option("kafka.bootstrap.servers", "host1:port1,host2:port2") \
  .option("subscribePattern", "topic.*") \
  .load()
df.selectExpr("CAST(key AS STRING)", "CAST(value AS STRING)")
```

Creating a Kafka Source for Batch Queries

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

Scala Java Python

```
# Subscribe to 1 topic defaults to the earliest and latest offsets
df = spark \
  .read \
  .format("kafka") \
  .option("kafka.bootstrap.servers", "host1:port1,host2:port2") \
  .option("subscribe", "topic1") \
  .load()
df.selectExpr("CAST(key AS STRING)", "CAST(value AS STRING)")

# Subscribe to multiple topics, specifying explicit Kafka offsets
df = spark \
  .read \
  .format("kafka") \
  .option("kafka.bootstrap.servers", "host1:port1,host2:port2") \
  .option("subscribe", "topic1,topic2") \
  .option("startingOffsets", """{"topic1":{"0":23,"1":-2},"topic2":{"0":-2}}""") \
  .option("endingOffsets", """{"topic1":{"0":50,"1":-1},"topic2":{"0":-1}}""") \
  .load()
df.selectExpr("CAST(key AS STRING)", "CAST(value AS STRING)")

# Subscribe to a pattern, at the earliest and latest offsets
df = spark \
  .read \
  .format("kafka") \
  .option("kafka.bootstrap.servers", "host1:port1,host2:port2") \
  .option("subscribePattern", "topic.*") \
  .option("startingOffsets", "earliest") \
  .option("endingOffsets", "latest") \
  .load()
df.selectExpr("CAST(key AS STRING)", "CAST(value AS STRING)")
```

Each row in the source has the following schema:

Column	Type
key	binary
value	binary
topic	string
partition	int
offset	long
timestamp	timestamp
timestampType	int
headers (optional)	array

The following options must be set for the Kafka source for both batch and streaming queries.

Option	value	meaning
assign	json string {"topicA":[0,1],"topicB":[2,4]}	Specific TopicPartitions to consume. Only one of "assign", "subscribe" or "subscribePattern" options can be specified for Kafka source.
subscribe	A comma-separated list of topics	The topic list to subscribe. Only one of "assign", "subscribe" or "subscribePattern" options can be specified for Kafka source.
subscribePattern	Java regex string	The pattern used to subscribe to topic(s). Only one of "assign, "subscribe" or "subscribePattern" options can be specified for Kafka source.
kafka.bootstrap.servers	A comma-separated list	The Kafka "bootstrap.servers" configuration.

	of host:port	

The following configurations are optional:

Option	value	default	query type	meaning
startingOffsetsByTimestamp	json string """ {"topicA":{"0": 1000, "1": 1000}, "topicB": {"0": 2000, "1": 2000}} """	none (the value of `startingOffsets` will apply)	streaming and batch	The start point of timestamp when a query is started, a json string specifying a starting timestamp for each TopicPartition. The returned offset for each partition is the earliest offset whose timestamp is greater than or equal to the given timestamp in the corresponding partition. If the matched offset doesn't exist, the query will fail immediately to prevent unintended read from such partition. (This is a kind of limitation as of now, and will be addressed in near future.) Spark simply passes the timestamp information to `KafkaConsumer.offsetsForTimes`, and doesn't interpret or reason about the value. For more details on `KafkaConsumer.offsetsForTimes`, please refer javadoc for details. Also the meaning of `timestamp` here can be vary according to Kafka configuration (`log.message.timestamp.type`): please refer Kafka documentation for further details. Note: This option requires Kafka 0.10.1.0 or higher. Note2: `startingOffsetsByTimestamp` takes precedence over `startingOffsets`. Note3: For streaming queries, this only applies when a new query is started, and that resuming will always pick up from where the query left off. Newly discovered partitions during a query will start at earliest.
startingOffsets	"earliest", "latest" (streaming only), or json string """ {"topicA": {"0":23,"1":-1},"topicB": {"0":-2}} """	"latest" for streaming, "earliest" for batch	streaming and batch	The start point when a query is started, either "earliest" which is from the earliest offsets, "latest" which is just from the latest offsets, or a json string specifying a starting offset for each TopicPartition. In the json, -2 as an offset can be used to refer to earliest, -1 to latest. Note: For batch queries, latest (either implicitly or by using -1 in json) is not allowed. For streaming queries, this only applies when a new query is started, and that resuming will always pick up from where the query left off. Newly discovered partitions during a query will start at earliest.
endingOffsetsByTimestamp	json string """ {"topicA":{"0": 1000, "1": 1000}, "topicB": {"0": 2000, "1": 2000}} """	latest	batch query	The end point when a batch query is ended, a json string specifying an ending timestamp for each TopicPartition. The returned offset for each partition is the earliest offset whose timestamp is greater than or equal to the given timestamp in the corresponding partition. If the matched offset doesn't exist, the offset will be set to latest. Spark simply passes the timestamp information to `KafkaConsumer.offsetsForTimes`, and doesn't interpret or reason about the value. For more details on `KafkaConsumer.offsetsForTimes`, please refer javadoc for details. Also the meaning of `timestamp` here can be vary according to Kafka configuration (`log.message.timestamp.type`): please refer Kafka documentation for further details.

				Note: This option requires Kafka 0.10.1.0 or higher. Note2: `endingOffsetsByTimestamp` takes precedence over `endingOffsets`.
endingOffsets	latest or json string {"topicA":{"0":23,"1":-1},"topicB":{"0":-1}}	latest	batch query	The end point when a batch query is ended, either "latest" which is just referred to the latest, or a json string specifying an ending offset for each TopicPartition. In the json, -1 as an offset can be used to refer to latest, and -2 (earliest) as an offset is not allowed.
failOnDataLoss	true or false	true	streaming and batch	Whether to fail the query when it's possible that data is lost (e.g., topics are deleted, or offsets are out of range). This may be a false alarm. You can disable it when it doesn't work as you expected.
kafkaConsumer.pollTimeoutMs	long	512	streaming and batch	The timeout in milliseconds to poll data from Kafka in executors.
fetchOffset.numRetries	int	3	streaming and batch	Number of times to retry before giving up fetching Kafka offsets.
fetchOffset.retryIntervalMs	long	10	streaming and batch	milliseconds to wait before retrying to fetch Kafka offsets
maxOffsetsPerTrigger	long	none	streaming and batch	Rate limit on maximum number of offsets processed per trigger interval. The specified total number of offsets will be proportionally split across topicPartitions of different volume.
minPartitions	int	none	streaming and batch	Desired minimum number of partitions to read from Kafka. By default, Spark has a 1-1 mapping of topicPartitions to Spark partitions consuming from Kafka. If you set this option to a value greater than your topicPartitions, Spark will divvy up large Kafka partitions to smaller pieces. Please note that this configuration is like a `hint`: the number of Spark tasks will be **approximately** `minPartitions`. It can be less or more depending on rounding errors or Kafka partitions that didn't receive any new data.
groupIdPrefix	string	spark-kafka-source	streaming and batch	Prefix of consumer group identifiers (`group.id`) that are generated by structured streaming queries. If "kafka.group.id" is set, this option will be ignored.
kafka.group.id	string	none	streaming and batch	The Kafka group id to use in Kafka consumer while reading from Kafka. Use this with caution. By default, each query generates a unique group id for reading data. This ensures that each Kafka source has its own consumer group that does not face interference from any other consumer, and therefore can read all of the partitions of its subscribed topics. In some scenarios (for example, Kafka group-based authorization), you may want to use a specific authorized group id to read data. You can optionally set the group id. However, do this with extreme caution as it can cause unexpected behavior. Concurrently running queries (both, batch and streaming) or sources with the same group id are likely interfere with each other causing each query to read only part of the data. This may also occur when queries are started/restarted in quick succession. To minimize such issues, set the Kafka consumer session timeout (by setting option "kafka.session.timeout.ms") to be very small.

				When this is set, option "groupIdPrefix" will be ignored.
includeHeaders	boolean	false	streaming and batch	Whether to include the Kafka headers in the row.

Consumer Caching

It's time-consuming to initialize Kafka consumers, especially in streaming scenarios where processing time is a key factor. Because of this, Spark pools Kafka consumers on executors, by leveraging Apache Commons Pool.

The caching key is built up from the following information:

- Topic name
- Topic partition
- Group ID

The following properties are available to configure the consumer pool:

Property Name	Default	Meaning	Since Version
spark.kafka.consumer.cache.capacity	64	The maximum number of consumers cached. Please note that it's a soft limit.	3.0.0
spark.kafka.consumer.cache.timeout	5m (5 minutes)	The minimum amount of time a consumer may sit idle in the pool before it is eligible for eviction by the evictor.	3.0.0
spark.kafka.consumer.cache.evictorThreadRunInterval	1m (1 minute)	The interval of time between runs of the idle evictor thread for consumer pool. When non-positive, no idle evictor thread will be run.	3.0.0
spark.kafka.consumer.cache.jmx.enable	false	Enable or disable JMX for pools created with this configuration instance. Statistics of the pool are available via JMX instance. The prefix of JMX name is set to "kafka010-cached-simple-kafka-consumer-pool".	3.0.0

The size of the pool is limited by `spark.kafka.consumer.cache.capacity`, but it works as "soft-limit" to not block Spark tasks.

Idle eviction thread periodically removes consumers which are not used longer than given timeout. If this threshold is reached when borrowing, it tries to remove the least-used entry that is currently not in use.

If it cannot be removed, then the pool will keep growing. In the worst case, the pool will grow to the max number of concurrent tasks that can run in the executor (that is, number of task slots).

If a task fails for any reason, the new task is executed with a newly created Kafka consumer for safety reasons. At the same time, we invalidate all consumers in pool which have same caching key, to remove consumer which was used in failed execution. Consumers which any other tasks are using will not be closed, but will be invalidated as well when they are returned into pool.

Along with consumers, Spark pools the records fetched from Kafka separately, to let Kafka consumers stateless in point of Spark's view, and maximize the efficiency of pooling. It leverages same cache key with Kafka consumers pool. Note that it doesn't leverage Apache Commons Pool due to the difference of characteristics.

The following properties are available to configure the fetched data pool:

Property Name	Default	Meaning	Since Version
spark.kafka.consumer.fetchedData.cache.timeout	5m (5 minutes)	The minimum amount of time a fetched data may sit idle in the pool before it is eligible for eviction by the evictor.	3.0.0
spark.kafka.consumer.fetchedData.cache.evictorThreadRunInterval	1m (1 minute)	The interval of time between runs of the idle evictor thread for fetched data pool. When non-positive, no idle evictor thread will be run.	3.0.0

Writing Data to Kafka

Here, we describe the support for writing Streaming Queries and Batch Queries to Apache Kafka. Take note that Apache Kafka only supports at least once write semantics. Consequently, when writing—either Streaming Queries or Batch Queries—to Kafka, some records may be duplicated; this can happen, for example, if Kafka needs to retry a message that was not acknowledged by a Broker, even though that Broker received and wrote the message record. Structured Streaming cannot prevent such duplicates from occurring due to these Kafka write semantics. However, if

writing the query is successful, then you can assume that the query output was written at least once. A possible solution to remove duplicates when reading the written data could be to introduce a primary (unique) key that can be used to perform de-duplication when reading.

The Dataframe being written to Kafka should have the following columns in schema:

Column	Type
key (optional)	string or binary
value (required)	string or binary
headers (optional)	array
topic (*optional)	string
partition (optional)	int

* The topic column is required if the “topic” configuration option is not specified.

The value column is the only required option. If a key column is not specified then a `null` valued key column will be automatically added (see Kafka semantics on how `null` valued key values are handled). If a topic column exists then its value is used as the topic when writing the given row to Kafka, unless the “topic” configuration option is set i.e., the “topic” configuration option overrides the topic column. If a “partition” column is not specified (or its value is `null`) then the partition is calculated by the Kafka producer. A Kafka partitioner can be specified in Spark by setting the `kafka.partitioner.class` option. If not present, Kafka default partitioner will be used.

The following options must be set for the Kafka sink for both batch and streaming queries.

Option	value	meaning
kafka.bootstrap.servers	A comma-separated list of host:port	The Kafka "bootstrap.servers" configuration.

The following configurations are optional:

Option	value	default	query type	meaning
topic	string	none	streaming and batch	Sets the topic that all rows will be written to in Kafka. This option overrides any topic column that may exist in the data.
includeHeaders	boolean	false	streaming and batch	Whether to include the Kafka headers in the row.

Creating a Kafka Sink for Streaming Queries

Scala Java Python

```
# Write key-value data from a DataFrame to a specific Kafka topic specified in an option
ds = df \
  .selectExpr("CAST(key AS STRING)", "CAST(value AS STRING)") \
  .writeStream \
  .format("kafka") \
  .option("kafka.bootstrap.servers", "host1:port1,host2:port2") \
  .option("topic", "topic1") \
  .start()

# Write key-value data from a DataFrame to Kafka using a topic specified in the data
ds = df \
  .selectExpr("topic", "CAST(key AS STRING)", "CAST(value AS STRING)") \
  .writeStream \
  .format("kafka") \
  .option("kafka.bootstrap.servers", "host1:port1,host2:port2") \
  .start()
```

Writing the output of Batch Queries to Kafka

Scala Java Python

```
# Write key-value data from a DataFrame to a specific Kafka topic specified in an option
df.selectExpr("CAST(key AS STRING)", "CAST(value AS STRING)") \
  .write \
  .format("kafka") \
  .option("kafka.bootstrap.servers", "host1:port1,host2:port2") \
  .option("topic", "topic1") \
  .save()

# Write key-value data from a DataFrame to Kafka using a topic specified in the data
df.selectExpr("topic", "CAST(key AS STRING)", "CAST(value AS STRING)") \
  .write \
  .format("kafka") \
  .option("kafka.bootstrap.servers", "host1:port1,host2:port2") \
  .save()
```

Producer Caching

Given Kafka producer instance is designed to be thread-safe, Spark initializes a Kafka producer instance and co-use across tasks for same caching key.

The caching key is built up from the following information:

- Kafka producer configuration

This includes configuration for authorization, which Spark will automatically include when delegation token is being used. Even we take authorization into account, you can expect same Kafka producer instance will be used among same Kafka producer configuration. It will use different Kafka producer when delegation token is renewed; Kafka producer instance for old delegation token will be evicted according to the cache policy.

The following properties are available to configure the producer pool:

Property Name	Default	Meaning	Since Version
spark.kafka.producer.cache.timeout	10m (10 minutes)	The minimum amount of time a producer may sit idle in the pool before it is eligible for eviction by the evictor.	2.2.1
spark.kafka.producer.cache.evictorThreadRunInterval	1m (1 minute)	The interval of time between runs of the idle evictor thread for producer pool. When non-positive, no idle evictor thread will be run.	3.0.0

Idle eviction thread periodically removes producers which are not used longer than given timeout. Note that the producer is shared and used concurrently, so the last used timestamp is determined by the moment the producer instance is returned and reference count is 0.

Kafka Specific Configurations

Kafka's own configurations can be set via `DataStreamReader.option` with `kafka.` prefix, e.g, `stream.option("kafka.bootstrap.servers", "host:port")`. For possible kafka parameters, see Kafka consumer config docs for parameters related to reading data, and Kafka producer config docs for parameters related to writing data.

Note that the following Kafka params cannot be set and the Kafka source or sink will throw an exception:

- **group.id**: Kafka source will create a unique group id for each query automatically. The user can set the prefix of the automatically generated group.id's via the optional source option `groupIdPrefix`, default value is "spark-kafka-source". You can also set "kafka.group.id" to force Spark to use a special group id, however, please read warnings for this option and use it with caution.
- **auto.offset.reset**: Set the source option `startingOffsets` to specify where to start instead. Structured Streaming manages which offsets are consumed internally, rather than rely on the kafka Consumer to do it. This will ensure that no data is missed when new topics/partitions are dynamically subscribed. Note that `startingOffsets` only applies when a new streaming query is started, and that resuming will always pick up from where the query left off.
- **key.deserializer**: Keys are always deserialized as byte arrays with ByteArrayDeserializer. Use DataFrame operations to explicitly deserialize the keys.
- **value.deserializer**: Values are always deserialized as byte arrays with ByteArrayDeserializer. Use DataFrame operations to explicitly deserialize the values.
- **key.serializer**: Keys are always serialized with ByteArraySerializer or StringSerializer. Use DataFrame operations to explicitly serialize the keys into either strings or byte arrays.
- **value.serializer**: values are always serialized with ByteArraySerializer or StringSerializer. Use DataFrame operations to explicitly serialize the values into either strings or byte arrays.
- **enable.auto.commit**: Kafka source doesn't commit any offset.
- **interceptor.classes**: Kafka source always read keys and values as byte arrays. It's not safe to use ConsumerInterceptor as it may break the query.

Deploying

As with any Spark applications, `spark-submit` is used to launch your application. `spark-sql-kafka-0-10_2.12` and its dependencies can be directly added to `spark-submit` using `--packages`, such as,

```
./bin/spark-submit --packages org.apache.spark:spark-sql-kafka-0-10_2.12:3.0.1 ...
```

For experimenting on `spark-shell`, you can also use `--packages` to add `spark-sql-kafka-0-10_2.12` and its dependencies directly,

```
./bin/spark-shell --packages org.apache.spark:spark-sql-kafka-0-10_2.12:3.0.1 ...
```

See Application Submission Guide for more details about submitting applications with external dependencies.

Security

Kafka 0.9.0.0 introduced several features that increases security in a cluster. For detailed description about these possibilities, see Kafka security docs.

It's worth noting that security is optional and turned off by default.

Spark supports the following ways to authenticate against Kafka cluster:

- **Delegation token (introduced in Kafka broker 1.1.0)**
- **JAAS login configuration**

Delegation token

This way the application can be configured via Spark parameters and may not need JAAS login configuration (Spark can use Kafka's dynamic JAAS configuration feature). For further information about delegation tokens, see Kafka delegation token docs.

The process is initiated by Spark's Kafka delegation token provider. When `spark.kafka.clusters.${cluster}.auth.bootstrap.servers` is set, Spark considers the following log in options, in order of preference:

- **JAAS login configuration**, please see example below.
- **Keytab file**, such as,

```
./bin/spark-submit \
    --keytab <KEYTAB_FILE> \
    --principal <PRINCIPAL> \
    --conf spark.kafka.clusters.${cluster}.auth.bootstrap.servers=<KAFKA_SERVERS> \
    ...
```

- **Kerberos credential cache**, such as,

```
./bin/spark-submit \
    --conf spark.kafka.clusters.${cluster}.auth.bootstrap.servers=<KAFKA_SERVERS> \
    ...
```

The Kafka delegation token provider can be turned off by setting `spark.security.credentials.kafka.enabled` to `false` (default: `true`).

Spark can be configured to use the following authentication protocols to obtain token (it must match with Kafka broker configuration):

- **SASL SSL (default)**
- **SSL**
- **SASL PLAINTEXT (for testing)**

After obtaining delegation token successfully, Spark distributes it across nodes and renews it accordingly. Delegation token uses `SCRAM` login module for authentication and because of that the appropriate `spark.kafka.clusters.${cluster}.sasl.token.mechanism` (default: `SCRAM-SHA-512`) has to be configured. Also, this parameter must match with Kafka broker configuration.

When delegation token is available on an executor Spark considers the following log in options, in order of preference:

- **JAAS login configuration**, please see example below.
- **Delegation token**, please see `spark.kafka.clusters.${cluster}.target.bootstrap.servers.regex` parameter for further details.

When none of the above applies then unsecure connection assumed.

Configuration

Delegation tokens can be obtained from multiple clusters and `${cluster}` is an arbitrary unique identifier which helps to group different configurations.

Property Name	Default	Meaning	Sin Ver
`spark.kafka.clusters.${cluster}.auth.bootstrap.servers`	None	A list of coma separated host/port pairs to use for establishing the initial connection to the Kafka cluster. For further details please see Kafka documentation. Only used to obtain delegation token.	3.0.

spark.kafka.clusters.${cluster}.target.bootstrap.servers.regex	.*	Regular expression to match against the bootstrap.servers config for sources and sinks in the application. If a server address matches this regex, the delegation token obtained from the respective bootstrap servers will be used when connecting. If multiple clusters match the address, an exception will be thrown and the query won't be started. Kafka's secure and unsecure listeners are bound to different ports. When both used the secure listener port has to be part of the regular expression.	3.0.
spark.kafka.clusters.${cluster}.security.protocol	SASL_SSL	Protocol used to communicate with brokers. For further details please see Kafka documentation. Protocol is applied on all the sources and sinks as default where bootstrap.servers config matches (for further details please see spark.kafka.clusters.${cluster}.target.bootstrap.servers.regex), and can be overridden by setting kafka.security.protocol on the source or sink.	3.0.
spark.kafka.clusters.${cluster}.sasl.kerberos.service.name	kafka	The Kerberos principal name that Kafka runs as. This can be defined either in Kafka's JAAS config or in Kafka's config. For further details please see Kafka documentation. Only used to obtain delegation token.	3.0.
spark.kafka.clusters.${cluster}.ssl.truststore.location	None	The location of the trust store file. For further details please see Kafka documentation. Only used to obtain delegation token.	3.0.
spark.kafka.clusters.${cluster}.ssl.truststore.password	None	The store password for the trust store file. This is optional and only needed if spark.kafka.clusters.${cluster}.ssl.truststore.location is configured. For further details please see Kafka documentation. Only used to obtain delegation token.	3.0.
spark.kafka.clusters.${cluster}.ssl.keystore.location	None	The location of the key store file. This is optional for client and can be used for two-way authentication for client. For further details please see Kafka documentation. Only used to obtain delegation token.	3.0.
spark.kafka.clusters.${cluster}.ssl.keystore.password	None	The store password for the key store file. This is optional and only needed if spark.kafka.clusters.${cluster}.ssl.keystore.location is configured. For further details please see Kafka documentation. Only used to obtain delegation token.	3.0.
spark.kafka.clusters.${cluster}.ssl.key.password	None	The password of the private key in the key store file. This is optional for client. For further details please see Kafka documentation. Only used to obtain delegation token.	3.0.
spark.kafka.clusters.${cluster}.sasl.token.mechanism	SCRAM-SHA-512	SASL mechanism used for client connections with delegation token. Because SCRAM login module used for authentication a compatible mechanism has to be set here. For further details please see Kafka documentation (sasl.mechanism). Only used to authenticate against Kafka broker with delegation token.	3.0.

Kafka Specific Configurations

Kafka's own configurations can be set with `kafka.` prefix, e.g, `--conf spark.kafka.clusters.${cluster}.kafka.retries=1`. For possible Kafka parameters, see Kafka adminclient config docs.

Caveats

- Obtaining delegation token for proxy user is not yet supported (KAFKA-6945).

JAAS login configuration

JAAS login configuration must placed on all nodes where Spark tries to access Kafka cluster. This provides the possibility to apply any custom authentication logic with a higher cost to maintain. This can be done several ways. One possibility is to provide additional JVM parameters, such as,

```
./bin/spark-submit \
    --driver-java-options "-Djava.security.auth.login.config=/path/to/custom_jaas.conf" \
    --conf spark.executor.extraJavaOptions=-Djava.security.auth.login.config=/path/to/custom_jaas.conf \
    ...
```