Streams Upgrade Guide

Upgrading from Confluent Platform 5.2.x (Kafka 2.2.x-cp1) to Confluent Platform 5.3.0 (Kafka 2.3.0-cp1)

Compatibility

Kafka Streams applications built with Confluent Platform 5.3.0 (Kafka 2.3.0-cp1) are forward and backward compatible with certain Kafka clusters.

- **Forward-compatible to Confluent Platform 5.3.0 clusters (Kafka 2.3.0-cp1):**Existing Kafka Streams applications built with Confluent Platform 3.0.x (Kafka 0.10.0.x-cp1), Confluent Platform 3.1.x (Kafka 0.10.1.x-cp2), Confluent Platform 3.2.x (Kafka 0.10.2.x-cp1), Confluent Platform 3.3.x (Kafka 0.11.0.x-cp1), Confluent Platform 4.0.x (Kafka 1.0.x-cp1), Confluent Platform 4.1.x (Kafka 1.1.x-cp1), Confluent Platform 5.0.x (Kafka 2.0.x-cp1), Confluent Platform 5.1.x (Kafka 2.1.x-cp1), or Confluent Platform 5.2.x (Kafka 2.2.x-cp1) will work with upgraded Kafka clusters running Confluent Platform 5.3.0 (Kafka 2.3.0-cp1).
- **Backward-compatible to older clusters down to Confluent Platform 3.1.x (Kafka 0.10.1.x-cp2):**New Kafka Streams applications built with Confluent Platform 5.3.0 (Kafka 2.3.0-cp1) will work with older Kafka clusters running Confluent Platform 3.1.x (Kafka 0.10.1.x-cp2), Confluent Platform 3.2.x (Kafka 0.10.2.x-cp1), Confluent Platform 3.3.x (Kafka 0.11.0.x-cp1), Confluent Platform 4.0.x (Kafka 1.0.x-cp1), Confluent Platform 4.1.x (Kafka 1.1.x-cp1), Confluent Platform 5.0.x (Kafka 2.0.x-cp1), Confluent Platform 5.1.x (Kafka 2.1.x-cp2), or Confluent Platform 5.2.x (Kafka 2.2.x-cp1). However, when exactly-once processing guarantee is required, your Kafka cluster needs to be upgraded to at least Confluent Platform 3.3.x (Kafka 0.11.0.x-cp1). Note, that exactly-once feature is disabled by default and thus a rolling bounce upgrade of your Streams application is possible if you don't enable this new feature explicitly. Kafka clusters running Confluent Platform 3.0.x (Kafka 0.10.0.x-cp1) are *not* compatible with new Confluent Platform 5.0.0 Kafka Streams applications.

Note

As of Confluent Platform 4.0.0 (Kafka 1.0.0-cp1), Kafka Streams requires message format 0.10 or higher. Thus, if you kept an older message format when upgrading your brokers to Confluent Platform 3.1 (Kafka 0.10.1-cp1) or a later version, Kafka Streams Confluent Platform 5.3.0 (Kafka 2.3.0-cp1) won't work. You will need to upgrade the message format to 0.10 before you upgrade your Kafka Streams application to Confluent Platform 5.3.0 (Kafka 2.3.0-cp1) or newer.

Compatibility Matrix:

Expand Content

	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).

Upgrading your Kafka Streams applications to Confluent Platform 5.3.0

To make use of Confluent Platform 5.3.0 (Kafka 2.3.0-cp1), you need to update the Kafka Streams dependency of your application to use the version number `2.3.0-cp1`, and you may need to make minor code changes (details below), and then recompile your application.

For example, in your `pom.xml` file:

```
<dependency>
    <groupId>org.apache.kafka</groupId>
    <artifactId>kafka-streams</artifactId>
    <!-- update version to 2.3.0-cp1 -->
    <version>2.3.0-cp1</version>
</dependency>
```

As of the 5.3.0 release, Kafka Streams depends on a RocksDB version that requires MacOS 10.13 or higher.

Note

As of Confluent Platform 4.0.0 (Kafka 1.0.0-cp1) a topology regression was introduced when source `KTable` instances were changed to have changelog topics instead of re-using the source topic. As of Confluent Platform 5.0.0 (Kafka 2.0.0-cp1) `KTable` instances re-using the source topic as the changelog topic has been reinstated, but is optional and must be configured by setting `StreamsConfig.TOPOLOGY_OPTIMIZATION` to `StreamsConfig.OPTIMIZE`.

This brings up some different scenarios depending on what you are upgrading from and what you are upgrading to.

- If you are upgrading from Kafka 2.0.x-cp1 to Kafka 2.3.0-cp1 it's recommended to keep the existing optimization configuration. Changing the optimization level might make your application vulnerable to data loss for a small window of time.
- If you are upgrading from using `StreamsBuilder` version Kafka 1.0.x-cp1/1.1.x-cp1 to Kafka 2.3.0-cp1 you can enable the optimization, but your topology will change so you'll need to restart your application with a new application ID. Additionally, if you want to perform a rolling upgrade, it is recommended not to enable the optimization. If you elect not to enable the optimization, then no

further changes are required.

- If you are upgrading from using `KStreamBuilder` version Kafka 1.0.x-cp1/1.1.x-cp1 to `StreamsBuilder` Kafka 2.3.0-cp1, it's recommended to enable the optimization as there are no changes to your topology. Please note that if you elect not to enable the optimization, there is a small window of time for possible data loss until the changelog topic contains one record per key.

Additionally, when starting with a new application ID, you can possibly end up reprocessing data, since the application ID has been changed. If you don't want to reprocess records, you'll need to create new output topics, so downstream user can cut over in a controlled fashion.

Streams API changes

Several new APIs have been added in Confluent Platform 5.3.x and a few default configurations have changed. Additionally, the RocksDB dependency was updated. See details below.

Scala Suppress Operator

The `Suppress` operator has been added to the `kafka-streams-scala` KTable API.

In-memory Window/Session Stores

Streams now offers an in-memory version of the window and the session store, in addition to the persistent ones based on RocksDB. The new public interfaces `inMemoryWindowStore()` and `inMemorySessionStore()` are added to `Stores` and provide the built-in in-memory window or session store.

Serde close() and configure()

We have added default implementation to `close()` and `configure()` for `Serializer`, `Deserializer` and `Serde` so that they can be implemented by lambda expression. For more details please read KIP-331.

Store timestamps in RocksDB

To improve operator semantics, new store types are added that allow storing an additional timestamp per key-value pair or window. Some DSL operators (for example KTables) are using those new stores. Hence, you can now retrieve the last update timestamp via Interactive Queries if you specify `TimestampedKeyValueStoreType<` or `TimestampedWindowStoreType` as your `QueryableStoreType`. While this change is mainly transparent, there are some corner cases that may require code changes: Caution: If you receive an untyped store and use a cast, you might need to update your code to cast to the correct type. Otherwise, you might get an exception similar to `java.lang.ClassCastException: class org.apache.kafka.streams.state.ValueAndTimestamp cannot be cast to class YOUR-VALUE-TYPE` upon getting a value from the store. Additionally, `TopologyTestDriver#getStateStore()` only returns non-built-in stores and throws an exception if a built-in store is accessed. For more details please read KIP-258.

New flatTransformValues() Operator

To improve type safety, a new operator `KStream#flatTransformValues` is added. For more details please readKIP-313.

max.poll.interval default

Kafka Streams used to set the configuration parameter `max.poll.interval.ms` to `Integer.MAX_VALUE`. This default value is removed and Kafka Streams uses the consumer default value now. For more details please read KIP-442.

Repartition topic defaults

Default configuration for repartition topic was changed: The segment size for index files `segment.index.bytes` is no longer 50MB, but uses the cluster default. Similarly, the configuration``segment.ms`` in no longer 10 minutes, but uses the cluster default configuration. Lastly, the retention period (`retention.ms`) is changed from `Long.MAX_VALUE` to -1`(infinite). For more details please readKIP-443.

RocksDBConfigSetter close()

To avoid memory leaks, `RocksDBConfigSetter` has a new `close()` method that is called on shutdown. Users should implement this method to release any memory used by RocksDB config objects, by closing those objects. More details can be found here.

Upgrade RocksDB to v5.18.3

RocksDB dependency was updated to version 5.18.3. The new version allows to specify more RocksDB configurations, including `WriteBufferManager` that helps to limit RocksDB off-heap memory usage. For more details please readMemory Management.

Upgrading older Kafka Streams applications to Confluent Platform 5.3.x

Streams API changes in Confluent Platform 5.2.0

Confluent Platform 5.2.x adds some new APIs and improves some existing functionality.

Simplified KafkaStreams Transitions

The KafkaStreams#state transition diagram is simplified in Confluent Platform 5.2.0: in older versions the state will transit from `CREATED` to `RUNNING`, and then to `REBALANCING` to get the first stream task assignment, and then back to `RUNNING`; starting in 5.2.0 it will transit from `CREATED` directly to `REBALANCING` and then to `RUNNING`. If you have registered a `StateListener` that captures state transition events, you may need to adjust your listener implementation accordingly for this simplification (in practice, your listener logic should be very unlikely to be affected at all).

TimeWindowSerde

In `WindowedSerdes`, there is a new static constructor to return a `TimeWindowSerde` with configurable window size. This is to help users construct time window serdes to read directly from a time-windowed store's changelog. More details can be found in KIP-393.

AutoCloseable

In Confluent Platform 5.2.0 a few public interfaces have been extended, including `KafkaStreams` to extend `AutoCloseable` so that they can be used in a try-with-resource statement. For a full list of public interfaces that are affected, see KIP-376.

Streams API changes in Confluent Platform 5.1.0

There are some Streams API changes in Confluent Platform 5.1.0 If your application uses them, then you need to update your code accordingly.

New config class Grouped

We've added a new class `Grouped` and deprecated `Serialized`. The intent of adding `Grouped` is the ability to name repartition topics created when performing aggregation operations. Users can name the potential repartition topic using the `Grouped#as()` method which takes a `String` and is used as part of the repartition topic name. The resulting repartition topic name will still follow the pattern of `${application-id}->name<-repartition`. The `Grouped` class is now favored over `Serialized` in `KStream#groupByKey()`, `KStream#groupBy()`, and `KTable#groupBy()`. Note that Kafka Streams does not automatically create repartition topics for aggregation operations. Additionally, we've updated the `Joined` class with a new method `Joined#withName` enabling users to name any repartition topics required for performing Stream/Stream or Stream/Table join.

```
// old API
KStream<String, String> stream = ...

stream.groupByKey(Grouped.with(Serdes.String(), Serdes.String()))


stream.groupBy( (key, value) -> ... , Grouped.with(Serdes.String(), Serdes.String()))


KStream<String, String> streamII = ...
streamII.join(stream, (valueII, valueI) -> ... , JoinWindows.of(20000), Joined.with(Serdes.String(),
                                                                  Serdes.String(),
                                                                  Serdes.String()))
KTable<String, String> table = ...
table.groupBy( (key, value) -> ... , Grouped.with(Serdes.String(), Serdes.String()))

// new API
KStream<String, String> stream = ...

stream.groupByKey(Grouped.with(Serdes.String(), Serdes.String()))

                            // providing name for possible reparitiion topic
                            // using a name for the repartition topic is optional
stream.groupByKey(Grouped.with("repartition-topic-name",
                               Serdes.String(),
                               Serdes.String()))

stream.groupBy( (key, value) -> ... , Grouped.with(Serdes.String(), Serdes.String()))

                                      // providing name for possible reparitiion topic
                                      // using a name for the repartition topic is optional
stream.groupBy( (key, value) -> ... , Grouped.with("reparition-topic-name",
                                                   Serdes.String(),
                                                   Serdes.String()))
KStream<String, String> streamII = ...


streamII.join(stream, (valueII, valueI) -> ... , JoinWindows.of(20000), Joined.with(Serdes.String(),
                                                                  Serdes.String(),
                                                                  Serdes.String()))

streamII.join(stream, (valueII, valueI) -> ... , JoinWindows.of(20000), Joined.with(Serdes.String(),
                                                                  Serdes.String(),
                                                                  Serdes.String(),
                                                                  "join-repartition-topic-name"))
                                                        // providing name for possible reparitiion topic
                                                        // using a name for the repartiton topic is optional

KTable<String, String> table = ...

table.groupBy( (key, value) -> ... , Grouped.with(Serdes.String(), Serdes.String()))

                                        // providing name for possible reparitiion topic
                                        // using a name for the repartition topic is optional
table.groupBy( (key, value) -> ... , Grouped.with("repartition-topic-name",
                                                  Serdes.String(),
                                                  Serdes.String()))
```

New UUID Serde

We added a new serde for UUIDs `Serdes.UUIDSerde`) that you can use via `Serdes.UUID()`.

Improved API Semantics

We updated a list of methods that take `long` arguments as either timestamp (fix point) or duration (time period) and replaced them with `Instant` and `Duration` parameters for improved semantics. Some old methods based on `long` are deprecated, and users are encouraged to update their code.

In particular, aggregation windows (hopping/tumbling/unlimited time windows and session windows) as well as join windows now take `Duration` arguments to specify window size, hop, and gap parameters. Also, window sizes and retention times are now specified as `Duration` type in the `Stores` class. The `Window` class has new methods `#startTime()` and `#endTime()` that return window start/end timestamp as `Instant`. For Interactive Queries, there are new `#fetch(...)` overloads taking `Instant` arguments. Additionally, punctuations are now registerd via `ProcessorContext#schedule(Duration interval, ...)`.

We deprecated `KafkaStreams#close(...)` and replaced it with `KafkaStreams#close(Duration)` that accepts a single timeout argument.

Note

The new `KafkaStreams#close(Duration)` method has improved (but slightly different) semantics than the old one. It does not block forever if the provided timeout is zero and does not accept any negative timeout values. If you want to block, you should pass in `Duration.ofMillis(Long.MAX_VALUE)`.

```
// old API

KStream stream = ...

// Tumbling/Hopping Windows
stream.groupByKey().
      .windowedBy(TimeWindows.of(5 * 60 * 1000)
                             .advanceBy(TimeUnit.MINUTES.toMillis(1)),
                 ...);

// Unlimited Windows
stream.groupByKey().
      .windowedBy(UnlimitedWindows.startOn(System.currentTimeMillis()), ...);

// Session Windows
stream.groupByKey().
      .windowedBy(SessionWindows.with(100), ...);

// Joining Streams
KStream secondStream = ...
stream.join(stream2, JoinWindows.of(5000)
                                .before(10000)
                                .after(10000), ...)

// Accessing Window start/end-timestamp
KTable<Windowed<K>, V> table = ...
table.toStream()
     .foreach((wKey, value) -> {
       // can still be used for performance reasons (eg, in a Processor)
       long windowStartTimeMs = wKey.window().start();
       long windowEndTimeMs = wKey.window().end();
       ...
     });

// Registering a Punctuation
MyProcessor implements Processor { // same for Transformer[WithKey] and ValueTransformer[WithKey]
  // other methods omitted for brevity

  void init(ProcessorContext context) {
    context.schedule(1000, PunctuationType.STREAM_TIME, timestamp -> {...});
  }
}

// Interactive Queries
KafkaStreams streams = ...
ReadOnlyWindowStore windowStore = streams.store(...);

long now = System.currentTimeMillis();
WindowStoreIterator<V> it = windowStore.fetch(key, now - 5000, now);
WindowStoreIterator<V> all = windowStore.all(now - 5000, now);

// Creating State Stores
Stores.persistentWindowStore("storeName",
                             24 * 3600 * 1000, // retention period in ms
                             3,                // number of segments
                             10 * 60 * 1000,   // window size in ms
                             false);           // retain duplicates
Stores.persistentSessionStore("storeName",
                              24 * 3600 * 1000); // retention period in ms

// Closing KafkaStreams
KafkaStreams streams = ...
streams.close(30, TimeUnit.SECONDS);
streams.close(0, TimeUnit.SECONDS); // block forever



// new API
```

```java
// New API

KStream stream = ...

// Tumbling/Hopping Windows
stream.groupByKey().
      .windowedBy(TimeWindow.of(Duration.ofMinutes(5)
                            .advanceBy(Duration.ofMinutes(1)),
                  ...);

// Unlimited Windows
stream.groupByKey().
      .windowedBy(UnlimitedWindows.startOn(Instant.now()), ...);

// Session Windows
stream.groupByKey().
      .windowedBy(SessionWindows.with(Duration.ofMillis(100)), ...);

// Joining Streams
KStream secondStream = ...
stream.join(stream2, JoinWindows.of(Duration.ofSeconds(5)
                                .before(Duration.ofSeconds(10))
                                .after(Duration.ofSeconds(10)), ...)

// Accessing Window start/end-timestamp
KTable<Windowed<K>, V> table = ...
table.toStream()
     .foreach((wKey, value) -> {
       // better semantics with new API
       Instant windowStartTime = wKey.window().startTime();
       Instant windowEndTime = wKey.window().endTime();
       // can still be used for performance reasons (eg, in a Processor)
       long windowStartTimeMs = wKey.window().start();
       long windowEndTimeMs = wKey.window().end();
       ...
     });

// Registering a Punctuation
MyProcessor implements Processor { // same for Transformer[WithKey] and ValueTransformer[WithKey]
  // other methods omitted for brevity

  void init(ProcessorContext context) {
    context.schedule(Duration.ofSeconds(1), PunctuationType.STREAM_TIME, timestamp -> {...});
  }
}

// Interactive Queries
KafkaStreams streams = ...
ReadOnlyWindowStore windowStore = streams.store(...);

Instant now = Instant.now();
WindowStoreIterator<V> it = windowStore.fetch(key, now.minus(Duration.ofSeconds(5), now);
WindowStoreIterator<V> all = windowStore.all(now.minus(Duration.ofSeconds(5), now);

// Creating State Stores
Stores.persistentWindowStore("storeName",
                             Duration.ofDay(1),      // retention period
                             // number of segments is removed
                             Duration.ofMinutes(10), // window size
                             false);                 // retain duplicates
Stores.persistentSessionStore("storeName",
                              Duration.ofDay(1)); // retention period

// Closing KafkaStreams
KafkaStreams streams = ...
streams.close(Duration.ofSeconds(30));
streams.close(Duration.ofMillis(Long.MAX_VALUE)); // block forever
```

Simplified Store Segments

We deprecated the notion of *number of segments* in window stores and replaced it with *segment interval*. In the old API, the segment interval was computed as `min( retention-period / (number-of-segments - 1) , 60_000L)` (with *60* seconds being a lower segment size bound). In the new API, there is no lower bound on the segment interval (i.e., minimum segment interval is `1` millisecond) and the number of segments is `1 + (retention-period / segment-interval)`.

If you used 3 segments in your store with a retention time of 24 hours, you should now use a segment interval of 12 hours *(12 = 24 / (3 - 1))* in the new API to get the same behavior.

Method `Windows#segments()` and variable `Windows#segments` were deprecated. Similarly, `WindowBytesStoreSupplier#segments()` was deprecated and replaced with `WindowBytesStoreSupplier#segmentInterval()`. If you implement custom windows or custom window stores, be aware that you will need to update your code when those methods are removed.

Finally, `Stores#persistentWindowStore(...)` was deprecated and replaced with a new overload that does not allow specifying the number of segments any longer because those are computed automatically based on retention time and segment interval. If you create persistent window stores explicitly, you should update your code accordingly.

```
// Old API

Stores.persistentWindowStore("storeName",
                             24 * 3600 * 1000, // retention period in ms
                             3,                // number of segments
                             10 * 60 * 1000,   // window size in ms
                             false);           // retain duplicates
// New API

Stores.persistentWindowStore("storeName",
                             Duration.ofDay(1),      // retention period (updated from `long` to `Duration`)
                             // number of segments is removed
                             Duration.ofMinutes(10), // window size (updated from `long` to `Duration`)
                             false);                 // retain duplicates
```

Out-of-Order Data Handling

We added a new config (max.task.idle.ms) to allow users to specify how to handle out-of-order data within a task that may be processing multiple Kafka topic partitions (see the Out-of-Order Handling section for more details). The default value is set to `0`, to favor minimized processing latency. If users would like to wait on processing when only part of the topic partitions of a given task have data available in order to reduce risks of handling out-of-order data, they can override this config to a larger value.

AdminClient Metrics Exposed

The newly exposed `AdminClient` metrics are now included with other available metrics when calling the `KafkaStream#metrics()` method. For more details on monitoring streams applications check out Monitoring Streams Applications.

Topology Description Improved

We updated the <code>TopologyDescription</code> API to allow for better runtime checking. Users are encouraged to use <code>#topicSet()</code> and <code>#topicPattern()</code> accordingly on <code>TopologyDescription.Source</code> nodes, instead of using <code>#topics()</code>, which has since been deprecated. Similarly, use <code>#topic()</code> and <code>#topicNameExtractor()</code> to get descriptions of <code>TopologyDescription.Sink</code> nodes.

```
// old API

TopologyDescription.Source source = ... // get Source node from a TopologyDescription
TopologyDescription.Sink sink = ... // get Sink node from a TopologyDescription

String topics = source.topics(); // comma separated list of topic names or pattern (as String)

String topic = sink.topic(); // return the output topic name (never null)


// new API

TopologyDescription.Source source = ... // get Source node from a TopologyDescription
TopologyDescription.Sink sink = ... // get Sink node from a TopologyDescription


Set<String> topics = source.topicSet(); // set of all topic names (can be null if pattern subscription is uses)
// or
Pattern pattern = source.topicPattern(); // topic pattern (can be null)

String topic = sink.topic(); // return the output topic name (can be null if dynamic topic routing is used)
// or
TopicNameExtractor topicNameExtractor = sink.topicNameExtractor(); // return the use TopicNameExtractor (can be null)
```

StreamsBuilder#build Method Overload

We've added an overloaded `StreamsBuilder#build` method that accepts an instance of `java.util.Properties` with the intent of using the``StreamsConfig#TOPOLOGY_OPTIMIZATION`` config added in Kafka Streams 2.0. Before 2.1, when building a topology with the DSL, Kafka Streams writes the physical plan as the user makes calls on the DSL. Now, by providing a `java.util.Properties` instance when executing a `StreamsBuilder#build` call, Kafka Streams can optimize the physical plan of the topology, provided the `StreamsConfig#TOPOLOGY_OPTIMIZATION` config is set to `StreamsConfig#OPTIMIZE`. By setting `StreamsConfig#OPTIMIZE` in addition to the `KTable` optimization of reusing the source topic as the changelog topic, the topology may be optimized to merge redundant repartition topics into one repartition topic. The original no parameter version of `StreamsBuilder#build` is still available if you don't want to optimize your topology. Note that enabling optimization of the topology may require you to do an application reset when redeploying the application. For more details, see Optimizing Kafka Streams

Full upgrade workflow

A typical workflow for upgrading Kafka Streams applications from Confluent Platform 5.0.x to Confluent Platform 5.1.0 has the following steps:

1. **Upgrade your application:** See upgrade instructions above.
2. **Stop the old application:** Stop the old version of your application, i.e. stop all the application instances that are still running the old version of the application.
3. **Optional, upgrade your Kafka cluster:** See kafka upgrade instructions. *Note, if you want to use exactly-once processing semantics, upgrading your cluster to at least Confluent Platform 3.3.x is mandatory.*
4. **Start the upgraded application:** Start the upgraded version of your application, with as many instances as needed. By default, the upgraded application will resume processing its input data from the point when the old version was stopped (see previous step).

Streams API changes in Confluent Platform 5.0

A few new Streams configurations and public interfaces are added into Confluent Platform 5.0.x release. Additionally, some deprecated APIs are removed in Confluent Platform 5.0.x release.

Skipped Records Metrics Refactored

Starting with Confluent Platform 5.0.0, Kafka Streams does not report the `skippedDueToDeserializationError-rate` and `skippedDueToDeserializationError-total` metrics.

Deserialization errors, and all other causes of record skipping, are now accounted for in the pre-existing metrics `skipped-records-rate` and `skipped-records-total`. When a record is skipped, the event is now logged at WARN level. Note these metrics are mainly for monitoring unexpected events; If there are systematic issues that caused too many unprocessable records to be skipped, and hence the resulted warning logs become burdensome, you should consider filtering our these unprocessable records instead of depending on record skipping semantics. For more details, see KIP-274.

As of right now, the potential causes of skipped records are:

- `null` keys in table sources.
- `null` keys in table-table inner/left/outer/right joins.
- `null` keys or values in stream-table joins.
- `null` keys or values in stream-stream joins.
- `null` keys or values in aggregations / reductions / counts on grouped streams.
- `null` keys in aggregations / reductions / counts on windowed streams.
- `null` keys in aggregations / reductions / counts on session-windowed streams.
- Errors producing results, when the configured `default.production.exception.handler` decides to `CONTINUE` (the default is to `FAIL` and throw an exception).
- Errors deserializing records, when the configured `default.deserialization.exception.handler` decides to `CONTINUE` (the default is to `FAIL` and throw an exception). This was the case previously captured in the `skippedDueToDeserializationError` metrics.
- Fetched records having a negative timestamp.

New Functions in Window Store Interface

Confluent Platform now supports methods in `ReadOnlyWindowStore` which allows you to query the key-value pair of a single window. If you have customized window store implementations on the above interface, you must update your code to implement the newly added method. For more details, see KIP-261.

Simplified KafkaStreams Constructor

The `KafakStreams` constructor was simplfied. Instead of requiring the user to create a boilderplate `StreamsConfig` object, the constructor now directly accepts the `Properties` object that specifies the actual user configuration.

```
StreamsBuilder builder = new StreamsBuilder();
// define processing logic
Topology topology = builder.build();

// or

Topology topology = new Topology();
// define processing logic


Properties props = new Properties();
// define configuration


// old API

KafkaStream stream = new KafkaStreams(topology, new StreamsConfig(props));
KafkaStream stream = new KafkaStreams(topology, new StreamsConfig(props), /* pass in KafkaClientSupplier or Time */);


// new API

KafkaStream stream = new KafkaStreams(topology, props);
KafkaStream stream = new KafkaStreams(topology, props, /* pass in KafkaClientSupplier or Time */);
```

Support Dynamic Routing at Sink

In this release you can now dynamically route records to Kafka topics. More specifically, in both the lower-level `Topology#addSink` and higher-level `KStream#to` APIs, we have added variants that take a `TopicNameExtractor` instance instead of a specific `String` topic name. For each record received from the upstream processor, the `TopicNameExtractor` will dynamically determine which Kafka topic to write to based on the record's key and value, as well as record context. Note that all output Kafka topics are still considered user topics and hence must be pre-created. Also, we have modified the `StreamPartitioner` interface to add the topic name parameter since the topic name now may not be known beforehand; users who have customized implementations of this interface would need to update their code while upgrading their application.

Support Message Headers

In this release there is message header support in the `Processor API`. In particular, we have added a new API `ProcessorContext#headers()` which returns a `Headers` object that keeps track of the headers of the source topic's message that is being processed. Through this object, users can manipulate the headers map that is being propagated throughout the processor topology as well, for example `Headers#add(String key, byte[] value)` and `Headers#remove(String key)`. When Streams DSL is used, users can call `process` or `transform` in which they can also access the `ProcessorContext` to access and manipulate the message header; if user does not manipulate the header, it will still be preserved and forwarded while the record traverses through the processor topology. When the resulted record is sent to the sink topics, the preserved message header will also be encoded in the sent record.

KTable Now Supports Transform Values

In this release another new API, *KTable#transformValues*, was added. For more information, see *KIP-292 <https://cwiki.apache.org/confluence/display/KAFKA/KIP-292%3A+Add+transformValues%28%29+method+to+KTable>*__.

Improved Windowed Serde Support

We added helper class `WindowedSerdes` that allows you to create time- and session-windowed serdes without the need to know the details how windows are de/serialized. The created window serdes wrap a user-provided serde for the inner key- or value-data type. Furthermore, two new configs `default.windowed.key.serde.inner` and `default.windowed.value.serde.inner` were added that allow to specify the default inner key- and value-serde for windowed types. Note, these new configs are only effective, if `default.key.serde` or `default.value.serde` specifies a windowed serde (either `WindowedSerdes.TimeWindowedSerde` or `WindowedSerdes.SessionWindowedSerde`).

Allow Timestamp Manipulation

Using the Processor API, it is now possible to set the timestamp for output messages explicitly. This change implies updates to the `ProcessorContext#forward()` method. Some existing methods were deprecated and replaced by new ones. In particular, it is not longer possible to send records to a downstream processor based on its index.

```
// old API

public class MyProcessor implements Processor<String, Integer> {
  private ProcessorContext context;

  @Override
  public void init(ProcessorContext context) {
    this.context = context;
  }

  @Override
  public void process(String key, Integer value) {
    // do some computation

    // send record to all downstream processors
    context.forward(newKey, newValue);
    // send record to particular downstream processors (if it exists; otherwise drop record)
    context.forward(newKey, newValue, "downstreamProcessorName");
    // send record to particular downstream processors per index (throws if index is invalid)
    int downStreamProcessorIndex = 2;
    context.forward(newKey, newValue, downStreamProcessorIndex);
  }

  @Override
  public void close() {} // nothing to do
}


// new API

public class MyProcessor implements Processor<String, Integer> {
  // omit other methods that don't change for brevity

  @Override
  public void process(String key, Integer value) {
    // do some computation

    // send record to all downstream processors
    context.forward(newKey, newValue); // same as old API
    context.forward(newKey, newValue, To.all()); // new; same as line above
    // send record to particular downstream processors (if it exists; otherwise drop record)
    context.forward(newKey, newValue, To.child("downstreamProcessorName"));
    // send record to particular downstream processors per index (throws if index is invalid)
    // -> not supported in new API

    // new: set record timestamp
    long outputRecordTimestamp = 42L;
    context.forward(newKey, newValue, To.all().withTimestamp(outputRecordTimestamp));
    context.forward(newKey, newValue, To.child("downstreamProcessorName").withTimestamp(outputRecordTimestamp));
  }
}
```

Public Test-Utils Artifact

Confluent Platform now ships with a `kafka-streams-test-uitls` artifact that contains utility classes to unit test your Kafka Streams application. Check out Testing Streams Code section for more details.

Scala API

Confluent Platform now ships with the Apache Kafka Scala API for Kafka Streams. You can add the dependency for Scala 2.11 or 2.12 artifacts:

```
<dependency>
    <groupId>org.apache.kafka</groupId>
    <artifactId>kafka-streams-scala_2.11</artifactId>
    <!-- or Scala 2.12
    <artifactId>kafka-streams-scala_2.12</artifactId>
    -->
    <version>2.0.0-cp1</version>
</dependency>
```

Deprecated APIs are Removed

The following deprecated APIs are removed in Confluent Platform 5.0.0:

1. **KafkaStreams#toString** no longer returns the topology and runtime metadata; to get topology metadata you can cal `Topology#describe()`, and to get thread runtime metadata you can call `KafkaStreams#localThreadsMetadata` (deprecated since Confluent Platform 4.0.0). For detailed guidance on how to update your code please read here.
2. **TopologyBuilder** and **KStreamBuilder** are removed and replaced by `Topology` and `StreamsBuidler` respectively (deprecated since Confluent Platform 4.0.0).
3. **StateStoreSupplier** are removed and replaced with `StoreBuilder` (deprecated since Confluent Platform 4.0.0); and the corresponding **Stores#create** and **KStream, KTable, KGroupedStream**'s overloaded functions that use it have also been removed.
4. **KStream, KTable, KGroupedStream** overloaded functions that requires serde and other specifications explicitly are removed and replaced with simpler overloaded functions that use `Consumed, Produced, Serialized, Materialized, Joined` (deprecated since Confluent Platform 4.0.0).
5. **Processor#punctuate**, **ValueTransformer#punctuate**, **ValueTransformer#punctuate** and **RecordContext#schedule(long)** are removed and replaced by `RecordContext#schedule(long, PunctuationType, Punctuator)` (deprecated since Confluent Platform 4.0.0).
6. The second `boolean` typed parameter **loggingEnabled in ProcessorContext#register** has been removed; you can now use `StoreBuilder#withLoggingEnabled, #withLoggingDisabled` to specify the behavior when they create the state store (deprecated since Confluent Platform 3.3.0).
7. **KTable#writeAs, #print, #foreach, #to, #through** are removed as their semantics are more confusing than useful, you can call `KTable#tostream()#writeAs` etc instead for the same purpose (deprecated since Confluent Platform 3.3.0).
8. **StreamsConfig#KEY_SERDE_CLASS_CONFIG, #VALUE_SERDE_CLASS_CONFIG, #TIMESTAMP_EXTRACTOR_CLASS_CONFIG** are removed and replaced with `StreamsConfig#DEFAULT_KEY_SERDE_CLASS_CONFIG, #DEFAULT_VALUE_SERDE_CLASS_CONFIG, #DEFAULT_TIMESTAMP_EXTRACTOR_CLASS_CONFIG` respectively (deprecated since Confluent Platform 3.3.0).
9. **StreamsConfig#ZOOKEEPER_CONNECT_CONFIG** is removed as we do not need ZooKeeper dependency in Streams any more (deprecated since Confluent Platform 3.2.0).

Streams API changes in Confluent Platform 4.1

A few new Streams configurations and public interfaces are added into Confluent Platform 4.1.x release.

Changes in bin/kafka-streams-application-reset

Added options to specify input topics offsets to reset according toKIP-171.

Embedded Admin Client Configuration

You can now customize the embedded admin client inside your Streams application which would be used to send all the administrative requests to Kafka brokers, such as internal topic creation, etc. This is done via the additional `KafkaClientSupplier#getAdminClient(Map<String, Object>)` interface; for example, users can provide their own `AdminClient` implementations to override the default ones in their integration testing. In addition, users can also override the configs that are passed into `KafkaClientSupplier#getAdminClient(Map<String, Object>)` to configure the returned `AdminClient`. Such overridden configs can be specified via the `StreamsConfig` by adding the admin configs with the prefix as defined by `StreamsConfig#adminClientPrefix(String)`. Any configs that aren't admin client configs will be ignored.

For example:

```
Properties streamProps = ...;
// use retries=10 for the embedded admin client
streamsProps.put(StreamsConfig.adminClientPrefix("retries"), 10);
```

Streams API changes in Confluent Platform 4.0

Kafka Streams and its API were improved and modified in the Confluent Platform 4.0.x release. All of these changes are backward compatible, thus it's not require to update the code of your Kafka Streams applications immediately. However, some methods were deprecated and thus it is recommend to update your code eventually to allow for future upgrades. In this section we focus on deprecated APIs.

Building and running a topology

The two main classes to specify a topology, `KStreamBuilder` and `TopologyBuilder`, were deprecated and replaced by `StreamsBuilder` and `Topology`. Note, that both new classes are in package `org.apache.kafka.streams` and that `StreamsBuilder` does not extend `Topology`, i.e., the class hierarchy is different now. This change also affects `KafkaStreams` constructors that now only accept a `Topology`. If you use `StreamsBuilder` you can obtain the constructed topology via `StreamsBuilder#build()`.

The new classes have basically the same methods as the old ones to build a topology via DSL or Processor API. However, some internal methods that were public in `KStreamBuilder` and `TopologyBuilder`, but not part of the actual API, are no longer included in the new classes.

```
// old API

KStreamBuilder builder = new KStreamBuilder(); // for DSL
// or
TopologyBuilder builder = new TopologyBuilder(); // for Processor API

Properties props = new Properties();
KafkaStreams streams = new KafkaStreams(builder, props);

// new API

StreamsBuilder builder = new StreamsBuilder(); // for DSL
// ... specify computational logic
Topology topology = builder.build();
// or
Topology topology = new Topology(); // for Processor API

Properties props = new Properties();
KafkaStreams streams = new KafkaStreams(topology, props);
```

Describing topology and stream task metadata

`KafkaStreams#toString()` and `KafkaStreams#toString(final String indent)`, which were previously used to retrieve the user-specified processor topology information as well as runtime stream tasks metadata, are deprecated in 4.0.0. Instead, a new method of `KafkaStreams`, namely `localThreadsMetadata()` is added which returns an `org.apache.kafka.streams.processor.ThreadMetadata` object for each of the local stream threads that describes the runtime state of the thread as well as its current assigned tasks metadata. Such information will be very helpful in terms of debugging and monitoring your streams applications. For retrieving the specified processor topology information, users can now call `Topology#describe()` which returns an `org.apache.kafka.streams.TopologyDescription` object that contains the detailed description of the topology (for DSL users they would need to call `StreamsBuilder#build()` to get the `Topology` object first).

Merging KStreams:

As mentioned above, `KStreamBuilder` was deprecated in favor of `StreamsBuilder`. Additionally, `KStreamBuilder#merge(KStream...)` was replaced by `KStream#merge(KStream)` and thus `StreamsBuilder` does not have a `merge()` method. Note: instead of merging an arbitrary number of `KStream` instances into a single `KStream` as in the old API, the new `#merge()` method only accepts a single `KStream` and thus merges two `KStream` instances into one. If you want to merge more than two `KStream` instances, you can call `KStream#merge()` multiple times.

```
// old API

KStreamBuilder builder = new KStreamBuilder();

KStream<Long, String> firstStream = ...;
KStream<Long, String> secondStream = ...;
KStream<Long, String> thirdStream = ...;

KStream<Long, String> mergedStream = builder.merge(
  firstStream,
  secondStream,
  thirdStream);

// new API

StreamsBuilder builder = new StreamsBuilder();

KStream<Long, String> firstStream = ...;
KStream<Long, String> secondStream = ...;
KStream<Long, String> thirdStream = ...;

KStream<Long, String> mergedStream = firstStream.merge(secondStream)
                                                .merge(thirdStream);
```

Punctuation functions

The Processor API was extended to allow users to schedule `punctuate` functions either based on event-time (i.e. `PunctuationType.STREAM_TIME`) or *wall-clock-time* (i.e. `PunctuationType.WALL_CLOCK_TIME`). Before this, users could only schedule based on *event-time* and hence the `punctuate` function was data-driven only. As a result, the original `ProcessorContext#schedule` is deprecated with a new overloaded function. In addition, the `punctuate` function inside `Processor` is also deprecated, and is replaced by the newly added `Punctuator#punctuate` interface.

```java
// old API (punctuate defined in Processor, and schedule only with stream-time)

public class WordCountProcessor implements Processor<String, String> {

    private ProcessorContext context;
    private KeyValueStore<String, Long> kvStore;

    @Override
    @SuppressWarnings("unchecked")
    public void init(ProcessorContext context) {
        // keep the processor context locally because we need it in punctuate() and commit()
        this.context = context;

        // call this processor's punctuate() method every 1000 milliseconds
        this.context.schedule(1000);

        // retrieve the key-value store named "Counts"
        kvStore = (KeyValueStore) context.getStateStore("Counts");
    }

    @Override
    public void punctuate(long timestamp) {
        KeyValueIterator<String, Long> iter = this.kvStore.all();
        while (iter.hasNext()) {
            KeyValue<String, Long> entry = iter.next();
            context.forward(entry.key, entry.value.toString());
        }
        iter.close();

        // commit the current processing progress
        context.commit();
    }

    // .. other functions
}


// new API (punctuate defined in Punctuator, and schedule can be either stream-time or wall-clock-time)

public class WordCountProcessor implements Processor<String, String> {

    private ProcessorContext context;
    private KeyValueStore<String, Long> kvStore;

    @Override
    @SuppressWarnings("unchecked")
    public void init(ProcessorContext context) {
        // keep the processor context locally because we need it in punctuate() and commit()
        this.context = context;

        // retrieve the key-value store named "Counts"
        kvStore = (KeyValueStore) context.getStateStore("Counts");

        // schedule a punctuate() method every 1000 milliseconds based on stream time
        this.context.schedule(1000, PunctuationType.STREAM_TIME, (timestamp) -> {
            KeyValueIterator<String, Long> iter = this.kvStore.all();
            while (iter.hasNext()) {
                KeyValue<String, Long> entry = iter.next();
                context.forward(entry.key, entry.value.toString());
            }
            iter.close();

            // commit the current processing progress
            context.commit();
        });
    }

    // .. other functions
}
```

Streams Configuration

You can now override the configs that are used to create internal repartition and changelog topics. You provide these configs via the `StreamsConfig` by adding the topic configs with the prefix as defined by `StreamsConfig#topicPrefix(String)`. Any properties in the `StreamsConfig` with the prefix will be applied when creating internal topics. Any configs that aren't topic configs will be ignored. If you are already using `StateStoreSupplier` or `Materialized` to provide configs for changelogs, then they will take precedence over those supplied in the config.

For example:

```java
Properties streamProps = ...;
// use cleanup.policy=delete for internal topics
streamsProps.put(StreamsConfig.topicPrefix("cleanup.policy"), "delete");
```

New classes for optional DSL parameters

Several new classes were introduced, i.e., `Serialized`, `Consumed`, `Produced` etc. to enable us to reduce the overloads in the DSL. These classes mostly have a static method `with` to create an instance, i.e., `Serialized.with(Serdes.Long(), Serdes.String())`.

Scala users should be aware that they will need to surround `with` with backticks.

For example:

```
// When using Scala: enclose "with" with backticks
Serialized.`with`(Serdes.Long(), Serdes.String())
```

Streams API changes in Confluent Platform 3.3

Kafka Streams and its API were improved and modified since the release of Confluent Platform 3.2.x. All of these changes are backward compatible, thus it's not require to update the code of your Kafka Streams applications immediately. However, some methods and configuration parameters were deprecated and thus it is recommend to update your code eventually to allow for future upgrades. In this section we focus on deprecated APIs.

Streams Configuration

The following configuration parameters were renamed and their old names were deprecated.

> - `key.serde` renamed to `default.key.serde`
> - `value.serde` renamed to `default.value.serde`
> - `timestamp.extractor` renamed to `default.timestamp.extractor`

Thus, `StreamsConfig#KEY_SERDE_CONFIG`, `StreamsConfig#VALUE_SERDE_CONFIG`, and `StreamsConfig#TIMESTAMP_EXTRACTOR_CONFIG` were deprecated, too.

Additionally, the following method changes apply:

> - method `keySerde()` was deprecated and replaced by `defaultKeySerde()`
> - method `valueSerde()` was deprecated and replaced by `defaultValueSerde()`
> - new method `defaultTimestampExtractor()` was added

Local timestamp extractors

The Streams API was extended to allow users to specify a per stream/table timestamp extractor. This simplifies the usage of different timestamp extractor logic for different streams/tables. Before, users needed to apply an `if-then-else` pattern within the default timestamp extractor to apply different logic to different input topics. The old behavior introduced unnecessary dependencies and thus limited code modularity and code reuse.

To enable the new feature, the methods `KStreamBuilder#stream()`, `KStreamBuilder#table()`, `KStream#globalTable()`, `TopologyBuilder#addSource()`, and `TopologyBuilder#addGlobalStore()` have new overloads that allow to specify a "local" timestamp extractor that is solely applied to the corresponding input topics.

```java
// old API (single default TimestampExtractor that is applied globally)

public class MyTimestampExtractor implements TimestampExtractor {
    @Override
    public long extract(ConsumerRecord record, long previousTimestamp) {
        long timestamp;

        String topic = record.topic();
        switch (topic) {
            case "streamInputTopic":
                timestamp = record.value().getDataTimestamp(); // assuming that value type has a method #getDataTimesta
mp()
                break;
            default:
                timestamp = record.timestamp();
        }

        if (timestamp < 0) {
            throw new RuntimeException("Invalid negative timestamp.");
        }

        return timestamp;
    }
}

KStreamBuilder builder = new KStreamBuilder();
KStream stream = builder.stream(keySerde, valueSerde, "streamInputTopic");
KTable table= builder.table("tableInputTopic");

Properties props = new Properties(); // omitting mandatory configs for brevity
// set MyTimestampExtractor as global default extractor for all topics
config.set("default.timestamp.extractor", MyTimestampExtractor.class);

KafkaStreams streams = new KafkaStreams(builder, props);


// new API (custom TimestampExtractor for topic "streamInputTopic" only; returns value embedded timestamp)

public class StreamTimestampExtractor implements TimestampExtractor {
    @Override
    public long extract(ConsumerRecord record, long previousTimestamp) {
        long timestamp = record.value().getDataTimestamp(); // assuming that value type has a method #getDataTimestamp()


        if (timestamp < 0) {
            throw new RuntimeException("Invalid negative timestamp.");
        }

        return timestamp;
    }
}

KStreamBuilder builder = new KStreamBuilder();
// set StreamTimestampExtractor explicitly for "streamInputTopic"
KStream stream = builder.stream(new StreamTimestampExtractor(), keySerde, valueSerde, "streamInputTopic");
KTable table= builder.table("tableInputTopic");

Properties props = new Properties(); // omitting mandatory configs for brevity

KafkaStreams streams = new KafkaStreams(builder, props);
```

KTable Changes

The following methods have been deprecated on the `KTable` interface

> - `void foreach(final ForeachAction<? super K, ? super V> action)`
> - `void print()`
> - `void print(final String streamName)`
> - `void print(final Serde<K> keySerde, final Serde<V> valSerde)`
> - `void print(final Serde<K> keySerde, final Serde<V> valSerde, final String streamName)`
> - `void writeAsText(final String filePath)`
> - `void writeAsText(final String filePath, final Serde<K> keySerde, final Serde<V> valSerde)`
> - `void writeAsText(final String filePath, final String streamName)`
> - `void writeAsText(final String filePath, final String streamName, final Serde<K> keySerde, final Serde<V> valSerde)`

These methods have been deprecated in favor of using the Interactive Queries API.

If you want to query the current content of the state store backing the `KTable`, use the following approach:

> - Make a call to `KafkaStreams.store(String storeName, QueryableStoreType<T> queryableStoreType)` followed by a call to `ReadOnlyKeyValueStore.all()` to iterate over the keys of a `KTable`.

If you want to view the changelog stream of the `KTable` then you could do something along the lines of the following:

> - Call `KTable.toStream()` then call `KStream#print()`.

Streams API changes in Confluent Platform 3.2

Kafka Streams and its API were improved and modified since the release of Confluent Platform 3.1.x. Some of these changes are breaking changes that require you to update the code of your Kafka Streams applications. In this section we focus on only these breaking changes.

Handling Negative Timestamps and Timestamp Extractor Interface

Kafka Streams behavior with regard to invalid (i.e., negative) timestamps was improved. By default you will still get an exception on an invalid timestamp. However, you can reconfigure your application to react more gracefully to invalid timestamps which was not possible before.

Even if you do not use a custom timestamp extractor, you need to recompile your application code, because the `TimestampExtractor` interface was changed in an incompatible way.

The internal behavior of Kafka Streams with regard to negative timestamps was changed. Instead of raising an exception if the timestamp extractor returns a negative timestamp, the corresponding record will be dropped silently and not be processed. This allows to process topic for which only a few records cannot provide a valid timestamp. Furthermore, the `TimestampExtractor` interface was changed and now has one additional parameter. This parameter provides a timestamp that can be used, for example, to return an estimated timestamp, if no valid timestamp can be extracted from the current record.

The old default timestamp extractor `ConsumerRecordTimestampExtractor` was replaced with `FailOnInvalidTimestamp`, and two new extractors which both extract a record's built-in timestamp were added (`LogAndSkipOnInvalidTimestamp` and `UsePreviousTimeOnInvalidTimestamp`). The new default extractor (`FailOnInvalidTimestamp`) raises an exception in case of a negative built-in record timestamp such that Kafka Streams' default behavior is kept (i.e., fail-fast on negative timestamp). The two newly added extractors allow to handle negative timestamp more gracefully by implementing a log-and-skip and timestamp-estimation strategy.

```
// old interface
public class TimestampExtractor {
  // returning -1 results in an exception
  long extract(ConsumerRecord<Object, Object> record);
}

// new interface
public class TimestampExtractor {
  // provides a timestamp that could be used as a timestamp estimation,
  // if no valid timestamp can be extracted from the current record
  //
  // allows to return -1, which tells Kafka Streams to not process the record (it will be dropped silently)
  long extract(ConsumerRecord<Object, Object> record, long previousTimestamp);
}
```

Metrics

If you provide custom metrics by implementing interface `StreamsMetrics` you need to update your code as the interface has many new methods allowing to register finer grained metrics than before. More details are available in KIP-114.

```
// old interface
public interface StreamsMetrics {
  // Add the latency sensor.
  Sensor addLatencySensor(String scopeName, String entityName, String operationName, String... tags);

  // Record the given latency value of the sensor.
  void recordLatency(Sensor sensor, long startNs, long endNs);
}

// new interface
public interface StreamsMetrics {
  // Get read-only handle on global metrics registry.
  Map<MetricName, ? extends Metric> metrics();

  // Add a latency and throughput sensor for a specific operation
  Sensor addLatencyAndThroughputSensor(final String scopeName,
                                       final String entityName,
                                       final String operationName,
                                       final Sensor.RecordingLevel recordingLevel,
                                       final String... tags);

  // Record the given latency value of the sensor.
  void recordLatency(final Sensor sensor,
                     final long startNs,
                     final long endNs);

  // Add a throughput sensor for a specific operation:
  Sensor addThroughputSensor(final String scopeName,
                             final String entityName,
                             final String operationName,
                             final Sensor.RecordingLevel recordingLevel,
                             final String... tags);

  // Record the throughput value of a sensor.
  void recordThroughput(final Sensor sensor,
                        final long value);

  // Generic method to create a sensor.
  Sensor addSensor(final String name,
                   final Sensor.RecordingLevel recordingLevel);

  // Generic method to create a sensor with parent sensors.
  Sensor addSensor(final String name,
                   final Sensor.RecordingLevel recordingLevel,
                   final Sensor... parents);

  // Remove a sensor.
  void removeSensor(final Sensor sensor);
}
```

Scala

Starting with 0.10.2.0, if your application is written in Scala, you may need to declare types explicitly in order for the code to compile. The StreamToTableJoinScalaIntegrationTest has an example where the types of return variables are explicitly declared.

Streams API changes in Confluent Platform 3.1

Stream grouping and aggregation

Grouping (i.e., repartitioning) and aggregation of the `KStream` API was significantly changed to be aligned with the `KTable` API. Instead of using a single method with many parameters, grouping and aggregation is now split into two steps. First, a `KStream` is transformed into a `KGroupedStream` that is a repartitioned copy of the original `KStream`. Afterwards, an aggregation can be performed on the `KGroupedStream`, resulting in a new `KTable` that contains the result of the aggregation.

Thus, the methods `KStream#aggregateByKey(...)`, `KStream#reduceByKey(...)`, and `KStream#countByKey(...)` were replaced by `KStream#groupBy(...)` and `KStream#groupByKey(...)` which return a `KGroupedStream`. While `KStream#groupByKey(...)` groups on the current key, `KStream#groupBy(...)` sets a new key and re-partitions the data to build groups on the new key. The new clas `KGroupedStream` provides the corresponding methods `aggregate(...)`, `reduce(...)`, and `count(...)`.

```
KStream stream = builder.stream(...);
Reducer reducer = new Reducer() { /* ... */ };

// old API
KTable newTable = stream.reduceByKey(reducer, name);

// new API, Group by existing key
KTable newTable = stream.groupByKey().reduce(reducer, name);
// or Group by a different key
KTable otherTable = stream.groupBy((key, value) -> value).reduce(reducer, name);
```

Auto Repartitioning

Previously when performing `KStream#join(...)`, `KStream#outerJoin(...)` or `KStream#leftJoin(...)` operations after a key changing operation, i.e, `KStream#map(...)`, `KStream#flatMap(...)`, `KStream#selectKey(...)` the developer was required to call `KStream#through(...)` to repartition the mapped `KStream` This is no longer required. Repartitioning now happens automatically for all join operations.

```
KStream streamOne = builder.stream(...);
KStream streamTwo = builder.stream(...);
KeyValueMapper selector = new KeyValueMapper() { /* ... */ };
ValueJoiner joiner = new ValueJoiner { /* ... */ };
JoinWindows windows = JoinWindows.of("the-join").within(60 * 1000);

// old API
KStream oldJoined = streamOne.selectKey(selector)
                             .through("repartitioned-topic")
                             .join(streamTwo,
                                   joiner,
                                   windows);

// new API
KStream newJoined = streamOne.selectKey((key,value) -> value)
                             .join(streamTwo,
                                   joiner,
                                   windows);
```

TopologyBuilder

Two public method signatures have been changed on `TopologyBuilder`, `TopologyBuilder#sourceTopics(String applicationId)` and `TopologyBuilder#topicGroups(String applicationId)`. These methods no longer take `applicationId` as a parameter and instead you should call `TopologyBuilder#setApplicationId(String applicationId)` before calling one of these methods.

```
TopologyBuilder builder = new TopologyBuilder();
...

// old API
Set<String> topics = topologyBuilder.sourceTopics("applicationId");
Map<Integer, TopicsInfo> topicGroups = topologyBuilder.topicGroups("applicationId");

// new API
topologyBuilder.setApplicationId("applicationId");
Set<String> topics = topologyBuilder.sourceTopics();
Map<Integer, TopicsInfo> topicGroups = topologyBuilder.topicGroups();
```

DSL: New parameters to specify state store names

Apache Kafka `0.10.1` introduces Interactive Queries, which allow you to directly query state stores of a Kafka Streams application. This new feature required a few changes to the operators in the DSL. Starting with Kafka `0.10.1`, state stores must be always be "named", which includes both explicitly used state stores (e.g., defined by the user) and internally used state stores (e.g., created behind the scenes by operations such as `count()`). This naming is a prerequisite to make state stores queryable. As a result of this, the previous "operator name" is now the state store name. This change affects `KStreamBuilder#table(...)` and *windowed* aggregates `KGroupedStream#count(...)`, `#reduce(...)`, and `#aggregate(...)`.

```
// old API
builder.table("topic");
builder.table(keySerde, valSerde, "topic");

table2 = table1.through("topic");

stream.countByKey(TimeWindows.of("windowName", 1000)); // window has a name

// new API
builder.table("topic", "storeName"); // requires to provide a store name to make KTable queryable
builder.table(keySerde, valSerde, "topic", "storeName"); // requires to provide a store name to make KTable queryable

table2 = table1.through("topic", "storeName"); // requires to provide a store name to make KTable queryable

// for changes of countByKey() -> groupByKey().count(...), please see example above
// for changes of TimeWindows.of(...), please see example below
stream.groupByKey().count(TimeWindows.of(1000), "countStoreName"); // window name removed, store name added
```

Windowing

The API for `JoinWindows` was improved. It is not longer possible to define a window with a default size (of zero). Furthermore, windows are not named anymore. Rather, any such naming is now done for state stores. See section DSL: New parameters to specify state store names above).

```
// old API
JoinWindows.of("name"); // defines window with size zero
JoinWindows.of("name").within(60 * 1000L);

TimeWindows.of("name", 60 * 1000L);
UnlimitedWindows.of("name", 60 * 1000L);

// new API, no name, requires window size
JoinWindows.of(0); // no name; set window size explicitly to zero
JoinWindows.of(60 * 1000L); // no name

TimeWindows.of(60 * 1000L); // not required to specify a name anymore
UnlimitedWindows.of(); // not required to specify a name anymore
```

© Copyright 2019, Confluent, Inc. Privacy Policy | Terms & Conditions. Apache, Apache Kafka, Kafka and the Kafka logo are trademarks of the Apache Software Foundation. All other trademarks, servicemarks, and copyrights are the property of their respective owners.

Please report any inaccuracies on this page or suggest an edit.

Rate this page

Last updated on Sep 10, 2019.