Streams FAQ

Attention

We are looking for feedback on APIs, operators, documentation, and really anything that will make the end user experience better. Feel free to provide your feedback via email to `users@kafka.apache.org`.

General

Is Kafka Streams a project separate from Kafka?

No, it is not. The Kafka Streams API -- aka Kafka Streams -- is a component of the Apache Kafka® open source project, and thus included in Kafka 0.10+ releases. The source code is available at https://github.com/apache/kafka/tree/trunk/streams.

Is Kafka Streams a proprietary library of Confluent?

No, it is not. The Kafka Streams API -- aka Kafka Streams -- is a component of the Kafka open source project, and thus included in Kafka 0.10+ releases. The source code is available at https://github.com/apache/kafka/tree/trunk/streams.

Do Kafka Streams applications run inside the Kafka brokers?

No, they don't run inside the Kafka brokers. "Kafka Streams applications" are normal Java applications that happen to use the Kafka Streams library. You would run these applications on client machines at the perimeter of a Kafka cluster. In other words, Kafka Streams application do not run inside the Kafka brokers (servers) or the Kafka cluster -- they are client-side applications.

Why Does My Kafka Streams Application Use So Much Memory?

If your Kafka Streams application is using a permanent state store, you must take care to close any `Iterator`s after you are done using them to reclaim resouces. Not closing an `Iterator` can lead to OOM issues.

What are the system dependencies of Kafka Streams?

Kafka Streams has no external dependencies on systems other than Kafka.

How do I migrate my older Kafka Streams applications to the latest Confluent Platform version?

We provide instructions in our Upgrade Guide.

Which versions of Kafka clusters are supported by Kafka Streams?

The following versions are supported:

	Kafka Broker (columns)		
Streams API (rows)	3.0.x / 0.10.0.x	3.1.x / 0.10.1.x and 3.2.x / 0.10.2.x	3.3.x / 0.11.0.x and 4.0.x / 1.0.x and 4.1.x / 1.1.x and 5.0.x / 2.0.x and 5.1.x / 2.1.x and 5.2.x / 2.2.x and 5.3.x / 2.3.x
3.0.x / 0.10.0.x	compatible	compatible	compatible
3.1.x / 0.10.1.x and 3.2.x / 0.10.2.x		compatible	compatible
3.3.x / 0.11.0.x		compatible with exactly-once turned off (requires broker version Confluent Platform 3.3.x or higher)	compatible
4.0.x / 1.0.x and 4.1.x / 1.1.x and 5.0.x / 2.0.x and 5.1.x / 2.1.x and 5.2.x / 2.2.x and 5.3.x / 2.3.x		compatible with exactly-once turned off (requires broker version Confluent Platform 3.3.x or higher); requires message format 0.10 or higher	compatible; requires message format 0.10 or higher

The Streams API is not compatible with Kafka clusters running older Kafka versions (0.7, 0.8, 0.9).

What programming languages are supported?

The Kafka Streams API is implemented in Java. TheDeveloper Guide provides several example applications written in Java 7 and Java 8+. Additionally, Kafka Streams ships with a Scala wrapper on top of Java. You can also write applications in other JVM-based languages such as Kotlin or Clojure, but there is no native support for those languages.

Why is my application re-processing data from the beginning?

Kafka remembers your application by storing consumer offsets in a special topic. Offsets are numbers assigned to messages by the Kafka broker(s) indicating the order in which they arrived at the broker(s). By remembering your application's last committed offset, your application is only going to process newly arrived messages.

The configuration setting `offsets.retention.minutes` controls how long Kafka will remember offsets in the special topic. The default value is 10,080 minutes (7 days). Note that the default value in older versions is only 1,440 minutes (24 hours).

If your application is stopped (hasn't connected to the Kafka cluster) for a while, you could end up in a situation where you start reprocessing data on application restart because the broker(s) have deleted the offsets in the meantime. The actual startup behavior depends on your `auto.offest.reset` configuration that can be set to "earliest", "latest", or "none".

To avoid this problem, it is recommended to increase `offsets.retention.minutes` to an appropriately large value.

Scalability

Maximum parallelism of my application? Maximum number of app instances I can run?

Slightly simplified, the maximum parallelism at which your application may run is determined by the maximum number of partitions of the input topic(s) the application is reading from in its processing topology. The number of input partitions determines how many stream tasks Kafka Streams will create for the application, and the amount of stream tasks is the upper bound on the application's parallelism.

Let's take an example. Imagine your application is reading from an input topic that has 5 partitions. How many app instances can we run here?

The short answer is that we can run up to 5 instances of this application, because the application's maximum parallelism is 5. If we run more than 5 app instances, then the "excess" app instances will successfully launch but remain idle. If one of the busy instances goes down, one of the idle instances will resume the former's work.

The longer, more detailed answer for this example would be:

- **5 stream tasks** will be created, each of which will process one of the input partitions. And it's actually the number of stream tasks that determines the maximum parallelism of an application -- the number of input partitions is simply the main parameter from which the number of stream tasks is computed.

Now that we know the application's theoretical maximum parallelism, the next question is how to actually run the application at its maximum parallelism? To do so, we must ensure that all 5 stream tasks are running in parallel, i.e. there should be 5 processing threads, with each thread executing one task. There are several options at your disposal:

- **Option 1 is to scale horizontally (scaling out):** You run five single-threaded instances of your application, each of which executes one thread/task (i.e., `num.stream.threads` would be set to `1`; see optional configuration parameters). This option is used, for example, when you have many low-powered machines for running the app instances.
- **Option 2 is to scale vertically (scaling up):** You run a single multi-threaded instance of your application that will execute all threads/tasks (i.e., `num.stream.threads` would be set to `5`). This option is useful, for example, when you have a very powerful machine to run the app instance.
- **Option 3 combines options 1 and 2:** You run multiple instances of your application, each of which is running multiple threads. This option is used, for example, when your application runs at large scale. Here, you may even chose to run multiple app instances per machine.

Processing

Accessing record metadata such as topic, partition, and offset information?

Record metadata is accessible through the Processor API. It is also accessible indirectly through the DSL thanks to its Processor API integration.

With the Processor API, you can access record metadata through a `ProcessorContext`. You can store a reference to the context in an instance field of your processor during `Processor#init()`, and then query the processor context within `Processor#process()`, for example (same for `Transformer`). The context is updated automatically to match the record that is currently being processed, which means that methods such as `ProcessorContext#partition()` always return the current record's metadata. Some caveats apply when calling the processor context within scheduled `punctuate()` function, see the Javadocs for details.

If you use the DSL combined with a custom `Transformer`, for example, you could transform an input record's value to also include partition and offset metadata, and subsequent DSL operations such as `map` or `filter` could then leverage this information.

Difference between `map`, `peek`, `foreach` in the DSL?

The three methods `map`, `peek`, and `foreach` are quite similar to each other. In some sense, `peek` and `foreach` are but variants of `map`. An important reason for explicit support of all three is that it allows a developer to clearly communicate the *intent* of an operation:

- `map`: "When I use `map`, my intent is to **modify** the input stream and **continue processing** the (modified) output, i.e. I want to create an output stream that contains the modified data so that I can perform additional operations on the modified data."
- `foreach`: "When I use `foreach`, my intent is to **cause some side effects** based on the (unmodified) input stream and then I want to **terminate the processing**." That's why `foreach` returns `void` and thus is labeled as a terminal operation.
- `peek`: "When I use `peek`, my intent is to **cause some side effects** based on the (unmodified) input data and then I want to **continue the processing** of the (unmodified) input data."

See also the Java 8 documentation on java.util.Stream, which supports map, peek, and foreach operations, too.

How to avoid data repartitioning if you know it's not required?

Kafka Streams inserts a repartitioning step if a key-based operation like aggregation or join is preceded by a key changing operation like `selectKey()`, `map`, or `flatMap()`.

It is generally preferable to use `mapValues()` and `flatMapValues()` as they ensure the key has not been modified and thus, the repartitioning step can be omitted.

Often, users want to get read-only access to the key while modifying the value. For this case, you can ca `mapValues()` with a `ValueMapperWithKey` instead of using the `map()` operator. The `XxxWithKey` extension is available for multiple operators.

There are special cases for which you want to modify the key in a way that you know that the partitioning is preserved and thus repartitioning is actually not required. For these cases, it's sometimes possible, to apply the key changing operation *after* the aggregation to avoid the repartitioning step. You can apply this strategy if the original and modified key result in the same "groups of data".

Finally, you can always fall back to the Processor API and do a custom aggregation vi `process()`, `transform()` or `transformValues()`. Those operations do not automatically trigger a repartitioning step even if the key might have been modified in a previous operation.

Serdes `config` method

If you implement a custom Serde and specify this `Serde` in the config properties, and if this class implements `org.apache.kafka.common.Configurable`, `Serde#configure(...)` will be called automatically and your code should forward this call to the corresponding serializer and deserializer. The serdes that are provided by the library implement this pattern already.

If you manually call `new` to create a Serde and pass in this Serde via a method call, you must manually ca `configure(...)` for the library and custom Serdes.

Tip

Manually call `configure()` immediately after you have created the Serde object so that this step is not forgotten.

How can I replace RocksDB with a different store?

There is no global setting to replace RocksDB. However, you can set a custom store for each operator individually. For example `aggregate()` (and other operators with an internal store) has an overload that accepts a `Materialized` that enables specifying a `KeyValueBytesStoreSupplier`. You can use those overloads to provide a supplier that returns a different store that replaces RocksDB for the corresponding operator.

Note: if you want to replace RocksDB in every operator, you need to provide a differen `XxxByteStoreSupplier` for each operator.

Next to a RocksDB-based state store, the Kafka Streams API also ships with an in-memory store. This in-memory store will be backed by a changelog topic and is fully fault-tolerant. You can get an in-memory supplier via:

```
KeyValueBytesStoreSupplier inMemoryStoreSupplier = Stores.inMemoryKeyValueStore("myStoreName-mustBeUniqueForEachOperat
or");
Materialized materialized = Materialized.as(inMemoryStoreSupplier);
```

You can disable fault-tolerance for the in-memory store by an additional call to `Materialized#disableLogging()` before passing it to `aggregate()` (or any other operator), too.

If you want to plug-in any other third party store, you must implement the `StoreSupplier` and `StateStore` interfaces. The `StateStore` interface is for the actual integration code of the third party store into Kafka Streams. The `StoreSupplier` interface returns a *new* instance of your implemented store on each call of `get()`.

Note: if you integrate a third party store, it is your sole responsibility to take care of fault-tolerance. There will be no store backup via a changelog topic out-of-the-box, but you will need to implement it by yourself if required.

Failure and exception handling

Handling corrupted records and deserialization errors ("poison pill records")?

You might experience that some of the incoming records from your Kafka Streams application are corrupted, or that the serializer/deserializer might be incorrect or buggy, or cannot handle all record types. These types of records are referred to as “poison pills”.

You can use the `org.apache.kafka.streams.errors.DeserializationExceptionHandler` interface to customize how to handle such poison pill records. The library also includes multiple implementations that represent the common handling patterns:

- Option 1: Log the error and shut down the application
- Option 2: Skip corrupted records and log the error
- Option 3: Quarantine corrupted records (dead letter queue)
- Option 4: Interpret corrupted records as a sentinel value

Option 1: Log the error and shut down the application

`LogAndFailExceptionHandler` implements `DeserializationExceptionHandler` and is the default setting in Kafka Streams. It handles any encountered deserialization exceptions by logging the error and throwing a fatal error to stop your Streams application. If your application is configured to use `LogAndFailExceptionHandler`, then an instance of your application will fail-fast when it encounters a corrupted record by terminating itself.

```
Properties streamsSettings = new Properties();
streamsSettings.put(
  StreamsConfig.DEFAULT_DESERIALIZATION_EXCEPTION_HANDLER_CLASS_CONFIG,
  LogAndFailExceptionHandler.class.getName()
);
```

Option 2: Skip corrupted records and log the error

`LogAndContinueExceptionHandler` is an alternative implementation of `DeserializationExceptionHandler`. Your application will log an ERROR message whenever it encounters a corrupted record, skip the processing of that record, and continue processing the next record. The rate of skipped records is recorded in a processor-node level metric named `skippedDueToDeserializationError`. For a full list of metrics that you can use for monitoring and alerting, see the documentation).

```
Properties streamsSettings = new Properties();
streamsSettings.put(
  StreamsConfig.DEFAULT_DESERIALIZATION_EXCEPTION_HANDLER_CLASS_CONFIG,
  LogAndContinueExceptionHandler.class.getName()
);
```

Option 3: Quarantine corrupted records (dead letter queue)

You can provide your own `DeserializationExceptionHandler` implementation. For example, you can choose to forward corrupt records into a quarantine topic (think: a "dead letter queue") for further processing. To do this, use the Producer API to write a corrupted record directly to the quarantine topic. The drawback of this approach is that "manual" writes are side effects that are invisible to the Kafka Streams runtime library, so they do not benefit from the end-to-end processing guarantees of the Streams API.

Code example:

```
public class SendToDeadLetterQueueExceptionHandler implements DeserializationExceptionHandler {
    KafkaProducer<byte[], byte[]> dlqProducer;
    String dlqTopic;

    @Override
    public DeserializationHandlerResponse handle(final ProcessorContext context,
                                                 final ConsumerRecord<byte[], byte[]> record,
                                                 final Exception exception) {

        log.warn("Exception caught during Deserialization, sending to the dead queue topic; " +
                 "taskId: {}, topic: {}, partition: {}, offset: {}",
                 context.taskId(), record.topic(), record.partition(), record.offset(),
                 exception);

        dlqProducer.send(new ProducerRecord<>(dlqTopic, record.timestamp(), record.key(), record.value()));

        return DeserializationHandlerResponse.CONTINUE;
    }

    @Override
    public void configure(final Map<String, ?> configs) {
        dlqProducer = .. // get a producer from the configs map
        dlqTopic = .. // get the topic name from the configs map
    }
}

Properties streamsSettings = new Properties();
streamsSettings.put(
  StreamsConfig.DEFAULT_DESERIALIZATION_EXCEPTION_HANDLER_CLASS_CONFIG,
  SendToDeadLetterQueueExceptionHandler.class.getName()
);
```

Option 4: Interpret corrupted records as a sentinel value

You can also implement a custom serde to handle corrupt records internally rather than relying on `DeserializationExceptionHandler` settings. For example, you can implement a `org.apache.kafka.common.serialization.Deserializer` that, instead of throwing an exception, returns a "special-purpose" record that acts as sentinel record of your choice (e.g. null). This allows downstream processors of your application to recognize and handle such sentinel records accordingly whenever deserialization fails.

Code example:

```
public class ExceptionHandlingDeserializer implements Deserializer<MyObject> {

    public void configure(Map<String, ?> configs, boolean isKey) {
        // nothing to do
    }

    public MyObject deserialize(String topic, byte[] data) {
        try {
            // Attempt deserialization
            MyObject deserializedValue = deserialize(topic, data)

            // Ok, the record is valid (not corrupted).
            return deserializedValue;
        } catch (SerializationException e) {
            log.warn("Exception caught during Deserialization: {}", e.getMessage());

            // return the sentinel record upon corrupted data
            return null;
        }
    }

    public void close() {
        // nothing to do
    }
}
```

Important

Be careful when returning null `null` record values have special semantics for tables, where they are interpreted as tombstones that will cause the deletion of record keys from the table.

Once your custom serde is implemented, you can eitherconfigure your application to use it as the default key and/or value serde, or use it explicitly when calling API methods such as `KStream#to()`, or both.

Sending corrupt records to a quarantine topic or dead letter queue?

See Option 3: Quarantine corrupted records (dead letter queue)as described in Handling corrupted records and deserialization errors ("poison pill records")?.

Interactive Queries

Handling InvalidStateStoreException: "the state store may have migrated to another instance"?

When attempting to access a local state store, you may run into an error such as the following:

```
org.apache.kafka.streams.errors.InvalidStateStoreException: the state store, my-key-value-store, may have migrated to an
ther instance.
    at org.apache.kafka.streams.state.internals.StreamThreadStateStoreProvider.stores(StreamThreadStateStoreProvider.j
ava:49)
    at org.apache.kafka.streams.state.internals.QueryableStoreProvider.getStore(QueryableStoreProvider.java:55)
    at org.apache.kafka.streams.KafkaStreams.store(KafkaStreams.java:699)
```

Typically this happens for two reasons:

- The local `KafkaStreams` instance is not yet ready (i.e., not yet in runtime state `RUNNING`, see Run-time Status Information) and thus its local state stores cannot be queried yet.
- The local `KafkaStreams` instance is ready (e.g. in runtime state `RUNNING`), but the particular state store was just migrated to another instance behind the scenes. This may notably happen during the startup phase of a distributed application or when you are adding/removing application instances.

There are a couple of ways to prevent running into `InvalidStateStoreException`, but keep in mind that, in any case, you are working with information that is valid only for a particular point in time -- state stores may be migrated at any time behind the scenes, so your application should accept the fact that access to local state stores will come and go.

The simplest approach is to guard against `InvalidStateStoreException` when calling `KafkaStreams#store()`:

```
// Example: Wait until the store of type T is queryable.  When it is, return a reference to the store.
public static <T> T waitUntilStoreIsQueryable(final String storeName,
                                              final QueryableStoreType<T> queryableStoreType,
                                              final KafkaStreams streams) throws InterruptedException {
  while (true) {
    try {
      return streams.store(storeName, queryableStoreType);
    } catch (InvalidStateStoreException ignored) {
      // store not yet ready for querying
      Thread.sleep(100);
    }
  }
}
```

An end-to-end example is available at ValidateStateWithInteractiveQueriesLambdaIntegrationTest.

Security

Application fails when running against a secured Kafka cluster?

When your application processes data from a secured Kafka cluster, you may run into error messages such as:

```
> Could not create internal topics: Could not create topic: <NAME OF TOPIC> due to Cluster authorization failed”
```

Make sure that the principal running your application has the ACL `--cluster --operation Create` set so that the application has the permissions to create internal topics.

Troubleshooting and debugging

Easier to interpret Java stacktraces?

When you write Kafka Streams applications, you often create chains of method calls such as the following:

```
// Code all on one line.  Unfortunately, this is bad practice when it comes to stacktraces.
myStream.map(...).filter(...).groupByKey(...).count(...);
```

Now if your code happens to trigger a runtime error, the Java stacktrace may not be very helpful because the JVM provides only line-number information about where an error occurred ("NullPointerException at line 123"). So, to pick the example above, you may deduce that *some* operation in the map/filter/countByKey chain failed, but the stacktrace will not tell you where exactly in the affected line.

A simple trick is to split your method chains across multiple lines, so that the row number returned by the stacktrace will more easily identify the actual culprit:

```
// Split the same code across multiple lines to benefit from more actionable stacktraces.
myStream
  .map(...)
  .filter(...)
  .groupByKey(...)
  .count(...);
```

Visualizing topologies?

You can visualize a topology of a Kafka Streams application by accessing its `TopologyDescription`. A `TopologyDescription` contains *static* information (i.e., not runtime information) about the processing graph, i.e., all nodes, (global) stores, and how nodes or stores are connected to each other. For each node detailed information about node-name, input/output topics (for sources and sinks) etc are available. Additionally, nodes are grouped in sub-topology (i.e., groups of nodes that are only connected via topics).

```
// for DSL
StreamsBuilder builder = new StreamsBuilder();
Topology topology = builder.build();
// for Processor API
Topology topology = new Topology();

TopologyDescription description = topology.describe();

// Get sub-topologies
Set<Subtopology> subtopologies = description.subtopologies();
// You can also get all nodes of a sub-topology (can be a `Source`, `Processor`, or `Sink`)
// All nodes have a name and a set of predecessors as well as successors
// Source and Sink also have input/output topic information
// Processor can have stores attached
// Each subtopology has a unique ID
Subtopology subtopology = ...
Set<Node> nodes = subtopology.nodes();
// access node informaton...

// You can also get information about global stores
// a GlobalStore has a source topic and an `Processor`
Set<GlobalStore> globalStores = description.globalStores();
// access global store informaton...

// Or simply print all information at once
System.out.println(description);
```

Printing the `TopologyDescription` for WordCountLambdaExample would look as follows:

```
Sub-topologies:
  Sub-topology: 0
    Processor: KSTREAM-FILTER-0000000005(stores: []) --> KSTREAM-SINK-0000000004 <-- KSTREAM-KEY-SELECT-0000000002
    Processor: KSTREAM-KEY-SELECT-0000000002(stores: []) --> KSTREAM-FILTER-0000000005 <-- KSTREAM-FLATMAPVALUES-000000
0001
    Processor: KSTREAM-FLATMAPVALUES-0000000001(stores: []) --> KSTREAM-KEY-SELECT-0000000002 <-- KSTREAM-SOURCE-000000
0000
    Source: KSTREAM-SOURCE-0000000000(topics: inputTopic) --> KSTREAM-FLATMAPVALUES-0000000001
    Sink: KSTREAM-SINK-0000000004(topic: Counts-repartition) <-- KSTREAM-FILTER-0000000005
  Sub-topology: 1
    Source: KSTREAM-SOURCE-0000000006(topics: Counts-repartition) --> KSTREAM-AGGREGATE-0000000003
    Processor: KTABLE-TOSTREAM-0000000007(stores: []) --> KSTREAM-SINK-0000000008 <-- KSTREAM-AGGREGATE-0000000003
    Sink: KSTREAM-SINK-0000000008(topic: outputTopic) <-- KTABLE-TOSTREAM-0000000007
    Processor: KSTREAM-AGGREGATE-0000000003(stores: [Counts]) --> KTABLE-TOSTREAM-0000000007 <-- KSTREAM-SOURCE-0000000
006
Global Stores:
  none
```

Older releases

For releases prior to Confluent Platform 4.0.x, you can visualise the internal topology of a Kafka Streams application by calling the `KafkaStreams#toString()` method:

```
KafkaStreams streams = new KafkaStreams(topology, config);
// Start the Kafka Streams threads
streams.start();
// Print the internal topology to stdout
System.out.println(streams.toString());
```

An example topology for WordCountLambdaExample that is output from this call is shown below. All sources, sinks, and their backing topics, as well as all intermediate nodes and their state stores are collected and printed:

```
KafkaStreams processID: 5fe8281e-d756-42ec-8a92-111e4af3d4ca
    StreamsThread appId: wordcount-lambda-integration-test
    StreamsThread clientId: wordcount-lambda-integration-test-5fe8281e-d756-42ec-8a92-111e4af3d4ca
    StreamsThread threadId: wordcount-lambda-integration-test-5fe8281e-d756-42ec-8a92-111e4af3d4ca-StreamThread-1
    Active tasks:
        Running:
            StreamsTask taskId: 0_0
                ProcessorTopology:
                    KSTREAM-SOURCE-0000000000:
                        topics:        [inputTopic]
                        children:    [KSTREAM-FLATMAPVALUES-0000000001]
                    KSTREAM-FLATMAPVALUES-0000000001:
                        children:    [KSTREAM-KEY-SELECT-0000000002]
                    KSTREAM-KEY-SELECT-0000000002:
                        children:    [KSTREAM-FILTER-0000000005]
                    KSTREAM-FILTER-0000000005:
                        children:    [KSTREAM-SINK-0000000004]
                    KSTREAM-SINK-0000000004:
                        topic:        wordcount-lambda-integration-test-Counts-repartition
                Partitions [inputTopic-0]
            StreamsTask taskId: 1_0
                ProcessorTopology:
                    KSTREAM-SOURCE-0000000006:
                        topics:        [wordcount-lambda-integration-test-Counts-repartition]
                        children:    [KSTREAM-AGGREGATE-0000000003]
                    KSTREAM-AGGREGATE-0000000003:
                        states:        [Counts]
                        children:    [KTABLE-TOSTREAM-0000000007]
                    KTABLE-TOSTREAM-0000000007:
                        children:    [KSTREAM-SINK-0000000008]
                    KSTREAM-SINK-0000000008:
                        topic:        outputTopic
                Partitions [wordcount-lambda-integration-test-Counts-repartition-0]

            Suspended:
            Restoring:
            New:
        Standby tasks:
            Running:
            Suspended:
            Restoring:
            New:
```

Note that `KafkaStreams#toString()` returns more information than `TopologyDescription`, for example, the tasks currently assigned to each of the running stream threads. Starting in 4.0.x such runtime information is retrievable via the `KafkaStreams#localThreadsMetadata()` API, and as a result `KafkaStreams#toString()` is deprecated.

Inspecting streams and tables?

To inspect the records of a stream or a table, you can call the `KStream#print()` method. For a `KTable` you can inspect changes to it by getting a `KTable`'s changelog stream via `KTable#toStream()`. You can use `print()` to print the elements to `STDOUT` as shown below or you can write into a file via `Printed.toFile("fileName")`.

Here is an example that uses `KStream#print(Printed.toSysOut())`:

```
import java.util.concurrent.TimeUnit;
KStream<String, Long> left = ...;
KStream<String, Long> right = ...;

// Java 8+ example, using lambda expressions
KStream<String, String> joined = left
    .join(right,
      (leftValue, rightValue) -> leftValue + " --> " + rightValue, /* ValueJoiner */
      JoinWindows.of(Duration.ofMinutes(5)),
      Joined.with(
        Serdes.String(), /* key */
        Serdes.Long(),   /* left value */
        Serdes.Long()))  /* right value */
    .print(Printed.toSysOut());
```

The output would be the records after the join; e.g., if we were joining two records with the same key `K` and values `V1` and `V2`, then what is printed on the console would be `K, V1 --> V2` as shown for some sample data below:

```
alice, 5 --> 7
bob, 234 --> 19
charlie, 9 --> 10
```

If you want to inspect the content of a `KTable`'s internal store, you can use Interactive Queries.

Invalid Timestamp Exception

If you get an exception similar to the one shown below, there are multiple possible causes:

```
Exception in thread "StreamThread-1" org.apache.kafka.streams.errors.StreamsException: Input record {...} has invalid
(negative) timestamp. \
        Possibly because a pre-0.10 producer client was used to write this record to Kafka without embedding a timestamp
\
        or because the input topic was created before upgrading the Kafka cluster to 0.10+. \
        Use a different TimestampExtractor to process this data.
  at org.apache.kafka.streams.processor.FailOnInvalidTimestamp.onInvalidTimestamp(FailOnInvalidTimestamp.java:62)
```

This error means that the timestamp extractor of your Kafka Streams application failed to extract a valid timestamp from a record. Typically, this points to a problem with the record (e.g., the record does not contain a timestamp at all), but it could also indicate a problem or bug in the timestamp extractor used by the application.

When does a record not contain a valid timestamp?

- If you are using the default `FailOnInvalidTimestamp` timestamp extractor, it is most likely that your records do not carry an embedded timestamp (embedded record timestamps were introduced in Kafka's message format in Kafka `0.10`). This might happen, if for example, you consume a topic that is written by old Kafka producer clients (i.e., version `0.9` or earlier) or by third-party producer clients. Another situation where this may happen is after upgrading your Kafka cluster from `0.8` or `0.9` to `0.10`, where all the data that was generated with `0.8` or `0.9` does not include the `0.10` message timestamps. You can consider using an alternative timestamp extractor like `UsePreviousTimeOnInvalidTimestamp` or `LogAndSkipOnInvalidTimestamp`, both of which handle negative timestamps more gracefully. See Kafka Streams Developer Guide: Timestamp Extractor for details.
- If you are using a custom timestamp extractor, make sure that your extractor is properly handling invalid (negative) timestamps, where "properly" depends on the semantics of your application. For example, you can return a default timestamp or an estimated timestamp if you cannot extract a valid timestamp directly from the record.
- You can also switch from event-time processing to processing-time semantics via `WallclockTimestampExtractor`; whether such a fallback is an appropriate response to this situation depends on your use case.

However, as a first step you should identify and fix the root cause for why such problematic records were written to Kafka in the first place. In a second step you may consider applying workarounds as described above when dealing with such records. Another option is to regenerate the records with correct timestamps and write them to a new Kafka topic.

When the timestamp extractor causes the problem?

In this situation you should debug and fix the erroneous extractor. If the extractor is built into Kafka, please report the bug to the Kafka developer mailing list at dev@kafka.apache.org (see instructions at https://kafka.apache.org/contact); in the meantime, you may write a custom timestamp extractor that fixes the problem and configure your application to use that extractor for the time being.

Why do I get an `IllegalStateException` when accessing record metadata?

If you attach a new `Processor/Transformer/ValueTransformer` to your topology using a corresponding supplier, you need to make sure that the supplier returns a *new* instance each time `get()` is called. If you return the same object, a single `Processor/Transformer/ValueTransformer` would be shared over multiple tasks resulting in an `IllegalStateException` with error message `"This should not happen as topic() should only be called while a record is processed"` (depending on the method you are calling it could also be `partition()`, `offset()`, or `timestamp()` instead of `topic()`).

Why is `punctuate()` not called?

If you scheduled the `punctuate()` function based on *event-time* (i.e. `PunctuationType.STREAM_TIME`), then this function is triggered not based on *wall-clock-time* but purely data-driven (i.e., driven by internally tracked event-time). The event-time is derived from the extracted record timestamps provided by the used `TimestampExtractor`.

For example, let's assume you scheduled a `punctuate()` function every 10 seconds based on `PunctuationType.STREAM_TIME`. If you were to process a stream of 60 records with consecutive timestamps from 1 (first record) to 60 seconds (last record), then `punctuate()` would be called 6 times -- regardless of the time required to actually process those records; i.e., `punctuate()` would be called 6 times no matter whether processing these 60 records would take a second, a minute, or an hour.

Attention

Event-time is only advanced if all input partitions over all input topics have new data (with newer timestamps) available. If at least one partition does not have any new data available, event-time will not be advanced and thus `punctuate()` will not be triggered. This behavior is independent of the configured timestamp extractor, i.e., using `WallclockTimestampExtractor` does not enable wall-clock triggering of `punctuate()`.

Scala: compile error "no type parameter", "Java-defined trait is invariant in type T"

When using Scala without theScala wrapper, you may occasionally run into a Scala compiler error similar to the following

```
Error:(156, 8) no type parameters for method leftJoin:
    (x$1: org.apache.kafka.streams.kstream.KTable[String,VT],
     x$2: org.apache.kafka.streams.kstream.ValueJoiner[_ >: Long, _ >: VT, _ <: VR])
    org.apache.kafka.streams.kstream.KStream[String,VR]
exist so that it can be applied to arguments
    (org.apache.kafka.streams.kstream.KTable[String,String],
     org.apache.kafka.streams.kstream.ValueJoiner[Long,String,(String, Long)] with Serializable)
 --- because ---
argument expression's type is not compatible with formal parameter type;
 found   : org.apache.kafka.streams.kstream.ValueJoiner[Long,String,(String, Long)] with Serializable
 required: org.apache.kafka.streams.kstream.ValueJoiner[_ >: Long, _ >: ?VT, _ <: ?VR]
Note: Long <: Any (and org.apache.kafka.streams.kstream.ValueJoiner[Long,String,(String, Long)]
      with Serializable <: org.apache.kafka.streams.kstream.ValueJoiner[Long,String,(String, Long)]),
      but Java-defined trait ValueJoiner is invariant in type V1.
You may wish to investigate a wildcard type such as `_ <: Any`. (SLS 3.2.10)
Note: String <: Any (and org.apache.kafka.streams.kstream.ValueJoiner[Long,String,(String, Long)]
      with Serializable <: org.apache.kafka.streams.kstream.ValueJoiner[Long,String,(String, Long)]),
      but Java-defined trait ValueJoiner is invariant in type V2.
You may wish to investigate a wildcard type such as `_ <: Any`. (SLS 3.2.10)
```

The root cause of this problem is Scala-Java interoperability -- the Kafka Streams API is implemented in Java, but your application is written in Scala. Notably, this problem is caused by how the type systems of Java and Scala interact. Generic wildcards in Java, for example, are often causing such Scala issues.

It is recommended to use theScala wrapper to avoid this issue. If this is not possible, you must declare types explicitly in your Scala application in order for the code to compile. For example, you may need to break a single statement that chains multiple DSL operations into multiple statements, where each statement explicitly declares the respective return types. The StreamToTableJoinScalaIntegrationTest demonstrates how the types of return variables are explicitly declared.

How can I convert a KStream to a KTable without an aggregation step?

If you want to convert a derived `KStream` (i.e., a `KStream` that is not read from a Kafka topic) into a `KTable` you have two options.

Option 1: Write KStream to Kafka, read back as KTable

You can write the `KStream` into a Kafka topic and read it back as a `KTable`. As per our general recommendation, you should manually pre-create this topic to ensure it has the correct number of partitions, topic settings, and so on.

```
StreamsBuilder builder = new StreamsBuilder();
KStream<String, Long> stream = ...; // some computation that creates the derived KStream

// You should manually create the dummy topic before starting your application.
//
// Also, because you want to read the topic back as a KTable, you might want to enable
// log compaction for this topic to align the topic's cleanup policy with KTable semantics.
stream.to("dummy-topic", Produced.with(Serdes.String(), Serdes.Long()));
KTable<String, Long> table = builder.table("dummy-topic", Consumed.with(Serdes.String(), Serdes.Long()));
```

This is the simplest approach with regard to the code. However, it has the disadvantages that (a) you need to manage an additional topic and that (b) it results in additional network traffic because data is written to and re-read from Kafka.

Option 2: Perform a dummy aggregation

As an alternative to option 1, you can choose to create a dummy aggregation step:

```
StreamsBuilder builder = new StreamsBuilder();
KStream<String, Long> stream = ...; // some computation that creates the derived KStream

// Java 8+ example, using lambda expressions
KTable<String, Long> table = stream.groupByKey().reduce(
    (aggValue, newValue) -> newValue);

// Java 7 example
KTable<String, Long> table = stream.groupByKey().reduce(
    new Reducer<Long>() {
      @Override
      public Long apply(Long aggValue, Long newValue) {
        return newValue;
      }
    });
```

This approach is somewhat more complex with regard to the code compared to option 1 but has the advantage that (a) no manual topic management is required and (b) re-reading the data from Kafka is not necessary.

In option 2, Kafka Streams will create an internal changelog topic to back up the KTable for fault tolerance. Thus, both approaches require some additional storage in Kafka and result in additional network traffic. Overall, it's a trade-off between slightly more complex code in option 2 versus manual topic management in option 1.

RocksDB behavior in 1-core environments

There is a known issue with RocksDB when running in environments with just one CPU core. In some scenarios, the symptom is that the application's performance might be very slow or unresponsive. The workaround is to set a particular RocksDB configuration using the RocksDB config setter as follows:

```
 public static class CustomRocksDBConfig implements RocksDBConfigSetter {
   @Override
   public void setConfig(final String storeName, final Options options, final Map<String, Object> configs) {
      // Workaround: We must ensure that the parallelism is set to >= 2.
      int compactionParallelism = Math.max(Runtime.getRuntime().availableProcessors(), 2);
      // Set number of compaction threads (but not flush threads).
      options.setIncreaseParallelism(compactionParallelism);
   }
 }

Properties streamsSettings = new Properties();
streamsConfig.put(StreamsConfig.ROCKSDB_CONFIG_SETTER_CLASS_CONFIG, CustomRocksDBConfig.class);
```

---

2 Votes

Last updated on Sep 10, 2019.