confluent

Monitoring Streams Applications

Metrics

Accessing Metrics

Accessing Metrics via JMX and Reporters

The Kafka Streams library reports a variety of metrics through JMX. It can also be configured to report stats using additional pluggable stats reporters using the `metrics.reporters` configuration option. The easiest way to view the available metrics is through tools such as JConsole, which allow you to browse JMX MBeans.

Accessing Metrics Programmatically

The entire metrics registry of a `KafkaStreams` instance can be accessed read-only through the method `KafkaStreams#metrics()`. The metrics registry will contain all the available metrics listed below. See the documentation of `KafkaStreams` in the Kafka Streams Javadocs for details.

Configuring Metrics Granularity

By default Kafka Streams has metrics with two recording levels: `debug` and `info`. The `debug` level records all metrics, while the `info` level records only some of them. Use the `metrics.recording.level` configuration option to specify which metrics you want collected, see Optional configuration parameters.

Built-in Metrics

Thread Metrics

All the following metrics have a recording level of `info`.

MBean: kafka.streams:type=stream-metrics,thread.client-id=[threadId]

> `[commit | poll | process | punctuate]-latency-[avg | max]`
> The [average | maximum] execution time in ms, for the respective operation, across all running tasks of this thread.
>
> `[commit | poll | process | punctuate]-rate`
> The average number of respective operations per second across all tasks.
>
> `task-created-rate`
> The average number of newly created tasks per second.
>
> `task-closed-rate`
> The average number of tasks closed per second.
>
> `skipped-records-total`
> The total number of skipped records. Malformed records are skipped for a number of reasons, depending on your configur
> addition to incrementing this metric, Streams logs a warning for each skip, so you should check the logs to track down the

Expand Content

for unexpected skips.

`skipped-records-rate`
The average number of skipped records per second.

Task Metrics

MBean: kafka.streams:type=stream-task-metrics,client-id=[threadId],task-id=[taskId]

`commit-latency-[avg | max]`
(`debug`) The [average | maximum] commit time in ns for this task.

`commit-rate`
(`debug`) The average number of commit calls per second.

`record-lateness-[avg | max]`
(`info`) The [average | maximum] observed lateness (stream time - record timestamp).

Processor Node Metrics

All the following metrics have a recording level of `debug`.

MBean: kafka.streams:type=stream-processor-node-metrics,client-id=[threadId],task-id=[taskId],processor-node-id=[processorNodeId]

`[process | punctuate | create | destroy]-latency-[avg | max]`
The [average | maximum] execution time in ns, for the respective operation.

`[process | punctuate | create | destroy]-rate`
The average number of respective operations per second.

`forward-rate`
The average rate of records being forwarded downstream, from source nodes only, per second. This metric can be used to understand how fast the library is consuming from source topics.

State Store Metrics

All the following metrics have a recording level of `debug`.

MBean: kafka.streams:type=stream-[storeType]-state-metrics,client-id=[threadId],task-id=[taskId],[storeType]-state-id=[storeName]

`[put | put-if-absent | get | delete | put-all | all | range | flush | restore]-latency-[avg | max]`
The average execution time in ns, for the respective operation.

`[put | put-if-absent | get | delete | put-all | all | range | flush | restore]-rate`
The average rate of respective operations per second for this store.

Record Cache Metrics

All the following metrics have a recording level of `debug`.

MBean: kafka.streams:type=stream-record-cache-metrics,client-id=[threadId],task-id=[taskId],record-cache-id=[storeName]

> **`hitRatio-[avg | min | max]`**
> The [average | minimum | maximum] cache hit ratio defined as the ratio of cache read hits over the total cache read requests.

Suppression Buffer Metrics

All the following metrics have a recording level of `debug`. The `bufferName` can be set via `Suppressed.withName(bufferName)`, otherwise it will be generated.

MBean: kafka.streams:type=stream-buffer-metrics,client-id=[threadId],task-id=[taskId],buffer-id=[bufferName]

> **`suppression-buffer-size-[current | avg | max]`**
> The [current | average | maximum] size of buffered data. This helps you choose a value for `BufferConfig.maxBytes(...)`, if desired.
>
> **`suppression-buffer-count-[current | avg | max]`**
> The [current | average | maximum] number of records in the buffer. This helps you choose a value for `BufferConfig.maxRecords(...)`, if desired.

Adding Your Own Metrics

Application developers using the low-level Processor API can add additional metrics to their application. The `ProcessorContext#metrics()` method provides a handle to the `StreamMetrics` object, which you can use to:

- Add latency and throughput metrics via `StreamMetrics#addLatencyAndThroughputSensor` and `StreamMetrics#addThroughputSensor()`.
- Add any other type of metric via `StreamMetrics#addSensor()`.

Runtime Status Information

Status of `KafkaStreams` instances

> **Important**
>
> Don't confuse the runtime state of a `KafkaStreams` instance (e.g. created, rebalancing) with state stores!

A Kafka Streams instance may be in one of several run-time states, as defined in the enum `KafkaStreams.State`. For example, it might be created but not running; or it might be rebalancing and thus its state stores are not available for querying. Users can access the current runtime state programmatically using the method `KafkaStreams#state()`. The documentation of `KafkaStreams.State` in the Kafka Streams Javadocs lists all the available states.

Also, you can use `KafkaStreams#setStateListener()` to register a `KafkaStreams#StateListener` method that will be triggered whenever the state changes.

Use the `KafkaStreams#localThreadsMetadata()` method to check the runtime state of the current `KafkaStreams` instance. The `localThreadsMetadata()` method returns a `ThreadMetadata` object for each local stream thread. The `ThreadMetadata` object describes the runtime state of a thread and the metadata for the thread's currently assigned tasks.

Get Runtime Information on KafkaStreams Clients

You can get runtime information on these local `KafkaStreams` clients:

- Admin client
- Producer clients
- Consumer client
- Restore consumer client

There is one admin client per `KafkaStreams` instance, and all other clients are per `StreamThread`.

Get the names of local `KafkaStreams` clients by calling the client ID methods on the `ThreadMetadata` class, like `producerClientIds()`.

Client names are based on a client ID value, which is assigned according to the `StreamsConfig.CLIENT_ID_CONFIG` and `StreamsConfig.APPLICATION_ID_CONFIG` configuration settings.

- If `CLIENT_ID_CONFIG` is set, Kafka Streams uses `CLIENT_ID_CONFIG` for the client ID value.
- If `CLIENT_ID_CONFIG` isn't set, Kafka Streams uses `APPLICATION_ID_CONFIG` and appends a random unique identifier (UUID):

  ```
  clientId = StreamsConfig.APPLICATION_ID_CONFIG + "-" + <random-UUID>
  ```

Kafka Streams creates names for specific clients by appending a thread ID and a descriptive string to the main client ID.

```
specificClientId = clientId + "-StreamThread-" + <thread-number> + <description>
```

For example, if `CLIENT_ID_CONFIG` is set to "MyClientId", the `consumerClientId()` method returns a value that resembles `MyClientId-StreamThread-2-consumer`. If `CLIENT_ID_CONFIG` isn't set, and `APPLICATION_ID_CONFIG` is set to "MyApplicationId", the `consumerClientId()` method returns a value that resembles `MyApplicationId-8d8ce4a7-85bb-41f7-ac9c-fe6f3cc0959e-StreamThread-2-consumer`.

Call the `threadName()` method to get the thread ID:

```
threadId = clientId + "-StreamThread-" + <thread-number>
```

Depending on the configuration settings, an example thread ID resembles `MyClientId-StreamThread-2` or `MyApplicationId-8d8ce4a7-85bb-41f7-ac9c-fe6f3cc0959e-StreamThread-2`.

adminClientId()

Gets the ID of the client application, which is the main client ID value, appended with `-admin`. Depending on configuration settings, the return value resembles `MyClientId-admin` or `MyApplicationId-8d8ce4a7-85bb-41f7-ac9c-fe6f3cc0959e-admin`.

producerClientIds()

Gets the names of producer clients. If exactly-once semantics (EOS) is active, returns the list of task producer names, otherwise returns the thread producer name. All producer client names are the main thread ID appended with `-producer`. If EOS is active, a `-<taskId>` is included.

A task ID is a sub-topology ID and a partition number, `<subTopologyId>_<partition>`. The `subTopologyId` is an integer greater than or equal to zero.

If EOS is active, the `producerClientIds()` method returns a `Set` of client names that have different task IDs. Depending on configuration settings, the return value resembles `MyClientId-StreamThread-2-1_4-producer`.

If EOS isn't active, the return value is a single client name that doesn't have a task ID, for example `MyClientId-StreamThread-2-producer`.

For more information, see Stream Partitions and Tasks.

consumerClientId()

Gets the name of the consumer client. The consumer client name is the main thread ID appended with `-consumer`, for example, `MyClientId-StreamThread-2-consumer`.

restoreConsumerClientId()

Gets the name of the restore consumer client. The restore consumer client name is the main thread ID appended with `-restore-consumer`, for example, `MyClientId-StreamThread-2-restore-consumer`

Monitoring the Restoration Progress of Fault-tolerant State Stores

When starting up your application any fault-tolerant state stores don't need a restoration process as the persisted state is read from local disk. But there could be situations when a full restore from the backing changelog topic is required (e.g., a failure wiped out the local state or your application runs in a stateless environment and persisted data is lost on re-starts).

If you have a significant amount of data in the changelog topic, the restoration process could take a non-negligible amount of time. Given that processing of new data won't start until the restoration process is completed, having a window into the progress of restoration is useful.

In order to observe the restoration of all state stores you provide your application an instance of the `org.apache.kafka.streams.processor.StateRestoreListener` interface. You set the `org.apache.kafka.streams.processor.StateRestoreListener` by calling the `KafkaStreams#setGlobalStateRestoreListener` method.

A basic implementation example that prints restoration status to the console:

```
import org.apache.kafka.common.TopicPartition;
import org.apache.kafka.streams.processor.StateRestoreListener;

 public class ConsoleGlobalRestoreListerner implements StateRestoreListener {

    @Override
    public void onRestoreStart(final TopicPartition topicPartition,
                               final String storeName,
                               final long startingOffset,
                               final long endingOffset) {

        System.out.print("Started restoration of " + storeName + " partition " + topicPartition.partition());
        System.out.println(" total records to be restored " + (endingOffset - startingOffset));
    }

    @Override
    public void onBatchRestored(final TopicPartition topicPartition,
                                final String storeName,
                                final long batchEndOffset,
                                final long numRestored) {

        System.out.println("Restored batch " + numRestored + " for " + storeName + " partition " + topicPartition.part
ition());

    }

    @Override
    public void onRestoreEnd(final TopicPartition topicPartition,
                             final String storeName,
                             final long totalRestored) {

        System.out.println("Restoration complete for " + storeName + " partition " + topicPartition.partition());
    }
}
```

Attention

The `StateRestoreListener` instance is shared across all `org.apache.kafka.streams.processor.internals.StreamThread` instances and also used for global stores. Furthermore, it is assumed all methods are stateless. If any stateful operations are desired, then the user will need to provide synchronization internally

Integration with Confluent Control Center

Since the 3.2 release, Confluent Control Center will display the underlying producer metrics and consumer metrics of a Kafka Streams application, which the Streams API uses internally whenever data needs to be read from or written to Apache Kafka® topics. These metrics can be used, for example, to monitor the so-called "consumer lag" of an application, which indicates whether an application -- at its current capacity and available computing resources -- is able to keep up with the incoming data volume.

A Kafka Streams application, i.e. all its running instances, appear as a single consumer group in Control Center.

Note

Restore consumers of an application are displayed separately: Behind the scenes, the Streams API uses a dedicated "restore" consumer for the purposes of fault tolerance and state management. This restore consumer manually assigns and manages the topic partitions it consumes from and is not a member of the application's consumer group. As a result, the restore consumers will be displayed separately from their application.

Last updated on Sep 10, 2019.